Index

1.Breadth First Search & Iterative Depth First Search

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.

Theory:

Part 1: Breadth First Search algorithm:

Breadth First Search (BFS) is an algorithm used to traverse or search a graph or tree data structure. It explores all the vertices of a graph in breadth-first order, starting from a specified vertex known as the "source" vertex. BFS is often used to find the shortest path between two vertices in an unweighted graph or to visit all the vertices in a connected component of a graph.

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.

Python Code for Breadth First Search algorithm

```
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 neighbors
            neighbors = graph[vertex]
            for neighbor in neighbors:
                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)
```

Part 2: Iterative Depth First Search algorithm:

Iterative Depth First Search (DFS) is an algorithm used to traverse or search a graph or tree data structure. It explores all the vertices of a graph in depth-first order, starting from a specified vertex known as the "source" vertex. The iterative approach of DFS avoids potential stack overflow errors that can occur with large graphs in recursive DFS implementations.
The iterative DFS algorithm works as follows:

   a) Start by initializing a stack and a set to keep track of visited vertices.
   b) Push the source vertex onto the stack and mark it as visited.
   c) Repeat the following steps until the stack becomes empty:
   d) Pop a vertex from the top of the stack.
   e) Process the vertex (print it, store it, or perform any other desired operation).
   f) Retrieve all the unvisited neighbors of the vertex.
   g) For each unvisited neighbor, push it onto the stack and mark it as visited.
   h) The algorithm terminates when the stack becomes empty, indicating that all reachable vertices have been processed.

Python code for Iterative Depth First Search algorithm

```python
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 from {start_node} to {end_node}: {shortest_path}")
    else:
        print(f"No path found from {start_node} to {end_node}")
```

Part 3: Comparing the performance and efficiency of the two algorithms

In the comparison of efficiency and performance between Breadth-First Search (BFS) and Iterative Deepening Depth-First Search (IDDFS) algorithms, let's analyze how they perform based on the given example:

Efficiency:

Time Complexity:

BFS: The time complexity of BFS is influenced by the branching factor of the graph or tree. In the worst case scenario, BFS may need to explore all nodes up to a certain depth. Hence, the time complexity of BFS is typically high, especially in graphs with high branching factors.
IDDFS: IDDFS combines the advantages of both BFS and DFS. It performs a series of depth-limited DFS searches, gradually increasing the depth limit until the goal node is found. The time complexity of IDDFS is better than BFS in terms of average-case scenarios, as it avoids exploring unnecessary nodes. However, in the worst case, where the goal node is at the maximum depth, IDDFS may explore the entire search space.

Space Complexity:

BFS: BFS requires additional memory to store the open queue, closed set, and node information. The space complexity of BFS is influenced by the number of nodes at each level of the graph. In the worst case, BFS may require a lot of memory to store all the nodes at a particular depth.

IDDFS: IDDFS has a better space complexity compared to BFS since it performs a depth-limited DFS search. It only keeps track of the current path being explored, resulting in lower memory consumption. The space complexity of IDDFS is dependent on the maximum depth of the search. Performance:

Search Behavior:
BFS: BFS systematically explores all nodes at a given level before moving to the next level. This property ensures that the optimal path is found as long as the edges have uniform costs. However, BFS may expand a large number of nodes, making it less efficient in terms of time and memory consumption, especially in graphs with a high branching factor.

IDDFS: IDDFS combines the advantages of both BFS and DFS. It performs depth-limited DFS searches, incrementing the depth limit with each iteration. This iterative approach allows IDDFS to avoid exploring unnecessary nodes at greater depths. IDDFS is complete and guarantees finding the optimal solution when the branching factor is finite.

Based on the above analysis, BFS and IDDFS have different strengths and weaknesses:

BFS is generally more suitable for scenarios where memory is not a constraint, and finding the optimal path is a priority. However, it may be inefficient in terms of time and space, especially in large and highly branching graphs.

IDDFS is memory-efficient due to its depth-limited search approach. It performs well in scenarios with limited memory and can find the optimal path when the branching factor is finite. However, in graphs with high branching factors or when the goal node is deep, IDDFS may still explore a significant number of nodes.

It's important to consider the specific characteristics of the problem and the trade-offs between time complexity and space complexity when choosing between BFS and IDDFS. If memory is limited, IDDFS may be a more practical choice, while BFS may be preferable when finding the optimal path is crucial and memory is not a constraint.

2.A* Search and Recursive Best-First Search

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.

Theory:
Part 1: A* Search algorithm:
A* (A-star) is a widely used search algorithm that combines the best features of both Dijkstra's algorithm and heuristic search. It is commonly applied to solve the path-finding problem in graphs or grids, where the goal is to find the shortest path from a start node to a goal node.

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.

