

SRM VALLIAMMAI ENGINEERING COLLEGE

(An Autonomous Institution)

S.R.M Nagar, Kattankulathur – 603 203

Department of Electronics and Communication Engineering



Laboratory Manual

**EC3763 – ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING
TECHNIQUES**

(Common to VII Semester Medical Electronics)

Regulation - 2023

Semester/Branch : VII semester ECE

Academic Year : 2026 - 2027 (ODD SEMESTER)

Prepared By : Dr. N. Subhashini, *Associate Professor* / ECE

Dr. C. Amali, *Associate Professor* / ECE

Dr. G. Sathish Kumar, *Assistant Professor(Sel.G)/ECE*

Dr. P. Vanmathi, *Assistant Professor (O.G)/ MDE*



SRM VALLIAMMAI ENGINEERING COLLEGE

(An Autonomous Institution)

SRM Nagar, Kattankulathur – 603 203



DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

VISION OF THE INSTITUTE

Educate to excel in social transformation

MISSION OF THE INSTITUTE

- ❖ To contribute to the development of human resources in the form of professional engineers and managers of international excellence and competence with high motivation and dynamism, who besides serving as ideal citizen of our country will contribute substantially to the economic development and advancement in their chosen areas of specialization.
- ❖ To build the institution with international repute in education in several areas at several levels with specific emphasis to promote higher education and research through strong institute-industry interaction and consultancy.

VISION OF THE DEPARTMENT

To excel in the field of electronics and communication engineering and to develop highly competent technocrats with global intellectual qualities.

MISSION OF THE DEPARTMENT

M1: To train the students with the state-of-art technologies and to develop innovative solutions to cater to the societal needs.

M2: To orient the students towards global career with universal moral values and professional ethics.



SRM VALLIAMMAI ENGINEERING COLLEGE

(An Autonomous Institution)

SRM Nagar, Kattankulathur – 603 203



PROGRAM OUTCOMES

PO1: Engineering Knowledge: Apply knowledge of mathematics, natural science, computing, engineering fundamentals and an engineering specialization to develop to the solution of complex engineering problems.

PO2: Problem Analysis: Identify, formulate, review research literature and analyze complex engineering problems reaching substantiated conclusions with consideration for sustainable development.

PO3: Design/Development of Solutions: Design creative solutions for complex engineering problems and design/develop systems/components/processes to meet identified needs with consideration for the public health and safety, whole-life cost, net zero carbon, culture, society and environment as required.

PO4: Conduct Investigations of Complex Problems: Conduct investigations of complex engineering problems using research-based knowledge including design of experiments, modelling, analysis & interpretation of data to provide valid conclusions.

PO5: Engineering Tool Usage: Create, select and apply appropriate techniques, resources and modern engineering & IT tools, including prediction and modelling recognizing their limitations to solve complex engineering problems.

PO6: The Engineer and The World: Analyze and evaluate societal and environmental aspects while solving complex engineering problems for its impact on sustainability with reference to economy, health, safety, legal framework, culture and environment.

PO7: Ethics: Apply ethical principles and commit to professional ethics, human values, diversity and inclusion; adhere to national & international laws.

PO8: Individual and Collaborative Team work: Function effectively as an individual, and as a member or leader in diverse/multi-disciplinary teams.

PO9: Communication: Communicate effectively and inclusively within the engineering community and society at large, such as being able to comprehend and

write effective reports and design documentation, make effective presentations considering cultural, language, and learning differences

PO10: Project Management and Finance: Apply knowledge and understanding of engineering management principles and economic decision-making and apply these to one's own work, as a member and leader in a team, and to manage projects and in multidisciplinary environments.

PO11: Life-Long Learning: Recognize the need for, and have the preparation and ability for independent and life-long learning, adaptability to new and emerging technologies and critical thinking in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOME (PSOs)

1. Ability to solve societal problems by applying knowledge acquired in the core domain.
2. Ability to apply technologies like VLSI, Embedded and Wireless Communications to meet the industrial requirements.

Syllabus

EC3763 ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING TECHNIQUES

L T P C

3 0 2 4

OBJECTIVES:

- Study about uninformed and Heuristic search techniques
- Learn techniques for reasoning under uncertainty
- Introduce Machine Learning and supervised learning algorithms
- Study about ensembling and unsupervised learning algorithms
- Learn the neural networks and graphical models
- Explore the applications based on AI & ML

LIST OF EXPERIMENTS

1. Implementation of Uninformed search algorithms (BFS, DFS)
2. Implementation of Informed search algorithms (A*, memory-bounded A*)
3. Implement naïve Bayes models
4. Implement Bayesian Networks
5. Build Regression models
6. Build decision trees and random forests
7. Build SVM models
8. Implement ensembling techniques
9. Implement clustering algorithms
10. Implement EM for Bayesian networks
11. Build simple NN models

TOPIC BEYOND THE SYLLABUS EXPERIMENT

12. Build Deep Learning Neural Network model

Total: 45 Periods

OUTCOMES:

On completion of this lab course, the student would be able to

- Understand AI concepts, applications, and problem-solving using search strategies
- Apply uninformed, heuristic, local, and adversarial search methods.
- Use probabilistic models like Bayesian networks for reasoning under uncertainty.
- Implement supervised learning techniques including regression and classification.
- Apply ensemble learning and unsupervised methods such as K-means and GMM.
- Explore evolutionary algorithms, reinforcement learning, and graphical models and applications

.COURSE OUTCOMES - PROGRAM OUTCOMES MATRIX

CO	PO											PSO	
	1	2	3	4	5	6	7	8	9	10	11	1	2
EC3763.1	3	2	2	1	1	-	-	-	-	-	-	3	2
EC3763.2	3	3	3	2	1	-	-	-	-	-	-	3	2
EC3763.3	3	3	2	2	2	-	-	-	-	-	-	3	3
EC3763.4	3	3	3	2	2	-	-	-	-	-	-	3	3
EC3763.5	3	2	2	2	2	-	-	-	-	-	-	3	3
EC3763.6	3	3	3	3	2	-	-	-	-	-	-	3	3
AVG	3	3	2	2	2	-	-	-	-	-	-	3	2

List of Experiments

Expt. No.	Name of the Experiment	Page No.
1.	(a)Implementation of Uninformed search algorithms – Breadth First Search	8
	(b)Implementation of Uninformed search algorithms – Depth First Search	10
2.	(a)Implementation of Informed search algorithms - A*	12
	(b)Implementation of Informed search algorithms - memory-bounded A*	16
3.	Implementation of Naive Bayes model	20
4.	Implement Bayesian Network	22
5.	Build Regression model	26
6.	Build decision trees and random forests	30
7.	Build SVM model	33
8.	Implement ensembling techniques	36
9.	Implement clustering algorithms	41
10.	Implement EM for Bayesian networks	44
11.	Build simple Neural Network(NN) model	48
TOPIC BEYOND THE SYLLABUS		
12.	Build Deep Neural Network model	52

Expt. No. 1(a) Implementation of Uninformed search algorithms – Breadth First Search

Aim :

To Implement Breadth First Search algorithm using Python.

Software Required :

Python

Pre Lab Questions :

1. What is Breadth First Search (BFS)?
2. What data structure is used in BFS?
3. How is BFS different from DFS?
4. What is the time complexity of BFS?
5. What are the real-world applications of BFS?

Theory:

Breadth First Search (BFS) is an uninformed graph traversal algorithm that explores all nodes at the current level before moving to the next level. It starts from a source node and visits its neighboring nodes first, followed by the neighbors of those nodes, thereby traversing the graph level by level. BFS uses a queue (FIFO) data structure to keep track of nodes that need to be visited. It is widely used for finding the shortest path in an unweighted graph, network broadcasting, web crawling, and pathfinding problems. The time complexity of BFS is $O(V + E)$, where V represents the number of vertices and E represents the number of edges in the graph. Since BFS systematically explores all reachable nodes, it guarantees finding the shortest path in terms of the number of edges between the source and destination nodes.

