V.S.B. Engineering College

(Recognition of College under Section 2 (f) & 12 (B) of the UGC Act, 1956)
Approved by AICTE, New Delhi and Affiliated to Anna University, Chennai
NAAC Accredited & ISO 9001 : 2015 Certified Institution
Karudayampalayam Post, KARUR - 639 111. Tamilnadu.

(AN AUTONOMOUS INSTITUTION)

DEPARTMENT OF

ELECTRICAL AND ELECTRONICS ENGINEERING

ACADEMIC YEAR 2024-2025 (EVEN)

OCS351 ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING FUNDAMENTALS LABORATORY

LAB MANUAL

Nmae of the student : _____

Register Number :_____

Year/Semseter/Section :_____

V.S.B. Engineering College
(Recognition of College under Section 2 (f) & 12 (B) of the UGC Act, 1956)
Approved by AICTE, New Delhi and Affiliated to Anna University, Chennai
NAAC Accredited & ISO 9001 : 2015 Certified Institution
Karudayampalayam Post, KARUR - 639 111. Tamilnadu.

(AN AUTONOMOUS INSTITUTION)

Vision of the Institution:

We endeavour to impart futuristic technical education of the highest quality to the student community and to inculcate discipline in them to face the world with self-confidence and thus we prepare them for life as responsible citizens to uphold human values and to be of service at large. We strive to bring of the Institution as an Institution of academic excellence of International standard.

Mission of the Institution:

We transform persons into personalities by the state-of the art infrastructure, time consciousness, quick response and the best academic practices through assessment and advice.

Vision of the Department:

- To create dynamic and challenging electrical engineers with social responsibilities.

Mission of the Department:

- To provide technical proficiency by adopting well defined teaching learning process.
- To create an environment to practice ethical codes.
- To prepare the graduates to be professionally competent to meet out the industrial needs.
- To motivate the students to pursue higher studies and research activities.

Programme Educational Objectives (PEOs)

PEO #1: Graduates of the program will be able to have a successful career in core and allied engineering or associated industries or in higher education or as entrepreneurs or in research.

PEO#2: Graduates of the program will be proficient to provide optimized solution for complex engineering problems in chosen Technical areas.

PEO#3: Graduates of the program are equipped to exhibit continuous improvement in their profession through life-long learning.

Program Outcomes (POs)

1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems

2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences

3. Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations

4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions

5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations

6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice

7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development

8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice

9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings

10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments

12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change

Program Specific Outcome (PSOs)

PSO1: Provide optimal solution in the field of Power sector.

PSO2: Apply suitable Electronic controllers for Power conversion, Control and Automation.

PSO3: Make use of appropriate technique and modern tools to analyze and evaluate the performance of Electrical machines and Electronic circuits.

SYLLABUS

OCS351 ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING FUNDAMENTALS LABORATORY

Programs for Problem solving with Search

1. Implement breadth first search
2. Implement depth first search
3. Analysis of breadth first and depth first search in terms of time and space
4. Implement and compare Greedy and A* algorithms.

Supervised learning

5. Implement the non-parametric locally weighted regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs
6. Write a program to demonstrate the working of the decision tree based algorithm.
7. Build an artificial neural network by implementing the back propagation algorithm and test the same using appropriate data sets.
8. Write a program to implement the naïve Bayesian classifier.

Unsupervised learning

9. Implementing neural network using self-organizing maps
10. Implementing k-Means algorithm to cluster a set of data.
11. Implementing hierarchical clustering algorithm.

Open Ended Experiment

12. Reinforcement Learning and Q-Learning Algorithm

CONTENTS

1. Implementation of breadth first search algorithms(bfs)

Aim:

To implement of breadth first search algorithms such as

Algorithm(bfs):

Step 1: Set the status to 1 (ready state) for each node in the graph.
Step 2: Enqueue the starting node and set its status to 2 (waiting state).
Step 3: Repeat steps 4 and 5 until the queue is empty.
Step 4: Dequeue a node, process it, and set its status to 3 (processed state).
Step 5: Enqueue all the neighbors of the node that are in the ready state (status = 1) and set their status to 2 (waiting state).
Step 6: Exit.

Program(bfs):

```
from collections import defaultdict

class Graph:
  def __init__(self):
  self.graph = defaultdict(list)

  def add_edge(self, u, v):
  self.graph[u].append(v)

  def bfs(self, s):
  visited = [False] * (len(self.graph))
  queue = []
  queue.append(s)
  visited[s] = True

  while queue:
  s = queue.pop(0)
  print(s, end=" ")

  for i in self.graph[s]:
  if not visited[i]:
  queue.append(i)
  visited[i] = True

  # Create a graph and test BFS
 g = Graph()
  g.add_edge(0, 1)
  g.add_edge(0, 2)
  g.add_edge(1, 2)
  g.add_edge(2, 0)
  g.add_edge(2, 3)
  g.add_edge(3, 3)

  print("Following is Breadth First Traversal (starting from vertex 2):")
  g.bfs(2)
```

Output(bfs):

Following isbreadth first traversal(startingfromvertex2)20 3
1

Particulars	Marks Allotted	Marks Obtained
Performance	50	
Viva Voce	10	
Record	15	
Total	75	

Result:

Thus the breadth first search algorithms such as bfs and been executed success fully and the output got verified.

2. Implementation of depth first search algorithms(dfs)

Aim:

To implementdepth first search algorithms dfs.

Algorithm:

Step 1: Set the status to 1 (ready state) for each node in the graph.
Step 2: Push the starting node onto the stack and set its status to 2 (waiting state).
Step 3: Repeat steps 4 and 5 until the stack is empty.
Step 4: Pop the top node, process it, and set its status to 3 (processed state).
Step 5: Push onto the stack all the neighbors of the node that are in the ready state (status = 1) and set their status to 2 (waiting state).
Step 6: Exit.

Program:

```
from collections import defaultdict

class Graph:
def __init__(self):
self.graph = defaultdict(list)

def add_edge(self, u, v):
self.graph[u].append(v)

def dfs_util(self, v, visited):
visited.add(v)
print(v, end=" ")

for neighbour in self.graph[v]:
if neighbour not in visited:
self.dfs_util(neighbour, visited)

def dfs(self, v):
visited = set()
self.dfs_util(v, visited)

if __name__ == "__main__":
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 0)
```

```
        g.add_edge(2, 3)
        g.add_edge(3, 3)

        print("Following is Depth First Traversal (starting from vertex 2):")
    g.dfs(2)
```

Output(dfs):

```
Following is depth first traversal(startingfromvertex2)
2 0 1 3
```

Particulars	Marks Allotted	Marks Obtained
Performance	50	
Viva Voce	10	
Record	15	
Total	75	

Result:

Thus the depth first search algorithms such as dfs and been executed success fully and the output got verified

3.Analysis of breadth first search algorithm and depth first search algorithms in terms of time and space

Aim:

To implement Analysis of breadth first search algorithm and depth first search Algorithms in terms of time and space

Algorithm

1.Breadth First Search — Time Complexity

Step 1:In BFS, the time complexity is also determined by the number of vertices (nodes) and edges in the graph.

