A LAB MANUAL

on

ARTIFICIAL INTELLIGENCE LAB (U21CD5L1)

B.E. III Year I Semester

By

Mr. Md Anjar Ahsan
Assistant Professor
Dept. of CSE – Data Science

DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING (DATA SCIENCE)

LORDS INSTITUTE OF ENGINEERING & TECHNOLOGY
(UGC Autonomous)
Approved by AICTE | Affiliated to Osmania University | Estd.2003
Accredited by **NBA**, Accredited '**A**' Grade by NAAC
(Academic Year: 2025 – 2026)

INDEX

Vision of the Department:

To develop the world's next generation Data Scientist by offering top-notch education with cutting edge technologies.

Mission of the Department:

DM1: Providing students to understand the principles of Data Science with hands-on experience.

DM2: Offering students to analyze the data through effective teaching learning methods and cutting edge technologies in multi disciplinary fields.

DM3: Preparing students for R&D, Industrial design, entrepreneurship and employment.

DM4: Encouraging students with interaction and Industry Institute partnership through various organizations.

Note: DM: Department Mission

Head of the Department

Head of the Department
CSE (Data Science)
Lords Institute of Engg. & Tech.
Hyderabad-500091. T.S.

B.E. Computer Science and Engineering (Data Science) Program Educational Objectives (PEOs):

PEO1	Students will have strong foundation In Basic Science, Mathematics, Statistics, Computer Science and Allied engineering.
PEO2	Students will develop Data Science applications at advanced level for employability in IT industry especially in niche technologies.
PEO3	Students will be provided with strong foundation in Data Science and its applications for their carrier as Data Scientist/Data Engineers.

Head of the Department

Head of the Department
CSE (Data Science)
Lords Institute of Engg. & Tech.
Hyderabad-500091. T.S.

LORDS INSTITUTE OF ENGINEERING & TECHNOLOGY

(UGC Autonomous)

Approved by AICTE | Affiliated to Osmania University | Estd.2003.

Department of Computer Science and Engineering (Data Science)

B.E. Computer Science and Engineering (Data Science) Program Outcomes (POs):

Engineering Graduates will be able to:

S. No.	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 modelling 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.

Head of the Department

Head of the Department
CSE (Data Science)
Lords Institute of Engg. & Tech.
Hyderabad-500091. T.S.

B.E. Computer Science and Engineering (Data Science) Program Specific Outcomes (PSO's):

PSO1	**Professional Skills:** The ability to apply knowledge of Data Science in real-time software project development using open-source and commercial programming environment to deliver quality software product for the organization's success.
PSO2	**Problem-Solving Skills:** The ability to acquaint with the current trends in industrial/research areas and there by innovate through novel solutions to the existing problems.

Head of the Department

Head of the Department
CSE (Data Science)
Lords Institute of Engg. & Tech.
Hyderabad-500091. T.S.

Course Outcomes: U21CD5L1 Artificial Intelligence Lab

Student will able to

CO. No.	Description	Bloom's Taxonomy Level
C52.1	Use AI for problem-solving: Apply AI principles to solve problems creatively.	**BTL2**
C52.2	Build intelligent systems: Create systems that can learn and adapt.	**BTL3**
C52.3	Choose the right learning method: Select the best approach for different tasks.	**BTL2**
C52.4	Use expert systems: Apply AI to make informed decisions.	**BTL4**
C52.5	Apply machine learning: Use AI to analyze data and make predictions.	**BTL5**

Note: Bloom's Taxonomy Levels

BTL1-Remember	**BTL2-Understand**	**BTL3 - Apply**
BTL4-Analyze	**BT5 - Evaluate**	**BTL6 - Create**

Course Articulation Matrix:

Mapping of Course Outcomes (CO) with Program Outcomes (PO) and Program Specific Outcomes (PSO's):

Course Title: Artificial Intelligence Lab	**Course code:** U23CD5L1
Sem: V	**Academic Year:** 2025-26
Name of the Faculty: Mr. Md Anjar Ahsan	

Course Outcomes (CO)	Program Outcomes (PO)												Program Specific Outcomes (PSO's)	
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2
C56.1	3	3	3	3	-	-	1	-	-	-	3	-	3	-
C56.2	3	3	-	-	3	-	-	-	3	-	3	3	3	3
C56.3	-	3	-	3	3	-	-	3	-	-	-	3	-	3
C56.4	-	-	3	3	-	3	2	-	-	-	-	2	3	-
C56.5	-	3	-	-	2	-	-	3	3	-	2	-	-	3
C52	2.71	3	2	2.25	2.67	3	1	2	2	3	2.5	2.8	2	3

- **Level 1**: Low correlation
- **Level 2**: Medium correlation
- **Level 3**: High correlation

POs Defined:

- **PO1**: Engineering knowledge
- **PO2**: Problem analysis
- **PO3**: Design/Development of solutions
- **PO4**: Investigation of complex problems
- **PO5**: Modern tool usage
- **PO6**: The engineer and society
- **PO7**: Environment and sustainability
- **PO8**: Ethics
- **PO9**: Individual and teamwork
- **PO10**: Communication
- **PO11**: Project management and finance
- **PO12**: Life-long learning

PSOs Defined:

- **PSO1**: Professional Skills
- **PSO2**: Problem-Solving Skills

Signature of the Faculty

Course Code	Course Title						Core/Elective
U23CD5L1	ARTIFICIAL INTELLIGENCE LAB						CORE
Prerequisite	Contact Hours Per Week				CIE	SEE	Credits
Python Programming	L	T	D	P			
	-	-	-	3	25	50	1.5

Course Objectives

Develop ability to

1. Understand the importance of the field of AI by discussing its history and various applications.
2. Learn about one of the basic applications of A.I, search state formulations
3. Learn knowledge representation implementation.
4. Learn how to reason when an agent has only uncertain information about its task.
5. Know various supervised and unsupervised learning algorithms

Course Outcomes

At the end of the course, student would be able to

1. Illustrate basic principles of AI in solutions that require problem solving, search, inference
2. Demonstrate understanding of steps involved in building of intelligent agents, expert systems, Bayesian networks
3. Differentiate between learning paradigms to be applied for an application
4. Demonstrate Expert system its utilization
5. Illustrate AI application machine learning & its types.