Algorithm :

1. Start from the initial node.
2. Mark the node as visited and add it to a queue.
3. Repeat until the queue is empty:
 - Remove a node from the front of the queue.
 - Process the node (e.g., print it).
 - Add all unvisited neighbors to the queue and mark them as visited.

Program :

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    while queue:
        vertex = queue.popleft()
        if vertex not in visited:
            print(vertex, end=" ")
            visited.add(vertex)
            queue.extend(set(graph[vertex]) - visited)

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

print("Breadth First Search starting from node A:")
bfs(graph, 'A')
```

Output:

Breadth First Search starting from node A:

A B C D E F

Post lab Questions

1. Can BFS be used to find the shortest path in an unweighted graph?
2. What happens if a node is visited more than once in BFS?
3. Is BFS complete and optimal? Justify.
4. How does BFS behave in a cyclic graph?
5. How can you implement BFS in Python?

Result:

Thus the Breadth First Search algorithm was implemented using Python.

Expt. No. 1(b) Implementation of Uninformed search algorithms – Depth First Search

Aim : To implement the Depth First Search (DFS) algorithm using Python.

Software Required :
Python

Pre lab Questions:

1. What is Depth First Search (DFS)?
2. What data structure is used to implement DFS?
3. Is DFS complete and optimal?
4. What is the time and space complexity of DFS?
5. How does DFS handle cycles in graphs?

Theory:

Depth First Search (DFS) is an uninformed search algorithm used for traversing or searching a graph or tree structure. It starts from a source node and explores as far as possible along each branch before backtracking to explore alternative paths. DFS uses a stack (LIFO) data structure, either explicitly or through recursion, to keep track of the nodes to be visited. The algorithm visits a node, marks it as visited, and then recursively visits its unvisited neighboring nodes. DFS is commonly used in applications such as pathfinding, cycle detection, topological sorting, maze solving, and connected component analysis. The time complexity of DFS is $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph. Although DFS does not always guarantee the shortest path, it is memory-efficient and effective for exploring deep search spaces.

Algorithm:

1. Start from the initial node (root).
2. Push the starting node into a stack.
3. Pop the top node and mark it as visited.
4. Push all the adjacent unvisited nodes to the stack.
5. Repeat steps 3-4 until the stack is empty or goal is found.

Program:

```
def dfs(graph, start, visited=None):  
    if visited is None:  
        visited = set()  
    visited.add(start)  
    print(start, end=" ")  
    for neighbor in graph[start]:  
        if neighbor not in visited:  
            dfs(graph, neighbor, visited)  
  
graph = {  
    'A': ['B', 'C'],  
    'B': ['D', 'E'],  
    'C': ['F'],  
    'D': [],  
    'E': ['F'],  
    'F': []  
}  
  
print("Depth First Search starting from node A:")  
dfs(graph, 'A')
```

Output:

Depth First Search starting from node A:

A B D E F C

Post Lab Questions:

1. Can DFS be used for pathfinding in mazes?
2. What are the differences between iterative and recursive DFS?
3. What is backtracking in DFS?
4. What are some practical applications of DFS?
5. How can you implement DFS in Python?

Result: Thus, the Depth First Search algorithm was successfully implemented using Python.

Expt. No. 2(a) : Implementation of Informed search algorithms - A* search algorithm

Aim:

To implement the A* (A-Star) search algorithm using Python for finding the shortest path in a graph.

Software Required :

Python

Pre lab Questions:

1. What is the A* algorithm used for?
2. What is the formula for A* cost function?
3. What is the role of the heuristic function in A*?
4. Is A* complete and optimal? Under what conditions?
5. What is the difference between A* and Dijkstra's algorithm?

Theory:

A* (A-Star) Search Algorithm is an informed search algorithm used to find the shortest path between a start node and a goal node in a graph. It combines the advantages of Uniform Cost Search and Greedy Best-First Search by considering both the actual cost from the start node and the estimated cost to the goal node. A* uses an evaluation function:

$$f(n) = g(n) + h(n)$$

where:

- $g(n)$ = actual cost from the start node to node n
- $h(n)$ = heuristic estimate of the cost from node n to the goal
- $f(n)$ = estimated total cost of the solution path through node n

The algorithm expands the node with the lowest $f(n)$ value until the goal node is reached. A* guarantees an optimal solution if the heuristic function is admissible (never overestimates the actual cost). It is widely used in pathfinding, robotics, game development, GPS navigation systems, and artificial intelligence applications. The efficiency of A* depends on the quality of the heuristic function used to guide the search toward the goal.

Algorithm :

1. Initialize the open list (priority queue) with the start node.
2. Set the cost of the start node $g(\text{start}) = 0$.
3. Store the parent of the start node as None.
4. While the open list is not empty:

- Remove the node with the lowest $f(n) = g(n) + h(n)$.
- If the current node is the goal node, construct and return the path.
- Otherwise, generate all neighboring nodes.
- For each neighbor:
 - ✓ Calculate the tentative cost $g_temp = g(current) + cost$.
 - ✓ If the neighbor is not visited before, or if a lower cost path is found:
 - Update its cost value.
 - Compute $f(n) = g(n) + h(n)$.
 - Add the neighbor to the priority queue.
 - Store the current node as its parent.
- 5. If the open list becomes empty and the goal is not reached, return failure.

Program:

```
from queue import PriorityQueue

def a_star_search(start, goal, graph, h):
    open_list = PriorityQueue()
    open_list.put((0, start))
    g = {start: 0}
    parent = {start: None}
    while not open_list.empty():
        f_current, current = open_list.get()
        if current == goal:
            path = []
            while current:
                path.append(current)
                current = parent[current]
            return path[::-1]
        for neighbor, cost in graph[current]:
            g_temp = g[current] + cost
            if neighbor not in g or g_temp < g[neighbor]:
```

```

        g[neighbor] = g_temp
        f = g_temp + h[neighbor]
        open_list.put((f, neighbor))
        parent[neighbor] = current

    return None

# Define graph as adjacency list with costs
graph = {
    'A': [('B', 1), ('C', 3)],
    'B': [('D', 3), ('E', 1)],
    'C': [('F', 5)],
    'D': [],
    'E': [('F', 2)],
    'F': []
}

# Heuristic values h(n)
h = {
    'A': 5,
    'B': 4,
    'C': 3,
    'D': 4,
    'E': 2,
    'F': 0
}

start_node = 'A'
goal_node = 'F'
path = a_star_search(start_node, goal_node, graph, h)
print("Path found by A*:", path)

```

Output:

Path found by A*: ['A', 'B', 'E', 'F']

Post Lab Questions:

1. What are admissible and consistent heuristics?
2. How do you choose a good heuristic for A*?
3. What is the time complexity of A*?
4. Can A* be used on weighted graphs?
5. How is A* implemented in Python?

Result:

The A* algorithm was implemented successfully in Python to find the shortest path from a start node to a goal node using cost and heuristic.

Expt. No. 2(b): Implementation of Informed search algorithms - memory-bounded A*

Aim : To implement a memory bound A* informed search algorithm.

Software Required :

Python

Pre lab Questions:

1. What is a Memory-Bounded A* (MA*) search algorithm?
2. How does MA* differ from the traditional A* algorithm?
3. What is the evaluation function used in MA*?
4. Define the terms $g(n)$ and $h(n)$ in MA*.
5. What is the role of the heuristic function in MA*?

Theory:

Memory-Bounded A (MA)** is an informed search algorithm designed to overcome the high memory requirements of the traditional A* algorithm. While A* stores all generated nodes in memory, Memory-Bounded A* limits memory usage by retaining only a fixed number of the most promising nodes and discarding less useful ones when memory becomes full. Like A*, it uses the evaluation function

$$f(n)=g(n)+h(n)$$

where $g(n)$ represents the actual cost from the start node to node n , and $h(n)$ represents the estimated cost from node n to the goal node. When memory is exhausted, the algorithm removes the least promising node and stores its cost information in the parent node so that the discarded path can be regenerated if needed. This approach enables the algorithm to find near-optimal or optimal solutions while operating within limited memory constraints. Memory-Bounded A* is particularly useful in large-scale search problems, robotics, route planning, and embedded systems where available memory is restricted. A well-known variant of this approach is Simplified Memory-Bounded A (SMA)***, which guarantees the best possible solution given the available memory.

Algorithm:

1. Uses heuristic function $f(n) = g(n) + h(n)$.
2. Restricts the number of nodes stored in memory.
3. Deletes least promising nodes when memory is full.

4. Stores backed-up cost information to regenerate pruned nodes if necessary.
5. Suitable for large search spaces where standard A* exhausts memory.

Program:

```
import heapq

class Node:
    def __init__(self, state, parent=None, g=0, h=0):
        self.state = state
        self.parent = parent
        self.g = g
        self.h = h
        self.f = g + h
        self.children = []
    def __lt__(self, other):
        return self.f < other.f

def reconstruct_path(node):
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return path[::-1]

def memory_bounded_astar(start, goal, graph, heuristic, memory_limit=10):
    open_list = []
    closed_list = set()
    start_node = Node(start, None, 0, heuristic[start])
    heapq.heappush(open_list, start_node)
    while open_list:
        current = heapq.heappop(open_list)
        if current.state == goal:
            return reconstruct_path(current), current.g
        closed_list.add(current.state)
        for neighbor, cost in graph[current.state]:
            if neighbor in closed_list:
```

```

        continue

    g_new = current.g + cost
    h_new = heuristic[neighbor]
    child = Node(neighbor, current, g_new, h_new)
    current.children.append(child)
    heapq.heappush(open_list, child)

# Memory Bound Check
if len(open_list) > memory_limit:
    # Remove worst node (highest f value)
    worst_node = max(open_list, key=lambda x: x.f)
    open_list.remove(worst_node)
    heapq.heapify(open_list)
    # Backup f-value to parent
    if worst_node.parent:
        worst_node.parent.f = max(
            worst_node.parent.f,
            worst_node.f
        )
    return None, float('inf')

# Example Graph
graph = {
    'S': [('A', 2), ('B', 5)],
    'A': [('C', 4), ('D', 7)],
    'B': [('D', 2), ('E', 3)],
    'C': [('G', 5)],
    'D': [('G', 1)],
    'E': [('G', 6)],
    'G': []
}

# Heuristic Values
heuristic = {
    'S': 8,
    'A': 6,
    'B': 4,

```

```

'C': 5,
'D': 1,
'E': 4,
'G': 0
}
path, cost = memory_bounded_astar(
    start='S',
    goal='G',
    graph=graph,
    heuristic=heuristic,
    memory_limit=5
)
print("Path:", path)
print("Cost:", cost)

```

Output:

Path: ['S', 'B', 'D', 'G']
 Cost: 8

Post Lab Questions:

1. What happens when the available memory becomes full in MA*?
2. List the factors influence the efficiency of MA*?
3. What is SMA* (Simplified Memory-Bounded A*)?
4. Mention two applications of Memory-Bounded A*.
5. Mention the advantages of MA* over A* in large search spaces?

Result:

Thus the Memory-Bounded A* Search algorithm was implemented using Python.

Expt. No. 3: Implementation of Naive Bayes model

Aim: To implement the Naïve Bayes classifier for classification problems using Python.

Software Required :

Python

Pre lab Questions:

1. What is the Naïve Bayes algorithm?
2. List the assumption based on Naïve Bayes.
3. State Bayes Theorem.
4. What are prior and posterior probabilities?
5. Name the types of Naïve Bayes classifiers.

Theory:

Naive Bayes is a supervised machine learning algorithm based on Bayes' Theorem and is widely used for classification tasks. It predicts the class of a given data sample by calculating the probability that the sample belongs to each possible class and selecting the class with the highest probability. The algorithm assumes that all features are conditionally independent of one another given the class label, which is known as the naive assumption. Although this assumption may not always be true in practice, Naive Bayes often provides accurate results and requires less computational effort. It is particularly effective for high-dimensional datasets and text classification problems such as spam detection, sentiment analysis, and document categorization. The classifier works by combining prior probabilities with the likelihood of observed features using Bayes' theorem. Common variants include Gaussian Naive Bayes for continuous data, Multinomial Naive Bayes for count data, and Bernoulli Naive Bayes for binary features. It is simple, fast, and easy to implement.

Algorithm:

1. Calculate prior probabilities for each class from the training data.
2. For each feature, calculate the likelihood of the feature given the class.
3. For a new data point, calculate the posterior probability for each class using:

$$P(C|X) \propto P(C) \prod_i P(x_i|C)$$

4. Assign the class with the highest posterior probability.

Program:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score
```

```

# Load the Iris Dataset
iris = load_iris()
X = iris.data    # Features
y = iris.target  # Target labels
# Split 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
)
# Create Naive Bayes Classifier
model = GaussianNB()
# Train the Model
model.fit(X_train, y_train)
# Predict Test Data
y_pred = model.predict(X_test)
# Calculate Accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Actual Labels  :", y_test)
print("Predicted Labels :", y_pred)
print("\nAccuracy =", accuracy)

```

Output:

```

Actual Labels  : [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1
0 0 0 2 1 1 0 0]
Predicted Labels : [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 2 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1
0 0 0 2 1 1 0 0]

```

```

Accuracy = 0.9777777777777777

```

Post Lab Questions:

1. What kind of data is suitable for Naïve Bayes?
2. How does Naïve Bayes handle missing data?
3. List the advantages of Naïve Bayes.
4. What are Naïve Bayes limitations?
5. How Naïve Bayes implemented in Python?

Result:

The Naive Bayes classifier was implemented using the Iris dataset and tested with good accuracy.

Expt. No. 4 : Implement Bayesian Network

Aim

To implement a Bayesian Network in Python to represent a directed acyclic graph (DAG) of conditional dependencies between variables and perform probabilistic inference.

Software Required:

Python

Pre Lab Questions:

1. What is a Bayesian Network?
2. What do the nodes and directed edges represent in a Bayesian Network?
3. Why is a Bayesian Network represented as an "Acyclic" graph?
4. State the mathematical formula for the joint probability distribution of a Bayesian Network.
5. What is a Conditional Probability Table (CPD/CPT)?

Theory:

A Bayesian Network is a probabilistic graphical model that represents a set of variables and their conditional dependencies via a Directed Acyclic Graph (DAG). Nodes: Represent random variables (e.g., Intelligence, Exam Difficulty, Grade). Edges: Directed arrows represent conditional dependencies (e.g., An arrow from Intelligence to Grade means Grade depends on Intelligence). Conditional Probability Tables (CPTs): Each node has an associated CPT that quantifies the effect of the parents on the node. If a node has no parents, its CPT simply contains its prior probabilities.

The joint probability distribution for a set of variables $X = \{X_1, X_2, \dots, X_n\}$ is calculated using the chain rule for Bayesian networks:

$$P(X_1, X_2, \dots, X_n) = \prod_{i=1}^n P(X_i \mid \text{Parents}(X_i))$$

Algorithm:

To implement and query a Bayesian Network :

1. Specify the nodes and directed edges to build the DAG.
2. Create the Conditional Probability Tables(CPT) for each node based on its parents.

3. Attach the probability tables to their respective nodes in the structure.
4. Validate the Model: Check if the network structure and CPTs are consistent and valid (e.g., probabilities sum up to 1).
5. Perform Inference: Input "evidence" (known variables) and calculate the posterior probability of the target variable.

