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:

a) Start by initializing a queue and a set to keep track of visited vertices.
b) En-queue the source vertex into the queue and mark it as visited.
c) Repeat the following steps until the queue becomes empty:
d) De-queue a vertex from the front of the queue.
e) Process the vertex (print it, store it, or perform any other desired operation).
f) En-queue all the unvisited neighbors of the vertex into the queue and mark them as visited.
g) 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:

```
PS D:\TYCS Files\AI> & "C:/Users/COMPUTER LAB/AppData/Local/Programs/Python/Python36/python.exe" "d:/TYCS Files/AI/prac_1_1.py"
A
B
C
D
E
F
PS D:\TYCS Files\AI>
```

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:

```
PS D:\TYCS Files\AI> & "C:/Users/COMPUTER LAB/AppData/Local/Programs/Python/Python36/python.exe" "d:/TYCS Files/AI/prac_1_2.py"
Shortest Path Start form 1 to 9 : [1, 2, 4, 9]
PS D:\TYCS Files\AI>
```

Practical No: 2

Aim:

4) **Implement the A* Search algorithm for solving a path finding problem.**
5) **Implement the Recursive Best-First Search algorithm for the same problem.**
6) **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:

a) Initialize the open list, closed list, and set the g-value of the start node to 0.
b) 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.
c) Enqueue the start node to the open list with its f-value as the priority.
d) Repeat the following steps until the open list becomes empty or the goal node is reached:
    I. Dequeue the node with the lowest f-value from the open list. This node becomes the current node.
    II. If the current node is the goal node, the algorithm terminates, and the path has been found.
    III. Add the current node to the closed list to mark it as visited.
    IV. Explore the neighboring nodes of the current node:
        i. 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.
        ii. If the neighbor is not in the closed list or its tentative g-value is lower than its current g-value:
            1) Update the g-value of the neighbor to the new lower value.
            2) Calculate the f-value of the neighbor by adding its g-value and h-value.
            3) If the neighbor is not in the open list, enqueue it with its f-value as the priority.
            4) If the neighbor is already in the open list, update its priority if the new f-value is lower.
            5) Set the parent of the neighbor to the current node.
e) If the open list becomes empty before reaching the goal node, there is no path available.

f) 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},
```

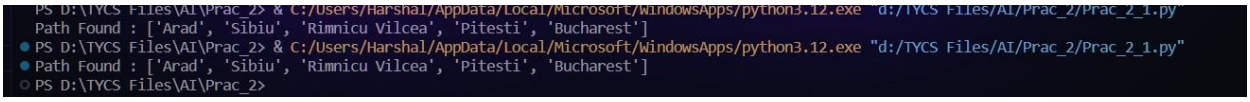
```
    'Zerind': {'Arad': 75, 'Oradea': 71},
    'Timisoara': {'Arad': 118, 'Lugoj': 111},
    'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80},
    'Oradea': {'Zerind': 71, 'Sibiu': 151},
    'Lugoj': {'Timisoara': 111, 'Mehadia': 70},
    'Fagaras': {'Sibiu': 99, 'Bucharest': 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:
```

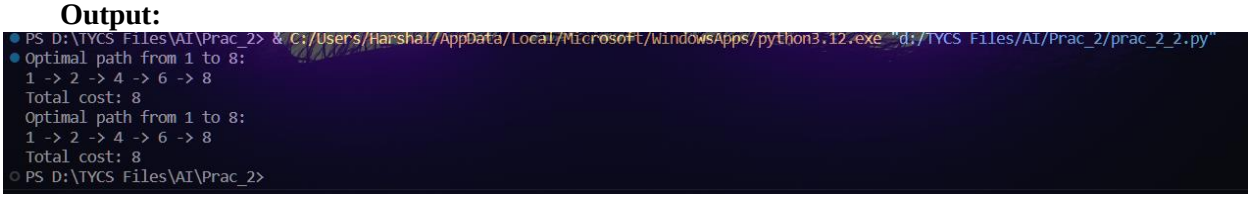
```
            return None, best.f