Step 2 :BFS visits all the vertices at each level of the graph before moving to the next level.

Step 3 :In the worst case (as we always talk about upper bound in Big O notation), BFS may visit all vertices and edges in the graph.

Step 4:Therefore, 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.

2.Breadth First Search — Space Complexity

Step1:The space complexity of BFS depends on the maximum number of vertices in the queue at any given time.

Step 2:In the worst case, if the graph is a complete graph, all vertices at each level will be stored in the queue.

Step 3:Therefore, the space complexity of BFS is O(V), where V represents the number of vertices in the graph.

Algorithm

1.Depth First Search — Time Complexity

Step 1: In DFS, the time complexity is determined by the number of vertices (nodes) and edges in the graph.

Step 2:For each vertex, DFS visits **all its adjacent vertices** recursively.

Step 3:In the **worst case**, DFS may visit **all vertices and edges** in the graph.

Step 4: Therefore, the time complexity of DFS is **O(V + E)**, where V represents the number of vertices and E represents the number of edges in the graph.

2.Depth First Search — Space Complexity

Step 1:The space complexity of DFS depends on the maximum depth of recursion.

Step 2:In the worst case, if the graph is a straight line or a long path, the DFS recursion can go as deep as the number of vertices.

Step 3:Therefore, the space complexity of DFS is **O(V)**, where V represents the number of vertices in the graph

Program

```
import time
import psutil
from collections import deque

class Graph:
    def __init__(self, graph_dict=None):
        if graph_dict is None:
            graph_dict = {}
        self.graph_dict = graph_dict

    def add_edge(self, node, neighbour):
        if node not in self.graph_dict:
            self.graph_dict[node] = []
        self.graph_dict[node].append(neighbour)

    def bfs(self, start_node):
        visited = set()
        queue = deque([start_node])
        visited.add(start_node)
        while queue:
            current_node = queue.popleft()
            print(current_node, end=' ')
            for neighbour in self.graph_dict.get(current_node, []):
                if neighbour not in visited:
                    queue.append(neighbour)
                    visited.add(neighbour)

    def dfs(self, start_node, visited=None):
        if visited is None:
            visited = set()
        visited.add(start_node)
        print(start_node, end=' ')
        for neighbour in self.graph_dict.get(start_node, []):
            if neighbour not in visited:
                self.dfs(neighbour, visited)

# Memory usage helper
def get_memory_usage():
    process = psutil.Process()
    return process.memory_info().rss / 1024  # Memory in KB

# Example usage:
graph = Graph({

    0: [1, 2],
    1: [2],
    2: [0, 3],
    3: [3]
})
```

```
# Measure BFS execution time and memory usage
print("BFS traversal:")
start_time = time.perf_counter()
bfs_memory_before = get_memory_usage()
graph.bfs(0)
bfs_memory_after = get_memory_usage()
end_time = time.perf_counter()
bfs_execution_time = end_time - start_time
bfs_memory_usage = bfs_memory_after - bfs_memory_before
print(f"\nBFS Execution Time: {bfs_execution_time:.6f} seconds")
print(f"BFS Memory Used: {bfs_memory_usage:.2f} KB")

# Measure DFS execution time and memory usage
print("\nDFS traversal:")
start_time = time.perf_counter()
dfs_memory_before = get_memory_usage()
graph.dfs(0)
dfs_memory_after = get_memory_usage()
end_time = time.perf_counter()
dfs_execution_time = end_time - start_time
dfs_memory_usage = dfs_memory_after - dfs_memory_before
print(f"\nDFS Execution Time: {dfs_execution_time:.6f} seconds")
print(f"DFS Memory Used: {dfs_memory_usage:.2f} KB")
```

Output:

```
BFS Execution Time: 0.000684 seconds
BFS Memory Used: 8.00 KB

DFS traversal:
0 1 2 3
DFS Execution Time: 0.000131 seconds
DFS Memory Used: 0.00 KB
```

Particulars	Marks Allotted	Marks Obtained
Performance	50	
VivaVoce	10	
Record	15	
Total	75	

Result:

Thus the depth first search algorithms and breadth first search algorithm time and space complexity has been executed success fully and the output got verified.

4.Implement and compare Greedy and A* algorithms

Aim:

To Implement and compare Greedy and A* algorithms

Algorithm:

Step 1:Selects the node which appears to be the closest to the goal based on a heuristic function.
Step 2:Consider the cost of reaching the current node, only the estimated distance to the goal.
Step 3: It considers both the cost of reaching the current node and the estimated distance to the goal.
Step 4: Greedy BFS because it considers the total cost of reaching the goal, not just the heuristic.
Step 5: A* guarantees optimal solutions if the heuristic function is admissible never overestimates the true cost to reach the goal.
Step 6: A* can be seen as a generalization of GBFS, where A* reduces to GBFS when the cost-to-go heuristic is 0.

Program:

```
import heapq

class Graph:
    def __init__(self, graph_dict=None):
        if graph_dict is None:
            graph_dict = {}
        self.graph_dict = graph_dict

    def add_edge(self, node, neighbour, cost):
        if node not in self.graph_dict:
            self.graph_dict[node] = []
        self.graph_dict[node].append((neighbour, cost))

    def greedy_best_first_search(self, start, goal):
        frontier = [(0, start, [start])]  # (heuristic, current_node, path)
        explored = set()

        while frontier:
            _, current_node, path = heapq.heappop(frontier)

            if current_node == goal:
                print("Greedy Best-First Search path:", path)
                return path

            explored.add(current_node)

            for neighbour, cost in self.graph_dict.get(current_node, []):
                if neighbour not in explored:
                    heapq.heappush(frontier, (self.heuristic(neighbour, goal), neighbour, path + [neighbour]))
                print("Goal not reachable")
                    return None
            def astar_search(self, start, goal):
             frontier = [(0 + self.heuristic(start, goal), 0, start, [start])]  # (f(n), g(n), current_node, path)
              explored = set()
```

```
        while frontier:
            _, g_cost, current_node, path = heapq.heappop(frontier)

            if current_node == goal:
                print("A* Search path:", path)
                return path

            explored.add(current_node)

            for neighbour, cost in self.graph_dict.get(current_node, []):
                if neighbour not in explored:
                    heapq.heappush(
                        frontier,
                        (g_cost + cost + self.heuristic(neighbour, goal), g_cost + cost, neighbour, path + [neighbour])
                    )

        print("Goal not reachable")
        return None

    def heuristic(self, node, goal):
        # Simplified heuristic: returns 0 to mimic uniform traversal
        # For more complex graphs, you can customize this function
        return 0

# Create the graph
graph = Graph({
    'A': [('B', 5)],
    'B': [('C', 7), ('F', 2)],
    'C': [('D', 3)],
    'D': [('F', 1)],
    'F': []
})

# Perform searches
start = 'A'
goal = 'F'

print("Greedy Best-First Search:")
graph.greedy_best_first_search(start, goal)

print("\nA* Search:")
graph.astar_search(start, goal)
```

Output:

```
Greedy Best-First Search:
Greedy Best-First Search path: ['A', 'B', 'C', 'D', 'F']

A* Search:
A* Search path: ['A', 'B', 'F']
```

Particulars	Marks Allotted	Marks Obtained
Performance	50	
Viva Voce	10	
Record	15	
Total	75	

Result:

Thus the compare Greedy and A* algorithms has been executed success fully and the output got verified.

