2025

Artificial Intelligence Practical

TYCS

Practical based on Machine learning, basically supervised learning and some search algorithms both Informed and Uninformed Search

Prof. Ismail H. Popatia
Maharashtra College

Index

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

```
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.

For Video demonstration of the practical click on the link or scan the QR-code

Iterative Depth-First Search https://youtu.be/2aBPyxr76kQ

Breadth-First Search https://youtu.be/u2UEttAkmEs

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:

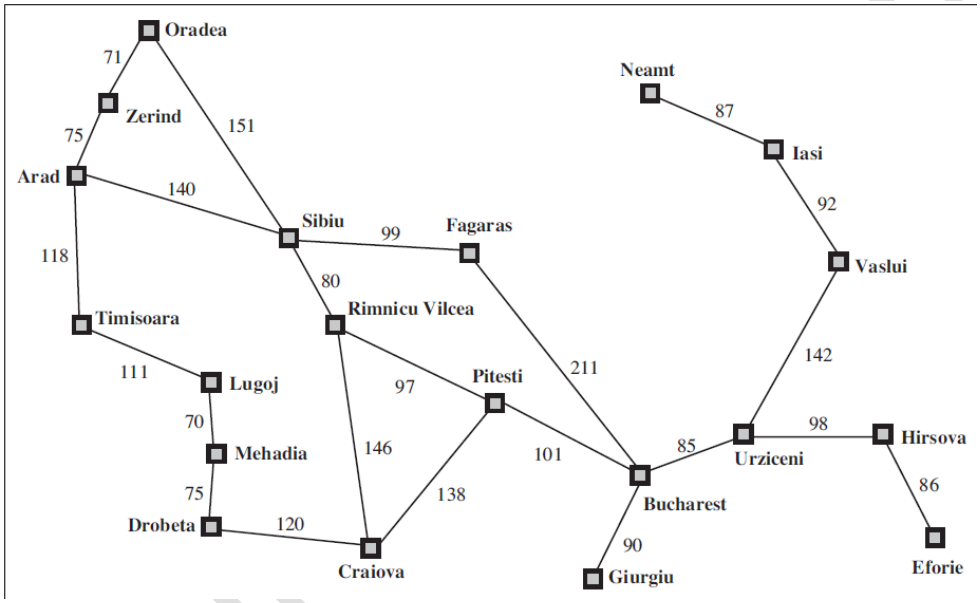
a) Initialize the open list, closed list, and set the g-value of the start node to 0.
b) Calculate the h-value for each node in the graph or grid based on a heuristic function. The heuristic function estimates the cost from each node to the goal node. Common heuristic functions include Euclidean distance, 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:

```
import heapq  # For using a priority queue (min-heap)

# Romania Map: Each city maps to its neighbors and the distance to them
romania_map = {
    'Arad': {'Zerind': 75, 'Sibiu': 140, 'Timisoara': 118},
    'Zerind': {'Arad': 75, 'Oradea': 71},
    'Oradea': {'Zerind': 71, 'Sibiu': 151},
    'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80},
    'Timisoara': {'Arad': 118, 'Lugoj': 111},
    'Lugoj': {'Timisoara': 111, 'Mehadia': 70},
    'Mehadia': {'Lugoj': 70, 'Dobreta': 75},
    'Dobreta': {'Mehadia': 75, 'Craiova': 120},
    'Craiova': {'Dobreta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},
    'Rimnicu Vilcea': {'Sibiu': 80, 'Craiova': 146, 'Pitesti': 97},
    'Fagaras': {'Sibiu': 99, 'Bucharest': 211},
    '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}
}

# Heuristic: Straight-line distance from each city to Bucharest
heuristic = {
    'Arad': 366, 'Bucharest': 0, 'Craiova': 160, 'Dobreta': 242, 'Eforie': 161,
    'Fagaras': 178, 'Giurgiu': 77, 'Hirsova': 151, 'Iasi': 226, 'Lugoj': 244,
    'Mehadia': 241, 'Neamt': 234, 'Oradea': 380, 'Pitesti': 98, 'Rimnicu Vilcea': 193,
    'Sibiu': 253, 'Timisoara': 329, 'Urziceni': 80, 'Vaslui': 199, 'Zerind': 374
}

# Node class to represent each city in the search
class Node:
    def __init__(self, city, parent=None, g=0, h=0):
        self.city = city          # Current city name
        self.parent = parent      # Parent node (used to trace back the path)
        self.g = g                # Cost from start to this city
        self.h = h                # Heuristic cost to goal
        self.f = g + h            # Total cost = actual + estimated

    def __lt__(self, other):
        return self.f < other.f   # Needed for heapq to compare nodes by f value