1. Write a Program to Implement Breadth First Search
2. Write a Program to Implement Depth First Search
3. Write a Program to Implement Tic-Tac-Toe game
4. Write a Program to Implement 8-Puzzle problem
5. Write a Program to Implement Water-Jug problem
6. Write a Program to Implement Travelling Salesman Problem
7. Write a Program to Implement Tower of Hanoi
8. Write a Program to Implement Monkey Banana Problem
9. Write a Program to Implement Missionaries-Cannibals Problems
10. Write a Program to Implement N-Queens Problem using Python.
11. Write a program to train and validate the following classifiers for given data (scikit-learn):

a) Decision Tree

b) Multi-layer Feed Forward neural network

c) Implementation of Gaussian Naive Bayes classifier using scikit-learn (Any two classifiers).

12. Write a program to Implementation of Linear Regression using python(Any Two Algorithm).

Text Books:

1. Rajjan Shinghal, Pattern Recognition, Oxford University Press, 2006.
2. Tom M. Mitchell, Machine Learning, Mc Graw Hill, 1997
3. Stephen Marsland, Machine Learning - An Algorithmic Perspective, CRC Press, 2009
4. Margaret H Dunham, Data Mining, Pearson Edition., 2003.
5. Galit Shmueli, Nitin R Patel, Peter C Bruce, Data Mining for Business Intelligence, Wiley India Edition, 2007.

Suggested Readings:

1. Stuart Russell and Peter Norvig-Artificial Intelligence – A Modern Approach, Third edition, Pearson Education Press,.
2. Saroj Kaushik, Artificial Intelligence, Cengage Learning, 2011

Experiment No. 01

Q. Write a python program to implement Depth First Search Traversal.

Algorithms Steps:

1. **Start** at the root node (or an arbitrary node).
2. Mark the node as visited.
3. For each adjacent unvisited node, perform the following:
   - Recursively apply DFS.
4. Backtrack when no unvisited adjacent nodes are left.

Code:

```
# Step 1: Define the graph as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

# Step 2: Implement the DFS function
def dfs(graph, node, visited):
    if node not in visited:
        print(node, end=' ')
        visited.add(node)
        for neighbor in graph[node]:
            dfs(graph, neighbor, visited)

# Step 3: Main program
if __name__ == "__main__":
    visited = set()  # Set to keep track of visited nodes
    print("Depth First Search Traversal starting from vertex 'A':")
    dfs(graph, 'A', visited)
```

Output:

```
Depth First Search Traversal starting from vertex 'A':
A B D E F C
```

Viva Questions:

1. How can we modify DFS to detect cycles in a graph?
2. Where can DFS be applied in real life? Give examples.
3. What are the key differences between DFS and Breadth First Search (BFS)?
4. What are the time and space complexities of DFS?
5. What are the pros and cons of using a stack for DFS instead of recursion?

Experiment No. 02

Q. Write a python program to implement Breadth First Search Traversal.

Algorithms Steps:

1. Start at the root node (or any arbitrary node).
2. Mark the node as visited and enqueue it.
3. While the queue is not empty:
    - Dequeue a node from the front of the queue.
    - Visit all its unvisited neighbors, mark them as visited, and enqueue them.
4. Repeat until all reachable nodes have been visited.

Code:

```
# Step 1: Define the graph as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

# Step 2: Implement the BFS function
from collections import deque

def bfs(graph, start):
    visited = set()  # Set to keep track of visited nodes
    queue = deque([start])  # Initialize the queue with the start node

    while queue:
        node = queue.popleft()  # Dequeue a node
        if node not in visited:
            print(node, end=' ')
            visited.add(node)  # Mark it as visited
            # Enqueue all unvisited neighbors
```

```
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)

# Step 3: Main program
if __name__ == "__main__":
    print("Breadth First Search Traversal starting from vertex 'A':")
    bfs(graph, 'A')
```

Output:

```
Breadth First Search Traversal starting from vertex 'A':
A B C D E F
```

Viva Questions:

1. How can we use BFS to find cycles in a graph?
2. Where is BFS used in real life? Can you give an example?
3. What are the main differences between BFS and Depth First Search (DFS)?
4. What are the time and space complexities of BFS?
5. Why do we use a queue in BFS, and how does it help with traversal?

Experiment No. 03:

Q. Write a Program to Implement Tic-Tac-Toe game

Algorithms Steps:

1. **Initialize the board** as a 3x3 grid, empty at the start.
2. **Display the board** after each move so players can see the current state.
3. **Players take turns** placing their marks ('X' and 'O') on the board.
4. After each move, **check if a player has won** by:
5. Checking rows, columns, and diagonals for three identical marks.
6. Also, check if the board is full — if yes and no winner, it's a **draw**.
7. Continue alternating moves until there's a winner or a draw.
8. Announce the result.

