Roll No:- 15 **Practical No:- 1** Date:-

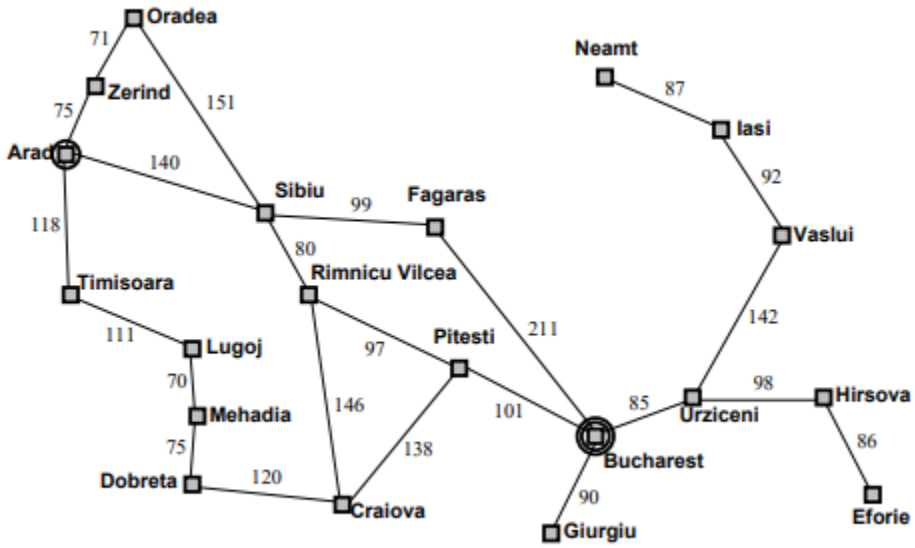
Aim:- Breadth First Search & Iterative Depth First Search :

i) Implement the Breadth First Search algorithm to solve a given problem.
ii) Implement the Iterative Depth First Search algorithm to solve the same problem.
iii) Compare the performance and efficiency of both algorithms.

i) Implement the Breadth First Search algorithm to solve a given problem.

- **Purpose**: Explores all neighbors at the present depth prior to moving on to nodes at the next depth level.
- **Data Structure**: Utilizes a queue to keep track of cities to explore.

We consider the Map of Romania problem and solve it using BFS

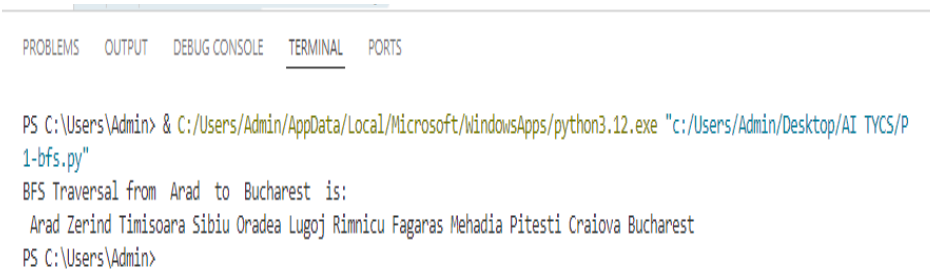


Source code:-

```
import queue as Q
from RMP import dict_gn
start='Arad'
goal='Bucharest'
result=''
def BFS(city, cityq, visitedq):
    global result
if city==start:
    result=result+' '+city
for eachcity in dict_gn[city].keys():
   if eachcity==goal:
      result=result+' '+eachcity
      return
```

```
            if eachcity not in cityq.queue and eachcity not in visitedq.queue:
              cityq.put(eachcity)
              result=result+' '+eachcity
            visitedq.put(city)
            BFS(cityq.get(),cityq,visitedq)
        def main():
        cityq=Q.Queue()
        visitedq=Q.Queue()
        BFS(start, cityq, visitedq)
        print("BFS Traversal from ",start," to ",goal," is: ")
        print(result)
        main()
```

Output:-



ii) Implement the Iterative Depth First Search algorithm to solve the same problem.

- **Purpose**: Combines the depth-first search's space efficiency with breadth-first search's completeness, running depth-first search multiple times with increasing depth limits.
- **Data Structure**: Utilizes a stack (via list) to track cities and their visited states.

Source code:-

```
        import queue as Q
        from RMP import dict_gn

start='Arad'
goal='Bucharest'
result=''

def DLS(city, visitedstack, startlimit, endlimit):
    global result
    found=0
    result=result+city+' '
    visitedstack.append(city)
```

```
    if city==goal:
        return 1
    if startlimit==endlimit:
        return 0
    for eachcity in dict_gn[city].keys():
        if eachcity not in visitedstack:
            found=DLS(eachcity, visitedstack, startlimit+1, endlimit)
            if found:
                return found

def IDDFS(city, visitedstack, endlimit):
    global result
    for i in range(0, endlimit):
        print("Searching at Limit: ",i)
        found=DLS(city, visitedstack, 0, i)
        if found:
            print("Found")
            break
        else:
            print("Not Found! ")
            print(result)
            print("-----")
            result=' '
            visitedstack=[]

def main():
    visitedstack=[]
    IDDFS(start, visitedstack, 9)
    print("IDDFS Traversal from ",start," to ", goal," is: ")
    print(result)
  main()
```

Output:-

```
2-iddfs.py"
Searching at Limit:  0
Not Found!
Arad
-----
Searching at Limit:  1
Not Found!
 Arad Zerind Timisoara Sibiu
-----
Searching at Limit:  2
Not Found!
 Arad Zerind Oradea Timisoara Lugoj Sibiu Rimnicu Fagaras
-----
Searching at Limit:  3
Not Found!
 Arad Zerind Oradea Sibiu Timisoara Lugoj Mehadia
-----
Searching at Limit:  4
Not Found!
 Arad Zerind Oradea Sibiu Rimnicu Fagaras Timisoara Lugoj Mehadia Drobeta
-----
Searching at Limit:  5
Found
IDDFS Traversal from  Arad  to  Bucharest  is:
 Arad Zerind Oradea Sibiu Rimnicu Pitesti Craiova Fagaras Bucharest
PS C:\Users\Admin>
```

iii) Compare the performance and efficiency of both algorithms.

1) **Time Complexity**

- **BFS**:
  - Time complexity: O(V+E) where V is the number of vertices (cities) and E is the number of edges (connections between cities).
  - This is because every vertex and edge is explored once.
- **IDDFS**:
  - Time complexity: O(b^d)) where b is the branching factor (the average number of successors per state) and d is the depth of the solution. This makes it less efficient in the worst case, especially for deep trees.
  - However, for shallow solutions, it can perform well.

2) **Space Complexity**

- **BFS**:
  - Space complexity: O(V) since it stores all the vertices in the queue at the deepest level.
- **IDDFS**:
  - Space complexity: O(d) where d is the maximum depth of the search, as it only stores nodes on the current path in the stack.