The A* algorithm is both complete (able to find a solution if one exists) and optimal (guaranteed to find the shortest path) under certain conditions. The heuristic used must be admissible, meaning it never overestimates the actual cost to reach the goal node. Additionally, the heuristic must be consistent (or monotonic), meaning the estimated cost from a node to its successor plus the heuristic value of the successor is always less than or equal to the estimated cost from the current node to the goal node.

A* is widely used in various applications such as path planning, robotics, navigation systems, and game AI due to its efficiency and optimality. It efficiently explores the search space by prioritizing nodes that are most likely to lead to the goal while considering the actual cost from the start node.

We consider the Map of Romania problem and solve it using A* Search
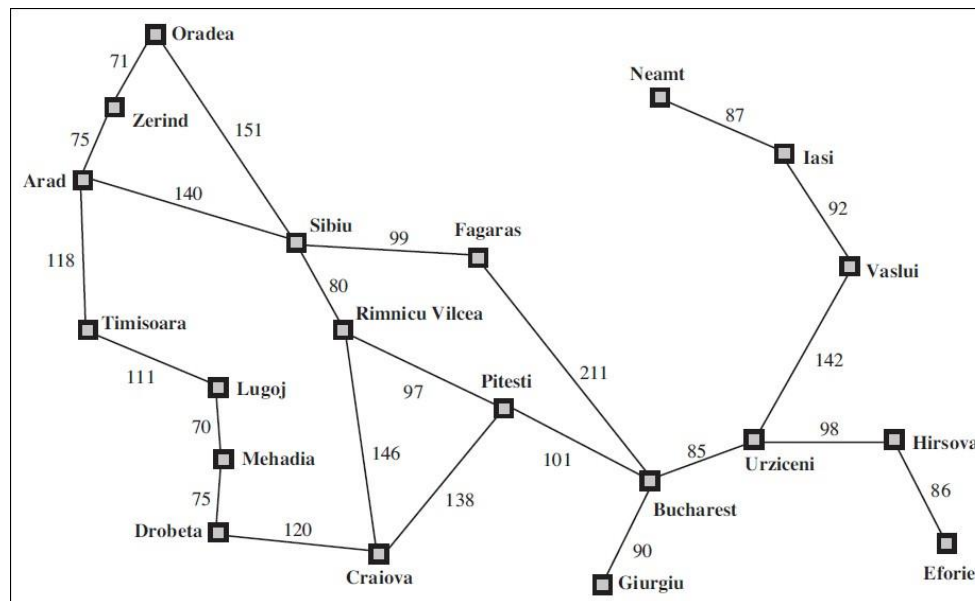
Python code:

```python
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")
```

Output: ['Arad', 'Sibiu', 'Rimnicu Vilcea', 'Pitesti', 'Bucharest']

Part 2: Recursive Best-First Search algorithm:

Recursive Best-First Search (RBFS) is an informed search algorithm used for finding the shortest path from a start node to a goal node in a graph or tree. RBFS is a memory-bounded variant of the A* algorithm that avoids the need for storing the entire search tree explicitly.

RBFS utilizes a heuristic function that estimates the cost from each node to the goal node. It maintains a priority queue of nodes to be explored, sorted based on their f-values, which is the sum of the cost to reach the node from the start node (g-value) and the estimated cost from the node to the goal node (h-value).

The RBFS algorithm follows these steps:

a) Initialize the recursive function RBFS with the current node, goal node, and a limit value (initially set to infinity).
b) Check if the current node is the goal node. If it is, return the path containing only the current node, indicating success.
c) Generate the successors of the current node and calculate their f-values using a heuristic function.
d) If there are no successors, return None to indicate failure.
e) Loop over the successors in order of their f-values:
   i. Recursively call RBFS on the successor with the minimum f-value, the goal node, and the minimum of the current limit and the f-value of the next best successor. This recursive call effectively explores the subtree rooted at the selected successor.
   ii. If the recursive call returns a path, return the path concatenated with the current node, indicating success.
   iii. If the recursive call returns None, update the limit to the maximum f-value among the successors. This limit represents the threshold beyond which RBFS will not explore further.
   iv. If all successors have been explored and none of them resulted in a path, return None to indicate failure.
   v. The RBFS algorithm continues to recursively explore the graph/tree, using the minimum limit from the failed recursive calls as the new limit for subsequent iterations. This allows RBFS to "backtrack" and reconsider previously rejected nodes if a better path is discovered.

RBFS terminates when a path is found or when it exhaustively explores all nodes without finding a path. The algorithm is memory-efficient since it only keeps track of the current path and the best f-value encountered so far.

RBFS is especially useful in scenarios where memory is limited, and it prioritizes exploring promising nodes based on their f-values, leading to efficient search in large state spaces.

