SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA 3rd semester

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING (PCA20D07J)

Lab Manual

Lab : 1 Simple AI Techniques implementation

Title

Implementation of Simple AI Techniques

Aim

To understand and implement basic Artificial Intelligence techniques such as searching algorithms (e.g., Breadth-First Search, Depth-First Search) or sorting algorithms in an AI context.

Procedure

1. **Understand the Problem:** Define a simple problem that can be solved using a basic AI technique (e.g., pathfinding in a simple grid, or a simple puzzle).
2. **Choose a Technique:** Select an appropriate AI technique (e.g., BFS or DFS for pathfinding).
3. **Design the Algorithm:** Outline the steps involved in implementing the chosen technique.
4. **Write the Code:** Implement the algorithm in a suitable programming language (e.g., Python).
5. **Test the Implementation:** Run the code with various inputs to verify its correctness.
6. **Analyze Results:** Observe the output and understand how the technique solves the problem.

Source Code

```
# Example: Breadth-First Search (BFS) for a simple graph

graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

def bfs(graph, start_node):
    visited = []
    queue = [start_node]

    while queue:
        node = queue.pop(0)
        if node not in visited:
            visited.append(node)
```

```
            neighbors = graph[node]
            for neighbor in neighbors:
                queue.append(neighbor)
    return visited

# print(bfs(graph, 'A'))
```

Input

```
# For the BFS example:
Starting node: 'A'
Graph:
A -> B, C
B -> D, E
C -> F
E -> F
```

Expected Output

```
# For the BFS example:
['A', 'B', 'C', 'D', 'E', 'F']
```

Lab 2 : Implementation of TicTac-Toe Game and Travelling Sales man problem

Title

Implementation of Tic-Tac-Toe Game and Travelling Salesman Problem

Aim

To implement a simple AI agent for the Tic-Tac-Toe game and to understand and implement a solution for the Travelling Salesman Problem (TSP) using a heuristic or approximate algorithm.

Procedure

For Tic-Tac-Toe:

1. **Game Board Representation:** Design a data structure to represent the 3x3 Tic-Tac-Toe board.
2. **Game Logic:** Implement functions for checking win conditions, making moves, and determining valid moves.
3. **AI Player (Minimax/Heuristic):** Develop an AI player using a minimax algorithm or a simpler heuristic approach to choose optimal moves.
4. **User Interface:** Create a basic command-line or graphical interface for playing the game.

For Travelling Salesman Problem (TSP):

1. **Problem Representation:** Represent cities and distances (e.g., adjacency matrix or list).
2. **Algorithm Selection:** Choose an appropriate algorithm (e.g., Nearest Neighbor, Genetic Algorithm, Simulated Annealing, or a brute-force approach for small instances).
3. **Implementation:** Implement the chosen algorithm to find an optimal or near-optimal tour.
4. **Evaluation:** Test with sample city sets and analyze the tour length.

Source Code

```
# Example: Tic-Tac-Toe (simplified AI turn)
# This is a very basic example, a full game would be much longer.

board = [' ' for _ in range(9)]

def print_board(board):
    for i in range(0, 9, 3):
        print(f"| {board[i]} | {board[i+1]} | {board[i+2]} |")
        if i < 6:
            print("-------------")

def check_win(board, player):
    # Check rows, columns, and diagonals for a win
    win_conditions = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],  # Rows
        [0, 3, 6], [1, 4, 7], [2, 5, 8],  # Columns
        [0, 4, 8], [2, 4, 6]              # Diagonals
    ]
    for condition in win_conditions:
        if all(board[i] == player for i in condition):
```

```
            return True
    return False

def ai_move(board):
    # Simple AI: choose first available spot
    for i in range(9):
        if board[i] == ' ':
            return i
    return -1 # No move possible

# Example TSP (Nearest Neighbor heuristic)
import math

def calculate_distance(p1, p2):
    return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

def nearest_neighbor_tsp(cities):
    num_cities = len(cities)
    if num_cities == 0:
        return [], 0

    start_city = 0 # Start from the first city
    current_path = [start_city]
    unvisited_cities = set(range(num_cities))
    unvisited_cities.remove(start_city)
    total_distance = 0

    while unvisited_cities:
        last_city = current_path[-1]
        next_city = -1
        min_dist = float('inf')

        for city_idx in unvisited_cities:
            dist = calculate_distance(cities[last_city], cities[city_idx])
            if dist < min_dist:
                min_dist = dist
                next_city = city_idx

        current_path.append(next_city)
        unvisited_cities.remove(next_city)
        total_distance += min_dist

    # Complete the tour by returning to the start city
    total_distance += calculate_distance(cities[current_path[-1]],
cities[start_city])
    current_path.append(start_city)
    return current_path, total_distance

# cities_coords = [(0,0), (1,3), (4,1), (2,5)]
# path, dist = nearest_neighbor_tsp(cities_coords)
# print(f"TSP Path: {path}, Total Distance: {dist}")
```

Input

For Tic-Tac-Toe:

```
User's moves (e.g., 0, 4, 2)
```

For TSP:

```
List of city coordinates:
[(0,0), (1,3), (4,1), (2,5)]
```

Expected Output

For Tic-Tac-Toe:

```
Initial Board:
|   |   |   |
-------------
|   |   |   |
-------------
|   |   |   |

(After some moves)
| X | O |   |
-------------
|   | X |   |
-------------
|   |   | O |

AI makes a move...
```

For TSP:

```
TSP Path: [0, 1, 3, 2, 0], Total Distance: 13.06... (approx)
```

Lab 3 : Implementation of intelligent agents

Title

Implementation of Intelligent Agents

Aim

To understand the concept of intelligent agents and implement a simple agent (e.g., a vacuum cleaner agent or a simple reflex agent) that perceives its environment and acts rationally.

Procedure

1. **Define Environment:** Create a simple environment for the agent (e.g., a 2-square vacuum world, or a simple game environment).
2. **Define Agent Percepts:** Determine what information the agent can perceive from its environment.
3. **Define Agent Actions:** List the actions the agent can perform.
4. **Implement Agent Function:** Write the agent's decision-making logic (mapping percepts to actions).
5. **Simulate Agent Behavior:** Run the simulation and observe the agent's actions and their impact on the environment.
6. **Evaluate Performance:** Assess how well the agent achieves its goals.