3)Conclusion

- **BFS** is more efficient for shallow searches and guarantees the shortest path in unweighted graphs.
- **IDDFS** is preferable for large and infinite-depth trees where space is a concern, but it can be slower for deep searches due to the repeated visits of nodes.

Roll No:- 15 **Practical No:- 2** **Date:-**

Aim:- A* Search and Recursive Best-First Search :

i)Implement the A* Search algorithm for solving a pathfinding problem.

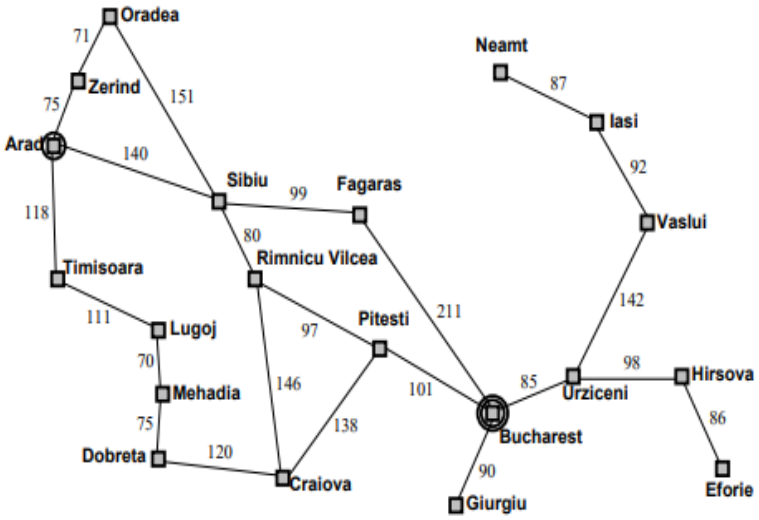
ii)Implement the Recursive Best-First Search algorithm for the same problem .

iii)Compare the performance and effectiveness of both algorithms

i)Implement the A* Search algorithm for solving a pathfinding problem.

The A* Search algorithm is a popular and powerful pathfinding and graph traversal algorithm commonly used in various applications, such as AI for games, robotics, and network routing. It efficiently finds the shortest path from a starting node to a goal node while considering both the cost of reaching the node and an estimate of the cost to reach the goal.

We consider the Map of Romania problem and solve it using A* algorithm

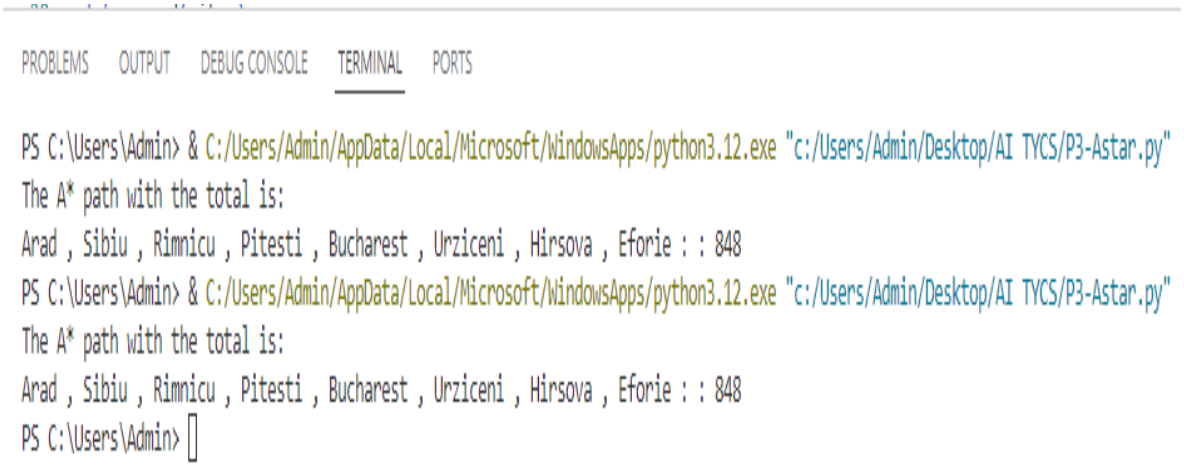


Source code:-

```
import queue as Q
from RMP import dict_gn
from RMP import dict_hn
start='Arad'
goal='Eforie'
result=''
def get_fn(citystr):
    cities=citystr.split(" , ")
    hn=gn=0
    for ctr in range(0, len(cities)-1):
        gn=gn+dict_gn[cities[ctr]][cities[ctr+1]]
    hn=dict_hn[cities[len(cities)-1]]
    return(hn+gn)
```

```
def expand(cityq):
    global result
    tot, citystr, thiscity=cityq.get()
    if thiscity==goal:
        result=citystr+" : : "+str(tot)
        return
    for cty in dict_gn[thiscity]:
        cityq.put((get_fn(citystr+" , "+cty), citystr+" , "+cty,
cty))
    expand(cityq)
def main():
    cityq=Q.PriorityQueue()
    thiscity=start
    cityq.put((get_fn(start),start,thiscity))
    expand(cityq)
    print("The A* path with the total is: ")
    print(result)

main()
```

Output:-



ii)Implement the Recursive Best-First Search algorithm for the same problem .

Recursive Best-First Search (RBFS) is a search algorithm that combines the benefits of depth-first search (DFS) with the heuristic guidance of best-first search. It is particularly useful for solving problems where the search space is large and where memory efficiency is a concern. RBFS is an optimal algorithm in certain conditions and is often applied in domains such as AI, pathfinding, and game playing.

Source code:-

```
import queue as Q
```

```
from RMP import dict_gn
from RMP import dict_hn
start='Arad'
goal='Bucharest'
result=''
def get_fn(citystr):
    cities=citystr.split(',')
    hn=gn=0
    for ctr in range(0,len(cities)-1):
        gn=gn+dict_gn[cities[ctr]][cities[ctr+1]]
    hn=dict_hn[cities[len(cities)-1]]
    return(hn+gn)
def printout(cityq):
    for i in range(0,cityq.qsize()):
        print(cityq.queue[i])

def expand(cityq):
    global result
    tot,citystr,thiscity=cityq.get()
    nexttot=999
    if not cityq.empty():
        nexttot,nextcitystr,nextthiscity=cityq.queue[0]
    if thiscity==goal and tot<nexttot:
        result=citystr+'::'+str(tot)
        return
    print("Expanded city------------------------------",thiscity)
    print("Second best f(n)------------------------------",nexttot)
    tempq=Q.PriorityQueue()
    for cty in dict_gn[thiscity]:
            tempq.put((get_fn(citystr+','+cty),citystr+','+cty,cty))
    for ctr in range(1,3):
        ctrtot,ctrcitystr,ctrthiscity=tempq.get()
        if ctrtot<nexttot:
            cityq.put((ctrtot,ctrcitystr,ctrthiscity))
        else:
            cityq.put((ctrtot,citystr,thiscity))
            break
    printout(cityq)
    expand(cityq)
def main():
    cityq=Q.PriorityQueue()
    thiscity=start
    cityq.put((999,"NA","NA"))
    cityq.put((get_fn(start),start,thiscity))
```

```
            expand(cityq)
            print(result)
        main()
```