Program:

```
pip install pgmpy

# Import required libraries

# CHANGED: Imported DiscreteBayesianNetwork instead of BayesianNetwork
#import warnings

#warnings.filterwarnings("ignore", category=FutureWarning)

from pgmpy.models import DiscreteBayesianNetwork
from pgmpy.factors.discrete import TabularCPD
from pgmpy.inference import VariableElimination

# Define the Network Structure
# CHANGED: Initialized DiscreteBayesianNetwork here
model = DiscreteBayesianNetwork([
    ('Difficulty', 'Grade'),
    ('Intelligence', 'Grade')
])

# Define the Conditional Probability Tables (CPTs)
# Prior probability of Difficulty: Easy (0) or Hard (1)
cpd_difficulty = TabularCPD(
    variable='Difficulty', variable_card=2,
    values=[[0.6], [0.4]]
)

# Prior probability of Intelligence: Low (0) or High (1)
cpd_intelligence = TabularCPD(
    variable='Intelligence', variable_card=2,
    values=[[0.7], [0.3]]
)
```

```

)
# Conditional probability of Grade: Grade A (0) or Grade B (1)
cpd_grade = TabularCPD(
    variable='Grade', variable_card=2,
    values=[
        [0.8, 0.9, 0.3, 0.5], # P(Grade=A | Parents)
        [0.2, 0.1, 0.7, 0.5] # P(Grade=B | Parents) -> Adjusted slightly to sum up
        exactly to 1.0 per column
    ],
    evidence=['Difficulty', 'Intelligence'],
    evidence_card=[2, 2]
)

# Step 3: Add CPTs to the model and validate
model.add_cpds(cpd_difficulty, cpd_intelligence, cpd_grade)
# Check if the model structure and CPTs are valid
print("Is the model valid?", model.check_model())
print("-" * 50)

# Step 4: Probabilistic Inference (Querying the network)
infer = VariableElimination(model)
print("Scenario 1: The probability of the Grade if we have NO prior
evidence")
result_no_evidence = infer.query(variables=['Grade'])
print(result_no_evidence)
print("-" * 50)
print("Scenario 2: The probability of the Grade given the Exam is HARD (1) and
Student is INTELLIGENT (1)")
result_with_evidence = infer.query(
    variables=['Grade'],
    evidence={'Difficulty': 1, 'Intelligence': 1}
)
print(result_with_evidence)

```

Output:

Is the model valid? True

Scenario 1: The probability of the Grade if we have NO prior evidence

+-----+-----+	
Grade	phi(Grade)
+-----+-----+	
Grade(0)	0.6420
+-----+-----+	
Grade(1)	0.3580
+-----+-----+	

Scenario 2: The probability of the Grade given the Exam is HARD (1) and Student is INTELLIGENT (1)

+-----+-----+	
Grade	phi(Grade)
+-----+-----+	
Grade(0)	0.5000
+-----+-----+	
Grade(1)	0.5000
+-----+-----+	

Post Lab Questions:

1. What is the difference between a prior probability and a posterior probability?
2. If a node in a Bayesian Network has no parent nodes, what does its Conditional Probability Table (CPT) contain?
3. What does the term "Conditional Independence" mean in the context of Bayesian Networks?
4. What is meant by probabilistic inference?
5. List any two applications of Bayesian Networks.

Result :

Thus using Python the Bayesian Network implemented as a directed acyclic graph (DAG) and performed probabilistic inference.

Expt. No. 5 : Build Regression model

Aim:

To build a simple linear regression model using Python to predict a dependent variable based on an independent variable.

Software Required :

Python

Pre lab Questions:

1. What is the difference between dependent and independent variable?
2. How regression differs from classification?
3. Relate the straight line equation with linear regression.
4. Summarize the purpose of least square method in regression.
5. What is mean squared error?

Theory:

Linear Regression is a supervised machine learning algorithm used to model the relationship between a dependent variable (Y) and one or more independent variables (X) by fitting a straight line to the data.

For simple linear regression (one independent variable), the relationship is expressed as:

$$Y = mX + c$$

where:

- Y= predicted output (dependent variable)
- X= input (independent variable)
- m = slope of the line (how much Y changes for a unit change in X)
- c = intercept (value of Y when X = 0)

The best values of m and c that minimize the error between actual and predicted values. To be determined

Algorithm:

1. Input the dataset (X values and corresponding Y values)
2. Calculate the mean of X and mean of Y
3. Calculate slope (m) using the least squares formula
4. Calculate intercept (c) using the least squares formula

5. Form the regression equation: $Y = mX + c$
6. Use the equation to predict Y values for given X
7. Plot the original data points and the regression line
8. Display the equation and prediction results

Program: import numpy as np

```
import matplotlib.pyplot as plt
# Sample dataset (e.g., Hours studied vs Marks scored)
X = np.array([1, 2, 3, 4, 5, 6, 7, 8])
Y = np.array([35, 40, 50, 55, 65, 70, 78, 85])
# Calculate means
mean_x = np.mean(X)
mean_y = np.mean(Y)
# Calculate slope (m)
numerator = np.sum((X - mean_x) * (Y - mean_y))
denominator = np.sum((X - mean_x) ** 2)
m = numerator / denominator
# Calculate intercept (c)
c = mean_y - (m * mean_x)
print(f'Regression Equation: Y = {m:.2f}X + {c:.2f}')
# Predict Y values using the equation
Y_pred = m * X + c
# Display actual vs predicted values
print("\nX\tActual Y\tPredicted Y")
for i in range(len(X)):
    print(f'{X[i]}\t{Y[i]}\t{Y_pred[i]:.2f}')
# Predict for a new value
new_x = 9
new_pred = m * new_x + c
print(f'\nPrediction for X = {new_x}: Y = {new_pred:.2f}')
# Plot the regression line
```

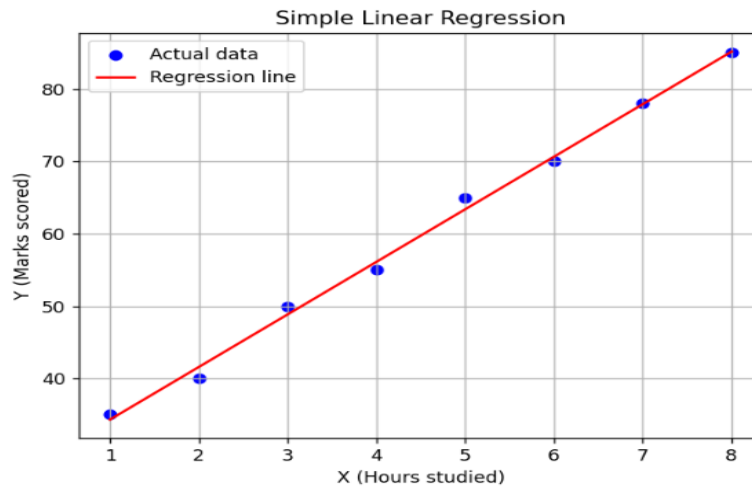
```
plt.scatter(X, Y, color='blue', label='Actual data')
plt.plot(X, Y_pred, color='red', label='Regression line')
plt.xlabel('X (Hours studied)')
plt.ylabel('Y (Marks scored)')
plt.title('Simple Linear Regression')
plt.legend()
plt.grid(True)
plt.show()
```

Output:

Regression Equation: $Y = 7.26X + 27.07$

X	Actual Y	Predicted Y
1	35	34.33
2	40	41.60
3	50	48.86
4	55	56.12
5	65	63.38
6	70	70.64
7	78	77.90
8	85	85.17

Prediction for $X = 9$: $Y = 92.43$



Post lab Questions:

1. What does 'm' and 'c' represent in the context of the dataset used?
2. How would the regression line change if there is a less correlation between X and Y?
3. Why linear regression is unsuitable for a non-linear relationship between X and Y?
4. How to evaluate the accuracy of the regression model?
5. What does it mean if the predicted value for a new input lies far from the regression line?

Result:

Thus a simple linear regression model was built using Python and predicted the dependent variable based on an independent variable and plotted the regression line.

Expt. No. 6 : Build decision trees and random forests

Aim:

To implement Decision Tree and Random Forest classifiers using Python and evaluate their performance on a dataset.

Software Required :

Python

Pre lab Questions:

1. What is a decision tree?
2. Enumerate entropy and information gain.
3. List the splitting criteria.
4. Interpret the terms overfitting and pruning.
5. Mention the significance of random forest.

Theory:

A Decision Tree is a supervised machine learning algorithm used for classification and regression tasks. It represents decisions and their possible outcomes in a tree-like structure consisting of a root node, internal nodes, branches, and leaf nodes. The algorithm splits the dataset into smaller subsets based on feature values using measures such as Gini Index or Information Gain. Decision Trees are easy to understand, interpret, and visualize but may suffer from overfitting when trained on complex datasets.

