

RAJALAKSHMI ENGINEERING COLLEGE

RAJALAKSHMI NAGAR, THANDALAM, CHENNAI 602105

**DEPARTMENT OF COMPUTER SCIENCE AND
ENGINEERING**



**RAJALAKSHMI
ENGINEERING COLLEGE**

An AUTONOMOUS Institution
Affiliated to ANNA UNIVERSITY, Chennai

AI19341

PRINCIPLES OF ARTIFICIAL INTELLIGENCE

THIRD YEAR

FIFTH SEMESTER

INDEX

S.NO	DATE	EXP NAME	VIVA MARK	SIGNATURE
1	09 - 08 - 2024	8QUEENS PROBLEM		
2	16 - 08 - 2024	DEPTH FIRST SEARCH		
3	23 - 08 - 2024	DEPTH FIRST SEARCH - WATER JUG PROBLEM		
4	30 - 08 - 2024	MINIMAX ALGORITHM		
5	06 - 09 - 2024	A* SEARCH ALGORITHM		
6	27 - 09 - 2024	INTRODUCTION TO PROLOG		
7	04 - 10 - 2024	PROLOG FAMILY TREE		
8	18 - 10 - 2024	IMPLEMENTING ARTIFICIAL NEURAL NETWORKS FOR AN APPLICATION USING PYTHON - REGRESSION		
9	25 - 10 - 2024	IMPLEMENTATION OF DECISION TREE CLASSIFICATION TECHNIQUES		
10	08 - 11 - 2024	IMPLEMENTATION OF CLUSTERING TECHNIQUES K - MEANS		

8- QUEENS PROBLEM

AIM:

To implement an 8-Queens problem using Python.

You are given an 8x8 board; find a way to place 8 queens such that no queen can attack any other queen on the chessboard. A queen can only be attacked if it lies on the same row, same column, or the same diagonal as any other queen. Print all the possible configurations.

To solve this problem, we will make use of the Backtracking algorithm. The backtracking algorithm, in general checks all possible configurations and test whether the required result is obtained or not. For the given problem, we will explore all possible positions the queens can be relatively placed at. The solution will be correct when the number of placed queens = 8.



PROGRAM:

```
def share_diagonal(x0, y0, x1, y1):
    dx = abs(x0 - x1)
    dy = abs(y0 - y1)
    return dy == dx

def col_clashes(bs, c):
    for i in range(c):
        if share_diagonal(i, bs[i], c, bs[c]):
            return True
    return False

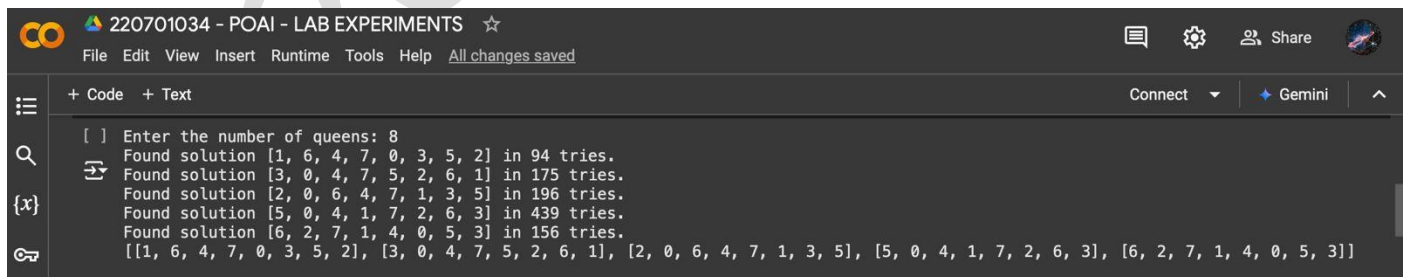
def has_clashes(the_board):
```

```

for col in range(1, len(the_board)):
    if col_clashes(the_board, col):
        return True
    return False
return False
def main():
    import random
    n=int(input("Enter the number of queens: "))
    rng = random.Random()
    bd = list(range(n))
    num_found = 0
    tries = 0
    result = []
    while num_found < 5:
        rng.shuffle(bd)
        tries += 1
        if not has_clashes(bd) and bd not in result:
            print("Found solution {0} in {1} tries.".format(bd, tries))
            tries = 0
            num_found += 1
            result.append(list(bd))
    print(result)
main()