Source Code

```
# Example: Simple Vacuum Cleaner Agent

class VacuumEnvironment:
    def __init__(self):
        self.location_A = 'Dirty'
        self.location_B = 'Dirty'
        self.current_location = 'A'

    def display(self):
        print(f"Location A: {self.location_A}, Location B: {self.location_B},
Current: {self.current_location}")

class VacuumAgent:
    def __init__(self, environment):
        self.environment = environment

    def perceive(self):
        if self.environment.current_location == 'A':
            return self.environment.location_A, 'A'
        else:
            return self.environment.location_B, 'B'

    def act(self, percept):
        status, location = percept
        if status == 'Dirty':
            print(f"Sucking dirt at {location}")
            if location == 'A':
                self.environment.location_A = 'Clean'
            else:
                self.environment.location_B = 'Clean'
            return 'Suck'
```

```
        elif location == 'A':
            print("Moving to B")
            self.environment.current_location = 'B'
            return 'Move Right'
        elif location == 'B':
            print("Moving to A")
            self.environment.current_location = 'A'
            return 'Move Left'

# env = VacuumEnvironment()
# agent = VacuumAgent(env)
#
# for _ in range(5): # Simulate 5 steps
#     env.display()
#     percept = agent.perceive()
#     action = agent.act(percept)
#     print(f"Agent performs: {action}\n")
```

Input

```
Initial environment state:
Location A: Dirty, Location B: Dirty, Current: A
```

Expected Output

```
Location A: Dirty, Location B: Dirty, Current: A
Sucking dirt at A
Agent performs: Suck

Location A: Clean, Location B: Dirty, Current: A
Moving to B
Agent performs: Move Right

Location A: Clean, Location B: Dirty, Current: B
Sucking dirt at B
Agent performs: Suck

Location A: Clean, Location B: Clean, Current: B
Moving to A
Agent performs: Move Left

Location A: Clean, Location B: Clean, Current: A
Moving to B
Agent performs: Move Right
```

Lab : 4 Knowledge implementation

Title

Implementation of Knowledge Representation

Aim

To implement a simple knowledge base using a chosen representation method (e.g., propositional logic, semantic networks, or frames) and perform basic querying or inference.

Procedure

1. **Choose Representation:** Select a method for representing knowledge (e.g., a set of propositional logic sentences, a simple semantic network structure).
2. **Define Knowledge:** Create a small set of facts and rules related to a specific domain (e.g., animal characteristics, family relationships).
3. **Implement Data Structure:** Design data structures to store the knowledge.
4. **Implement Query/Inference:** Write functions to query the knowledge base or perform simple inference (e.g., checking if a statement is true, finding relationships).
5. **Test:** Verify that queries and inferences yield correct results.

Source Code

```
# Example: Simple Knowledge Base using Python dictionary (facts)

knowledge_base = {
    "mammal": ["dog", "cat", "elephant"],
    "bird": ["sparrow", "eagle"],
    "can_fly": ["sparrow", "eagle"],
    "has_fur": ["dog", "cat"],
    "eats": {
        "dog": "meat",
        "cat": "fish",
        "sparrow": "seeds"
    }
}

def is_a(entity, category):
    return entity in knowledge_base.get(category, [])

def what_eats(entity):
    return knowledge_base["eats"].get(entity, "Unknown")

# print(is_a("dog", "mammal"))
# print(is_a("sparrow", "mammal"))
# print(what_eats("cat"))
```

Input

```
Query 1: Is "dog" a "mammal"?
Query 2: Is "sparrow" a "mammal"?
Query 3: What does "cat" eat?
```

Expected Output

```
True
False
fish
```

Lab : 5 Implementations of FOPL and Rules

Title

Implementations of First-Order Predicate Logic (FOPL) and Rules

Aim

To understand and implement basic concepts of First-Order Predicate Logic (FOPL) and rule-based systems, including representing facts, rules, and performing simple forward or backward chaining.

Procedure

1. **FOPL Representation:** Define a way to represent predicates, constants, variables, and quantifiers in your chosen programming language.
2. **Rule Definition:** Implement a structure for defining rules (e.g., `IF condition THEN conclusion`).
3. **Knowledge Base:** Populate a knowledge base with FOPL statements (facts) and rules.
4. **Inference Mechanism:** Implement a simple forward-chaining or backward-chaining inference engine to derive new conclusions or answer queries.
5. **Testing:** Test the system with various facts and rules to ensure correct inference.

Source Code

```
# Example: Simple Rule-Based System (Forward Chaining)

facts = {
    "has_feathers": ["Tweety"],
    "can_fly": [],
    "is_bird": []
}

rules = [
    {"if": ["has_feathers(X)"], "then": "is_bird(X)"},
    {"if": ["is_bird(X)", "can_fly(X)"], "then": "is_flying_bird(X)"} #
Placeholder for more complex rules
]

def apply_rules_forward(facts, rules, max_iterations=5):
    inferred_facts = set()
    for category, items in facts.items():
        for item in items:
            inferred_facts.add(f"{category}({item})")

    for _ in range(max_iterations): # Iterate to allow chaining
        new_facts_inferred_this_iteration = False
        for rule in rules:
            conditions_met = True
            bindings = {} # Simple binding for single variable X

            # Check if all 'if' conditions are met
            for condition in rule["if"]:
                predicate, entity = condition.split('(')[0],
condition.split('(')[1][:-1]
                if entity == 'X':
                    # Try to find a binding for X
                    found_binding = False
```

```
                    for existing_fact in inferred_facts:
                        if existing_fact.startswith(predicate):
                            bound_entity = existing_fact.split('(')[1][:-1]
                            if 'X' not in bindings or bindings['X'] ==
bound_entity:
                                bindings['X'] = bound_entity
                                found_binding = True
                                break
                    if not found_binding:
                        conditions_met = False
                        break
                elif f"{predicate}({entity})" not in inferred_facts:
                    conditions_met = False
                    break

            if conditions_met:
                # Apply 'then' part
                then_part = rule["then"]
                if 'X' in bindings:
                    then_part = then_part.replace('(X)', f"({bindings['X']})")

                if then_part not in inferred_facts:
                    inferred_facts.add(then_part)
                    new_facts_inferred_this_iteration = True
        if not new_facts_inferred_this_iteration:
            break # No new facts inferred, stop

    return inferred_facts

# inferred = apply_rules_forward(facts, rules)
# print(inferred)
```

Input

```
Initial facts:
- Tweety has feathers.

Rules:
- IF an entity has feathers THEN it is a bird.
- IF an entity is a bird AND can fly THEN it is a flying bird.
```

Expected Output

```
{'has_feathers(Tweety)', 'is_bird(Tweety)'}
```

Lab : 6 Implementation of Ontology and FOL

Title

Implementation of Ontology and First-Order Logic (FOL)

Aim

To understand and implement basic concepts of ontology representation and its relation to First-Order Logic (FOL), focusing on defining classes, properties, and instances within a simple domain.