Python code for Recursive Best-First Search algorithm

```python
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.")
```

Part 3: Comparing the performance and efficiency of the two algorithms

In the comparison of efficiency and performance between the A* and RBFS algorithms, we can consider factors such as time complexity, space complexity, and search behavior as follows
Efficiency:

Time Complexity:

A*: The time complexity of A* depends on the heuristic function used. In the worst case, where the heuristic is not admissible, A* may explore more nodes than necessary. However, with an admissible heuristic, the time complexity is typically much better than blind search algorithms like Breadth-First Search or Depth-First Search.
RBFS: The time complexity of RBFS is generally exponential, as it explores nodes recursively. It explores the search space by backtracking and re-examining previously rejected nodes, which can lead to redundant exploration.
Space Complexity:

A*: A* requires additional memory to store the open list, closed list, and node information. The space complexity of A* depends on the branching factor of the graph and the size of the search space.
RBFS: RBFS has a better space complexity compared to A* since it doesn't require storing the entire search tree explicitly. RBFS only keeps track of the current path and the best f-value encountered so far, resulting in lower memory consumption.

Performance:

Search Behavior:
A*: A* is guaranteed to find the optimal path when using an admissible heuristic. It efficiently explores the search space by prioritizing nodes based on their f-values. However, if the heuristic is not admissible or inconsistent, A* may expand more nodes than necessary.
RBFS: RBFS is not guaranteed to find the optimal path. It explores the search space recursively, considering the best successor node based on f-values. However, due to its backtracking nature, RBFS may revisit nodes and spend more time exploring redundant paths.
Based on the above analysis, A* generally outperforms RBFS in terms of efficiency and performance. A* with an admissible and consistent heuristic tends to explore fewer nodes compared to RBFS. It provides an optimal path and can be more time-efficient, especially in larger search spaces. However, it requires more memory to store the open and closed lists.

On the other hand, RBFS is memory-efficient and can handle larger search spaces with limited memory. However, it may spend more time exploring redundant paths and is not guaranteed to find the optimal solution.

It's important to note that the performance of both algorithms heavily relies on the specific problem and the quality of the heuristic function used. It's advisable to carefully choose or design an appropriate heuristic for the problem domain to maximize the efficiency and performance of these algorithms.

3.Decision Tree Learning

Aim

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

Theory:

A Decision Tree is a supervised machine learning algorithm used for both classification and regression tasks. It's a tree-like model that makes decisions based on a series of conditions or rules learned from the training data. Each internal node in the tree represents a decision rule based on a feature, and each leaf node represents a class label (in classification) or a predicted value (in regression). Decision trees are characterized by their simplicity, interpretability, and ability to handle both categorical and numerical data.

Decision trees work in the following way
1. Root Node: At the root of the tree, the algorithm selects the feature that best separates the data based on a criterion (e.g., Gini impurity or entropy for classification, mean squared error for regression). This feature becomes the root node.

2. Internal Nodes: The algorithm recursively selects features to split the data into subsets at each internal node. These splits are determined to minimize impurity (for classification) or reduce error (for regression).

3. Leaf Nodes: The process continues until a stopping criterion is met, such as a maximum depth of the tree or a minimum number of data points in a node. The final nodes are called leaf nodes and represent the predicted class or value.

Decision trees have several advantages that make them a popular choice for various machine learning and data analysis tasks. Some of the key advantages of decision trees include:

1. Interpretability: Decision trees provide a clear and intuitive representation of the decision-making process. It's easy to understand and explain the logic of a decision tree to non-technical stakeholders. This makes decision trees valuable in domains where model interpretability is essential, such as healthcare and finance.

2. Handling Mixed Data Types: Decision trees can handle both categorical and numerical data without the need for extensive data pre-processing. Other algorithms may require encoding categorical variables or scaling numerical features.

3. No Assumptions About Data Distribution: Decision trees do not assume that the data follows a particular statistical distribution, making them versatile for various types of data.

4. Non-Linearity: Decision trees can capture non-linear relationships between features and the target variable. They can model complex decision boundaries, which is especially useful when the relationship between variables is not linear.

5. Feature Importance: Decision trees can naturally identify feature importance by evaluating which features are used for splitting nodes higher in the tree. This information can guide feature selection and dimensionality reduction efforts.

6. Robustness to Outliers: Decision trees are relatively robust to outliers in the data. Outliers may affect individual branches of the tree but tend to have a limited impact on the overall structure.

7. Scalability: Decision trees are computationally efficient, and the training time complexity is generally linear in the number of training examples and features. This makes them suitable for both small and large datasets.

8. Ensemble Methods: Decision trees can be combined into ensemble methods like Random Forest and Gradient Boosting, which can significantly improve predictive performance by reducing overfitting and increasing robustness.

9. Handling Missing Data: Decision trees can handle missing data by making decisions based on the available features, reducing the need for imputation or data removal.

10. Wide Range of Applications: Decision trees can be applied to various tasks, including classification, regression, anomaly detection, and recommendation systems.

11. Balance Between Bias and Variance: Decision trees can be controlled and pruned to find an appropriate trade-off between bias and variance, which helps mitigate overfitting.