```

OUTPUT:



```

220701034 - POAI - LAB EXPERIMENTS ☆
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Connect Gemini
[ ] Enter the number of queens: 8
Found solution [1, 6, 4, 7, 0, 3, 5, 2] in 94 tries.
Found solution [3, 0, 4, 7, 5, 2, 6, 1] in 175 tries.
Found solution [2, 0, 6, 4, 7, 1, 3, 5] in 196 tries.
Found solution [5, 0, 4, 1, 7, 2, 6, 3] in 439 tries.
Found solution [6, 2, 7, 1, 4, 0, 5, 3] in 156 tries.
[[1, 6, 4, 7, 0, 3, 5, 2], [3, 0, 4, 7, 5, 2, 6, 1], [2, 0, 6, 4, 7, 1, 3, 5], [5, 0, 4, 1, 7, 2, 6, 3], [6, 2, 7, 1, 4, 0, 5, 3]]

```

RESULT:

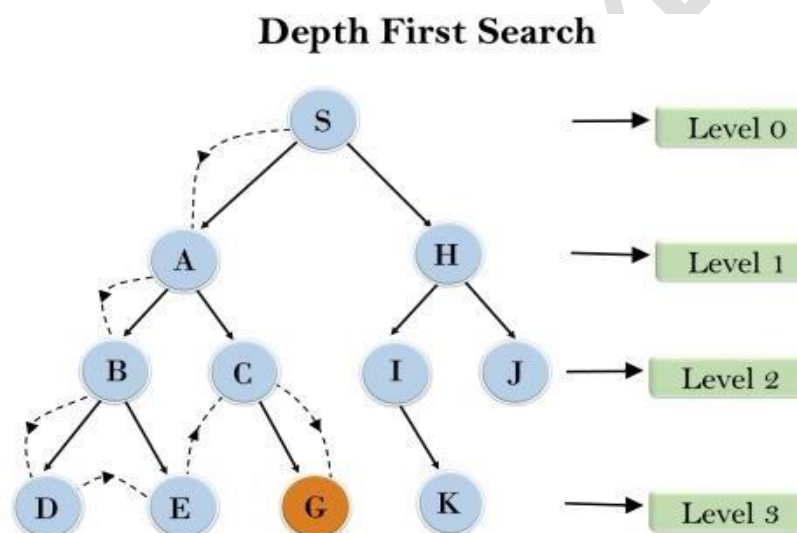
Thus, the 8Queens problem was implemented successfully using backtracking algorithm.

DEPTH FIRST SEARCH

AIM:

To implement a depth-first search problem using Python.

- Depth-first search (DFS) algorithm or searching technique starts with the root node of graph G, and then travel deeper and deeper until we find the goal node or the node which has no children by visiting different node of the tree.
- The algorithm, then backtracks or returns back from the dead end or last node towards the most recent node that is yet to be completely unexplored.
- The data structure (DS) which is being used in DFS Depth-first search is stack. The process is quite similar to the BFS algorithm.
- In DFS, the edges that go to an unvisited node are called discovery edges while the edges that go to an already visited node are called block edges.



PROGRAM:

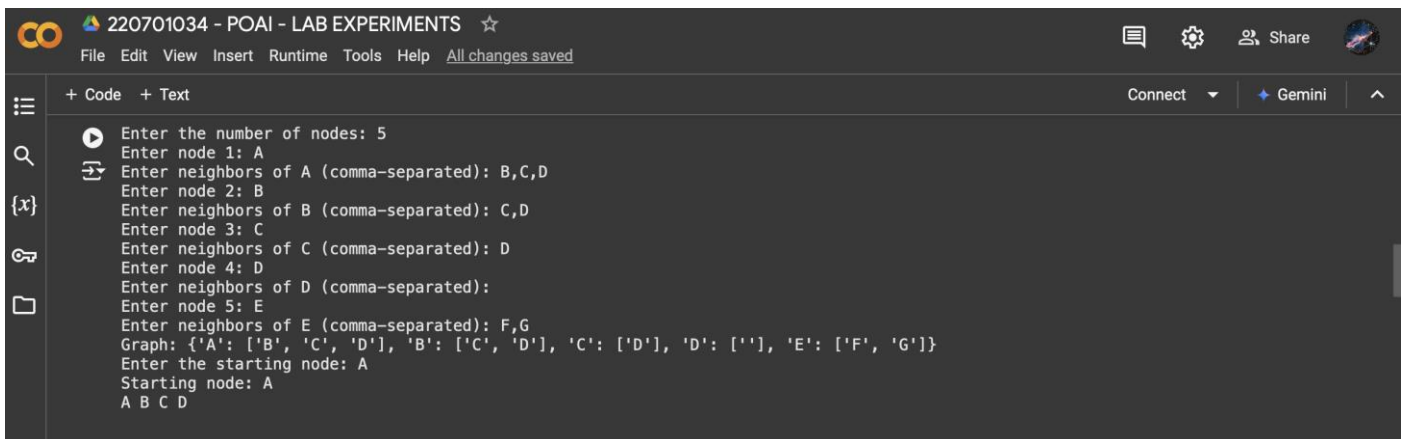
```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=" ")
    for neighbour in graph.get(start, []):
        if neighbour not in visited:
            dfs(graph, neighbour, visited)
num_nodes = int(input("Enter the number of nodes: "))
graph = {}
```

```

for i in range(num_nodes):
    node = input("Enter node " + str(i+1) + ": ").strip()
    neighbors = input("Enter neighbors of " + node + " (comma-separated): ").strip().split(',')
    neighbors = [n.strip() for n in neighbors]
    graph[node] = neighbors
print("Graph:", graph)
start_node = input("Enter the starting node: ").strip()
print("Starting node:", start_node)
dfs(graph, start_node)

```

OUTPUT:



```

220701034 - POAI - LAB EXPERIMENTS ☆
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Connect Gemini
Enter the number of nodes: 5
Enter node 1: A
Enter neighbors of A (comma-separated): B,C,D
Enter node 2: B
Enter neighbors of B (comma-separated): C,D
Enter node 3: C
Enter neighbors of C (comma-separated): D
Enter node 4: D
Enter neighbors of D (comma-separated):
Enter node 5: E
Enter neighbors of E (comma-separated): F,G
Graph: {'A': ['B', 'C', 'D'], 'B': ['C', 'D'], 'C': ['D'], 'D': [], 'E': ['F', 'G']}
Enter the starting node: A
Starting node: A
A B C D

```

RESULT:

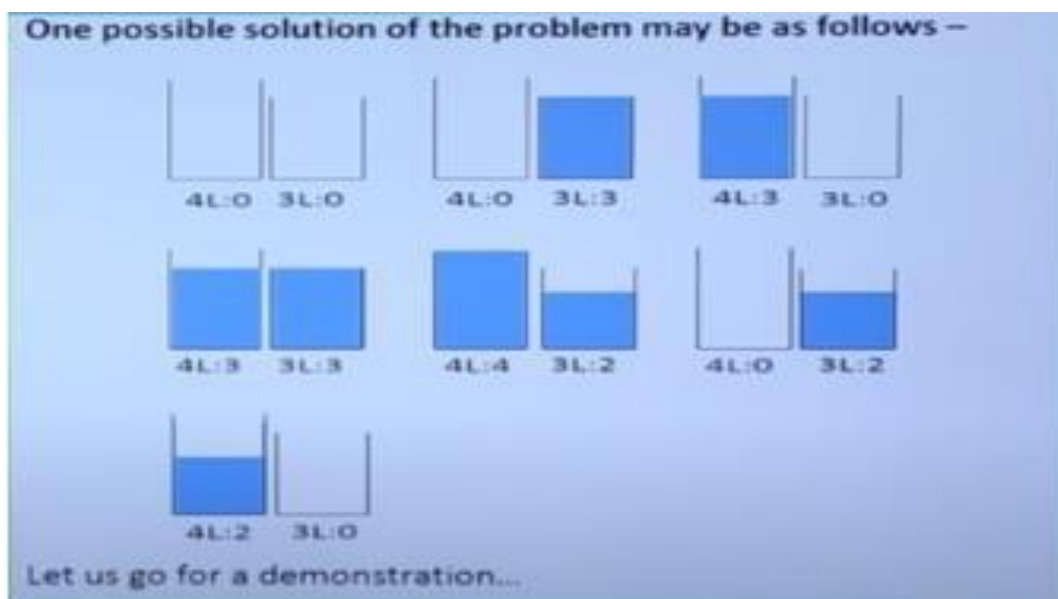
Thus, a searching technique using Depth-First search algorithm was implemented successfully.

