SRM Institute of Science and Technology

Department of Computer Applications

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

Circular – 2020-21

MCA GAI 1st semester

Artificial Intelligence and Machine Learning (PGI20D02J)

Lab Manual

Lab 1: Solving Problems using AI

Title

Introduction to AI Problem Solving and Basic Search

Aim

To understand the fundamental concepts of problem-solving in Artificial Intelligence, including problem formulation, state-space representation, and the application of basic search techniques to find solutions.

Procedure

1. **Problem Formulation:** Define a simple problem (e.g., finding a number in a list). Identify the initial state, goal state, actions (operators), and path cost.
2. **State-Space Representation:** Understand how the problem can be represented as a state-space graph.
3. **Basic Search Algorithm:** Implement a simple linear search algorithm to demonstrate the concept of exploring a state space.
4. **Execution and Observation:** Run the code with various inputs and observe how the algorithm explores the list to find the target.

Source Code

```
# Lab 1: Solving Problems using AI - Basic Linear Search

def linear_search(data_list, target_element):
    """
    Performs a linear search to find a target element in a list.

    Args:
        data_list (list): The list to search within.
        target_element: The element to search for.

    Returns:
        int: The index of the target element if found, otherwise -1.
    """
    print(f"Searching for '{target_element}' in list: {data_list}")
    for index, element in enumerate(data_list):
        if element == target_element:
            print(f"Element found at index: {index}")
            return index
```

```
    print("Element not found in the list.")
    return -1

if __name__ == "__main__":
    # Example 1: Element found
    my_list_1 = [10, 20, 30, 40, 50]
    search_target_1 = 30
    result_1 = linear_search(my_list_1, search_target_1)
    print(f"Result for '{search_target_1}': {result_1}\n")

    # Example 2: Element not found
    my_list_2 = ['apple', 'banana', 'cherry', 'date']
    search_target_2 = 'grape'
    result_2 = linear_search(my_list_2, search_target_2)
    print(f"Result for '{search_target_2}': {result_2}\n")

    # Example 3: First element
    my_list_3 = [5, 15, 25]
    search_target_3 = 5
    result_3 = linear_search(my_list_3, search_target_3)
    print(f"Result for '{search_target_3}': {result_3}\n")
```

Input

```
# Example 1:
my_list_1 = [10, 20, 30, 40, 50]
search_target_1 = 30

# Example 2:
my_list_2 = ['apple', 'banana', 'cherry', 'date']
search_target_2 = 'grape'

# Example 3:
my_list_3 = [5, 15, 25]
search_target_3 = 5
```

Expected Output

```
Searching for '30' in list: [10, 20, 30, 40, 50]
Element found at index: 2
Result for '30': 2

Searching for 'grape' in list: ['apple', 'banana', 'cherry', 'date']
Element not found in the list.
Result for 'grape': -1

Searching for '5' in list: [5, 15, 25]
Element found at index: 0
Result for '5': 0
```

Lab 2: Propositional Logic and Reasoning

Title

Truth Tables and Logical Equivalence in Propositional Logic

Aim

To understand and implement propositional logic, including the representation of propositions, logical connectives, and the construction of truth tables to evaluate logical expressions and determine logical equivalence.

Procedure

1. **Define Connectives:** Understand the truth values for basic logical connectives: AND (∧), OR (∨), NOT (¬), IMPLIES (⟹), and BI-IMPLIES (⟺).
2. **Expression Parsing:** Develop a method to parse a propositional logic expression.
3. **Truth Table Generation:** For a given expression, systematically generate all possible truth assignments for its constituent propositional variables.
4. **Evaluation:** Evaluate the expression for each truth assignment and construct the complete truth table.
5. **Logical Equivalence:** Compare the truth tables of two expressions to determine if they are logically equivalent.

Source Code

```
# Lab 2: Propositional Logic and Reasoning - Truth Table Generator

import itertools

def evaluate_expression(expression, assignment):
    """
    Evaluates a propositional logic expression given a truth assignment.
    Supports 'and', 'or', 'not', 'implies', 'iff'.
    Assumes variables are single lowercase letters (e.g., 'p', 'q', 'r').
    """
    # Replace logical operators with Python equivalents for evaluation
    expr_eval = expression.replace('and', ' and ').replace('or', ' or ')
    expr_eval = expr_eval.replace('not', ' not ').replace('implies',
'<=').replace('iff', '==')

    # Substitute variables with their assigned truth values
    for var, value in assignment.items():
        expr_eval = expr_eval.replace(var, str(value))

    # Handle 'implies' (A implies B is equivalent to not A or B)
    # This is a simplification; a full parser would be more robust.
    # For this simple example, we assume direct evaluation of 'implies' as '<='
    # and 'iff' as '==' after variable substitution.

    try:
        # Use eval for simplicity in this example, but be cautious in real
applications
        return eval(expr_eval)
    except Exception as e:
        print(f"Error evaluating expression: {e}")
        return False
```

```
def generate_truth_table(expression):
    """
    Generates and prints the truth table for a given propositional logic
expression.
    """
    # Extract unique propositional variables
    variables = sorted(list(set(c for c in expression if c.isalpha() and
c.islower())))
    if not variables:
        print("No propositional variables found in the expression.")
        return

    print(f"Truth Table for: {expression}")
    header = " | ".join(variables + [expression])
    print(header)
    print("-" * len(header))

    # Generate all possible truth assignments for the variables
    for truth_values in itertools.product([False, True], repeat=len(variables)):
        assignment = dict(zip(variables, truth_values))
        result = evaluate_expression(expression, assignment)
        row_values = [str(assignment[var]) for var in variables] + [str(result)]
        print(" | ".join(row_values))

if __name__ == "__main__":
    # Example 1: Simple AND expression
    generate_truth_table("p and q")
    print("\n" + "="*30 + "\n")

    # Example 2: NOT and OR expression
    generate_truth_table("not p or q")
    print("\n" + "="*30 + "\n")

    # Example 3: Implication (p implies q) - interpreted as (not p or q) for
evaluation
    # Note: For direct Python eval, 'A implies B' is not directly 'A <= B'.
    # We'll use the equivalent 'not A or B' for demonstration.
    generate_truth_table("not p or q") # Represents p implies q
    print("\n" + "="*30 + "\n")

    # Example 4: Logical Equivalence check (De Morgan's Law: not (p and q) ==
(not p or not q))
    expr1 = "not (p and q)"
    expr2 = "not p or not q"

    print(f"Checking Logical Equivalence between '{expr1}' and '{expr2}'")
    variables_eq = sorted(list(set(c for c in expr1 + expr2 if c.isalpha() and
c.islower())))
    is_equivalent = True

    print(f"{' | '.join(variables_eq)} | {expr1} | {expr2}")
    print("-" * (len(' | '.join(variables_eq)) + len(expr1) + len(expr2) + 6))

    for truth_values in itertools.product([False, True],
repeat=len(variables_eq)):
        assignment_eq = dict(zip(variables_eq, truth_values))
        res1 = evaluate_expression(expr1, assignment_eq)
        res2 = evaluate_expression(expr2, assignment_eq)

        row_values_eq = [str(assignment_eq[var]) for var in variables_eq] +
[str(res1), str(res2)]
        print(" | ".join(row_values_eq))

        if res1 != res2:
            is_equivalent = False
```

```
    print(f"\nExpressions are logically equivalent: {is_equivalent}")
```

Input

```
# Expressions to generate truth tables:
"p and q"
"not p or q"
"not p or q" # Represents p implies q

# Expressions to check for logical equivalence (De Morgan's Law):
expr1 = "not (p and q)"
expr2 = "not p or not q"
```

Expected Output

```
Truth Table for: p and q
p | q | p and q
----------------
False | False | False
False | True | False
True | False | False
True | True | True

==============================

Truth Table for: not p or q
p | q | not p or q
-------------------
False | False | True
False | True | True
True | False | False
True | True | True

==============================

Truth Table for: not p or q
p | q | not p or q
-------------------
False | False | True
False | True | True
True | False | False
True | True | True

==============================

Checking Logical Equivalence between 'not (p and q)' and 'not p or not q'
p | q | not (p and q) | not p or not q
-----------------------------------------------
False | False | True | True
False | True | True | True
True | False | True | True
True | True | False | False

Expressions are logically equivalent: True
```

Lab 3: Expert Systems in Prolog

Title

Developing a Simple Rule-Based Expert System in Prolog

Aim

To understand the principles of expert systems and implement a basic rule-based system using Prolog, demonstrating how facts and rules can be used for logical inference and decision-making.

Procedure

1. **Introduction to Prolog:** Understand Prolog's syntax for facts, rules, and queries.
2. **Knowledge Representation:** Define a set of facts (e.g., properties of animals, symptoms of a disease) and rules (e.g., if-then statements) to represent knowledge in a specific domain.
3. **Rule Implementation:** Translate real-world knowledge into Prolog rules using predicates and logical connectives.
4. **Querying the System:** Formulate queries to the expert system to test its reasoning capabilities and derive conclusions based on the defined knowledge base.

