Theory-1

Breadth First Search & Iterative Depth First Search

Breadth First Search

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

Key Points:
1. Breadth-First Traversal. BFS is a graph traversal algorithm that explores vertices in breadthward motion. It starts from the root (or any arbitrary node in the case of graphs) and explores all the neighbor nodes at the current depth before moving on to nodes at the next level of depth.

2. Queue Data Structure: BFS uses a queue data structure to manage the order of exploration. The queue follows the First-In-First-Out (FIFO) principle, ensuring that nodes are visited in the order they are added to the queue.

3. Level Order Traversal: BFS naturally performs a level-order traversal of the graph. This means that it explores all nodes at the current depth before moving on to the nodes at the next depth level.

4. Visited Nodes: To avoid revisiting nodes and prevent infinite loops, BFS maintains a set (or array) of visited nodes. This set helps ensure that each node is visited only once.

Iterative Depth First Search

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

Key Points:
1. Combines BFS and DFS: Iterative Depth-First Search (IDFS) is a hybrid search algorithm that combines the benefits of both Breadth-First Search (BFS) and Depth-First Search (DFS). It performs a series of DFS searches with increasing depth limits.

2. Incremental Depth Limits: IDFS starts by performing a DFS search with a depth limit of 1, then 2, and so on. It gradually increases the depth limit in each iteration.

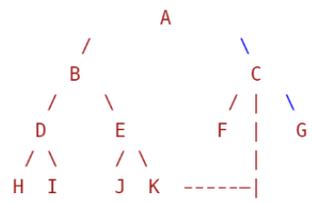
3. Memory Efficiency: One of the key advantages of IDFS is its memory efficiency. Unlike BFS, which can consume a lot of memory due to the need to store all nodes at a level, IDFS only needs to keep track of nodes within the current depth limit.

Practical-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.

Graph:



Problem Description: Find shortest path from A to K

Aim-1: Implement the Breadth First Search algorithm to solve a given problem

Source code:

```
def BFS(graph, start, goal):
    visited = []  # list for visited nodes
    queue = []    # list for queue
    parent = {}   # dictionary to track the path

    visited.append(start)  # add start node into visited list
    queue.append(start)    # add start node into queue

    while queue:  # while queue is not empty
        n = queue.pop(0)  # pop the front node in queue

        if n == goal:  # check if we reached the goal
            path = [] # list for path
            while n is not None:  # reconstruct the path while n is not empty
                path.append(n)
                n = parent.get(n)
            print(f"BFS: Shortest path from {start} to {goal}: {' -> '.join(reversed(path))}")
            return

        for neighbor in graph[n]:  # visit all neighbors of current node
            if neighbor not in visited:  # if neighbor is not visited
                visited.append(neighbor)  # visit the neighbor
                queue.append(neighbor)     # add the neighbor node into queue
                parent[neighbor] = n       # set parent for path reconstruction

    print(f"BFS: {goal} not found")


graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F', 'K'],
    'D': ['B', 'H', 'I'],
    'E': ['B', 'J', 'K'],
    'F': ['C', 'K'],
    'G': ['C'],
    'H': ['D'],
    'I': ['D'],
    'J': ['E'],
    'K': ['E']
}

# Run BFS
BFS(graph, "A", "K")
```

Output:

```
BFS: Shortest path from A to K: A -> C -> K
```

Aim-2: Implement the Iterative Depth First Search algorithm to solve the same problem

Source code:

```
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'K', 'G'],
    'D': ['H', 'I'],
    'E': ['J', 'K'],
    'F': ['K'],
    'G': [],
    'H': [],
    'I': [],
    'J': [],
    'K': []
}


def DFS(currentNode, destination, graph, maxDepth, path):
    path.append(currentNode)  # Add current node to the path
    #print("Current Node: ", currentNode)

    if currentNode == destination:
        return True, path  # Return True and the current path

    if maxDepth <= 0:
        path.pop()  # Remove the current node before backtracking
        return False, path

    for node in graph[currentNode]:
        found, resultPath = DFS(node, destination, graph, maxDepth - 1, path)
        if found:
            return True, resultPath  # If found, return True and the path

    path.pop()  # Remove the current node before backtracking
    return False, path

def IDFS(currentNode, destination, graph, maxDepth):
    for i in range(maxDepth):
        found, resultPath = DFS(currentNode, destination, graph, i, [])
        if found:  # If a path is found
            return resultPath  # Return the shortest path
    return None  # No path found

# Driver code to find and print the shortest path
shortest_path = IDFS('A', 'K', graph, 4)
if shortest_path is None:
    print("Path is not available")
else:
    print("IDFS: Shortest path from A to K:", " -> ".join(shortest_path))
```

Output:

```
IDFS: Shortest path from A to K: A -> C -> K
```

Aim-3: Compare the performance and efficiency of both algorithms.

The comparison of efficiency and performance between Breadth-First Search (BFS) and Iterative Depth-First Search (IDFS) algorithms is given below:

Performance:
Time Complexity:
BFS: The time complexity of BFS is O(V + E), where V represents the number of vertices and E represents the number of edges in the graph, and 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.

IDFS: The time complexity of IDFS is O(b^d), where b is the branching factor and d is the depth of the goal node or the depth at which the DFS function iteration terminates. IDFS 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 IDFS 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, IDFS may explore the entire search space.

Space Complexity:
BFS: Space Complexity of BFS is O(n), where n is the number of nodes in the current level of the graph. 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.

IDFS: Space Complexity of IDFS is O(d), where d is the depth of the goal node, accounting for the maximum call stack size in a recursive DFS. IDFS 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 IDFS is dependent on the maximum depth of the search.

Efficiency:
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.

IDFS: IDFS 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 IDFS to avoid exploring unnecessary nodes at greater depths. IDFS is complete and guarantees finding the optimal solution when the branching factor is finite.

Based on the above analysis, BFS and IDFS 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.

IDFS 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, IDFS 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 IDFS. If memory is limited, IDFS may be a more practical choice, while BFS may be preferable when finding the optimal path is crucial and memory is not a constraint.

Result and Discussion:

We have successfully implemented and analyzed BFS and IDFS effectiveness and performance.

Learning Outcomes:

1. Successfully implemented BFS and IDFS algorithm.
2. Understood the strengths and weaknesses of both algorithms.

Course Outcomes:

1. We have learned about BFS and IDFS algorithm.
2. We have analyzed the effectiveness of both the algorithms.

Conclusion:

We have successfully implemented and analyzed BFS and IDFS.

Viva Question:

1. What is Breadth-First-Search?
2. What is Depth-First-Search?
3. What is Iterative-Depth-First-Search?
4. What are the disadvantages of BFS?

For Faculty Use

Correction Parameters	Formative Assessment [ ]	Timely completion practical [ ]	Attendance Learning Attitude[ ]

Theory-2
A* Search and Recursive Best-First Search

A* (A-star) search

A* (A-star) search is an informed search as it comes under the category of informed search also called as heuristic search. It 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.

Recursive Best-First Search (RBFS)

Recursive Best-First Search (RBFS) algorithm is a memory-bounded heuristic search algorithm and it requires less memory than A* algorithm. It is a simple recursive algorithm that attempts to mimic the operation of standard bst-first search, but using only linear space.It is used for finding the optimal path from a starting point to a goal in a graph or search space.It is a variant of the Best-First Search algorithm, which is used for searching through large sets of possibilities.

