UNIVERSITY COLLEGE OF ENGINEERING NAGERCOIL

(ANNA UNIVERSITY CONSTITUENT COLLEGE)

KONAM, NAGERCOIL - 629004

RECORD NOTE BOOK

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING -CS3491

Register No :

Name :

Year/Semester :

Department :

UNIVERSITY COLLEGE OF ENGINEERING NAGERCOIL

(ANNA UNIVERSITY CONSTITUENT COLLEGE)

KONAM, NAGERCOIL - 629004

Register No:

Certified that, this is the bonafide record of work done by Mr/Ms. ……………………………………………………………… of IV Semester in Computer Science and Engineering of this college, in the Artificial Intelligence And Machine Learning (CS3491) during academic year 2024-2025 in partial fulfillment of the requirements of the B.E Degree course of the Anna University Chennai.

Staff-in-charge **Head of the Department**

This record is submitted for the University Practical Examination held on …………………………

Internal Examiner **External Examiner**

LIST OF EXPERIMENTS

S.No	Date	Name of the Experiments	Page Number	Sign
1a		IMPLEMENTATION OF UNINFORMED SEARCH ALGORITHMS-BFS		
1b		IMPLEMENTATION OF UNINFORMED SEARCH ALGORITHMS-UCS		
1c		IMPLEMENTATION OF UNINFORMED SEARCH ALGORITHMS-DLS		
1d		IMPLEMENTATION OF UNINFORMED SEARCH ALGORITHMS-IDS		
1e		IMPLEMENTATION OF UNINFORMED SEARCH ALGORITHMS-DFS		
2a		IMPLEMENTATION OF INFORMED SEARCH ALGORITHMS-GREEDY BEST FIRST SEARCH		
2b		IMPLEMENTATION OF INFORMED SEARCH ALGORITHMS-A* SEARCH		
2c		IMPLEMENTATION OF INFORMED SEARCH ALGORITHMS-MEMORY BOUNDED A* SEARCH		
3		IMPLEMENT NAVIE BAYES MODELS		
4		IMPLEMENT BAYESIAN NETWORKS		
5a		BUILD REGRESSION MODELS-LINEAR REGRESSION COEFFICIENTS		
5b		BUILD REGRESSION MODELS-LOGISTIC REGRESSION MODEL		
6a		BUILD DECISION TREE		
6b		BUILD RANDOM FORESTS		
7		BUILD SVM MODELS		
8		IMPLEMENT ENSEMBLING TECNIQUES		
9		IMPLEMENT CLUSTERING ALGORITHMS		
10		BUILD DEEPLEARNING NN MODELS		

Exp: 1a Date:	IMPLEMENTATION OF UNINFORMED SEARCH ALGORITHMS – BFS	Pg no:

AIM:

To implement a python program for Breadth First Search (BFS).

ALGORITHM:

Step 1:Start

Step 2: We start the process by considering any random node as the starting vertex.

Step 3: We enqueue (insert) it to the queue and mark it as visited.

Step 4: Then we mark and enqueue all of its unvisited neighbours at the current depth or continue to the next depth level if there is any.

Step 5: The visited vertices are removed from the queue.

Step 6: The process ends when the queue becomes empty.

Step 7: Stop

PROGRAM:

```
from collections import deque

def bfs(graph, start):

  visited = set()  # Set to track visited nodes

  queue = deque([start])  # Initialize queue with start node

  while queue:

    node = queue.popleft()  # Dequeue a node

    if node not in visited:

      print(node, end=" ")  # Process the node

      visited.add(node)  # Mark node as visited

      queue.extend(graph[node])  # Add all neighbors (even visited ones)

# Define the graph as an adjacency list

graph = {
```

```
    1: [2, 3],
    2: [1, 4, 5],
    3: [1, 6],
    4: [2],
    5: [2, 7],
    6: [3],
    7: [5]
}
# Perform BFS traversal from node 1
print("BFS Traversal:")
bfs(graph, 1)
```

RESULT:

Thus the program for breadth-first search was implemented and executed successfully

Exp: 1b Date:	IMPLEMENTATION OF UNINFORMED SEARCH ALGORITHMS-UCS	Pg no:

AIM:

To implement a python code for Uniform-Cost Search (UCS)

ALGORITHM:

Step 1: Start

Step 2: We start the process by considering any random node as the starting vertex.

Step 3: We enqueue (insert) it to the priority queue and mark it as visited when removed.

Step 4: Then we mark and enqueue all of its unvisited neighbors with their total cost added.

Step 5: The visited vertices are not re-added to the queue again.

Step 6: If the goal node is reached, the process returns the least cost to reach it.

Step 7: If not, we repeat the process with the next minimum-cost node from the queue.

Step 8: The process continues until the priority queue becomes empty.

Step 9: If the goal node is not found even after the queue is empty, it means no path exists.

Step 10: Stop

PROGRAM:

```
import heapq  # Import heap queue for priority queue

def uniform_cost_search(graph, start, goal):

   priority_queue = [(0, start)]  # Min-Heap storing (cost, node)

   visited = set()

   while priority_queue:

      cost, node = heapq.heappop(priority_queue)  # Get node with lowest cost

      if node in visited:

         continue  # Skip if already visited
```

```
        visited.add(node)
        if node == goal:
            return cost  # Return the least-cost path
        for neighbor, edge_cost in graph.get(node, []):
            if neighbor not in visited:
                heapq.heappush(priority_queue, (cost + edge_cost, neighbor))
    return float("inf")  # Return if no path found
# Example graph as adjacency list
graph = {
    'A': [('B', 1), ('C', 4)],
    'B': [('D', 2), ('E', 5)],
    'C': [('F', 3)],
    'D': [('G', 1)],
    'E': [('G', 2)],
    'F': [('G', 6)],
    'G': []  # Goal node
}
# Run UCS from 'A' to 'G'
start_node = 'A'
goal_node = 'G'
result = uniform_cost_search(graph, start_node, goal_node)
print(f"Shortest Cost from {start_node} to {goal_node}: {result}")
```

RESULT:

Thus the program for Uniform-Cost Search was implemented and executed successfully

Exp: 1c Date:	IMPLEMENTATION OF UNINFORMED SEARCH ALGORITHMS-DLS	Pg no:

AIM:

To implement a python code for Depth-Limited Search (DLS)

ALGORITHM:

Step 1: Start

Step 2: Begin from the starting node at depth 0.

Step 3: Visit the current node and check if it is the goal.

Step 4: If the current node is the goal, return success (True).

Step 5: If the current depth equals the limit, stop exploring further down this path and return failure (False).

Step 6: Otherwise, recursively explore each unvisited neighbor of the current node by increasing the depth by 1.

Step 7: If any recursive call finds the goal, return success (True).

Step 8: If no neighbors lead to the goal within the depth limit, return failure (False).

Step 9: Repeat steps for all possible paths until either the goal is found or all paths up to the depth limit are explored.

Step 10: Stop

PROGRAM:

```
def depth_limited_search(graph, node, goal, limit, depth=0):
  print(f"Visiting: {node}, Depth: {depth}")
  if node == goal:
    return True  # Goal found
  if depth >= limit:  # Stop at depth limit
    return False
  for neighbor in graph.get(node, []):  # Explore neighbors
```

```
        if depth_limited_search(graph, neighbor, goal, limit, depth + 1):
            return True
    return False  # Goal not found within limit
# Defining a simple graph using an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': [],
    'E': [],
    'F': [],
    'G': []
}
# Running Depth-Limited Search
start_node = 'A'
goal_node = 'G'
depth_limit = 2
found = depth_limited_search(graph, start_node, goal_node, depth_limit)



if found:
    print(f"\nGoal '{goal_node}' found within depth limit {depth_limit}")
else:
    print(f"\nGoal '{goal_node}' NOT found within depth limit {depth_limit}")
```

RESULT:

Thus the program for Depth-Limited Search was implemented and executed successfully

Exp: 1d Date:	IMPLEMENTATION OF UNINFORMED SEARCH ALGORITHMS-IDS	Pg no:

AIM:

To implement a python code for Iterative Deepening Search (IDS)

ALGORITHM:

Step 1: Start

Step 2: Set the initial depth limit to 0.

Step 3: Perform Depth-Limited Search (DLS) up to the current depth limit.

Step 4: If the goal is found within the current depth, return success and stop.

Step 5: If the goal is not found, increase the depth limit by 1.

Step 6: Repeat Steps 3 to 5 until the goal is found or the maximum depth limit is reached.

Step 7: If the goal is not found within the maximum depth, return failure.