Output:-

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Admin\Desktop\AI TYCS> c:; cd 'c:\Users\Admin\Desktop\AI TYCS'; & 'c:\Users\Admin\AppData\Local\Microsoft\WindowsApps\p
ython3.12.exe' 'c:\Users\Admin\.vscode\extensions\ms-python.debugpy-2024.10.0-win32-x64\bundled\libs\debugpy\adapter/../..\debugpy\l
auncher' '26128' '--' 'c:\Users\Admin\Desktop\AI TYCS\P4-rbfs.py'
Expanded city------------------------------ Arad
Second best f(n)------------------------------ 999
(393, 'Arad,Sibiu', 'Sibiu')
(999, 'NA', 'NA')
(447, 'Arad,Timisoara', 'Timisoara')
Expanded city------------------------------ Sibiu
Second best f(n)------------------------------ 447
(413, 'Arad,Sibiu,Rimnicu', 'Rimnicu')
(415, 'Arad,Sibiu,Fagaras', 'Fagaras')
(447, 'Arad,Timisoara', 'Timisoara')
(999, 'NA', 'NA')
Expanded city------------------------------ Rimnicu
Second best f(n)------------------------------ 415
(415, 'Arad,Sibiu,Fagaras', 'Fagaras')
(417, 'Arad,Sibiu,Rimnicu', 'Rimnicu')
(447, 'Arad,Timisoara', 'Timisoara')
(999, 'NA', 'NA')
```

```
(999, 'NA', 'NA')
(447, 'Arad,Timisoara', 'Timisoara')
Expanded city------------------------------ Sibiu
Second best f(n)------------------------------ 447
(413, 'Arad,Sibiu,Rimnicu', 'Rimnicu')
(415, 'Arad,Sibiu,Fagaras', 'Fagaras')
(447, 'Arad,Timisoara', 'Timisoara')
(999, 'NA', 'NA')
Expanded city------------------------------ Rimnicu
Second best f(n)------------------------------ 415
(415, 'Arad,Sibiu,Fagaras', 'Fagaras')
(417, 'Arad,Sibiu,Rimnicu', 'Rimnicu')
(447, 'Arad,Timisoara', 'Timisoara')
(999, 'NA', 'NA')
Expanded city------------------------------ Fagaras
Second best f(n)------------------------------ 417
(417, 'Arad,Sibiu,Rimnicu', 'Rimnicu')
(450, 'Arad,Sibiu,Fagaras', 'Fagaras')
(447, 'Arad,Timisoara', 'Timisoara')
(999, 'NA', 'NA')
Expanded city------------------------------ Rimnicu
Second best f(n)------------------------------ 447
(417, 'Arad,Sibiu,Rimnicu,Pitesti', 'Pitesti')
(447, 'Arad,Timisoara', 'Timisoara')
(999, 'NA', 'NA')
(450, 'Arad,Sibiu,Fagaras', 'Fagaras')
(526, 'Arad,Sibiu,Rimnicu', 'Rimnicu')
Expanded city------------------------------ Pitesti
Second best f(n)------------------------------ 447
(418, 'Arad,Sibiu,Rimnicu,Pitesti,Bucharest', 'Bucharest')
(447, 'Arad,Timisoara', 'Timisoara')
(607, 'Arad,Sibiu,Rimnicu,Pitesti', 'Pitesti')
(526, 'Arad,Sibiu,Rimnicu', 'Rimnicu')
(450, 'Arad,Sibiu,Fagaras', 'Fagaras')
(999, 'NA', 'NA')
Arad,Sibiu,Rimnicu,Pitesti,Bucharest::418
```

iii)Compare the performance and effectiveness of both algorithms

1. **Time Complexity**

- **A***:
    - In the worst case, the time complexity is O(bd)O(b^d)O(bd), where bbb is the branching factor and ddd is the depth of the solution. This is due to the exploration of nodes in a potentially exponential manner.

- **RBFS**:
    - Also has a worst-case time complexity of O(bd)O(b^d)O(bd). However, because it only keeps track of the current path, its exploration can be less comprehensive than A* under certain conditions.

2. Space Complexity

- **A***:
    - Space complexity is O(bd)O(b^d)O(bd) because it stores all generated nodes in the priority queue. This can lead to high memory usage, especially for large search spaces.
- **RBFS**:
    - Space complexity is O(d)O(d)O(d), as it only maintains the current path in memory. This makes it more memory efficient for deep searches.

3. Optimality

- **A***:
    - A* is optimal if the heuristic h(n)h(n)h(n) is admissible (i.e., it never overestimates the true cost to the goal). This guarantees that the shortest path will be found.
- **RBFS**:
    - RBFS is optimal if the heuristic is admissible as well. However, it can suffer from inefficiency in cases where it needs to backtrack frequently due to limited exploration.

In summary, both A* and RBFS have their strengths and weaknesses:

- **A*** is generally more efficient and effective for pathfinding in practice, particularly with a well-chosen heuristic. It is optimal and can handle complex search spaces but at the cost of higher memory usage.
- **RBFS** is more memory-efficient, making it suitable for deep search spaces, but it can be less efficient in finding solutions and may require more backtracking.

Roll No:- 15 **Practical No:- 3** Date:-

Aim:- Decision tree learning

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

Decision tree learning is a popular and intuitive method for both classification and regression tasks in machine learning. It creates a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.

Source code:-

```
import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix, accuracy_score,
classification_report
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, plot_tree
import matplotlib.pyplot as plt

# Function to import dataset
def importdata():
    # Importing dataset from file path
    balance_data =
pd.read_csv(r"C:\Users\Preetika\OneDrive\Desktop\TYsem5\AI
TYCS\P5-balance-scale.data", header=None)

    # Printing dataset information
    print("Dataset Length: ", len(balance_data))
    print("Dataset Head:\n", balance_data.head())

    # Return the dataset
    return balance_data

# Function to split the dataset
def splitdataset(balance_data):
    # Separate the features (X) and target (Y)
    X = balance_data.iloc[:, 1:5].values
    Y = balance_data.iloc[:, 0].values
```

```
    # Split the dataset into training and testing sets (70% train, 30% test)
    X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3,
random_state=100)
    return X_train, X_test, y_train, y_test

# Function to train the decision tree using entropy criterion
def train_using_entropy(X_train, y_train):
    # Initialize Decision Tree with entropy criterion
    clf_entropy = DecisionTreeClassifier(criterion="entropy", random_state=100,
max_depth=3, min_samples_leaf=5)