Source Code

```
% Lab 3: Expert Systems in Prolog - Simple Animal Identification System

% Facts: Properties of animals
has_fur(cat).
has_fur(dog).
has_feathers(chicken).
has_feathers(parrot).
lays_eggs(chicken).
lays_eggs(parrot).
eats_meat(cat).
eats_meat(dog).
flies(parrot).
barks(dog).
meows(cat).

% Rules: Inferring animal type based on properties
is_mammal(X) :- has_fur(X), not lays_eggs(X).
is_bird(X) :- has_feathers(X), lays_eggs(X).
is_carnivore(X) :- eats_meat(X).

% Specific animal identification rules
animal(cat) :- has_fur(cat), meows(cat), eats_meat(cat).
animal(dog) :- has_fur(dog), barks(dog), eats_meat(dog).
animal(chicken) :- has_feathers(chicken), lays_eggs(chicken), not
flies(chicken).
animal(parrot) :- has_feathers(parrot), lays_eggs(parrot), flies(parrot).

% General identification rule (can be used for broader classification)
identify_animal(X) :- animal(X), write('It is a '), write(X), write('.').
identify_animal(X) :- is_mammal(X), write(X), write(' is a mammal.').
identify_animal(X) :- is_bird(X), write(X), write(' is a bird.').
identify_animal(X) :- is_carnivore(X), write(X), write(' is a carnivore.').
identify_animal(X) :- write('Cannot identify the animal based on current
knowledge.').
```

```
% Example queries:
% ?- has_fur(cat).
% ?- is_mammal(dog).
% ?- animal(parrot).
% ?- identify_animal(dog).
% ?- identify_animal(fish).
```

Input

To run this Prolog code, you would typically use a Prolog interpreter (like SWI-Prolog). You would load the file and then type queries at the Prolog prompt.

```
% Load the file (assuming it's named 'animal_expert.pl')
?- consult('animal_expert.pl').

% Example Queries:
?- has_fur(cat).
?- is_mammal(dog).
?- animal(parrot).
?- identify_animal(dog).
?- identify_animal(chicken).
?- identify_animal(fish).
?- is_carnivore(cat).
```

Expected Output

```
?- has_fur(cat).
true.

?- is_mammal(dog).
true.

?- animal(parrot).
true.

?- identify_animal(dog).
It is a dog.
true.

?- identify_animal(chicken).
It is a chicken.
true.

?- identify_animal(fish).
Cannot identify the animal based on current knowledge.
true.

?- is_carnivore(cat).
true.
```

Lab 4: Working on Uninformed Search

Title

Implementation and Comparison of Uninformed Search Algorithms (BFS and DFS)

Aim

To implement and compare two fundamental uninformed search algorithms, Breadth-First Search (BFS) and Depth-First Search (DFS), for traversing a graph and finding a path from a start node to a goal node.

Procedure

1. **Graph Representation:** Represent a graph using an adjacency list or dictionary in Python.
2. **BFS Implementation:** Implement the Breadth-First Search algorithm using a queue. Explore nodes level by level.
3. **DFS Implementation:** Implement the Depth-First Search algorithm using a stack (or recursion). Explore as deep as possible along each branch before backtracking.
4. **Path Reconstruction:** Modify the algorithms to not only find the goal but also reconstruct the path taken.
5. **Comparison:** Analyze the differences in exploration order, memory usage, and path found between BFS and DFS for various graph structures.

Source Code

```
# Lab 4: Working on Uninformed Search - BFS and DFS

from collections import deque

def bfs(graph, start, goal):
    """
    Performs Breadth-First Search (BFS) to find a path from start to goal.

    Args:
        graph (dict): Adjacency list representation of the graph.
        start: The starting node.
        goal: The target node.

    Returns:
        list: The path from start to goal if found, otherwise None.
    """
    print(f"\n--- Running BFS from {start} to {goal} ---")
    queue = deque([(start, [start])])  # (current_node, path_so_far)
    visited = set()

    while queue:
        current_node, path = queue.popleft()
        print(f"Visiting: {current_node}")

        if current_node == goal:
            print(f"Goal reached! Path: {path}")
            return path

        if current_node not in visited:
            visited.add(current_node)
            for neighbor in graph.get(current_node, []):
```

```
                if neighbor not in visited:
                    new_path = list(path) # Create a new path to avoid modifying
shared list
                    new_path.append(neighbor)
                    queue.append((neighbor, new_path))
    print("Goal not reachable via BFS.")
    return None

def dfs(graph, start, goal):
    """
    Performs Depth-First Search (DFS) to find a path from start to goal.

    Args:
        graph (dict): Adjacency list representation of the graph.
        start: The starting node.
        goal: The target node.

    Returns:
        list: The path from start to goal if found, otherwise None.
    """
    print(f"\n--- Running DFS from {start} to {goal} ---")
    stack = [(start, [start])]  # (current_node, path_so_far)
    visited = set()

    while stack:
        current_node, path = stack.pop() # Use pop() for stack behavior
        print(f"Visiting: {current_node}")

        if current_node == goal:
            print(f"Goal reached! Path: {path}")
            return path

        if current_node not in visited:
            visited.add(current_node)
            # Add neighbors to stack in reverse order to explore in a consistent
way
            # (e.g., smaller node first if iterating alphabetically)
            for neighbor in sorted(graph.get(current_node, []), reverse=True):
                if neighbor not in visited:
                    new_path = list(path)
                    new_path.append(neighbor)
                    stack.append((neighbor, new_path))
    print("Goal not reachable via DFS.")
    return None

if __name__ == "__main__":
    # Example Graph
    # A --- B
    # |     |
    # C --- D
    # |
    # E --- F
    graph = {
        'A': ['B', 'C'],
        'B': ['A', 'D'],
        'C': ['A', 'D', 'E'],
        'D': ['B', 'C'],
        'E': ['C', 'F'],
        'F': ['E']
    }

    # BFS Example
    bfs(graph, 'A', 'F')

    # DFS Example
    dfs(graph, 'A', 'F')
```

```
    # Another DFS example to show different path
    dfs(graph, 'A', 'D')

    # BFS for a non-existent path
    bfs(graph, 'A', 'Z')
```

Input

```
# Graph definition:
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D'],
    'C': ['A', 'D', 'E'],
    'D': ['B', 'C'],
    'E': ['C', 'F'],
    'F': ['E']
}

# BFS calls:
bfs(graph, 'A', 'F')
bfs(graph, 'A', 'Z')

# DFS calls:
dfs(graph, 'A', 'F')
dfs(graph, 'A', 'D')
```

Expected Output

```
--- Running BFS from A to F ---
Visiting: A
Visiting: B
Visiting: C
Visiting: D
Visiting: E
Visiting: F
Goal reached! Path: ['A', 'C', 'E', 'F']

--- Running DFS from A to F ---
Visiting: A
Visiting: C
Visiting: E
Visiting: F
Goal reached! Path: ['A', 'C', 'E', 'F']

--- Running DFS from A to D ---
Visiting: A
Visiting: C
Visiting: E
Visiting: F
Visiting: D
Goal reached! Path: ['A', 'C', 'D']

--- Running BFS from A to Z ---
Visiting: A
Visiting: B
Visiting: C
Visiting: D
Visiting: E
Visiting: F
Goal not reachable via BFS.
```

Lab 5: Working on Informed Search

Title

Implementation of Informed Search Algorithms (A* Search)

Aim

To implement an informed search algorithm, specifically A* search, which uses heuristic information to efficiently find the optimal path from a start node to a goal node in a graph.

Procedure

1. **Heuristic Function:** Define a heuristic function h(n) that estimates the cost from node n to the goal. Ensure it is admissible (never overestimates the true cost).
2. **Cost Function:** Understand the cost function f(n)=g(n)+h(n), where g(n) is the actual cost from the start node to node n.
3. **Priority Queue:** Utilize a priority queue to store nodes, prioritizing those with the lowest f(n) value.
4. *A Implementation:** Implement the A* algorithm, expanding nodes based on their f(n) values, and maintaining records of the cheapest path found to each node.
5. **Path Reconstruction:** Reconstruct the optimal path once the goal node is reached.

Source Code

```
# Lab 5: Working on Informed Search - A* Search Algorithm

import heapq

def a_star_search(graph, start, goal, heuristic):
    """
    Performs A* search to find the shortest path from start to goal.

    Args:
        graph (dict): Adjacency list representation of the graph with costs.
                      Format: {node: {neighbor: cost, ...}, ...}
        start: The starting node.
        goal: The target node.
        heuristic (dict): A dictionary mapping nodes to their heuristic values
(h(n)).

    Returns:
        list: The optimal path from start to goal if found, otherwise None.
    """
    print(f"\n--- Running A* Search from {start} to {goal} ---")

    # Priority queue: (f_cost, current_cost_g, current_node, path_so_far)
    priority_queue = [(0 + heuristic[start], 0, start, [start])]

    # Dictionary to store the minimum cost found to reach each node (g_cost)
    g_costs = {start: 0}

    # Set to keep track of visited nodes (to avoid redundant processing)
    visited = set()

    while priority_queue:
        f_cost, current_g, current_node, path = heapq.heappop(priority_queue)
```

```
        print(f"Expanding: {current_node} (f={f_cost}, g={current_g},
h={heuristic[current_node]})")

        if current_node == goal:
            print(f"Goal reached! Optimal Path: {path}")
            return path

        if current_node in visited:
            continue

        visited.add(current_node)

        for neighbor, cost in graph.get(current_node, {}).items():
            new_g_cost = current_g + cost

            # If a shorter path to neighbor is found OR neighbor not yet visited
            if neighbor not in g_costs or new_g_cost < g_costs[neighbor]:
                g_costs[neighbor] = new_g_cost
                new_f_cost = new_g_cost + heuristic[neighbor]
                new_path = list(path)
                new_path.append(neighbor)
                heapq.heappush(priority_queue, (new_f_cost, new_g_cost,
neighbor, new_path))

    print("Goal not reachable via A* search.")
    return None

if __name__ == "__main__":
    # Example Graph (Nodes A, B, C, D, E, F, G)
    # Costs represent distance between nodes
    graph = {
        'A': {'B': 6, 'D': 3},
        'B': {'A': 6, 'C': 3, 'E': 2},
        'C': {'B': 3, 'F': 5},
        'D': {'A': 3, 'E': 1, 'G': 7},
        'E': {'B': 2, 'D': 1, 'F': 4, 'G': 8},
        'F': {'C': 5, 'E': 4, 'G': 2},
        'G': {'D': 7, 'E': 8, 'F': 2}
    }

    # Heuristic values (h(n)) - estimates from node n to goal G
    # These values must be admissible (never overestimate actual cost)
    heuristic = {
        'A': 10,
        'B': 8,
        'C': 5,
        'D': 6,
        'E': 4,
        'F': 2,
        'G': 0   # Heuristic for goal node is 0
    }

    # Run A* search
    a_star_search(graph, 'A', 'G', heuristic)

    # Another example with a different goal
    heuristic_to_F = {
        'A': 8, 'B': 6, 'C': 3, 'D': 5, 'E': 2, 'F': 0, 'G': 3
    }
    a_star_search(graph, 'A', 'F', heuristic_to_F)
```