DEPTH FIRST SEARCH-WATER JUG PROBLEM

AIM:

To implement **Water – jug problem** using depth first search algorithm.

In the water jug problem in Artificial Intelligence, we are provided with two jugs: one having the capacity to hold 3 gallons of water and the other has the capacity to hold 4 gallons of water. There is no other measuring equipment available and the jugs also do not have any kind of marking on them. So, the agent's task here is to fill the 4-gallon jug with 2 gallons of water by using only these two jugs and no other material. Initially, both our jugs are empty.



PROGRAM:

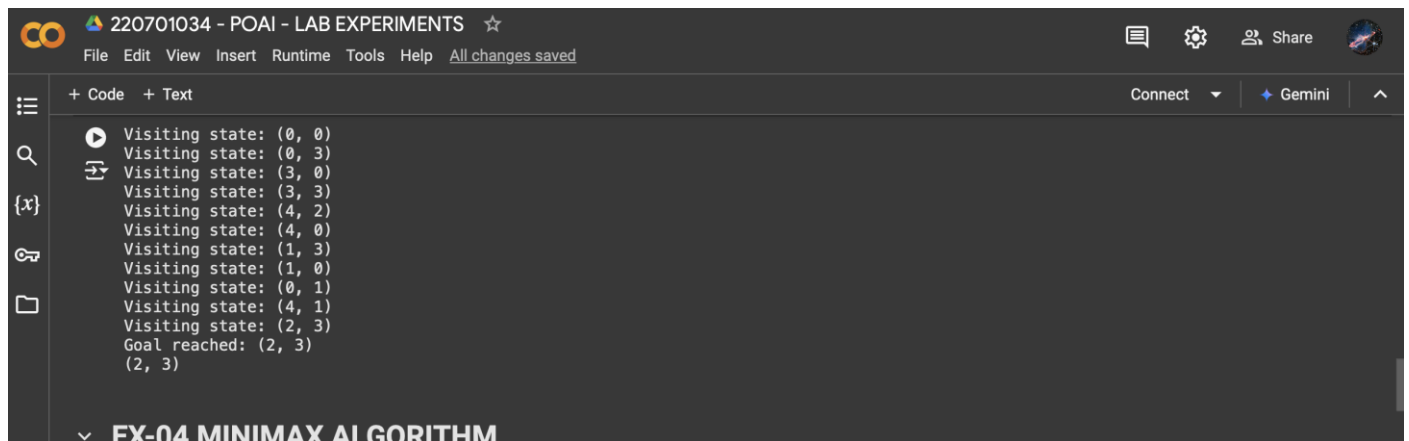
```
def fill_4_gallon(x, y, x_max, y_max):
    return (x_max, y)
def fill_3_gallon(x, y, x_max, y_max):
    return (x, y_max)
def empty_4_gallon(x, y, x_max, y_max):
    return (0, y)
def empty_3_gallon(x, y, x_max, y_max):
    return (x, 0)
def pour_4_to_3(x, y, x_max, y_max):
    220701034
```

```

transfer = min(x, y_max - y)
return (x - transfer, y + transfer)
def pour_3_to_4(x, y, x_max, y_max):
    transfer = min(y, x_max - x)
    return (x + transfer, y - transfer)
def dfs_water_jug(x_max, y_max, goal_x, visited=None, start=(0, 0)):
    if visited is None:
        visited = set()
    stack = [start]
    while stack:
        state = stack.pop()
        x, y = state
        if state in visited:
            continue
        visited.add(state)
        print(f"Visiting state: {state}")
        if x == goal_x:
            print(f"Goal reached: {state}")
            return state
        next_states = [
            fill_4_gallon(x, y, x_max, y_max),
            fill_3_gallon(x, y, x_max, y_max),
            empty_4_gallon(x, y, x_max, y_max),
            empty_3_gallon(x, y, x_max, y_max),
            pour_4_to_3(x, y, x_max, y_max),
            pour_3_to_4(x, y, x_max, y_max)
        ]
        for new_state in next_states:
            if new_state not in visited:
                stack.append(new_state)
    return None
x_max = 4
y_max = 3
goal_x = 2
dfs_water_jug(x_max, y_max, goal_x)