5.Implement the non-parametric locally weighted regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs.

Aim :

To Implement the non-parametric locally weighted regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs.

Algorithm:

Step 1:Read the Given data Sample to X and the curve (linear or non linear) to Y

Step 2.:Set the value for Smoothening parameter or Free parameter say τ

Step 3:Set the bias /Point of interest set x0 which is a subset of X

Step 4: Determine the weight matrix using

Step 5:Determine the value of model term parameter β using

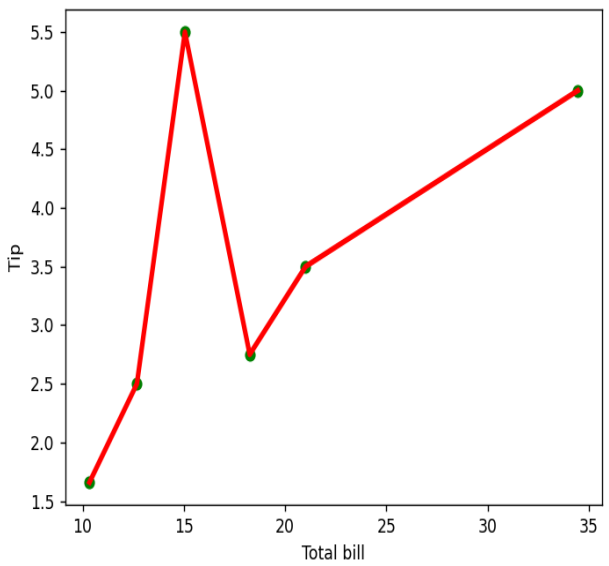
Step 6: Set the bias /Point of interest set x0 which is a subset of X.

Program:

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

def kernel(point, xmat, k):
    """Calculate the weights for the local regression."""
    m, n = np.shape(xmat)
    weights = np.eye(m)  # Identity matrix for weights
    for j in range(m):
        diff = point - xmat[j]
        weights[j, j] = np.exp(-(diff @ diff.T) / (2 * k**2))  # Gaussian kernel
    return weights

def localWeight(point, xmat, ymat, k):
    """Compute the locally weighted regression coefficients."""
    wei = kernel(point, xmat, k)
    XtWX = xmat.T @ (wei @ xmat)
    if np.linalg.det(XtWX) == 0:  # Check if the matrix is singular
        XtWX += np.eye(XtWX.shape[0]) * 1e-5  # Add small value to diagonal
    W = np.linalg.inv(XtWX) @ (xmat.T @ (wei @ ymat))
    return W
```

```
def localWeightRegression(xmat, ymat, k):
    """Perform locally weighted regression."""
    m, n = np.shape(xmat)
    ypred = np.zeros(m)
    for i in range(m):
        ypred[i] = xmat[i] @ localWeight(xmat[i], xmat, ymat, k)
    return ypred

# Load dataset
data = pd.read_csv("10-dataset.csv")  # Ensure this file exists
bill = np.array(data["total_bill"]).reshape(-1, 1)
tip = np.array(data["tip"]).reshape(-1, 1)

# Add bias column of ones
m = np.shape(bill)[0]
one = np.ones((m, 1))
X = np.hstack((one, bill))  # Add intercept term

# Set kernel bandwidth
k = 0.5
ypred = localWeightRegression(X, tip, k)

# Sort predictions for smooth plotting
SortIndex = X[:, 1].argsort().flatten()
xsort = X[SortIndex]

# Plot the results
plt.figure(figsize=(8, 6))
plt.scatter(bill, tip, color="green", label="Data Points")
plt.plot(xsort[:, 1], ypred[SortIndex], color="red", linewidth=3, label="Locally Weighted Regression")
plt.xlabel("Total Bill ($)")
plt.ylabel("Tip ($)")
plt.legend()
plt.show()
```

Output :



Particulars	Marks Allotted	Marks Obtained
Performance	50	
Viva Voce	10	
Record	15	
Total	75	

Result:

Thus the non-parametric locally weighted regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs has been executed successfully and the output got verified.

6.Write a program to demonstrate the working of the decision tree based algorithm

Aim:
To implement the concept of a decision tree, which is suitable for real-world problems, using the CART algorithm.

Algorithm:

Step 1: It begins with the original sets as the root node.

Step 2: On each iteration of the algorithm, it iterates through every unused attribute of the sets and calculates the Gini index of this attribute.

Step 3: Gini index works with the categorical target variable “success” or “failure.” It performs only binary splits.

Step 4: The sets are then split by the selected attribute to produce a subset of the data.

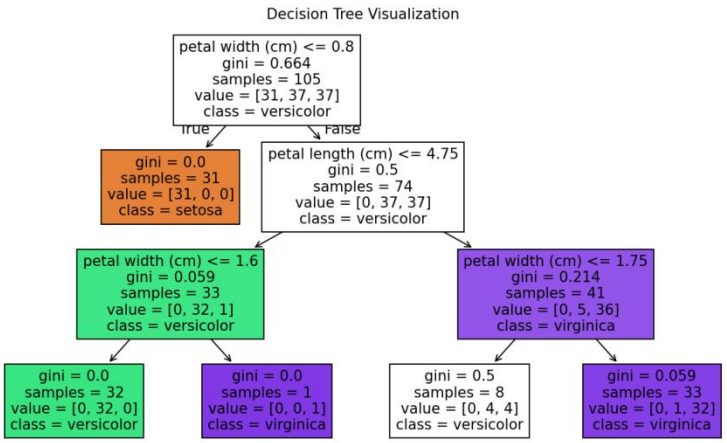
Step 5: The algorithm continues to recur on each subset, considering only attributes never selected before.

Program:

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, export_text, plot_tree
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features (sepal and petal dimensions)
y = iris.target  # Labels (setosa, versicolor, virginica)

# Add a new feature: 'age'
# For demonstration, randomly generate age values between 1 and 100
np.random.seed(42)  # For reproducibility
age = np.random.randint(1, 101, size=X.shape[0])  # Random ages
X = np.column_stack((X, age))  # Append 'age' to the feature set

# Update feature names to include 'age'
feature_names = iris.feature_names + ['age']

# Split the dataset 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)



# Create and train the Decision Tree Classifier
classifier = DecisionTreeClassifier(criterion='gini', max_depth=3, random_state=42)
```

```
classifier.fit(X_train, y_train)


# Make predictions on the test set
y_pred = classifier.predict(X_test)

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

# Display the Decision Tree rules
tree_rules = export_text(classifier, feature_names=feature_names)
print("\nDecision Tree Rules:\n")
print(tree_rules)

# Visualize the Decision Tree
plt.figure(figsize=(12, 8))
plot_tree(classifier, feature_names=feature_names, class_names=iris.target_names, filled=True)
plt.title("Decision Tree Visualization")
plt.show()
```

Output of decision tree without pruning:



Particulars	Marks Allotted	Marks Obtained
Performance	50	
VivaVoce	10	
Record	15	
Total	75	

Result:

Thus, the program to implement the concept of a decision tree with a suitable dataset from real world problems using the CART algorithm has been executed successfully.