Procedure

1. **Ontology Design:** Choose a small domain (e.g., a simple university domain with professors, students, courses) and define its key concepts (classes), relationships (properties), and instances.
2. **FOL Mapping:** Express these ontological elements as FOL statements (e.g., `Professor(John)`, `teaches(John, AI_Course)`).
3. **Implementation:** Use a programming language to represent these FOL statements (e.g., as lists of tuples, or using a simple custom class structure).
4. **Querying:** Implement functions to query the ontology (e.g., "Who teaches AI_Course?", "Is X a student?").
5. **Consistency Check (Optional):** Briefly discuss how one might check for consistency or infer new facts.

Source Code

```
# Example: Simple Ontology Representation in Python

# Classes/Concepts
classes = ["Professor", "Student", "Course"]

# Instances (Individuals)
individuals = {
    "Professor": ["Dr. Smith", "Dr. Jones"],
    "Student": ["Alice", "Bob", "Charlie"],
    "Course": ["AI_Course", "ML_Course"]
}

# Properties (Relationships) - represented as tuples (predicate, subject,
object)
properties = [
    ("teaches", "Dr. Smith", "AI_Course"),
    ("teaches", "Dr. Jones", "ML_Course"),
    ("enrolls_in", "Alice", "AI_Course"),
    ("enrolls_in", "Bob", "ML_Course"),
    ("enrolls_in", "Charlie", "AI_Course")
]

# Functions to query the ontology
def get_teachers_of_course(course_name):
    teachers = []
    for prop, subject, obj in properties:
        if prop == "teaches" and obj == course_name:
            teachers.append(subject)
    return teachers
```

```
def is_instance_of(individual, class_name):
    return individual in individuals.get(class_name, [])

# print(get_teachers_of_course("AI_Course"))
# print(is_instance_of("Alice", "Student"))
# print(is_instance_of("Dr. Smith", "Student"))
```

Input

```
Ontology definitions:
Classes: Professor, Student, Course
Individuals: Dr. Smith (Professor), Alice (Student), AI_Course (Course)
Properties: Dr. Smith teaches AI_Course, Alice enrolls_in AI_Course

Queries:
1. Who teaches "AI_Course"?
2. Is "Alice" a "Student"?
3. Is "Dr. Smith" a "Student"?
```

Expected Output

```
['Dr. Smith']
True
False
```

Lab : 7 Concept Learning task

Title

Concept Learning Task Implementation

Aim

To implement a basic concept learning algorithm (e.g., Find-S or Candidate-Elimination - though Candidate-Elimination is a separate lab, a simpler version could be introduced here) to learn a target concept from a set of examples.

Procedure

1. **Define Hypothesis Space:** Understand the representation of hypotheses (e.g., conjunction of attribute values).
2. **Prepare Training Data:** Create a dataset of examples, each with attributes and a classification (positive or negative for the target concept).
3. **Implement Learning Algorithm:** Write code for a concept learning algorithm (e.g., Find-S to find the most specific hypothesis).
4. **Train the Model:** Run the algorithm on the training data to learn the concept.
5. **Test and Evaluate:** Test the learned hypothesis on new examples and discuss its accuracy.

Source Code

```
# Example: Find-S Algorithm for Concept Learning

# Training data: (Sky, AirTemp, Humidity, Wind, Water, Forecast, EnjoySport)
# Attributes: Sky (Sunny, Cloudy, Rainy), AirTemp (Warm, Cold), Humidity
(Normal, High),
#             Wind (Strong, Weak), Water (Warm, Cold), Forecast (Same, Change)
# Target Concept: EnjoySport (Yes, No)

data = [
    (['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')
]

def find_s(training_examples):
    # Initialize hypothesis with the first positive example
    hypothesis = ['?', '?', '?', '?', '?', '?'] # Most general hypothesis

    for example, label in training_examples:
        if label == 'Yes':
            if hypothesis == ['?', '?', '?', '?', '?', '?']: # If first positive
example
                hypothesis = list(example)
            else:
                for i in range(len(hypothesis)):
                    if hypothesis[i] != example[i]:
                        hypothesis[i] = '?'
    return hypothesis

# learned_hypothesis = find_s(data)
# print(learned_hypothesis)
```

Input

```
Training Examples:
1. (Sunny, Warm, Normal, Strong, Warm, Same), Yes
2. (Sunny, Warm, High, Strong, Warm, Same), Yes
3. (Rainy, Cold, High, Strong, Warm, Change), No
4. (Sunny, Warm, High, Strong, Cool, Change), Yes
```

Expected Output

```
['Sunny', 'Warm', '?', 'Strong', '?', '?']
```

Lab : 8 Design a Learning System

Title

Design of a Learning System

Aim

To design a conceptual learning system for a specific task, outlining its components (e.g., performance element, learning element, critic, problem generator) and how they interact. This lab is more about design than coding.

Procedure

1. **Choose a Task:** Select a real-world problem that can be addressed by a learning system (e.g., playing chess, medical diagnosis, spam detection).
2. **Define Performance Element:** Describe how the system will perform the task.
3. **Identify Learning Element:** Explain how the system will improve its performance over time.
4. **Design Critic:** Detail how the system will evaluate its own performance and provide feedback to the learning element.
5. **Develop Problem Generator:** Describe how the system will generate new learning experiences or explore the environment.
6. **Illustrate Interaction:** Draw a diagram or write a detailed explanation of how these components interact.

Source Code

```
# No specific source code for this lab, as it's a design exercise.
# However, you could provide a conceptual class structure if desired.

class LearningSystem:
    def __init__(self, environment, goals):
        self.environment = environment
        self.goals = goals
        self.performance_element = None # e.g., a chess player
        self.learning_element = None    # e.g., an algorithm to update strategy
        self.critic = None              # e.g., evaluates game outcome
        self.problem_generator = None   # e.g., generates new game scenarios

    def operate(self):
        # Conceptual flow of the system
        pass

# Example: Conceptual components for a Chess Learning System
# performance_element: Chess playing engine
# learning_element: Reinforcement learning algorithm
# critic: Evaluates game result (win/loss/draw)
# problem_generator: Plays against itself or other players
```

Input

```
Conceptual task: Designing a learning system for playing Chess.
```

Expected Output

A detailed design document or diagram outlining the components and their interactions for a Chess playing learning system.

Lab : 9 Implementation of candidate elimination algorithm

Title

Implementation of Candidate-Elimination Algorithm

Aim

To implement the Candidate-Elimination algorithm for concept learning, which maintains a set of all hypotheses consistent with the training examples.

