EX. NO : 1 SIMPLE ML - PROGRAM – LINEAR REGRESSION

Date :14.07.2025

Aim:

To implement a simple linear regression model from scratch using Python, visualize the data points and regression line, and predict the value of y for a given value of x.

Procedure:

1. Import necessary libraries
   Import matplotlib.pyplot for plotting graphs.
2. Prepare the data
   Use sample data points for x and y.
3. Calculate means
   Calculate the mean of x values and mean of y values.
4. Calculate slope (m)
   Use the formula for slope m:

$$m = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

5. Calculate intercept (c) Use the formula:

   $c=\bar{y}-m\cdot\bar{x}$

6. Form the linear equation
   Print the equation of the line y=mx+c
7. Predict y for a new value x=
   Use the linear equation to predict y.
8. Plot the data points and regression line
   - Plot original data points as scatter plot.
   - Plot the regression line.
   - Mark the predicted point on the graph.
9. Display the plot
   Use plt.show() to visualize the results.

Program Coding:

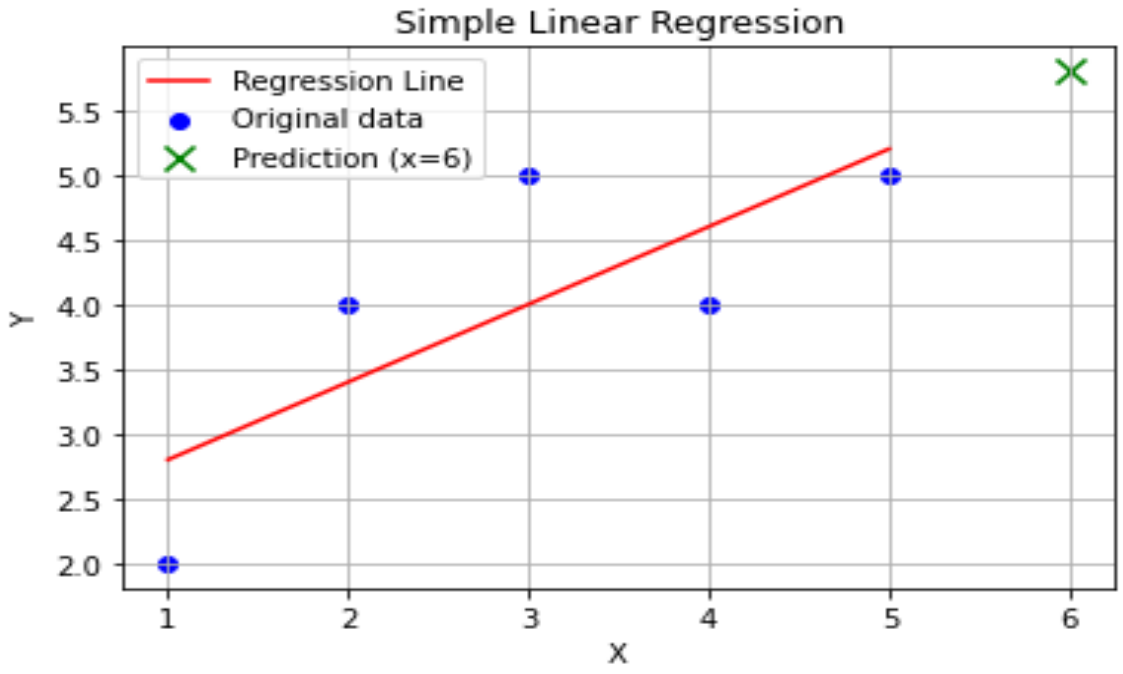
```
import matplotlib.pyplot as plt

# Sample data

x = [1, 2, 3, 4, 5]

y = [2, 4, 5, 4, 5]

# Calculate means

mean_x = sum(x) / len(x)

mean_y = sum(y) / len(y)

# Calculate slope (m) and intercept (c)

numerator = sum((x[i] - mean_x) * (y[i] - mean_y) for i in range(len(x)))

denominator = sum((x[i] - mean_x)**2 for i in range(len(x)))

m = numerator / denominator

c = mean_y - m * mean_x

# Print equation

print(f"Equation of line: y = {m:.2f}x + {c:.2f}")

# Predict y for x = 6

x_test = 6

y_pred = m * x_test + c

print(f"Predicted y for x = {x_test} is {y_pred:.2f}")

# Plot original data

plt.scatter(x, y, color='blue', label='Original data')

# Plot regression line

regression_line = [m * xi + c for xi in x]
```

```
plt.plot(x, regression_line, color='red', label='Regression Line')

# Plot predicted point

plt.scatter(x_test, y_pred, color='green', marker='x', s=100, label=f'Prediction (x={x_test})')

plt.title("Simple Linear Regression")

plt.xlabel("X")

plt.ylabel("Y")

plt.legend()

plt.grid(True)

plt.show()
```