    # Train the model
    clf_entropy.fit(X_train, y_train)
    return clf_entropy

# Function to make predictions on the test set
def prediction(X_test, clf_object):
    # Predicting the labels for the test set
    y_pred = clf_object.predict(X_test)
    print("Predicted Values:\n", y_pred)
    return y_pred

# Function to calculate and print the accuracy and classification report
def cal_accuracy(y_test, y_pred):
    print("Accuracy: ", accuracy_score(y_test, y_pred) * 100)
    print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
    print("Classification Report:\n", classification_report(y_test, y_pred))

# Function to visualize the decision tree
def visualize_tree(clf_object):
    plt.figure(figsize=(12,8))
    plot_tree(clf_object, filled=True, feature_names=["Feature 1", "Feature 2",
"Feature 3", "Feature 4"], class_names=["L", "B", "R"])
    plt.title("Decision Tree Visualization")
    plt.show()

# Main function
def main():
    # Step 1: Import the dataset
    data = importdata()

    # Step 2: Split the dataset into training and testing sets
```

```
    X_train, X_test, y_train, y_test = splitdataset(data)

    # Step 3: Train the decision tree using entropy criterion
    clf_entropy = train_using_entropy(X_train, y_train)

    # Step 4: Predict the test set results
    print("Results using entropy:")
    y_pred_entropy = prediction(X_test, clf_entropy)