It structure is similar to that of recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the f-limit variable to keep track of the f-value of the best alternative path available from any ancestor of the current node.

If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the f-value of each node along the path with a backed-up value i.e. the best f-value of its children.

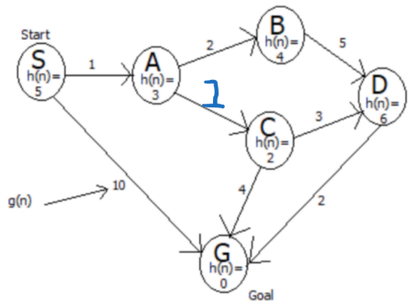
RBFS is more efficient than A* search algorithm as it requires less memory than A* search.

Practical-2: **A* Search and Recursive Best-First Search**

Aim:

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

Graph:



f(n)= g(n)+ h(n)

Problem description: Find the shortest path from S to G

Aim-1: Implement the A* Search algorithm for solving a pathfinding problem.

Source code:

```
def astaralgo(start_node,stop_node):

    open_set = set(start_node)
    closed_set= set()

    g={}
    parents = {}

    g[start_node] = 0

    parents[start_node]  = start_node

    while len(open_set) > 0:
        n=None

        for v in open_set:
            if n==None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n=v
        if n==stop_node or Graph_nodes[n]==None:
                pass

        else:

            for(m,weight) in get_neighbors(n):
                if m not in open_set  and m not in closed_set:
                        open_set.add(m)
                        parents[m]=n
                        g[m] = g[n] + weight
                else:
                        if g[m] > g[n] +  weight:
                            g[m] = g[n] + weight
                            parents[m]=n

                            if m in closed_set:
                                closed_set.remove(m)
                                open_set.add(m)
        if n==None:
            print("path does not exist!")
            return None
```

```
        if n==stop_node:
            path=[]
            total_cost = g[n]

            while parents[n]!=n:
                path.append(n)
                n=parents[n]

            path.append(start_node)

            path.reverse()

            print("Optimal Path found: {}".format(path))
            print("Total Path Cost: {}".format(total_cost))
            return path


        open_set.remove(n)
        closed_set.add(n)
    print("Path does not exist!")
    return None


def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]

    else:
        return None

def heuristic(n):
    H_dist = {
        'S' : 5,
        'A' : 3,
        'B' : 4,
        'C' : 2,
        'D' : 6,
        'G' : 0,
    }
    return H_dist[n]


Graph_nodes = {
    'S' : [('A',1),('G',10)],
    'A' : [('B',2),('C',1)],
    'B' : [('D',5)],
    'C' : [('D',3), ('G',4)],
    'D' : [('G',2)],
    'G': None
}

astaralgo('S','G')
```

Output:

```
Optimal Path found: ['S', 'A', 'C', 'G']
Total Path Cost: 6
```

Aim-2: Implement the Recursive Best-First Search algorithm for the same problem.

Source code:

```
from queue import PriorityQueue

# Heuristic distances for each node
H_dist = {
    'S': 5,
    'A': 3,
    'B': 4,
    'C': 2,
    'D': 6,
    'G': 0,
}

# Graph representation
Graph_nodes = {
    'S': [('A', 1), ('G', 10)],
    'A': [('B', 2), ('C', 1)],
    'B': [('D', 5)],
    'C': [('D', 3), ('G', 4)],
    'D': [('G', 2)],
    'G': None
}

def recursive_best_first_search(current_node, target, visited, current_cost, parent):
    # Base case: if we reach the target
    if current_node == target:
        path = []
        while current_node is not None:
            path.append(current_node)
            current_node = parent[current_node]
        path.reverse()

        print(f"Optimal Path: {' -> '.join(path)}")
        print(f"Total Cost: {current_cost}")
        return True

    visited.add(current_node)

    # Create a list of neighbors with their costs
    neighbors = Graph_nodes.get(current_node, [])
    # Sort neighbors based on heuristic + cost
    neighbors.sort(key=lambda x: (current_cost + x[1] + H_dist[x[0]]))

    # Explore neighbors
    for neighbor, cost in neighbors:
        if neighbor not in visited:
            parent[neighbor] = current_node
            total_cost = current_cost + cost

            # Recursive call
            if recursive_best_first_search(neighbor, target, visited, total_cost, parent):
                return True

            # Backtrack
            del parent[neighbor]

    return False

def best_first_search(source, target):
    visited = set()  # To keep track of visited nodes
    parent = {source: None}  # To reconstruct the path
    recursive_best_first_search(source, target, visited, 0, parent)

best_first_search('S', 'G')
```

Output:

```
Optimal Path: S -> A -> C -> G
Total Cost: 6
```

Aim-3: Compare the performance and effectiveness of both algorithms.

Source code:

Performance:

A* Search

- Time Complexity: The time complexity of A* is generally expressed as O(b^d), where:
- b is the branching factor (the average number of successors per state).
- d is the depth of the solution.
- This complexity arises because A* explores all possible paths to find the optimal one, influenced heavily by the heuristic used. A well-designed heuristic can significantly reduce the number of nodes expanded, improving performance in practice.

- Space Complexity: A* has a space complexity of O(b^d) as well, since it needs to store all generated nodes in memory. This can be a significant drawback when dealing with large search spaces, as it may exhaust available memory resources.

Recursive Best First Search (RBFS)

- Time Complexity: The time complexity of RBFS is also $O(b^d)$ in the worst case. However, it can vary depending on the heuristic used and how effectively it prunes the search space.
- RBFS performs depth-first search with backtracking, which can lead to fewer nodes being expanded compared to A*, especially in cases with limited memory or when the search space is large.

- Space Complexity: RBFS has a more favorable space complexity of O(d), where d is the depth of the shallowest goal node. This is because RBFS only needs to store a single path from the root to a leaf node along with some additional information for backtracking. This makes RBFS more memory-efficient than A* in many scenarios.

Effectiveness:

A* Search

- Effectiveness: A* is highly effective for pathfinding problems due to its ability to use heuristics that guide its search towards the goal. It guarantees finding the optimal path if the heuristic is admissible (never overestimates the cost) and consistent (satisfies the triangle inequality). Its performance can be significantly improved with a good heuristic.

Recursive Best First Search (RBFS)

- Effectiveness: While RBFS can also find optimal paths, its effectiveness heavily relies on the quality of heuristics. It may not be as efficient as A* in terms of exploring fewer nodes, especially when dealing with complex graphs or when optimal solutions are deep within the search space. However, its lower memory usage can make it preferable in memory-constrained environments.

Summary Table:

Feature	A* search	Recursive Best First Search
Time Complexity	$O(b^d)$	$O(b^d)$
Space Complexity	$O(b^d)$	$O(d)$
Optimality Guarantee	Yes	Yes
Memory Efficiency	Less efficient	More efficient
Heuristic Dependence	High	Moderate

Result and Discussion: We have successfully implemented and analyzed A* search and recursive best-first search effectiveness and performance.

Learning Outcomes:

1. Successfully implemented A* search and Recursive best-first search algorithms.
2. Understood the strengths and weaknesses of both algorithms.

Course Outcomes:

1. We have learned about A* search and recursive best-first search algorithm.
2. We have analyzed the effectiveness of both the algorithms.