Step 8: Stop

PROGRAM:

```
def depth_limited_search(graph, node, goal, limit):
    if node == goal:
        return True  # Goal found
    if limit == 0:
        return False  # Stop if depth limit is reached
    for neighbor in graph.get(node, []):
        if depth_limited_search(graph, neighbor, goal, limit - 1):
            return True
    return False
# Function for Iterative Deepening Search (IDS)
```

```
def iterative_deepening_search(graph, start, goal, max_depth):
  for depth in range(max_depth + 1):
    print(f"Searching at depth {depth}...")
    if depth_limited_search(graph, start, goal, depth):
      print(f"Goal '{goal}' found at depth {depth}")
      return
  print(f"Goal '{goal}' NOT found within depth {max_depth}")
# Example Graph Representation (Adjacency List)
graph = {
  'A': ['B', 'C'],
  'B': ['D', 'E'],
  'C': ['F', 'G'],
  'D': [],
  'E': ['H'],
  'F': [],
  'G': [],
  'H': []
}

# Run IDS to find goal node 'H' starting from 'A'
iterative_deepening_search(graph, 'A', 'H', 3)
```

RESULT:

Thus the program for Iterative Deepening Search was implemented and executed successfully

Exp: 1e Date:	IMPLEMENTATION OF UNINFORMED SEARCH ALGORITHMS-DFS	Pg no:

AIM:

To implement python code for Depth First Search (DFS)

ALGORITHM:

Step 1: Strat

Step 2: Pick any node. If it is unvisited, mark it as visited and recur on allits adjacent nodes.

Step 3: Repeat until all the nodes are visited, or the node to be searched isfound.

Step 4: visited is a set that is used to keep track of visited nodes.

Step 5: The dfs function is called and is passed the visited set, the graph in the form of a dictionary, and A, which is the starting node.

Step 6: dfs follows the algorithm described above:

- It first checks if the current node is unvisited - if yes, it is appended in the visited set.
- Then for each neighbor of the current node, the dfs function isinvoked again.
- The base case is invoked when all the nodes are visited. Thefunction then returns.

Step 7: Stop

PROGRAM:

```
def dfs(graph, node, visited=None):
  if visited is None:
    visited = set()  # Initialize the visited set
  if node not in visited:
    print(node, end=" ")  # Process the node
    visited.add(node)  # Mark as visited
    for neighbor in graph[node]:  # Explore neighbors recursively
```

```
        dfs(graph, neighbor, visited)
# Define the graph as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'H'],
    'F': ['C'],
    'H': ['E']
}
# Perform DFS traversal from node 'A'
print("DFS Traversal: ")
dfs(graph, 'A')
```

RESULT:

Thus the program for Depth-First Search was implemented and executed successfully.

Exp: 2a Date:	IMPLEMENTATION OF INFORMED SEARCH ALGORITHMS- GREEDY BEST-FIRST SEARCH	Pg no:

AIM:

To implement a path finding using Greedy Best-First Search algorithm.

ALGORITHM:

Step 1: Start

Step 2: Initialize the priority queue with the start node

Step 3: Create a visited set and an empty path list

Step 4: Expand the node with the lowest heuristic from the priority queue

Step 5: Check if the current node is the goal node

Step 6: If yes, return the path and stop

Step 7: Mark the current node as visited to avoid cycles

Step 8: Expand all unvisited neighbors and add them to the priority queue

Step 9: Repeat steps 4 to 8 until the queue is empty or the goal is found

Step 10: If the queue is empty and goal is not found, return None

Step 11: Stop

PROGRAM:

```
import heapq

def greedy_bfs(graph, heuristics, start, goal):

  queue = [(heuristics[start], start)]  # Priority queue (heuristic, node)

  visited = set()

  path = []

  while queue:

    _, current = heapq.heappop(queue)  # Get the node with the lowest heuristic

    path.append(current)

    if current == goal:
```

```
            return path  # Return the path when the goal is reached
        visited.add(current)
        for neighbor in graph[current]:
            if neighbor not in visited:
                heapq.heappush(queue, (heuristics[neighbor], neighbor))
    return None  # No path found
# Example Graph
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': [],
    'E': ['H'],
    'F': [],
    'G': [],
    'H': []
}
# Heuristic Values (Estimated cost to goal)
heuristics = {'A': 6, 'B': 4, 'C': 4, 'D': 3, 'E': 2, 'F': 4, 'G': 3, 'H': 0}
# Run Search
start, goal = 'A', 'H'
path = greedy_bfs(graph, heuristics, start, goal)
# Output Result
print(f"Path found: {' → '.join(path)}" if path else "No path found")
```

RESULT:

Thus the program for Greedy Best First search was implemented and executed successfully.

Exp: 2b Date:	IMPLEMENTATION OF INFORMED SEARCH ALGORITHMS- A* SEARCH	Pg no:

AIM:

To implement a path finding using A* search algorithm

ALGORITHM:

Step 1: Place the starting node into OPEN and find its f (n) value.

Step 2: Remove the node from OPEN, having the smallest f (n) value. If it is a goal node then stop and return success.

Step 3: Else remove the node from OPEN, find all its successors.

Step 4: Find the f (n) value of all successors; place them into OPEN and place the removed node into CLOSE.

Step 5: Go to Step-2.

Step 6: Exit.

PROGRAM:

```
import heapq

# Define the graph as an adjacency list with costs

graph = {

    'A': {'B': 2, 'C': 4},

    'B': {'D': 3, 'E': 1, 'C': 2},

    'C': {'F': 5, 'G': 3},

    'D': {},

    'E': {'H': 4},

    'F': {'H': 3},

    'G': {'H': 2},

    'H': {}  # Goal node

}
```

```
# Define the heuristic function (Estimated cost to goal 'H')
heuristic = {
  'A': 7, 'B': 6, 'C': 2, 'D': 5, 'E': 4,
  'F': 3, 'G': 1, 'H': 0  # H is the goal node
}
# A* Search Algorithm
def a_star(graph, heuristic, start, goal):
  queue = [(0, start)]  # Priority queue (F-score, Node)
  g_score = {node: float('inf') for node in graph}  # Initialize g-scores
  g_score[start] = 0
  came_from = {}  # Store path

  while queue:
    _, current = heapq.heappop(queue)  # Pick node with lowest F-score
    if current == goal:  # If goal reached, reconstruct the path
      path = []
      while current in came_from:
        path.append(current)
        current = came_from[current]
      path.append(start)
      return path[::-1]  # Reverse path
    for neighbor, cost in graph[current].items():
      temp_g = g_score[current] + cost  # New g-score
      if temp_g < g_score[neighbor]:  # If better path found
        g_score[neighbor] = temp_g
        f_score = temp_g + heuristic[neighbor]  # F = G + H
        heapq.heappush(queue, (f_score, neighbor))
        came_from[neighbor] = current  # Store best path
```

```
    return None  # No path found

# Run A* Algorithm

start = 'A'

goal = 'H'

path = a_star(graph, heuristic, start, goal)

# Print the result

print(f"Shortest Path: {' -> '.join(path)}" if path else "No path found")
```

RESULT:

Thus the program for A*search was implemented and executed successfully.

Exp: 2c Date:	IMPLEMENTATION OF INFORMED SEARCH ALGORITHMS- MEMORY BOUNDED A* SEARCH	Pg no:

AIM:

To implement memory bounded A* search for path finding problem

ALGORITHM:

Step 1: Start

Step 2: Add the start node to the priority queue with f = g + h

Step 3: Create a closed list to track visited nodes

Step 4: While the queue is not empty:

Step 5: If queue size exceeds memory limit, remove the worst node

Step 6: Pop the node with the lowest f from the queue

Step 7: If it is the goal, return the path and cost

Step 8: Add the node to the closed list

Step 9: For each neighbor:

Step 10: Calculate new g and f

Step 11: If neighbor is not in closed list or has a better g, add to queue

Step 12: Repeat and If goal not found, return failure

Step 13: I Stop

PROGRAM:

```
import heapq

def memory_bounded_a_star(graph, heuristic, start, goal, memory_limit=5):

    open_list = []  # Priority queue: (f, g, node, path)

    heapq.heappush(open_list, (heuristic[start], 0, start, [start]))  # (f, g, node, path)

    closed_list = {}  # Stores visited nodes and their best g-cost

    while open_list:

        if len(open_list) > memory_limit:
```

```python
            open_list.pop()  # Remove least promising node if memory is full
        f, g, node, path = heapq.heappop(open_list)
        if node == goal:
            return path, g  # Goal reached
        closed_list[node] = g
        for neighbor, cost in graph.get(node, {}).items():
            new_g = g + cost
            new_f = new_g + heuristic.get(neighbor, 0)
            if neighbor not in closed_list or new_g < closed_list[neighbor]:
                heapq.heappush(open_list, (new_f, new_g, neighbor, path + [neighbor]))
    return None, float('inf')  # No path found
# Example graph (Adjacency list with costs)
graph = {
    'A': {'B': 2, 'C': 4},
    'B': {'D': 3, 'E': 1},
    'C': {'F': 5, 'G': 3},
    'D': {}, 'E': {'H': 4}, 'F': {'H': 3}, 'G': {'H': 2}, 'H': {}
}
# Heuristic values (Estimated distance to goal H)
heuristic = {'A': 7, 'B': 6, 'C': 2, 'D': 5, 'E': 4, 'F': 3, 'G': 1, 'H': 0}
# Run Memory-Bounded A*
path, cost = memory_bounded_a_star(graph, heuristic, 'A', 'H', memory_limit=5)
# Output
if path:
    print("Optimal Path:", " -> ".join(path))
    print("Total Cost:", cost)
else:
    print("No path found.")
```

RESULT:

Thus the program for memory bounded A* search was implemented and executed successfully.