Code:

```
import random
import time

# Constants for the game
COMPUTER = 1
HUMAN = 2
SIDE = 3
COMPUTERMOVE = 'O'
HUMANMOVE = 'X'

# Function to initialise the game / Tic-Tac-Toe board
def initialise():
    board = [[' ' for _ in range(SIDE)] for _ in range(SIDE)]
    moves = [i for i in range(SIDE*SIDE)]
    random.shuffle(moves)
    return board, moves

# Function to print the Tic-Tac-Toe board
def showBoard(board):
    print("\n\n")
    print("\t\t\t  {} | {}  | {}  ".format(board[0][0], board[0][1],     board[0][2]))
    print("\t\t\t--------------")
    print("\t\t\t  {} | {}  | {}  ".format(board[1][0], board[1][1], board[1][2]))
```

```
    print("\t\t\t--------------")
    print("\t\t\t  {} | {}  | {}  \n\n".format(board[2][0], board[2][1], board[2][2]))

# Function to show the instructions
def showInstructions():
    print("\t\t\t  Tic-Tac-Toe\n\n")
    print("Choose a cell numbered from 1 to 9 as below and play\n\n")
    print("\t\t\t  1 | 2  | 3  ")
    print("\t\t\t--------------")
    print("\t\t\t  4 | 5  | 6  ")
    print("\t\t\t--------------")
    print("\t\t\t  7 | 8  | 9  \n\n")
    print("-\t-\t-\t-\t-\t-\t-\t-\t-\n\n")

# Function to declare the winner of the game
def declareWinner(whoseTurn):
    if whoseTurn == COMPUTER:
        print("COMPUTER has won")
    else:
        print("HUMAN has won")
# Functions to check if any of the rows, columns, or diagonals have been crossed
def rowCrossed(board):
    for i in range(SIDE):
        if board[i][0] == board[i][1] and board[i][1] == board[i][2] and board[i][0] != ' ':
            return True
    return False
def columnCrossed(board):
    for i in range(SIDE):
        if board[0][i] == board[1][i] and board[1][i] == board[2][i] and board[0][i] != ' ':
            return True
    return False
def diagonalCrossed(board):
    if board[0][0] == board[1][1] and board[1][1] == board[2][2] and board[0][0] != ' ':
        return True
    if board[0][2] == board[1][1] and board[1][1] == board[2][0] and board[0][2] != ' ':
        return True
    return False
# Function to check if the game is over
def gameOver(board):
    return rowCrossed(board) or columnCrossed(board) or diagonalCrossed(board)
# Function to play Tic-Tac-Toe
def playTicTacToe(whoseTurn):
```

```
    board, moves = initialise()
    showInstructions()
    moveIndex = 0
    while not gameOver(board) and moveIndex != SIDE*SIDE:
        if whoseTurn == COMPUTER:
            x = moves[moveIndex] // SIDE
            y = moves[moveIndex] % SIDE
            board[x][y] = COMPUTERMOVE
            print("COMPUTER has put a {} in cell {}".format(COMPUTERMOVE,    moves[moveIndex]+1))
            showBoard(board)
            moveIndex += 1
            whoseTurn = HUMAN
        elif whoseTurn == HUMAN:
            x = moves[moveIndex] // SIDE
            y = moves[moveIndex] % SIDE
            board[x][y] = HUMANMOVE
            print("HUMAN has put a {} in cell {}".format(HUMANMOVE, moves[moveIndex]+1))
            showBoard(board)
            moveIndex += 1
            whoseTurn = COMPUTER
    if not gameOver(board) and moveIndex == SIDE*SIDE:
        print("It's a draw")
    else:
        if whoseTurn == COMPUTER:
            whoseTurn = HUMAN
        elif whoseTurn == HUMAN:
            whoseTurn = COMPUTER
        declareWinner(whoseTurn)
# Driver function
if __name__ == "__main__":
    # Let us play the game with COMPUTER starting first
    playTicTacToe(COMPUTER)
```

Output:

```
Tic-Tac-Toe
Choose a cell numbered from 1 to 9 as below and play

                    1 | 2  | 3
                   --------------
                    4 | 5  | 6
                   --------------
                    7 | 8  | 9

COMPUTER has put a O in cell 5
                      |    |
                   --------------
                      | O  |
                   --------------
                      |    |
HUMAN has put a X in cell 9
                      |    |
                   --------------
                      | O  |
                   --------------
                      |    | X

COMPUTER has put a O in cell 6
                      |    |
                   --------------
                      | O  | O
                   --------------
                      |    | X

HUMAN has put a X in cell 4
                      |    |
                   --------------
                    X | O  | O
                   --------------
                      |    | X

COMPUTER has put a O in cell 8
                      |    |
                   --------------
                    X | O  | O
                   --------------
                      | O  | X

HUMAN has put a X in cell 2
                      | X  |
                   --------------
                    X | O  | O
                   --------------
                      | O  | X
```

```
COMPUTER has put a O in cell 7
     |  X  |
  -------------
    X | O  | O
  -------------
    O | O  | X


HUMAN has put a X in cell 3
     |  X  | X
  -------------
    X | O  | O
  -------------
    O | O  | X

COMPUTER has put a O in cell 1
    O | X  | X
  -------------
    X | O  | O
  -------------
    O | O  | X

It's a draw
```

Viva Questions:

1. What data structure did you use to represent the board?
2. How did you check for a win or draw condition?
3. Did you implement a single-player or two-player version?
4. How does the computer choose its move (if applicable)? Did you use the Minimax algorithm?
5. What is the time complexity of the Minimax algorithm?

Experiment No. 04

Q. Write a Program to Implement 8-Puzzle problem

Algorithms Steps:

1. You have a 3x3 grid with numbers 1-8 and one empty spot (0).
2. The goal is to move tiles around until you get the target arrangement.
3. We use a method called **A* search** which tries to find the shortest way to solve the puzzle.
4. A* uses:

   - **g(n):** number of moves taken so far.
   - **h(n):** how far the puzzle is from the goal (we use Manhattan distance).
   - **f(n) = g(n) + h(n):** total cost score.

5. Start with the initial state, then explore neighbors (states you get by sliding tiles).
6. Always pick the state with the lowest **f(n)** to explore next.
7. Repeat until you find the goal state or no states left.

Code:

```
# Import necessary libraries
from collections import deque

# Define the dimensions of the puzzle
N = 3

# Structure to store a state of the puzzle
class PuzzleState:
    def __init__(self, board, x, y, depth):
        self.board = board
        self.x = x
        self.y = y
        self.depth = depth

# Possible moves: Left, Right, Up, Down
row = [0, 0, -1, 1]
col = [-1, 1, 0, 0]

# Function to check if a given state is the goal state
def is_goal_state(board):
    goal = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    return board == goal

# Function to check if a move is valid
```

```
def is_valid(x, y):
    return 0 <= x < N and 0 <= y < N

# Function to print the board
def print_board(board):
    for row in board:
        print(' '.join(map(str, row)))
    print("--------")

# Depth-First Search to solve the 8-puzzle problem
def solve_puzzle_dfs(start, x, y):
    stack = []
    visited = set()

    stack.append(PuzzleState(start, x, y, 0))
    visited.add(tuple(map(tuple, start)))

    while stack:
        curr = stack.pop()

        # Print the current board
        print(f'Depth: {curr.depth}')
        print_board(curr.board)

        # Check if goal state is reached
        if is_goal_state(curr.board):
            print(f'Goal state reached at depth {curr.depth}')
            return

        # Explore possible moves
        for i in range(4):
            new_x = curr.x + row[i]
            new_y = curr.y + col[i]

            if is_valid(new_x, new_y):
                new_board = [row[:] for row in curr.board]
                # Swap the tiles
                new_board[curr.x][curr.y], new_board[new_x][new_y] =
new_board[new_x][new_y], new_board[curr.x][curr.y]

                # If this state has not been visited before, push to stack
                board_tuple = tuple(map(tuple, new_board))
                if board_tuple not in visited:
                    visited.add(board_tuple)
                    stack.append(PuzzleState(new_board, new_x, new_y,
curr.depth + 1))
```

```
    print('No solution found (DFS Brute Force reached depth limit)')

# Driver Code
if __name__ == '__main__':
    start = [[1, 2, 3], [4, 0, 5], [6, 7, 8]]
    x, y = 1, 1

    print('Initial State:')
    print_board(start)

    solve_puzzle_dfs(start, x, y)
# Example usage:
start = (1, 2, 3,
         4, 0, 6,
         7, 5, 8)

goal = (1, 2, 3,
        4, 5, 6,
        7, 8, 0)

solution = a_star(start, goal)

if solution:
    print(f"Solution found in {len(solution)-1} moves:")
    for step in solution:
        print(step[0:3])
        print(step[3:6])
        print(step[6:9])
        print()
else:
    print("No solution found.")
```

Output:

Initial State:

1 2 3

4 0 5

6 7 8

Depth: 0

1 2 3

4 0 5

6 7 8

Depth: 1

1 2 3

4 7 5

6 0 8

Depth: 2

1 2 3

4 7 5

6 8 0

Depth: 3

1 2 3

4 7 0

6 8 5

-------...