7. Build an artificial neural network by implementing the back propagation algorithm and test the same using appropriate data sets.

Aim:

To implement the neural network by using the backpropagation algorithm and test it with appropriate datasets.

Algorithm:

Step 1: Image Acquisition
The first step is to acquire images of paper documents with the help of optical scanners. This way, an original image can be captured and stored.

Step 2: Pre-processing
The noise level on an image should be optimized, and areas outside the text should be removed. Pre-processing is especially vital for recognizing handwritten documents, as they are more sensitive to noise.

Step 3: Segmentation
The process of segmentation aims to group characters into meaningful chunks. There can be predefined classes for characters, so images can be scanned for patterns that match these classes.

Step 4: Feature Extraction
This step involves splitting the input data into a set of features. The goal is to identify essential characteristics that make one pattern recognizable from another.

Step 5: Training an MLP Neural Network

1. Start with the input layer and propagate data forward to the output layer. This step is known as **forward propagation**.
2. Based on the output, calculate the **error** (the difference between the predicted and known outcome). The error needs to be minimized.
3. Perform **backpropagation** of the error by finding its derivative with respect to each weight in the network and updating the model accordingly.

Step 6: Post-processing
This stage involves refining the OCR model, as some corrections may be required. However, achieving 100% recognition accuracy is not always possible. The identification of characters heavily depends on the context.

Program:

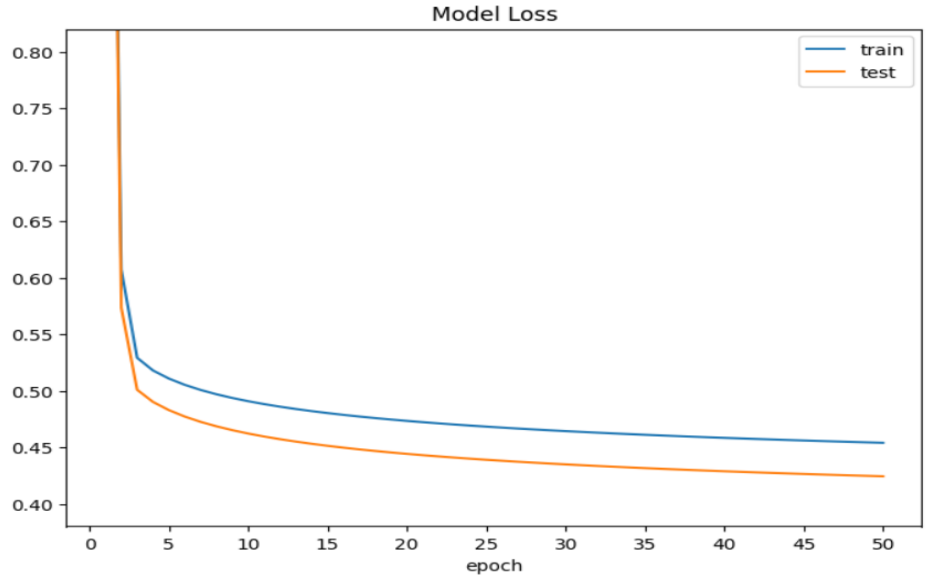
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split

class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size): # Changed _init_ to __init__
        self.input_size = input_size
```

```
self.hidden_size = hidden_size
        self.output_size = output_size

        self.weights1 = np.random.randn(self.input_size, self.hidden_size) * 0.9
        self.biases1 = np.zeros((1, self.hidden_size))
        self.weights2 = np.random.randn(self.hidden_size, self.output_size) * 0.9
        self.biases2 = np.zeros((1, self.output_size))

    # ... (rest of the class code remains the same)

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        return x * (1 - x)

    def forward(self, X):
        self.z1 = np.dot(X, self.weights1) + self.biases1
        self.a1 = self.sigmoid(self.z1)
        self.z2 = np.dot(self.a1, self.weights2) + self.biases2
        self.a2 = self.sigmoid(self.z2)
        return self.a2

    def backward(self, X, y, learning_rate):
        m = X.shape[0]

        dz2 = self.a2 - y
        dw2 = (1/m) * np.dot(self.a1.T, dz2)
        db2 = (1/m) * np.sum(dz2, axis=0, keepdims=True)

        dz1 = dz2.dot(self.weights2.T) * self.sigmoid_derivative(self.a1)
        dw1 = (1/m) * np.dot(X.T, dz1)
        db1 = (1/m) * np.sum(dz1, axis=0, keepdims=True)

        self.weights1 -= learning_rate * dw1
        self.biases1 -= learning_rate * db1
        self.weights2 -= learning_rate * dw2
        self.biases2 -= learning_rate * db2

    def train(self, X, y, epochs=10000, learning_rate=0.1, print_loss=False):
        for i in range(epochs):
            output = self.forward(X)
            self.backward(X, y, learning_rate)
            if print_loss and i % 1000 == 0:
                loss = self.calculate_loss(X, y)
                print(f"Epoch {i}, Loss: {loss}")

    def calculate_loss(self, X, y, lambda_l2=0):  # Added L2 regularization (optional)
        output = self.forward(X)
        m = X.shape[0]
        loss = (-y * np.log(output) - (1 - y) * np.log(1 - output))
        mean_loss = (1/m) * np.sum(loss)
        # L2 regularization (optional - uncomment to use)
```

```
        # regularization_term = (lambda_l2 / (2 * m)) * (np.sum(self.weights1*2) + np.sum(self.weights2*2))
            # return mean_loss + regularization_term
            return mean_loss #Without regularization

        def predict(self, X):
            output = self.forward(X)
            predictions = (output > 0.5)
            return predictions

    # Generate a more complex dataset
    X, y = make_moons(n_samples=200, noise=0.25, random_state=42)
    y = y.reshape(-1, 1)

    # Split 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)

    # Adjust hidden_size to get ~730k parameters
    hidden_size = 50  # Experiment with this value!

    nn = NeuralNetwork(input_size=2, hidden_size=hidden_size, output_size=1)

    total_params = nn.weights1.size + nn.biases1.size + nn.weights2.size + nn.biases2.size
    print(f"Total parameters: {total_params}")

    # --- Learning Rate Schedule (Example: Time-based decay) ---
    initial_learning_rate = 0.9
    decay_rate = 0.99 # Adjust as needed

    # --- L2 Regularization (Optional - uncomment to use) ---
    lambda_l2 = 0.99  # Adjust as needed

    losses_train = []
    losses_test = []
    epochs_list = []

    num_epochs = 50
    for i in range(num_epochs):
        # --- Learning Rate Update ---
        learning_rate = initial_learning_rate / (1 + decay_rate * i)

        nn.train(X_train, y_train, epochs=1, learning_rate=learning_rate, print_loss=False)

