SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)
TIRUCHIRAPPALLI

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
TIRUCHIRAPPALLI CAMPUS

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING – (PCA20D07J)

Record Work

Register No. :________________________________

Name of the Student :________________________________

Semester :________________________________

Programme : **MCA**

November 2024

TIRUCHIRAPPALLI

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
TIRUCHIRAPPALLI CAMPUS

BONAFIDE CERTIFICATE

Certified as the bonafide record of work done by ________________________________,

Register No.__________________________________of_________(Semester/Year),

MCA____Programme in the practical course **PCA20D07J – Artificial Intelligence and Machine Learning** at **SRM Institute of Science and Technology, Tiruchirappalli Campus** during the Year Nov - 2024-2025.

Faculty In-charge **Head of the Department**

Submitted for the End Semester Examination held on _________________

Examiner-1 **Examiner-2**

INDEX

1 Simple AI Techniques Implementation

Aim:

To implement simple Artificial Intelligence (AI) techniques using basic algorithms.

Algorithm:

Step 1: Import Necessary Libraries

Step 2: Load Dataset

Step 3: Data Preprocessing

Step 4: Train Model

Step 5: Prediction

Step 6: Evaluation

Program Code:

```
def diagnose_fever(symptoms):
    """Diagnoses fever based on symptoms"""
    if 'high temperature' in symptoms:
        if 'headache' in symptoms and 'chills' in symptoms:
            return "Diagnosis: You likely have a fever."
        elif 'sweating' in symptoms:
            return "Diagnosis: You might have a mild fever."
        else:
            return "Diagnosis: Check for other symptoms, but fever is possible."
    else:
        return "Diagnosis: You do not appear to have a fever."


if __name__ == "__main__":
    print("Enter your symptoms separated by commas (e.g., 'high temperature, headache, sweating'):")
    user_input = input().split(', ')
    result = diagnose_fever(user_input)
    print(result)
```

Output:

Enter your symptoms separated by commas (e.g., 'high temperature, headache, sweating'):

Sweating

Diagnosis: You do not appear to have a fever.

Result:

Thus implementation of a simple decision tree program was executed successfully.

2 Tic-Tac-Toe Game Implementation

Aim:

To implement a Tic-Tac-Toe game in Python where two players take turns to play, and the program checks for a win, loss, or draw.

Algorithm:

Step 1 : Create a 3x3 grid representing the Tic-Tac-Toe board. Each cell can be empty, or filled with either ‘X’ or ‘O’.

Step 2 : Write a function to print the current state of the board after each player’s move

Step 3 : Allow two players to input their moves by selecting a cell (1-9).

Step 4 : After every move, check if the current player has won by forming a line (horizontally, vertically, or diagonally).

Step 5 : If the board is full and no player has won, declare the game a draw.

Step 6 : After each move, switch between Player 1 (X) and Player 2 (O).

Program Code:

```
def print_board(board):
    """Prints the Tic-Tac-Toe board"""
    for row in board:
        print("|".join(row))
        print("-" * 5)


def check_winner(board, player):
    """Check if the given player has won"""
    for row in board:
        if all([spot == player for spot in row]):
            return True
    for col in range(3):
        if all([board[row][col] == player for row in range(3)]):
            return True
```

```
    if all([board[i][i] == player for i in range(3)]) or all([board[i][2 - i] == player for i in range(3)]):
        return True
    return False


def is_draw(board):
    """Check if the game is a draw"""
    return all([spot != ' ' for row in board for spot in row])


def play_game():
    """Main function to play the game"""
    board = [[' ' for _ in range(3)] for _ in range(3)]
    current_player = 'X'

    while True:
        print_board(board)
        print(f"Player {current_player}'s turn")
        try:
            row, col = map(int, input("Enter row and column (0, 1, or 2) separated by space: ").split())
            if board[row][col] != ' ':
                print("Cell is already taken. Try again.")
                continue
        except (ValueError, IndexError):
            print("Invalid input. Please enter two numbers between 0 and 2.")
            continue
        board[row][col] = current_player
        if check_winner(board, current_player):
            print_board(board)
            print(f"Player {current_player} wins!")
```

```
            break
        if is_draw(board):
            print_board(board)
            print("It's a draw!")
            break

        current_player = 'O' if current_player == 'X' else 'X'

if __name__ == "__main__":
    play_game()
```