Output:

Linear Regression Equation: y = 0.60x + 2.20

Predicted y for x = 6 is 5.80



Result:

Thus, the simple linear regression program was successfully implemented.

EX. NO : 2a **Tic Tac Toe Game**

Date :21.07.2025

Aim:

To develop a console-based 2-player Tic Tac Toe game in Python, where two players alternate turns to place their marks (X and O) on a 3x3 board, and the program determines the winner or a draw.

Procedure:

1. Initialize the board:
   Create a 3x3 matrix (list of lists) filled with empty spaces " ".
2. Display the board:
   Implement a function to print the current state of the board after every move, showing the grid clearly.
3. Take input from players:
   Alternate between Player X and Player O, asking for row and column inputs (0 to 2).
4. Validate moves:
   Check if the selected cell is empty; if not, prompt the player to try again.
5. Update the board:
   Place the player's mark in the selected cell.
6. Check for a win:
   After each move, check rows, columns, and diagonals for three identical marks.
7. Declare results:
   - If a player wins, display the winner and end the game.
   - If all 9 moves are made without a winner, declare the game a draw.

Program Coding:

```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 5)
def check_win(board, player):
```

```
    # Rows, columns, diagonals
    for i in range(3):
        if all([cell == player for cell in board[i]]): return True
        if all([board[j][i] == player for j in range(3)]): return True
    if all([board[i][i] == player for i in range(3)]): return True
    if all([board[i][2 - i] == player for i in range(3)]): return True
    return False
def play_game():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current = "X"

    for turn in range(9):
        print_board(board)
        row = int(input(f"Player {current}, enter row (0-2): "))
        col = int(input(f"Player {current}, enter col (0-2): "))
            if board[row][col] != " ":
            print("Cell taken! Try again.")
            continue

        board[row][col] = current
        if check_win(board, current):
            print_board(board)
            print(f"Player {current} wins!")
            return
        current = "O" if current == "X" else "X"
        print_board(board)
```

```
    print("It's a draw!")

play_game()
```

Output:

- A 3×3 board is created.
- Two players ("X" and "O") take turns.
- The game checks for a win or draw after each move.

```
Player X, enter row (0-2): 0
Player X, enter col (0-2): 0
X |   |
-----
  |   |
-----
  |   |
Player O, enter row (0-2): 1
Player O, enter col (0-2): 1

X |   |
-----
  | O |
-----
  |   |
Player X, enter row (0-2): 0
Player X, enter col (0-2): 1
X | X |
```

| O |

| |

Player O, enter row (0-2): 2

Player O, enter col (0-2): 2

X | X |

| O |

| | O

Player X, enter row (0-2): 0

Player X, enter col (0-2): 2

X | X | X

| O |

| | O

✅ Player X wins!

Result:

Thus, the Tic Tac Toe game program was successfully implemented

EX. NO: 2b **Traveling Salesman Problem (Brute Force)**

Date :04.08.2025

Aim:

To implement the Traveling Salesman Problem (TSP) using a brute force method in Python to find the shortest possible route that visits each city exactly once and returns to the origin city.