        # --- Calculate Loss with Regularization (if used) ---
        loss_train = nn.calculate_loss(X_train, y_train, lambda_l2=lambda_l2)
        losses_train.append(loss_train)
        loss_test = nn.calculate_loss(X_test, y_test, lambda_l2=lambda_l2)
        losses_test.append(loss_test)
        epochs_list.append(i + 1)

        if i % 10 == 0:
            print(f"Epoch {i + 1}, Train Loss: {loss_train}, Test Loss: {loss_test}")
```

```
predictions = nn.predict(X)
print("Predictions:")
print(predictions)

# Plotting
plt.figure(figsize=(8, 6))
plt.plot(epochs_list, losses_train, label='train', color='tab:blue')
plt.plot(epochs_list, losses_test, label='test', color='tab:orange')

plt.xlabel('epoch')
plt.ylabel('')
plt.title('Model Loss')
plt.legend(loc='upper right')

plt.ylim(0.38, 0.82)
plt.yticks(np.arange(0.40, 0.81, 0.05))

plt.xticks(np.arange(0, num_epochs + 1, 5))

plt.text(0, 0.25, f"Total params: {total_params}")
plt.text(0, 0.20, f"Trainable params: {total_params}")
plt.text(0, 0.15, "Non-trainable params: 0")

plt.grid(False)

plt.show()
```

output:



Total params: 201

Trainable params: 201

Non-trainable params: 0

Particulars	MarksAllotted	MarksObtained
Performance	50	
VivaVoce	10	
Record	15	
Total	75	

Result:

Thus, the program to implement the neural network model for the given dataset has been successfully executed.

8.Write a program to implement the naïve Bayesian classifier

Aim:

To diagnose heart patients and predict disease using the heart disease dataset with the Naïve Bayes classifier algorithm.

Algorithm:

Step 1: Read the training dataset.

Step 2: Calculate the mean and standard deviation of the predictor variables in each class.

Step 3: Repeat the following steps until the probability of all predictor variables (f_1, f_2, f_3, ..., f_n) has been calculated:

- Compute the probability of each feature using the Gaussian density equation in each class.

Step 4: Calculate the likelihood for each class.

Step 5: Identify the class with the greatest likelihood.

Program:

```
import csv
import numpy as np
from sklearn.metrics import confusion_matrix, f1_score, roc_curve, auc
import matplotlib.pyplot as plt
from itertools import cycle
from numpy import interp
import warnings
import random
import math

def generate_heartdisease_data(num_rows=303):
    """Generates sample data for heartdisease.csv."""
    data = []
    for _ in range(num_rows):
        row = [
            random.randint(29, 77),  # age
            random.randint(0, 1),  # sex
            random.randint(0, 3),  # cp
            random.randint(94, 200),  # restbp
            random.randint(126, 564),  # chol
            random.randint(0, 1),  # fbs
            random.randint(0, 2),  # restecg
            random.randint(71, 202),  # thalach
            random.randint(0, 1),  # exang
            round(random.uniform(0.0, 6.2), 1),  # oldpeak
            random.randint(0, 2),  # slope
```

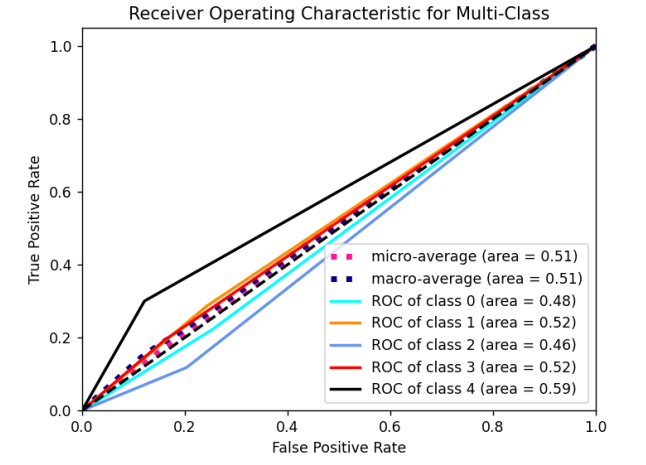
```
                random.randint(0, 3),  # ca
                random.randint(0, 3),  # thal (changed to 0-3 for simplicity)
                random.randint(0, 4)  # num (changed to 0-4 for simplicity)
            ]

        data.append(row)

    # Save data to CSV file
    with open('heartdisease.csv', 'w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(['age', 'sex', 'cp', 'restbp', 'chol', 'fbs', 'restecg', 'thalach', 'exang',
'oldpeak', 'slope', 'ca', 'thal', 'num'])
        writer.writerows(data)

    print("CSV file 'heartdisease.csv' has been generated.")

# Call the function to generate the data before the main execution:
generate_heartdisease_data()

# ... (Rest of your code, starting with reading the dataset) ...
dataset = []
with open('heartdisease.csv', 'r') as file:
    reader = csv.reader(file)
    # Skip the header row
    next(reader)

    for row_index, row in enumerate(reader):  # Added row_index for error reporting
        try:
            dataset.append([float(x) for x in row])
        except ValueError as e:
            print(f"Error on row {row_index + 2}: {e}")  # +2 to account for header and 0-based
indexing
            print(f"Row content: {row}")  # Print the offending row
            # You might want to handle the error here, e.g., skip the row, fill with a default value,
etc.

# Convert txt file to csv
with open('heartdisease.csv', 'r') as in_file:
    stripped = (line.strip() for line in in_file)
    lines = [line.split(",") for line in stripped if line]

with open('heartdisease.csv', 'w', newline='') as out_file:
    writer = csv.writer(out_file)
    writer.writerow(('age', 'sex', 'cp', 'restbp', 'chol', 'fbs', 'restecg',
                     'thalach', 'exang', 'oldpeak', 'slope', 'ca', 'thal', 'num'))
    writer.writerows(lines)

warnings.filterwarnings("ignore")

# Function to calculate mean
def mean(column_values):
    return sum(map(float, column_values)) / len(column_values)
```

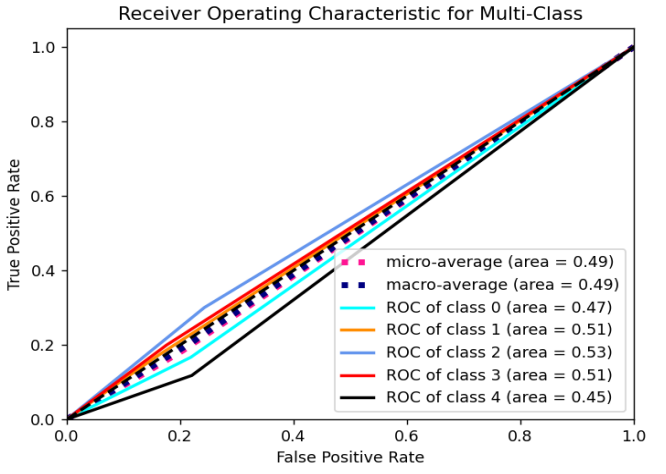
```
# Function to calculate standard deviation
def stdev(column_values):
    avg = mean(column_values)
    variance = sum((float(x) - avg) ** 2 for x in column_values) / (len(column_values) - 1)
    return math.sqrt(variance)