Despite these advantages, it's important to note that decision trees also have limitations, such as their susceptibility to overfitting, instability with small changes in the data, and potential bias towards features with many categories. To address some of these limitations, ensemble methods like Random Forest and Gradient Boosting are often used, combining multiple decision trees to improve model performance.

Data Set:

We download the Iris data set and use it for the present case. As we are performing the practical on Google Colab, we upload the dataset as Iris.csv

Python Code:

```python
# 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 'your_dataset.csv' with the actual path to your dataset file
data = pd.read_csv('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 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()
```

```python
# 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:

Decision Tree Visualization
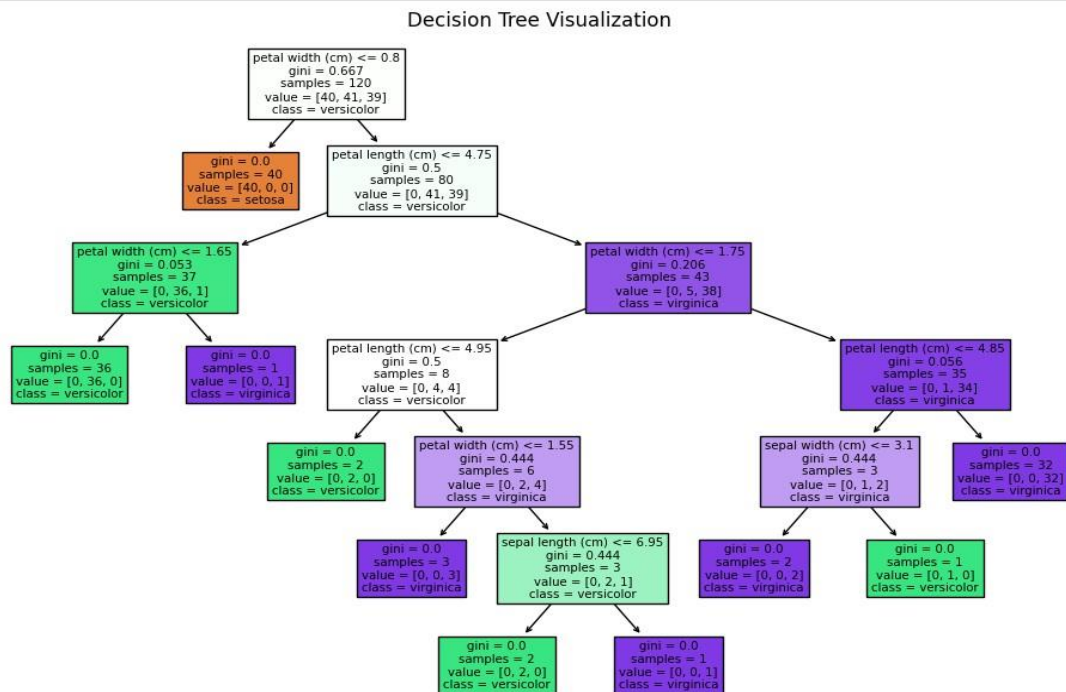
4.Feedforward Backpropagation Neural Network

Aim:

1. Implement the Feed Forward Backpropagation algorithm to train a neural network.
2. Use a given dataset to train the neural network for a specific task.
3. Evaluate the performance of the trained network on test data

Theory: Feedforward neural networks, also known as feedforward neural networks (FNNs) or multilayer perceptrons (MLPs), are a fundamental type of artificial neural network used in machine learning and deep learning. They are designed to model complex relationships between inputs and outputs by stacking multiple layers of interconnected neurons (also called nodes or units).

Here are the key characteristics of feedforward neural networks:

1. Feedforward Structure: FNNs have a strict feedforward structure, meaning information flows in one direction, from the input layer through one or more hidden layers to the output layer. There are no feedback loops or recurrent connections in this architecture.

2. Layers: An FNN typically consists of three main types of layers:
   - Input Layer: This layer contains neurons that represent the features or input data. Each neuron corresponds to a specific input feature.
   - Hidden Layers: These intermediate layers, which can be one or more, perform complex transformations on the input data. Each neuron in a hidden layer is connected to all neurons in the previous layer and feeds its output to the next layer.
   - Output Layer: The final layer produces the network's predictions or outputs. The number of neurons in the output layer depends on the problem; for regression tasks, it may be one neuron, while for classification tasks, it can be one neuron per class.

3. Activation Functions: Non-linear activation functions (e.g., ReLU, sigmoid, or tanh) are applied to the output of each neuron in the hidden layers. These functions introduce non-linearity into the network, enabling it to capture complex patterns in the data.

4. Weights and Biases: Every connection between neurons has an associated weight that determines the strength of the connection. Additionally, each neuron has a bias term that helps shift the activation function. These weights and biases are learned during training to optimize the network's performance.

5. Training: FNNs are trained using supervised learning methods, such as gradient descent and backpropagation. The network is presented with labelled training data, and it adjusts its weights and biases to minimize the difference between its predictions and the actual target values.

6. Loss Function: A loss function (also called a cost or objective function) quantifies the error between the network's predictions and the actual target values. The goal during training is to minimize this loss function.

7. Applications: Feedforward neural networks are used in a wide range of applications,

including image and speech recognition, natural language processing, financial modeling, and many other machine learning tasks.

8. Deep Learning: When FNNs have multiple hidden layers, they are referred to as deep neural networks (DNNs). Deep learning leverages these deep architectures to model intricate relationships in data, making them suitable for handling complex and high-dimensional problems.

In summary, feedforward neural networks are a foundational component of deep learning, allowing machines to learn and make predictions from data by forming increasingly abstract and hierarchical representations through multiple layers of interconnected neurons.

For the present case we implement the XOR- operation using Feedforward Backpropagation Neural Network. The XOR-operation for a 2-input variable is as follows

Inputs		Target
X1	**X0**	**T**
0	0	0
0	1	1
1	0	1
1	1	0

The following is the Python code to implement Feedforward Backpropagation Neural Network

Python Code	