Viva Questions:

1. What is the goal of the 8-puzzle problem?
2. Which algorithm did you use to solve it (e.g., BFS, DFS, A*)?
3. What is the heuristic used in A* (e.g., Manhattan distance)?
4. Why can't all initial states be solved?
5. What is the time and space complexity of A* in this case?

Experiment No. 05

Q. Write a python program to implement Water Jug Problem.

Algorithms Steps:

1. Initialize: Create a queue for states and a set for visited states, starting with both jugs empty.
2. Check Goal: If ZZZ is greater than the maximum jug capacity, print "Not possible" and exit.
3. Loop:
4. Dequeue the current state (amounts in both jugs).
5. Check if either jug has ZZZ liters or if their total equals ZZZ. If so, print the result and exit.
6. Mark as Visited: Add the current state to the visited set.
7. Generate States: Create new states based on allowed operations (fill, empty, pour).
8. Enqueue Valid States: Add unvisited states to the queue for exploration.
9. Return: If the queue is exhausted with no solution, print *"Not possible."*

Code:

```
from collections import deque

def water_jug_problem(x, y, z):
    if z > max(x, y):
        return "Not possible"

    queue = deque([(0, 0)])  # (Jug X, Jug Y)
    visited = set()

    while queue:
        a, b = queue.popleft()

        # Check if we've reached the target
        if a == z or b == z or a + b == z:
            return (a, b)

        # Mark the state as visited
        visited.add((a, b))

        # Possible operations
        states = [
            (x, b),  # Fill Jug X
```

```
            (a, y),  # Fill Jug Y
            (0, b),  # Empty Jug X
            (a, 0),  # Empty Jug Y
            (a - min(a, y - b), b + min(a, y - b)),  # Pour from X to Y
            (a + min(b, x - a), b - min(b, x - a))   # Pour from Y to X
        ]

        for state in states:
            if state not in visited:
                queue.append(state)

    return "Not possible"

# Example usage
x, y, z = 4, 3, 2  # Jug capacities and target
result = water_jug_problem(x, y, z)
print(f"Result: {result}")
```

Output:

```
Result: (4, 2)
```

Viva Questions:

1. What is the Water Jug Problem?
2. What approach is used to solve it in the program?
3. How does the program check if ZZZ can be measured?
4. What output is produced if ZZZ exceeds both jug capacities?
5. Can the problem be solved if the GCD of the jug capacities does not divide ZZZ?

Experiment No. 06

Q. Write a python program to Implement Travelling Salesman Problem using Python.

Algorithms Steps:

1. **Initialize**: List cities and set minimum cost to infinity.
2. **Generate Routes**: Use permutations to create all possible routes.
3. **Calculate Cost**: For each route, sum the distances between consecutive cities.
4. **Update Minimum**: If the current route cost is lower than the recorded minimum, update the minimum and best route.
5. **Return Result**: Output the optimal route and its total cost.

Code:

```
from itertools import permutations

def tsp_bruteforce(graph):
    # Generate all possible routes
    cities = list(graph.keys())
    min_path = float('inf')
    best_route = None

    for perm in permutations(cities[1:]):
        current_path = [cities[0]] + list(perm) + [cities[0]]
        current_cost = sum(graph[current_path[i]][current_path[i + 1]]
for i in range(len(current_path) - 1))

        if current_cost < min_path:
            min_path = current_cost
            best_route = current_path

    return best_route, min_path

# Example usage
graph = {
    'A': {'B': 10, 'C': 15, 'D': 20},
    'B': {'A': 10, 'C': 35, 'D': 25},
    'C': {'A': 15, 'B': 35, 'D': 30},
    'D': {'A': 20, 'B': 25, 'C': 30}
}
```

```
route, cost = tsp_bruteforce(graph)
print(f"Optimal route: {route}, Cost: {cost}")
```

Output:

```
Optimal route: ['A', 'B', 'D', 'C', 'A'], Cost: 80
```

Viva Questions:

1. What is the Travelling Salesman Problem (TSP)?
2. What method is used in the program to solve TSP?
3. How does the program generate possible routes for TSP?
4. What is the time complexity of the brute-force approach for TSP?
5. How can the TSP be solved more efficiently than the brute-force method?

Experiment No. 07

Q. Write a Program to Implement Tower of Hanoi

Algorithms Steps:

1. **Base Case**: If there is only 1 disk:
   - Move it directly from the source rod to the destination rod.
2. **Recursive Case**:
   - Move the top n-1 disks from **source** to **auxiliary** rod.
   - Move the nth (largest) disk from **source** to **destination** rod.
   - Move the n-1 disks from **auxiliary** to **destination** rod.

Code:

```
# Recursive Python function to solve tower of hanoi
def TowerOfHanoi(n, from_rod, to_rod, aux_rod):
    if n == 0:
        return
    TowerOfHanoi(n-1, from_rod, aux_rod, to_rod)
    print("Move disk", n, "from rod", from_rod, "to rod", to_rod)
    TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)
# Driver code
N = 3
# A, C, B are the name of rods
TowerOfHanoi(N, 'A', 'C', 'B')
# Contributed By Harshit Agrawal
```

Output:

Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 3 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C

Viva Questions:

1. What is the base condition for the Tower of Hanoi recursive function?
2. How many moves are required to solve Tower of Hanoi with n disks?
3. Explain the recursive approach.
4. Can this be solved iteratively?
5. What is the time complexity of this solution?

Experiment No. 08

Q. Write a Program to Implement Monkey Banana Problem.

Algorithms Steps:

1. **Define the initial state**:
   - Monkey is at some location.
   - Box is at some location.
   - Monkey is on the floor.
   - Banana is at a fixed position (on ceiling).
2. **Define goal state**:
   - Monkey has the banana.
3. **Define actions**:
   - Go(x): move to location x.
   - Push(box, from, to): push the box from one location to another.
   - Climb(box): climb on the box.
   - Grasp: grasp the banana.
4. Use **simple rule-based logic** to find a path from the initial state to the goal.