Procedure

1. **Define Hypothesis Space:** Understand the representation of hypotheses (e.g., conjunction of attribute values).
2. **Prepare Training Data:** Create a dataset of examples with attributes and classifications (positive/negative).
3. **Initialize General (G) and Specific (S) Boundaries:** Set G0 to the most general hypothesis and S0 to the most specific hypothesis.
4. **Iterate Through Examples:**
    - For each positive example, generalize hypotheses in S and remove inconsistent hypotheses from G.
    - For each negative example, specialize hypotheses in G and remove inconsistent hypotheses from S.
5. **Implement Algorithm:** Write code for the Candidate-Elimination algorithm.
6. **Test and Evaluate:** Run the algorithm on the training data and observe the final version space (S and G sets).

Source Code

```
# Example: Candidate-Elimination Algorithm (Simplified)

# Attributes: Sky (Sunny, Cloudy), AirTemp (Warm, Cold), Humidity (Normal, High)
# Target Concept: EnjoySport (Yes, No)

# Helper function to check if a hypothesis covers an example
def covers(hypothesis, example):
    for i in range(len(hypothesis)):
        if hypothesis[i] != '?' and hypothesis[i] != example[i]:
            return False
    return True

# Helper function to generalize a hypothesis
def generalize(hypothesis, example):
    new_hypothesis = list(hypothesis)
    for i in range(len(new_hypothesis)):
        if new_hypothesis[i] == '?':
            continue
        elif new_hypothesis[i] != example[i]:
            new_hypothesis[i] = '?'
    return tuple(new_hypothesis)

# Helper function to specialize a hypothesis
def specialize(hypothesis, example, attributes_domain):
    specialized_hypotheses = set()
    for i in range(len(hypothesis)):
```

```
        if hypothesis[i] == '?':
            for val in attributes_domain[i]:
                if val != example[i]:
                    new_h = list(hypothesis)
                    new_h[i] = val
                    specialized_hypotheses.add(tuple(new_h))
        elif hypothesis[i] == example[i]:
            # If current attribute matches, specialize by making it more
specific
            # This part is more complex in full CE, often involves removing the
current value
            # and replacing with other values from domain, or making it empty.
            # For simplicity, we'll skip this specific case for now.
            pass
    return specialized_hypotheses


def candidate_elimination(training_examples, attributes_domain):
    # Initialize G and S
    S = {('0', '0', '0')} # Most specific hypothesis (all attributes must match)
    G = {('?', '?', '?')} # Most general hypothesis (any value)

    for example, label in training_examples:
        if label == 'Yes': # Positive example
            # Remove hypotheses in G that do not cover the example
            G = {g for g in G if covers(g, example)}
            # For each s in S that does not cover the example, generalize s
            new_S = set()
            for s in S:
                if not covers(s, example):
                    new_S.add(generalize(s, example))
                else:
                    new_S.add(s)
            S = new_S
            # Remove redundant hypotheses in S (subsumed by others) - omitted
for simplicity
            # Remove hypotheses in S that are inconsistent with G - omitted for
simplicity
        else: # Negative example
            # Remove hypotheses in S that cover the example
            S = {s for s in S if not covers(s, example)}
            # For each g in G that covers the example, specialize g
            new_G = set()
            for g in G:
                if covers(g, example):
                    # Specialize g to exclude the negative example
                    # This is the most complex part of CE.
                    # For simplicity, we'll just add a placeholder for
specialization.
                    # A full implementation would generate new, more specific
hypotheses.
                    # For now, we'll just remove 'g' if it covers the negative
example.
                    pass # Placeholder for actual specialization
                else:
                    new_G.add(g)
            G = new_G
            # Remove redundant hypotheses in G (subsumed by others) - omitted
for simplicity
            # Remove hypotheses in G that are inconsistent with S - omitted for
simplicity
    return S, G

# Simplified attribute domains for example
# attributes_domain = {
#     0: ['Sunny', 'Cloudy'],
#     1: ['Warm', 'Cold'],
```

```
#     2: ['Normal', 'High']
# }
#
# training_data = [
#     (('Sunny', 'Warm', 'Normal'), 'Yes'),
#     (('Sunny', 'Warm', 'High'), 'Yes'),
#     (('Sunny', 'Cold', 'High'), 'No'),
#     (('Cloudy', 'Warm', 'High'), 'Yes')
# ]
#
# S_final, G_final = candidate_elimination(training_data, attributes_domain)
# print("Final S:", S_final)
# print("Final G:", G_final)
```

Input

```
Training Examples:
1. (Sunny, Warm, Normal), Yes
2. (Sunny, Warm, High), Yes
3. (Sunny, Cold, High), No
4. (Cloudy, Warm, High), Yes

Attribute Domains:
Sky: {Sunny, Cloudy}
AirTemp: {Warm, Cold}
Humidity: {Normal, High}
```

Expected Output

```
Final S: {('Sunny', 'Warm', '?'), ('?', 'Warm', 'High')} (Example, actual output
depends on full implementation)
Final G: {('?', '?', '?')} (Example, actual output depends on full
implementation)
```

Lab : 10 Decision tree implementation

Title

Decision Tree Implementation

Aim

To implement a decision tree algorithm (e.g., a simplified version of ID3 or C4.5) for classification tasks.

Procedure

1. **Understand Decision Trees:** Grasp the concepts of nodes, branches, leaves, and splitting criteria (e.g., Information Gain, Gini Impurity).
2. **Prepare Dataset:** Obtain a dataset with categorical or numerical features and a target class.
3. **Implement Splitting Criterion:** Write a function to calculate the chosen splitting criterion (e.g., Information Gain).
4. **Implement Tree Building:** Develop a recursive function to build the decision tree:
    - Select the best attribute to split on.
    - Create child nodes for each value of the attribute.
    - Recursively build subtrees until a stopping condition is met (e.g., all examples in a node belong to the same class, no more attributes, or max depth reached).
5. **Implement Prediction:** Write a function to traverse the tree and make predictions for new examples.
6. **Test and Evaluate:** Test the decision tree on unseen data and evaluate its performance.