```python
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): # 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
```

```python
            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	
	Input: [0 0], Predicted Output: [0.16943356] Input: [0 1], Predicted Output: [0.8562007] Input: [1 0], Predicted Output: [0.85655176] Input: [1 1], Predicted Output: [0.12905042]

We conclude that the Predicted Output is very close to the Target

5.Support Vector Machines

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..

Theory:

Support Vector Machines (SVM) are a powerful and versatile class of supervised machine learning algorithms used for classification and regression tasks. Here's some essential theory about SVMs:

1. Linear Separability: SVMs are primarily designed for binary classification problems. They work by finding the optimal hyperplane that best separates two classes in the feature space. This hyperplane is chosen to maximize the margin between the two classes. When the classes are linearly separable, SVMs can find the hyperplane with the maximum margin.

2. Margin: The margin is the distance between the hyperplane and the nearest data point from either class. SVM aims to maximize this margin because a larger margin generally indicates a better separation and better generalization to unseen data.

3. Support Vectors: Support vectors are the data points that are closest to the hyperplane and directly influence its position and orientation. These are the critical data points that determine the margin. The SVM algorithm focuses on these support vectors during training.

4. Kernel Trick: SVMs can be extended to handle non-linearly separable data by using a kernel function. A kernel function transforms the original feature space into a higher-dimensional space, where the data may become linearly separable. Common kernel functions include the linear, polynomial, radial basis function (RBF/Gaussian), and sigmoid kernels.

5. C Parameter: The C parameter is a regularization parameter in SVM that balances the trade-off between maximizing the margin and minimizing classification errors. A smaller C value results in a larger margin but may allow some training points to be misclassified, while a larger C value tries to classify all training points correctly but may result in a smaller margin.

6. Soft Margin SVM: In cases where the data is not linearly separable, a soft margin SVM allows for some misclassification of training points by introducing a slack variable. This approach makes the SVM more robust to noisy data and outliers.

7. Multi-Class Classification: SVMs can be extended to handle multi-class classification problems through techniques like one-vs-one (OvO) or one-vs-all (OvA) classification, where multiple binary classifiers are trained to distinguish between different pairs of classes.

8. SVM Regression: SVMs can also be used for regression tasks, where the goal is to predict a continuous target variable. In SVM regression, the algorithm aims to fit a hyperplane that maximizes the margin while allowing some deviations from the target values.

9. Advantages: SVMs are known for their ability to handle high-dimensional data effectively, robustness to overfitting (especially with a well-chosen kernel and regularization parameter), and versatility in handling both linear and non-linear classification problems.

10. Challenges: SVMs can be computationally expensive for large datasets, and selecting the appropriate kernel and tuning hyperparameters like C can be challenging. Interpreting the SVM model can also be less intuitive compared to some other machine learning algorithms.

SVMs are a valuable tool in the machine learning toolbox, known for their flexibility and ability to handle a wide range of classification and regression tasks, from image classification to financial modeling. Understanding the theory behind SVMs is crucial for effectively applying them in practice.

Data Set:

We download the Iris data set and use it for the present case. As we are performing the practical on Google Colab, we upload the dataset as iris.csv

Python Code:

```python
# Import necessary libraries import numpy as
np
import pandas as pd
import matplotlib.pyplot as plt from
sklearn.svm import SVC
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 'your_dataset.csv' with the actual path to your dataset file
data = pd.read_csv('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 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 an SVM classifier
svm_classifier = SVC(kernel='linear')  # You can choose different kernels here

# Fit the classifier to the training data svm_classifier.fit(X_train, y_train)

# Make predictions on the test data y_pred =
svm_classifier.predict(X_test)

# Calculate the accuracy of the model accuracy =
accuracy_score(y_test, y_pred) print(f"Accuracy: {accuracy:.2f}")
```

Output:

6.Naïve Bayes Classifier

Aim:
1. To implement the Naïve Bayes' algorithm for classification.
2. Train a Naïve Bayes' model using a given dataset and calculate class probabilities.
3. Evaluate the accuracy of the model on test data and analyze the results.