Output:

```
||
-----
 ||
-----
 ||
Player X's turn
Enter row and column (0, 1, or 2) separated by space: 0 0

X||
-----
 ||
-----
 ||
Player O's turn
Enter row and column (0, 1, or 2) separated by space: 1 1

X||
-----
 |O|
```

```
-----
 | |
Player X's turn
Enter row and column (0, 1, or 2) separated by space: 0 1

X|X|
-----
 |O|
-----
 | |
Player O's turn
Enter row and column (0, 1, or 2) separated by space: 2 2

X|X|
-----
 |O|
-----
 | |O
Player X's turn
Enter row and column (0, 1, or 2) separated by space: 0 2

X|X|X
-----
 |O|
-----
 | |O
Player X wins!
```

Result:

The Tic-Tac-Toe game was successfully implemented.

3 Implementation of an Intelligent Agent

Aim:

To implement an intelligent agent in Python that interacts with a simple environment and performs actions based on the perception of its surroundings, demonstrating decision-making capabilities.

Algorithm:

Step 1: Create a grid environment representing a room where each cell is either "clean" or "dirty."

Step 2: The agent starts at a random position on the grid.

Step 3: The agent perceives the state (clean/dirty) of the cell it currently occupies.

Step 4:

- If the cell is dirty, the agent cleans it.
- If the cell is clean, the agent moves to the next cell.

Step 5: The agent repeats the process of perception and action until the entire environment is clean.

Program code:

```
import random

class Environment:

  def __init__(self, rows, cols):

    self.grid = [[random.choice(['Clean', 'Dirty']) for _ in range(cols)] for _ in range(rows)]

    self.agent_position = [0, 0]  # Agent starts at top-left corner


  def is_dirty(self, row, col):

    return self.grid[row][col] == 'Dirty'


  def clean(self, row, col):

    self.grid[row][col] = 'Clean'


  def display(self):

    for row in self.grid:
```

```
            print(row)
        print()
class VacuumAgent:
    def __init__(self, env):
        self.env = env
    def sense_and_act(self):
        row, col = self.env.agent_position

        if self.env.is_dirty(row, col):
            print(f"Cleaning position: {row}, {col}")
            self.env.clean(row, col)
        else:
            print(f"Position: {row}, {col} is already clean")
        self.move()
    def move(self):
        row, col = self.env.agent_position
        if col < len(self.env.grid[0]) - 1:  # Move right if possible
            self.env.agent_position = [row, col + 1]
        elif row < len(self.env.grid) - 1:  # Move down if possible
            self.env.agent_position = [row + 1, 0]
if __name__ == "__main__":
    rows, cols = 2, 3  # 2x3 grid environment
    env = Environment(rows, cols)
      agent = VacuumAgent(env)
      print("Initial Environment:")
    env.display()
    for _ in range(rows * cols):  # Perform actions for all grid cells
        agent.sense_and_act()
    print("Final Environment:")
    env.display()
```

Output:

Initial Environment:

['Clean', 'Clean', 'Clean']

['Clean', 'Dirty', 'Clean']

Position: 0, 2 is already clean

Position: 1, 0 is already clean

Position: 0, 2 is already clean

Position: 1, 0 is already clean

Position: 1, 2 is already clean

Final Environment:

['Clean', 'Clean', 'Clean']

['Clean', 'Clean', 'Clean']

Result:

Thus, the intelligent agent was successfully implemented.

4. Implementation of Ontology and FOL

Aim:

To Write the program to implementing ontology and FOL is to represent structured knowledge and enable logical reasoning for automated inference and decision-making.