Exp: 3 Date:	IMPLEMENT NAÏVE BAYES MODELS	Pg no:

AIM:

To implement the Naive Bayes classification algorithm to predict whether a person will play or not based on weather conditions. The algorithm uses features like Outlook, Temperature, Humidity, and Windy to predict the target variable Play (Yes/No). The program will read the training dataset from an Excel file and then classify a new instance using the trained model.

ALGORITHM:

Step 1: Start

Step 2: Load the weather dataset from an Excel file

Step 3: Preprocess the data

- Use LabelEncoder to convert categorical columns (Outlook, Temperature, Humidity, Windy, Play) into numerical values

Step 4: Split the dataset

- Define features X = [Outlook, Temperature, Humidity, Windy]
- Define target y = Play

Step 5: Train the model

- Use the Categorical Naive Bayes classifier to train the model on X and y

Step 6: Prepare a new test instance (e.g., Outlook=Rainy, Temperature=Cool, Humidity=High, Windy=True)

- Encode the test instance using the same LabelEncoder

Step 7: Predict the outcome using the trained model

Step 8: Output the predicted result (Play = Yes or No)

Step 9: Stop

PROGRAM:

```
import pandas as pd
from sklearn.naive_bayes import CategoricalNB
from sklearn.preprocessing import LabelEncoder
df = pd.read_excel("weather_data.xlsx")
# Encode all categorical columns using label encoding
label_encoders = {}
for column in df.columns:
   le = LabelEncoder()
   df[column] = le.fit_transform(df[column])
   label_encoders[column] = le
# Separate the dataset into input features and target variable
X = df[['Outlook', 'Temperature', 'Humidity', 'Windy']]
y = df['Play']
# Train a Naive Bayes classifier on the dataset
model = CategoricalNB()
model.fit(X, y)
# Transform a new input instance using the same label encoders
test_instance = [
   label_encoders['Outlook'].transform(['Rainy'])[0],
   label_encoders['Temperature'].transform(['Cool'])[0],
   label_encoders['Humidity'].transform(['High'])[0],
   label_encoders['Windy'].transform([True])[0]
]
# Predict the class label for the input instance and decode it
predicted = model.predict([test_instance])
predicted_label = label_encoders['Play'].inverse_transform(predicted)
print("Predicted outcome for 'Play':", predicted_label[0])
```

OUTPUT:

Predicted outcome for 'Play': No

DATASET:

weather_data.xlsx.

Outlook	Temperature	Humidity	Windy	Play
Sunny	Hot	High	False	No
Sunny	Hot	High	True	No
Overcast	Hot	High	False	Yes
Rainy	Mild	High	False	Yes
Rainy	Cool	Normal	False	Yes
Rainy	Cool	Normal	True	No
Overcast	Cool	Normal	True	Yes
Sunny	Mild	High	False	No
Sunny	Cool	Normal	False	Yes
Rainy	Mild	Normal	False	Yes
Sunny	Mild	Normal	True	Yes
Overcast	Mild	High	True	Yes
Overcast	Hot	Normal	False	Yes
Rainy	Mild	High	True	No

RESULT:

Thus to implement the Naive Bayes classification algorithm to predict whether a person will play or not based on weather conditions is executed successfully

Exp: 4 Date:	IMPLEMENT BAYESIAN NETWORKS	Pg no:

AIM:

The aim of this program is to calculate the probability that:

- An **alarm has sounded**, but there has been **no burglary** and **no earthquake**.
- Both **David and Sophia** called **Harry** when the alarm went off.

This is done using **Bayes' Theorem** and simple probability calculations.

ALGORITHM:

Step 1: Start

Step 2: Load the dataset from the Excel file named 'events_data.xlsx'

Step 3: Extract the following probabilities from the dataset:

- P(A): Probability that the alarm sounded
- P(B): Probability that a burglary occurred
- P(E): Probability that an earthquake occurred
- P(D|A): Probability that David calls Harry given the alarm sounded
- P(S|A): Probability that Sophia calls Harry given the alarm sounded
- P(¬B): Probability that no burglary occurred
- P(¬E): Probability that no earthquake occurred

Step 4: Calculate the probability that the alarm sounded, but no burglary and no earthquake occurred $P(A \wedge \neg B \wedge \neg E) = P(A) * P(\neg B) * P(\neg E)$

Step 5: Calculate the conditional probability that both David and Sophia called Harry given the alarm sounded $P(D \wedge S \mid A) = P(D|A) * P(S|A)$

Step 6: Calculate the final probability:

Final Probability = $P(A \wedge \neg B \wedge \neg E) * P(D \wedge S \mid A)$

Step 7: Display the final probability

Step 8: Stop

PROGRAM:

```
import pandas as pd

# Load the dataset

df = pd.read_excel('events_data.xlsx')


# Extract the probabilities from the dataset

P_A = df[df['Event'] == 'Alarm sounded (A)']['Probability'].values[0]

P_B = df[df['Event'] == 'Burglary occurred (B)']['Probability'].values[0]

P_E = df[df['Event'] == 'Earthquake occurred (E)']['Probability'].values[0]

P_D_given_A = df[df['Event'] == 'David calls Harry (D|A)']['Probability'].values[0]

P_S_given_A = df[df['Event'] == 'Sophia calls Harry (S|A)']['Probability'].values[0]

P_not_B = df[df['Event'] == 'No Burglary (¬B)']['Probability'].values[0]

P_not_E = df[df['Event'] == 'No Earthquake (¬E)']['Probability'].values[0]


# Calculate the probability of the alarm sounding but no burglary and no earthquake

P_A_and_not_B_and_not_E = P_A * P_not_B * P_not_E


#  Calculate the conditional probability that David and Sophia both called Harry given
the alarm sounded

P_D_and_S_given_A = P_D_given_A * P_S_given_A


#  Multiply the results to get the final probability

final_probability = P_A_and_not_B_and_not_E * P_D_and_S_given_A


# Output the result

print(f"The probability that the alarm has sounded, there was no burglary, no
earthquake, and both David and Sophia called Harry is: {final_probability:.4f}")
```

OUTPUT:

The probability that the alarm has sounded, there was no burglary, no earthquake, and both David and Sophia called Harry is: 0.5807

DATASET:

events_data.xlsx

Event	Probability
Alarm sounded (A)	0.95
Burglary occurred (B)	0.02
Earthquake occurred (E)	0.01
David calls Harry (D|A)	0.90
Sophia calls Harry (S|A)	0.70
No Burglary (¬B)	0.98
No Earthquake (¬E)	0.99

RESULT:

Thus the program to implement Bayesian networks was implemented successfully

Exp: 5a Date:	BUILD REGRESSION MODELS – LINEAR REGRESSION COEFFICIENTS	Pg no:

AIM:

To calculate the regression coefficients and determine the lines of regression for a given set of data points(x , y). The goal is to find

- The regression line of y on x : y=a+bx
- The regression line of x on y : x=a'+b'y

Additionally the program will plot the data points and the corresponding regression line using python.