    # Step 5: Calculate accuracy and display performance report
    cal_accuracy(y_test, y_pred_entropy)

    # Step 6: Visualize the decision tree
    visualize_tree(clf_entropy)

# Running the main function
if __name__ == "__main__":
    main()
```

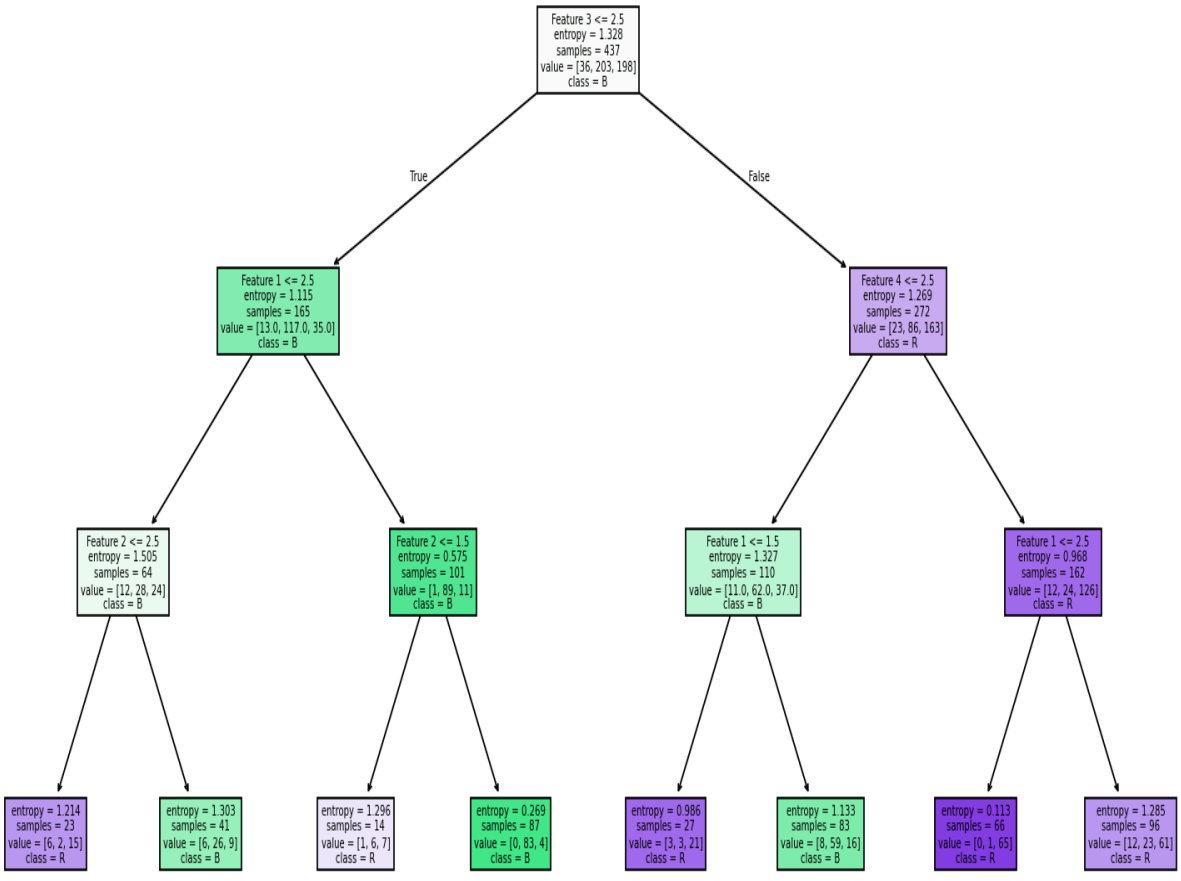
Output:-

```
[Running] python -u "c:\Users\Preetika\OneDrive\Desktop\TYsem5\AI TYCS\P5-decision_learning_tree.py"
Dataset Length:  625
Dataset Head:
   0  1  2  3  4
0  B  1  1  1  1
1  R  1  1  1  2
2  R  1  1  1  3
3  R  1  1  1  4
4  R  1  1  1  5
Results using entropy:
Predicted Values:
 ['R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'L'
 'L' 'R' 'L' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'L' 'L'
 'L' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'L' 'R' 'L' 'L' 'R' 'L' 'L'
 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'L' 'R' 'L' 'L' 'L' 'R'
 'R' 'L' 'R' 'L' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'R' 'R' 'L' 'R' 'L'
 'R' 'R' 'L' 'L' 'L' 'R' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'R' 'R' 'R' 'R' 'R'
 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L'
 'L' 'L' 'L' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R'
 'L' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'R' 'R'
 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'L' 'R'
 'R' 'R' 'L' 'L' 'L' 'R' 'R' 'R']
Accuracy:  70.74468085106383
Confusion Matrix:
 [[ 0  6  7]
 [ 0 63 22]
 [ 0 20 70]]
```

Classification Report:

	precision	recall	f1-score	support
B	0.00	0.00	0.00	13
L	0.71	0.74	0.72	85
R	0.71	0.78	0.74	90
accuracy			0.71	188
macro avg	0.47	0.51	0.49	188
weighted avg	0.66	0.71	0.68	188

Decision Tree Visualization



Roll No:- 15 **Practical No:- 4** **Date:-**

Aim:- Feed Forward Backpropagation Neural Network

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

The **Feed Forward Backpropagation Algorithm** is a supervised learning method used in artificial neural networks (ANNs).

It consists of two main phases:

1. **Feed Forward Phase:**
   - Input data is passed through the network layers (input, hidden, and output layers) one at a time, with each neuron processing the data using weights and activation functions.
   - The output layer generates predictions based on the input.
2. **Backpropagation Phase:**
   - The error between the predicted output and the actual output is calculated using a loss function.
   - This error is propagated backward through the network, updating the weights in each layer to reduce the error, typically using gradient descent.

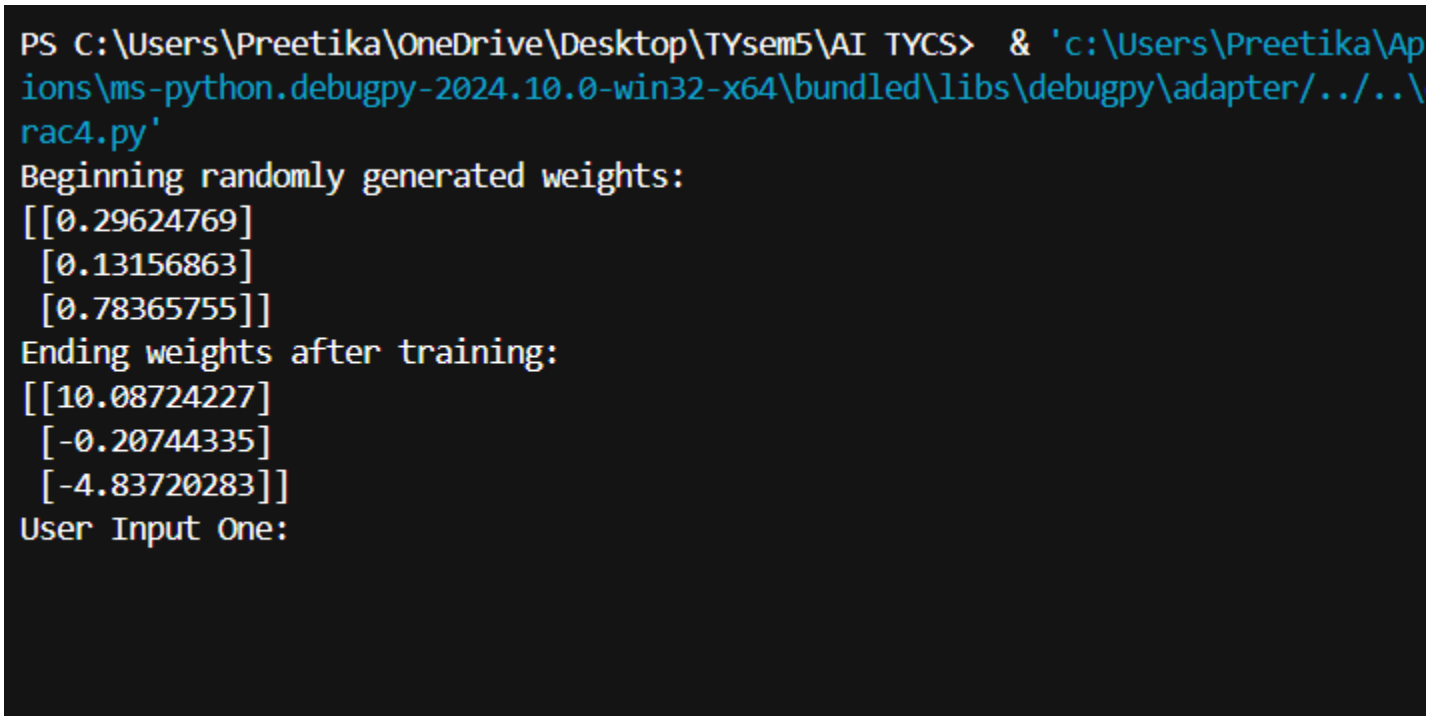
Source code:

```
import numpy as np

class NeuralNetwork():

    def __init__(self):

        #seeding for random number generation

        np.random.seed()

        #converting weights to a 3 by 1 matrix

        self.synaptic_weights=2*np.random.random((3,1))-1

    #x is output variable

    def sigmoid(self, x):

        #applying the sigmoid function

        return 1/(1+np.exp(-x))

    def sigmoid_derivative(self,x):
```

```
        #computing derivative to the sigmoid function
        return x*(1-x)
    def train(self,training_inputs,training_outputs,training_iterations):
        #training the model to make accurate predictions while adjusting
        for iteration in range(training_iterations):
            #siphon the training data via the neuron
            output=self.think(training_inputs)
            error=training_outputs-output
            #performing weight adjustments
            adjustments=np.dot(training_inputs.T,error*self.sigmoid_derivative(output))
            self.synaptic_weights+=adjustments
    def think(self,inputs):
        #passing the inputs via the neuron to get output
        #converting values to floats

        inputs=inputs.astype(float)
        output=self.sigmoid(np.dot(inputs,self.synaptic_weights))
        return output
if __name__=="__main__":
    #initializing the neuron class
    neural_network=NeuralNetwork()
    print("Beginning randomly generated weights: ")
    print(neural_network.synaptic_weights)
    #training data consisting of 4 examples--3 inputs & 1 output
    training_inputs=np.array([[0,0,1],[1,1,1],[1,0,1],[0,1,1]])
    training_outputs=np.array([[0,1,1,0]]).T
    #training taking place
```

```
neural_network.train(training_inputs,training_outputs,15000)

print("Ending weights after training: ")

print(neural_network.synaptic_weights)

user_input_one=str(input("User Input One: "))

user_input_two=str(input("User Input Two: "))

user_input_three=str(input("User Input Three: "))

print("Considering new situation: ",user_input_one,user_input_two,user_input_three)

print("New output data: ")