Algorithm:

Step 1: Create an empty ontology structure with sets for concepts, individuals, and relationships.

Step 2: Initialize a knowledge base to store facts and rules.

Step 3: Add C to the set of concepts in the ontology.

Step 4: Add R to the set of relationships between concepts or individuals.

Step 5: Assign I to a corresponding concept C in the ontology. Store this assignment as a fact in the knowledge base (e.g., is a(John, Human)).

Step 6: For each rule R in First-Order Logic in the rule's conditions and consequences (e.g., if is a(X, Human) and is a(Human, Mammal), then is a(X, Mammal)).

Program Code:

Ontology

```
class Animal:
    def __init__(self, name):
        self.name = name
    def is_alive(self):
        return True
class Mammal(Animal):
    def has_hair(self):
        return True
class Bird(Animal):
    def can_fly(self):
        return True
dog = Mammal('Dog')
eagle = Bird('Eagle')
```

```
print(f"{dog.name} is a mammal with hair: {dog.has_hair()}")
print(f"{eagle.name} is a bird that can fly: {eagle.can_fly()}")
```

Output:

Dog is a mammal with hair: True

Eagle is a bird that can fly: True

FOL

```
def human(x):
    return x in ['Socrates', 'Plato', 'Aristotle']
def mortal(x):
    return human(x)
def is_mortal(x):
    if human(x):
        return True
    return False
person = 'Socrates'
print(f"{person} is mortal: {is_mortal(person)}")
```

Output:

Socrates is mortal: True

Result:

Thus the Implementation of Ontology & FOL Program was executed Successfully.

5. Concept learning task

Aim:

To write the concept learning task is to identify a general rule or hypothesis that accurately classifies data based on a given set of labeled examples.

Algorithm:

Step 1: Set the initial hypothesis h_0 to the most specific hypothesis (denoted as S) or the most general hypothesis (denoted as G).

Step 2: Iterate through each training example (x, y) in the dataset.

Step 3: Narrow down the version space by eliminating inconsistent hypotheses that fail to classify the examples correctly.

Step 4: Once all training examples have been processed, output the most specific hypothesis consistent with the training data.

Program Code:

```
def find_s_algorithm(training_data):
    hypothesis = None
     for instance in training_data:
       features, label = instance[:-1], instance[-1]
       if label == 'positive':
           if hypothesis is None:
               hypothesis = features.copy()
           else:
               for i in range(len(hypothesis)):
                   if hypothesis[i] != features[i]:
                       hypothesis[i] = '?'  # Replace differing values with '?'
    return hypothesis
```

```
training_data = [
    ['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same', 'positive'],
    ['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same', 'positive'],
    ['Rainy', 'Cold', 'High', 'Strong', 'Warm', 'Change', 'negative'],
    ['Sunny', 'Warm', 'High', 'Strong', 'Cool', 'Change', 'positive']
]
hypothesis = find_s_algorithm(training_data)
print("Most specific hypothesis:", hypothesis)
```

Output:

Most specific hypothesis: ['Sunny', 'Warm', '?', 'Strong', '?', '?']

Result:

Thus the above program was executed successfully.

6. Implementation of candidate elimination algorithm

Aim:

To write a program to Implementation of candidate elimination algorithm

Algorithm:

Step 1: Set the most specific hypothesis S to [0, 0, ..., 0] and the most general hypothesis G to [?, ?, ..., ?]

Step 2: For each instance in the dataset, split it into attribute values and the class label (positive or negative).

Step 3: Ensure S and G remain consistent by retaining only the hypotheses that classify the examples correctly.

Step 4: After all training examples are processed, output the refined most specific hypothesis S and the most general hypothesis G.

Step 5: Finish the program.