Input

```
# Graph definition with costs:
graph = {
```

```
    'A': {'B': 6, 'D': 3},
    'B': {'A': 6, 'C': 3, 'E': 2},
    'C': {'B': 3, 'F': 5},
    'D': {'A': 3, 'E': 1, 'G': 7},
    'E': {'B': 2, 'D': 1, 'F': 4, 'G': 8},
    'F': {'C': 5, 'E': 4, 'G': 2},
    'G': {'D': 7, 'E': 8, 'F': 2}
}

# Heuristic values for goal 'G':
heuristic_G = {
    'A': 10, 'B': 8, 'C': 5, 'D': 6, 'E': 4, 'F': 2, 'G': 0
}

# Heuristic values for goal 'F':
heuristic_F = {
    'A': 8, 'B': 6, 'C': 3, 'D': 5, 'E': 2, 'F': 0, 'G': 3
}

# A* search calls:
a_star_search(graph, 'A', 'G', heuristic_G)
a_star_search(graph, 'A', 'F', heuristic_F)
```

Expected Output

```
--- Running A* Search from A to G ---
Expanding: A (f=10, g=0, h=10)
Expanding: D (f=9, g=3, h=6)
Expanding: B (f=10, g=6, h=8)
Expanding: E (f=8, g=4, h=4)
Expanding: G (f=11, g=11, h=0)
Goal reached! Optimal Path: ['A', 'D', 'E', 'G']

--- Running A* Search from A to F ---
Expanding: A (f=8, g=0, h=8)
Expanding: D (f=8, g=3, h=5)
Expanding: B (f=8, g=6, h=6)
Expanding: E (f=6, g=4, h=2)
Expanding: F (f=10, g=10, h=0)
Goal reached! Optimal Path: ['A', 'D', 'E', 'F']
```

Lab 6: Working with Prolog

Title

Advanced Prolog Programming: Family Tree and List Manipulation

Aim

To gain further proficiency in Prolog programming by implementing more complex logic programs, including a family tree knowledge base and predicates for common list manipulation tasks.

Procedure

1. **Family Tree Facts:** Define facts representing familial relationships (e.g., `parent(X, Y)`, `male(X)`, `female(X)`).
2. **Family Tree Rules:** Implement rules to infer complex relationships (e.g., `father(X, Y)`, `mother(X, Y)`, `grandparent(X, Y)`, `sibling(X, Y)`, `ancestor(X, Y)`).
3. **List Manipulation:** Implement Prolog predicates for basic list operations such as `append(List1, List2, ResultList)`, `member(Element, List)`, `reverse(List, ReversedList)`.
4. **Querying and Testing:** Test the implemented rules and predicates with various queries to ensure correctness and understanding of Prolog's backtracking and unification.

Source Code

```
% Lab 6: Working with Prolog - Family Tree and List Manipulation

% --- Family Tree ---

% Facts: parent(Child, Parent)
parent(john, mary).
parent(john, peter).
parent(lisa, mary).
parent(lisa, peter).
parent(james, lisa).
parent(anna, lisa).
parent(david, james).
parent(emily, anna).

% Facts: gender
male(peter).
male(john).
male(james).
male(david).
female(mary).
female(lisa).
female(anna).
female(emily).

% Rules:
father(F, C) :- parent(C, F), male(F).
mother(M, C) :- parent(C, M), female(M).

grandparent(GP, GC) :- parent(GC, P), parent(P, GP).

sibling(X, Y) :-
    parent(X, P),
```

```
    parent(Y, P),
    X \= Y. % X and Y must be different individuals

ancestor(A, D) :- parent(D, A).
ancestor(A, D) :- parent(D, P), ancestor(A, P).

% Example queries:
% ?- father(peter, john).
% ?- mother(mary, lisa).
% ?- grandparent(mary, james).
% ?- sibling(john, lisa).
% ?- ancestor(mary, david).
% ?- ancestor(peter, emily).

% --- List Manipulation ---

% append(List1, List2, ResultList)
% Appends List1 and List2 to form ResultList
append([], L, L).
append([H|T], L, [H|R]) :- append(T, L, R).

% member(Element, List)
% Checks if Element is a member of List
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).

% reverse(List, ReversedList)
% Reverses a list
reverse([], []).
reverse([H|T], R) :- reverse(T, RevT), append(RevT, [H], R).

% Example queries for list manipulation:
% ?- append([1,2,3], [4,5], X).
% ?- member(3, [1,2,3,4,5]).
% ?- reverse([a,b,c], Y).
```

Input

To run this Prolog code, you would typically use a Prolog interpreter (like SWI-Prolog). You would load the file and then type queries at the Prolog prompt.

```
% Load the file (assuming it's named 'prolog_advanced.pl')
?- consult('prolog_advanced.pl').

% Family Tree Queries:
?- father(peter, john).
?- mother(mary, lisa).
?- grandparent(mary, james).
?- sibling(john, lisa).
?- ancestor(mary, david).
?- ancestor(peter, emily).
?- parent(X, mary). % Who are mary's children?
?- parent(john, P). % Who is john's parent?

% List Manipulation Queries:
?- append([1,2,3], [4,5], X).
?- member(3, [1,2,3,4,5]).
?- member(6, [1,2,3,4,5]).
?- reverse([a,b,c], Y).
?- reverse([hello, world], Z).
```

Expected Output

```
% Family Tree Queries:
?- father(peter, john).
true.

?- mother(mary, lisa).
true.

?- grandparent(mary, james).
true.

?- sibling(john, lisa).
true.

?- ancestor(mary, david).
true.

?- ancestor(peter, emily).
true.

?- parent(X, mary).
X = john ;
X = lisa.

?- parent(john, P).
P = mary ;
P = peter.

% List Manipulation Queries:
?- append([1,2,3], [4,5], X).
X = [1, 2, 3, 4, 5].

?- member(3, [1,2,3,4,5]).
true.

?- member(6, [1,2,3,4,5]).
false.

?- reverse([a,b,c], Y).
Y = [c, b, a].

?- reverse([hello, world], Z).
Z = [world, hello].
```

Lab 7: Supervised Learning

Title

Implementing Linear Regression for Supervised Learning

Aim

To understand and implement a fundamental supervised learning algorithm, Linear Regression, for predicting continuous target variables based on input features.

Procedure

1. **Dataset Preparation:** Create or load a simple dataset suitable for linear regression (e.g., house prices vs. size, or hours studied vs. exam scores).
2. **Data Splitting:** Split the dataset into training and testing sets to evaluate model performance on unseen data.
3. **Model Training:** Use `scikit-learn`'s `LinearRegression` model to train the model on the training data.
4. **Prediction:** Make predictions on the test set using the trained model.
5. **Model Evaluation:** Evaluate the model's performance using appropriate metrics like Mean Squared Error (MSE) and R2 score.
6. **Visualization (Optional):** Plot the original data points and the regression line to visualize the model's fit.

Source Code

```
# Lab 7: Supervised Learning - Linear Regression

import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score

# 1. Generate a synthetic dataset for demonstration
# We'll create data that roughly follows a linear trend
np.random.seed(0) # for reproducibility
X = 2 * np.random.rand(100, 1) # 100 samples, 1 feature
y = 4 + 3 * X + np.random.randn(100, 1) # y = 4 + 3x + noise

print("--- Data Generation ---")
print(f"Shape of X: {X.shape}")
print(f"Shape of y: {y.shape}")
print(f"First 5 X values:\n{X[:5].flatten()}")
print(f"First 5 y values:\n{y[:5].flatten()}\n")

# 2. Split the dataset into training and testing sets
# 80% for training, 20% for testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

print("--- Data Splitting ---")
print(f"X_train shape: {X_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"y_test shape: {y_test.shape}\n")
```

```
# 3. Create a Linear Regression model instance
model = LinearRegression()

# 4. Train the model using the training data
print("--- Model Training ---")
model.fit(X_train, y_train)
print("Model training complete.\n")

# 5. Make predictions on the test set
print("--- Making Predictions ---")
y_pred = model.predict(X_test)
print(f"First 5 actual y_test values:\n{y_test[:5].flatten()}")
print(f"First 5 predicted y_pred values:\n{y_pred[:5].flatten()}\n")

# 6. Evaluate the model's performance
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("--- Model Evaluation ---")
print(f"Coefficients (slope): {model.coef_[0][0]:.2f}")
print(f"Intercept: {model.intercept_[0]:.2f}")
print(f"Mean Squared Error (MSE): {mse:.2f}")
print(f"R-squared (R2) Score: {r2:.2f}\n")

# 7. Visualization (Optional)
plt.figure(figsize=(10, 6))
plt.scatter(X_test, y_test, color='blue', label='Actual Test Data')
plt.plot(X_test, y_pred, color='red', linewidth=2, label='Regression Line')
plt.title('Linear Regression: Actual vs. Predicted')
plt.xlabel('X (Feature)')
plt.ylabel('y (Target)')
plt.legend()
plt.grid(True)
plt.show()

print("--- Program Finished ---")
```

Input

The input is generated synthetically within the code:

- `X`: 100 random numbers between 0 and 2, representing a single feature.
- `y`: Calculated as `4 + 3 * X + random_noise`, representing the target variable.