Conclusion:

We have successfully implemented and analyzed A* search and recursive best-first search.

Viva Question:

1. What is an Informed search?
2. What is A* search?
3. What is Recursive Best First Search?
4. What is heuristic value?

For Faculty Use

Correction Parameters	Formative Assessment [ ]	Timely completion practical [ ]	Attendance Learning Attitude[ ]

Theory-3
Decision Tree Learning

A Decision Tree is a supervised machine learning algorithm used for classification and regression tasks. It represents decisions and their possible consequences in a tree-like structure. Each internal node of the tree represents a feature (attribute), each branch represents a decision rule, and each leaf node represents an outcome.

To build a Decision Tree, we first need to load our dataset, which can be efficiently handled using the Pandas library in Python. pandas allows us to read various data formats, including CSV files, and manipulate the data for analysis.

Once the dataset is loaded, we can preprocess it as necessary, converting categorical variables into numerical formats if required. After preparing the data, we can implement the Decision Tree using the Scikit-learn library, which provides a robust framework for creating and training machine learning models.

Finally, to visualize the structure of the Decision Tree and understand its decision-making process, we can use visualization libraries such as Matplotlib or dtreeviz.Matplotlib allows for basic plotting capabilities, while dtreeviz offers more advanced visualizations specifically tailored for decision trees, making it easier to interpret how decisions are made based on input features.

In summary, the workflow involves:
1. Loading the dataset with Pandas.
2. Implementing the Decision Tree with Scikit-learn.
3. Visualizing the tree using Matplotlib or dtreeviz to gain insights into the model's decisions.

Key Components of a Decision Tree:
Root Node: The top node in the tree where the first split occurs.
Decision Nodes: Nodes that represent tests on features.
Leaf Nodes: Terminal nodes that signify the predicted outcome.
Branches: The connections between nodes that represent decision paths.

Advantages of Decision Trees:
Easy to understand and interpret.
Can handle both numerical and categorical data.
Requires little data preprocessing (no need for normalization).
Can be visualized easily.

Disadvantages:
Prone to overfitting, especially with complex trees.
Sensitive to noisy data.

Practical-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.

Dataset:

https://gist.github.com/greyart93/bb3dd1d009a87412d7219b0952669e1a#file-play_tennis-csv

	outlook	humidity	windy	play
0	overcast	high	False	yes
1	overcast	normal	True	yes
2	overcast	high	True	yes
3	overcast	normal	False	yes
4	rainy	high	False	yes
5	rainy	normal	False	yes
6	rainy	normal	True	no
7	rainy	normal	False	yes
8	rainy	high	True	no
9	sunny	high	False	no
10	sunny	high	True	no
11	sunny	high	False	no
12	sunny	normal	False	yes
13	sunny	normal	True	yes

Aim-1: Implement the Decision Tree Learning algorithm to build a decision tree for a given dataset.

Source code:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import accuracy_score
import requests
from io import StringIO

url = "https://shorturl.at/OFeB1"
headers = {"User-Agent": "Firefox/66.0"}
req = requests.get(url, headers=headers)
data = StringIO(req.text)

df = pd.read_csv(data)

df['outlook'] = df['outlook'].map({'overcast': 0, 'sunny': 1, 'rainy': 2})
df['humidity'] = df['humidity'].map({'high': 0, 'normal': 1})
df['windy'] = df['windy'].map({False: 0, True: 1})
df['play'] = df['play'].map({'no': 0, 'yes': 1})

X = df[['outlook', 'humidity', 'windy']]
y = df['play']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1)

clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)
```

Output:

```
DecisionTreeClassifier
DecisionTreeClassifier()
```

Aim-2: Evaluate the accuracy and effectiveness of the decision tree on test data.
Source code (continued):

```
y_pred = clf.predict(X_test)
accuracy_score(y_test, y_pred)
```

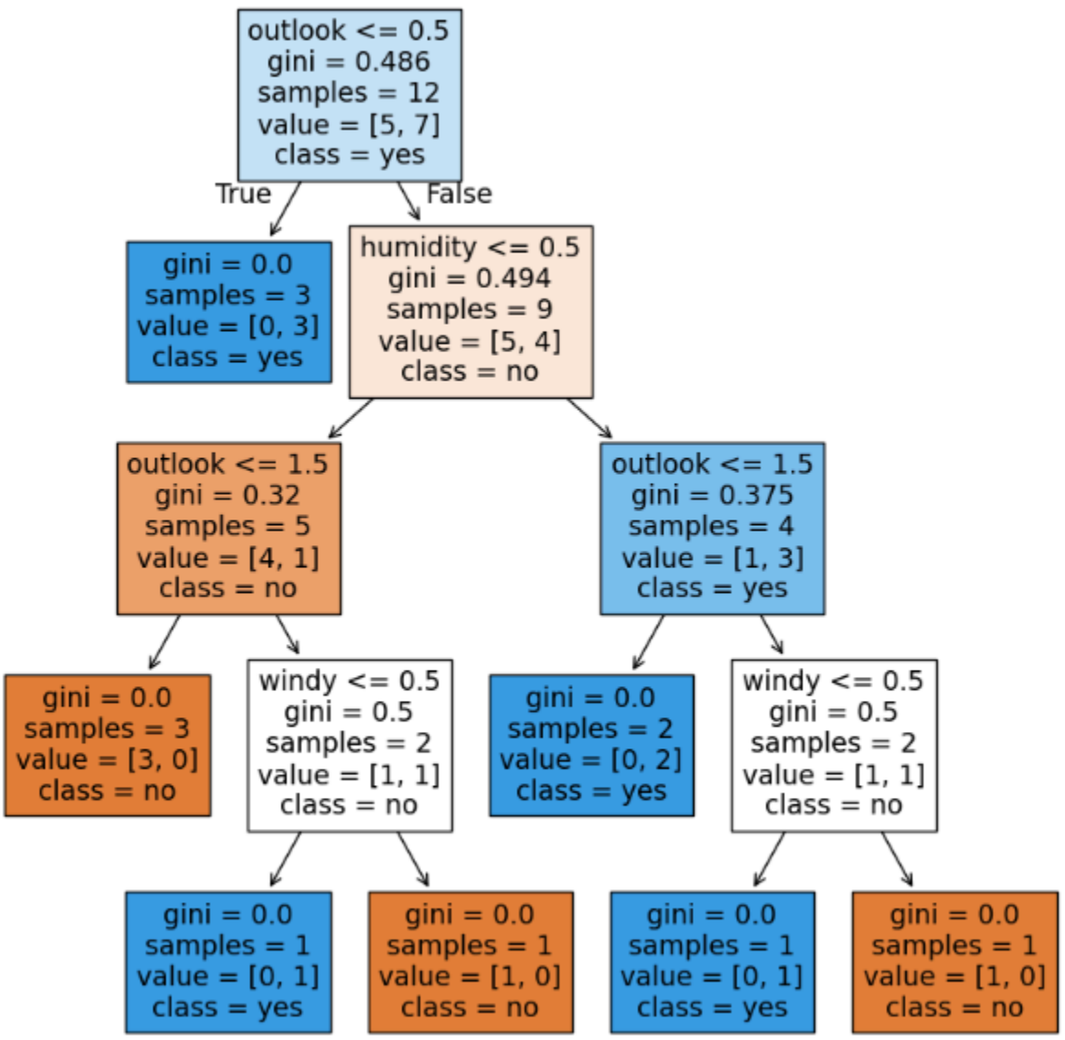
Output:

```
accuracy_score(y_test, y_pred)

1.0
```

Aim-3: Visualize and interpret the generated decision tree.
Source code:

```
from matplotlib.pylab import rcParams
rcParams['figure.figsize'] = 10, 10

plot_tree(clf, feature_names=X.columns, class_names=['no', 'yes'], filled=True)
```

Output:



Result and Discussion:

We have successfully implemented decision-tree learning algorithm and analyzed its effectiveness and performance.

Learning Outcomes:

1. Successfully implemented decision-tree learning algorithms.
2. Understood the accuracy and effectiveness of the decision tree on test data.

Course Outcomes:

1. We have learned about decision tree learning algorithm.
2. We have analyzed the effectiveness of this algorithm.

Conclusion:

We have successfully implemented and analyzed the decision tree learning algorithm.

Viva Question:

1. What is decision tree?
2. Which python library is used to implement decision tree?
3. Which python library is used to read the csv file?
4. Which python library is used to visualize the decision tree?

For Faculty Use

Correction Parameters	Formative Assessment [ ]	Timely completion practical [ ]	Attendance Learning Attitude[ ]

Theory-4
Feed Forward Backpropagation Neural Network

Artificial Neural Network (ANN):

- ANN is an interconnected group of artificial neurons and is inspired by the neurons in human brains.
- The group of neurons creates a layer of neurons, there are 3 types of layers in ANN:
- Input, Hidden, and Output layer
- Each network typically have one input and output layer but can have more than one hidden layer.
- The Input layer contains input neurons , similarly Hidden and Output layer contains hidden and output neurons respectively.
- Each Hidden layer and Output layer have a bias value associated with them.
- The branches connect the neurons.
- Each branches have weight associated with them.
- The aim of the input neurons is to provide input values to hidden nodes.
- Then these values are processed in hidden nodes/ neurons as:
- $H = \sum(I * W) + b$
- H is hidden node
- Where I is input value
- W is the weight of the branch
- b is the bias of hidden node
- This calculation is done by activation function to calculate the output of the hidden node
- Then the output of hidden node is passed to output node, with the same formalu as above
- Then this calculated output value ($\hat{y}$) is compared to the desired or target output value (y) to find the error between these two values
- After that we backtrack the nodes and updates the weights to get closer to the desired output i.e. error = 0, this operation is a trial and error method, it is repeated until we get the desired output

Forward Feed:

- The flow or movement of information from left to right i.e from input layer to output layer in ANN is called as forward feed or forward propagation
- It does not create a loop and is forward in nature i.e. the signal goes only forward and it does not go back.
- During forward propagation, the input is fed into the network and the network calculates the output

Back propagation:

- The flow or movement of information from right to left i.e. from output layer to input layer in ANN is called as back propagation
- During back propagation, the error between the predicted output and the actual output is calculated, and the weights and bias of each neuron are adjusted to reduce the error and get an output value closer to the desired or target output value .

Practical-4: Feed Forward 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.

Aim-1: Implement the Feed Forward Backpropagation algorithm to train a neural network.

Source code:

```
class NeuralNetwork(object):
    def __init__(self):
        #parameters
        self.inputSize = 2
        self.outputSize = 1
        self.hiddenSize = 3

        #weights
         # (2x3) weight matrix from input to hidden layer
        self.W1 = np.random.rand(self.inputSize, self.hiddenSize)
         # (3x1) weight matrix from hidden to output layer
        self.W2 = np.random.rand(self.hiddenSize, self.outputSize)

    def feedForward(self, X):
        #forward propogation through the network
        self.z = np.dot(X, self.W1)
        self.z2 = self.sigmoid(self.z) #activation function
        #dot product of hidden layer (z2) and second set of weights (3x1)
        self.z3 = np.dot(self.z2, self.W2)
        output = self.sigmoid(self.z3)
        return output

    def sigmoid(self, s, deriv=False):
        if (deriv == True):
            return s * (1 - s)
        return 1/(1 + np.exp(-s))

    def backward(self, X, y, output):
        #backward propogate through the network
        self.output_error = y - output # error in output
        self.output_delta = self.output_error * self.sigmoid(output, deriv=True)

        #z2 error: how much our hidden layer weights contribute to output error
        self.z2_error = self.output_delta.dot(self.W2.T)
         #applying derivative of sigmoid to z2 error
        self.z2_delta = self.z2_error * self.sigmoid(self.z2, deriv=True)

         # adjusting first set (input → hidden) weights
        self.W1 += X.T.dot(self.z2_delta)
         # adjusting second set (hidden → output) weights
        self.W2 += self.z2.T.dot(self.output_delta)

    def train(self, X, y):
        output = self.feedForward(X)
        self.backward(X, y, output)

NN = NeuralNetwork()
```

Aim-2: Use a given dataset to train the neural network for a specific task.
Source code (continued):

```
# X = (hours sleeping, hours studying), y = test score of the student
X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)

# scale units
X = X/np.amax(X, axis=0) #maximum of X array
y = y/100 # maximum test score is 100
```

Aim-3: Evaluate the performance of the trained network on test data.
Source code (continued):

```
for i in range(1000): #trains the NN 1000 times
    if (i % 100 == 0):
        print("Loss: " + str(np.mean(np.square(y - NN.feedForward(X)))))
    NN.train(X, y)

print("Input: " + str(X))
print("Actual Output: " + str(y))
print("Loss: " + str(np.mean(np.square(y - NN.feedForward(X)))))
print("\n")
print("Predicted Output: " + str(NN.feedForward(X)))
```

Output:

```
Loss: 0.04900842792395361
Loss: 8.468318390394242e-05
Loss: 7.351116939465712e-05
Loss: 6.824135773392977e-05
Loss: 6.365662172566185e-05
Loss: 5.9656106811613264e-05
Loss: 5.615859128996524e-05
Loss: 5.309266770823725e-05
Loss: 5.039594272773297e-05
Loss: 4.801422381335035e-05
Input: [[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output: [[0.92]
 [0.86]
 [0.89]]
Loss: 4.590070499393195e-05