Program Code:

```
def candidate_elimination(data):
    S, G = ['0'] * (len(data[0]) - 1), ['?'] * (len(data[0]) - 1)
    for instance in data:
        x, label = instance[:-1], instance[-1]
        if label == 'Yes':
            S = [xi if si == '0' else '?' if si != xi else si for si, xi in zip(S, x)]
        elif label == 'No':
            G = [gi if xi == gi else '?' for gi, xi in zip(G, x)]
    return S, G
```

```
data = [
    ['Sunny', 'Warm', 'Normal', 'Strong', 'Yes'],
    ['Sunny', 'Warm', 'High', 'Strong', 'Yes'],
    ['Rainy', 'Cold', 'High', 'Strong', 'No'],
    ['Sunny', 'Warm', 'High', 'Weak', 'Yes']
]
S, G = candidate_elimination(data)
print("S:", S)
print("G:", G)
```

Output:

```
S: [ 'Sunny', 'Warm', '?', '?' ]
G: [ '?', '?', '?', '?' ]
```

Result:

Thus the above Implementation of candidate elimination algorithm was executed successfully.

7 Decision tree implementation

Aim:

The aim of this implementation is to create a decision tree classifier that can be used to predict the class of new data points based on a given dataset.

Algorithm:

Step 1: Start the Program.

Step 2: Import Libraries: Import necessary libraries such as NumPy, Pandas, Matplotlib, and Scikit-learn.

Step 3: Load the Dataset: Load the Iris dataset, which contains features of iris flowers and their corresponding species.

Step 4: **Data Preprocessing**: Split the dataset into features and labels, and then into training and testing sets.

Step 5: Create a decision tree classifier and fit it to the training data.

Step 6: Make Predictions: Use the trained model to predict the species of the test data.

Step 7:Evaluate the Model: Calculate accuracy and visualize the results.

Step 8:Visualize the Decision Tree: Plot the decision tree for better understanding.

Step 9: Stop the Program.

Algorithm:

```
import numpy as np

class Node:

    def __init__(self, feature=None, threshold=None, left=None, right=None, value=None):

        self.feature = feature

        self.threshold = threshold

        self.left = left

        self.right = right

        self.value = value
```

```python
class DecisionTree:
    def __init__(self, max_depth=5):
        self.max_depth = max_depth
        self.tree = None
    def gini(self, y):
        classes = np.unique(y)
        gini = 1.0
        for c in classes:
            p = np.sum(y == c) / len(y)
            gini -= p ** 2
        return gini
    def split(self, X, y, feature, threshold):
        left_mask = X[:, feature] <= threshold
        right_mask = X[:, feature] > threshold
        return X[left_mask], X[right_mask], y[left_mask], y[right_mask]
    def best_split(self, X, y):
        best_feature, best_threshold, best_gini = None, None, float('inf')
        for feature in range(X.shape[1]):
            thresholds = np.unique(X[:, feature])
            for threshold in thresholds:
                X_left, X_right, y_left, y_right = self.split(X, y, feature, threshold)
                if len(y_left) == 0 or len(y_right) == 0:
                    continue
                gini_left = self.gini(y_left)
                gini_right = self.gini(y_right)
                weighted_gini = (len(y_left) / len(y) * gini_left) + (len(y_right) / len(y) * gini_right)
                if weighted_gini < best_gini:
```

```
                best_feature, best_threshold, best_gini = feature, threshold, weighted_gini
        return best_feature, best_threshold
    def build_tree(self, X, y, depth=0):
        num_samples_per_class = [np.sum(y == c) for c in np.unique(y)]
        most_common_class = np.argmax(num_samples_per_class)

        if depth >= self.max_depth or len(np.unique(y)) == 1:
            return Node(value=most_common_class)
        feature, threshold = self.best_split(X, y)
        if feature is None:
            return Node(value=most_common_class)
        X_left, X_right, y_left, y_right = self.split(X, y, feature, threshold)
        left_child = self.build_tree(X_left, y_left, depth + 1)
        right_child = self.build_tree(X_right, y_right, depth + 1)
        return Node(feature=feature, threshold=threshold, left=left_child, right=right_child)
    def fit(self, X, y):
        self.tree = self.build_tree(X, y)
    def predict_one(self, x, node):
        if node.value is not None:
            return node.value
        if x[node.feature] <= node.threshold:
            return self.predict_one(x, node.left)
        else:
            return self.predict_one(x, node.right)
    def predict(self, X):
        return [self.predict_one(x, self.tree) for x in X]
if __name__ == "__main__":
    X = np.array([[2, 3],
```

```
            [1, 1],
            [3, 6],
            [6, 7],
            [7, 2],
            [8, 4]])

y = np.array([0, 0, 1, 1, 0, 1])  # Target labels
clf = DecisionTree(max_depth=3)
clf.fit(X, y)
predictions = clf.predict(X)
print("Predictions:", predictions)
```