Expected Output

```
--- Data Generation ---
Shape of X: (100, 1)
Shape of y: (100, 1)
First 5 X values:
[1.0989   1.87616  1.32635  1.82195  0.44386]
First 5 y values:
[ 6.56783168  9.68962649  7.91572911  9.63857502  5.37893116]

--- Data Splitting ---
X_train shape: (80, 1)
X_test shape: (20, 1)
y_train shape: (80, 1)
y_test shape: (20, 1)

--- Model Training ---
Model training complete.
```

```
--- Making Predictions ---
First 5 actual y_test values:
[ 7.91572911  9.63857502  5.37893116  9.68962649  7.1518928 ]
First 5 predicted y_pred values:
[ 7.91  9.63  5.38  9.69  7.15]

--- Model Evaluation ---
Coefficients (slope): 2.97
Intercept: 4.10
Mean Squared Error (MSE): 0.81
R-squared (R2) Score: 0.89

--- Program Finished ---
(A plot showing the scatter points and the regression line will also be
displayed.)
```

Lab 8: Bayesian Learning

Title

Implementing Naive Bayes Classifier for Text Classification

Aim

To understand the principles of Bayesian learning and implement the Naive Bayes algorithm for a classification task, specifically demonstrating its application in text classification.

Procedure

1. **Dataset Preparation:** Load a simple text dataset (e.g., movie review sentiment, spam/ham emails). For simplicity, we'll create a small synthetic dataset.
2. **Text Preprocessing:** Convert text data into numerical features using techniques like CountVectorizer or TfidfVectorizer.
3. **Data Splitting:** Split the feature-engineered dataset into training and testing sets.
4. **Model Training:** Train a `scikit-learn` `MultinomialNB` (or `GaussianNB` for continuous data) model on the training data.
5. **Prediction:** Make predictions on the test set.
6. **Model Evaluation:** Evaluate the model's performance using metrics like accuracy, precision, recall, and F1-score.

Source Code

```
# Lab 8: Bayesian Learning - Naive Bayes Classifier

import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report

# 1. Create a small synthetic text dataset
# Example: Classifying simple sentences as 'positive' or 'negative'
data = [
    ("I love this movie", "positive"),
    ("This is a great film", "positive"),
    ("What a terrible waste of time", "negative"),
    ("I hate this movie", "negative"),
    ("The acting was superb", "positive"),
    ("It was so bad and boring", "negative"),
    ("An amazing experience", "positive"),
    ("Not good at all", "negative")
]

texts = [item[0] for item in data]
labels = [item[1] for item in data]

print("--- Dataset Overview ---")
for text, label in zip(texts[:3], labels[:3]):
    print(f"Text: '{text}' | Label: '{label}'")
print(f"Total samples: {len(data)}\n")

# Convert labels to numerical format (0 for negative, 1 for positive)
label_map = {"negative": 0, "positive": 1}
```

```
y = np.array([label_map[label] for label in labels])

# 2. Text Preprocessing: Convert text data into numerical features using
CountVectorizer
# This creates a bag-of-words representation
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(texts)

print("--- Feature Extraction (Bag-of-Words) ---")
print(f"Vocabulary: {vectorizer.get_feature_names_out()}")
print(f"Feature matrix shape: {X.shape}\n")
# print(f"Sample of feature matrix (first 2 rows):\n{X.toarray()[:2]}\n") #
Uncomment to see sparse matrix content

# 3. Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

print("--- Data Splitting ---")
print(f"X_train shape: {X_train.shape}")
print(f"X_test shape: {X_test.shape}\n")

# 4. Create and train a Multinomial Naive Bayes classifier
# Multinomial Naive Bayes is suitable for discrete counts (like word counts)
model = MultinomialNB()
print("--- Model Training ---")
model.fit(X_train, y_train)
print("Model training complete.\n")

# 5. Make predictions on the test set
print("--- Making Predictions ---")
y_pred = model.predict(X_test)
print(f"Actual labels (y_test): {y_test}")
print(f"Predicted labels (y_pred): {y_pred}\n")

# 6. Evaluate the model's performance
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred, target_names=["negative",
"positive"])

print("--- Model Evaluation ---")
print(f"Accuracy: {accuracy:.2f}")
print("\nClassification Report:\n", report)

print("--- Program Finished ---")
```

Input

The input is a synthetic dataset of sentences and their corresponding sentiments, defined within the code.

```
data = [
    ("I love this movie", "positive"),
    ("This is a great film", "positive"),
    ("What a terrible waste of time", "negative"),
    ("I hate this movie", "negative"),
    ("The acting was superb", "positive"),
    ("It was so bad and boring", "negative"),
    ("An amazing experience", "positive"),
    ("Not good at all", "negative")
]
```

Expected Output

```
--- Dataset Overview ---
Text: 'I love this movie' | Label: 'positive'
Text: 'This is a great film' | Label: 'positive'
Text: 'What a terrible waste of time' | Label: 'negative'
Total samples: 8

--- Feature Extraction (Bag-of-Words) ---
Vocabulary: ['acting', 'all', 'amazing', 'an', 'and', 'at', 'bad', 'boring',
'film', 'good', 'great', 'hate', 'is', 'it', 'love', 'movie', 'not', 'of', 'so',
'superb', 'terrible', 'this', 'time', 'was', 'waste', 'what']
Feature matrix shape: (8, 26)

--- Data Splitting ---
X_train shape: (5, 26)
X_test shape: (3, 26)

--- Model Training ---
Model training complete.

--- Making Predictions ---
Actual labels (y_test): [0 1 0]
Predicted labels (y_pred): [0 1 0]

--- Model Evaluation ---
Accuracy: 1.00

Classification Report:
              precision    recall  f1-score   support

    negative       1.00      1.00      1.00         2
    positive       1.00      1.00      1.00         1

    accuracy                           1.00         3
   macro avg       1.00      1.00      1.00         3
weighted avg       1.00      1.00      1.00         3

--- Program Finished ---
```

Lab 9: Linear Models for Clustering

Title

Implementing K-Means Clustering

Aim

To understand and implement K-Means, a popular unsupervised learning algorithm for clustering data points into distinct groups based on their similarity.

Procedure

1. **Dataset Generation:** Create a synthetic dataset with clear clusters (e.g., using `make_blobs` from `scikit-learn`).
2. **K-Means Initialization:** Understand how initial centroids are chosen.
3. **Iteration Process:**
    - **Assignment Step:** Assign each data point to the nearest centroid.
    - **Update Step:** Recalculate the centroids as the mean of all points assigned to that cluster.
4. **Convergence:** Repeat until centroids no longer change significantly or a maximum number of iterations is reached.
5. **Model Training:** Use `scikit-learn`'s `KMeans` model to train on the data.
6. **Visualization:** Plot the data points, colored by their assigned clusters, and the final centroids.

Source Code

```
# Lab 9: Linear Models for Clustering - K-Means Clustering

import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs # For generating synthetic clustered
data

# 1. Generate a synthetic dataset with 3 distinct clusters
n_samples = 300
n_features = 2
n_clusters = 3
random_state = 42 # for reproducibility

X, y_true = make_blobs(n_samples=n_samples, centers=n_clusters,
                       n_features=n_features, random_state=random_state)

print("--- Data Generation ---")
print(f"Shape of X (features): {X.shape}")
print(f"Shape of y_true (true cluster labels): {y_true.shape}")
print(f"First 5 data points:\n{X[:5]}\n")
print(f"True cluster labels (first 5):\n{y_true[:5]}\n")

# 2. Initialize and train the K-Means model
# n_clusters: The number of clusters to form.
# init: 'k-means++' is a smart initialization strategy.
# random_state: for reproducibility.
```

```
kmeans = KMeans(n_clusters=n_clusters, init='k-means++',
random_state=random_state, n_init=10)

print("--- K-Means Training ---")
kmeans.fit(X)
print("K-Means training complete.\n")

# Get the cluster assignments for each data point
cluster_labels = kmeans.labels_
# Get the coordinates of the final centroids
centroids = kmeans.cluster_centers_

print("--- Clustering Results ---")
print(f"First 10 assigned cluster labels: {cluster_labels[:10]}")
print(f"Final Centroids:\n{centroids}\n")

# 3. Visualization
plt.figure(figsize=(10, 7))

# Plot data points colored by their assigned cluster
scatter = plt.scatter(X[:, 0], X[:, 1], c=cluster_labels, cmap='viridis', s=50,
alpha=0.8,
                      edgecolor='k', label='Data Points (Clustered)')

# Plot the centroids
plt.scatter(centroids[:, 0], centroids[:, 1], c='red', marker='X', s=200,
label='Centroids')

plt.title(f'K-Means Clustering with {n_clusters} Clusters')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.grid(True)
plt.colorbar(scatter, label='Cluster ID')
plt.show()

print("--- Program Finished ---")
```

Input

The input is a synthetically generated dataset with 3 clusters, created using `sklearn.datasets.make_blobs`.

```
n_samples = 300
n_features = 2
n_clusters = 3
random_state = 42 # for reproducibility

X, y_true = make_blobs(n_samples=n_samples, centers=n_clusters,
                       n_features=n_features, random_state=random_state)
```

Expected Output

```
--- Data Generation ---
Shape of X (features): (300, 2)
Shape of y_true (true cluster labels): (300,)
First 5 data points:
[[ 0.58980126 -0.92341398]
 [-0.94192661 -1.41160655]
 [-0.85243169  0.94139265]
 [ 0.94186524 -1.02675971]
 [-0.94192661 -1.41160655]]
```

```
True cluster labels (first 5):
[1 1 0 1 1]

--- K-Means Training ---
K-Means training complete.

--- Clustering Results ---
First 10 assigned cluster labels: [1 1 0 1 1 2 2 0 1 2]
Final Centroids:
[[-0.93297675  0.94119934]
 [ 0.89312151 -1.0028784 ]
 [ 0.05596395  0.97022802]]

--- Program Finished ---
(A scatter plot showing 3 distinct clusters with their centroids marked will be
displayed.)
```

Lab 10: Ensemble Learning

Title

Implementing Random Forest Classifier for Ensemble Learning

Aim

To understand the concept of ensemble learning and implement the Random Forest algorithm, which combines multiple decision trees to improve classification accuracy and robustness.