Code:

```
# Possible locations
locations = ['door', 'window', 'middle']

# Initial state
state = {
    'monkey_at': 'door',
    'box_at': 'window',
    'monkey_on_box': False,
    'has_banana': False
}

def print_state(state):
    print(f"Monkey at: {state['monkey_at']}")
    print(f"Box at: {state['box_at']}")
    print(f"Monkey on box: {state['monkey_on_box']}")
    print(f"Has banana: {state['has_banana']}")
    print("-" * 30)

def go_to(state, location):
    print(f"Monkey walks to {location}.")
    state['monkey_at'] = location

def push_box(state, to_location):
    if state['monkey_at'] == state['box_at']:
        print(f"Monkey pushes the box to {to_location}.")
        state['box_at'] = to_location
```

```
        state['monkey_at'] = to_location
    else:
        print("Monkey is not at the box location to push it.")

def climb_box(state):
    if state['monkey_at'] == state['box_at']:
        print("Monkey climbs onto the box.")
        state['monkey_on_box'] = True
    else:
        print("Monkey must be at the box location to climb.")

def grasp_banana(state):
    if state['monkey_on_box'] and state['monkey_at'] == 'middle':
        print("Monkey grasps the banana!")
        state['has_banana'] = True
    else:
        print("Monkey can't grasp the banana yet.")

def monkey_banana_problem():
    print("Initial State:")
    print_state(state)

    # Step 1: Go to box
    go_to(state, 'window')

    # Step 2: Push box to middle (under banana)
    push_box(state, 'middle')

    # Step 3: Climb box
    climb_box(state)

    # Step 4: Grasp banana
    grasp_banana(state)

    # Final state
    print("\nFinal State:")
    print_state(state)

    if state['has_banana']:
        print("Monkey successfully got the banana! 🐒🍌")
    else:
        print("Monkey failed to get the banana.")

# Run the problem
if __name__ == "__main__":
    monkey_banana_problem()
```

Output:

Initial State:

Monkey at: door

Box at: window

Monkey on box: False

Has banana: False

Monkey walks to window.

Monkey pushes the box to middle.

Monkey climbs onto the box.

Monkey grasps the banana!

Final State:

Monkey at: middle

Box at: middle

Monkey on box: True

Has banana: True

Monkey successfully got the banana!

Viva Questions:

1. What kind of AI problem is this search, planning, or constraint satisfaction?
2. How did you represent the monkey's actions and states?
3. What are the preconditions and effects for each action?
4. Did you use STRIPS or any planning algorithm?
5. What is the goal state?

Experiment No. 09

Q. Write a Program to Implement Missionaries-Cannibals Problems.

Algorithms Steps:

1. **Represent state** as `(M_left, C_left, boat_position)`, where:
   - `M_left` = number of missionaries on the left bank.
   - `C_left` = number of cannibals on the left bank.
   - `boat_position` = `'left'` or `'right'` indicating boat location.
2. **Initial state**: `(3, 3, 'left')` — everyone on the left bank.
3. **Goal state**: `(0, 0, 'right')` — everyone safely moved to the right bank.
4. **Allowed moves**:
   - The boat can carry 1 or 2 people (missionaries and/or cannibals).
   - Possible passenger combinations:
     - (1 missionary), (2 missionaries), (1 cannibal), (2 cannibals), (1 missionary + 1 cannibal).
5. **Move the boat** across the river with chosen passengers, updating the state.
6. **Check if the state is valid**:
   - Missionaries are never outnumbered by cannibals on either side (unless missionaries are zero on that side).
   - Number of missionaries and cannibals on each side is between 0 and 3.
7. Use **Breadth First Search (BFS)** to explore states:
   - Start from initial state.
   - Generate all valid next states.
   - Keep track of visited states to avoid loops.
   - Stop when goal state is reached.
8. **Return the path** (sequence of states) from start to goal.

Code:

```
from collections import deque

def is_valid_state(m_left, c_left):
    m_right = 3 - m_left
    c_right = 3 - c_left

    # Check if numbers are within bounds
    if m_left < 0 or c_left < 0 or m_right < 0 or c_right < 0:
        return False

    # Check missionaries are safe on left side
    if m_left > 0 and m_left < c_left:
        return False

    # Check missionaries are safe on right side
    if m_right > 0 and m_right < c_right:
        return False

    return True

def get_possible_moves():
    # Possible passengers in the boat (missionaries, cannibals)
    return [(1,0), (2,0), (0,1), (0,2), (1,1)]

def get_next_states(state):
    m_left, c_left, boat = state
    next_states = []

    for m_move, c_move in get_possible_moves():
        if boat == 'left':
            new_m_left = m_left - m_move
            new_c_left = c_left - c_move
            new_boat = 'right'
        else:
            new_m_left = m_left + m_move
            new_c_left = c_left + c_move
            new_boat = 'left'

        if is_valid_state(new_m_left, new_c_left):
            next_states.append((new_m_left, new_c_left, new_boat))
    return next_states

def bfs():
```

```
    initial_state = (3, 3, 'left')
    goal_state = (0, 0, 'right')

    queue = deque()
    queue.append((initial_state, [initial_state]))  # state and path to reach
it
    visited = set()
    visited.add(initial_state)

    while queue:
        current_state, path = queue.popleft()

        if current_state == goal_state:
            return path

        for next_state in get_next_states(current_state):
            if next_state not in visited:
                visited.add(next_state)
                queue.append((next_state, path + [next_state]))
    return None

def print_solution(solution):
    if not solution:
        print("No solution found.")
        return

    print("Solution steps:")
    for i, state in enumerate(solution):
        m_left, c_left, boat = state
        m_right = 3 - m_left
        c_right = 3 - c_left
        print(f"Step {i}: Left bank: {m_left}M, {c_left}C | Right bank:
{m_right}M, {c_right}C | Boat: {boat}")

if __name__ == "__main__":
    solution = bfs()
    print_solution(solution)
```