Output:

```
Predictions: [np.int64(0), np.int64(0), np.int64(0), np.int64(0), np.int64(0), np.int64(0)]
```

Result:

Thus Program was executed Successfully.

8 Implementation of k-means algorithm

Aim:

The aim of this implementation is to create a k-means algorithm to cluster a set of data pointsinto distinct groups based on their features, minimizing the variance within each group.

Procedure:

Step 1: Start the Program.

Step 2: Initialization: Choose KKK, the number of clusters and randomly select KKK initial centroids from the data points.

Step 3: Assignment Step: For each data point, assign it to the nearest centroid based on the Euclidean distance.

Step 4: **Update Step**: Recalculate the centroids by taking the mean of all data points assigned to each cluster.

Step 5: **Convergence Check**: Repeat the Assignment and Update steps until the centroids do not change significantly or a maximum number of iterations is reached.

Step 6: Stop the Program.

Algorithm:

```
import numpy as np

class KMeans:

    def __init__(self, k=2, max_iters=100):

        self.k = k

        self.max_iters = max_ iterations

        self.centroids = None        centroids

    def fit(self, X):

        np.random.seed(42)

        self.centroids =

X[np.random.choice(range(len(X)), self.k,
```

```
replace=False)]

        for _ in range(self.max_iters):
            centroid
            labels = self.assign_clusters(X)
            new_centroids = np.array([X[labels == i].mean(axis=0) for i in range(self.k)])
            if np.all(new_centroids == self.centroids):
                break
            self.centroids = new_centroids
    def assign_clusters(self, X):
        distances = np.array([np.linalg.norm(X - centroid, axis=1) for centroid in self.centroids])
        return np.argmin(distances, axis=0)
    def predict(self, X):
        return self.assign_clusters(X)
if __name__ == "__main__":
    X = np.array([[1, 2], [2, 3], [3, 4], [8, 7], [9, 8], [10, 9]])
    kmeans = KMeans(k=2)
    kmeans.fit(X)
     labels = kmeans.predict(X)
       print("Cluster labels:", labels)
    print("Centroids:", kmeans.centroids)
```

Output:

```
Cluster labels: [0 0 0 1 1 1]
Centroids: [[2. 3.]
```

Result:

Thus the Program was executed Successfully.

9 Implementation of Id3 Algorithm

Aim:

The aim of this implementation to build a decision tree for classification tasks by using the ID3 algorithm.

Procedure:

Step 1: Start the Program.

Step 2: Initialization: Start with the entire dataset and a set of features.

Step 3: Check for Stopping Criteria: If all instances belong to the same class, return a leaf node with that class.If no features are left, return a leaf node with the majority class.

Step 4: **Select Attribute**: Calculate the information gain for each feature.Choose the feature with the highest information gain to split the dataset.

Step 5: **Split the Dataset**: Partition the dataset based on the selected attribute.

Step 6: Recursive Construction: Repeat steps 2-4 for each partitioned subset of the dataset until the stopping criteria are met.

Step 7: Output the Tree: Return the constructed decision tree.

Step 8: Stop the Program.

