1. Implement decision tree learning algorithm for the restaurant waiting problem

```
import math
from collections import Counter

# Calculate entropy of the dataset
def entropy(data):
    labels = [row[-1] for row in data]
    label_counts = Counter(labels)
    total = len(data)
    entropy_val = 0.0

    for count in label_counts.values():
        p = count / total
        entropy_val -= p * math.log2(p)

    return entropy_val

# Calculate information gain for an attribute
def information_gain(data, attr_index):
    total_entropy = entropy(data)
    values = set([row[attr_index] for row in data])

    weighted_entropy = 0.0
    total = len(data)

    for value in values:
        subset = [row for row in data if row[attr_index] == value]
        weighted_entropy += (len(subset) / total) * entropy(subset)

    return total_entropy - weighted_entropy

# Recursively build the decision tree
def build_tree(data, attributes):
    # Base case: if all examples have the same label, return the label
    labels = [row[-1] for row in data]
    if len(set(labels)) == 1:
        return labels[0]

    # Base case: if no more attributes to split on, return the majority label
    if len(attributes) == 0:
        return Counter(labels).most_common(1)[0][0]

    # Find the best attribute to split on
    gains = [information_gain(data, i) for i in range(len(attributes))]
    best_attr_index = gains.index(max(gains))
    best_attr = attributes[best_attr_index]

    # Create a node and split the dataset
    tree = {best_attr: {}}
    values = set([row[best_attr_index] for row in data])

    for value in values:
        subset = [row for row in data if row[best_attr_index] == value]
        subtree = build_tree(subset, attributes[:best_attr_index] + attributes[best_attr_index+1:])
        tree[best_attr][value] = subtree

    return tree

# Example dataset (Restaurant waiting problem)
data = [
    ['No', 'No', 'No', 'Yes', 'Some', '$$', 'No', 'No', 'Thai', '0-10', 'Yes'],
    ['No', 'No', 'No', 'Yes', 'Full', '$', 'No', 'No', 'Burger', '0-10', 'No'],
    ['Yes', 'No', 'No', 'No', 'Some', '$', 'No', 'No', 'Thai', '0-10', 'Yes'],
    # ... (Add more examples here)
]

attributes = ['Alternate', 'Bar', 'Fri/Sat', 'Hungry', 'Patrons', 'Price', 'Raining', 'Reservation', 'Type', 'Wait Estimate']

# Build and print the decision tree
tree = build_tree(data, attributes)
print(tree)
```

2. Implement recursive best first search algorithm for Romanian map problem or any other map with 4 or more cities.

pip install pandas

```
import pandas as pd  # Importing pandas for data handling

# Load the dataset from the specified path
data = pd.read_csv('/content/Ecom_Cust_Survey.csv')

# Display the first few rows of the dataset (optional)
print(data.head())

class Graph:
    def __init__(self):
        self.edges = {}

    def add_edge(self, u, v, cost):
        self.edges.setdefault(u, {})[v] = cost
        self.edges.setdefault(v, {})[u] = cost  # Undirected graph

    def get_neighbors(self, node):
        return self.edges.get(node, {})

def heuristic(node, goal):
    # Simplified heuristic based on straight-line distance (for example purposes)
    heuristic_values = {
        'Bucharest': 0,
        'Arad': 366,
        'Sibiu': 253,
        'Timisoara': 329,
        'Fagaras': 178,
        'Craiova': 160,
        'Pitesti': 100,
        'Neamt': 234,
        'Iasi': 226
    }
    return heuristic_values.get(node, float('inf'))

def recursive_best_first_search(graph, node, goal, g, f_limit):
    f = g + heuristic(node, goal)

    if f > f_limit:
        return f, None  # Current path is not feasible

    if node == goal:
        return f, [node]  # Goal found

    best_f = float('inf')
    best_path = None

    for neighbor, cost in graph.get_neighbors(node).items():
        if g + cost < f_limit:  # Only consider neighbors that can lead to a solution
            g_new = g + cost
            result_f, result_path = recursive_best_first_search(graph, neighbor, goal, g_new, f_limit)
            if result_path is not None:
                return result_f, [node] + result_path

            if result_f < best_f:
                best_f = result_f
                best_path = result_path

    return best_f, best_path

def rbfs(graph, start, goal):
    f_limit = heuristic(start, goal)
    return recursive_best_first_search(graph, start, goal, 0, f_limit)

# Create the Romanian map graph
romanian_map = Graph()
romanian_map.add_edge('Bucharest', 'Arad', 140)
romanian_map.add_edge('Bucharest', 'Pitesti', 100)
romanian_map.add_edge('Bucharest', 'Fagaras', 211)
romanian_map.add_edge('Arad', 'Sibiu', 140)
romanian_map.add_edge('Sibiu', 'Fagaras', 99)
romanian_map.add_edge('Pitesti', 'Craiova', 138)
romanian_map.add_edge('Timisoara', 'Arad', 118)
romanian_map.add_edge('Craiova', 'Pitesti', 138)
```

```
romanian_map.add_edge('Timisoara', 'Sibiu', 151)
romanian_map.add_edge('Neamt', 'Iasi', 92)

# Example usage
start_city = 'Arad'
goal_city = 'Bucharest'
result = rbfs(romanian_map, start_city, goal_city)

# Print the result
if result[1] is not None:
    print("Path found:", " -> ".join(result[1]))
else:
    print("No path found from", start_city, "to", goal_city)
```

3. Implement Iterative deep depth first search for Romanian map problem or any other map with 4 or more cities.

```
import pandas as pd  # Import pandas for data handling

# Load the dataset from the specified path
data = pd.read_csv('/content/Ecom_Cust_Survey.csv')

# Display the first few rows of the dataset (optional)
print(data.head())

class Graph:
    def __init__(self):
        self.edges = {}

    def add_edge(self, u, v):
        self.edges.setdefault(u, []).append(v)
        self.edges.setdefault(v, []).append(u)  # Undirected graph

    def get_neighbors(self, node):
        return self.edges.get(node, [])

def depth_limited_search(graph, node, goal, depth):
    if depth == 0 and node == goal:
        return [node]
    elif depth > 0:
        for neighbor in graph.get_neighbors(node):
            path = depth_limited_search(graph, neighbor, goal, depth - 1)
            if path is not None:
                return [node] + path
    return None

def iterative_deepening_dfs(graph, start, goal):
    depth = 0
    while True:
        path = depth_limited_search(graph, start, goal, depth)
        if path is not None:
            return path  # Goal found
        depth += 1  # Increase depth limit

# Create the Romanian map graph
romanian_map = Graph()
romanian_map.add_edge('Bucharest', 'Arad')
romanian_map.add_edge('Bucharest', 'Pitesti')
romanian_map.add_edge('Bucharest', 'Fagaras')
romanian_map.add_edge('Arad', 'Sibiu')
romanian_map.add_edge('Sibiu', 'Fagaras')
romanian_map.add_edge('Pitesti', 'Craiova')
romanian_map.add_edge('Timisoara', 'Arad')
romanian_map.add_edge('Craiova', 'Pitesti')
romanian_map.add_edge('Timisoara', 'Sibiu')
romanian_map.add_edge('Neamt', 'Iasi')

# Example usage
start_city = 'Arad'
goal_city = 'Bucharest'
result = iterative_deepening_dfs(romanian_map, start_city, goal_city)
```

```
# Print the result
if result:
    print("Path found:", " -> ".join(result))
else:
    print("No path found from", start_city, "to", goal_city)
```