        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

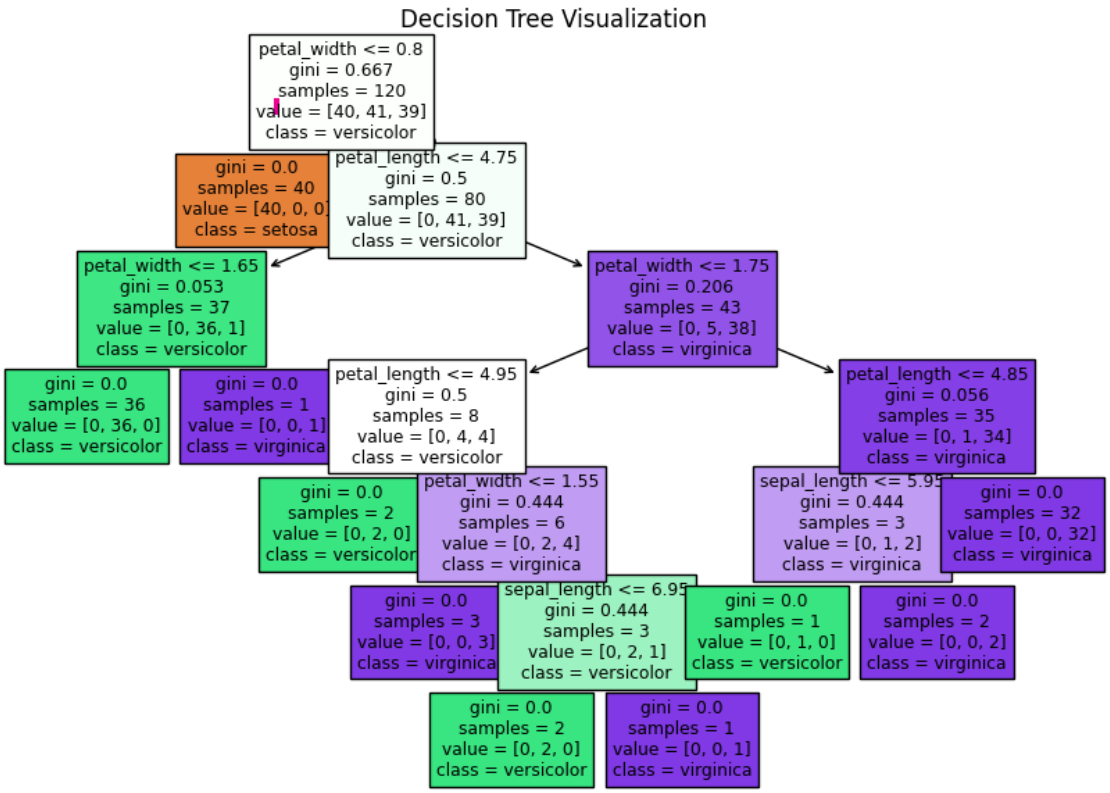
- **Implement the Decision Tree Learning algorithm to build a decision tree for a given dataset.**
- **Evaluate the accuracy and effectiveness of the decision tree on test data.**
- **Visualize and interpret the generated decision tree.**

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:

```
PS D:\TYCS Files\AI> & "C:/Users/COMPUTER LAB/AppData/Local/Programs/Python/Python312/python.exe"
Files/AI/prac_4/prac_4.py"
Input: [0 0], Predicted Output: [0.08970809]
Input: [0 1], Predicted Output: [0.91676068]
Input: [1 0], Predicted Output: [0.9190354]
Input: [1 1], Predicted Output: [0.06455968]
PS D:\TYCS Files\AI>
```

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:

```
Accuracy: 1.00
                precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        10
Iris-versicolor       1.00      1.00      1.00         9
 Iris-virginica       1.00      1.00      1.00        11

       accuracy                           1.00        30
      macro avg       1.00      1.00      1.00        30
   weighted avg       1.00      1.00      1.00        30
```