Predicted Output: [[0.91054416]
 [0.8621394 ]
 [0.89661152]]
```

Result and Discussion: We have successfully implemented Feed Forward Backpropagation Neural Network algorithm and analyzed its performance.

Learning Outcomes:

1. Successfully implemented Feed Forward Backpropagation Neural Network algorithm.
2. Understood the accuracy and effectiveness of the Feed Forward Backpropagation Neural Network algorithm on test data.

Course Outcomes:

1. We have learned about Feed Forward Backpropagation Neural Network algorithm.
2. We have analyzed the effectiveness of this algorithm.

Conclusion:

We have successfully implemented the Feed Forward Backpropagation Neural Network algorithm.

Viva Question:

1. What is Artificial Neural Network (ANN)?
2. What are the types of layers in ANN?
3. Name the 3 types of layers in ANN?
4. What is activation function in ANN?

For Faculty Use

Correction Parameters	Formative Assessment [ ]	Timely completion practical [ ]	Attendance Learning Attitude[ ]

Theory-5
Support Vector Machines (SVM)

- A support vector machine (SVM) is a type of supervised learning algorithm used in machine learning to solve classification and regression tasks; SVMs are particularly good at solving binary classification problems, which require classifying the elements of a data set into two groups.
- E.g. Is this picture of a dog or cat?
- E.g. Is the stock going up or down?
- SVM is the simplest and most elegant method for classification.
- Each object we want to classify is represented as a point in an n-dimensional space and the co-ordinates of these points are usually called features.
- SVM performs the classification by drawing a hyperplane i.e.a line in 2 or a plane in 3D in such a way that all the points of one category are on one side of the hyperplane and all points of the other category are on the other side of the hyperplane.
- There could be multiple hyperplanes in SVM, but SVM tries to find the best one i.e. that best separates the 2 categories in a sense that it maximizes the distance to points of both categories.
- This distance from the hyperplane to the points of either category is called the margin.
- The points that falls exactly on the margin are called supporting vectors.
- To find this hyperplane in the first place, SVM requires a training data set or set of points that are already labeled with the correct categories. This is why SVM is said to be a supervised learning algorithm.
- In the background, SVM solves a convex optimization problem that maximizes the margin and where the constraints says that points of each category should be on the correct side of the hyperplane. This operation is abstracted in python.
- Advantages of SVM:
- Easy to understand, implement, use, and interpret.
- Effective when the size of training data is small.
- Disadvantages of SVM:
- Simplicity of SVM could be a problem where points cannot be separated by hyperplane. A common workaround in this case is to augment data with some non-linear features that are computed from existing ones, then find the separating hyperplane in higher dimensional space, then project back to the original space. Kernel trick allows us to perform all these steps in a very efficient manner.
- Uses of SVM:
- Face detection
- Spam email detection
- Text recognition
- Simple Implementation of SVM to predict if a student's result is pass or fail based on their pointers:

```
from sklearn import svm

# if X > 5 then P else F

X = [ [5], [6], [2], [7], [9]] # features (pointers/gpa)

y = ['F', 'P', 'F', 'P', 'P'] # lables (result)

clf = svm.SVC(kernel='linear').fit(X,y) # fit it in SVM

clf.predict([ [4] ]) # output -> F

clf.predict([ [8] ]) # output -> P
```

Practical 5: Support Vector machine

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.

Aim-1: Implement the SVM algorithm for binary classification.

Source code:

```
from sklearn import svm
# if X > 5 then P else F
X = [ [5], [6], [2], [7], [9]] # features (pointers/gpa)
y = ['F', 'P', 'F', 'P', 'P'] # lables (result)
clf = svm.SVC(kernel='linear').fit(X,y) # fit it in SVM
clf.predict([ [4] ]) # output → F
clf.predict([ [8] ]) # output → P
```

Output:

```
array(['F'], dtype='<U1')
array(['P'], dtype='<U1')
```

Aim-2: Train an SVM model using a given dataset and optimize its parameters.

Source code:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

data = sns.load_dataset('iris')

# independent var
''' i.e. X = data['sepal_length'
                 'sepal_width',
                 'petal_length',
                 'petal-width' ]
'''
X = data.iloc[:, :-1]
# dependent var i.e. y = data['species']
y = data.iloc[:, -1]

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                        test_size=0.2,
                                        random_state=99)

from sklearn.svm import SVC
model = SVC(kernel='linear', C=1.0, random_state=5)
model.fit(X_train, y_train)

pred = model.predict(X_test)
pred
```

Output:

```
array(['virginica', 'setosa', 'versicolor', 'virginica', 'setosa',
       'versicolor', 'versicolor', 'virginica', 'setosa', 'virginica',
       'virginica', 'versicolor', 'virginica', 'versicolor', 'virginica',
       'setosa', 'setosa', 'versicolor', 'versicolor', 'virginica',
       'versicolor', 'versicolor', 'setosa', 'versicolor', 'virginica',
       'setosa', 'setosa', 'virginica', 'versicolor', 'virginica'],
      dtype=object)
```

Aim-3: Evaluate the performance of the SVM model on test data and analyze the results.

Source code:

```
from sklearn.metrics import classification_report
print(classification_report(y_test, pred))

from sklearn.metrics import accuracy_score
print(accuracy_score(y_test, pred))
```

Output:

```
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00         8
  versicolor       1.00      0.92      0.96        12
   virginica       0.91      1.00      0.95        10

    accuracy                           0.97        30
   macro avg       0.97      0.97      0.97        30
weighted avg       0.97      0.97      0.97        30

0.9666666666666667
```

Result and Discussion: We have successfully implemented SVM algorithm for binary classification and analyzed its performance.

Learning Outcomes:

1. Successfully implemented SVM algorithm.
2. Understood the accuracy and effectiveness of the SVM algorithm on test data.

Course Outcomes:

1. We have learned about SVM algorithm.
2. We have analyzed the effectiveness of this algorithm.

Conclusion:

We have successfully implemented the Feed Forward Backpropagation Neural Network algorithm.

Viva Question:

1. What is supervised learning?
2. What is binary classification?
3. What is SVM?
4. What is a hyperplane in SVM?

For Faculty Use

Correction Parameters	Formative Assessment [ ]	Timely completion practical [ ]	Attendance Learning Attitude[ ]

Theory-6
AdaBoost Ensemble Learning

- AdaBoost is an acronym for Adaptive boost
- Adaboost is a boosting method
- It uses the complete dataset to train the weak learners
- Weak learners are decision trees with max depth of 1, and because they have max depth of 1 they are only trained based on 1 feature of the dataset and thus they have low accuracy.
- Decision tree is a supervised machine learning algorithm and it is also a binary classification method in which the dataset is divided into half based on certain rules it infers from the features of the dataset.
- AdaBoost is a sequential process where each subsequent model tries to correct the errors of the previous model.
- The succeeding models are dependent on the previous model.
- Each model is trained on the same dataset but each sample have different weights based on the previous model's success and error.
- The weights are re-assigned in each iteration to build a strong classifier that learns from the mistakes of the previous weak learners in the ensemble.
- The weights for mispredicted samples are increased so that the weak learner stresses on their prediction.

working of adaboost:
- first a subset is selected from the original dataset.
- initially, all points/ training examples are given the same weights.
- a base model is trained on this subset.
- this model is used to make prediction on all dataset
- errors are calculated using actual values and predicted values
- for the next round/stage, th misclassified examples are assigned higher weights
- furthermore, the correctly classified examples are assigned lower weights.
- in the next round this new dataset is given to another weak learner model to make prediction on this dataset.
- this model then focuses more on the training examples that have the largest weights.
- that means this model tries to correct the errors from the previous model.
- Similarly, multiple models are built and each model corrects the errors of the previous model.
- the final model is the weighted mean of all the models.

so, the adaboost algorithm combine a number of weak learners

Thus, each model boosts the performance of the ensemble.

Adaboost is a very powerful ensemble method in which each subsequent model tries to correct the errors of the previous model. We can use AdaBoost to decrease bias. But, AdaBoost is ineffective in reducing model variance. The main aim of the AdaBoost method is to improve prediction accuracy by combining multiple weak classifiers ie.weak learners into a strong classifier.

Practical-6: Adaboost Ensemble Learning

Aim:

1) Implement the Adaboost algorithm to create an ensemble of weak classifiers.
2) Train the ensemble model on a given dataset and evaluate its performance.
3) Compare the results with individual weak classifiers.