Output:

```
Solution steps:

Step 0: Left bank: 3M, 3C |

Right bank: 0M, OC

,

Boat: left

Step 1: Left bank: 3M, 1C

Right bank: 0M, 2C |

Boat: right

Step 2: Left bank: 3M, 2C

,

Right bank: 0M, 1C

Boat: left

Step 3: Left bank: 3M, OC

,

Right bank: 0M, 3C

Boat: right

Step 4: Left bank: 3M, 1C

,

Right bank: 0M, 2C |

Boat: left

Step 5: Left bank: 1M, 1C |

Right bank: 2M, 2C |

Boat: right

Step 6: Left bank: 2M, 2C
```

```
,
Right bank: 1M, 1C
Boat: left
Step 7: Left bank: 0M, 2C |
Right bank: 3M, 1C
Boat: right
Step 8: Left bank: 0M, 3C |
Right bank: 3M, OC |
Boat: left
Step 9: Left bank: 0M, 1C |
Right bank: 3M, 2C |
Boat: right
Step 10: Left bank: 1M, 1C
Right bank: 2M, 2C
,
Boat: left
Step 11: Left bank: 0M, OC |
Right bank: 3M, 3C |
Boat: right
```

Viva Questions:

1. What are the constraints of the problem?
2. How did you represent each state?
3. What algorithm did you use to search for a solution?
4. How did you avoid invalid or repeated states?
5. Can you explain why some states are unsafe?

Experiment No. 10

Q. Write a Program to Implement N-Queens Problem using Python.

Algorithms Steps:

Algorithm Steps (Backtracking):

1. Start from the first row.
2. Try placing a queen in each column of the current row.
3. For each position, check if it's safe:
   - No other queen in the same column.
   - No other queen on the upper-left diagonal.
   - No other queen on the upper-right diagonal.
4. If safe, place the queen and move to the next row.
5. If placing queen in all columns fails for a row, backtrack to the previous row and move the queen to next possible column.
6. Repeat until all queens are placed successfully.
7. Print or store the solution.

Code:

```
def is_safe(board, row, col, n):
    # Check column for another queen
    for i in range(row):
        if board[i][col] == 1:
            return False

    # Check upper-left diagonal
    i, j = row - 1, col - 1
    while i >= 0 and j >= 0:
        if board[i][j] == 1:
            return False
        i -= 1
        j -= 1

    # Check upper-right diagonal
    i, j = row - 1, col + 1
```

```
    while i >= 0 and j < n:
        if board[i][j] == 1:
            return False
        i -= 1
        j += 1

    return True

def solve_n_queens(board, row, n):
    # If all queens are placed
    if row == n:
        print_board(board, n)
        print()
        return True

    res = False
    for col in range(n):
        if is_safe(board, row, col, n):
            board[row][col] = 1  # Place queen

            # Recursively place rest of queens
            res = solve_n_queens(board, row + 1, n) or res

            # Backtrack
            board[row][col] = 0
    return res

def print_board(board, n):
    for row in range(n):
        for col in range(n):
            print("Q" if board[row][col] == 1 else ".", end=" ")
        print()

if __name__ == "__main__":
    n = int(input("Enter number of queens (N): "))
    board = [[0]*n for _ in range(n)]
    if not solve_n_queens(board, 0, n):
        print("No solution exists.")
```

Output:

Enter number of queens (N): 4

. Q . .
. . . Q
Q . . .
. . Q .

. . Q .
Q . . .
. . . Q
. Q . .

Viva Questions:

1. What is the objective of the N-Queens problem?
2. What approach did you use — Backtracking or any other?
3. How do you check if placing a queen is valid?
4. What is the time complexity of your solution?
5. Can you extend your program to count all possible solutions?

Experiment No. 11

Q. Write a program to train and validate the following classifiers for given data (scikit-learn):

a) Decision Tree
b) Multi-layer Feed Forward neural network
c) Implementation of Gaussian Naive Bayes classifier using scikit-learn (Any two classifiers).

Experiment No. 11 (a)

a) Decision Tree

Algorithms Steps:

Algorithm Steps for Decision Tree Classifier:

1. Load dataset (we'll use the Iris dataset as an example).
2. Split the dataset into training and testing sets.
3. Create a Decision Tree classifier object.
4. Train (fit) the classifier using the training data.
5. Make predictions on the testing data.
6. Evaluate the model's performance using accuracy.
7. (Optional) Print detailed classification report for better understanding.

Code:

```
# Step 1: Import libraries
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report

# Step 2: Load dataset
data = load_iris()
X = data.data      # Features (input)
y = data.target    # Labels (output)

# Step 3: Split dataset into training and testing sets
# 70% training, 30% testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

```
# Step 4: Create Decision Tree classifier object
dt_classifier = DecisionTreeClassifier()

# Step 5: Train the classifier on training data
dt_classifier.fit(X_train, y_train)

# Step 6: Predict the labels for test data
y_pred = dt_classifier.predict(X_test)

# Step 7: Evaluate the classifier
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Decision Tree classifier: {accuracy:.2f}")

# Optional: Detailed classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

Output:

```
Accuracy of Decision Tree classifier: 1.00

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

Viva Questions:

1. What is a decision tree?
2. What are entropy and information gain?
3. What criteria did you use to split the nodes? (gini or entropy)?
4. What is overfitting in decision trees, and how can you prevent it?
5. How do you control the depth of a decision tree in scikit-learn?
6. How do you validate the accuracy of your model?

Experiment No. 11 (b)

(b) Multi-layer Feed Forward neural network

Algorithms Steps:

1. **Load dataset** (e.g., Iris dataset).
2. **Split the dataset** into training and testing parts.
3. **Create an MLPClassifier object** specifying hidden layers.
4. **Train the neural network** on training data.
5. **Predict the labels** on test data.
6. **Evaluate the model** using accuracy score.
7. (Optional) Print detailed classification report.

Code:

```
# Step 1: Import necessary libraries
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, classification_report

# Step 2: Load dataset
data = load_iris()
X = data.data
y = data.target

# Step 3: Split dataset into train and test (70% train, 30% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Step 4: Create MLP classifier (neural network)
# hidden_layer_sizes=(100,) means 1 hidden layer with 100 neurons
mlp = MLPClassifier(hidden_layer_sizes=(100,), max_iter=500, random_state=42)

# Step 5: Train the model
mlp.fit(X_train, y_train)

# Step 6: Predict test labels
y_pred = mlp.predict(X_test)

# Step 7: Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of MLP Neural Network: {accuracy:.2f}")
```

```
# Optional: Detailed classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