# A* search function
def a_star_search(start, goal):
    open_list = []  # Cities to explore (priority queue)
    heapq.heappush(open_list, Node(start, None, 0, heuristic[start]))  # Add start node
    closed_set = set()  # Cities already explored

    while open_list:
        current = heapq.heappop(open_list)  # Get the city with lowest f cost

        if current.city == goal:  # If goal is reached
            path = []
            total_cost = current.g
            while current:
                path.append(current.city)  # Trace back path from goal to start
                current = current.parent
            return path[::-1], total_cost  # Return reversed path and total cost

        closed_set.add(current.city)  # Mark current city as visited

        # Explore each neighbor of current city
        for neighbor, cost in romania_map[current.city].items():
```

```
            if neighbor in closed_set:
                continue  # Skip if already visited

            g = current.g + cost      # New cost from start to neighbor
            h = heuristic[neighbor]   # Heuristic to goal from neighbor
            neighbor_node = Node(neighbor, current, g, h)  # Create new node

            # Check if neighbor is already in open_list with better f
            skip = False
            for node in open_list:
                if node.city == neighbor and node.f <= neighbor_node.f:
                    skip = True
                    break

            if not skip:
                heapq.heappush(open_list, neighbor_node)  # Add neighbor to queue

    return None, float('inf')  # If goal not reachable

# Main program: Find path from Arad to Bucharest
if __name__ == "__main__":
    start_city = "Arad"
    goal_city = "Bucharest"
    path, cost = a_star_search(start_city, goal_city)

    if path:
        print("Path found:")
        print(" -> ".join(path))     # Display path step-by-step
        print("Total cost:", cost)   # Display total distance
    else:
        print("No path found.")
```

Output:

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

```
import math

# Heuristic: straight-line distance to Bucharest (used as h(n))
heuristics = {
    'Arad': 366, 'Zerind': 374, 'Oradea': 380, 'Sibiu': 253, 'Timisoara': 329,
    'Lugoj': 244, 'Mehadia': 241, 'Drobeta': 242, 'Craiova': 160, 'Rimnicu Vilcea':
193,
    'Fagaras': 178, 'Pitesti': 98, 'Bucharest': 0, 'Giurgiu': 77, 'Urziceni': 80,
    'Hirsova': 151, 'Eforie': 161, 'Vaslui': 199, 'Iasi': 226, 'Neamt': 234
}

# Graph: roads with costs between cities
romania_map = {
    'Arad': {'Zerind': 75, 'Timisoara': 118, 'Sibiu': 140},
    'Zerind': {'Arad': 75, 'Oradea': 71},
    'Oradea': {'Zerind': 71, 'Sibiu': 151},
    'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80},
    'Timisoara': {'Arad': 118, 'Lugoj': 111},
    'Lugoj': {'Timisoara': 111, 'Mehadia': 70},
    'Mehadia': {'Lugoj': 70, 'Drobeta': 75},
    'Drobeta': {'Mehadia': 75, 'Craiova': 120},
    'Craiova': {'Drobeta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},
    'Rimnicu Vilcea': {'Sibiu': 80, 'Craiova': 146, 'Pitesti': 97},
```

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

# Node class for RBFS
class Node:
    def __init__(self, city, parent=None, g=0):
        self.city = city
        self.parent = parent
        self.g = g  # Cost from start to this node
        self.h = heuristics[city]
        self.f = max(self.g + self.h, getattr(parent, 'f', 0))

# Recursive Best-First Search
def rbfs(node, goal, f_limit):
    print(f"Visiting {node.city}, f={node.f}, limit={f_limit}")

    if node.city == goal:
        return node, 0

    successors = []
    for neighbor, cost in romania_map[node.city].items():
        if node.parent and neighbor == node.parent.city:
            continue  # Avoid going back to parent

        g = node.g + cost
        child = Node(neighbor, parent=node, g=g)
        successors.append(child)

    if not successors:
        return None, math.inf

    while True:
        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 math.inf

        result, best.f = rbfs(best, goal, min(f_limit, alternative))

        if result is not None:
            return result, best.f
```

```
# Trace path from goal to start
def extract_path(node):
    path = []
    while node:
        path.append(node.city)
        node = node.parent
    return path[::-1]

# Main
if __name__ == "__main__":
    start = 'Arad'
    goal = 'Bucharest'

    start_node = Node(start)
    result, _ = rbfs(start_node, goal, math.inf)

    if result:
        path = extract_path(result)
        print("\nPath found:")
        print(" -> ".join(path))
        print("Total cost:", result.g)
    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.

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:

```
# 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()
```

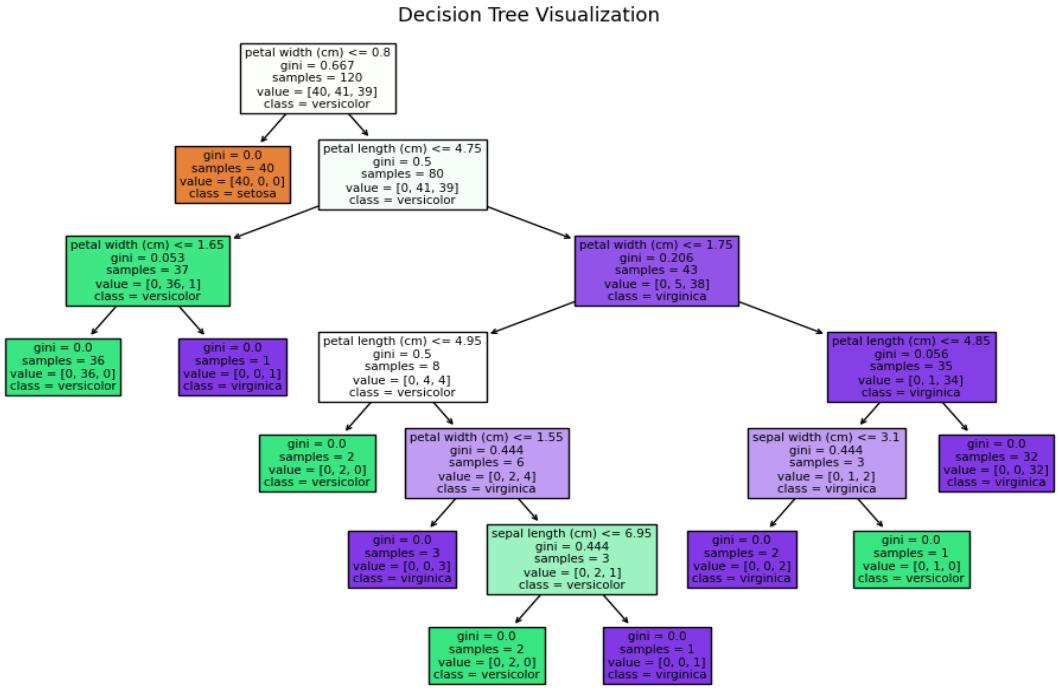
```
# Fit the classifier to the training data
clf.fit(X_train, y_train)

# 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:



For Video demonstration of the practical click on the link or scan the QR-code

https://youtu.be/fBfRp0to2uA

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

```
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
        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

For Video demonstration of the practical click on the link or scan the QR-code

https://youtu.be/PGyUlMe2dPg

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:

```
# 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:

For Video demonstration of the practical click on the link or scan the QR-code

https://youtu.be/ZW4dZH4xBUI

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

```
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).

```
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	

For Video demonstration of the practical click on the link or scan the QR-code

https://youtu.be/_tcjQjh2rEE

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

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Load the dataset from the CSV file
df = pd.read_csv('Iris.csv')

# Extract relevant columns for the feature matrix (exclude 'ID' and 'Target' columns)
X = df.drop(['ID', 'Target'], axis=1).values

# Target variable for classification
y = df['Target'].values

# Split the data 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)

# Define the value of k for K-NN
k = 3

# Classification with K-NN
clf = KNeighborsClassifier(n_neighbors=k)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
```

	# Evaluate classification accuracy
	classification_accuracy = accuracy_score(y_test, y_pred)
	print("Classification Accuracy:", classification_accuracy)

Output	

For Video demonstration of the practical click on the link or scan the QR-code

https://youtu.be/inZu6CectOc

Demo of OpenAI/TensorFlow Tools

Aim:
1. Explore and experiment with OpenAI or TensorFlow tools and libraries.
2. Perform a demonstration or mini-project showcasing the capabilities of the tools.
3. Discuss and present the findings and potential applications.

Theory:

TensorFlow is an open-source platform for machine learning developed by Google. It allows developers to build and train models for a wide variety of tasks such as image classification, natural language processing, and recommendation systems.

In this experiment, we:

- Prepare a small dataset of emails labeled as spam (1) or non-spam (0).
- Use **Tokenizer** and **pad_sequences** from TensorFlow Keras for preprocessing text data.
- Build a neural network model with **Embedding**, **Flatten**, and **Dense** layers.
- Train the model using labeled data.
- Test the model with a new email to predict if it is spam.

Key Concepts Used:

1. **Tokenization:** Converting text into numerical tokens that can be processed by a machine learning model.
2. **Padding:** Ensuring all sequences have the same length by adding zeros or truncating.
3. **Embedding Layer:** Converts word indices into dense vectors of fixed size.
4. **Binary Classification:** Output layer uses sigmoid activation to classify into two categories (spam or not spam).

Tools & Technologies Used:

- **Language:** Python
- **Libraries:** TensorFlow, NumPy
- **IDE:** Any Python IDE or Jupyter Notebook
- **Dataset:** Custom small set of labeled emails

Algorithm / Procedure:

1. **Prepare Dataset**: Create a list of sample email texts and assign spam (1) or non-spam (0) labels.
2. **Tokenization**:
    - Create a Tokenizer object.
    - Fit tokenizer on email dataset.
    - Convert emails to sequences of integers.
3. **Padding**: Pad sequences to make them of equal length.
4. **Model Creation**:
    - Add an **Embedding layer** for text vector representation.
    - Flatten the output.

- Add hidden layers with **ReLU** activation.
- Add output layer with **sigmoid** activation for binary classification.

5. **Compile Model**: Use Adam optimizer and binary crossentropy loss function.
6. **Train Model**: Fit the model with training data for a fixed number of epochs.
7. **Test Model**:
   - Read new email text from "Spam.txt".
   - Tokenize and pad it.
   - Predict spam probability using the trained model.
   - Classify as SPAM or NOT SPAM based on threshold (0.5).

The following is the Python code