Procedure:

1. Understand the problem:
   The TSP requires finding the minimum-cost Hamiltonian circuit (a route visiting every node once and returning to start).
2. Represent the graph:
   Use a dictionary of dictionaries to represent the graph where keys are nodes and values are dictionaries of neighboring nodes with travel costs.
3. Generate permutations:
   Use Python's itertools.permutations to generate all possible sequences of visiting the nodes.
4. Calculate cost for each route:
   For each permutation, sum the travel costs between consecutive nodes plus the return trip to the starting node.
5. Track the minimum cost and path:
   Compare each permutation's total cost to find and store the minimum cost and corresponding path.
6. Output the result:
   Display the shortest path and its total cost.

Program Coding:

```
from itertools import permutations

def tsp_brute_force(graph):

    nodes = list(graph.keys())
```

```
    min_path = None
    min_cost = float('inf')
    for perm in permutations(nodes):
        cost = 0
        for i in range(len(perm) - 1):
            cost += graph[perm[i]][perm[i + 1]]
        cost += graph[perm[-1]][perm[0]]  # return to start
            if cost < min_cost:
            min_cost = cost
            min_path = perm
    return min_path + (min_path[0],), min_cost
# Example graph
graph = {
    'A': {'A': 0, 'B': 10, 'C': 15, 'D': 20},
    'B': {'A': 10, 'B': 0, 'C': 35, 'D': 25},
    'C': {'A': 15, 'B': 35, 'C': 0, 'D': 30},
    'D': {'A': 20, 'B': 25, 'C': 30, 'D': 0}
}path, cost = tsp_brute_force(graph)
print("Shortest path:", path)
print("Total cost:", cost)
```

Output:

 Given a distance matrix (graph), it tries all permutations of the cities.

 Returns the shortest path and total distance

Shortest path: ('A', 'B', 'D', 'C', 'A')

Total cost: 80

Result:

Thus, the Traveling Salesman Problem (TSP) using the brute force approach was successfully implemented.

EX. NO: 3 **Simple AI Agent (Rule-based)**

Date: 21.07.2025

Aim:

To create a simple rule-based AI agent in Python that responds to user inputs with predefined replies based on keyword matching

Procedure:

1. Design the AI logic:
   Use rule-based (reflex) logic where specific keywords or phrases in user input trigger predefined responses.
2. Normalize user input:
   Convert the user's input to lowercase to simplify matching rules regardless of capitalization.
3. Define response rules:
   Create conditional checks (if-elif-else) for key phrases such as greetings, asking the agent's name, weather queries, and exit commands.
4. Implement a main loop:
   Continuously prompt the user for input, pass it to the AI agent function, and print the response.
5. Exit mechanism:
   Allow the user to type 'bye' to end the conversation and exit the program.

```
def simple_ai_agent(user_input):
    # Convert input to lowercase for easy matching
    user_input = user_input.lower()
    # Define rules (reflex-based logic)
    if "hello" in user_input or "hi" in user_input:
        return "Hello! How can I help you today?"
    elif "your name" in user_input:
        return "I'm a simple AI agent created in Python."
    elif "weather" in user_input:
```

```
        return "I can't check real-time weather, but I hope it's nice where you are!"
    elif "bye" in user_input:
        return "Goodbye! Have a great day!"
    else:
        return "I'm not sure how to respond to that."
# Main loop
print("Welcome to the Simple AI Agent. Type 'bye' to exit.")
while True:
    user_input = input("You: ")
    response = simple_ai_agent(user_input)
    print("AI: " + response)
    if "bye" in user_input.lower():
        break
```

Output:

```
Welcome to the Simple AI Agent. Type 'bye' to exit.
You: Hi
AI: Hello! How can I help you today?
You: What's your name?
AI: I'm a simple AI agent created in Python.
You: How's the weather?
AI: I can't check real-time weather, but I hope it's nice where you are!
You: Tell me a joke.
AI: I'm not sure how to respond to that.
You: bye
AI: Goodbye! Have a great day!
```

Result:

Thus, the simple AI agent was successfully implemented using a rule-based approach.