print(neural_network.think(np.array([user_input_one,user_input_two,user_input_three])))
```

Output:-



Roll No:- 15 **Practical No:- 5** **Date:-**

Aim:- Support Vector Machines (SVM)

- Implement the SVM algorithm for binary classification.
- Train an SVM model using a given dataset and optimize its parameters.
- Evaluate the performance of the SVM model on test data and analyze the results.

i) **Support Vector Machines (SVM)** is a supervised machine learning algorithm used for classification and regression tasks. The key idea behind SVM is to find the optimal hyperplane that best separates data points of different classes in a dataset.

- **Classification:** SVM aims to maximize the margin between two classes by finding the hyperplane (or decision boundary) that creates the largest possible distance between the closest data points of each class, called support vectors.
- **Kernel Trick:** If the data is not linearly separable, SVM uses a "kernel trick" to transform the data into a higher-dimensional space where a hyperplane can separate the classes.

ii) SVM is widely known for its effectiveness, especially in high-dimensional spaces.

Source code:-

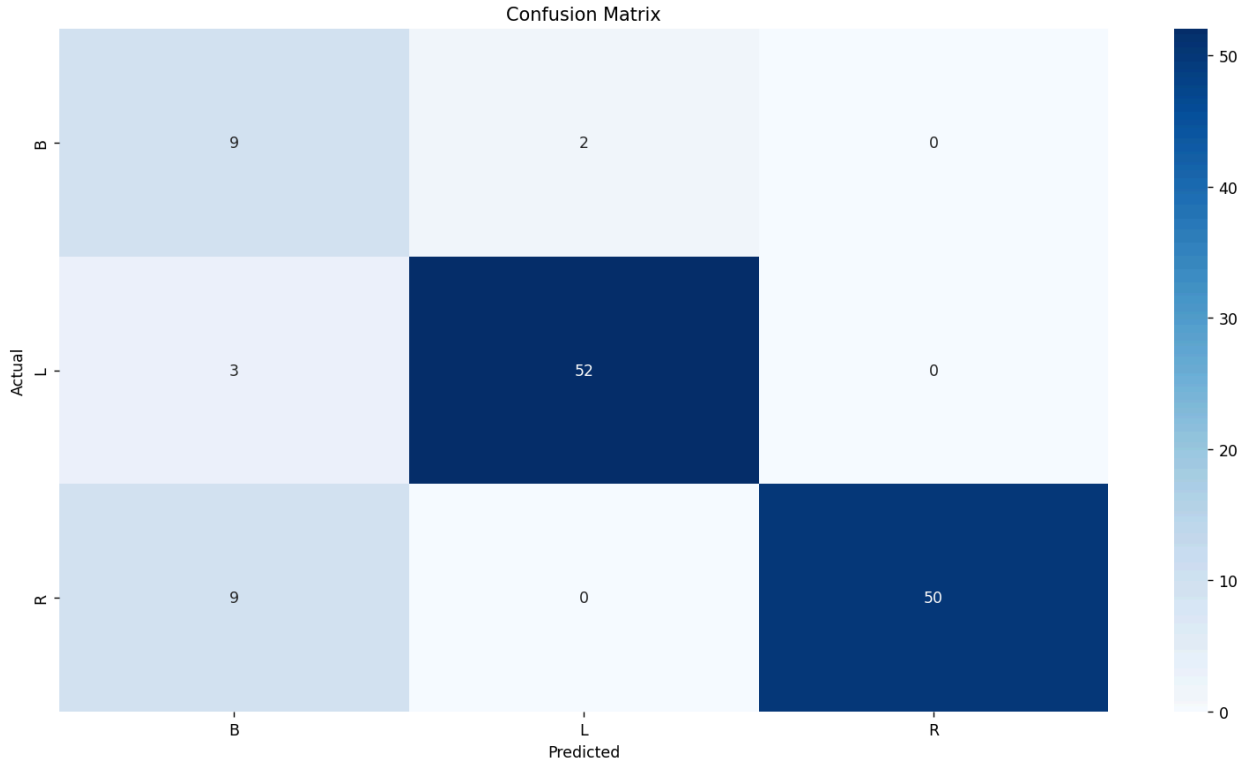
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import seaborn as sns

# Importing the dataset
data = pd.read_csv(r'C:\Users\Preetika\OneDrive\Desktop\TYsem5\AI
TYCS\P5-balance-scale.data', header=None)
# Assigning column names (assuming the dataset does not have headers)
# The first column is the target and the remaining are the features
data.columns = ['target', 'feature1', 'feature2', 'feature3', 'feature4']
# Separating features and target variable
X = data.drop('target', axis=1)
y = data['target']
# Splitting the dataset into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Training the SVM model with a linear kernel
model = SVC(kernel='linear')
# Fitting the model
model.fit(X_train, y_train)
# Predicting the test set results
```

```
y_pred = model.predict(X_test)
# Calculating accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
# Confusion Matrix and Classification Report
print("Classification Report:\n", classification_report(y_test, y_pred))
# Plotting Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=np.unique(y_test),
yticklabels=np.unique(y_test))
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
# Parameter tuning using GridSearchCV
param_grid = {'C': [0.1, 1, 10, 100], 'kernel': ['linear', 'rbf']}
grid = GridSearchCV(SVC(), param_grid, refit=True, verbose=3)
grid.fit(X_train, y_train)
# Best parameters and estimator
print("Best Parameters: ", grid.best_params_)
print("Best Estimator: ", grid.best_estimator_)
# Predicting again using the best model
y_pred_optimized = grid.predict(X_test)
# Accuracy after optimization
optimized_accuracy = accuracy_score(y_test, y_pred_optimized)
print(f"Optimized Accuracy: {optimized_accuracy:.2f}")
# Confusion Matrix and Classification Report after optimization
print("Optimized Classification Report:\n", classification_report(y_test,
y_pred_optimized))
```

Output:-

```
PS C:\Users\Preetika\OneDrive\Desktop\TYsem5\AI TYCS>  c:; cd 'c:\Users\Preet
thon\Python312\python.exe' 'c:\Users\Preetika\.vscode\extensions\ms-python.de
8' '--' 'c:\Users\Preetika\OneDrive\Desktop\TYsem5\AI TYCS\SVM.py'
Accuracy: 0.89
Classification Report:
              precision    recall  f1-score   support

           B       0.43      0.82      0.56        11
           L       0.96      0.95      0.95        55
           R       1.00      0.85      0.92        59

    accuracy                           0.89       125
   macro avg       0.80      0.87      0.81       125
weighted avg       0.93      0.89      0.90       125
```

Figure 1



Preetika nadar

Roll No:- 15 **Practical No:- 6** **Date:-**

Aim:- Adaboost Ensemble Learning

- Implement the Adaboost algorithm to create an ensemble of weak classifiers.
- Train the ensemble model on a given dataset and evaluate its performance.
- Compare the results with Individual weak classifiers

i) **AdaBoost (Adaptive Boosting)** is an ensemble learning algorithm that combines multiple weak classifiers to create a strong classifier. It works as follows:

1. **Initialization:** Each data point is assigned an equal weight.
2. **Training Weak Learners:** Multiple weak classifiers (often decision trees) are trained sequentially. After each iteration, the algorithm focuses more on the misclassified data points by increasing their weights.
3. **Combining Weak Learners:** The final classifier is a weighted sum of the weak learners, where better-performing classifiers contribute more.
4. **Prediction:** The final model makes predictions based on the combined output of all weak learners.