# # Reading csv file
# dataset = []
# with open('heartdisease.csv', 'r') as file:
#     lines = csv.reader(file)
#     dataset = list(lines)[1:]
#     dataset = [[float(x) for x in row] for row in dataset]

for z in range(5):
    print("\n\n\nTest-Train Split No.", z + 1, "\n\n\n")
    train_size = int(len(dataset) * 0.75)
    train_set = []
    test_set = list(dataset)

    for _ in range(train_size):
        index = random.randrange(len(test_set))
        train_set.append(test_set.pop(index))

    # Separate list according to class
    class_list = {}
    for row in train_set:
        class_num = row[-1]
        if class_num not in class_list:
            class_list[class_num] = []
        class_list[class_num].append(row)

    # Prepare data class-wise
    class_data = {}
    for class_num, rows in class_list.items():
        class_data[class_num] = [(mean(col), stdev(col)) for col in zip(*rows)][:-1]

    # Getting test vector
    y_test = [row[-1] for row in test_set]

    # Getting prediction vector
    y_pred = []
    for row in test_set:
        class_probability = {}
        for class_num, stats in class_data.items():
            class_probability[class_num] = 1
            for j, (calculated_mean, calculated_dev) in enumerate(stats):
                x = row[j]
                if calculated_dev != 0:
                    exponent = math.exp(-(math.pow(x - calculated_mean, 2) / (2 *
math.pow(calculated_dev, 2))))
                    probability = (1 / (math.sqrt(2 * math.pi) * calculated_dev)) * exponent
                    class_probability[class_num] *= probability
```

```
            resultant_class = max(class_probability, key=class_probability.get)
                y_pred.append(resultant_class)

            # Getting accuracy
            accuracy = sum(1 for i in range(len(test_set)) if test_set[i][-1] == y_pred[i]) / len(test_set)
* 100

            print("\n\nAccuracy:", accuracy, "%")

            # Confusion matrix and F1 score
            y_test = np.array(y_test)
            y_pred = np.array(y_pred)
            print("\n\nConfusion Matrix")
            print(confusion_matrix(y_test, y_pred))
            print("\n\nF1 Score")
            print(f1_score(y_test, y_pred, average='weighted'))

            # ROC Curve
            n_classes = 5
            y_test_bin = np.zeros((len(y_test), n_classes))
            y_pred_bin = np.zeros((len(y_pred), n_classes))

            for i in range(len(y_test)):
                y_test_bin[i][int(y_test[i])] = 1
            for i in range(len(y_pred)):
                y_pred_bin[i][int(y_pred[i])] = 1

            fpr, tpr, roc_auc = {}, {}, {}
            for i in range(n_classes):
                fpr[i], tpr[i], _ = roc_curve(y_test_bin[:, i], y_pred_bin[:, i])
                roc_auc[i] = auc(fpr[i], tpr[i])

            fpr["micro"], tpr["micro"], _ = roc_curve(y_test_bin.ravel(), y_pred_bin.ravel())
            roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

            all_fpr = np.unique(np.concatenate([fpr[i] for i in range(n_classes)]))
            mean_tpr = np.zeros_like(all_fpr)
            for i in range(n_classes):
                mean_tpr += interp(all_fpr, fpr[i], tpr[i])
            mean_tpr /= n_classes

            fpr["macro"], tpr["macro"] = all_fpr, mean_tpr
            roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

            # Plot all ROC curves
            plt.figure()
            plt.plot(fpr["micro"], tpr["micro"], label='micro-average (area =
{0:0.2f})'.format(roc_auc["micro"]),
                     color='deeppink', linestyle=':', linewidth=4)
            plt.plot(fpr["macro"], tpr["macro"], label='macro-average (area =
{0:0.2f})'.format(roc_auc["macro"]),
                     color='navy', linestyle=':', linewidth=4)
```

```
        colors = cycle(['aqua', 'darkorange', 'cornflowerblue', 'red', 'black'])
            for i, color in zip(range(n_classes), colors):
                plt.plot(fpr[i], tpr[i], color=color, lw=2, label='ROC of class {0} (area = {1:0.2f})'.format(i, roc_auc[i]))

            plt.plot([0, 1], [0, 1], 'k--', lw=2)
            plt.xlim([0.0, 1.0])
            plt.ylim([0.0, 1.05])

        plt.xlabel('False Positive Rate')
            plt.ylabel('True Positive Rate')
            plt.title('Receiver Operating Characteristic for Multi-Class')
            plt.legend(loc="lower right")
    plt.show()
```

Output:





Particulars	MarksAllotted	MarksObtained
Performance	50	
VivaVoce	10	
Record	15	
Total	75	

Result:

Thus, the program to diagnose heart patients and predict disease using the heart disease dataset with the Naïve Bayes classifier algorithm has been executed successfully, and the output has been verified.

9.Implementing neural network using self-organizing maps

Aim:

To Implementing neural network using self-organizing maps

Algorithm:

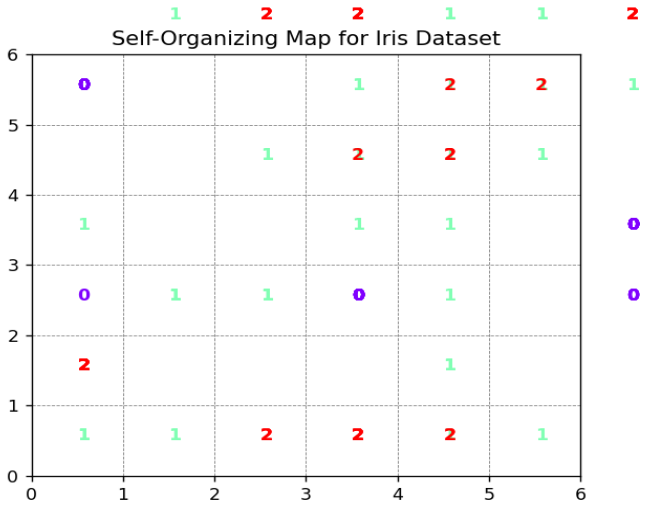
Step 1: Initialize the weights of the neurons.
Step 2:Iterate through the input data points.
Step 3:For each input data point, find the Best Matching Unit (BMU) - the neuron with the closest weights to the input vector.
Step 4: Update the weights of the BMU and its neighbors based on a learning rate and neighborhood function.
Step 5: Repeat steps 3-4 for a certain number of epochs.

Program :

```
import numpy as np
from minisom import MiniSom
from sklearn.datasets import load_iris
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt

# Load the Iris dataset
iris = load_iris()
data = iris.data
target = iris.target

# Normalize the data
scaler = MinMaxScaler()
data = scaler.fit_transform(data)

# Initialize the SOM
som_size = (7, 7)  # Dimensions of the SOM grid
som = MiniSom(som_size[0], som_size[1], data.shape[1], sigma=1.0, learning_rate=0.5)

# Train the SOM
som.random_weights_init(data)
print("Training SOM...")
som.train_random(data, num_iteration=100)  # Number of iterations
print("Training completed.")