```


OUTPUT:



The screenshot shows a code editor window titled "220701034 - POAI - LAB EXPERIMENTS". The editor displays the output of a minimax algorithm for the water-jug problem. The output consists of a series of "Visiting state:" messages followed by a "Goal reached:" message. The states are represented as (jug1, jug2) pairs. The sequence of states is: (0, 0), (0, 3), (3, 0), (3, 3), (4, 2), (4, 0), (1, 3), (1, 0), (0, 1), (4, 1), (2, 3), and finally (2, 3) where the goal is reached. The editor also shows a sidebar with icons for file explorer, search, and other functions. At the bottom of the editor, there is a tab labeled "EX-04 MINIMAX ALGORITHM".

```
Visiting state: (0, 0)
Visiting state: (0, 3)
Visiting state: (3, 0)
Visiting state: (3, 3)
Visiting state: (4, 2)
Visiting state: (4, 0)
Visiting state: (1, 3)
Visiting state: (1, 0)
Visiting state: (0, 1)
Visiting state: (4, 1)
Visiting state: (2, 3)
Goal reached: (2, 3)
(2, 3)
```

EX-04 MINIMAX ALGORITHM

RESULT:

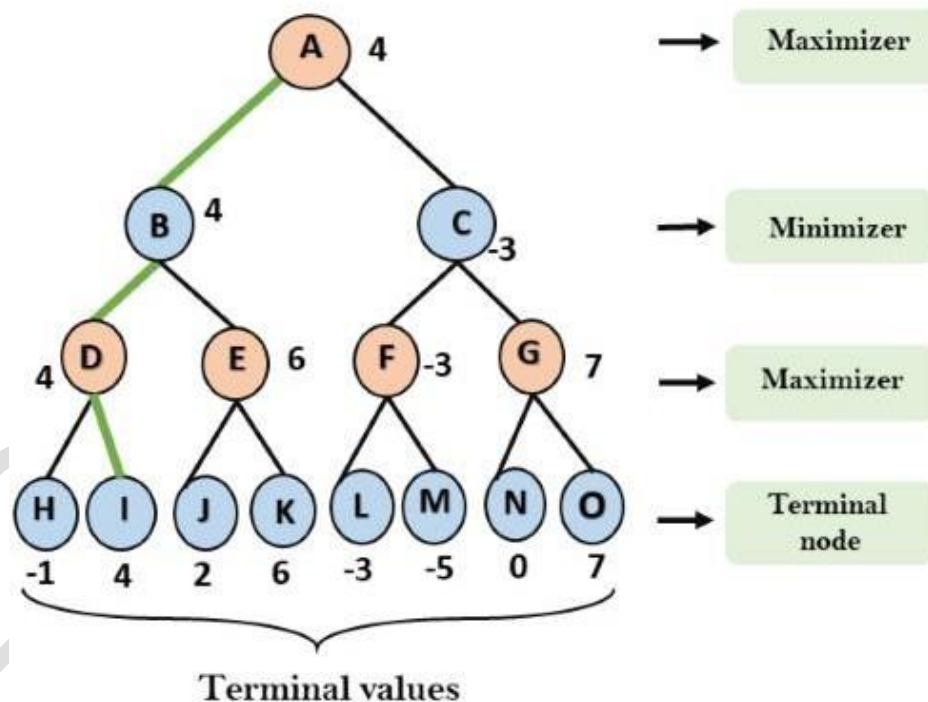
Thus the water-jug problem is implemented successfully using depth-first search algorithm.