Algorithm:

```
import numpy as np
import pandas as pd


class Node:
    def __init__(self, feature=None, value=None, left=None, right=None, output=None):
        self.feature = feature
        self.value = value
        self.left = left
        self.right = right
        self.output = output


class ID3:
    def __init__(self):
```

```
        self.tree = None

    def entropy(self, y):
        """Calculate the entropy of the target variable."""
        value_counts = y.value_counts()
        probabilities = value_counts / len(y)
        return -np.sum(probabilities * np.log2(probabilities + 1e-9))
    def information_gain(self, X, y, feature):
        """Calculate the information gain for a feature."""
        total_entropy = self.entropy(y)
        values = X[feature].unique()
        weighted_entropy = 0
        for value in values:
            subset = y[X[feature] == value]
            weighted_entropy += (len(subset) / len(y)) * self.entropy(subset)
        return total_entropy - weighted_entropy
    def best_feature(self, X, y):
        """Select the best feature to split on based on information gain."""
        gains = {feature: self.information_gain(X, y, feature) for feature in X.columns}
        return max(gains, key=gains.get)
    def build_tree(self, X, y):
        """Recursively build the decision tree."""
        if len(y.unique()) == 1:
            return Node(output=y.iloc[0])
        if X.empty:
            return Node(output=y.mode()[0])
        best_feat = self.best_feature(X, y)
        tree = Node(feature=best_feat)
        for value in X[best_feat].unique():
            subset = X[X[best_feat] == value]
            target_subset = y[X[best_feat] == value]
            subtree = self.build_tree(subset.drop(columns=[best_feat]), target_subset)
            if value == 1:
```

```
            tree.right = subtree  # Right branch for 1
        else:
            tree.left = subtree   # Left branch for 0
    return tree
def fit(self, X, y):
    """Fit the ID3 model to the data."""
    self.tree = self.build_tree(X, y)
def predict_one(self, node, x):
    """Make a prediction for a single instance."""
    if node.output is not None:
        return node.output
    if x[node.feature] == 1:
        return self.predict_one(node.right, x)
    else:
        return self.predict_one(node.left, x)
def predict(self, X):
    """Make predictions for a DataFrame."""
    return X.apply(lambda x: self.predict_one(self.tree, x), axis=1)
if __name__ == "__main__":
  data = {
      'Outlook': ['Sunny', 'Sunny', 'Overcast', 'Rainy', 'Rainy', 'Rainy', 'Overcast', 'Sunny',
                  'Sunny', 'Rainy', 'Sunny', 'Overcast', 'Overcast', 'Rainy'],
      'Temperature': ['Hot', 'Hot', 'Hot', 'Mild', 'Cool', 'Cool', 'Mild', 'Mild', 'Hot', 'Mild',
                                    'Cool', 'Mild', 'Cool', 'Cool'],
      'Humidity': ['High', 'High', 'High', 'Normal', 'Normal', 'Normal', 'High', 'High',
                  'Normal', 'Normal', 'Normal', 'High', 'Normal', 'High'],
      'Windy': [False, True, False, False, False, True, True, False, False, False, True,
                  True, False, True],
      'Play': ['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes',
                  'No']
          }

    df = pd.DataFrame(data)
```

```
X = df[['Outlook', 'Temperature', 'Humidity', 'Windy']]
y = df['Play']
X = pd.get_dummies(X, drop_first=True)

id3 = ID3()
id3.fit(X, y)

predictions = id3.predict(X)
print("Predictions:")
print(predictions.tolist())
```

Output:

```
Predictions:
['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'No']
```

Result:

Thus Program was executed Successfully.

10 Neural Network Implementation

Aim:

To implement a basic artificial neural network (ANN) for classification tasks using Python,leveraging libraries.

Algorithm:

Step 1: Start the program

Step 2: Import necessary libraries such as TensorFlow, Keras, NumPy, etc.

Step 3: Design Neural Network Architecture define input layer based on the features.

Step 4: Insert one or more hidden layers with activation functions.

Step 5: Compile the program an optimizer loss function and evaluation metrics.

Step 6: Train the neural network using the training data for a specified number of epochs.

Step 7: Evaluate the model on test data and make predictions for unseen data.

Step 8: Stop the program.