# Visualize the SOM
plt.figure(figsize=(8, 8))
for i, x in enumerate(data):
    w = som.winner(x)  # Get the winning neuron for the data sample
    plt.text(
        w[0] + 0.5,
        w[1] + 0.5,
        str(target[i]),
        color=plt.cm.rainbow(target[i] / 2),
        fontdict={'weight': 'bold', 'size': 10}
    )
```

```
plt.xticks(range(som_size[0]))
plt.yticks(range(som_size[1]))
plt.grid(which='both', color='gray', linestyle='--', linewidth=0.5)
plt.title("Self-Organizing Map for Iris Dataset")
plt.show()
```

Output:



Particulars	MarksAllotted	MarksObtained
Performance	50	
VivaVoce	10	
Record	15	
Total	75	

Result:

This is a basic implementation of a Self-Organizing Map on your specific use case and requirements, you may need to customize has been executed sucessfully.

10.Implementing k-Means algorithm to cluster a set of data

Aim:

To Implementing k-Means algorithm to cluster a set of data

Algorithm:

Step 1: Select the number kkk of neighbors.
Step 2: Calculate the Euclidean distance of the kkk nearest neighbors.
Step 3: Take the kkk nearest neighbors based on the calculated Euclidean distance.
Step 4: Among these kkk neighbors, count the number of data points in each category.
Step 5: Assign the new data point to the category for which the number of neighbors is maximum.
Step 6: Our model is ready.

Program:

```
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import precision_score, recall_score, f1_score

class KMeans:
    def __init__(self, k=3, max_iters=100):
        """
        Initialize KMeans with the number of clusters (k) and maximum iterations.

        :param k: Number of clusters (default: 3)
        :param max_iters: Maximum iterations for convergence (default: 100)
        """
        self.k = k
        self.max_iters = max_iters
        self.centroids = None
        self.labels = None

    def _initialize_centroids(self, X):
        """
        Randomly initialize centroids from the data points.

        :param X: Input data
        :return: Initial centroids
        """
        indices = np.random.choice(X.shape[0], self.k, replace=False)
        return X[indices, :]

    def _assign_labels(self, X):
        """
        Assign each data point to the closest centroid.

        :param X: Input data
        :return: Assigned labels
```

```
        """
        distances = np.sqrt(((X - self.centroids[:, np.newaxis]) ** 2).sum(axis=2))
        return np.argmin(distances, axis=0)

    def _update_centroids(self, X):
        """
        Update centroids as the mean of all data points in a cluster.

        :param X: Input data
        :return: Updated centroids
        """
        return np.array([X[self.labels == i].mean(axis=0) for i in range(self.k)])

    def fit(self, X):
        """
        Run the k-Means algorithm to cluster the data.

        :param X: Input data
        """
        self.centroids = self._initialize_centroids(X)
        for _ in range(self.max_iters):
            prev_centroids = self.centroids
            self.labels = self._assign_labels(X)
            self.centroids = self._update_centroids(X)
            if np.all(self.centroids == prev_centroids):
                break

    def predict(self, X):
        """
        Predict the cluster labels for new data.

        :param X: New input data
        :return: Predicted labels
        """
        return self._assign_labels(X)


# **Example Usage**
if __name__ == "__main__":
    # Generate a sample dataset with known cluster labels
    X, y = make_blobs(n_samples=30, centers=3, n_features=2, random_state=1, cluster_std=0.8)

    # Create a KMeans instance with k=3
    kmeans = KMeans(k=3)

    # Fit the model to the data
    kmeans.fit(X)

    # Predict cluster labels for the data
    predicted_labels = kmeans.labels

    # **Evaluation**
    # Encode cluster labels to match the predicted labels' range (0 to k-1)
    le = LabelEncoder()
```

```
y_encoded = le.fit_transform(y)

# Ensure the number of clusters matches
assert len(np.unique(y_encoded)) == len(np.unique(predicted_labels))

# **Print Evaluation Metrics**
print("Accuracy:")
print("          ", accuracy_score(y_encoded, predicted_labels))
print()
print("Precision\tRecall\tf1-score\tSupport")
for i in np.unique(y_encoded):
   precision = precision_score(y_encoded, predicted_labels, average=None)[i]
   recall = recall_score(y_encoded, predicted_labels, average=None)[i]
   f1 = f1_score(y_encoded, predicted_labels, average=None)[i]
   support = (y_encoded == i).sum()
   print(f"{i}\t{precision:.2f}\t{recall:.2f}\t{f1:.2f}\t{support}")
print()
print("Accuracy\t", accuracy_score(y_encoded, predicted_labels))
print("Macro Avg\t",
     precision_score(y_encoded, predicted_labels, average='macro'),
     recall_score(y_encoded, predicted_labels, average='macro'),
     f1_score(y_encoded, predicted_labels, average='macro'),
     len(y_encoded))
print("Weighted Avg\t",
     precision_score(y_encoded, predicted_labels, average='weighted'),
     recall_score(y_encoded, predicted_labels, average='weighted'),
     f1_score(y_encoded, predicted_labels, average='weighted'),
len(y_encoded))
```

Output:

```
Accuracy:
          1.0

Precision       Recall  f1-score        Support
0       1.00    1.00    1.00    10
1       1.00    1.00    1.00    10
2       1.00    1.00    1.00    10

Accuracy          1.0
Macro Avg         1.0 1.0 1.0 30
Weighted Avg      1.0 1.0 1.0 30
```

Particulars	MarksAllotted	MarksObtained
Performance	50	
VivaVoce	10	
Record	15	
Total	75	

Result:

Thus, the program to implement the K-Nearest Neighbors algorithm for clustering the Iris dataset has been executed successfully, and the output has been verified.

11. **Implementing hierarchical clustering algorithm.**

Aim:

To implement the hierarchical clustering algorithm, perform agglomerative hierarchical clustering using single linkage.

Algorithm:

Step 1: Compute the proximity matrix using a particular distance metric

Step 2: Each data point is assigned to a cluster

Step 3: Merge the clusters based on a metric for the similarity between clusters

Step 4: Update the distance matrix

Step 5: Repeat Step 3 and Step 4 until only a single cluster remains.

Program:

```
import numpy as np
from scipy.cluster.hierarchy import linkage, fcluster
from scipy.spatial.distance import pdist

# **Input Data**
X = np.array([[1, 2],
              [5, 8],
              [1.5, 1.8],
              [8, 8],
              [1, 0.6],
              [9, 11]])

# **Hierarchical Clustering**
def hierarchical_clustering(X, method='single', threshold=0.5):
    # Calculate the distance matrix
    dist = pdist(X)