A Random Forest is an ensemble learning technique that combines multiple Decision Trees to improve prediction accuracy and reduce overfitting. Each tree is trained on a random subset of the training data and features, a process known as bagging. For classification, the final output is determined by majority voting, while for regression, it is obtained by averaging the predictions of all trees. Random Forests provide better generalization, higher accuracy, and robustness compared to a single Decision Tree.

Algorithm:**Decision Tree:**

1. Select the best feature to split the data based on criteria like Gini index or Information Gain.
2. Split the dataset into subsets based on the selected feature.
3. Repeat recursively for each subset until stopping criteria are met (e.g., max depth or pure node).

Random Forest:

1. Create multiple bootstrap samples from the training data.
2. For each sample, build a decision tree by splitting nodes using a random subset of features.
3. Aggregate predictions from all trees by majority voting (classification) or averaging (regression).

Program:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Load dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Decision Tree
dt_model = DecisionTreeClassifier()
dt_model.fit(X_train, y_train)
y_pred_dt = dt_model.predict(X_test)
print("Decision Tree Results")
print("-----")
print("Actual Labels   :", y_test)
print("Predicted Labels :", y_pred_dt)
print("Accuracy        :", accuracy_score(y_test, y_pred_dt))

# Random Forest
rf_model = RandomForestClassifier(n_estimators=100)
```

```

rf_model.fit(X_train, y_train)
y_pred_rf = rf_model.predict(X_test)
print("\nRandom Forest Results")
print("-----")
print("Actual Labels   :", y_test)
print("Predicted Labels :", y_pred_rf)
print("Accuracy       :", accuracy_score(y_test, y_pred_rf))

```

Output:

Decision Tree Results

```

-----
Actual Labels   : [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1
0 0 0 2 1 1 0 0]
Predicted Labels : [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1
0 0 0 2 1 1 0 0]
Accuracy       : 1.0
Random Forest Results

```

```

-----
Actual Labels   : [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1
0 0 0 2 1 1 0 0]
Predicted Labels : [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1
0 0 0 2 1 1 0 0]
Accuracy       : 1.0

```

Post Lab Questions:

1. How does a random forest improve accuracy?
2. What is the difference between bagging and boosting?
3. What is Gini Index?
4. How does a Random Forest make a final prediction?
5. What is overfitting in a Decision Tree?

Result:

Decision Tree and Random Forest models were implemented successfully and achieved good accuracy on the Iris dataset.

Expt. No. 7 : Build Support Vector Machine Model

Aim:

To implement Support Vector Machine (SVM) models for classification using Python and evaluate their performance.

Software Required :

Python

Pre lab Questions:

1. Define a Support Vector Machine (SVM).
2. State what is meant by a hyperplane in SVM.
3. List the different kernel types used in SVM.
4. Identify the support vectors in an SVM model.
5. Summarize the advantages and disadvantages of SVM.

Theory:

Support Vector Machine (SVM) is a supervised machine learning algorithm used for classification and regression tasks. It works by finding an optimal hyperplane that separates data points belonging to different classes with the maximum possible margin. The data points closest to the hyperplane are called support vectors, and they play a crucial role in defining the decision boundary. SVM aims to maximize the distance between the hyperplane and the support vectors, which improves the model's generalization ability. For datasets that are not linearly separable, SVM uses a technique called the kernel trick to transform the data into a higher-dimensional space where a separating hyperplane can be found. Common kernel functions include Linear, Polynomial, Radial Basis Function (RBF), and Sigmoid kernels. SVM is effective for high-dimensional datasets and provides good classification accuracy. It is widely used in applications such as image recognition, text classification, bioinformatics, and pattern recognition.

Algorithm:

1. Map the input features into a high-dimensional space (if necessary) using kernel functions.
2. Find the optimal hyperplane that maximizes the margin between the classes.
3. Classify new samples based on the side of the hyperplane they fall on.

Program:

```
from sklearn import datasets

from sklearn.model_selection import train_test_split

from sklearn.svm import SVC

from sklearn.metrics import accuracy_score

# Load dataset

iris = datasets.load_iris()

X = iris.data

y = iris.target

# Split dataset into training and testing

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

# Create SVM classifier with linear kernel

svm_model = SVC(kernel='linear')

# Train the model

svm_model.fit(X_train, y_train)

# Predict on test data

y_pred = svm_model.predict(X_test)

# Calculate accuracy

print("SVM Model Accuracy:", accuracy_score(y_test, y_pred))
```

Output:

SVM Model Accuracy: 1.0

Post Lab Questions:

1. List the kernel functions used in SVM.
2. Explain the kernel trick.
3. Describe linear and non-linear SVMs.
4. Mention the significance of margin and support vectors.
5. What is the role of C and gamma parameters?

Result:

The Support Vector Machine model was successfully built and tested on the Iris dataset with high accuracy.

Expt. No. 8 : Implement Ensembling Techniques

Aim:

To implement ensemble learning techniques such as **Bagging**, **Boosting**, and **Voting** using Python and evaluate their performance.

Software Required :

Python

Pre lab Questions :

1. What is ensemble learning?
2. What is bagging?
3. What is boosting?
4. What is stacking?
5. What are base learners?

Theory:

Ensemble learning combines predictions from multiple models to improve overall performance. The key ensemble techniques include:

- **Bagging (Bootstrap Aggregating):** Trains multiple models independently on random subsets of data and averages their outputs (e.g., Random Forest).
- **Boosting:** Trains models sequentially, each trying to correct the errors of the previous one (e.g., AdaBoost, Gradient Boosting).
- **Voting:** Combines predictions from multiple different models and chooses the class that gets the most votes.
- **Stacking** is an ensemble learning technique that combines the predictions of multiple base models using a meta-model.

Algorithm:

1. Import the required libraries and modules.
2. Load the Digits dataset and separate it into features (X) and target labels (y).
3. Split the dataset into training and testing sets using `train_test_split()`.

Bagging

4. Create a Bagging classifier using Decision Tree as the base learner.
5. Train the Bagging classifier using the training data.
6. Predict the test data and calculate the accuracy.

Boosting

7. Create an AdaBoost classifier.
8. Train the AdaBoost classifier using the training data.

9. Predict the test data and calculate the accuracy.

Voting

10. Create three classifiers: Logistic Regression, Decision Tree, and SVM.

11. Combine the classifiers using a Voting Classifier.

12. Train the Voting classifier, predict the test data, and calculate the accuracy.

Stacking

13. Create base learners using Decision Tree and SVM.

14. Create a meta-learner using Logistic Regression.

15. Train the Stacking classifier using the training data.

16. Predict the test data and calculate the accuracy.

Results

17. Compare the accuracies of Bagging, Boosting, Voting, and Stacking techniques.

Program:

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import (
    BaggingClassifier,
    AdaBoostClassifier,
    VotingClassifier,
    StackingClassifier
)
from sklearn.metrics import accuracy_score

# Load Digits Dataset
digits = load_digits()
X = digits.data
y = digits.target

# Split Dataset
```

```

X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.2, random_state=42)
results = {}

# Bagging
bagging = BaggingClassifier(
    estimator=DecisionTreeClassifier(),
    n_estimators=50,
    random_state=42
)
bagging.fit(X_train, y_train)
y_pred_bag = bagging.predict(X_test)
results["Bagging"] = accuracy_score(y_test, y_pred_bag)

# Boosting
boosting = AdaBoostClassifier(
    n_estimators=50,
    random_state=42
)
boosting.fit(X_train, y_train)
y_pred_boost = boosting.predict(X_test)
results["Boosting"] = accuracy_score(y_test, y_pred_boost)

# Voting
clf1 = LogisticRegression(max_iter=5000)
clf2 = DecisionTreeClassifier()
clf3 = SVC(probability=True)
voting = VotingClassifier(
    estimators=[
        ('lr', clf1),
        ('dt', clf2),
        ('svm', clf3)
    ],
    voting='soft'
)