ALGORITHM:

Step 1: Start

Step 2: Input data arrays x and y

Step 3: Calculate means:
    x_mean = mean(x)
    y_mean = mean(y)

Step 4: Calculate numerator:
    numerator = sum((x[i] - x_mean) * (y[i] - y_mean))

Step 5: Calculate denominator for y on x:
    denominator_yx = sum((x[i] - x_mean)^2)

Step 6: Calculate slope and intercept for y on x:
    b_yx = numerator / denominator_yx
    a_yx = y_mean - b_yx * x_mean

Step 7: Calculate denominator for x on y:
    denominator_xy = sum((y[i] - y_mean)^2)

Step 8: Calculate slope and intercept for x on y:
    b_xy = numerator / denominator_xy
    a_xy = x_mean - b_xy * y_mean

Step 9: Print regression equations

Step 10: Plot data points and regression lines

Step 11: Stop

PROGRAM:

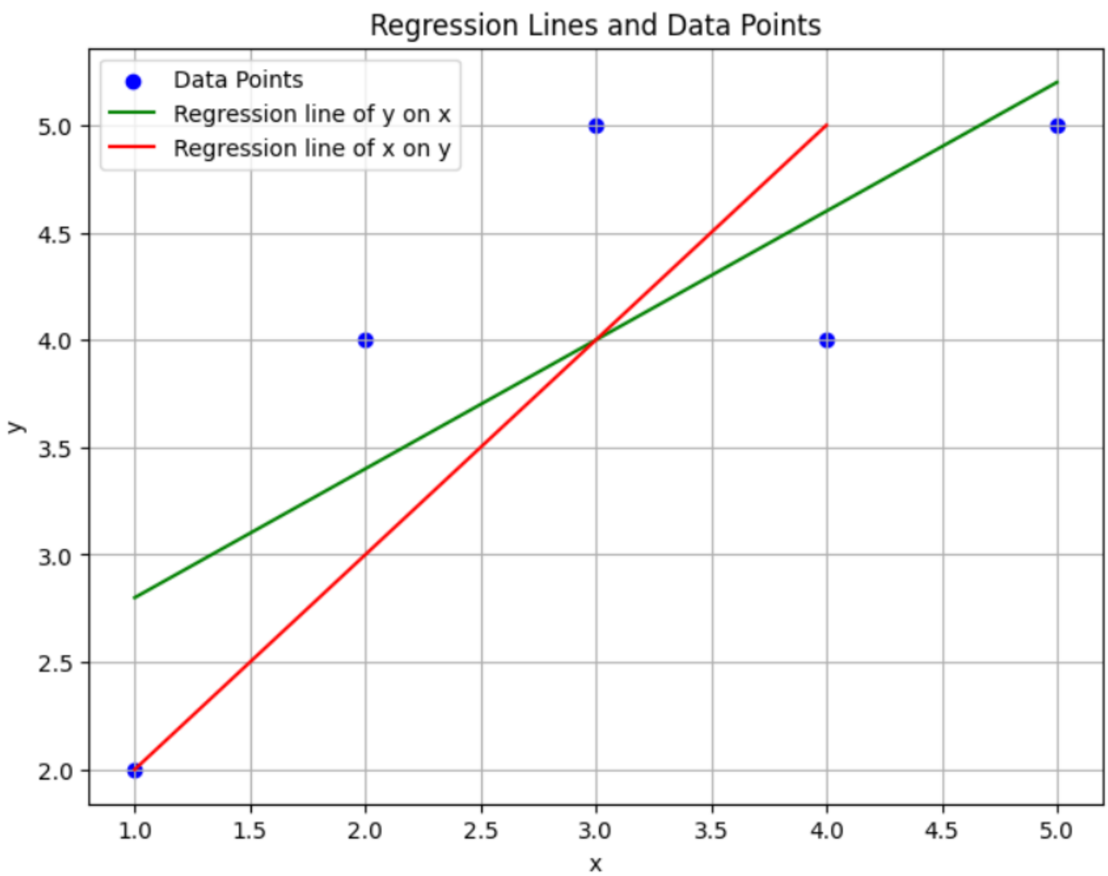
```
import numpy as np

import matplotlib.pyplot as plt

# Sample data

x = np.array([1, 2, 3, 4, 5])

y = np.array([2, 4, 5, 4, 5])

# Calculate means

x_mean = np.mean(x)

y_mean = np.mean(y)

# Regression line of y on x: y = a + bx

b_yx = np.sum((x - x_mean) * (y - y_mean)) / np.sum((x - x_mean) ** 2)

a_yx = y_mean - b_yx * x_mean

y_pred = a_yx + b_yx * x

# Regression line of x on y: x = a' + b'y

b_xy = np.sum((x - x_mean) * (y - y_mean)) / np.sum((y - y_mean) ** 2)

a_xy = x_mean - b_xy * y_mean

x_pred = a_xy + b_xy * y

# Print regression lines

print(f"Regression line of y on x: y = {a_yx:.2f} + {b_yx:.2f}x")

print(f"Regression line of x on y: x = {a_xy:.2f} + {b_xy:.2f}y")

# Plotting

plt.figure(figsize=(8, 6))

plt.scatter(x, y, color='blue', label='Data Points')

# y on x line

plt.plot(x, y_pred, color='green', label='Regression line of y on x')

# x on y line (need to re-sort for correct line shape)

sorted_y = np.sort(y)

sorted_x_pred = a_xy + b_xy * sorted_y
```

```
plt.plot(sorted_x_pred, sorted_y, color='red', label='Regression line of x on y')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Regression Lines and Data Points')
plt.legend()
plt.grid(True)
plt.show()
```

OUTPUT:

Regression line of y on x: y = 2.20 + 0.60x

Regression line of x on y: x = -1.00 + 1.00y



RESULT:

Thus to calculate the regression coefficients and determine the lines of regression for a given set of data points(x , y) and to plot the data points and the corresponding regression line using python was executed successfully.

Exp: 5b Date:	BUILD REGRESSION MODELS – LOGISTIC REGRESSION MODEL	Pg no:

AIM:

To develop a **Logistic Regression model** in Python that predicts a student's **admission status** (Admitted or Not Admitted) based on key factors such as:

- Grade Point Average (GPA)
- SAT Score
- Number of Extracurricular Activities

The model will be trained using data from an Excel file and will evaluate its performance using metrics like accuracy and visual plots.

ALGORITHM:

Step 1: Start

Step 2: Import required libraries (pandas, sklearn, matplotlib, etc.)

Step 3: Load dataset using pandas.read_excel()

Step 4: Preprocess data
- Extract feature columns (GPA, SAT_Score, Extracurriculars)
- Extract target column (Admission)
- Handle missing or invalid values if needed

Step 5: Split dataset into training and testing sets with train_test_split()

Step 6: Train Logistic Regression model using LogisticRegression().fit(X_train, y_train)

Step 7: Make predictions on test data using model.predict(X_test)

Step 8: Evaluate model
- Calculate accuracy with accuracy_score()
- Show confusion matrix or prediction probabilities

Step 9: Plot GPA vs SAT Score, coloring points by admission status

Step 10: (Optional) Predict admission and probability for new student data using

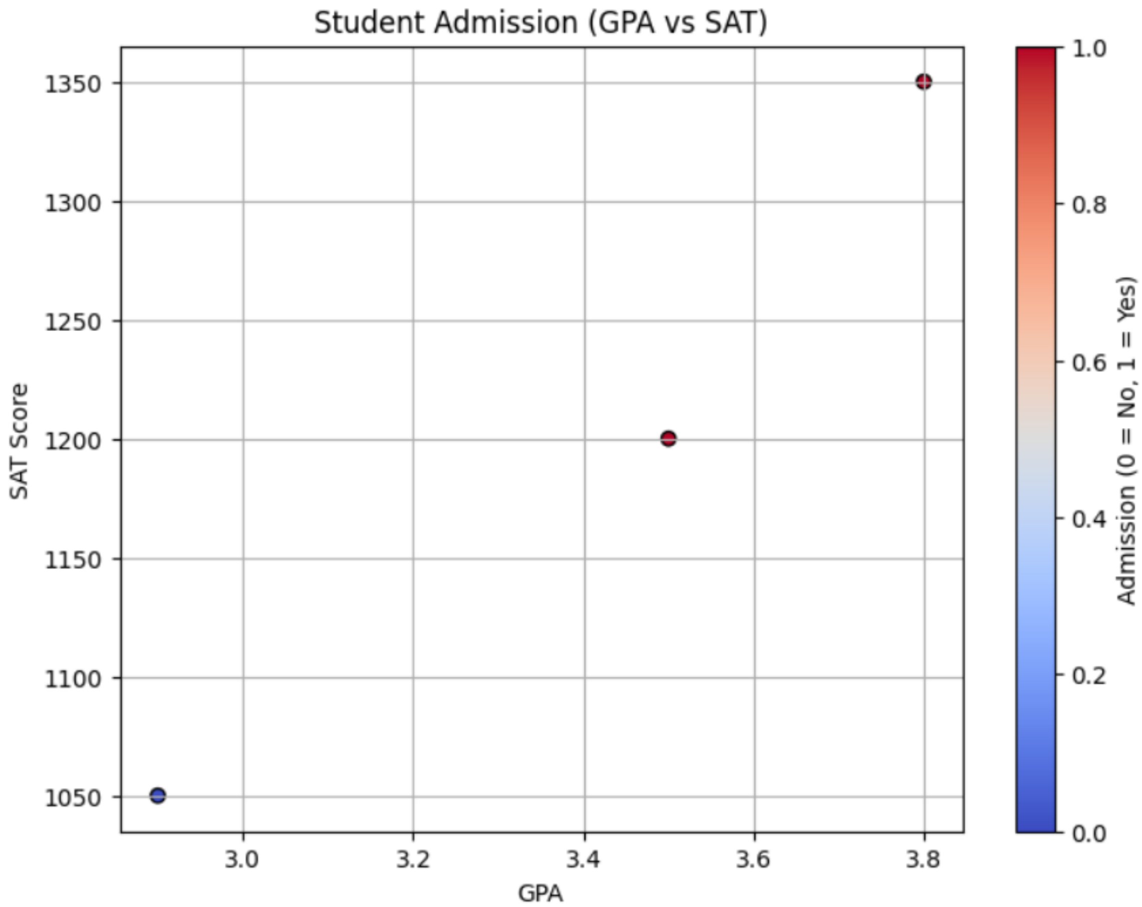
model.predict() and model.predict_proba()