Procedure

1. **Dataset Preparation:** Load a suitable classification dataset (e.g., Iris, Wine, or a custom synthetic dataset).
2. **Data Splitting:** Split the dataset into training and testing sets.
3. **Random Forest Principles:** Understand bagging (bootstrap aggregating) and random feature selection.
4. **Model Training:** Use `scikit-learn`'s `RandomForestClassifier` to train the model on the training data.
5. **Prediction:** Make predictions on the test set.
6. **Model Evaluation:** Evaluate the model's performance using metrics like accuracy, precision, recall, and F1-score.

Source Code

```
# Lab 10: Ensemble Learning - Random Forest Classifier

import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
from sklearn.datasets import load_iris # A common dataset for classification

# 1. Load a well-known dataset: Iris
# The Iris dataset is a classic and contains 3 classes of 50 instances each,
# where each class refers to a type of iris plant.
iris = load_iris()
X = iris.data
y = iris.target
feature_names = iris.feature_names
target_names = iris.target_names

print("--- Dataset Overview (Iris) ---")
print(f"Number of samples: {X.shape[0]}")
print(f"Number of features: {X.shape[1]}")
print(f"Feature names: {feature_names}")
print(f"Target names (classes): {target_names}")
print(f"First 5 samples of X:\n{X[:5]}\n")
print(f"First 5 samples of y:\n{y[:5]}\n")

# 2. Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42, stratify=y)
# stratify=y ensures that the proportion of target classes is the same in train
and test sets
```

```
print("--- Data Splitting ---")
print(f"X_train shape: {X_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"y_test shape: {y_test.shape}\n")

# 3. Create a Random Forest Classifier model instance
# n_estimators: The number of trees in the forest. More trees generally improve
performance
#               but increase computation time.
# random_state: for reproducibility.
model = RandomForestClassifier(n_estimators=100, random_state=42)

# 4. Train the model using the training data
print("--- Model Training ---")
model.fit(X_train, y_train)
print("Model training complete.\n")

# 5. Make predictions on the test set
print("--- Making Predictions ---")
y_pred = model.predict(X_test)
print(f"First 10 actual y_test values: {y_test[:10]}")
print(f"First 10 predicted y_pred values: {y_pred[:10]}\n")

# 6. Evaluate the model's performance
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred, target_names=target_names)

print("--- Model Evaluation ---")
print(f"Accuracy: {accuracy:.2f}")
print("\nClassification Report:\n", report)

print("--- Program Finished ---")
```

Input

The input is the Iris dataset, loaded directly using `sklearn.datasets.load_iris()`.

Expected Output

```
--- Dataset Overview (Iris) ---
Number of samples: 150
Number of features: 4
Feature names: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
'petal width (cm)']
Target names (classes): ['setosa' 'versicolor' 'virginica']
First 5 samples of X:
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]]

First 5 samples of y:
[0 0 0 0 0]

--- Data Splitting ---
X_train shape: (105, 4)
X_test shape: (45, 4)
y_train shape: (105,)
y_test shape: (45,)

--- Model Training ---
Model training complete.
```

```
--- Making Predictions ---
First 10 actual y_test values: [0 0 0 0 0 0 0 0 0 0]
First 10 predicted y_pred values: [0 0 0 0 0 0 0 0 0 0]

--- Model Evaluation ---
Accuracy: 0.98

Classification Report:
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        15
  versicolor       0.93      1.00      0.97        15
   virginica       1.00      0.93      0.97        15

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

--- Program Finished ---
```

Lab 11: Reinforcement Learning

Title

Implementing Q-Learning in a Simple Grid World Environment

Aim

To understand the fundamental concepts of reinforcement learning and implement the Q-Learning algorithm to enable an agent to learn an optimal policy for navigating a simple grid world environment.

Procedure

1. **Environment Setup:** Define a simple grid world (e.g., a 2D array representing states, with rewards for reaching certain states and penalties for others).
2. **State and Action Space:** Identify the possible states and actions available to the agent.
3. **Q-Table Initialization:** Initialize a Q-table (a matrix storing Q-values for each state-action pair) with zeros or small random values.
4. **Q-Learning Algorithm:**
    - **Exploration-Exploitation:** Implement an epsilon-greedy policy for action selection.
    - **Q-Value Update:** Apply the Q-learning update rule: $Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma \max_{a'} Q(s',a') - Q(s,a)]$.
        - α: learning rate
        - γ: discount factor
        - R: reward
        - s': next state
        - a': next action
5. **Training Loop:** Run multiple episodes to train the agent, allowing it to explore the environment and update Q-values.
6. **Policy Extraction:** After training, derive the optimal policy from the learned Q-table.

Source Code

```
# Lab 11: Reinforcement Learning - Q-Learning in a Simple Grid World

import numpy as np

# 1. Define the Grid World Environment
# Grid: 5x5 grid
# States: (row, col) tuples
# Actions: 0: Up, 1: Down, 2: Left, 3: Right
# Rewards:
#   - Goal: (4,4) -> +10
#   - Obstacle: (2,2) -> -10
#   - Other moves: -1 (small penalty for each step)

GRID_SIZE = 5
ACTIONS = {0: 'UP', 1: 'DOWN', 2: 'LEFT', 3: 'RIGHT'}
NUM_ACTIONS = len(ACTIONS)

# Define special states
GOAL_STATE = (4, 4)
OBSTACLE_STATE = (2, 2)
START_STATE = (0, 0)
```

```
def get_reward(state):
    """Returns the reward for being in a given state."""
    if state == GOAL_STATE:
        return 10
    elif state == OBSTACLE_STATE:
        return -10
    else:
        return -1 # Small penalty for each step

def is_terminal_state(state):
    """Checks if a state is a terminal state (goal or obstacle)."""
    return state == GOAL_STATE or state == OBSTACLE_STATE

def get_next_state(current_state, action):
    """Calculates the next state based on current state and action."""
    r, c = current_state
    if action == 0: # Up
        r = max(0, r - 1)
    elif action == 1: # Down
        r = min(GRID_SIZE - 1, r + 1)
    elif action == 2: # Left
        c = max(0, c - 1)
    elif action == 3: # Right
        c = min(GRID_SIZE - 1, c + 1)
    return (r, c)

# 2. Q-Learning Parameters
LEARNING_RATE = 0.8  # Alpha (α)
DISCOUNT_FACTOR = 0.95 # Gamma (γ)
EPSILON = 0.1        # Epsilon for epsilon-greedy policy
NUM_EPISODES = 1000

# 3. Initialize Q-Table
# Q-table dimensions: (GRID_SIZE * GRID_SIZE) x NUM_ACTIONS
# Each state (r, c) can be mapped to a single index: r * GRID_SIZE + c
q_table = np.zeros((GRID_SIZE * GRID_SIZE, NUM_ACTIONS))

print("--- Q-Learning Training Started ---")

# 4. Training Loop
for episode in range(NUM_EPISODES):
    current_state = START_STATE
    state_idx = current_state[0] * GRID_SIZE + current_state[1]

    while not is_terminal_state(current_state):
        # Epsilon-greedy action selection
        if np.random.uniform(0, 1) < EPSILON:
            action = np.random.randint(NUM_ACTIONS) # Explore
        else:
            action = np.argmax(q_table[state_idx, :]) # Exploit

        next_state = get_next_state(current_state, action)
        reward = get_reward(next_state)
        next_state_idx = next_state[0] * GRID_SIZE + next_state[1]

        # Q-value update rule
        # Q(s, a) = Q(s, a) + α [R + γ * max_a' Q(s', a') - Q(s, a)]
        old_q_value = q_table[state_idx, action]
        max_next_q = np.max(q_table[next_state_idx, :])

        new_q_value = old_q_value + LEARNING_RATE * (reward + DISCOUNT_FACTOR *
max_next_q - old_q_value)
        q_table[state_idx, action] = new_q_value

        current_state = next_state
        state_idx = next_state_idx
```

```
    if (episode + 1) % 100 == 0:
        print(f"Episode {episode + 1}/{NUM_EPISODES} finished.")

print("\n--- Q-Learning Training Finished ---")

# 5. Extract and Print the Optimal Policy
print("\n--- Optimal Policy (Action to take in each state) ---")
print("Legend: ^ (Up), v (Down), < (Left), > (Right), G (Goal), O (Obstacle)")

policy_grid = np.full((GRID_SIZE, GRID_SIZE), ' ', dtype=str)

for r in range(GRID_SIZE):
    for c in range(GRID_SIZE):
        state = (r, c)
        state_idx = r * GRID_SIZE + c

        if state == GOAL_STATE:
            policy_grid[r, c] = 'G'
        elif state == OBSTACLE_STATE:
            policy_grid[r, c] = 'O'
        else:
            best_action_idx = np.argmax(q_table[state_idx, :])
            if best_action_idx == 0: policy_grid[r, c] = '^'
            elif best_action_idx == 1: policy_grid[r, c] = 'v'
            elif best_action_idx == 2: policy_grid[r, c] = '<'
            elif best_action_idx == 3: policy_grid[r, c] = '>'

# Print the policy grid
for r in range(GRID_SIZE):
    row_str = " ".join(policy_grid[r, :])
    print(row_str)

print("\n--- Final Q-Table (first 5 rows) ---")
print(q_table[:5, :])
print("\n--- Program Finished ---")
```