EX. NO: 4 **Simple Knowledge-Based AI System**

Date: 04.08.2025

Aim:

To develop a simple knowledge-based AI system in Python that diagnoses possible illnesses based on user-reported symptoms using predefined rules.

Procedure:

1. Design the knowledge base:
   Define a set of rules that associate specific symptom combinations with possible diagnoses.
2. Collect user input:
   Prompt the user to answer a series of yes/no questions regarding common symptoms.
3. Store symptoms:
   Record symptoms for which the user answers "yes".
4. Apply rules to diagnose:
   Match the collected symptoms against the knowledge base to infer a diagnosis.
5. Display the diagnosis:
   Show the inferred condition or advise consulting a doctor if symptoms do not match known patterns.

Program Coding:

```
def diagnose(symptoms):
    # Knowledge base: rules
    if "fever" in symptoms and "cough" in symptoms and "fatigue" in symptoms:
        return "You might have the flu."
    elif "headache" in symptoms and "nausea" in symptoms:
        return "You might have a migraine."
    elif "sore throat" in symptoms and "fever" in symptoms:
        return "You might have a throat infection."
    else:
```

```
        return "Diagnosis unclear. Please consult a doctor."
# Get input from user
print("Welcome to the Simple Medical Diagnosis System.")
print("Please answer the following questions with 'yes' or 'no'.")
# Collect user symptoms
symptoms = []
questions = {
    "fever": "Do you have a fever?",
    "cough": "Do you have a cough?",
    "fatigue": "Are you feeling tired or fatigued?",
    "headache": "Do you have a headache?",
    "nausea": "Are you feeling nauseous?",
    "sore throat": "Do you have a sore throat?"
}
for symptom, question in questions.items():
    answer = input(question + " ").strip().lower()
    if answer == "yes":
        symptoms.append(symptom)
# Run diagnosis
result = diagnose(symptoms)
print("\nDiagnosis:", result)
```

Output:

```
Welcome to the Simple Medical Diagnosis System.
Please answer the following questions with 'yes' or 'no'.
Do you have a fever? yes
Do you have a cough? yes
Do you have a headache? no
Do you have a sore throat? no
Do you have nausea? no
Do you have fatigue? yes
Diagnosis: You might have the flu.
```

Result:

Thus, the simple knowledge-based AI system was successfully implemented.

EX. NO: 5 Implementations of Ontology with FOL

Date :15.09.2025

Aim:

To implement a simple First Order Logic with Ontology Program.

Procedure :

To Create a First-Order Logic + Ontology Program in Python

Step 1: Define Your Ontology Components

Identify Classes (concepts) — e.g., Human, Animal, Mortal.

Identify Individuals (instances) — e.g., Socrates, Doggo.

Identify Relationships between individuals — e.g., hasPet.

Step 2: Represent Ontology in Python Data Structures

Use dictionaries or sets to hold classes and their members.

Use a dictionary to map individuals to their classes.

Use nested dictionaries for relationships.

Step 3: Define First-Order Logic Rules as Python Functions

Each rule represents a logical statement you want to enforce.

Example:Rule: *All humans are mortal* → For every individual who is a Human, add them to Mortal.

Step 4: Implement Reasoning Procedure

Write a main function that:

Prints initial ontology info.

Applies inference rules.

Prints results after reasoning.

Displays relationships.

Step 5: Run and Test

Execute the program.

Verify that the inference works (e.g., Socrates inferred as Mortal).

Check the output matches expectati

Step 6 (Optional): Extend Ontology or Add More Rules

Add more classes, individuals, and relationships.

Implement more complex rules, e.g., “If x has a pet y, and y is an animal, then...”

Use libraries like owlready2 for richer ontologies and reasoning support.