Step 11: Stop

PROGRAM:

```
import pandas as pd

import matplotlib.pyplot as plt

from sklearn.linear_model import LogisticRegression

from sklearn.model_selection import train_test_split

# Load dataset

data = pd.read_excel('your_dataset.xlsx')

# Features and target

X = data[['GPA', 'SAT_Score', 'Extracurriculars']]

y = data['Admission']

# Split dataset

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

# Train model

model = LogisticRegression()

model.fit(X_train, y_train)

# Plot GPA vs SAT with color by admission

plt.figure(figsize=(8, 6))

scatter = plt.scatter(data['GPA'], data['SAT_Score'], c=data['Admission'],
cmap='coolwarm', edgecolor='k')

plt.xlabel('GPA')

plt.ylabel('SAT Score')

plt.title('Student Admission (GPA vs SAT)')

plt.colorbar(scatter, label='Admission (0 = No, 1 = Yes)')

plt.grid(True)

plt.show()
```

OUTPUT:



DATASET:

Your_dataset.xlsx

GPA	SAT_Score	Extracurriculars	Admission
3.8	1350	2	1
2.9	1050	1	0
3.5	1200	3	1

RESULT:

Thus to develop a Logistic Regression model in Python that predicts a student's admission status was executed successfully

Exp: 6a Date:	BUILD DECISION TREES	Pg no:

AIM:

To design and implement a Decision Tree Classifier to predict whether a person buys a computer based on the attributes:

- Age
- Income
- Student
- Credit Rating

The target variable is: Buys_computer (Yes/No)

Algorithm (ID3 - Iterative Dichotomiser 3)

The ID3 algorithm builds the decision tree using entropy and information gain.

ALGORITHM:

Step 1: Start with the full dataset as the root.

Step 2: Calculate entropy of the target attribute (e.g., Buys_computer).

Step 3: For each attribute in the dataset:
 - Calculate information gain for splitting on that attribute.

Step 4: Select the attribute with the highest information gain; this attribute becomes the decision node.

Step 5: Split the dataset into subsets based on the selected attribute's possible values.

Step 6: For each subset:
 - If all examples have the same classification, create a leaf node with that classification.
 - Else, repeat Steps 2–6 recursively on the subset.

Step 7: Continue the recursion until all attributes are used or the tree is complete.

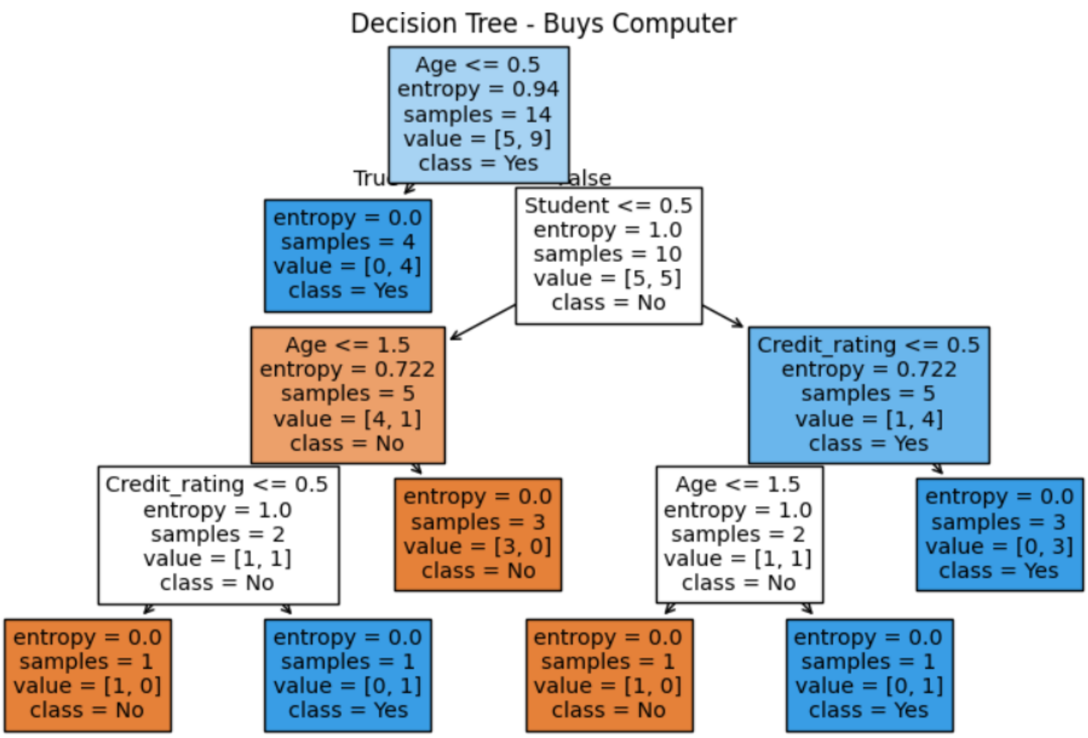
Step 8: Stop.

PROGRAM:

```
import pandas as pd

from sklearn.preprocessing import LabelEncoder

from sklearn.tree import DecisionTreeClassifier, plot_tree

import matplotlib.pyplot as plt

# Step 1: Define dataset

data = {

  'Age': ['Youth', 'Youth', 'Middle-aged', 'Senior', 'Senior', 'Senior',

      'Middle-aged', 'Youth', 'Youth', 'Senior', 'Youth', 'Middle-aged',

      'Middle-aged', 'Senior'],

  'Income': ['High', 'High', 'High', 'Medium', 'Low', 'Low',

        'Low', 'Medium', 'Low', 'Medium', 'Medium', 'Medium',

        'High', 'Medium'],

  'Student': ['No', 'No', 'No', 'No', 'Yes', 'Yes',

        'Yes', 'No', 'Yes', 'Yes', 'Yes', 'No',

        'Yes', 'No'],

  'Credit_rating': ['Fair', 'Excellent', 'Fair', 'Fair', 'Fair', 'Excellent',

           'Excellent', 'Fair', 'Fair', 'Fair', 'Excellent', 'Excellent',

           'Fair', 'Excellent'],

  'Buys_computer': ['No', 'No', 'Yes', 'Yes', 'Yes', 'No',

           'Yes', 'No', 'Yes', 'Yes', 'Yes', 'Yes',

           'Yes', 'No']

}

df = pd.DataFrame(data)

# Step 2: Encode data

le = LabelEncoder()

for column in df.columns:

  df[column] = le.fit_transform(df[column])
```

```
# Step 3: Define features and target
X = df.drop('Buys_computer', axis=1)
y = df['Buys_computer']
# Step 4: Train the model
model = DecisionTreeClassifier(criterion='entropy')
model.fit(X, y)
# Step 5: Plot the tree
plt.figure(figsize=(10, 6))
plot_tree(model, feature_names=X.columns, class_names=['No', 'Yes'], filled=True)
plt.title("Decision Tree - Buys Computer")
plt.show()
```

OUTPUT:



RESULT:

To design and implement a Decision Tree Classifier to predict whether a person buys a computer based on the attributes: Age , Income ,Student ,Credit Rating was executed using python program was executed successfully

Exp: 6b Date:	BUILD RANDOM FORESTS	Pg no:

AIM:

To classify fruits based on their characteristics (such as weight, color, size, and shape) using the Random Forest algorithm. The model will predict the fruit type (e.g., Apple, Orange, Banana) based on the given input features

ALGORITHM:

Step 1: Load and preprocess the dataset (encode categorical variables).

Step 2: Split data into features and target variable.

Step 3: Train multiple decision trees:
- For each tree, randomly select data subset (with replacement) and feature subset.
- Grow tree by splitting data on best features.

Step 4: Combine all trees into the Random Forest model.

Step 5: Predict new data by aggregating tree predictions:
- Classification: majority voting
- Regression: average prediction

Step 6: Evaluate model accuracy on test data.