Aim-1: Implement the Adaboost algorithm to create an ensemble of weak classifiers.

Source code:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

df = sns.load_dataset('iris')
df.head()

X = df.iloc[:, :-1]
y = df.iloc[:, -1]

X = X.values
y = y.values

from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y = le.fit_transform(y)
y # setosa -> 0, versicolor -> 1, virginica -> 2

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    stratify=y,
                                                    test_size=0.2,
                                                    random_state=9)

from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(max_depth=1, random_state=42)
tree
```

Output:

```
DecisionTreeClassifier
DecisionTreeClassifier(max_depth=1, random_state=42)
```

Aim-2: Train the ensemble model on a given dataset and evaluate its performance.
Source code (continued):

```
tree = DecisionTreeClassifier(max_depth=1, random_state=42)
tree = tree.fit(X_train, y_train)
y_train_pred = tree.predict(X_train)
y_test_pred = tree.predict(X_test)

from sklearn.metrics import accuracy_score
tree_train = accuracy_score(y_train, y_train_pred)
tree_test = accuracy_score(y_test, y_test_pred)
```

Output:

```
Decision Tree accuracy
Train: 0.6666666666666666
Test: 0.6666666666666666
```

Aim-3: Compare the results with individual weak classifiers.
Source code:

```
from sklearn.ensemble import AdaBoostClassifier
ada = AdaBoostClassifier(n_estimators=100,
                         learning_rate=0.5,
                         random_state=42,
                         algorithm='SAMME')
ada = ada.fit(X_train, y_train)
y_train_pred = ada.predict(X_train)
y_test_pred = ada.predict(X_test)
ada_train = accuracy_score(y_train, y_train_pred)
ada_test = accuracy_score(y_test, y_test_pred)
print(f'Decision Tree accuracy\nTrain: {tree_train}\nTest: {tree_test}\n')
print(f'Adaboost accuracy\nTrain: {ada_train}\nTest: {ada_test}\n')


percentage_increase = ((ada_train - tree_train) / tree_train) * 100
print(f'Percentage Increase in Training Accuracy: {percentage_increase:.2f}%')
percentage_increase = ((ada_test - tree_test) / tree_test) * 100
print(f'Percentage Increase in testing Accuracy: {percentage_increase:.2f}%')
```

Output:

```
Decision Tree accuracy
Train: 0.6666666666666666
Test: 0.6666666666666666

Adaboost accuracy
Train: 1.0
Test: 0.9666666666666667

Percentage Increase in Training Accuracy: 50.00%
Percentage Increase in testing Accuracy: 45.00%
```

Result and Discussion:

We have successfully implemented Adaboost algorithm to create an ensemble of weak classifiers and analyzed its performance.

Learning Outcomes:

1. Successfully implemented Adaboost algorithm.
2. Understood the accuracy and effectiveness of the Adaboost Algorithm on test data.

Course Outcomes:

1. We have learned about Adaboost algorithm.
2. We have analyzed the effectiveness of this algorithm.

Conclusion:

We have successfully implemented the AdaBoost algorithm.

Viva Question:

1. What is the full-form of AdaBoost?
2. What are weak learners in AdaBoost?
3. What are decision trees?
4. What is the main aim of the AdaBoost method?

For Faculty Use

Correction Parameters	Formative Assessment [ ]	Timely completion practical [ ]	Attendance Learning Attitude[ ]

Theory-7
Naive Bayes' Classifier

- Naïve Bayes algorithm is a supervised learning algorithm, which is based on Bayes theorem and used for solving classification problems.
- It is mainly used in text classification that includes a high-dimensional training dataset.
- Naïve Bayes Classifier is one of the simple and most effective Classification algorithms which helps in building the fast machine learning models that can make quick predictions.
- It is a probabilistic classifier, which means it predicts on the basis of the probability of an object.
- Some popular examples of Naïve Bayes Algorithm are spam filtration, Sentimental analysis, and classifying articles.
- It is used to predict from which class an event belongs to.
- Naive in Naive Bayes means that it assumes that all the variables or events are independent of each other i.e. they all are independent variables that means that the outcome of one variable does not affect the outcome of another one.
- The two events are independent if and only if $P(A \cap B) = P(A) * P(B)$, if this relationship exists then we say that those two events are independent of each other.
- Bayes theorem provides a way to calculate the conditional probability of an event with reversed conditional probability and some other values.
- Conditional Probability is the measure of the probability of an event that will occur given that another event has already occurred.
- Conditional Probability formula: $P(A|B) = P(A \cap B) / P(B)$
- Bayes Theorem:

From conditional probability:
P(A|B) = P(A ∩ B) / P(B) ..(1)
P(B|A) = P(B ∩ A) / P(A) ..(2)

This implies:
P(A ∩ B) = P(A|B) * P(B) …(3) … from (1)
P(B ∩ A) = P(B|A) * P(A) …(4) … from (2)

Since P(A ∩ B) = P(B ∩ A) we get:

P(A ∩ B) = P(B ∩ A)
P(A|B) * P(B) = P(B|A) * P(A) …from (3) and (4)

Now dividing both sides by P(B) we get:

P(A|B) * P(B) / P(B) = P(B|A) * P(A) / P(B) …from (3) and (4)

Canceling P(B) in denominator and numerator we get:
P(A|B) = P(B|A) * P(A) / P(B)

This equation is the bayes equation

where ,
A,B = events
P(A|B) = probability of A given B is true
P(B|A) = probability of B given A is true (also called as reversed conditional probability)
P(A), P(B) = independent probabilities of A and B

Thus, Bayes' theorem gives a mathematical rule for inverting conditional probabilities, allowing us to find the probability of a cause given its effect.

Practical-7: Naive Bayes' Classifier

Aim:

1) Implement the Naive Bayes' algorithm for classification.
2) Train a Naive Bayes' model using a given dataset and calculate class probabilities.
3) Evaluate the accuracy of the model on test data and analyze the results.

Aim-1: Implement the Naive Bayes' algorithm for classification.

Source 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

iris = load_iris()

X_train, X_test, y_train, y_test = train_test_split(iris.data,
                                                    iris.target,
                                                    test_size=0.5,
                                                    random_state=42)


gnb = GaussianNB()
gnb
```

Output:

```
GaussianNB()
```

Aim-2: Train a Naive Bayes' model using a given dataset and calculate class probabilities.

Source code:

```
X_train, X_test, y_train, y_test = train_test_split(iris.data,
                                                    iris.target,
                                                    test_size=0.5,
                                                    random_state=42)


gnb.fit(X_train, y_train)
```

Aim-3: Evaluate the accuracy of the model on test data and analyze the results.