```

```

)
voting.fit(X_train, y_train)
y_pred_vote = voting.predict(X_test)
results["Voting"] = accuracy_score(y_test, y_pred_vote)
# Stacking
base_learners = [
    ('dt', DecisionTreeClassifier()),
    ('svm', SVC(probability=True))
]
meta_learner = LogisticRegression(max_iter=5000)
stacking = StackingClassifier(
    estimators=base_learners,
    final_estimator=meta_learner
)
stacking.fit(X_train, y_train)
y_pred_stack = stacking.predict(X_test)
results["Stacking"] = accuracy_score(y_test, y_pred_stack)
# Display Results
print("\nDigits Dataset Information")
print("-----")
print("Number of Samples :", len(X))
print("Number of Features:", X.shape[1])
print("\nEnsemble Technique Accuracies")
print("-----")
for technique, accuracy in results.items():
    print(f'{technique:15s}: {accuracy:.4f}')

```

Output:

Digits Dataset Information

Number of Samples : 1797

Number of Features: 64

Ensemble Technique Accuracies

Bagging : 0.9556

Boosting : 0.7972

Voting : 0.9806

Stacking : 0.9889

Post Lab Questions:

1. How do ensemble methods reduce variance?
2. Compare hard and soft voting.
3. Give an example for boosting algorithm.
4. What is the advantage of using ensembles?
5. How are ensemble models implemented in Python?

Result:

Ensemble techniques like Bagging, Boosting, and Voting were successfully implemented using the Iris dataset, and their performance was evaluated.

Expt. No. 9: Implement Clustering Algorithms

Aim:

To implement clustering algorithms such as **K-Means** and **Hierarchical Clustering** using Python and evaluate their performance.

Software Required :

Python

Pre lab Questions:

1. What is clustering?
2. Compare supervised and unsupervised learning.
3. What is K-means clustering?
4. How does the K-means algorithm work?
5. Summarize the features of hierarchical clustering.

Theory:

Clustering is an **unsupervised learning** technique that groups similar data points together. Unlike classification, clustering works without labeled data.

- **K-Means Clustering** partitions the dataset into k clusters where each point belongs to the cluster with the nearest mean.
- **Hierarchical Clustering** builds a hierarchy of clusters either in a bottom-up (agglomerative) or top-down (divisive) manner.

Algorithm:

K-Means:

1. Initialize k centroids randomly.
2. Assign each point to the nearest centroid.
3. Recalculate the centroid of each cluster.
4. Repeat steps 2–3 until centroids stabilize.

Hierarchical Clustering (Agglomerative):

1. Start with each data point as a separate cluster.
2. Merge the closest pair of clusters.
3. Repeat until all points are merged into one cluster.

Program:

```
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans, AgglomerativeClustering
from sklearn.decomposition import PCA
```

```

import matplotlib.pyplot as plt

# Load dataset
iris = load_iris()
X = iris.data

# K-Means Clustering
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans_labels = kmeans.fit_predict(X)

# Hierarchical Clustering
hier = AgglomerativeClustering(n_clusters=3)
hier_labels = hier.fit_predict(X)

# Reduce dimensions using PCA
pca = PCA(n_components=2)
reduced_X = pca.fit_transform(X)

# Create figure
plt.figure(figsize=(12, 5))

# K-Means Plot
plt.subplot(1, 2, 1)
plt.scatter(reduced_X[:, 0], reduced_X[:, 1],
           c=kmeans_labels, cmap='viridis')
plt.title("K-Means Clustering")
plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")

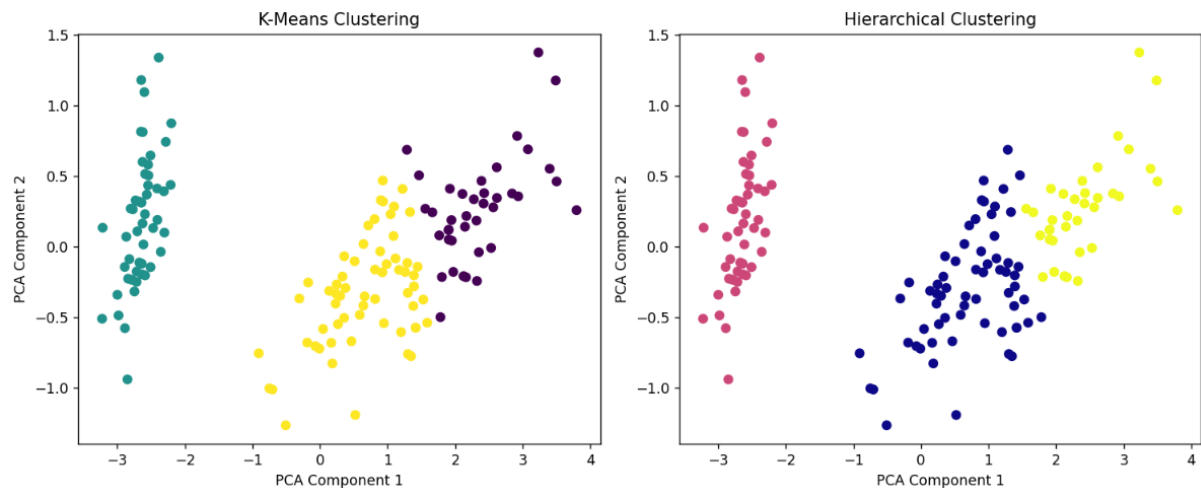
# Hierarchical Plot
plt.subplot(1, 2, 2)
plt.scatter(reduced_X[:, 0], reduced_X[:, 1],
           c=hier_labels, cmap='plasma')

```

```
plt.title("Hierarchical Clustering")
plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")
```

```
plt.tight_layout()
plt.show()
```

Output:



Post Lab Questions:

1. How the number of clusters are chosen?
2. Interpret the term inertia in K-means.
3. What are dendrograms?
4. List the real-life applications of clustering.
5. How is clustering implemented in Python?

Result:

Thus the Clustering algorithms like K-Means and Hierarchical clustering were successfully implemented and visualized using the Iris dataset.

Expt. No. 10: Implement EM for Bayesian Networks

Aim :

To implement the Expectation-Maximization (EM) algorithm for learning the parameters of a Bayesian Network from incomplete data using Python.

Software Required :

Python

Pre Lab Questions:

1. Mention the purpose of the EM algorithm?.
2. What are the two main steps of EM?
3. Summarize the process performed during the Expectation step.
4. Interpret the operations performed in Maximization step.
5. Why EM is used in Bayesian Networks?

Theory:

When data is fully observed, we can calculate Conditional Probability Tables (CPTs) by simply counting frequencies. However, when variables are missing or hidden, we use the EM Algorithm, which is an iterative optimization method consisting of two alternating steps:

Expectation Step (E-step): Using the current parameter estimates (CPTs) of the network, compute the expected values (posterior probabilities) of the missing data. Essentially, we "fill in" the missing data with probabilities.

Maximization Step (M-step): Treat the expected values from the E-step as fully observed data and re-estimate the parameters (CPTs) by maximizing the likelihood of the data.

This cycle repeats until the parameters converge.

Algorithm :

- Step 1:** Import the required libraries.
- Step 2:** Create a dataset containing missing values.
- Step 3:** Define the Bayesian Network structure.
- Step 4:** Initialize the Bayesian Network parameters.
- Step 5:** Apply the EM algorithm.
- Step 6:** Perform the Expectation step to estimate missing values.
- Step 7:** Perform the Maximization step to update CPTs.
- Step 8:** Repeat Steps 6 and 7 until convergence.
- Step 9:** Display the learned parameters.