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:

```
PS D:\TYCS Files\AI\prac_6_Complete> & "C:/Users/COMPUTER LAB/AppData/Local/Programs/Python/Python312/
n.exe" "d:/TYCS Files/AI/prac_6_Complete/AI_prac_6.py"
Weak Classifier Accuracy: 0.7111111111111111
AdaBoost Classifier Accuracy: 1.0
```

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:

```
Accuracy: 1.0
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      1.00      1.00         9
           2       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

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:

```
PS D:\TYCS Files\AI> & "C:/Users/COMPUTER LAB/AppData/Local/Programs/Python/Python312/python.exe"
Files/AI/prac_8/prac_8.py"
Accuracy: 1.0
```

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:

```
PS D:\TYCS Files\AI> & "C:/Users/COMPUTER LAB/AppData/Local/Programs/Python/Python312/python.exe" "d:/T
             antecedents    consequents  antecedent support  ...  leverage  conviction  zhangs_metric
0                 (beer)        (bread)                 0.8  ...     -0.04         0.8      -0.250000
1                (bread)         (beer)                 0.8  ...     -0.04         0.8      -0.250000
2              (diaper)         (beer)                 0.4  ...      0.08         inf       0.333333
3                (eggs)         (beer)                 0.6  ...     -0.08         0.6      -0.333333
4                (beer)         (milk)                 0.8  ...     -0.04         0.8      -0.250000
5                (milk)         (beer)                 0.8  ...     -0.04         0.8      -0.250000
6                (eggs)        (bread)                 0.6  ...     -0.08         0.6      -0.333333
7              (bread)         (milk)                 0.8  ...     -0.04         0.8      -0.250000
8                (milk)        (bread)                 0.8  ...     -0.04         0.8      -0.250000
9                (milk)         (eggs)                 0.8  ...      0.12         1.6       1.000000
10               (eggs)         (milk)                 0.6  ...      0.12         inf       0.500000
11      (bread, diaper)         (beer)                 0.2  ...      0.04         inf       0.250000
12        (beer, bread)         (milk)                 0.6  ...     -0.08         0.6      -0.333333
13         (beer, milk)        (bread)                 0.6  ...     -0.08         0.6      -0.333333
14        (bread, milk)         (beer)                 0.6  ...     -0.08         0.6      -0.333333
15        (eggs, diaper)         (beer)                 0.2  ...      0.04         inf       0.250000
16        (milk, diaper)         (beer)                 0.2  ...      0.04         inf       0.250000
17         (beer, milk)         (eggs)                 0.6  ...      0.04         1.2       0.250000
18         (eggs, milk)         (beer)                 0.6  ...     -0.08         0.6      -0.333333
19         (beer, eggs)         (milk)                 0.4  ...      0.08         inf       0.333333
20               (eggs)   (beer, milk)                 0.6  ...      0.04         1.2       0.250000
21        (bread, milk)         (eggs)                 0.6  ...      0.04         1.2       0.250000
22         (eggs, milk)        (bread)                 0.6  ...     -0.08         0.6      -0.333333
23        (bread, eggs)         (milk)                 0.4  ...      0.08         inf       0.333333
24               (eggs)   (bread, milk)                 0.6  ...      0.04         1.2       0.250000
25        (milk, diaper)         (eggs)                 0.2  ...      0.08         inf       0.500000
26        (eggs, diaper)         (milk)                 0.2  ...      0.04         inf       0.250000
27   (beer, bread, eggs)         (milk)                 0.2  ...      0.04         inf       0.250000
28  (beer, milk, diaper)         (eggs)                 0.2  ...      0.08         inf       0.500000
29  (eggs, milk, diaper)         (beer)                 0.2  ...      0.04         inf       0.250000
30  (beer, eggs, diaper)         (milk)                 0.2  ...      0.04         inf       0.250000
31        (milk, diaper)   (beer, eggs)                 0.2  ...      0.12         inf       0.750000
32        (eggs, diaper)   (beer, milk)                 0.2  ...      0.08         inf       0.500000

[33 rows x 10 columns]
```