    # Perform hierarchical clustering
    Z = linkage(dist, method=method)
```

```
    # Form flat clusters from the hierarchical clustering
    clusters = fcluster(Z, t=threshold, criterion='distance')

    return Z, clusters

# **Print Cluster Hierarchy**
def print_cluster_hierarchy(Z, X, threshold=0.5):
    print("Hierarchical Clustering:")
    print("------------------------")
    print("Data Points:")
    for i, point in enumerate(X):
        print(f"{i}: {point}")

    print("\nCluster Hierarchy (Distance < {:.1f}):".format(threshold))
    print("-------------------------------------")
    for i, merge in enumerate(Z):
        if merge[2] < threshold:  # Only print merges below the threshold
            print(f"Merge {i} - Distance: {merge[2]:.2f}")
            print(f"  Cluster {int(merge[0])} + Cluster {int(merge[1])}")
            print()

# **Main Execution**
if __name__ == "__main__":
    Z, clusters = hierarchical_clustering(X, method='single', threshold=0.5)
    print_cluster_hierarchy(Z, X, threshold=0.5)
    print("\nFinal Cluster Assignments (Threshold = 0.5):")
    print("--------------------------------------------")
    for i, cluster in enumerate(clusters):
        print(f"Point {i}: {X[i]} -> Cluster {cluster}")
```

Output:

```
Hierarchical Clustering:
-------------------------
Data Points:
0: [1.  2. ]
1: [5.  8. ]
2: [1.5 1.8]
3: [8.  8. ]
4: [1.  0.6]
5: [ 9. 11. ]

Cluster Hierarchy (Distance < 0.5):
----------------------------------------
Merge 0 - Distance: 0.43
  Cluster 0 + Cluster 2

Merge 1 - Distance: 0.44
  Cluster 4 + Cluster 0

Final Cluster Assignments (Threshold = 0.5):
-----------------------------------------
Point 0: [1.  2.] -> Cluster 1
Point 1: [5.  8.] -> Cluster 2
Point 2: [1.5 1.8] -> Cluster 1
Point 3: [8.  8.] -> Cluster 3
Point 4: [1.  0.6] -> Cluster 1
Point 5: [ 9. 11.] -> Cluster 3
```

Particulars	MarksAllotted	MarksObtained
Performance	50	
VivaVoce	10	
Record	15	
Total	75	

Result:

This implementation performs agglomerative hierarchical clustering by iteratively merging the two closest clusters until only a single cluster remains has been executed successfully.

12. Reinforcement Learning and Q-Learning Algorithm

Aim:

The aim of this program is to implement a Q-learning algorithm for training an agent to navigate a Grid World environment and reach a goal.

Algorithm:

- **Compute the Proximity Matrix:**
  - Calculate the pairwise distance (using a metric like Euclidean distance) between each pair of data points to form a proximity matrix.
- **Initialize Clusters:**
  - Treat each data point as an individual cluster. Assign each data point to its own cluster.
- **Merge the Closest Clusters:**
  - Identify the two closest clusters using a similarity metric (e.g., minimum distance for single linkage).
  - Merge these two clusters into a new cluster.
- **Update the Proximity Matrix:**
  - Update the proximity matrix to reflect the new distances between the merged cluster and the remaining clusters.
- **Repeat the Process:**
  - Repeat Steps 3 and 4 iteratively until all data points are merged into a single cluster.
- **Exit:**
  - Once there is only one remaining cluster, stop the process. This final cluster represents the entire dataset.

```
import numpy as np
import random

# Create the Grid World environment
class GridWorld:
    def __init__(self, rows, cols, start, goal):
        self.rows = rows
        self.cols = cols
        self.start = start
        self.goal = goal
        self.state = start  # agent starts at the start position

    def reset(self):
        self.state = self.start
```

```
        return self.state

    def step(self, action):
        # Define actions: 0 = up, 1 = right, 2 = down, 3 = left
        row, col = self.state
        if action == 0:  # Move up
            row = max(row - 1, 0)
        elif action == 1:  # Move right
            col = min(col + 1, self.cols - 1)
        elif action == 2:  # Move down
            row = min(row + 1, self.rows - 1)
        elif action == 3:  # Move left
            col = max(col - 1, 0)

        self.state = (row, col)

        # Reward is -1 for each step until the goal is reached
        if self.state == self.goal:
            return self.state, 0  # Reached goal, no penalty
        return self.state, -1  # Negative reward for each move

    def get_actions(self):
        return [0, 1, 2, 3]  # 0 = up, 1 = right, 2 = down, 3 = left

# Q-learning implementation
class QLearningAgent:
    def __init__(self, env, alpha=0.1, gamma=0.9, epsilon=0.1):
        self.env = env
        self.alpha = alpha  # Learning rate
        self.gamma = gamma  # Discount factor
        self.epsilon = epsilon  # Exploration rate
        self.q_table = np.zeros((env.rows, env.cols, len(env.get_actions())))  # Initialize Q-table

    def choose_action(self, state):
        # Epsilon-greedy action selection
        if random.uniform(0, 1) < self.epsilon:
            return random.choice(self.env.get_actions())  # Explore (random action)
        else:
            row, col = state
            return np.argmax(self.q_table[row, col])  # Exploit (best action)

    def update_q_table(self, state, action, reward, next_state):
        row, col = state
        next_row, next_col = next_state
        best_next_action = np.argmax(self.q_table[next_row, next_col])  # Best action in next state
        # Q-learning update rule
        self.q_table[row, col, action] = self.q_table[row, col, action] + self.alpha * (
            reward + self.gamma * self.q_table[next_row, next_col, best_next_action] - self.q_table[row, col,
action]
        )

    def train(self, episodes=1000):
        for _ in range(episodes):
```

```
        state = self.env.reset()
        done = False
        while not done:
            action = self.choose_action(state)
            next_state, reward = self.env.step(action)
            self.update_q_table(state, action, reward, next_state)
            state = next_state
            if state == self.env.goal:
                done = True  # Goal reached

# Initialize the environment and agent
env = GridWorld(rows=5, cols=5, start=(0, 0), goal=(4, 4))
agent = QLearningAgent(env)

# Train the agent
agent.train(episodes=1000)

# Test the agent after training
state = env.reset()
done = False
steps = 0
while not done:
    action = agent.choose_action(state)
    next_state, _ = env.step(action)
    print(f"Step {steps}: Position {state} -> Action {action} -> New Position {next_state}")
    state = next_state
    steps += 1
    if state == env.goal:
        done = True

print(f"Goal reached in {steps} steps!")
```

Output:

```
Step 0: Position (0, 0) -> Action 1 -> New Position (0, 1)
Step 1: Position (0, 1) -> Action 1 -> New Position (0, 2)
Step 2: Position (0, 2) -> Action 1 -> New Position (0, 3)
Step 3: Position (0, 3) -> Action 2 -> New Position (1, 3)
Step 4: Position (1, 3) -> Action 2 -> New Position (2, 3)
Step 5: Position (2, 3) -> Action 2 -> New Position (3, 3)
Step 6: Position (3, 3) -> Action 2 -> New Position (4, 3)
Step 7: Position (4, 3) -> Action 1 -> New Position (4, 4)
```

Particulars	MarksAllotted	MarksObtained
Performance	50	
VivaVoce	10	
Record	15	
Total	75	

Result:

Reinforcement Learning and Q-Learning Algorithm has been executed successfully.