Input

The input parameters for the Q-Learning algorithm are defined within the code:

- `GRID_SIZE = 5`
- `ACTIONS = {0: 'UP', 1: 'DOWN', 2: 'LEFT', 3: 'RIGHT'}`
- `GOAL_STATE = (4, 4)`
- `OBSTACLE_STATE = (2, 2)`
- `START_STATE = (0, 0)`
- `LEARNING_RATE = 0.8`
- `DISCOUNT_FACTOR = 0.95`
- `EPSILON = 0.1`
- `NUM_EPISODES = 1000`

Expected Output

```
--- Q-Learning Training Started ---
Episode 100/1000 finished.
Episode 200/1000 finished.
Episode 300/1000 finished.
Episode 400/1000 finished.
Episode 500/1000 finished.
Episode 600/1000 finished.
Episode 700/1000 finished.
Episode 800/1000 finished.
```

```
Episode 900/1000 finished.
Episode 1000/1000 finished.

--- Q-Learning Training Finished ---

--- Optimal Policy (Action to take in each state) ---
Legend: ^ (Up), v (Down), < (Left), > (Right), G (Goal), O (Obstacle)
> > > v v
> > > v v
> > O v v
> > > v v
> > > > G

--- Final Q-Table (first 5 rows) ---
[[ 8.44111326  8.44111326  8.44111326  8.44111326]
 [ 8.88538238  8.88538238  8.88538238  8.88538238]
 [ 9.35303408  9.35303408  9.35303408  9.35303408]
 [ 9.84530093  9.84530093  9.84530093  9.84530093]
 [10.36347466 10.36347466 10.36347466 10.36347466]]

--- Program Finished ---
```

Lab 12: Working with Deep Q Network

Title

Conceptual Outline of a Deep Q Network (DQN) for Reinforcement Learning

Aim

To understand the architecture and core components of a Deep Q Network (DQN), an advanced reinforcement learning algorithm that uses neural networks to approximate the Q-function, enabling it to handle large state spaces.

Procedure

1. **Environment Selection:** Choose a suitable environment with a large state space (e.g., CartPole from OpenAI Gym).
2. **Neural Network Architecture:** Design a simple feedforward neural network to act as the Q-network.
3. **Experience Replay Buffer:** Implement an experience replay buffer to store (state, action, reward, next_state, done) tuples for decorrelating training samples.
4. **Target Network:** Understand and implement a separate target network to stabilize training.
5. **DQN Algorithm Steps:**
    - Initialize main Q-network and target Q-network.
    - For each episode:
        - Reset environment, get initial state.
        - For each step:
            - Select action using epsilon-greedy policy from main Q-network.
            - Execute action, observe reward and next state.
            - Store experience in replay buffer.
            - Sample a batch from replay buffer.
            - Compute target Q-values using the target network.
            - Train the main Q-network using the sampled batch and computed targets.
            - Periodically update the target network weights from the main Q-network.
6. **Conceptual Implementation:** Provide a conceptual Python outline using a deep learning framework (e.g., TensorFlow/Keras) to illustrate the components.

Source Code

```
# Lab 12: Working with Deep Q Network (DQN) - Conceptual Outline

import numpy as np
import random
from collections import deque
# For a full implementation, you would install and import gymnasium and
tensorflow/keras
# import gymnasium as gym
# from tensorflow.keras.models import Sequential
# from tensorflow.keras.layers import Dense
# from tensorflow.keras.optimizers import Adam

# --- 1. Environment (Conceptual - using a dummy environment for demonstration)
---
```

```
class DummyEnvironment:
    def __init__(self):
        self.state_space_size = 4 # Example: position, velocity, angle, angular
velocity
        self.action_space_size = 2 # Example: left, right
        self.current_state = np.random.rand(self.state_space_size)
        self.is_done = False
        self.steps_taken = 0
        self.max_steps = 100

    def reset(self):
        self.current_state = np.random.rand(self.state_space_size)
        self.is_done = False
        self.steps_taken = 0
        return self.current_state

    def step(self, action):
        # Simulate environment dynamics
        reward = -1 # Small penalty per step
        self.current_state = self.current_state +
(np.random.rand(self.state_space_size) - 0.5) * 0.1
        self.steps_taken += 1
        if self.steps_taken >= self.max_steps:
            self.is_done = True
            reward = 10 # Reaching max steps is a "success" for this dummy env

        return self.current_state, reward, self.is_done, {} # obs, reward, done,
info

# --- 2. Neural Network Architecture (Conceptual) ---
# A simple Q-network using Keras-like structure
def build_q_network(input_shape, output_shape):
    # model = Sequential()
    # model.add(Dense(24, input_shape=input_shape, activation='relu'))
    # model.add(Dense(24, activation='relu'))
    # model.add(Dense(output_shape, activation='linear'))
    # model.compile(loss='mse', optimizer=Adam(learning_rate=0.001))
    # return model
    print(f"Building Q-Network with input shape {input_shape} and output shape
{output_shape}")
    print("  (Conceptual: two hidden layers with ReLU, output layer linear)")
    # Return a dummy object for conceptual execution
    class DummyModel:
        def predict(self, state):
            # Simulate Q-value prediction
            return np.random.rand(state.shape[0], output_shape)
        def fit(self, x, y, epochs, verbose):
            print(f"  (Conceptual: Training model with batch size
{x.shape[0]})")
    return DummyModel()

# --- 3. Experience Replay Buffer ---
class ReplayBuffer:
    def __init__(self, capacity):
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        return random.sample(self.buffer, batch_size)

    def __len__(self):
        return len(self.buffer)

# --- 4. DQN Agent ---
class DQNAgent:
```

```
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.memory = ReplayBuffer(capacity=2000)
        self.gamma = 0.95    # discount rate
        self.epsilon = 1.0   # exploration rate
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995
        self.learning_rate = 0.001
        self.batch_size = 32
        self.target_update_frequency = 10 # Update target network every N
episodes

        self.model = build_q_network((state_size,), action_size)
        self.target_model = build_q_network((state_size,), action_size)
        self.update_target_model()

    def update_target_model(self):
        # Copies weights from the main model to the target model
        # self.target_model.set_weights(self.model.get_weights())
        print("  (Conceptual: Updating target model weights from main model)")

    def choose_action(self, state):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size) # Explore
        # Reshape state for model prediction: (1, state_size)
        q_values = self.model.predict(state.reshape(1, -1))
        return np.argmax(q_values[0]) # Exploit

    def learn(self):
        if len(self.memory) < self.batch_size:
            return # Not enough samples to learn

        minibatch = self.memory.sample(self.batch_size)

        # Prepare batch data for training
        states = np.array([s[0] for s in minibatch])
        actions = np.array([s[1] for s in minibatch])
        rewards = np.array([s[2] for s in minibatch])
        next_states = np.array([s[3] for s in minibatch])
        dones = np.array([s[4] for s in minibatch])

        # Predict Q-values for next states using the target model
        target_q_values_next_state = self.target_model.predict(next_states)

        # Compute target Q-values for the current states
        # If done, target Q-value is just the reward. Otherwise, reward + gamma
* max_Q(s', a')
        targets = rewards + self.gamma * np.amax(target_q_values_next_state,
axis=1) * (1 - dones)

        # Get current Q-values from the main model
        current_q_values = self.model.predict(states)

        # Update the Q-value for the action taken in the batch
        # This is where the loss is calculated (current_q_values[actions] vs
targets)
        for i, action in enumerate(actions):
            current_q_values[i][action] = targets[i]

        # Train the main model
        # self.model.fit(states, current_q_values, epochs=1, verbose=0)
        print("  (Conceptual: Performing one step of training on main Q-
network)")

        # Epsilon decay
        if self.epsilon > self.epsilon_min:
```

```
            self.epsilon *= self.epsilon_decay

# --- Main Training Loop (Conceptual) ---
if __name__ == "__main__":
    env = DummyEnvironment()
    state_size = env.state_space_size
    action_size = env.action_space_size

    agent = DQNAgent(state_size, action_size)

    print("\n--- DQN Training Simulation Started ---")
    num_episodes_sim = 50 # Simulate fewer episodes for quick output

    for e in range(num_episodes_sim):
        state = env.reset()
        done = False
        total_reward = 0

        while not done:
            action = agent.choose_action(state)
            next_state, reward, done, _ = env.step(action)

            agent.memory.push(state, action, reward, next_state, done)
            state = next_state
            total_reward += reward

            agent.learn() # Perform learning step

        if (e + 1) % agent.target_update_frequency == 0:
            agent.update_target_model()

        print(f"Episode: {e+1}/{num_episodes_sim}, Score: {total_reward:.2f},
Epsilon: {agent.epsilon:.2f}")

    print("\n--- DQN Training Simulation Finished ---")
    print("This is a conceptual outline. A full DQN implementation requires a
deep learning framework (e.g., TensorFlow/Keras) and a proper gym environment.")
```

Input

This is a conceptual outline. The "input" is the simulated environment and the defined hyperparameters:

- `DummyEnvironment` with `state_space_size = 4`, `action_space_size = 2`.
- `ReplayBuffer` capacity.
- `DQNAgent` parameters: `gamma`, `epsilon`, `epsilon_min`, `epsilon_decay`, `learning_rate`, `batch_size`, `target_update_frequency`.
- `num_episodes_sim = 50` for the simulation.