Step 7: Stop

PROGRAM:

```
import pandas as pd

from sklearn.ensemble import RandomForestClassifier

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score


# Load the dataset from Excel

df = pd.read_excel('fruit_data.xlsx')


# Convert 'Fruit Type' to numerical values
```

```python
df['Fruit_Type_Label'] = df['Fruit Type'].map({'Apple': 0, 'Orange': 1, 'Banana': 2})

# Define the features (X) and target variable (y)
X = df[['Weight (g)', 'Color (encoded)', 'Size (cm)', 'Shape (encoded)']]  # Features
y = df['Fruit_Type_Label']  # Target

# Split the data into training and testing sets (70% training, 30% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Train the Random Forest classifier
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

# Predict the test set results
y_pred = rf_model.predict(X_test)

# Evaluate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')

# Example of predicting a new fruit
def predict_fruit(weight, color, size, shape):
   prediction = rf_model.predict([[weight, color, size, shape]])
   fruit_types = {0: 'Apple', 1: 'Orange', 2: 'Banana'}
   print(f'The predicted fruit is: {fruit_types[prediction[0]]}')

# Test prediction with a new fruit
predict_fruit(160, 1, 6, 1)  # Example: Red Apple, Weight 160g, Size 6cm, Round shape
```

OUTPUT:

Accuracy: 100.00%

The predicted fruit is: Apple

DATASET:

fruit_data.xlsx

Weight (g)	**Colour (encoded)**	**Size (cm)**	**Shape (encoded)**	**Fruit Type**
150	1	6	1	Apple
200	2	7	2	Orange
120	1	5	1	Apple
180	2	6	2	Orange
160	1	6	1	Apple

RESULT:

Thus to classify fruits based on their characteristics (such as weight, color, size, and shape) using the Random Forest algorithm. The model will predict the fruit type (e.g., Apple, Orange, Banana) based on the given input features was executed successfully using python

Exp: 7 Date:	BUILD SVM MODELS	Pg no:

AIM:

To classify data points using a Support Vector Machine (SVM) with a linear kernel and to calculate the margin between the two classes. The program also visualizes the decision boundary and identifies the support vectors.

ALGORITHMS:

Step 1: Start

Step 2: Load dataset from Excel using pandas.read_excel().

Step 3: Extract features X and target y; encode y with LabelEncoder.

Step 4: Create and train SVM model with linear kernel on X and encoded y.

Step 5: Calculate margin and identify support vectors.

Step 6: Visualize data points by class; highlight support vectors and margin.

Step 7: Print support vectors and margin.

Step 8: Stop

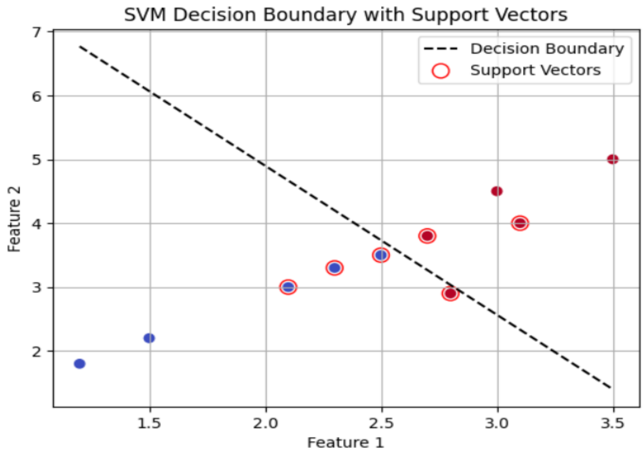
PROGRAM:

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.svm import SVC

from sklearn.preprocessing import LabelEncoder

# Step 1: Load the dataset from an Excel file

df = pd.read_excel('data.xlsx')  # Replace 'data.xlsx' with your actual file path

# Step 2: Preprocess the data

# Assuming the dataset has 'Feature1', 'Feature2' for features and 'Class' for target

X = df[['Feature1', 'Feature2']]  # Replace with your actual feature columns
```

```
y = df['Class']  # Replace with your actual target column

# Step 3: Encode the target variable (Class) into numeric values

label_encoder = LabelEncoder()

y_encoded = label_encoder.fit_transform(y)

# Step 4: Train an SVM model

model = SVC(kernel='linear')

model.fit(X, y_encoded)

# Step 5: Calculate margin (2 / ||w||)

w = model.coef_[0]  # Extract weight vector (coefficients)

margin = 2 / np.linalg.norm(w)  # Margin = 2 / ||w||

print(f"Maximum margin: {margin:.4f}")

# Step 6: Plot the decision boundary (if 2D features)

plt.scatter(X.iloc[:, 0], X.iloc[:, 1], c=y_encoded, cmap='coolwarm', marker='o')  # Data points

# Plot decision boundary

b = model.intercept_[0]

x_vals = np.linspace(X.iloc[:, 0].min(), X.iloc[:, 0].max(), 100)

y_vals = -(w[0] * x_vals + b) / w[1]

plt.plot(x_vals, y_vals, 'k--', label='Decision Boundary')

# Highlight the support vectors

support_vectors = model.support_vectors_  # The actual support vectors

plt.scatter(support_vectors[:, 0], support_vectors[:, 1], facecolors='none', edgecolors='red', s=100, label="Support Vectors")

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.title('SVM Decision Boundary with Support Vectors')

plt.legend()

plt.grid(True)

plt.show()
```

OUTPUT:

Maximum margin: 1.3131



DATASET:

data.xlsx

Feature1	Feature2	Class
2.5	3.5	Cat
1.2	1.8	Cat
3.1	4	Dog
2.8	2.9	Dog
3.5	5	Dog
1.5	2.2	Cat
2.7	3.8	Dog
2.3	3.3	Cat
3	4.5	Dog
2.1	3	Cat

RESULT:

Thus to classify data points using a Support Vector Machine (SVM) with a linear kernel and to calculate the margin between the two classes and visualizes the decision boundary and identifies the support vectors was executed successfully using python program

Exp: 8 Date:	IMPLEMENT ENSEMBLING TECHNIQUES	Pg no:

AIM:

To develop a predictive model that can help healthcare professionals diagnose heart disease based on a dataset of patient health metrics. The goal is to use machine learning models to predict whether a patient is at risk of heart disease (binary classification: Yes or No). Use data set: **heart.csv**

ALGORITHM:

1. Decision Tree:
   - Input: Feature data (e.g., age, cholesterol, etc.)
   - Output: Classification of heart disease (Yes/No)
   - Process: Recursively splits data based on feature values until reaching a leaf node.
2. Random Forest:
   - Input: Feature data (same as Decision Tree)
   - Output: Classification of heart disease
   - Process: Builds multiple decision trees using bootstrapped samples of the data and aggregates their results.
3. SVM (Support Vector Machine):
   - Input: Feature data (e.g., cholesterol, age, etc.)
   - Output: Classification of heart disease
   - Process: Finds the optimal hyperplane that maximizes the margin between two classes (Yes/No).
4. Voting Classifier:
   - Input: Predictions from Decision Tree, Random Forest, and SVM
   - Output: Majority vote (Yes/No) based on the predictions of the individual models.
   - Process: Takes the predictions from each individual model and chooses the class that appears most often (majority voting).

PROGRAM:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.svm import SVC
df = pd.read_csv('heart.csv')
# Split features (X) and target (y)
X = df.drop('target', axis=1)
y = df['target']
# Train/Test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Feature Scaling
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
# Define individual models
dt = DecisionTreeClassifier()
rf = RandomForestClassifier()
svm = SVC(probability=True)
# Train individual models
dt.fit(X_train, y_train)
rf.fit(X_train, y_train)
svm.fit(X_train, y_train)
# Predict using individual models
dt_pred = dt.predict(X_test)
rf_pred = rf.predict(X_test)
```

```
svm_pred = svm.predict(X_test)

# Combine predictions using VotingClassifier

voting_clf = VotingClassifier(estimators=[('dt', dt), ('rf', rf), ('svm', svm)], voting='hard')

voting_clf.fit(X_train, y_train)

ensemble_pred = voting_clf.predict(X_test)

# Convert predictions to "Yes" / "No"

def convert_to_yes_no(predictions):

    return ['Yes' if p == 1 else 'No' for p in predictions]

# Get predictions as "Yes" / "No"

dt_results = convert_to_yes_no(dt_pred)

rf_results = convert_to_yes_no(rf_pred)

svm_results = convert_to_yes_no(svm_pred)

ensemble_results = convert_to_yes_no(ensemble_pred)

# Limit the number of patients to 5

LIMIT = 5

print("Decision Tree Predictions:")

for i, result in enumerate(dt_results[:LIMIT], 1):

    print(f"Patient {i}: {result}")

print("\nRandom Forest Predictions:")

for i, result in enumerate(rf_results[:LIMIT], 1):

    print(f"Patient {i}: {result}")

print("\nSVM Predictions:")

for i, result in enumerate(svm_results[:LIMIT], 1):

    print(f"Patient {i}: {result}")

print("\nVoting Ensemble Predictions (Majority Vote):")

for i, result in enumerate(ensemble_results[:LIMIT], 1):

    print(f"Patient {i}: {result}")
```