Output:

```
Accuracy of MLP Neural Network: 1.00

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

Viva Questions:

1. What is a multi-layer perceptron?
2. Which activation function did you use? (e.g., ReLU, tanh, logistic)
3. What solver did you use? (adam, sgd, or lbfgs)
4. What are epochs and batch size?
5. What is the role of backpropagation?
6. What are the advantages of using neural networks over decision trees?

Experiment No. 11 (c)

c) Implementation of Gaussian Naive Bayes classifier using scikit-learn (Any two classifiers).

Algorithms Steps:

Algorithm Steps (General):

1. *Load the dataset (e.g., Iris dataset).*
2. *Split the dataset into training and testing sets.*
3. *Create classifier objects for Gaussian Naive Bayes and Decision Tree.*
4. *Train both classifiers using training data.*
5. *Predict the labels on the test set.*
6. *Evaluate the models using accuracy score.*
7. *(Optional) Print detailed classification report*

Code:

```
# Step 1: Import libraries
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report

# Step 2: Load dataset
data = load_iris()
X = data.data
y = data.target

# Step 3: Split dataset into training and testing sets (70% train, 30% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Step 4a: Create Gaussian Naive Bayes classifier and train it
gnb = GaussianNB()
gnb.fit(X_train, y_train)

# Step 5a: Predict with Gaussian Naive Bayes
y_pred_gnb = gnb.predict(X_test)

# Step 6a: Evaluate Gaussian Naive Bayes
accuracy_gnb = accuracy_score(y_test, y_pred_gnb)
print(f"Gaussian Naive Bayes Accuracy: {accuracy_gnb:.2f}")
```

```
print("Gaussian Naive Bayes Classification Report:")
print(classification_report(y_test, y_pred_gnb))

# Step 4b: Create Decision Tree classifier and train it
dt = DecisionTreeClassifier()
dt.fit(X_train, y_train)

# Step 5b: Predict with Decision Tree
y_pred_dt = dt.predict(X_test)

# Step 6b: Evaluate Decision Tree
accuracy_dt = accuracy_score(y_test, y_pred_dt)
print(f"\nDecision Tree Accuracy: {accuracy_dt:.2f}")
print("Decision Tree Classification Report:")
print(classification_report(y_test, y_pred_dt))
```

Output:

```
Gaussian Naive Bayes Accuracy: 0.98
Gaussian Naive Bayes Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      0.92      0.96        13
           2       0.93      1.00      0.96        13

    accuracy                           0.98        45
   macro avg       0.98      0.97      0.97        45
weighted avg       0.98      0.98      0.98        45


Decision Tree Accuracy: 1.00
Decision Tree Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

Viva Questions:

1. What assumptions does Gaussian Naive Bayes make?
2. Why is it called "Naive"?
3. What is the formula for Bayes' theorem?
4. How does Gaussian Naive Bayes handle continuous features?
5. How does it perform compared to Decision Trees or SVM?

Experiment No. 12

Q. Write a program to Implementation of Linear Regression using python(Any Two Algorithm).

Algorithms Steps:

1) Ordinary Least Squares (OLS) - scikit-learn

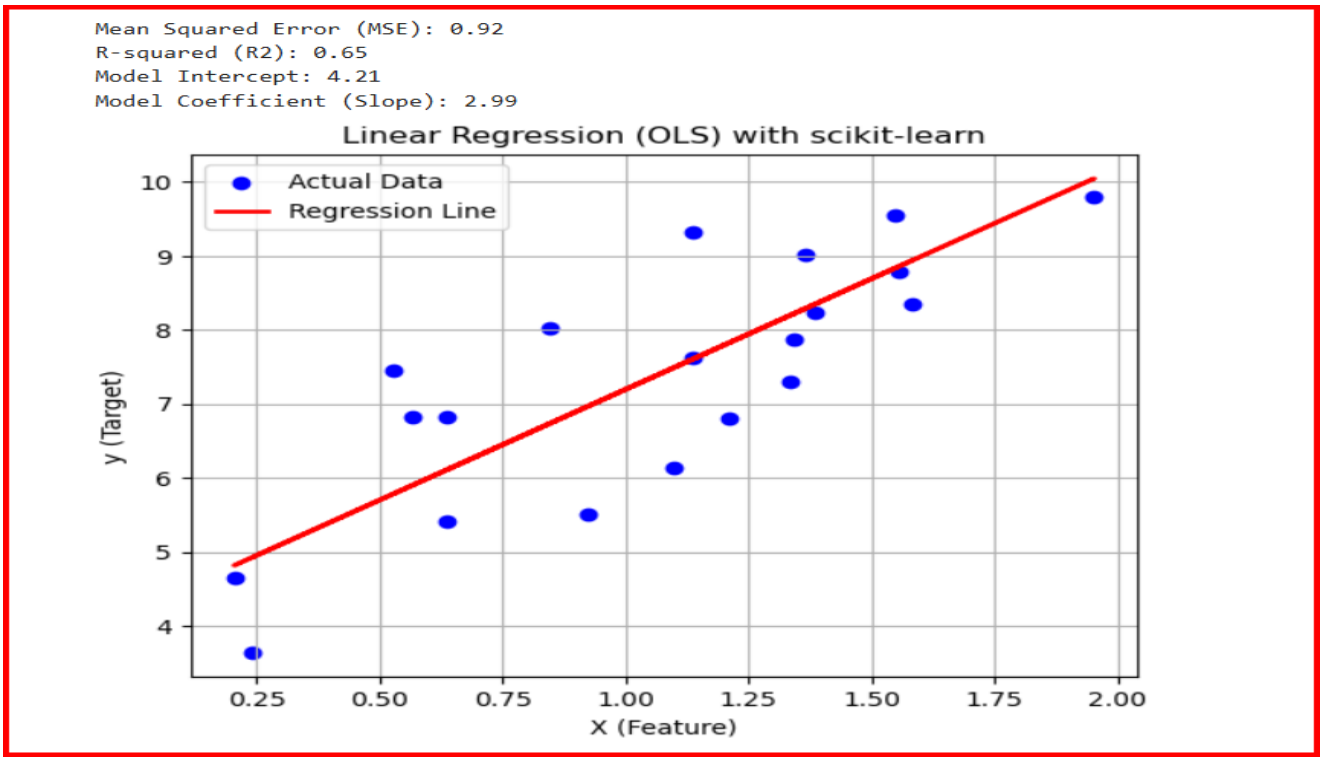
1. Load dataset (we'll use a simple generated dataset).
2. Split data into training and testing sets.
3. Create a Linear Regression model.
4. Train the model on training data.
5. Predict on test data.
6. Evaluate using Mean Squared Error (MSE) and R^2 score.