Source Code

```
# Example: Simplified Decision Tree (using Information Gain)

import math
from collections import Counter

# Calculate entropy of a dataset
def entropy(labels):
    n_labels = len(labels)
    if n_labels == 0:
        return 0
    counts = Counter(labels)
    ent = 0
    for count in counts.values():
        prob = count / n_labels
        ent -= prob * math.log2(prob)
    return ent

# Calculate information gain
def information_gain(data, attribute_index, target_index):
    total_entropy = entropy([row[target_index] for row in data])

    values = [row[attribute_index] for row in data]
    unique_values = set(values)

    weighted_entropy = 0
    for value in unique_values:
        subset = [row for row in data if row[attribute_index] == value]
```

```
        subset_labels = [row[target_index] for row in subset]
        weighted_entropy += (len(subset) / len(data)) * entropy(subset_labels)

    return total_entropy - weighted_entropy

# Simplified Decision Tree Node
class Node:
    def __init__(self, attribute=None, value=None, results=None, children=None):
        self.attribute = attribute    # Index of the attribute to split on
        self.value = value            # Value of the attribute if this is a
branch from parent
        self.results = results        # Dictionary of results if this is a leaf
node (e.g., {'Yes': 5, 'No': 2})
        self.children = children if children is not None else {} # Dictionary of
child nodes {attribute_value: Node}

# Build the decision tree (simplified recursive function)
def build_tree(data, attributes, target_index):
    labels = [row[target_index] for row in data]

    # Base cases:
    # 1. All examples have the same label
    if len(set(labels)) == 1:
        return Node(results=Counter(labels))

    # 2. No more attributes to split on
    if not attributes:
        return Node(results=Counter(labels))

    # Find the best attribute to split on
    best_gain = -1
    best_attribute = None

    for attr_idx in attributes:
        gain = information_gain(data, attr_idx, target_index)
        if gain > best_gain:
            best_gain = gain
            best_attribute = attr_idx

    if best_attribute is None: # No gain, make it a leaf
        return Node(results=Counter(labels))

    # Create a node for the best attribute
    node = Node(attribute=best_attribute)

    # Remove the chosen attribute from the list for sub-problems
    remaining_attributes = [attr for attr in attributes if attr !=
best_attribute]

    # Create branches for each value of the best attribute
    attribute_values = set([row[best_attribute] for row in data])
    for value in attribute_values:
        subset = [row for row in data if row[best_attribute] == value]
        if subset: # Only build child if subset is not empty
            node.children[value] = build_tree(subset, remaining_attributes,
target_index)
        else: # Handle empty subset (e.g., by making it a leaf with majority
class)
            node.children[value] = Node(results=Counter(labels)) # Fallback to
parent's majority

    return node

# Example data: (Outlook, Temperature, Humidity, Wind, PlayTennis)
# target_index = 4 (PlayTennis)
# attributes = [0, 1, 2, 3] (Outlook, Temperature, Humidity, Wind)
#
```

```
# play_tennis_data = [
#     ['Sunny', 'Hot', 'High', 'Weak', 'No'],
#     ['Sunny', 'Hot', 'High', 'Strong', 'No'],
#     ['Overcast', 'Hot', 'High', 'Weak', 'Yes'],
#     ['Rain', 'Mild', 'High', 'Weak', 'Yes'],
#     ['Rain', 'Cool', 'Normal', 'Weak', 'Yes'],
#     ['Rain', 'Cool', 'Normal', 'Strong', 'No'],
#     ['Overcast', 'Cool', 'Normal', 'Strong', 'Yes'],
#     ['Sunny', 'Mild', 'High', 'Weak', 'No'],
#     ['Sunny', 'Cool', 'Normal', 'Weak', 'Yes'],
#     ['Rain', 'Mild', 'Normal', 'Weak', 'Yes'],
#     ['Sunny', 'Mild', 'Normal', 'Strong', 'Yes'],
#     ['Overcast', 'Mild', 'High', 'Strong', 'Yes'],
#     ['Overcast', 'Hot', 'Normal', 'Weak', 'Yes'],
#     ['Rain', 'Mild', 'High', 'Strong', 'No']
# ]
#
# # Example usage:
# # tree = build_tree(play_tennis_data, [0, 1, 2, 3], 4)
# # print(tree.attribute) # Should be the index of 'Outlook'
```

Input

```
Training Data (Play Tennis Example):
Outlook, Temperature, Humidity, Wind, PlayTennis
Sunny, Hot, High, Weak, No
Sunny, Hot, High, Strong, No
Overcast, Hot, High, Weak, Yes
Rain, Mild, High, Weak, Yes
Rain, Cool, Normal, Weak, Yes
Rain, Cool, Normal, Strong, No
Overcast, Cool, Normal, Strong, Yes
Sunny, Mild, High, Weak, No
Sunny, Cool, Normal, Weak, Yes
Rain, Mild, Normal, Weak, Yes
Sunny, Mild, Normal, Strong, Yes
Overcast, Mild, High, Strong, Yes
Overcast, Hot, Normal, Weak, Yes
Rain, Mild, High, Strong, No
```

Expected Output

```
A representation of the decision tree structure (e.g., by printing node
attributes and children, or a graphical representation if implemented).
For the Play Tennis example, the root node would likely be 'Outlook'.
```

Lab : 11 Implementation of Decision tree and K - Mean algorithm

Title

Implementation of Decision Tree and K-Means Algorithm

Aim

To implement both a decision tree for classification and the K-Means algorithm for clustering, demonstrating two fundamental machine learning paradigms.

Procedure

For Decision Tree:

(Refer to Lab 10 procedure)

For K-Means Algorithm:

1. **Understand K-Means:** Grasp the concept of partitioning data into K clusters based on similarity.
2. **Prepare Dataset:** Obtain a numerical dataset for clustering.
3. **Initialize Centroids:** Randomly select K data points as initial cluster centroids.
4. **Iterate (Assignment & Update):**
    - **Assignment Step:** Assign each data point to the closest centroid.
    - **Update Step:** Recalculate the centroids as the mean of all data points assigned to that cluster.
5. **Stopping Condition:** Repeat until centroids no longer change significantly or a maximum number of iterations is reached.
6. **Implement K-Means:** Write code for the K-Means algorithm.
7. **Visualize/Evaluate:** Visualize the clusters (if 2D/3D data) or evaluate using metrics like inertia.

Source Code

```
# Example: K-Means Algorithm

import numpy as np

def euclidean_distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2)**2))

def kmeans(data, k, max_iterations=100):
    # 1. Initialize centroids randomly
    # Ensure k is not greater than the number of data points
    if k > len(data):
        print("Error: k cannot be greater than the number of data points.")
        return [], []

    # Randomly select k data points as initial centroids
    rng = np.random.default_rng()
    initial_indices = rng.choice(len(data), k, replace=False)
    centroids = data[initial_indices]
```

```
    for _ in range(max_iterations):
        # 2. Assignment Step: Assign each data point to the closest centroid
        clusters = [[] for _ in range(k)]
        for i, point in enumerate(data):
            distances = [euclidean_distance(point, centroid) for centroid in
centroids]
            closest_centroid_idx = np.argmin(distances)
            clusters[closest_centroid_idx].append(i) # Store index of data point

        # 3. Update Step: Recalculate centroids
        new_centroids = np.array([
            np.mean(data[cluster_indices], axis=0) if cluster_indices else
centroids[j]
            for j, cluster_indices in enumerate(clusters)
        ])

        # Check for convergence
        if np.allclose(centroids, new_centroids):
            break
        centroids = new_centroids

    # Assign final labels to data points
    labels = np.zeros(len(data), dtype=int)
    for i, point in enumerate(data):
        distances = [euclidean_distance(point, centroid) for centroid in
centroids]
        labels[i] = np.argmin(distances)

    return centroids, labels

# Example Data for K-Means
# data_points = np.array([
#     [1, 2], [1.5, 1.8], [5, 8], [8, 8], [1, 0.6],
#     [9, 11], [10, 12], [8, 9], [10, 2], [9, 3]
# ])
#
# k_value = 3
# final_centroids, cluster_labels = kmeans(data_points, k_value)
# print("Final Centroids:\n", final_centroids)
# print("Cluster Labels:", cluster_labels)
```