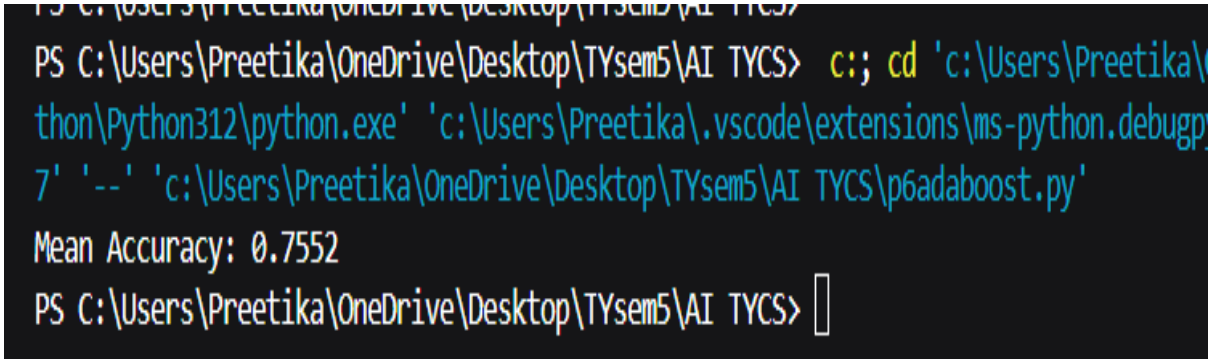
ii)AdaBoost is particularly effective in reducing bias and variance, improving classification performance.

Source code:-

```
import pandas as pd
from sklearn import model_selection
from sklearn.ensemble import AdaBoostClassifier
# Load dataset from URL
url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diab
etes.data.csv"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
dataframe = pd.read_csv(url, names=names)
# Splitting dataset into features (X) and target variable (Y)
array = dataframe.values
X = array[:, 0:8]  # First 8 columns (features)
Y = array[:, 8]    # Last column (target)
# Set random seed for reproducibility and define number of trees
seed = 7
num_trees = 30  # Number of weak learners (trees) to build

# Create an AdaBoost classifier with the SAMME algorithm
model = AdaBoostClassifier(n_estimators=num_trees, random_state=seed,
algorithm='SAMME')
```

```
# Use k-fold cross-validation (10 splits) to evaluate the model
kfold = model_selection.KFold(n_splits=10, shuffle=True, random_state=seed)
results = model_selection.cross_val_score(model, X, Y, cv=kfold)
# Print the average accuracy from the cross-validation
print(f"Mean Accuracy: {results.mean():.4f}")
```

Output:-



Aim:- Naive Bayes Classifiers :

- Implement the Naive Bayes algorithm for classification.
- Train and Naive Bayes model using a given data set and calculate class possibilities
- Evaluate the accuracy of the model on the test data and analyze the results.

i)**Naive Bayes** is a family of probabilistic algorithms based on Bayes' Theorem, used for classification tasks in machine learning. It assumes that the presence of a particular feature in a class is independent of the presence of any other feature, hence the term "naive."
ii)Naive Bayes is a straightforward yet powerful method for classification, particularly useful in scenarios where the independence assumption holds or when the computational efficiency of the algorithm is crucial.