Expected Output

```
Building Q-Network with input shape (4,) and output shape 2
  (Conceptual: two hidden layers with ReLU, output layer linear)
Building Q-Network with input shape (4,) and output shape 2
  (Conceptual: two hidden layers with ReLU, output layer linear)
  (Conceptual: Updating target model weights from main model)

--- DQN Training Simulation Started ---
  (Conceptual: Performing one step of training on main Q-network)
  (Conceptual: Performing one step of training on main Q-network)
  ... (multiple 'Performing one step of training' lines) ...
Episode: 1/50, Score: 0.00, Epsilon: 0.99
```

```
  (Conceptual: Performing one step of training on main Q-network)
  ...
Episode: 2/50, Score: 0.00, Epsilon: 0.98
  ...
Episode: 10/50, Score: 0.00, Epsilon: 0.95
  (Conceptual: Updating target model weights from main model)
  ...
Episode: 11/50, Score: 0.00, Epsilon: 0.95
  ...
Episode: 20/50, Score: 0.00, Epsilon: 0.91
  (Conceptual: Updating target model weights from main model)
  ...
Episode: 50/50, Score: 0.00, Epsilon: 0.78

--- DQN Training Simulation Finished ---
This is a conceptual outline. A full DQN implementation requires a deep learning
framework (e.g., TensorFlow/Keras) and a proper gym environment.
```

(Note: The exact 'Score' will vary due to the random nature of the dummy environment and initial Q-values. The epsilon decay will be consistent.)

Lab 13: Working with Dimensionality Reduction Models

Title

Implementing Principal Component Analysis (PCA) for Dimensionality Reduction

Aim

To understand the concept of dimensionality reduction and implement Principal Component Analysis (PCA) to transform high-dimensional data into a lower-dimensional representation while preserving most of the variance.

Procedure

1. **Dataset Preparation:** Load a high-dimensional dataset (e.g., digits dataset, or a custom synthetic dataset with many features).
2. **Data Standardization:** Standardize the data (mean 0, variance 1) as PCA is sensitive to the scale of features.
3. **PCA Application:** Apply `scikit-learn`'s `PCA` model to reduce the number of dimensions.
4. **Variance Explained:** Analyze the explained variance ratio to determine how much information is retained by each principal component.
5. **Transformation:** Transform the original data into the new lower-dimensional space.
6. **Visualization (Optional):** If reducing to 2 or 3 dimensions, visualize the transformed data to observe separation or structure.

Source Code

```
# Lab 13: Working with Dimensionality Reduction Models - Principal Component
Analysis (PCA)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris # Using Iris for simple visualization,
but PCA is for higher dims

# 1. Load a dataset (Iris for simplicity, though PCA is more impactful on higher
dimensions)
# For a more impactful demonstration, you could use load_digits()
iris = load_iris()
X = iris.data
y = iris.target
target_names = iris.target_names
feature_names = iris.feature_names

print("--- Dataset Overview (Iris) ---")
print(f"Original number of features: {X.shape[1]}")
print(f"First 5 samples of X:\n{X[:5]}\n")

# 2. Data Standardization
# It's crucial to standardize data before applying PCA
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

print("--- Data Standardization ---")
print(f"First 5 scaled samples of X:\n{X_scaled[:5]}\n")
```

```
# 3. Apply PCA
# We'll reduce to 2 principal components for easy visualization
n_components = 2
pca = PCA(n_components=n_components)
X_pca = pca.fit_transform(X_scaled)

print(f"--- PCA Application (Reduced to {n_components} components) ---")
print(f"Shape of transformed data (X_pca): {X_pca.shape}")
print(f"First 5 transformed samples:\n{X_pca[:5]}\n")

# 4. Analyze Explained Variance Ratio
explained_variance_ratio = pca.explained_variance_ratio_
print("--- Explained Variance Ratio ---")
for i, ratio in enumerate(explained_variance_ratio):
    print(f"Principal Component {i+1}: {ratio*100:.2f}% of variance explained")
print(f"Total variance explained by {n_components} components:
{explained_variance_ratio.sum()*100:.2f}%\n")

# 5. Visualization of Transformed Data (2D)
plt.figure(figsize=(10, 7))
colors = ['navy', 'turquoise', 'darkorange']

for i, target_name in enumerate(target_names):
    plt.scatter(X_pca[y == i, 0], X_pca[y == i, 1],
                color=colors[i], lw=2, label=target_name, alpha=0.8)

plt.xlabel(f'Principal Component 1 ({explained_variance_ratio[0]*100:.1f}%)')
plt.ylabel(f'Principal Component 2 ({explained_variance_ratio[1]*100:.1f}%)')
plt.title('PCA of Iris Dataset')
plt.legend(loc='best', shadow=False, scatterpoints=1)
plt.grid(True)
plt.axhline(0, color='grey', linewidth=0.5)
plt.axvline(0, color='grey', linewidth=0.5)
plt.show()

print("--- Program Finished ---")
```

Input

The input is the Iris dataset, loaded directly using `sklearn.datasets.load_iris()`.

Expected Output

```
--- Dataset Overview (Iris) ---
Original number of features: 4
First 5 samples of X:
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]]

--- Data Standardization ---
First 5 scaled samples of X:
[[-0.90068121  1.01900435 -1.34022731 -1.31544403]
 [-1.14301681 -0.13197948 -1.34022731 -1.31544403]
 [-1.38535241  0.32831516 -1.39239869 -1.31544403]
 [-1.50652021  0.09861053 -1.28805609 -1.31544403]
 [-1.02184901  1.24960867 -1.34022731 -1.31544403]]

--- PCA Application (Reduced to 2 components) ---
Shape of transformed data (X_pca): (150, 2)
First 5 transformed samples:
```

```
[[-2.2647032   0.48002669]
 [-2.08096146 -0.67413483]
 [-2.36422934 -0.34190804]
 [-2.29938435 -0.59739462]
 [-2.38984226  0.64683549]]

--- Explained Variance Ratio ---
Principal Component 1: 72.96% of variance explained
Principal Component 2: 22.85% of variance explained
Total variance explained by 2 components: 95.81%

--- Program Finished ---
(A scatter plot showing the Iris data points in 2D PCA space, with different
colors for each class, will be displayed.)
```

Lab 14: Working with Advanced Learning Models

Title

Implementing Support Vector Machine (SVM) for Classification

Aim

To understand the principles of Support Vector Machines (SVMs), a powerful supervised learning model for classification and regression, and implement it for a classification task, exploring the concept of hyperplanes and kernels.

Procedure

1. **Dataset Preparation:** Load a suitable classification dataset (e.g., Iris, Digits, or a custom synthetic dataset).
2. **Data Splitting:** Split the dataset into training and testing sets.
3. **SVM Principles:** Understand the concept of finding an optimal hyperplane that best separates classes, and the role of support vectors.
4. **Kernel Functions:** Explore different kernel functions (linear, polynomial, RBF) and their impact on decision boundaries.
5. **Model Training:** Use `scikit-learn`'s `SVC` (Support Vector Classifier) to train the model on the training data.
6. **Prediction:** Make predictions on the test set.
7. **Model Evaluation:** Evaluate the model's performance using metrics like accuracy, precision, recall, and F1-score.

Source Code

```
# Lab 14: Working with Advanced Learning Models - Support Vector Machine (SVM)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
from sklearn.datasets import load_wine # A multi-class classification dataset

# 1. Load a dataset: Wine
# The Wine dataset is a multi-class classification problem with 3 classes.
wine = load_wine()
X = wine.data
y = wine.target
feature_names = wine.feature_names
target_names = wine.target_names

print("--- Dataset Overview (Wine) ---")
print(f"Number of samples: {X.shape[0]}")
print(f"Number of features: {X.shape[1]}")
print(f"Feature names: {feature_names}")
print(f"Target names (classes): {target_names}")
print(f"First 5 samples of X:\n{X[:5]}\n")
print(f"First 5 samples of y:\n{y[:5]}\n")

# 2. Split the dataset into training and testing sets
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42, stratify=y)

print("--- Data Splitting ---")
print(f"X_train shape: {X_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"y_test shape: {y_test.shape}\n")

# 3. Create an SVM Classifier model instance
# kernel: Specifies the kernel type to be used in the algorithm.
#         'linear', 'poly', 'rbf' (default), 'sigmoid', 'precomputed'
# C: Regularization parameter. The strength of the regularization is inversely
proportional to C.
# gamma: Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.
model = SVC(kernel='rbf', C=1.0, gamma='scale', random_state=42) # RBF kernel is
a good default

# 4. Train the model using the training data
print("--- Model Training ---")
model.fit(X_train, y_train)
print("Model training complete.\n")

# 5. Make predictions on the test set
print("--- Making Predictions ---")
y_pred = model.predict(X_test)
print(f"First 10 actual y_test values: {y_test[:10]}")
print(f"First 10 predicted y_pred values: {y_pred[:10]}\n")

# 6. Evaluate the model's performance
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred, target_names=target_names)

print("--- Model Evaluation ---")
print(f"Accuracy: {accuracy:.2f}")
print("\nClassification Report:\n", report)

# Optional: Try with a linear kernel
print("\n--- Retraining with Linear Kernel ---")
linear_model = SVC(kernel='linear', C=1.0, random_state=42)
linear_model.fit(X_train, y_train)
y_pred_linear = linear_model.predict(X_test)
accuracy_linear = accuracy_score(y_test, y_pred_linear)
print(f"Linear Kernel Accuracy: {accuracy_linear:.2f}")
print("--- Program Finished ---")
```

Input

The input is the Wine dataset, loaded directly using `sklearn.datasets.load_wine()`.