Input

For Decision Tree:

(Same as Lab 10)

For K-Means:

```
Dataset of 2D points:
[[1, 2], [1.5, 1.8], [5, 8], [8, 8], [1, 0.6],
[9, 11], [10, 12], [8, 9], [10, 2], [9, 3]]
Number of clusters (K): 3
```

Expected Output

For Decision Tree:

(Same as Lab 10)

For K-Means:

```
Final Centroids:
(Example values, will vary based on initialization)
[[ 1.16666667  1.46666667]
 [ 7.66666667  9.33333333]
 [ 9.66666667  2.66666667]]

Cluster Labels:
[0 0 1 1 0 1 1 1 2 2] (Example, will vary based on initialization)
```

Lab : 12 Implementation of ID3 algorithm

Title

Implementation of ID3 Algorithm

Aim

To implement the ID3 (Iterative Dichotomiser 3) algorithm specifically for building decision trees, focusing on its use of Information Gain to select optimal splitting attributes.

Procedure

1. **Review ID3 Principles:** Reiterate the core concepts of ID3, including its reliance on entropy and information gain.
2. **Prepare Dataset:** Use a dataset with categorical features and a discrete target class.
3. **Implement Entropy Calculation:** Write a function to calculate the entropy of a set of labels.
4. **Implement Information Gain Calculation:** Write a function to calculate the information gain for a given attribute.
5. **Implement Tree Construction:** Develop a recursive function that:
    - Calculates information gain for all remaining attributes.
    - Selects the attribute with the highest information gain as the splitting criterion for the current node.
    - Creates child nodes for each unique value of the chosen attribute.
    - Recursively calls itself for each child node with the subset of data corresponding to that branch and the remaining attributes.
    - Defines stopping conditions (e.g., all examples in a subset belong to the same class, no more attributes to split on).
6. **Implement Prediction:** Create a function to classify new examples by traversing the built tree.
7. **Test and Evaluate:** Test the ID3 tree with new data and analyze its performance.

Source Code

```
# Example: ID3 Algorithm (similar to Lab 10, but explicitly named ID3)

import math
from collections import Counter

# Calculate entropy of a dataset
def entropy(labels):
    n_labels = len(labels)
    if n_labels == 0:
        return 0
    counts = Counter(labels)
    ent = 0
    for count in counts.values():
        prob = count / n_labels
        ent -= prob * math.log2(prob)
    return ent

# Calculate information gain
def information_gain(data, attribute_index, target_index):
    total_entropy = entropy([row[target_index] for row in data])
```

```
    values = [row[attribute_index] for row in data]
    unique_values = set(values)

    weighted_entropy = 0
    for value in unique_values:
        subset = [row for row in data if row[attribute_index] == value]
        subset_labels = [row[target_index] for row in subset]
        weighted_entropy += (len(subset) / len(data)) * entropy(subset_labels)

    return total_entropy - weighted_entropy

# ID3 Decision Tree Node
class ID3Node:
    def __init__(self, attribute=None, value=None, results=None, children=None):
        self.attribute = attribute    # Index of the attribute to split on
        self.value = value            # Value of the attribute if this is a
branch from parent
        self.results = results        # Dictionary of results if this is a leaf
node (e.g., {'Yes': 5, 'No': 2})
        self.children = children if children is not None else {} # Dictionary of
child nodes {attribute_value: ID3Node}

# Build the ID3 tree (recursive function)
def build_id3_tree(data, attributes, target_index):
    labels = [row[target_index] for row in data]

    # Base cases:
    # 1. All examples have the same label
    if len(set(labels)) == 1:
        return ID3Node(results=Counter(labels))

    # 2. No more attributes to split on
    if not attributes:
        return ID3Node(results=Counter(labels))

    # Find the best attribute to split on
    best_gain = -1
    best_attribute = None

    for attr_idx in attributes:
        gain = information_gain(data, attr_idx, target_index)
        if gain > best_gain:
            best_gain = gain
            best_attribute = attr_idx

    if best_attribute is None or best_gain == 0: # No gain, make it a leaf
        return ID3Node(results=Counter(labels))

    # Create a node for the best attribute
    node = ID3Node(attribute=best_attribute)

    # Remove the chosen attribute from the list for sub-problems
    remaining_attributes = [attr for attr in attributes if attr !=
best_attribute]

    # Create branches for each value of the best attribute
    attribute_values = set([row[best_attribute] for row in data])
    for value in attribute_values:
        subset = [row for row in data if row[best_attribute] == value]
        if subset: # Only build child if subset is not empty
            node.children[value] = build_id3_tree(subset, remaining_attributes,
target_index)
        else: # Handle empty subset (e.g., by making it a leaf with majority
class from parent)
            node.children[value] = ID3Node(results=Counter(labels)) # Fallback
to parent's majority
```

```
    return node

# Example data (same as Lab 10)
# play_tennis_data = [
#     ['Sunny', 'Hot', 'High', 'Weak', 'No'],
#     ['Sunny', 'Hot', 'High', 'Strong', 'No'],
#     ['Overcast', 'Hot', 'High', 'Weak', 'Yes'],
#     ['Rain', 'Mild', 'High', 'Weak', 'Yes'],
#     ['Rain', 'Cool', 'Normal', 'Weak', 'Yes'],
#     ['Rain', 'Cool', 'Normal', 'Strong', 'No'],
#     ['Overcast', 'Cool', 'Normal', 'Strong', 'Yes'],
#     ['Sunny', 'Mild', 'High', 'Weak', 'No'],
#     ['Sunny', 'Cool', 'Normal', 'Weak', 'Yes'],
#     ['Rain', 'Mild', 'Normal', 'Weak', 'Yes'],
#     ['Sunny', 'Mild', 'Normal', 'Strong', 'Yes'],
#     ['Overcast', 'Mild', 'High', 'Strong', 'Yes'],
#     ['Overcast', 'Hot', 'Normal', 'Weak', 'Yes'],
#     ['Rain', 'Mild', 'High', 'Strong', 'No']
# ]
#
# # target_index = 4 (PlayTennis)
# # attributes = [0, 1, 2, 3] (Outlook, Temperature, Humidity, Wind)
# # id3_tree = build_id3_tree(play_tennis_data, attributes, target_index)
# # print(id3_tree.attribute) # Should be the index of 'Outlook'
```