Code:

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

class LinearRegressionGD:
    def __init__(self, learning_rate=0.01, n_iterations=1000):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights = None
        self.bias = None

    def fit(self, X, y):
        # Initialize weights and bias
        self.weights = np.zeros(X.shape[1])
        self.bias = 0

        # Number of training examples
        m = len(y)

        # Gradient Descent loop
        for _ in range(self.n_iterations):
            # Calculate predictions
```

```
        y_predicted = np.dot(X, self.weights) + self.bias

        # Calculate gradients
        dw = (1/m) * np.dot(X.T, (y_predicted - y))
        db = (1/m) * np.sum(y_predicted - y)

        # Update weights and bias
        self.weights -= self.learning_rate * dw
        self.bias -= self.learning_rate * db

    def predict(self, X):
        return np.dot(X, self.weights) + self.bias

# Example Usage:
if __name__ == "__main__":
    # Generate some sample data
    X = 2 * np.random.rand(100, 1)
    y = 4 + 3 * X + np.random.randn(100, 1)

    # Create and train the model
    model = LinearRegressionGD(learning_rate=0.01, n_iterations=1000)
    model.fit(X, y.flatten()) # Flatten y to a 1D array

    # Make predictions
    y_pred = model.predict(X)

    # Print learned parameters
    print(f"Learned Weights: {model.weights[0]:.4f}")
    print(f"Learned Bias: {model.bias:.4f}")

    # Plot the results
    plt.scatter(X, y, label='Actual Data')
    plt.plot(X, y_pred, color='red', label='Linear Regression Line')
    plt.xlabel('X')
    plt.ylabel('y')
    plt.title('Linear Regression with Gradient Descent')
    plt.legend()
    plt.show()plt.show()
```

Output:



2) Gradient Descent (Manual Implementation)

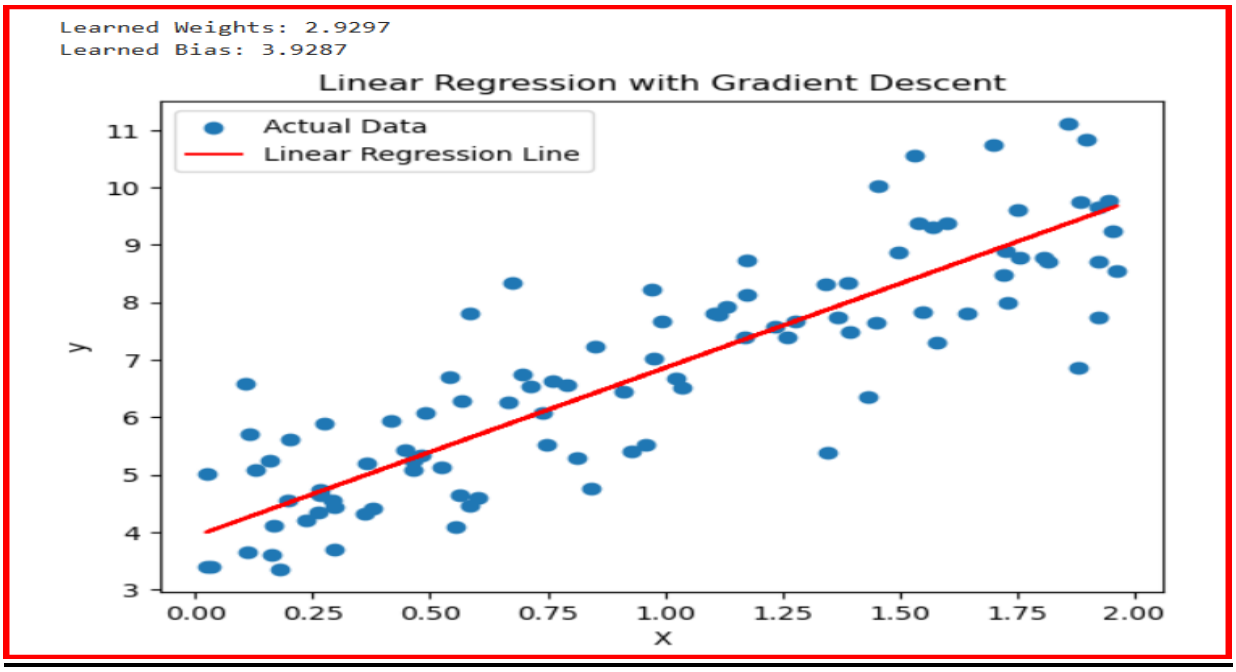
1. Initialize weights and bias.
2. Define learning rate and number of iterations.
3. For each iteration:
4. Predict values using current weights.
5. Calculate the error.
6. Update weights and bias by moving opposite to gradient of loss.
7. After training, predict on test data.
8. Evaluate performance.

Code:

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

class LinearRegressionGD:
    def __init__(self, learning_rate=0.01, n_iterations=1000):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights = None
        self.bias = None

    def fit(self, X, y):
        # Initialize weights and bias
        self.weights = np.zeros(X.shape[1])
        self.bias = 0

        # Number of training examples
        m = len(y)

        # Gradient Descent loop
        for _ in range(self.n_iterations):
            # Calculate predictions
            y_predicted = np.dot(X, self.weights) + self.bias

            # Calculate gradients
            dw = (1/m) * np.dot(X.T, (y_predicted - y))
            db = (1/m) * np.sum(y_predicted - y)

            # Update weights and bias
            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db

    def predict(self, X):
        return np.dot(X, self.weights) + self.bias

# Example Usage:
if __name__ == "__main__":
    # Generate some sample data
    X = 2 * np.random.rand(100, 1)
    y = 4 + 3 * X + np.random.randn(100, 1)
```

```
# Create and train the model
model = LinearRegressionGD(learning_rate=0.01, n_iterations=1000)
model.fit(X, y.flatten()) # Flatten y to a 1D array

# Make predictions
y_pred = model.predict(X)

# Print learned parameters
print(f"Learned Weights: {model.weights[0]:.4f}")
print(f"Learned Bias: {model.bias:.4f}")

# Plot the results
plt.scatter(X, y, label='Actual Data')
plt.plot(X, y_pred, color='red', label='Linear Regression Line')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Linear Regression with Gradient Descent')
plt.legend()
plt.show()
```

Output:



Viva Questions:

1. What is the difference between simple and multiple linear regression?
2. What is the cost function used in linear regression?
3. How is the line of best fit calculated?
4. What is the meaning of coefficients and intercept?
5. What is the R^2 score?
6. How is gradient descent used in linear regression?