OUTPUT:

Decision Tree Predictions:

Patient 1: No

Patient 2: No

Patient 3: No

Patient 4: No

Patient 5: Yes

Random Forest Predictions:

Patient 1: No

Patient 2: Yes

Patient 3: Yes

Patient 4: No

Patient 5: Yes

SVM Predictions:

Patient 1: No

Patient 2: No

Patient 3: Yes

Patient 4: No

Patient 5: Yes

Voting Ensemble Predictions (Majority Vote):

Patient 1: No

Patient 2: No

Patient 3: Yes

Patient 4: No

Patient 5: Yes

RESULT:

Thus to develop a predictive model that can help healthcare professionals diagnose heart disease based on a dataset of patient health metrics was executed successfully

Exp: 9 Date:	IMPLEMENT CLUSTERING ALGORITHMS	Pg no:

AIM:

The goal of this program is to perform clustering on student data based on their marks in Maths and Science. The program uses the K-Means clustering algorithm to divide students into two groups (clusters), and the results are visualized in a 2D scatter plot.

ALGORITHM:

Step 1: Start

Step 2: Load dataset from Excel using pandas.read_excel().

Step 3: Extract features X and target y; encode y with LabelEncoder.

Step 4: Create and train SVM model with linear kernel on X and encoded y.

Step 5: Calculate margin and identify support vectors.

Step 6: Visualize data points by class; highlight support vectors and margin.

Step 7: Print support vectors and margin.

Step 8: Stop

PROGRAM:

```
import pandas as pd

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

# Step 1: Load the Excel dataset (replace with your file path)

df = pd.read_excel('student_marks.xlsx',header=1)

# Step 2: Preview the data to ensure it's loaded correctly

print("Dataset:")

print(df)

# Step 3: Extract relevant columns (Maths and Science marks) for clustering

X = df[['Maths', 'Science']]

# Step 4: Apply K-Means clustering (using 2 clusters for simplicity)
```

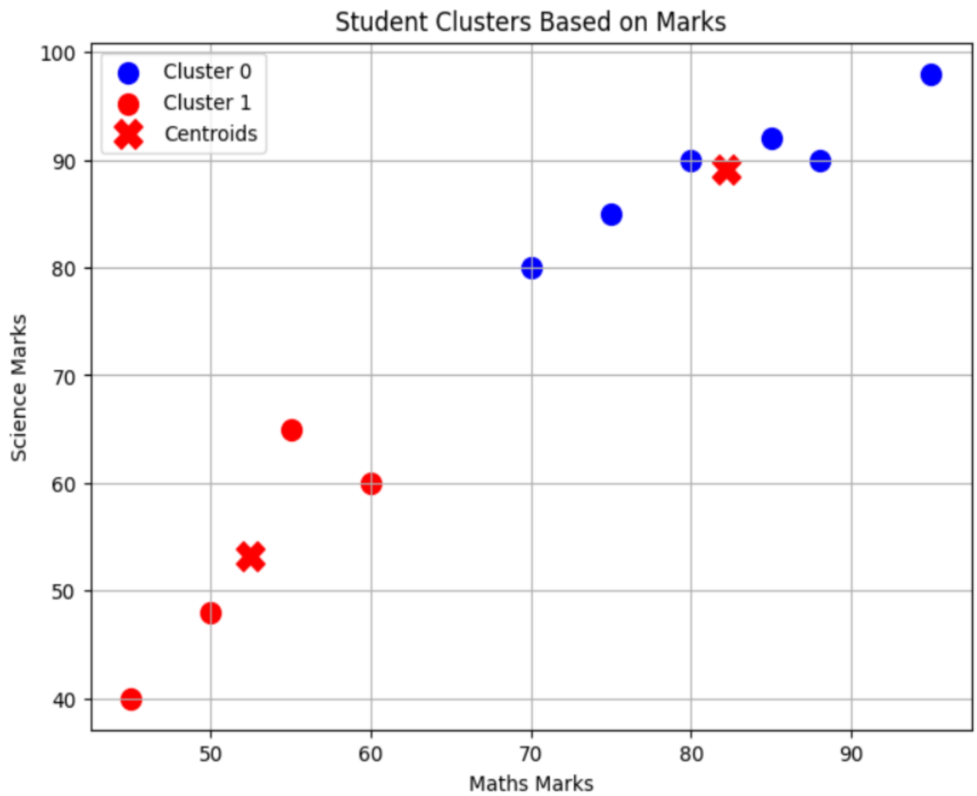
```
kmeans = KMeans(n_clusters=2, random_state=0)

kmeans_labels = kmeans.fit_predict(X)

# Step 5: Add KMeans cluster labels to the DataFrame

df['Cluster'] = kmeans_labels

# Step 6: Plot the clusters

plt.figure(figsize=(8, 6))

plt.scatter(df[df['Cluster'] == 0]['Maths'], df[df['Cluster'] == 0]['Science'],s=100, c='blue',
label='Cluster 0')

plt.scatter(df[df['Cluster'] == 1]['Maths'], df[df['Cluster'] == 1]['Science'],s=100, c='red',
label='Cluster 1')

# Mark the centroids

centroids = kmeans.cluster_centers_

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

# Labels and title

plt.xlabel('Maths Marks')

plt.ylabel('Science Marks')

plt.title('Student Clusters Based on Marks')

plt.legend()

plt.grid(True)

plt.show()

# Step 7: Display the DataFrame with cluster labels

print("\nCluster Results for Each Student:")

print(df
```

OUTPUT:

Dataset:

	Student	Maths	Science
0	S1	75	85
1	S2	80	90

2	S3	85	92
3	S4	60	60
4	S5	70	80
5	S6	95	98
6	S7	88	90
7	S8	45	40
8	S9	55	65
9	S10	50	48



Cluster Results for Each Student:

Student	Maths	Science	Cluster
S1	75	85	0
S2	80	90	0
S3	85	92	0

S4	60	60	1
S5	70	80	0
S6	95	98	0
S7	88	90	0
S8	45	40	1
S9	55	65	1
S10	50	48	1

DATASET:

'student_marks.xlsx'

Student	**Maths**	**Science**
S1	75	85
S2	80	90
S3	85	92
S4	60	60
S5	70	80
S6	95	98
S7	88	90
S8	45	40
S9	55	65
S10	50	48

RESULT:

Thus the goal of this program is to perform clustering on student data based on their marks in Maths and Science. The program uses the K-Means clustering algorithm to divide students into two groups (clusters), and the results are visualized in a 2D scatter plot was executed successfully using python program

Exp: 10 Date:	BUILD DEEPLEARNING NN MODELS	Pg no:

AIM:

To write a Python program is to build a deep learning model that can classify images or features of cats and dogs into two distinct categories:

- Cat (label = 1)
- Dog (label = 0)

ALGORITHM:

Step 1: Start
Step 2: Load dataset from Excel/CSV using pandas.read_excel() or read_csv()
Step 3: Split dataset into features (X) and labels (y)
Step 4: Split X and y into training and test sets using train_test_split()
Step 5: Create a Sequential neural network model
Step 6: Add input, hidden (ReLU), and output (sigmoid) layers
Step 7: Compile model with Adam optimizer, binary crossentropy loss, and accuracy metric
Step 8: Train the model using training data with validation on test data
Step 9: Evaluate model performance using test data and print accuracy
Step 10: Get model weights using model.get_weights()
Step 11: (Optional) Plot training/validation accuracy
Step 12: Stop

PROGRAM:

```
import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.optimizers import Adam

from sklearn.model_selection import train_test_split

import numpy as np

import pandas as pd

# Step 1: Load the dataset (assuming you have preprocessed features in an Excel file)
```

```
# Example: Features and labels are in the dataset (cat = 1, dog = 0)

dataset = pd.read_excel("dataset_features.xlsx")  # Update the path

# Step 2: Separate the features (input) and labels (output)

# Assuming last column is the label (1 for cat, 0 for dog) and all other columns are features

features = dataset.iloc[:, :-1].values  # All columns except last (features)

labels = dataset.iloc[:, -1].values    # The last column (labels)

# Step 3: Split the dataset into training and test sets (80% train, 20% test)

X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=0.2, random_state=42)

# Step 4: Build a simple neural network model

model = Sequential([

    Dense(128, input_dim=X_train.shape[1], activation='relu'),  # First hidden layer (128 neurons)

    Dense(64, activation='relu'),  # Second hidden layer (64 neurons)

    Dense(1, activation='sigmoid')  # Output layer with sigmoid activation (binary classification)

])

# Step 5: Compile the model


model.compile(optimizer=Adam(learning_rate=0.0001),

              loss='binary_crossentropy',

              metrics=['accuracy'])

# Step 6: Train the model

history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test))

# Step 7: Evaluate the model on the test data

loss, accuracy = model.evaluate(X_test, y_test)

print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