4. Implement Breadth first search algorithm for Romanian map problem or any other map with 4 or more cities.

```
import pandas as pd  # Import pandas for data handling

# Load the dataset from the specified path
data = pd.read_csv('/content/Ecom_Cust_Survey.csv')

# Display the first few rows of the dataset (optional)
print(data.head())

class Graph:
    def __init__(self):
        self.edges = {}

    def add_edge(self, u, v):
        self.edges.setdefault(u, []).append(v)
        self.edges.setdefault(v, []).append(u)  # Undirected graph

    def get_neighbors(self, node):
        return self.edges.get(node, [])

def breadth_first_search(graph, start, goal):
    queue = [[start]]  # Initialize queue with the start node
    visited = set()    # Keep track of visited nodes

    while queue:
        path = queue.pop(0)  # Dequeue the first path
        node = path[-1]      # Get the last node from the path

        if node in visited:
            continue         # Skip if the node has already been visited

        visited.add(node)    # Mark the node as visited

        if node == goal:
            return path      # Goal found, return the path

        for neighbor in graph.get_neighbors(node):
            new_path = list(path)  # Create a new path including the neighbor
            new_path.append(neighbor)
            queue.append(new_path)  # Enqueue the new path

    return None  # Return None if no path found

# Create the Romanian map graph
romanian_map = Graph()
romanian_map.add_edge('Bucharest', 'Arad')
romanian_map.add_edge('Bucharest', 'Pitesti')
romanian_map.add_edge('Bucharest', 'Fagaras')
romanian_map.add_edge('Arad', 'Sibiu')
romanian_map.add_edge('Sibiu', 'Fagaras')
romanian_map.add_edge('Pitesti', 'Craiova')
romanian_map.add_edge('Timisoara', 'Arad')
romanian_map.add_edge('Craiova', 'Pitesti')
romanian_map.add_edge('Timisoara', 'Sibiu')
romanian_map.add_edge('Neamt', 'Iasi')

# Example usage
start_city = 'Arad'
goal_city = 'Bucharest'
result = breadth_first_search(romanian_map, start_city, goal_city)

# Print the result
if result:
    print("Path found:", " -> ".join(result))
else:
    print("No path found from", start_city, "to", goal_city)
```

5. Implement A* search algorithm for Romanian map problem or any other map with 4 or more cities.

```
import pandas as pd  # Import pandas for data handling

# Load the dataset from the specified path
data = pd.read_csv('/content/Ecom_Cust_Survey.csv')

# Display the first few rows of the dataset (optional)
print(data.head())

class Graph:
    def __init__(self):
        self.edges = {}

    def add_edge(self, u, v, cost):
        self.edges.setdefault(u, {})[v] = cost
        self.edges.setdefault(v, {})[u] = cost  # Undirected graph

    def get_neighbors(self, node):
        return self.edges.get(node, {})

def heuristic(node, goal):
    # Simplified heuristic based on straight-line distance (for example purposes)
    heuristic_values = {
        'Bucharest': 0,
        'Arad': 366,
        'Sibiu': 253,
        'Timisoara': 329,
        'Fagaras': 178,
        'Craiova': 160,
        'Pitesti': 100,
        'Neamt': 234,
        'Iasi': 226
    }
    return heuristic_values.get(node, float('inf'))

def a_star_search(graph, start, goal):
    open_set = {start}  # Nodes to be evaluated
    came_from = {}      # Track the optimal path

    g_score = {node: float('inf') for node in graph.edges}
    g_score[start] = 0  # Cost from start to start is zero

    f_score = {node: float('inf') for node in graph.edges}
    f_score[start] = heuristic(start, goal)  # Estimated cost from start to goal

    while open_set:
        current = min(open_set, key=lambda node: f_score[node])  # Node in open set with lowest f_score

        if current == goal:
            return reconstruct_path(came_from, current)  # Path found

        open_set.remove(current)
        for neighbor, cost in graph.get_neighbors(current).items():
            tentative_g_score = g_score[current] + cost

            if tentative_g_score < g_score[neighbor]:  # A better path found
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = g_score[neighbor] + heuristic(neighbor, goal)

                if neighbor not in open_set:
                    open_set.add(neighbor)  # Add neighbor to open set

    return None  # Return None if no path found

def reconstruct_path(came_from, current):
    total_path = [current]
    while current in came_from:
        current = came_from[current]
        total_path.append(current)
    return total_path[::-1]  # Return reversed path

# Create the Romanian map graph
romanian_map = Graph()
romanian_map.add_edge('Bucharest', 'Arad', 140)
romanian_map.add_edge('Bucharest', 'Pitesti', 100)
romanian_map.add_edge('Bucharest', 'Fagaras', 211)
```

```
romanian_map.add_edge('Arad', 'Sibiu', 140)
romanian_map.add_edge('Sibiu', 'Fagaras', 99)
romanian_map.add_edge('Pitesti', 'Craiova', 138)
romanian_map.add_edge('Timisoara', 'Arad', 118)
romanian_map.add_edge('Craiova', 'Pitesti', 138)
romanian_map.add_edge('Timisoara', 'Sibiu', 151)
romanian_map.add_edge('Neamt', 'Iasi', 92)

# Example usage
start_city = 'Arad'
goal_city = 'Bucharest'
result = a_star_search(romanian_map, start_city, goal_city)

# Print the result
if result:
    print("Path found:", " -> ".join(result))
else:
    print("No path found from", start_city, "to", goal_city)
```