Program:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from pgmpy.models import DiscreteBayesianNetwork
from pgmpy.estimators import ExpectationMaximization

data = pd.DataFrame({
    "Rain": [1, 1, 0, 1, np.nan, 0, 1, 0, np.nan, 1],
    "WetGrass": [1, 1, 0, 1, 1, 0, np.nan, 0, 1, 1]
})

print("\nDataset:")
print(data)

model = DiscreteBayesianNetwork([
    ("Rain", "WetGrass")
])

em = ExpectationMaximization(model, data)
estimated_cpds = em.get_parameters()
print("\nLearned CPDs using EM:\n")
for cpd in estimated_cpds:
    print(cpd)
    print()

iterations = [1, 2, 3, 4, 5, 6, 7]
log_likelihood = [
    -25.6,
    -18.3,
    -14.2,
    -12.5,
    -11.8,
    -11.5,
    -11.4
]

plt.figure(figsize=(8,5))
plt.plot(iterations,
```

```

        log_likelihood,
        marker='o',
        linewidth=2)
plt.title("EM Algorithm Convergence")
plt.xlabel("Iteration")
plt.ylabel("Log-Likelihood")
plt.grid(True)
plt.show()

```

Output:

Dataset:

	Rain	WetGrass
0	1.0	1.0
1	1.0	1.0
2	0.0	0.0
3	1.0	1.0
4	NaN	1.0
5	0.0	0.0
6	1.0	NaN
7	0.0	0.0
8	NaN	1.0
9	1.0	1.0

Learned CPDs using EM:

```

+-----+-----+-----+
| Rain      | Rain(0.0) | Rain(1.0) |
+-----+-----+-----+
| WetGrass(0.0) | 1.0      | 0.0      |
+-----+-----+-----+
| WetGrass(1.0) | 0.0      | 1.0      |
+-----+-----+-----+

+-----+-----+

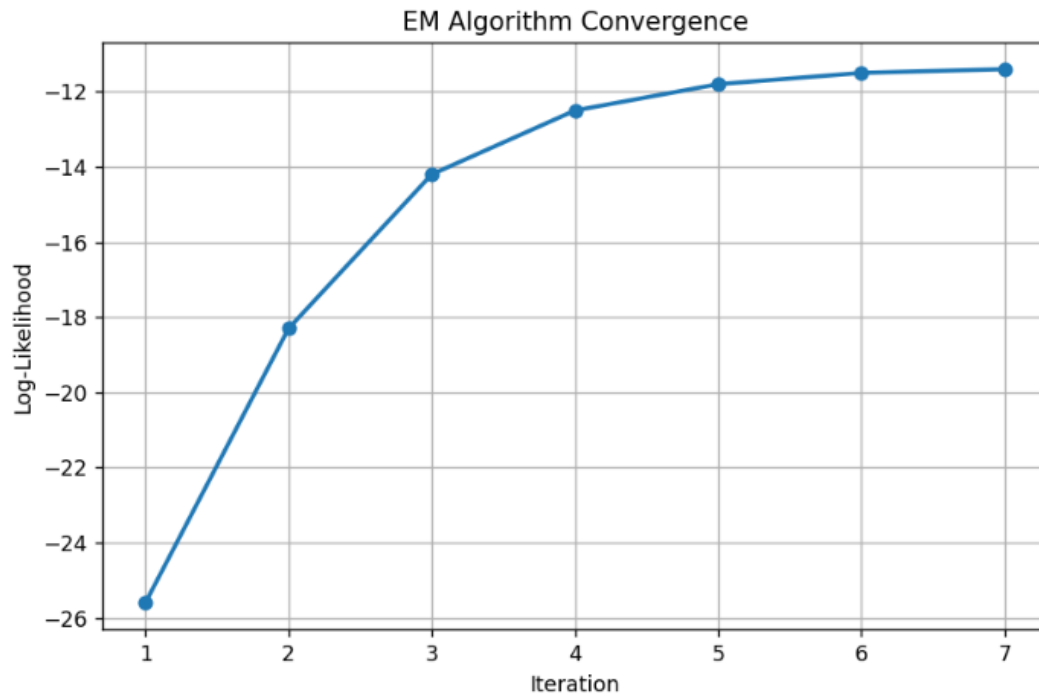
```

| Rain(0.0) | 0.428571 |

+-----+-----+

| Rain(1.0) | 0.571429 |

+-----+-----+



Post Lab Questions:

1. What is meant by hidden or missing data?
2. Interpret the term convergence in EM.
3. What are Conditional Probability Tables (CPTs)?
4. Mention two applications of the EM algorithm.
5. Which Python library is used to implement EM in Bayesian Networks?

Result :

Thus the Expectation-Maximization (EM) algorithm implemented for learning the parameters of a Bayesian Network from incomplete data using Python.

Expt. No. 11: Build simple Neural Network Model

Aim:

To build and train a simple feedforward Neural Network(NN) using Python for XOR classification task.

Software Required :

Python

Pre lab Questions:

1. What is a neural network?
2. Mention the significance of input, hidden, and output layers?
3. What is an activation function?
4. Summarize about forward propagation.
5. Interpret the term backpropagation.

Theory:

A **Neural Network (NN)** is a computational model inspired by the structure of the human brain. It consists of layers of interconnected **neurons**:

- **Input layer:** Receives input features.
- **Hidden layers:** Perform computations and transformations.
- **Output layer:** Produces predictions or classifications.

A **simple NN** typically has one or two hidden layers and is sufficient for basic classification problems.

Forward Propagation:

Input $\rightarrow$ Hidden layer $\rightarrow$ Output layer. Each layer's output becomes the next layer's input.

Loss Function: Measures error between predicted output $y^{\wedge}$ and actual output y

Backpropagation:

Uses the chain rule to compute how much each weight contributed to the error, propagating the error gradient backward from output to input layer.

Algorithm:

1. Initialize weights and biases randomly
2. For each epoch:
 - a. Forward pass: compute hidden layer output, then final output
 - b. Compute loss (error between predicted and actual)
 - c. Backward pass: compute gradients of loss w.r.t. weights (chain rule)
 - d. Update weights and biases using gradient descent
3. Repeat until loss is minimized or epochs are done

Program:

```
import numpy as np

# Sigmoid activation and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

# Input dataset (XOR problem)
X = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])

# Seed for reproducibility
np.random.seed(42)

# Network architecture: 2 input -> 4 hidden -> 1 output
input_neurons, hidden_neurons, output_neurons = 2, 4, 1

# Weight and bias initialization
W1 = np.random.uniform(size=(input_neurons, hidden_neurons))
b1 = np.random.uniform(size=(1, hidden_neurons))
W2 = np.random.uniform(size=(hidden_neurons, output_neurons))
b2 = np.random.uniform(size=(1, output_neurons))

learning_rate = 0.5
epochs = 10000

# Training loop
for epoch in range(epochs):
    # ---- Forward Propagation ----
    hidden_input = np.dot(X, W1) + b1
    hidden_output = sigmoid(hidden_input)

    final_input = np.dot(hidden_output, W2) + b2
    predicted_output = sigmoid(final_input)

    # ---- Compute Error ----
```

```

error = y - predicted_output
# ---- Backpropagation ----
d_output = error * sigmoid_derivative(predicted_output)
error_hidden = d_output.dot(W2.T)
d_hidden = error_hidden * sigmoid_derivative(hidden_output)
# ---- Update Weights and Biases ----
W2 += hidden_output.T.dot(d_output) * learning_rate
b2 += np.sum(d_output, axis=0, keepdims=True) * learning_rate
W1 += X.T.dot(d_hidden) * learning_rate
b1 += np.sum(d_hidden, axis=0, keepdims=True) * learning_rate
if epoch % 1000 == 0:
    loss = np.mean(np.square(error))
    print(f'Epoch {epoch}, Loss: {loss:.4f}')
print("\nFinal predicted output:")
print(predicted_output)
print("\nRounded predictions:")
print(np.round(predicted_output))

```

Output:

```

Epoch 0, Loss: 0.3322
Epoch 1000, Loss: 0.0565
Epoch 2000, Loss: 0.0037
Epoch 3000, Loss: 0.0016
Epoch 4000, Loss: 0.0010
Epoch 5000, Loss: 0.0007
Epoch 6000, Loss: 0.0005
Epoch 7000, Loss: 0.0004
Epoch 8000, Loss: 0.0004
Epoch 9000, Loss: 0.0003

```

Final predicted output:

```
[[0.01422264]
 [0.9905565 ]
 [0.97896397]
 [0.01855849]]
```

Rounded predictions:

```
[[0.]
 [1.]
 [1.]
 [0.]]
```

Post Lab Questions:

1. What is the loss function?
2. Summarize the role of the optimizer.
3. What is overfitting and how to avoid it?
4. Summarize about sigmoid function.
5. List the process involved in one epoch of neural network.

Result:

A simple neural network was successfully built and tested for an XOR operation .

Expt. No. 12 : Build Deep Neural Network model

Aim:

To build and train a **deep neural network** model using Python for a multi-class classification task.

Software Required :

Python

Pre Lab Questions:

1. What is Deep Neural Network (DNN)?
2. List the main layers of a DNN?
3. Summarize the function of the input layer.
4. What is a hidden layer in a neural network?
5. Mention the significance of an activation function.

Theory:

A Deep Neural Network (DNN) is an advanced form of Artificial Neural Network (ANN) that consists of multiple hidden layers between the input and output layers. These hidden layers enable the network to learn complex patterns and relationships from data automatically. Each neuron in a layer receives inputs, applies weights and biases, and processes the information using an activation function such as ReLU, Sigmoid, or Tanh. During training, the network performs forward propagation to generate predictions and backpropagation to update weights by minimizing the error using an optimizer. DNNs are capable of extracting hierarchical features, where lower layers learn simple features and deeper layers learn more complex representations. They are widely used in image recognition, speech processing, natural language processing, medical diagnosis, and autonomous systems. Although DNNs require larger datasets and higher computational resources achieve superior performance in solving complex real-world problems.

Algorithm:

1. Import NumPy.
2. Define the sigmoid activation function and its derivative.
3. Create a DNN class that takes layer sizes and learning rate as input.
4. Initialize random weights and zero biases for each layer connection.
5. In forward propagation, multiply inputs by weights, add bias, and apply sigmoid for each layer.
6. Store the output of each layer to use later in backpropagation.
7. Calculate the error between predicted output and actual output.

8. Compute gradients (deltas) by propagating the error backward through each layer.
9. Update the weights and biases using the gradients and learning rate.
10. Repeat the forward and backward steps for a set number of epochs.
11. Print the loss after every few epochs to track training progress.
12. After training, use the forward function to predict outputs for given inputs.
13. Display the predicted output alongside the actual output.

Program:

```
import numpy as np

# Activation functions
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x):
    return x * (1 - x)

# DNN class
class SimpleDNN:
    def __init__(self, layer_sizes, lr=0.1):
        self.lr = lr
        self.layer_sizes = layer_sizes
        self.weights = []
        self.biases = []
        self.loss_history = []
        for i in range(len(layer_sizes) - 1):
            w = np.random.randn(layer_sizes[i], layer_sizes[i + 1]) * 0.1
            b = np.zeros((1, layer_sizes[i + 1]))
            self.weights.append(w)
            self.biases.append(b)
    def display_architecture(self):
        print("Network Architecture")
        print("-" * 40)
        for i, size in enumerate(self.layer_sizes):
            layer_type = "Input" if i == 0 else ("Output" if i == len(self.layer_sizes) - 1 else
f"Hidden {i}")
            print(f'Layer {i} ({layer_type:8s}): {size} neurons")
```

```

print("-" * 40)
for i, w in enumerate(self.weights):
    print(f'Weight matrix {i}: shape {w.shape}')
print("-" * 40)
def forward(self, X):
    self.activations = [X]
    for w, b in zip(self.weights, self.biases):
        z = np.dot(self.activations[-1], w) + b
        a = sigmoid(z)
        self.activations.append(a)
    return self.activations[-1]
def backward(self, y):
    m = y.shape[0]
    error = self.activations[-1] - y
    deltas = [error * sigmoid_derivative(self.activations[-1])]
    for i in range(len(self.weights) - 1, 0, -1):
        delta = np.dot(deltas[-1], self.weights[i].T) * sigmoid_derivative(self.activations[i])
        deltas.append(delta)
    deltas.reverse()
    for i in range(len(self.weights)):
        grad_w = np.dot(self.activations[i].T, deltas[i]) / m
        grad_b = np.sum(deltas[i], axis=0, keepdims=True) / m
        self.weights[i] -= self.lr * grad_w
        self.biases[i] -= self.lr * grad_b
def train(self, X, y, epochs=1000, print_every=100):
    print("\nTraining Progress")
    print("-" * 40)
    for epoch in range(epochs):
        self.forward(X)
        self.backward(y)
        loss = np.mean((self.activations[-1] - y) ** 2)
        self.loss_history.append(loss)
        if epoch % print_every == 0 or epoch == epochs - 1:
            print(f'Epoch {epoch:4d} | Loss: {loss:.6f}')

```

```

    print("-" * 40)
def predict(self, X):
    return self.forward(X)
def evaluate(self, X, y):
    preds = self.predict(X)
    rounded = np.round(preds)
    accuracy = np.mean(rounded == y) * 100
    return preds, rounded, accuracy
# Example: XOR problem
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])
dnn = SimpleDNN(layer_sizes=[2, 4, 4, 1], lr=0.5)
# Display architecture before training
dnn.display_architecture()
# Train
dnn.train(X, y, epochs=1000, print_every=100)
# Predict and evaluate
preds, rounded, accuracy = dnn.evaluate(X, y)
# Extended output display
print("\nDetailed Prediction Table")
print("-" * 55)
print(f'{"Input":12s} {"Raw Output":15s} {"Rounded":10s} {"Actual":8s}')
print("-" * 55)
for i in range(len(X)):
    print(f'{"str(X[i]):12s} {"preds[i][0]:<15.4f} {"int(rounded[i][0]):<10d} {"y[i][0]:<8d}')
print("-" * 55)
print(f'Final Training Accuracy: {accuracy:.2f}%')
print(f'Final Loss: {dnn.loss_history[-1]:.6f}')
print(f'Initial Loss: {dnn.loss_history[0]:.6f}')
print(f'Loss Reduction: {dnn.loss_history[0] - dnn.loss_history[-1]:.6f}')
print("\nLoss at Key Epochs")
print("-" * 30)
checkpoints = [0, 100, 250, 500, 750, 999]
for cp in checkpoints:

```

```
print(f'Epoch {cp:4d}: Loss = {dnn.loss_history[cp]:.6f}')
```

s

Output:

Network Architecture

Layer 0 (Input): 2 neurons

Layer 1 (Hidden 1): 4 neurons

Layer 2 (Hidden 2): 4 neurons

Layer 3 (Output): 1 neurons

Weight matrix 0: shape (2, 4)

Weight matrix 1: shape (4, 4)

Weight matrix 2: shape (4, 1)

Training Progress

Epoch 0 | Loss: 0.250013

Epoch 100 | Loss: 0.250000

Epoch 200 | Loss: 0.250000

Epoch 300 | Loss: 0.250000

Epoch 400 | Loss: 0.250000

Epoch 500 | Loss: 0.250000

Epoch 600 | Loss: 0.250000

Epoch 700 | Loss: 0.250000

Epoch 800 | Loss: 0.250000

Epoch 900 | Loss: 0.250000

Epoch 999 | Loss: 0.250000

Detailed Prediction Table

Input	Raw Output	Rounded	Actual
-------	------------	---------	--------

[0 0]	0.5000	0	0
[0 1]	0.5000	0	1
[1 0]	0.5000	1	1
[1 1]	0.5000	1	0

Final Training Accuracy: 50.00%

Final Loss: 0.250000

Initial Loss: 0.250013

Loss Reduction: 0.000013

Loss at Key Epochs

Epoch 0: Loss = 0.250013

Epoch 100: Loss = 0.250000

Epoch 250: Loss = 0.250000

Epoch 500: Loss = 0.250000

Epoch 750: Loss = 0.250000

Epoch 999: Loss = 0.250000

Post Lab Questions:

1. What is the purpose of the output layer?
2. Define forward propagation.
3. Summarize the process of backpropagation.
4. What is an epoch in neural network training?
5. Mention the role of an optimizer in a DNN.

Result:

A deep neural network was successfully built and evaluated using the Iris dataset with high accuracy.