Theory: Naïve Bayes is a family of probabilistic machine learning algorithms based on Bayes' Theorem. It's particularly useful for classification tasks, where you want to predict the class label of an input based on its features. Despite its "Naïve" assumption, Naïve Bayes has been shown to work surprisingly well in many real-world scenarios, especially in text classification and other high-dimensional datasets.

1. Bayes' Theorem:
Naïve Bayes is built upon Bayes' Theorem, which is a fundamental concept in probability theory. Bayes' Theorem describes how to update the probability of a hypothesis (an event) based on new evidence. It's written as:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Where:
P(A | B) is the probability of event A given event
B. P(B | A) is the probability of event B given
event A. P(A) is the prior probability of event A.
P(B) is the evidence or marginal probability of event B.

2. Naïve Assumption:
The "Naïve" in Naïve Bayes refers to the assumption that the features (attributes) used to predict the class are conditionally independent of each other given the class label. In other words, the presence or absence of a particular feature doesn't affect the presence or absence of any other feature. This simplifies the calculations significantly and makes the algorithm computationally efficient.

3. Training:
During the training phase, Naïve Bayes estimates the probabilities needed for classification. It calculates the prior probabilities of each class based on the training data and then calculates the likelihood of each feature given each class.

For example, if you're classifying emails as "spam" or "not spam," you'd estimate the probability of certain words appearing in spam emails versus non-spam emails.

4. Prediction:
When making a prediction for a new input, Naïve Bayes calculates the probability of the input belonging to each class based on the features. It uses Bayes' Theorem to compute the posterior probability for each class, and the class with the highest posterior probability is chosen as the predicted class.

Types of Naïve Bayes Classifiers:
There are several variants of Naïve Bayes classifiers, including:

	- Gaussian Naïve Bayes: Assumes that the features follow a Gaussian distribution. - Multinomial Naïve Bayes: Used for discrete data, often in text classification (e.g., counting word occurrences). - Bernoulli Naïve Bayes: Used for binary data, such as presence or absence of features. - Categorical Naïve Bayes: Used for categorical data, where features have discrete values.

The following is the Python code to implement Naïve Bayes Classifier

(To run this code, Scikit-Learn must be installed (pip install scikit-learn).

| Python Code | ```python
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)
``` |
|---|---|

Another example with the following datasets

Match	Argentina	France	Result
1	1	0	Argentina
2	0	0	Draw
3	3	4	France
4	2	0	Argentina
5	0	0	Draw
6	1	0	Argentina
7	0	0	Draw
8	2	1	Argentina
9	0	2	France
10	1	0	Argentina
11	2	0	Argentina
12	3	4	France
13	4	2	Argentina

(To run this code, Pandas must be installed (pip install pandas).

```python
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score
import pandas as pd

# Load the football dataset from CSV
dataset = pd.read_csv('argfrc_dataset.csv')

# Separate features (X) and labels (y)
X = dataset[['Argentina', 'France']].values
y = dataset['Result'].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)