Source code:

```
y_pred = gnb.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

Output:

```
Accuracy: 0.9666666666666667
```

Result and Discussion:

We have successfully implemented Naive Bayes algorithm and analyzed its performance.

Learning Outcomes:

1. Successfully implemented Naive Bayes algorithm.
2. Understood the accuracy and effectiveness of the Naive Bayes Algorithm on test data.

Course Outcomes:

1. We have learned about Naive Bayes algorithm.
2. We have analyzed the effectiveness of this algorithm.

Conclusion:

We have successfully implemented the Naive Bayes algorithm.

Viva Question:

1. What is Naive Bayes Algorithm?
2. Naive Bayes is based on which theorem?
3. What is conditional probability?
4. What are independent variables?

For Faculty Use

Correction Parameters	Formative Assessment [ ]	Timely completion practical [ ]	Attendance Learning Attitude[ ]

Theory-8

K-Nearest Neighbors (K-NN)

K-Nearest Neighbors is a simple and elegant supervised learning algorithm
It is used for both classification and regression problems
Supervised learning means that the KNN model is given some labeled input and output data and based on it, it is trained so that it can predict the output of test input data.

Working of KNN:
Step-1: Initialization: In this step all the points of each classes are stored in KNN ML model using a an n-dimensional co-ordinate system where n is the number of input features and when an query point is given it is also stored in this same co-ordinate system
Step-2: Calculate Distance: In this step, the distance from the query point to all the other points is calculated
Step-3: Sort the distance: In this step, the distances calculated are sorted in ascending order in orer to get the nearest neighbor from the query point
Step-4: Select K-value: In this step, the K-value is selected, and this K-value is used to determine the number of nearest neighbors to consider
Step-5: Apply Majority rule: In this step, the selected points are counted to find the most common one i.e. the points with the majority class, and then that majority class is assigned to the query point.

For regression problem the step 5 is different, we simply return the average of n-nearest neighbors
Extra: regression problem means the outcome of that problem is continuous i.e.1,22,4,544,99…etc
And classification problem means that the outcome of that problem has only 2 values i.e. binary for e.g. true or false, yes or no, 0 or 1, high or low, hot or cold, strong or weak , etc.

The k-value is important value and it controls the balance between underfitting and overfitting
A small K-value leads to low bias but high variance
A high K-value leads to high bias but low variance
The best K-value controls the balances between underfitting and overfitting

Advantages:
KNN can be applied to any number of features
KNN is based on a simple principle which says that we are average of n-nearest neighbors
KNN is simple to understand, implement and interpret

Disadvantages:
KNN suffers from dimensional curse
KNN is not suitable for large dataset
KNN is a lazy learning algorithm
KNN takes more time in prediction step when there is more dataset

Practical-8: 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.

Aim1: Implement the K-NN algorithm for classification or regression.

Source code:

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=3)
knn
```

Output:

```
KNeighborsClassifier
KNeighborsClassifier(n_neighbors=3)
```

Aim-2: Apply the K-NN algorithm to a given dataset and predict the class or value for test data.

Source code:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

iris = load_iris()

X_train, X_test, y_train, y_test = train_test_split(iris.data,
                                                    iris.target,
                                                    test_size=0.5,
                                                    random_state=55)
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

y_pred = knn.predict(X_test)
y_pred # 0 → setosa, 1→ versicolor, 2→ virginica
```

Output:

```
array([0, 0, 0, 2, 2, 0, 2, 2, 0, 0, 0, 1, 2, 0, 2, 1, 1, 0, 1, 2, 1, 2,
       1, 2, 1, 1, 2, 1, 2, 1, 0, 0, 2, 2, 0, 1, 1, 0, 2, 1, 2, 0, 1, 0,
       1, 2, 2, 2, 0, 2, 2, 0, 1, 2, 1, 0, 0, 1, 0, 0, 1, 2, 0, 2, 1, 2,
       0, 2, 0, 0, 0, 0, 2, 1, 0])
```

Aim-3: Evaluate the accuracy or error of the predictions and analyze the results.

Source code:

```
mismatched = (y_pred != y_test).sum()
matched = (y_pred == y_test).sum()
print("Mismatched:", mismatched)
print("Matched:", matched)

accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

Output:

```
Mismatched: 1
Matched: 74
Accuracy: 0.9866666666666667
```

Result and Discussion:

We have successfully implemented KNN algorithm and analyzed its performance.

Learning Outcomes:

1. Successfully implemented KNN algorithm.
2. Understood the accuracy and effectiveness of the KNN Algorithm on test data.

Course Outcomes:

1. We have learned about KNN algorithm.
2. We have analyzed the effectiveness of this algorithm.

Conclusion:

We have successfully implemented the KNN algorithm.

Viva Question:

1. What is KNN algorithm?
2. What is supervised learning?
3. What are the steps performed in KNN algorithm?
4. Why k-value is important in KNN?

For Faculty Use

Correction Parameters	Formative Assessment [ ]	Timely completion practical [ ]	Attendance Learning Attitude[ ]

Theory-9
Association Rule Mining

Association rule mining or learning (ARM) is an unsupervised learning algorithm i.e. the model is trained using unlabeled data
ARM checks for dependency of one data item on another data item and maps accordingly to store their relationship
ARM derives IF THEN relationship between the data items, for e.g:
IF an item A is brought THEN there are chances that item B will also be brought
IF is also known as antecedent and THEN is also know as consequent
ARM is used in market basket analysis, medical diagnosis analysis, and protein sequence analysis
ARM has association metrics to measure the association, they are: Support, Confidence, Lift
Support: It is the ratio of frequency of data item in the data set and total number of transaction / records
Confidence: It is the ratio of frequency of items A and B together and the frequency of item A
Lift: It the ratio of support of item A and B and support of A * support of B
If Lift = 1, then items are independent
If Lift > 1, then items are dependent
If Lift < 1, then items are substitutable

ARM can be used with Apriori algorithm to create the rules in an efficient manner and to reduce the number of rules.
Apriori Algorithm uses frequent item sets to generate association rules
It is based on the concept that a subset of a frequency item set must also be a frequency item set.
Frequency set is an item set whose support value is greater than the threshold value

Practical-9: 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

Aim-1: Implement the Association Rule Mining algorithm (e.g., Apriori) to find frequent itemsets.

Source code:

```
import pandas as pd
from mlxtend.frequent_patterns import apriori, association_rules

# transaction data
data = {
    'Transaction': [1, 1, 1,
                    2, 2,
                    3, 3, 3,
                    4],
    'Item': ['Milk', 'Bread', 'Butter',
             'Milk', 'Bread',
             'Bread', 'Butter', 'Diaper',
             'Milk',]
}

# Create DataFrame
df = pd.DataFrame(data)
print('df: ',df)
# Create a basket matrix
basket = df.groupby(['Transaction', 'Item'])['Item'].count().unstack().fillna(0)
basket = basket.applymap(lambda x: 1 if x > 0 else 0)

print("Basket Matrix:")
print(basket)

# Aim 1: Implement Apriori to find frequent itemsets
frequent_itemsets = apriori(basket, min_support=0.5, use_colnames=True)
print("\nFrequent Itemsets:")
print(frequent_itemsets)
```

Output:

```
df:
     Transaction    Item
0            1        Milk
1            1       Bread
2            1      Butter
3            2        Milk
4            2       Bread
5            3       Bread
6            3      Butter
7            3      Diaper
8            4        Milk

Basket Matrix:
Item         Bread  Butter  Diaper  Milk
Transaction
1                1       1       0     1
2                1       0       0     1
3                1       1       1     0
4                0       0       0     1

Frequent Itemsets:
   support         itemsets
0     0.75          (Bread)
1     0.50         (Butter)
2     0.75           (Milk)
3     0.50  (Butter, Bread)
4     0.50    (Milk, Bread)
```

Aim-2: Generate association rules from the frequent itemsets and calculate their support and confidence.