Program Code:

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x):
    return x * (1 - x)
class NeuralNetwork:

    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.weights_input_hidden = np.random.rand(self.input_size, self.hidden_size)
        self.weights_hidden_output = np.random.rand(self.hidden_size, self.output_size)

    def forward(self, X):
        self.hidden_layer_input = np.dot(X, self.weights_input_hidden)
        self.hidden_layer_output = sigmoid(self.hidden_layer_input)
        self.output_layer_input = np.dot(self.hidden_layer_output, self.weights_hidden_output)
        self.output = sigmoid(self.output_layer_input)
```

```
        return self.output

    def backward(self, X, y, learning_rate=0.1):
        output_error = y - self.output  # Error in output
        output_delta = output_error * sigmoid_derivative(self.output)
        hidden_layer_error = output_delta.dot(self.weights_hidden_output.T)
        hidden_layer_delta = hidden_layer_error *
sigmoid_derivative(self.hidden_layer_output)

        self.weights_hidden_output += self.hidden_layer_output.T.dot(output_delta) *
learning_rate
        self.weights_input_hidden += X.T.dot(hidden_layer_delta) * learning_rate

    def train(self, X, y, epochs=10000):
        for _ in range(epochs):
            self.forward(X)
            self.backward(X, y)

if __name__ == "__main__":
    X = np.array([[0, 0],
                  [0, 1],
                  [1, 0],
                  [1, 1]])
    y = np.array([[0],
                  [1],
                  [1],
                  [0]])
    nn = NeuralNetwork(input_size=2, hidden_size=2, output_size=1)
    nn.train(X, y)
    output = nn.forward(X)
    print("Predicted output after training:")
    print(output)
```

Output:

```
Predicted output after training:
[[0.29641702]
 [0.66326077]
 [0.66319084]
 [0.43232419]]
```

RESULT:

Thus the above program was executed successfully and verified.

11 Implementation of Multilayer Neural Network

Aim:

To implement a Multilayer Perceptron Neural Network (MLP) using Python for solving aclassification problem.

Algorithm:

Step 1: Start the program

Step 2: Input the data into the input layer.

Step 3: Calculate the weighted sum of the inputs and the bias for each neuron.

Step 4: Repeat the process for the output layer using the outputs from hidden layer.

Step 5: Update the weights and biases for each layer by applying the computed gradients.

Step 6: Once training complete, input new data into the network and evaluate itspredictions using trained weights.

Step 7: Output the predicted values for the test data.

Step 8: Stop the program.

Program Code:

```
import numpy as np
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x):
    return x * (1 - x)
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        self.weights_input_hidden = np.random.randn(input_size, hidden_size)
        self.weights_hidden_output = np.random.randn(hidden_size, output_size)
    def forward(self, X):
```

```
        self.hidden_input = np.dot(X, self.weights_input_hidden)
        self.hidden_output = sigmoid(self.hidden_input)
        self.output_input = np.dot(self.hidden_output, self.weights_hidden_output)
        self.output = sigmoid(self.output_input)
        return self.output
    def backward(self, X, y, learning_rate=0.1):
        output_error = y - self.output
        output_delta = output_error * sigmoid_derivative(self.output)
        hidden_error = output_delta.dot(self.weights_hidden_output.T)
        hidden_delta = hidden_error * sigmoid_derivative(self.hidden_output)
        self.weights_hidden_output += self.hidden_output.T.dot(output_delta) * learning_rate
        self.weights_input_hidden += X.T.dot(hidden_delta) * learning_rate
    def train(self, X, y, iterations):
        for _ in range(iterations):
            self.forward(X)
            self.backward(X, y)
if __name__ == "__main__":
    X = np.array([[0, 0],
                  [0, 1],
                  [1, 0],
                  [1, 1]])
    y = np.array([[0], [1], [1], [0]])
    nn = NeuralNetwork(input_size=2, hidden_size=2, output_size=1)
    nn.train(X, y, iterations=10000)
    output = nn.forward(X)
    print("Predicted output:")
    print(output)
```

Output:

Predicted output:

[[0.07115026]

[0.65565445]

[0.65568419]

[0.66539303]]

RESULT:

Thus the above program was executed successfully and verified.

12 Back Propagation and Genetic Algorithm Implementation

Aim:

To implement a Back propagation and genetic algorithm implementation using pythonlibraries.

Algorithm:

Step 1: Start the program

Step 2: Initialize the Neural Network randomly set the weights and biases.

Step 3: Create a population of potential solutions, where each individual represents a set ofneural network weights.

Step 4: For each individual in the population, use the neural network with the weightsrepresented by that chromosome.

Step 5: Apply random mutations to the offspring to maintain genetic diversity.

Step 6: Replace the old population with new offspring generated from crossover and mutation.

Step 7: Repeat steps 3-8 for a set number of generations or until the neural network reaches an acceptable performance level.

Step 8: Stop the program.

Program Code:

Backpropagation:

```
import numpy as np
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x):
    return x * (1 - x)
class NeuralNetwork:
    def __init__(self):
        self.weights = np.random.randn(2, 1
```

```
    def forward(self, X):
        self.input = X
        self.output = sigmoid(np.dot(self.input, self.weights))
        return self.output
    def backward(self, y, learning_rate=0.1):
        error = y - self.output
        delta = error * sigmoid_derivative(self.output)
        self.weights += np.dot(self.input.T, delta) * learning_rate
    def train(self, X, y, iterations=10000):
        for _ in range(iterations):
            self.forward(X)
            self.backward(y)
if __name__ == "__main__":
    X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    y = np.array([[0], [1], [1], [0]])
    nn = NeuralNetwork()
    nn.train(X, y)
    output = nn.forward(X)
    print("Predicted output:")
    print(output)
```

Output:

```
Predicted output:
[[0.5]
 [0.5]
 [0.5]
 [0.5]]
```

Genetic Algorithm:

```
import numpy as np

def fitness(individual, target):
    return np.sum(individual == target)

def crossover(parent1, parent2):
    point = np.random.randint(1, len(parent1) - 1)
    child1 = np.concatenate((parent1[:point], parent2[point:]))
    child2 = np.concatenate((parent2[:point], parent1[point:]))
    return child1, child2

def mutate(individual, mutation_rate=0.01):
    for i in range(len(individual)):
        if np.random.rand() < mutation_rate:
            individual[i] = 1 - individual[i]  # Flip bit

def genetic_algorithm(target_str, population_size=100, generations=1000,
mutation_rate=0.01):
    target = np.array([int(bit) for bit in target_str])  # Convert target string to array
    chromosome_length = len(target)
    population = np.random.randint(2, size=(population_size, chromosome_length))
    for generation in range(generations):
        fitness_scores = np.array([fitness(ind, target) for ind in population])
        if np.max(fitness_scores) == chromosome_length:
            best_match_idx = np.argmax(fitness_scores)
            print(f"Target reached in generation {generation}!")
            return population[best_match_idx]
        probabilities = fitness_scores / fitness_scores.sum()
        selected_idx = np.random.choice(range(population_size), size=population_size//2,
p=probabilities)
        parents = population[selected_idx]
```

```
        next_population = []
        for i in range(0, len(parents), 2):
            parent1, parent2 = parents[i], parents[i+1]
            child1, child2 = crossover(parent1, parent2)
            mutate(child1, mutation_rate)
            mutate(child2, mutation_rate)
            next_population.extend([child1, child2])
        population = np.array(next_population)
    best_match_idx = np.argmax(fitness_scores)
    print(f"Best match after {generations} generations:")
    return population[best_match_idx]
if __name__ == "__main__":
    target = "1010101010"  # Target binary string
    best_solution = genetic_algorithm(target)
    print(f"Best solution found: {''.join(map(str, best_solution))}")
```

Output:

Target reached in generation 1!

Best solution found: 1010101010

RESULT:

Thus the above program was executed successfully and verified.