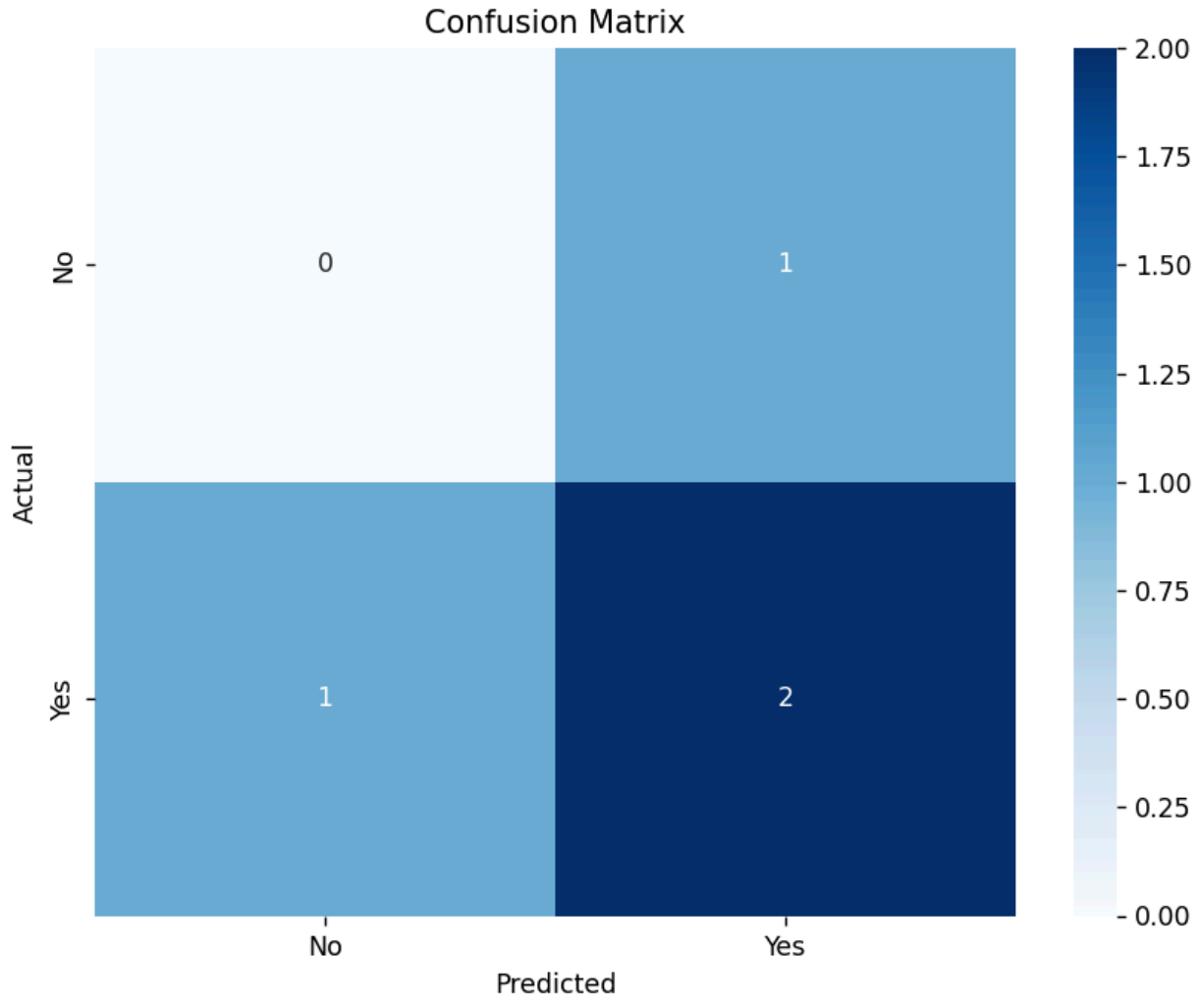
Source code:-

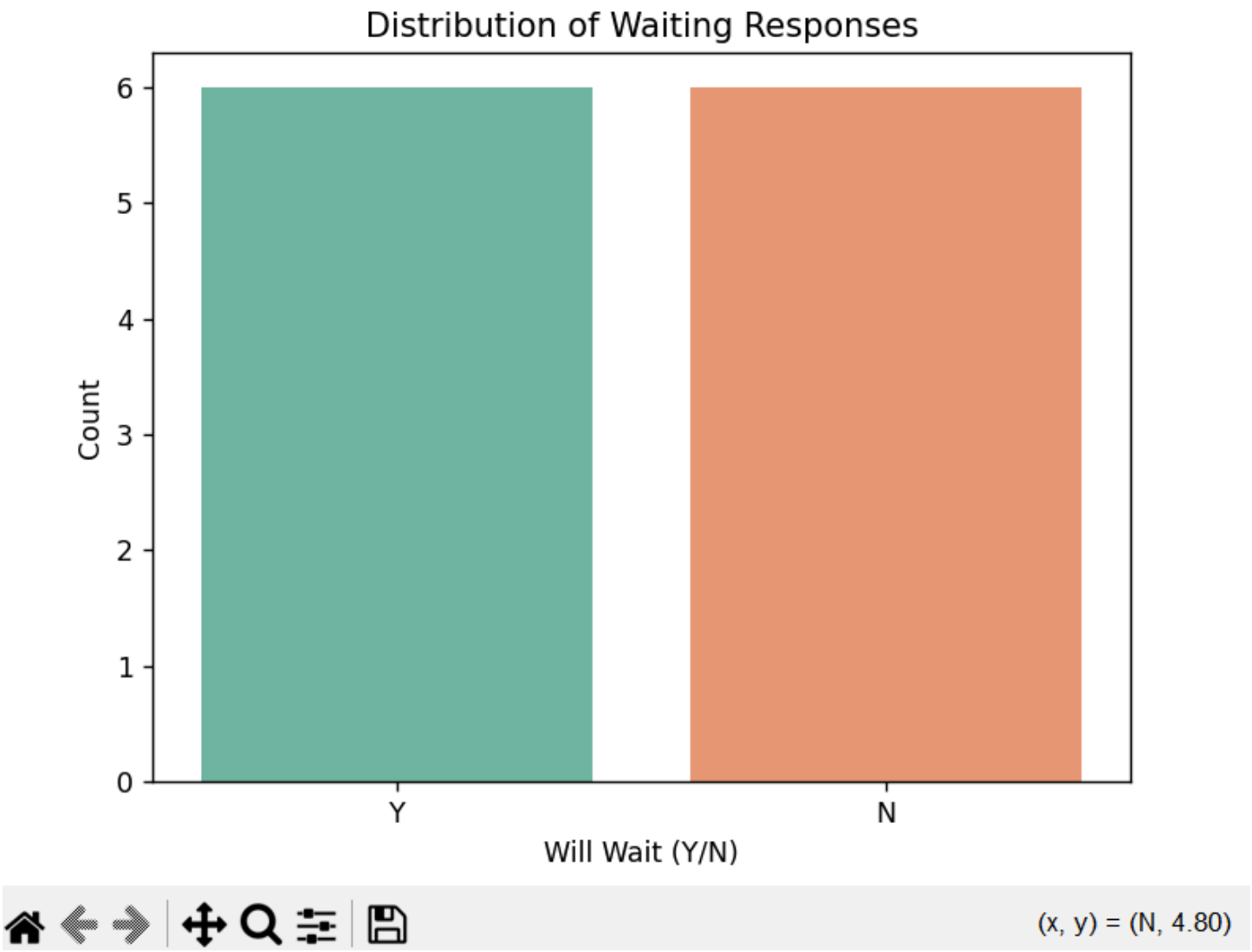
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
import seaborn as sns
# Sample dataset: You can replace this with your actual dataset
data = {
    'Alt': ['Y', 'N', 'Y', 'N', 'Y', 'N', 'Y', 'N', 'Y', 'N', 'Y', 'N'],
    'Est': ['0-10', '10-30', '30-60', '>60', '0-10', '10-30', '30-60', '>60', '0-10', '10-30',
'30-60', '>60'],
    'Pat': ['S', 'N', 'F', 'S', 'N', 'F', 'S', 'N', 'F', 'S', 'N', 'F'],
    'Type': ['F', 'I', 'B', 'T', 'F', 'I', 'B', 'T', 'F', 'I', 'B', 'T'],
    'ans': ['Y', 'N', 'Y', 'N', 'Y', 'Y', 'N', 'N', 'Y', 'Y', 'N', 'N']
}
# Create DataFrame
df = pd.DataFrame(data)
# Convert categorical data to numerical
```

```
df_encoded = pd.get_dummies(df, drop_first=True)
# Features and target variable
X = df_encoded.drop('ans_Y', axis=1)  # Features
y = df_encoded['ans_Y']  # Target variable (0: No, 1: Yes)
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)
# Initialize the Naive Bayes model
model = GaussianNB()
# Train the model
model.fit(X_train, y_train)
# Make predictions
y_pred = model.predict(X_test)
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(conf_matrix)
# Classification Report
class_report = classification_report(y_test, y_pred)
print("Classification Report:")
print(class_report)
# Visualize the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
        xticklabels=['No', 'Yes'], yticklabels=['No', 'Yes'])
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion Matrix')
plt.show()
# Visualize the results
sns.countplot(x='ans', data=df, palette='Set2', legend=False)
plt.title('Distribution of Waiting Responses')
plt.xlabel('Will Wait (Y/N)')
plt.ylabel('Count')
plt.show()
```

Output:-

```
PS C:\Users\Preetika\OneDrive\Desktop\TYsem5\AI TYCS>
PS C:\Users\Preetika\OneDrive\Desktop\TYsem5\AI TYCS>  c:; cd 'c:\Users\Preetika\OneDrive\Desktop\TYsem5\AI TYC
rs\Preetika\.vscode\extensions\ms-python.debugpy-2024.10.0-win32-x64\bundled\libs\debugpy\adapter/../..\debugpy
y'
Accuracy: 0.50
Confusion Matrix:
[[0 1]
 [1 2]]
Classification Report:
              precision    recall  f1-score   support

       False       0.00      0.00      0.00         1
        True       0.67      0.67      0.67         3

    accuracy                           0.50         4
   macro avg       0.33      0.33      0.33         4
weighted avg       0.50      0.50      0.50         4
```




Distribution of Waiting Responses
Count
0
1
2
3
4
5
6
Y
N
Will Wait (Y/N)
(x, y) = (N, 4.80)

Roll No:- 15 **Practical No:- 8** **Date:-**

Aim:- K-Nearest Neighbors:

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

i) The **K-NM (K-Nearest Mean)** algorithm is similar to K-Nearest Neighbors (K-NN). The primary difference is that instead of just considering the labels of the nearest neighbors (as in K-NN), K-NM averages or takes the most common value among the K nearest neighbors for predictions.

- **Classification**: The algorithm finds the most common class label among the K-nearest neighbors.
- **Regression**: The algorithm calculates the mean of the K-nearest neighbors' target values to predict the output.

Source code:-