Source code:

```
# Aim 2: Generate Association Rules
rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=0.5)
print("\nAssociation Rules:")
print(rules[['antecedents', 'consequents', 'support', 'confidence']])
```

Output:

```
Association Rules:
  antecedents consequents  support  confidence
0    (Butter)     (Bread)      0.5    1.000000
1     (Bread)    (Butter)      0.5    0.666667
2      (Milk)     (Bread)      0.5    0.666667
3     (Bread)      (Milk)      0.5    0.666667
```

Aim-3: Interpret and analyze the discovered association rules

Source code:

```
# Aim 3: Interpret and Analyze the Discovered Association Rules
for index, row in rules.iterrows():
    print(f"\nRule: {set(row['antecedents'])} → {set(row['consequents'])}")
    print(f"Support: {row['support']:.2f}, Confidence: {row['confidence']:.2f}")
```

Output:

```
Rule: {'Butter'} → {'Bread'}
Support: 0.50, Confidence: 1.00

Rule: {'Bread'} → {'Butter'}
Support: 0.50, Confidence: 0.67

Rule: {'Milk'} → {'Bread'}
Support: 0.50, Confidence: 0.67

Rule: {'Bread'} → {'Milk'}
Support: 0.50, Confidence: 0.67
```

Result and Discussion:

We have successfully implemented Association Rule Mining algorithm and analyzed its performance.

Learning Outcomes:

1. Successfully implemented Association Rule Mining algorithm.
2. Understood the accuracy and effectiveness of the Association Rule Mining Algorithm on test data.

Course Outcomes:

1. We have learned about Association Rule Mining algorithm.
2. We have analyzed the effectiveness of this algorithm.

Conclusion:

We have successfully implemented the KNN algorithm.

Viva Question:

1. What is Association rule mining or learning?
2. What is frequency set in Association rule mining?
3. What is support in Association rule mining?
4. What is apriori algorithm?

For Faculty Use

Correction Parameters	Formative Assessment [ ]	Timely completion practical [ ]	Attendance Learning Attitude[ ]

Theory-10

TensorFlow

TensorFlow is a python framework for building machine learning or deep learning models

It is developed by google and it is an open-source framework

It helps us to train and execute Neural networking recognition, Natural language processing, Digital classification and much more

It also reduces the complexity of implementing computations on large numerical datasets by providing us a a feature known as Computational graph

TensorFlow means Flow of tensors, where tensors means n-dimensional array

Computational Graph is the core of the TensorFlow, it the the path on which the tensors flows and it is also the reason why tensorflow is fast and efficient

Computational graph consist of set of nodes in a well-defined order, where we can specify the methods for computations

In Computational graph the tensors flows from one node to another and between the edges the tensors undergoes some operation

In short, in computational graph:

node = tensors variable

edges = mathematical operations

Practical-10: 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.

Aim-1: Explore and experiment with OpenAI or TensorFlow tools and libraries.

Source code:

```
import tensorflow as tf
print("TensorFlow version: ", tf.__version__)

# Hello world in TF
with tf.compat.v1.Session() as sess:
  hello = tf.constant("Hello, TensorFlow!")
  print(sess.run(hello))
sess.close()

# simple addition opeation
a = tf.constant(10)
b = tf.constant(32)
print(a+b)


# getting hands on Computational graph
with tf.compat.v1.Session() as sess:
  a = tf.constant([12,3], name = 'input_a') # Define a tensor
  b = tf.reduce_prod(a, name = 'prod_b') # Define a prod node
  c = tf.reduce_sum(a, name = 'sum_c') # Define a sum node
  final = tf.add(b,c, name = 'add_d') # Returns x+y element-wise

  # runing the graph
  sess = tf.compat.v1.Session()
  output = sess.run(final)

print(output)

# close the session
sess.close()
```

Output:

```
TensorFlow version:  2.17.0

b'Hello, TensorFlow!'

tf.Tensor(42, shape=(), dtype=int32)

51
```

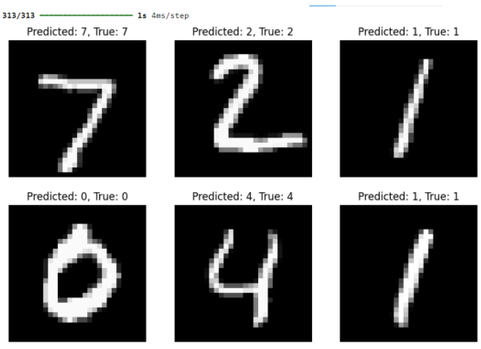
Aim-2: Perform a demonstration or mini-project showcasing the capabilities of the tools.

Source code:

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Normalize the images to values between 0 and 1
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

print(f'Training data shape: {x_train.shape}, Labels shape: {y_train.shape}')
print(f'Test data shape: {x_test.shape}, Labels shape: {y_test.shape}')

# Build the model
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),  # Flatten the input images
     keras.layers.Dense(128, activation='relu'),   # Hidden layer with ReLU activation
     keras.layers.Dropout(0.2),                    # Dropout layer to prevent overfitting
     keras.layers.Dense(10, activation='softmax')   # Output layer with softmax activation for classification
 ])

 # Compile the model
 model.compile(optimizer='adam',
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])
  # Train the model
 model.fit(x_train, y_train, epochs=5)

 # Evaluate the model on test data
 test_loss, test_acc = model.evaluate(x_test, y_test)
 print(f'\nTest accuracy: {test_acc:.4f}')

 # Make predictions on test data
 predictions = model.predict(x_test)

 # Display some predictions along with their corresponding images and true labels
 def display_predictions(images, predictions, true_labels):
     plt.figure(figsize=(10, 10))
     for i in range(9):
         plt.subplot(3, 3, i + 1)
         plt.imshow(images[i], cmap='gray')
         plt.title(f'Predicted: {np.argmax(predictions[i])}, True: {true_labels[i]}')
         plt.axis('off')
     plt.show()

 display_predictions(x_test[:9], predictions[:9], y_test[:9])
```

Output:

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step
```



Aim-3: Discuss and present the findings and potential applications.

Findings: The model achieves a high accuracy rate on the MNIST dataset after just a few epochs of training. The use of dropout layers helps in preventing overfitting.

Potential Applications:

Image Recognition: This approach can be extended to recognize various objects in images.
Medical Imaging: TensorFlow can be used to classify medical images for diagnostics.
Real-Time Object Detection: With further enhancements, models can be deployed in real-time applications such as autonomous vehicles or surveillance systems.

Result and Discussion:

We have successfully demonstrated an experiment with TensorFlow tools and libraries.

Learning Outcomes:

1. Successfully implemented the mini-project with TensorFlow tools and libraries.
2. Understood the TensorFlow tools and libraries.

Course Outcomes:

1. We have learned about TensorFlow tools and its libraries.
2. We have experimented with the TensorFlow tools and its libraries.

Conclusion:

We have successfully implemented the mini-project with TensorFlow tools and libraries and learned about TensorFlow.

Viva Question:

1. What is TensorFlow?
2. Who developed TensorFlow?
3. What are tensors in TensorFlow?
4. What is keras?

For Faculty Use

Correction Parameters	Formative Assessment [ ]	Timely completion practical [ ]	Attendance Learning Attitude[ ]