Input

```
Training Data (Same as Lab 10, Play Tennis Example):
Outlook, Temperature, Humidity, Wind, PlayTennis
Sunny, Hot, High, Weak, No
Sunny, Hot, High, Strong, No
Overcast, Hot, High, Weak, Yes
Rain, Mild, High, Weak, Yes
Rain, Cool, Normal, Weak, Yes
Rain, Cool, Normal, Strong, No
Overcast, Cool, Normal, Strong, Yes
Sunny, Mild, High, Weak, No
Sunny, Cool, Normal, Weak, Yes
Rain, Mild, Normal, Weak, Yes
Sunny, Mild, Normal, Strong, Yes
Overcast, Mild, High, Strong, Yes
Overcast, Hot, Normal, Weak, Yes
Rain, Mild, High, Strong, No
```

Expected Output

```
A representation of the ID3 decision tree structure. For the Play Tennis
example, the root node would be 'Outlook'.
```

Lab : 13 Neural Network model implementation

Title

Neural Network Model Implementation

Aim

To implement a basic single-layer or multi-layer perceptron (MLP) neural network from scratch for a simple classification or regression task, focusing on the forward pass and basic architecture.

Procedure

1. **Understand Neural Network Basics:** Grasp concepts like neurons, layers (input, hidden, output), weights, biases, and activation functions (e.g., sigmoid, ReLU).
2. **Define Network Architecture:** Decide on the number of input, hidden (if any), and output neurons.
3. **Initialize Weights and Biases:** Randomly initialize the network's weights and biases.
4. **Implement Activation Function:** Write functions for common activation functions.
5. **Implement Forward Pass:** Develop the logic for calculating the output of the network given an input (input -> hidden -> output).
6. **Prepare Data:** Create a simple dataset for a binary classification or regression problem.
7. **Test Forward Pass:** Test the network with sample inputs and observe the raw outputs before training.

Source Code

```
# Example: Simple Single-Layer Perceptron (Forward Pass Only)

import numpy as np

# Activation function (Sigmoid)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Simple Perceptron Class
class SimplePerceptron:
    def __init__(self, input_size, output_size):
        # Initialize weights and biases randomly
        self.weights = np.random.rand(input_size, output_size)
        self.bias = np.random.rand(output_size)

    def forward(self, inputs):
        # inputs: (num_samples, input_size)
        # Calculate weighted sum
        weighted_sum = np.dot(inputs, self.weights) + self.bias
        # Apply activation function
        output = sigmoid(weighted_sum)
        return output

# Example usage:
# input_data = np.array([[0.5, 0.1], [0.9, 0.8]]) # 2 samples, 2 features
# perceptron = SimplePerceptron(input_size=2, output_size=1)
# predictions = perceptron.forward(input_data)
# print("Predictions:\n", predictions)
```

Input

```
Input data for a 2-input, 1-output perceptron:
Sample 1: [0.5, 0.1]
Sample 2: [0.9, 0.8]
```

Expected Output

```
Predictions:
(Example values, will vary due to random initialization)
[[0.623...]
 [0.710...]]
```

Lab : 14 Implementation of Multi-layer neural network

Title

Implementation of Multi-layer Neural Network

Aim

To implement a complete Multi-Layer Perceptron (MLP) neural network, including the forward pass and the backpropagation algorithm for training.

Procedure

1. **Review MLP Architecture:** Understand input, hidden, and output layers, and the role of activation functions.
2. **Implement Forward Pass:** (Refer to Lab 13)
3. **Implement Loss Function:** Choose and implement a loss function (e.g., Mean Squared Error for regression, Binary Cross-Entropy for binary classification).
4. **Implement Backpropagation:** Develop the core backpropagation algorithm:
    - Calculate the error at the output layer.
    - Propagate the error backward through the network, calculating gradients for weights and biases in each layer.
    - Update weights and biases using an optimization algorithm (e.g., Gradient Descent).
5. **Prepare Data:** Create a suitable dataset for training (e.g., XOR problem, simple linear separation).
6. **Train the Network:** Iterate through epochs, performing forward and backward passes to train the network.
7. **Test and Evaluate:** Evaluate the trained network's performance on unseen data.

Source Code

```
# Example: Simple Multi-Layer Perceptron with Backpropagation

import numpy as np

# Activation functions and their derivatives
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

def relu(x):
    return np.maximum(0, x)

def relu_derivative(x):
    return (x > 0).astype(float)

# Mean Squared Error Loss
def mse_loss(y_true, y_pred):
    return np.mean(np.square(y_true - y_pred))

class MLP:
    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.1):
        self.learning_rate = learning_rate
```

```
        # Weights and biases for hidden layer
        self.weights_input_hidden = np.random.rand(input_size, hidden_size) *
0.01
        self.bias_hidden = np.zeros((1, hidden_size))

        # Weights and biases for output layer
        self.weights_hidden_output = np.random.rand(hidden_size, output_size) *
0.01
        self.bias_output = np.zeros((1, output_size))

    def forward(self, inputs):
        # Hidden layer
        self.hidden_layer_input = np.dot(inputs, self.weights_input_hidden) +
self.bias_hidden
        self.hidden_layer_output = sigmoid(self.hidden_layer_input)

        # Output layer
        self.output_layer_input = np.dot(self.hidden_layer_output,
self.weights_hidden_output) + self.bias_output
        self.output = sigmoid(self.output_layer_input)
        return self.output

    def backward(self, inputs, targets, output):
        # Calculate output layer error and delta
        output_error = targets - output
        output_delta = output_error * sigmoid_derivative(output)

        # Calculate hidden layer error and delta
        hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
        hidden_delta = hidden_error *
sigmoid_derivative(self.hidden_layer_output)

        # Update weights and biases
        self.weights_hidden_output += np.dot(self.hidden_layer_output.T,
output_delta) * self.learning_rate
        self.bias_output += np.sum(output_delta, axis=0, keepdims=True) *
self.learning_rate

        self.weights_input_hidden += np.dot(inputs.T, hidden_delta) *
self.learning_rate
        self.bias_hidden += np.sum(hidden_delta, axis=0, keepdims=True) *
self.learning_rate

    def train(self, X, y, epochs):
        for epoch in range(epochs):
            output = self.forward(X)
            self.backward(X, y, output)
            loss = mse_loss(y, output)
            if (epoch + 1) % 1000 == 0:
                print(f"Epoch {epoch+1}, Loss: {loss:.4f}")

# Example: XOR Problem
# X_xor = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
# y_xor = np.array([[0], [1], [1], [0]])
#
# mlp = MLP(input_size=2, hidden_size=4, output_size=1, learning_rate=0.1)
# mlp.train(X_xor, y_xor, epochs=10000)
#
# print("\nFinal Predictions for XOR:")
# print(mlp.forward(X_xor))
```