Program Coding:

```
# Ontology
classes = {
    "Human": set(),
    "Animal": set(),
    "Mortal": set()
}


individuals = {
    "Socrates": "Human",
    "Doggo": "Animal"
```

```
}


relationships = {
    "hasPet": {
        "Socrates": "Doggo"
    }
}


# First-order logic rules (implemented as functions)
# Rule 1: All humans are mortal
def infer_mortality():
    for person, person_type in individuals.items():
        if person_type == "Human":
            classes["Mortal"].add(person)
# Rule 2: If a human has a pet, the pet is an animal (already known in individuals)
# Rule 3: Print relationships
def print_relationships():
    print("\n--- Relationships ---")
    for subject, obj in relationships["hasPet"].items():
        print(f"{subject} has a pet named {obj}.")
# Reasoning
def main():
```

```
    print("--- Ontology Classes ---")
    for cls in classes:
        print(f"{cls}: {classes[cls]}")

    print("\n--- Individuals ---")
    for ind, cls in individuals.items():
        print(f"{ind} is a {cls}")

    # Apply rule: all humans are mortal
    infer_mortality()
    print("\n--- After Reasoning ---")
    print("Mortal:", classes["Mortal"])

    # Show relationships
    print_relationships()

# Run the program
if __name__ == "__main__":
    main()
```

Output:

--- Ontology Classes ---

Human: set()

Animal: set()

Mortal: set()

--- Individuals ---

Socrates is a Human

Doggo is a Animal

--- After Reasoning ---

Mortal: {'Socrates'}

--- Relationships ---

Socrates has a pet named Doggo.

Result:

The program successfully inferred that Socrates is mortal based on the rule "All humans are mortal" and displayed the existing relationship that Socrates has a pet named Doggo.

EX. NO : 6 Concept Learning & Design a Learning system- Find-S Algorithm

Date :22.09.2025

Aim:

To implement the Find-S algorithm and Candidate Elimination algorithm in Python for concept learning, and to identify the most specific and most general hypotheses consistent with training data.

Procedure:

1. Understand the problem:
   Concept learning involves finding a hypothesis that fits positive training examples and excludes negative ones.
2. Prepare the training data:
   Use a dataset with multiple attributes and a target concept label (e.g., “Yes” or “No”).
3. Implement Find-S algorithm:
   - Initialize the hypothesis to the first positive example.
   - Generalize the hypothesis only as needed to accommodate all positive examples by replacing differing attribute values with '?'.
4. Define helper functions for generalization and specialization:
   - Check if one hypothesis is more general than another.
   - Generalize the specific boundary (S) to include new positive examples.
   - Specialize the general boundary (G) to exclude negative examples.
5. Implement Candidate Elimination:
   - Initialize S to the first positive example and G to the maximally general hypothesis.
   - For each example, update S and G boundaries accordingly:
     - Generalize S when a positive example is encountered.
     - Specialize G when a negative example is encountered.
6. Extract and print the final specific (S) and general (G) hypotheses.