# 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)
```

Output	

7.K - Nearest Neighbors (K-NN)

Aim:
1. Implement the K-NN algorithm for classification or regression.
2. Apply the K-NN algorithm to a given dataset and predict the class or value for test data.
3. Evaluate the accuracy or error of the predictions and analyze the results

Theory:	K-Nearest Neighbors (K-NN) is a simple but powerful machine learning algorithm used for both classification and regression tasks.
	
	1. Intuition: K-NN is based on the idea that objects (data points) that are close to each other in a feature space are more likely to belong to the same class or have similar values (for regression).
	
	2. How it works:
	  - Classification: Given a new data point, K-NN finds the K nearest data points in the training dataset and assigns the class label that is most common among those K neighbors to the new point.
	  - Regression: For regression tasks, K-NN calculates the average (or another aggregation) of the target values of the K nearest neighbors and assigns this value to the new data point.
	
	3. Hyperparameter K: The choice of the hyperparameter K (the number of neighbors to consider) is critical. A small K may lead to a noisy model (sensitive to outliers), while a large K may lead to a biased model (smoothing over variations in the data). K is typically an odd number to avoid ties in voting.
	
	4. Distance Metric: K-NN uses a distance metric (e.g., Euclidean distance, Manhattan distance, etc.) to measure the similarity between data points. The choice of distance metric should be appropriate for your data and problem.
	
	5. Scaling Features: It's important to scale features before applying K-NN, especially when using distance-based metrics, to ensure that all features have equal influence on the results.
	
	6. Pros:
	  - Simple and easy to understand.
	  - No assumptions about the data distribution.
	  - Works well for both classification and regression tasks.
	  - Non-parametric (does not make assumptions about the functional form of relationships).
	
	7. Cons:
	  - Can be computationally expensive, especially for large datasets.
	  - Sensitive to the choice of K and the distance metric.
	  - Requires a sufficient amount of training data.
	  - May not perform well when the feature space is high-dimensional.
	
	8. Use Cases:

- K-NN is often used for tasks such as recommendation systems, image classification, and anomaly detection.
   - It can be used as a baseline model for comparison with more complex algorithms.

9. Model Evaluation: Common evaluation metrics for K-NN include accuracy (for classification) and mean squared error (for regression). Cross-validation is often used to estimate the model's generalization performance.

10. Implementation: Python libraries like scikit-learn provide easy-to-use implementations of K-NN for both classification and regression.

In summary, K-NN is a versatile and interpretable algorithm suitable for various tasks. It's important to choose appropriate values for K and the distance metric based on your data and problem domain to achieve good performance.

Dataset:	We use the iris dataset which is available in the public domain, although we can also create our own datasets We download the iris dataset and save in .csv format

The following is the Python code to implement K-NN Classifier

(To run this code, Scikit-Learn must be installed (pip install scikit-learn) and Pandas must be installed (pip install pandas).

Python Code	

Output	

8.Demo of TensorFlow Tools

Aim:
4. Explore and experiment with TensorFlow tools and libraries.
5. Perform a demonstration or mini-project showcasing the capabilities of the tools.
6. Discuss and present the findings and potential applications

Theory:	**1) Introduction to TensorFlow:** TensorFlow is an open-source machine learning framework developed by the Google Brain team. It is designed to facilitate the development of machine learning and deep learning models, particularly neural networks. TensorFlow is widely used in various fields, including computer vision, natural language processing, speech recognition, and more. Key aspects about TensorFlow 1. Tensors: TensorFlow is named after its core data structure, the "tensor." A tensor is a multi-dimensional array, which can be thought of as a generalization of vectors and matrices to higher dimensions. Tensors are fundamental for representing and manipulating data in machine learning models. 2. Graph-based computation: TensorFlow uses a graph-based approach to define and execute machine learning models. In this approach, you first create a computational graph that represents the operations in your model. Then, you feed data into this graph to perform computations. This allows for efficient execution, optimization, and distribution of computations across CPUs and GPUs. 3. Flexibility: TensorFlow offers flexibility in designing and implementing machine learning models. You can build various types of models, including feedforward neural networks, convolutional neural networks (CNNs), recurrent neural networks (RNNs), and more. TensorFlow also supports custom model architectures and operations, making it suitable for research and production applications. 4. High-level APIs: TensorFlow provides high-level APIs like Keras, which is integrated into TensorFlow as its official high-level API. Keras simplifies the process of building and training deep learning models, making it accessible to developers with varying levels of experience. 5. Ecosystem: TensorFlow has a rich ecosystem of libraries and tools that extend its functionality. Some of these include TensorFlow Serving for deploying models in production, TensorFlow Lite for mobile and embedded devices, and TensorFlow.js for running models in web browsers. **2) Installation and Setup** TensorFlow can be installed on individual machines and in case if it is not practically possible then we can explore cloud-based alternatives such as Google Colab, Microsoft Azure Notebooks, or AWS SageMaker provide pre-configured environments with TensorFlow, which can be accessed via a web browser

3) TensorFlow Tools and Libraries

Some key tools and libraries are as follows:

a) TensorFlow Hub: A library for reusable machine learning modules and pre-trained models.
b) TensorBoard: A visualization tool for inspecting and monitoring the training of machine learning models.
c) TensorFlow Lite: For deploying machine learning models on mobile and embedded devices.
d) TensorFlow.js: For running machine learning models in web browsers.

4) Demonstration or Mini-Project

We can select one or more TensorFlow tools or libraries to demonstrate or use in a mini-project. Some of the ideas are as follows:

a) Image Classification with TensorFlow Hub: We can show how to load a pre-trained image classification model from TensorFlow Hub and use it to classify images.
b) TensorBoard Visualization: We can train a simple neural network using TensorFlow and visualize its training progress and performance using TensorBoard.
c) Deploy a Model with TensorFlow Lite: If we have access to mobile devices, we can demonstrate the deployment of a pre-trained model using TensorFlow Lite and run it on a Smartphone or tablet.
d) Interactive Web Application with TensorFlow.js: We can create a simple web application that uses TensorFlow.js to perform image recognition or natural language processing tasks in real-time.

We will a practical demonstration using TensorFlow and a dataset for spam email classification

Before running the code we create two files having emails, one is spam mail and other is non-spam email and save them as Spam.txt and NSpam.txt

Spam.txt	Subject: Congratulations! You've Won $100,000 Cash Prize Dear George, I am thrilled to inform you that you are the lucky winner of our recent contest and have been awarded a cash prize of $100,000! Your participation and enthusiasm are truly appreciated, and we couldn't be happier to share this exciting news with you. Regards idontSmile
NSpam.txt	Subject: Invitation for Dinner Dear Friend, I hope this email finds you well. I wanted to extend a warm invitation to you for a dinner party on 18/8/2023 at my home next Friday, and it would be wonderful to have you join us. Best regards, iSmile

We must import the necessary libraries before running the code

| **Python Code** | ```python
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

# Step 1: Prepare a dataset of labeled emails (spam and non-spam) 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]

# Step 2: Tokenize and pad the email text data 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")

# Step 3: Define the neural network model 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')
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Step 4: Define training data and labels as NumPy arrays
# This is where you convert your training data and labels to NumPy arrays. # For this example, we will use the
same data as 'emails' and 'labels'. training_data = np.array(X_padded)
training_labels = np.array(labels)

# Step 5: Train the model
model.fit(training_data, training_labels, epochs=10)   # You can adjust the number of epochs

# Step 6: Test if 'Spam.txt' is spam or not file_path = "Spam.txt"

# Read the content of the 'Sdam.txt' file
with open(file_path, "r", encoding="utf-8") as file: sample_email_text = file.read()

# Tokenize and pad the sample email text
sequences_sample = tokenizer.texts_to_sequences([sample_email_text]) sample_email_padded =
pad_sequences(sequences_sample, maxlen=max_len, padding="post", truncating="post")

# Use the trained model to make predictions prediction =
model.predict(sample_email_padded)

# Set a classification threshold (e.g., 0.5) threshold = 0.5

# Classify the sample email based on the threshold if prediction > threshold:
    print(f"Sample Email ('{file_path}'): SPAM")
``` |

```
else:
    print(f"Sample Email ('{file_path}'): NOT SPAM")
```