```
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")
```

For Video demonstration of the practical click on the link or scan the QR-code

https://youtu.be/WB7W8zvggNM

Association Rule Mining

Aim:

1. Implement the Association Rule Mining algorithm (e.g., Apriori) to find frequent itemsets.
2. Generate association rules from the frequent itemsets and calculate their support and confidence.
3. Interpret and analyze the discovered association rules.

Theory: Association Rule Mining (ARM) is a technique used to discover interesting relationships and correlations among a set of items in a dataset. It's often used in market basket analysis to find patterns like "customers who buy bread and milk also tend to buy butter." The goal is to generate rules of the form **{Item A, Item B} -> {Item C}**.

The Apriori Algorithm

The Apriori algorithm is a classic and foundational algorithm for ARM. It works on the principle that if an itemset is frequent, then all of its subsets must also be frequent. Conversely, if an itemset is infrequent, then all of its supersets will also be infrequent. This principle helps to significantly reduce the number of itemsets that need to be considered.

The process involves two main steps:

1. **Finding Frequent Itemsets:** The algorithm iteratively generates itemsets of increasing size (e.g., single items, pairs of items, triplets, etc.) and prunes any that do not meet a minimum **support** threshold.
2. **Generating Association Rules:** Once all frequent itemsets are found, the algorithm generates rules from them. For each rule, it calculates its **confidence** and **lift** to determine its strength.

Key Metrics for Evaluation

Support: This measures how frequently an itemset appears in the dataset. A high support value means the itemset is common.

$$\textbf{Support(X} \rightarrow \text{Y)} = \frac{\text{Total number of transactions}}{\text{Number of tranactions containing X and Y}}$$

Confidence: This measures how often the rule is found to be true. A high confidence value means that whenever the antecedent (the "if" part) appears, the consequent (the "then" part) is also likely to appear.

$$\textbf{Confidence(X} \rightarrow \text{Y)} = \frac{\text{Support(X)}}{\text{Support(XUY)}}$$

Lift: This measures how much more likely the consequent is to occur *given* the antecedent, compared to its baseline probability. A lift value greater than 1 indicates a positive correlation between the items in the rule, a value less than 1 indicates a negative correlation, and a value of 1 indicates no correlation.

$$\textbf{Lift(X} \rightarrow \text{Y)} = \frac{\text{Support(Y)}}{\text{Support(X} \rightarrow \text{Y)}}$$

Python Code:

```
pip install mlxtend
from mlxtend.frequent_patterns import apriori
from mlxtend.frequent_patterns import association_rules
import pandas as pd

dataset =[
   ['milk','bread','nuts'],
   ['milk','bread'],
   ['milk','eggs','nuts'],
   ['milk','bread','eggs'],
   ['bread','nuts']
]

df = pd.DataFrame(dataset)
df_encoded = pd.get_dummies(df,prefix='',prefix_sep='')
frequent_itemsets = apriori(df_encoded,min_support=0.6,use_colnames=True)

print("Frequent Itemsets:")
print(frequent_itemsets)

rules = association_rules(frequent_itemsets,metric='confidence',min_threshold=0.7)
print("\nAssociation Rules:")
print(rules)
```

Output:

```
Frequent Itemsets:
   support       itemsets
0      0.8         (milk)
1      0.6        (bread)
2      0.6  (bread, milk)

Association Rules:
  antecedents consequents  antecedent support  consequent support  support  \
0     (bread)      (milk)                 0.6                 0.8      0.6
1      (milk)     (bread)                 0.8                 0.6      0.6

   confidence  lift  representativity  leverage  conviction  zhangs_metric  \
0        1.00  1.25               1.0      0.12         inf            0.5
1        0.75  1.25               1.0      0.12         1.6            1.0

   jaccard  certainty  kulczynski
0     0.75      1.000       0.875
1     0.75      0.375       0.875
```

For Video demonstration of the practical click on the link or scan the QR-code

https://youtu.be/v0r_hxNoF0Q