Program Coding:

```
def train_find_s(data):

  # Initialize hypothesis with the first positive example
```

```
    for example in data:
        if example[-1] == "Yes":
            hypothesis = example[:-1]
            break
    # Compare other positive examples
    for example in data:
        if example[-1] == "Yes":
            for i in range(len(hypothesis)):
                if hypothesis[i] != example[i]:
                    hypothesis[i] = "?"
    return hypothesis
# Training data (attribute1, attribute2, ..., PlayTennis)
dataset = [
    ["Sunny", "Warm", "Normal", "Strong", "Warm", "Same", "Yes"],
    ["Sunny", "Warm", "High", "Strong", "Warm", "Same", "Yes"],
    ["Rainy", "Cold", "High", "Strong", "Warm", "Change", "No"],
    ["Sunny", "Warm", "High", "Strong", "Cool", "Change", "Yes"]
]
# Run Find-S algorithm
final_hypothesis = train_find_s(dataset)
# Output the final hypothesis
print("Final Hypothesis:")
print(final_hypothesis)
```

```
Final Hypothesis:
['Sunny', 'Warm', '?', 'Strong', '?', '?']
```

```
def more_general(h1, h2):
```

```
    """Check if h1 is more general than h2."""
    more_general_parts = []
    for x, y in zip(h1, h2):
        mg = x == '?' or (x != '0' and (x == y or y == '0'))
        more_general_parts.append(mg)
    return all(more_general_parts)
def generalize_S(example, S):
    """Generalize S to include the positive example."""
    new_S = list(S)
    for i in range(len(S)):
        if S[i] != example[i]:
            new_S[i] = '?'
    return tuple(new_S)


def specialize_G(example, G, S):
    """Specialize G to exclude the negative example."""
    new_G = set()
    for g in G:
        if not more_general(g, example):
            continue
        for i in range(len(g)):
            if g[i] == '?':
                for val in attributes[i]:
                    if example[i] != val:
                        new_hypothesis = list(g)
                        new_hypothesis[i] = val
```

```
                    new_G.add(tuple(new_hypothesis))
            elif g[i] != example[i]:
                continue
    # Remove hypotheses that are more specific than S or not general enough
    return {h for h in new_G if more_general(h, S)}

# Training dataset: [attributes..., class]
dataset = [
    ["Sunny", "Warm", "Normal", "Strong", "Warm", "Same", "Yes"],
    ["Sunny", "Warm", "High", "Strong", "Warm", "Same", "Yes"],
    ["Rainy", "Cold", "High", "Strong", "Warm", "Change", "No"],
    ["Sunny", "Warm", "High", "Weak", "Cool", "Change", "Yes"]
]

# Extract attributes values
attributes = [set() for _ in range(len(dataset[0]) - 1)]
for row in dataset:
    for i, val in enumerate(row[:-1]):
        attributes[i].add(val)

# Initial boundaries
num_attributes = len(dataset[0]) - 1
S = tuple(dataset[0][:-1])  # First positive example
G = {tuple(['?'] * num_attributes)}

# Candidate Elimination
```

```
for row in dataset:
    x, label = row[:-1], row[-1]
    if label == "Yes":  # Positive example
        # Remove from G anything inconsistent with x
        G = {g for g in G if all(g[i] == x[i] or g[i] == '?' for i in range(num_attributes))}
        S = generalize_S(x, S)
    else:  # Negative example
        # Remove from S anything inconsistent with x
        G =specialize_G(x, G, S)
# Final Hypotheses
print("Final Specific Hypothesis S:")
print(S)
print("\nFinal General Hypotheses G:")
for g in G:
    print(g)
```

Output:

```
Final Specific Hypothesis S:
('Sunny', 'Warm', '?', '?', '?', '?')


Final General Hypotheses G:
('Sunny', 'Warm', '?', '?', '?', '?')
```

Result:

Thus, the Find-S and Candidate Elimination algorithms were successfully implemente

EX. NO : 7 Decision Tree implementation with K – mean Algorithm

Date :29.09.2025

Aim:

To implement the K-Means clustering algorithm in Python to partition a given dataset into K clusters by iteratively updating centroids and cluster assignments.

Procedure:

1. Initialize centroids:
   Randomly select K data points from the dataset as the initial centroids.
2. Assign data points to clusters:
   For each data point, compute the Euclidean distance to each centroid and assign it to the closest centroid's cluster.
3. Update centroids:
   For each cluster, calculate the mean of all points assigned to it to obtain the new centroid.
4. Check for convergence:
   Repeat steps 2 and 3 until the centroids no longer change or until a maximum number of iterations is reached.
5. Output the final centroids and clusters.

Program Coding:

```
import random

import math


# Sample data points

data = [

   [1, 2], [2, 3], [3, 1],

   [8, 9], [9, 10], [10, 8]

]
```

```
# Number of clusters
K = 2
# Randomly initialize K centroids from the data
centroids = random.sample(data, K)


# Euclidean distance function
def euclidean(p1, p2):
    return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)


# K-Means algorithm
for iteration in range(10):  # limit to 10 iterations
    clusters = [[] for _ in range(K)]
    # Assign points to nearest centroid
    for point in data:
        distances = [euclidean(point, centroid) for centroid in centroids]
        closest_index = distances.index(min(distances))
        clusters[closest_index].append(point)
    # Update centroids
    new_centroids = []
    for cluster in clusters:
        if cluster:  # avoid division by zero
            x_coords = [p[0] for p in cluster]
            y_coords = [p[1] for p in cluster]
            new_centroid = [sum(x_coords)/len(x_coords), sum(y_coords)/len(y_coords)]
            new_centroids.append(new_centroid)
```

```
        else:
            new_centroids.append(random.choice(data))  # reassign randomly if empty
    # Check for convergence
    if new_centroids == centroids:
        break
    else:
        centroids = new_centroids
# Output results
print("Final centroids:")
for c in centroids:
    print(c)
print("\nClusters:")
for i, cluster in enumerate(clusters):
    print(f"Cluster {i+1}: {cluster}")
```