Input

```
Training data for XOR problem:
Inputs (X):
```

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

Targets (y):
[[0],
 [1],
 [1],
 [0]]

Epochs: 10000
Learning Rate: 0.1
```

Expected Output

```
Epoch 1000, Loss: 0.24xx
...
Epoch 10000, Loss: 0.00xx

Final Predictions for XOR:
(Values close to 0 or 1)
[[0.0xx]
 [0.9xx]
 [0.9xx]
 [0.0xx]]
```

Lab : 15 Applying Backpropagation and genetic algorithm

Title

Applying Backpropagation and Genetic Algorithm

Aim

To implement and apply the Backpropagation algorithm for training a neural network (as in Lab 14) and to implement a basic Genetic Algorithm for an optimization problem.

Procedure

For Backpropagation (Neural Network Training):

(Refer to Lab 14 procedure)

For Genetic Algorithm:

1. **Understand Genetic Algorithm:** Grasp concepts like population, individuals (chromosomes), genes, fitness function, selection, crossover, and mutation.
2. **Define Problem:** Choose a simple optimization problem (e.g., finding the maximum of a function, solving a simple knapsack problem, or a simple string matching).
3. **Represent Individuals:** Define how solutions (individuals) are encoded (e.g., binary strings, arrays of numbers).
4. **Implement Fitness Function:** Create a function that evaluates the "goodness" of each individual.
5. **Implement Genetic Operators:**
    - **Selection:** Choose parents for reproduction (e.g., roulette wheel, tournament selection).
    - **Crossover:** Combine genetic material from parents to create offspring.
    - **Mutation:** Introduce random changes to offspring.
6. **Implement Main Loop:**
    - Initialize population.
    - Iterate through generations:
        - Evaluate fitness of each individual.
        - Select parents.
        - Generate offspring via crossover and mutation.
        - Form new population.
7. **Test and Analyze:** Run the algorithm and observe how the population evolves towards an optimal solution.

Source Code

```
# Example: Basic Genetic Algorithm for maximizing a simple function f(x) = -x^2
+ 5x + 10
# where x is an integer between 0 and 31 (represented by a 5-bit binary string)

import random
import numpy as np

# 1. Encoding (Binary representation for x)
def decode_chromosome(chromosome):
```

```
    # Convert binary string to integer
    return int("".join(map(str, chromosome)), 2)

# 2. Fitness Function
def fitness_function(x):
    return -x**2 + 5*x + 10 # Max at x=2 or x=3 (depending on precision)

# 3. Genetic Operators

# Selection (Roulette Wheel Selection)
def select_parents(population, fitnesses):
    total_fitness = sum(fitnesses)
    if total_fitness == 0: # Handle case where all fitnesses are zero
        return random.sample(population, 2)

    probabilities = [f / total_fitness for f in fitnesses]
    parents_indices = np.random.choice(len(population), size=2, p=probabilities)
    return [population[i] for i in parents_indices]

# Crossover (Single-point crossover)
def crossover(parent1, parent2):
    crossover_point = random.randint(1, len(parent1) - 1)
    child1 = parent1[:crossover_point] + parent2[crossover_point:]
    child2 = parent2[:crossover_point] + parent1[crossover_point:]
    return child1, child2

# Mutation (Bit-flip mutation)
def mutate(chromosome, mutation_rate):
    mutated_chromosome = list(chromosome)
    for i in range(len(mutated_chromosome)):
        if random.random() < mutation_rate:
            mutated_chromosome[i] = 1 - mutated_chromosome[i] # Flip the bit
    return mutated_chromosome

# Main Genetic Algorithm Loop
def genetic_algorithm(population_size, chromosome_length, generations,
mutation_rate):
    # Initialize population (random binary strings)
    population = [[random.randint(0, 1) for _ in range(chromosome_length)] for _
in range(population_size)]

    for generation in range(generations):
        # Evaluate fitness
        fitnesses = []
        for individual in population:
            x_val = decode_chromosome(individual)
            fitnesses.append(fitness_function(x_val))

        # Keep track of the best individual in current generation
        best_fitness_idx = np.argmax(fitnesses)
        best_individual_chromosome = population[best_fitness_idx]
        best_x = decode_chromosome(best_individual_chromosome)
        print(f"Generation {generation+1}: Best X = {best_x}, Fitness =
{fitness_function(best_x):.2f}")

        new_population = []
        for _ in range(population_size // 2): # Create pairs of children
            parent1, parent2 = select_parents(population, fitnesses)
            child1, child2 = crossover(parent1, parent2)
            new_population.append(mutate(child1, mutation_rate))
            new_population.append(mutate(child2, mutation_rate))

        population = new_population[:population_size] # Ensure population size
is maintained

    # Final best individual
    final_fitnesses = []
```

```
    for individual in population:
        x_val = decode_chromosome(individual)
        final_fitnesses.append(fitness_function(x_val))

    best_idx = np.argmax(final_fitnesses)
    final_best_chromosome = population[best_idx]
    final_best_x = decode_chromosome(final_best_chromosome)
    final_best_fitness = fitness_function(final_best_x)

    return final_best_x, final_best_fitness

# Example usage:
# pop_size = 10
# chrom_len = 5 # For x from 0 to 31
# num_generations = 50
# mut_rate = 0.05
#
# best_x_found, max_fitness_found = genetic_algorithm(pop_size, chrom_len,
num_generations, mut_rate)
# print(f"\nGenetic Algorithm Result: Best X = {best_x_found}, Max Fitness =
{max_fitness_found:.2f}")
```

Input

For Backpropagation:

(Same as Lab 14, XOR problem)

For Genetic Algorithm:

```
Problem: Maximize f(x) = -x^2 + 5x + 10, where x is an integer between 0 and 31.
Population Size: 10
Chromosome Length: 5 (for 5-bit binary representation of x)
Number of Generations: 50
Mutation Rate: 0.05
```

Expected Output

For Backpropagation:

(Same as Lab 14)

For Genetic Algorithm:

```
Generation 1: Best X = ..., Fitness = ...
...
Generation 50: Best X = 2 (or 3), Fitness = 16.00 (or 16.00)

Genetic Algorithm Result: Best X = 2 (or 3), Max Fitness = 16.00
```