```
# Step 8: Get the model's weights after training
# The weights of the model can be accessed via the `model.get_weights()` method.
weights = model.get_weights()
# The weights are a list of NumPy arrays:
# 1. The weights for the first layer (input to hidden)
# 2. The weights for the second layer (hidden to hidden)
# 3. The weights for the output layer (hidden to output)
print("Weights of the model after training:")
# For each layer, display the shape of the weights
for i, weight_matrix in enumerate(weights):
    print(f"Layer {i+1} weights shape: {weight_matrix.shape}")
# Optional: Accessing individual weight matrices
# Example: Weights of the first layer
first_layer_weights = weights[0]
print(f"First layer weights:\n{first_layer_weights}")
# Optional: Plotting the training history
import matplotlib.pyplot as plt
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training and Validation Accuracy')
plt.show()
```

OUTPUT:

Epoch 1/10

1/1 ━━━━━━━━━━━━━━━━━━━━ **2s** 2s/step - accuracy: 0.7500 - loss: 0.6866 - val_accuracy: 1.0000 - val_loss: 0.6753

Epoch 2/10

1/1 ━━━━━━━━━━━━━━━━━━━━ **0s** 320ms/step - accuracy: 0.7500 - loss: 0.6858 - val_accuracy: 1.0000 - val_loss: 0.6738

Epoch 3/10

1/1 ━━━━━━━━━━━━━━━━━━━━ **0s** 138ms/step - accuracy: 0.7500 - loss: 0.6850 - val_accuracy: 1.0000 - val_loss: 0.6724

Epoch 4/10

1/1 ━━━━━━━━━━━━━━━━━━━━ **0s** 88ms/step - accuracy: 0.7500 - loss: 0.6842 - val_accuracy: 1.0000 - val_loss: 0.6710

Epoch 5/10

1/1 ━━━━━━━━━━━━━━━━━━━━ **0s** 88ms/step - accuracy: 0.7500 - loss: 0.6835 - val_accuracy: 1.0000 - val_loss: 0.6696

Epoch 6/10

1/1 ━━━━━━━━━━━━━━━━━━━━ **0s** 86ms/step - accuracy: 0.7500 - loss: 0.6827 - val_accuracy: 1.0000 - val_loss: 0.6682

Epoch 7/10

1/1 ━━━━━━━━━━━━━━━━━━━━ **0s** 158ms/step - accuracy: 0.7500 - loss: 0.6819 - val_accuracy: 1.0000 - val_loss: 0.6668

Epoch 8/10

1/1 ━━━━━━━━━━━━━━━━━━━━ **0s** 91ms/step - accuracy: 0.7500 - loss: 0.6812 - val_accuracy: 1.0000 - val_loss: 0.6654

Epoch 9/10

1/1 ━━━━━━━━━━━━━━━━━━━━ **0s** 141ms/step - accuracy: 0.7500 - loss: 0.6804 - val_accuracy: 1.0000 - val_loss: 0.6640

Epoch 10/10

1/1 ━━━━━━━━━━━━━━━━━━━━ **0s** 86ms/step - accuracy: 0.7500 - loss: 0.6797 - val_accuracy: 1.0000 - val_loss: 0.6626

1/1 ━━━━━━━━━━━━━━━━━━━━ **0s** 41ms/step - accuracy: 1.0000 - loss: 0.6626

Test Accuracy: 100.00%

Weights of the model after training:

Layer 1 weights shape: (5, 128)

Layer 2 weights shape: (128,)

Layer 3 weights shape: (128, 64)

Layer 4 weights shape: (64,)

Layer 5 weights shape: (64, 1)

Layer 6 weights shape: (1,)

First layer weights:

```
[[ 0.09714473 -0.1719174  -0.11225949  0.1294992  -0.19674918  0.1751476
  -0.02220348  0.10591227 -0.18961126  0.02044471 -0.00172899  0.09937751
  -0.10491961  0.17839386  0.14961576  0.09998244 -0.14000212 -0.13770308
   0.04262323  0.13797009  0.03728966  0.0204394   0.1834835   0.00363517
  -0.04064725  0.10851409 -0.12853003 -0.16217075 -0.13056159 -0.04862206
  -0.05645804 -0.11473333  0.20774005 -0.04181623  0.16600795  0.00576559
  -0.05952976 -0.04022478 -0.18857554  0.16149926 -0.13330534  0.04987532
   0.05808955 -0.02290331 -0.00270355 -0.18048844  0.02393207  0.01750458
   0.16723609  0.14217053 -0.16420828  0.13604318  0.08229798 -0.202198
   0.03965089 -0.06381656 -0.02033089 -0.14581507 -0.03776546  0.10450901
  -0.16166592  0.09039405 -0.03779161 -0.0082919   0.17675318  0.07872729
   0.19711033 -0.15817265 -0.02702729  0.07874455  0.09818119 -0.10468626
   0.01790047  0.14449541  0.17724468  0.16966225  0.18440142 -0.0756184
  -0.118393    0.12196914  0.06593169  0.20310509 -0.02117275 -0.09875843
   0.07827477  0.01626248  0.14110526 -0.15824416 -0.20010619 -0.02444436
   0.05825669  0.17893863 -0.16772822 -0.15529187  0.19091299  0.13984472
   0.1454365  -0.06177621  0.04932488  0.12813266 -0.17574167 -0.0051205
   0.17640036 -0.123067    0.19789104 -0.09895314 -0.20623434  0.11705132
  -0.08669168  0.19371864 -0.1632496   0.0064434  -0.08984829  0.15026097
  -0.11181713 -0.07038777  0.02693863 -0.15556426 -0.08693971 -0.05283966
  -0.05707909  0.06530031  0.13473298  0.00350922 -0.15059549  0.05921122
```

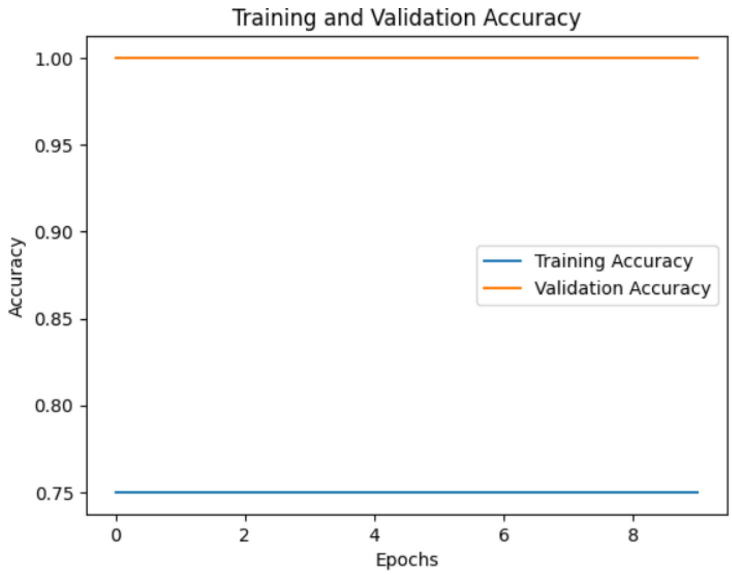
-0.02110778 -0.15924422 0.13396175 -0.12650302 -0.04290656 0.13560952

-0.13127641 0.06530203 0.14744535 -0.18232794 -0.14259335 0.00412669

-0.03137359 -0.12277763 -0.1151494 -0.17829737 0.04939231 0.06566492

 0.16455609 0.19895667 0.07615507 0.01769006 0.04613601 -0.11871487

-0.20090206 0.11605259]]



DATASET:

dataset_features.xlsx

Feature1	**Feature2**	**Feature3**	**Feature4**	**Feature5**	**Label (Cat=1, Dog=0)**
0.1	0.2	0.3	0.4	0.5	1
0.2	0.3	0.4	0.5	0.6	1
0.5	0.1	0.6	0.7	0.2	0
0.3	0.4	0.2	0.1	0.5	0
0.4	0.4	0.5	0.6	0.7	1

RESULT:

Thus to write a Python program is to build a deep learning model that can classify images or features of cats and dogs into two distinct categories:

Cat (label = 1) Dog (label = 0) was executed successfully