6. Implement feed forward back propagation neural network learning algorithm for the restaurant waiting problem.

```
import pandas as pd
import numpy as np
```

```
import pandas as pd
import numpy as np

# Load the dataset from the specified path
data = pd.read_csv('/content/Ecom_Cust_Survey.csv')

# Display the first few rows of the dataset (optional)
print(data.head())

# Assume the dataset has features for waiting time and a target column for customer satisfaction
X = data[['waiting_time', 'overall_experience', 'ease_of_use']].values
y = data['Cust_Satisfaction'].values  # Target variable

# Check if the selected columns exist in the DataFrame
selected_columns = ['WaitTime', 'Overall_Experience', 'Easy_to_Use', 'Cust_Satisfaction']
for column in selected_columns:
    if column not in data.columns:
        raise KeyError(f"Column '{column}' not found in the DataFrame. Please check the column names.")

# Normalize the input features for better performance
X = (X - np.mean(X, axis=0)) / np.std(X, axis=0)

# Define the neural network architecture
input_size = X.shape[1]  # Number of input features
hidden_size = 5          # Number of neurons in the hidden layer
output_size = 1          # Output is a single value (e.g., satisfaction score)

# Initialize weights
np.random.seed(0)  # For reproducibility
W1 = np.random.rand(input_size, hidden_size)  # Weights from input to hidden layer
b1 = np.random.rand(hidden_size)              # Bias for hidden layer
W2 = np.random.rand(hidden_size, output_size)  # Weights from hidden to output layer
b2 = np.random.rand(output_size)              # Bias for output layer

# Activation function (sigmoid)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of the sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)

# Training the neural network
learning_rate = 0.01
epochs = 10000

for epoch in range(epochs):
```

```
    # Forward pass
    hidden_layer_input = np.dot(X, W1) + b1
    hidden_layer_output = sigmoid(hidden_layer_input)

    output_layer_input = np.dot(hidden_layer_output, W2) + b2
    predicted_output = sigmoid(output_layer_input)

    # Compute the error
    error = y.reshape(-1, 1) - predicted_output

    # Backpropagation
    d_predicted_output = error * sigmoid_derivative(predicted_output)
    error_hidden_layer = d_predicted_output.dot(W2.T)
    d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)

    # Update weights and biases
    W2 += hidden_layer_output.T.dot(d_predicted_output) * learning_rate
    b2 += np.sum(d_predicted_output, axis=0) * learning_rate
    W1 += X.T.dot(d_hidden_layer) * learning_rate
    b1 += np.sum(d_hidden_layer, axis=0) * learning_rate

    # Print the error every 1000 epochs
    if epoch % 1000 == 0:
        print(f'Epoch {epoch}, Error: {np.mean(np.abs(error))}')

# Testing the trained model using an example from the dataset
test_index = 0  # Example index for testing
test_data = data[['WaitTime', 'Overall_Experience', 'Easy_to_Use']].iloc[test_index].values
test_data = (test_data - np.mean(X, axis=0)) / np.std(X, axis=0)  # Normalize
hidden_layer_input = np.dot(test_data, W1) + b1
hidden_layer_output = sigmoid(hidden_layer_input)
output_layer_input = np.dot(hidden_layer_output, W2) + b2
predicted_test_output = sigmoid(output_layer_input)

print("Predicted Customer Satisfaction for test data:", predicted_test_output[0][0])
```