Output:

Final centroids:

[2.0, 2.0]

[9.0, 9.0]

Clusters:

Cluster 1: [[1, 2], [2, 3], [3, 1]]

Cluster 2: [[8, 9], [9, 10], [10, 8]]

Result:

Thus, the K-Means clustering algorithm was successfully implemented. The program clustered the given data points into two groups based on their Euclidean distance to the centroids, iteratively refining the centroids until convergence was achieved.

EX. NO : 8 **Simple Perceptron Implementation for AND gate**

Date :06.10.2025

Aim:

To implement a simple perceptron algorithm to learn the AND logic gate.

Procedure:

1. Initialize weights and bias to zero.
2. For each training example, calculate weighted sum and apply step activation function.
3. Calculate error as difference between expected and predicted output.
4. Update weights and bias using the perceptron learning rule.
5. Repeat for several epochs until convergence.
6. Test the trained perceptron on all input combinations.

Progrm Coding:

```
# Training data: inputs and labels
training_inputs = [
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
]
labels = [0, 0, 0, 1]
# Initialize weights and bias
weights = [0.0, 0.0]
bias = 0.0
# Learning rate
lr = 0.1
```

```
# Activation function (step function)
def step(x):
    return 1 if x >= 0 else 0
# Training loop
for epoch in range(10):  # Train for 10 epochs
    print(f"Epoch {epoch+1}")
    for inputs, label in zip(training_inputs, labels):
        # Weighted sum
        linear_output = sum(w * inp for w, inp in zip(weights, inputs)) + bias
        prediction = step(linear_output)
        # Update weights and bias
        error = label - prediction
        for i in range(len(weights)):
            weights[i] += lr * error * inputs[i]
        bias += lr * error
        print(f"  Input: {inputs}, Prediction: {prediction}, Error: {error}")
    print(f"  Weights: {weights}, Bias: {bias}\n")
# Test perceptron after training
print("Testing trained perceptron:")
for inputs in training_inputs:
    linear_output = sum(w * inp for w, inp in zip(weights, inputs)) + bias
    prediction = step(linear_output)
    print(f"Input: {inputs}, Output: {prediction}")
```

Output:

Epoch 1

Input: [0, 0], Prediction: 1, Error: -1

Input: [0, 1], Prediction: 0, Error: 0

Input: [1, 0], Prediction: 0, Error: 0

Input: [1, 1], Prediction: 0, Error: 1

Weights: [0.1, 0.0], Bias: 0.0

...Testing trained perceptron:

Input: [0, 0], Output: 0

Input: [0, 1], Output: 0

Input: [1, 0], Output: 0

Input: [1, 1], Output: 1

Result:

The perceptron successfully learned the AND gate logic function by adjusting weights and bias to correctly classify all inputs.

EX. NO : 9 **Multiple Neural Network**

Date :13.10.2025

Aim:

To implement a simple feedforward neural network with one hidden layer to solve the XOR problem.

Procedure:

1. Initialize weights and biases randomly for input-to-hidden and hidden-to-output layers.
2. Use sigmoid activation function and its derivative for non-linear transformations.
3. Perform forward pass: compute activations of hidden and output layers.
4. Compute error between network output and expected output.
5. Perform backpropagation to calculate gradients for output and hidden layers.
6. Update weights and biases using gradient descent.
7. Repeat for a fixed number of epochs to minimize error.
8. Test the trained network on all input combinations.