We run the above code with the text file NSpam.txt (file_path = "NSpam.txt")**and get the following**

Output	Epoch 1/10
	1/1 [==============================] - 1s 963ms/step - loss: 0.6952 - accuracy: 0.5000
	Epoch 2/10
	1/1 [==============================] - 0s 13ms/step - loss: 0.6904 - accuracy: 0.7500
	Epoch 3/10
	1/1 [==============================] - 0s 14ms/step - loss: 0.6854 - accuracy: 0.7500
	Epoch 4/10
	1/1 [==============================] - 0s 15ms/step - loss: 0.6816 - accuracy: 1.0000
	Epoch 5/10
	1/1 [==============================] - 0s 28ms/step - loss: 0.6782 - accuracy: 1.0000
	Epoch 6/10
	1/1 [==============================] - 0s 12ms/step - loss: 0.6745 - accuracy: 1.0000
	Epoch 7/10
	1/1 [==============================] - 0s 17ms/step - loss: 0.6708 - accuracy: 1.0000
	Epoch 8/10
	1/1 [==============================] - 0s 28ms/step - loss: 0.6669 - accuracy: 1.0000
	Epoch 9/10
	1/1 [==============================] - 0s 24ms/step - loss: 0.6630 - accuracy: 1.0000
	Epoch 10/10
	1/1 [==============================] - 0s 13ms/step - loss: 0.6589 - accuracy: 1.0000
	1/1 [==============================] - 0s 139ms/step
	Sample Email ('NSpam.txt'): NOT SPAM

We run the above code with the text file Spam.txt (file_path = "Spam.txt"**) and get the following**

Output	Epoch 1/10
	1/1 [==============================] - 1s 934ms/step - loss: 0.6922 - accuracy: 0.7500
	Epoch 2/10
	1/1 [==============================] - 0s 13ms/step - loss: 0.6877 - accuracy: 0.5000
	Epoch 3/10
	1/1 [==============================] - 0s 11ms/step - loss: 0.6822 - accuracy: 1.0000
	Epoch 4/10
	1/1 [==============================] - 0s 14ms/step - loss: 0.6772 - accuracy: 1.0000
	Epoch 5/10
	1/1 [==============================] - 0s 14ms/step - loss: 0.6723 - accuracy: 1.0000
	Epoch 6/10
	1/1 [==============================] - 0s 14ms/step - loss: 0.6677 - accuracy: 1.0000
	Epoch 7/10
	1/1 [==============================] - 0s 14ms/step - loss: 0.6622 - accuracy: 1.0000
	Epoch 8/10
	1/1 [==============================] - 0s 16ms/step - loss: 0.6567 - accuracy: 1.0000
	Epoch 9/10
	1/1 [==============================] - 0s 11ms/step - loss: 0.6512 - accuracy: 1.0000
	Epoch 10/10
	1/1 [==============================] - 0s 12ms/step - loss: 0.6456 - accuracy: 1.0000
	1/1 [==============================] - 0s 66ms/step
	Sample Email ('Spam.txt'): SPAM