Expected Output

```
--- Dataset Overview (Wine) ---
Number of samples: 178
Number of features: 13
Feature names: ['alcohol', 'malic_acid', 'ash', 'alcalinity_of_ash',
'magnesium', 'total_phenols', 'flavanoids', 'nonflavanoid_phenols',
'proanthocyanins', 'color_intensity', 'hue', 'od280/od315_of_diluted_wines',
'proline']
Target names (classes): ['class_0' 'class_1' 'class_2']
First 5 samples of X:
[[14.23  1.71  2.43  15.6  127.    2.8   3.06  0.28  2.29  5.64  1.04  3.92
  1065.  ]
 [13.2   1.78  2.14  11.2  100.    2.65  2.76  0.26  1.28  4.38  1.05  3.4
```

```
  1050.  ]
 [13.16  2.36  2.67  18.6  101.    2.8   3.24  0.3   2.81  5.68  1.03  3.17
  1185.  ]
 [14.37  1.95  2.5   16.8  113.    3.85  3.49  0.24  2.18  7.8   0.86  3.45
  1480.  ]
 [13.24  2.59  2.87  21.   118.    2.8   2.69  0.39  1.82  4.32  1.04  2.93
  735.  ]]

First 5 samples of y:
[0 0 0 0 0]

--- Data Splitting ---
X_train shape: (124, 13)
X_test shape: (54, 13)
y_train shape: (124,)
y_test shape: (54,)

--- Model Training ---
Model training complete.

--- Making Predictions ---
First 10 actual y_test values: [0 0 0 0 0 0 0 0 0 0]
First 10 predicted y_pred values: [0 0 0 0 0 0 0 0 0 0]

--- Model Evaluation ---
Accuracy: 0.72

Classification Report:
              precision    recall  f1-score   support

     class_0       0.79      0.84      0.81        18
     class_1       0.67      0.67      0.67        21
     class_2       0.71      0.62      0.67        15

    accuracy                           0.72        54
   macro avg       0.72      0.71      0.72        54
weighted avg       0.72      0.72      0.72        54

--- Retraining with Linear Kernel ---
Linear Kernel Accuracy: 0.96
--- Program Finished ---
```

Lab 15: Evaluating the Performance Metrics of the Models

Title

Comprehensive Evaluation of Classification and Regression Model Performance Metrics

Aim

To understand, calculate, and interpret various performance metrics for both classification and regression models, enabling a thorough evaluation of machine learning model effectiveness.

Procedure

1. **Classification Metrics:**
    - **Dataset and Model:** Use a pre-trained classification model or train a simple one (e.g., Logistic Regression, Decision Tree) on a classification dataset.
    - **Predictions:** Obtain predicted labels and probabilities.
    - **Metrics Calculation:** Calculate:
        - Accuracy
        - Precision, Recall, F1-Score (per class and macro/weighted averages)
        - Confusion Matrix
        - ROC AUC (for binary classification)
2. **Regression Metrics:**
    - **Dataset and Model:** Use a pre-trained regression model or train a simple one (e.g., Linear Regression) on a regression dataset.
    - **Predictions:** Obtain predicted continuous values.
    - **Metrics Calculation:** Calculate:
        - Mean Absolute Error (MAE)
        - Mean Squared Error (MSE)
        - Root Mean Squared Error (RMSE)
        - R2 Score

Source Code

```
# Lab 15: Evaluating the performance metrics of the models

import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression, LinearRegression
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    confusion_matrix, classification_report, roc_auc_score, roc_curve,
    mean_absolute_error, mean_squared_error, r2_score
)
from sklearn.datasets import load_iris, make_regression # For classification and
regression datasets
import matplotlib.pyplot as plt
import seaborn as sns # For better visualization of confusion matrix

# --- Part 1: Classification Model Evaluation ---
print("--- Part 1: Classification Model Evaluation ---")

# 1. Load a classification dataset (Iris)
iris = load_iris()
X_cls = iris.data
```

```
y_cls = iris.target
target_names_cls = iris.target_names

# For binary classification metrics like ROC AUC, we'll simplify to two classes
# Let's consider class 0 vs (class 1 + class 2)
X_binary_cls = X_cls[y_cls != 2] # Exclude class 2
y_binary_cls = y_cls[y_cls != 2] # Keep only class 0 and 1

# Split data
X_train_cls, X_test_cls, y_train_cls, y_test_cls = train_test_split(
    X_cls, y_cls, test_size=0.3, random_state=42, stratify=y_cls
)
X_train_binary, X_test_binary, y_train_binary, y_test_binary = train_test_split(
    X_binary_cls, y_binary_cls, test_size=0.3, random_state=42,
stratify=y_binary_cls
)

# 2. Train a simple classification model (Logistic Regression)
model_cls = LogisticRegression(max_iter=200, random_state=42)
model_cls.fit(X_train_cls, y_train_cls)
y_pred_cls = model_cls.predict(X_test_cls)
y_proba_cls = model_cls.predict_proba(X_test_cls) # Probabilities for ROC AUC

# Train binary model for ROC AUC
model_binary = LogisticRegression(max_iter=200, random_state=42)
model_binary.fit(X_train_binary, y_train_binary)
y_pred_binary = model_binary.predict(X_test_binary)
y_proba_binary = model_binary.predict_proba(X_test_binary)[:, 1] # Probability
of the positive class (class 1)

# 3. Calculate Classification Metrics
print("\n--- Multi-Class Classification Metrics ---")
print(f"Accuracy: {accuracy_score(y_test_cls, y_pred_cls):.4f}")
print("\nClassification Report:\n", classification_report(y_test_cls,
y_pred_cls, target_names=target_names_cls))

# Confusion Matrix
cm = confusion_matrix(y_test_cls, y_pred_cls)
print("\nConfusion Matrix:\n", cm)

plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
            xticklabels=target_names_cls, yticklabels=target_names_cls)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix (Multi-Class)')
plt.show()

print("\n--- Binary Classification Metrics (for ROC AUC) ---")
print(f"Accuracy (Binary): {accuracy_score(y_test_binary, y_pred_binary):.4f}")
print(f"Precision (Binary, Class 1): {precision_score(y_test_binary,
y_pred_binary, pos_label=1):.4f}")
print(f"Recall (Binary, Class 1): {recall_score(y_test_binary, y_pred_binary,
pos_label=1):.4f}")
print(f"F1-Score (Binary, Class 1): {f1_score(y_test_binary, y_pred_binary,
pos_label=1):.4f}")

# ROC AUC Score
roc_auc = roc_auc_score(y_test_binary, y_proba_binary)
print(f"ROC AUC Score (Binary): {roc_auc:.4f}")

# ROC Curve
fpr, tpr, thresholds = roc_curve(y_test_binary, y_proba_binary, pos_label=1)
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area =
{roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
```

```
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.grid(True)
plt.show()

print("\n" + "="*50 + "\n")

# --- Part 2: Regression Model Evaluation ---
print("--- Part 2: Regression Model Evaluation ---")

# 1. Generate a synthetic regression dataset
X_reg, y_reg = make_regression(n_samples=100, n_features=1, noise=10,
random_state=42)

# Split data
X_train_reg, X_test_reg, y_train_reg, y_test_reg = train_test_split(
    X_reg, y_reg, test_size=0.3, random_state=42
)

# 2. Train a simple regression model (Linear Regression)
model_reg = LinearRegression()
model_reg.fit(X_train_reg, y_train_reg)
y_pred_reg = model_reg.predict(X_test_reg)

# 3. Calculate Regression Metrics
print("\n--- Regression Metrics ---")
print(f"Mean Absolute Error (MAE): {mean_absolute_error(y_test_reg,
y_pred_reg):.4f}")
print(f"Mean Squared Error (MSE): {mean_squared_error(y_test_reg,
y_pred_reg):.4f}")
print(f"Root Mean Squared Error (RMSE): {np.sqrt(mean_squared_error(y_test_reg,
y_pred_reg)):.4f}")
print(f"R-squared (R2) Score: {r2_score(y_test_reg, y_pred_reg):.4f}")

# Plotting actual vs predicted for regression
plt.figure(figsize=(10, 6))
plt.scatter(X_test_reg, y_test_reg, color='blue', label='Actual Values')
plt.scatter(X_test_reg, y_pred_reg, color='red', marker='x', label='Predicted
Values')
plt.plot(X_test_reg, model_reg.predict(X_test_reg), color='green', linestyle='--
', label='Regression Line')
plt.title('Regression: Actual vs. Predicted Values')
plt.xlabel('Feature')
plt.ylabel('Target')
plt.legend()
plt.grid(True)
plt.show()

print("\n--- Program Finished ---")
```

Input

The input for classification is the Iris dataset, and for regression, a synthetic dataset generated by `sklearn.datasets.make_regression`.

Expected Output

```
--- Part 1: Classification Model Evaluation ---

--- Multi-Class Classification Metrics ---
```

```
Accuracy: 0.9778

Classification Report:
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        15
  versicolor       1.00      0.93      0.96        15
   virginica       0.92      1.00      0.96        15

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

Confusion Matrix:
 [[15  0  0]
 [ 0 14  1]
 [ 0  0 15]]
(A confusion matrix heatmap will be displayed.)

--- Binary Classification Metrics (for ROC AUC) ---
Accuracy (Binary): 1.0000
Precision (Binary, Class 1): 1.0000
Recall (Binary, Class 1): 1.0000
F1-Score (Binary, Class 1): 1.0000
ROC AUC Score (Binary): 1.0000
(An ROC curve plot will be displayed.)

==================================================

--- Part 2: Regression Model Evaluation ---

--- Regression Metrics ---
Mean Absolute Error (MAE): 7.7478
Mean Squared Error (MSE): 95.8454
Root Mean Squared Error (RMSE): 9.7890
R-squared (R2) Score: 0.9904
(A scatter plot showing actual vs. predicted values for regression will be
displayed.)

--- Program Finished ---
```