Program coding:

```
import math
import random
# Sigmoid activation and its derivative
def sigmoid(x):
    return 1 / (1 + math.exp(-x))
def sigmoid_derivative(x):
    return x * (1 - x)
# Training dataset: XOR problem
training_inputs = [
    [0, 0],
```

```
    [0, 1],
    [1, 0],
    [1, 1]
]

training_outputs = [
    [0],
    [1],
    [1],
    [0]
]

# Initialize network parameters
input_neurons = 2
hidden_neurons = 2
output_neurons = 1
learning_rate = 0.5
# Initialize weights randomly
random.seed(42)
weights_input_hidden = [[random.uniform(-1, 1) for _ in range(hidden_neurons)] for _ in range(input_neurons)]
weights_hidden_output = [random.uniform(-1, 1) for _ in range(hidden_neurons)]
# Initialize biases
bias_hidden = [random.uniform(-1, 1) for _ in range(hidden_neurons)]
bias_output = random.uniform(-1, 1)
# Training loop
```

```
for epoch in range(10000):
    total_error = 0
    for inputs, expected in zip(training_inputs, training_outputs):
        # ---Forward pass ---
        # Hidden layer activations
        hidden_layer_input = []
        for j in range(hidden_neurons):
            activation = bias_hidden[j]
            for i in range(input_neurons):
                activation += inputs[i] * weights_input_hidden[i][j]
            hidden_layer_input.append(sigmoid(activation))
        # Output layer activation
        output_activation = bias_output
        for j in range(hidden_neurons):
            output_activation += hidden_layer_input[j] * weights_hidden_output[j]
        output = sigmoid(output_activation)
        # --- Calculate error ---
        error = expected[0] - output
        total_error += error**2
        # --- Backward pass ---
        # Output layer delta
        delta_output = error * sigmoid_derivative(output)
        # Hidden layer deltas
        delta_hidden = []
        for j in range(hidden_neurons):
```

```
            delta = delta_output * weights_hidden_output[j] *
sigmoid_derivative(hidden_layer_input[j])
            delta_hidden.append(delta)
        # --- Update weights and biases ---
        # Update weights hidden → output
        for j in range(hidden_neurons):
            weights_hidden_output[j] += learning_rate * delta_output * hidden_layer_input[j]
        bias_output += learning_rate * delta_output
        # Update weights input → hidden
        for i in range(input_neurons):
            for j in range(hidden_neurons):
                weights_input_hidden[i][j] += learning_rate * delta_hidden[j] * inputs[i]
        for j in range(hidden_neurons):
            bias_hidden[j] += learning_rate * delta_hidden[j]
    # Print error every 1000 epochs
    if epoch % 1000 == 0:
        print(f"Epoch {epoch} - Error: {total_error:.4f}")
# Test trained network
print("\nTesting trained network:")
for inputs in training_inputs:
    hidden_layer_input = []
    for j in range(hidden_neurons):
        activation = bias_hidden[j]
        for i in range(input_neurons):
            activation += inputs[i] * weights_input_hidden[i][j]
        hidden_layer_input.append(sigmoid(activation))
```

```
    output_activation = bias_output
    for j in range(hidden_neurons):
        output_activation += hidden_layer_input[j] * weights_hidden_output[j]
    output = sigmoid(output_activation)
    print(f"Input: {inputs} => Output: {output:.4f}")
```

Output:

Epoch 0 - Error: 1.3576

Epoch 1000 - Error: 0.2632

Epoch 2000 - Error: 0.1235

Epoch 3000 - Error: 0.0739

Epoch 4000 - Error: 0.0492

Epoch 5000 - Error: 0.0364

Epoch 6000 - Error: 0.0281

Epoch 7000 - Error: 0.0224

Epoch 8000 - Error: 0.0183

Epoch 9000 - Error: 0.0150

Testing trained network:

Input: [0, 0] => Output: 0.0334

Input: [0, 1] => Output: 0.9660

Input: [1, 0] => Output: 0.9662

Input: [1, 1] => Output: 0.0376

Result:

The multi-layer neural network successfully learned the XOR function, achieving low error and producing outputs close to expected binary results.