MINIMAX ALGORITHM

AIM:

To implement the Minimax Algorithm for a two-player game, with Maximizer maximizing the score and Minimizer minimizing it through DFS evaluation.

- A simple example can be used to explain how the minimax algorithm works. We've included an example of a game-tree below, which represents a two-player game.
- There are two players in this scenario, one named Maximizer and the other named Minimizer.
- Maximizer will strive for the highest possible score, while Minimizer will strive for the lowest possible score.
- Because this algorithm uses DFS, we must go all the way through the leaves to reach the terminal nodes in this game-tree.
- The terminal values are given at the terminal node, so we'll compare them and retrace the tree till we reach the original state.



PROGRAM:

```
import math
def minimax(depth, node_index, is_maximizer, scores, height):
    if depth == height:
        return scores[node_index]
```

```

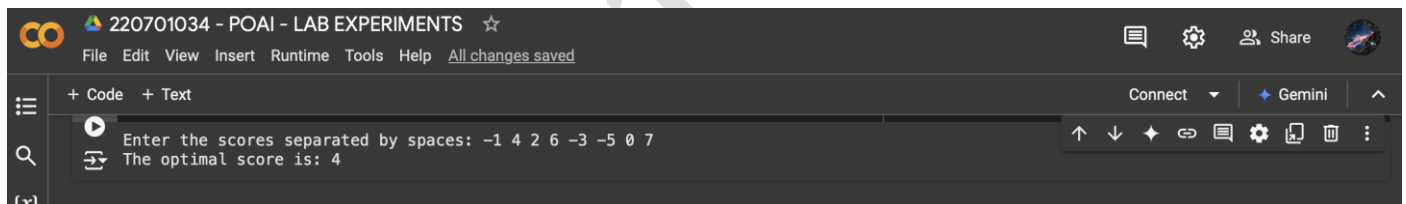
if is_maximizer:
    return max(minimax(depth + 1, node_index * 2, False, scores, height),
               minimax(depth + 1, node_index * 2 + 1, False, scores, height))
else:
    return min(minimax(depth + 1, node_index * 2, True, scores, height),
               minimax(depth + 1, node_index * 2 + 1, True, scores, height))

def calculate_tree_height(num_leaves):
    return math.ceil(math.log2(num_leaves))

scores = list(map(int, input("Enter the scores separated by spaces: ").split()))
tree_height = calculate_tree_height(len(scores))
optimal_score = minimax(0, 0, True, scores, tree_height)
print(f"The optimal score is: {optimal_score}")

```

OUTPUT:



The screenshot shows a code editor window titled "220701034 - POAI - LAB EXPERIMENTS". The code is executed, and the output is displayed in the console: "Enter the scores separated by spaces: -1 4 2 6 -3 -5 0 7" followed by "The optimal score is: 4".

RESULT:

Thus ,the Minimax Algorithm successfully determines the optimal moves for both players by evaluating the game-tree and selecting the best possible scores for Maximizer and Minimizer.

A* SEARCH ALGORITHM**AIM:**

To implement a A* heuristic algorithm to find the least-cost path in a graph using node weights and heuristic approximations for efficient traversal.

A heuristic algorithm sacrifices optimality, with precision and accuracy for speed, to solve problems faster and more efficiently.

All graphs have different nodes or points which the algorithm has to take, to reach the final node. The paths between these nodes all have a numerical value, which is considered as the weight of the path. The total of all paths transverse gives you the cost of that route.

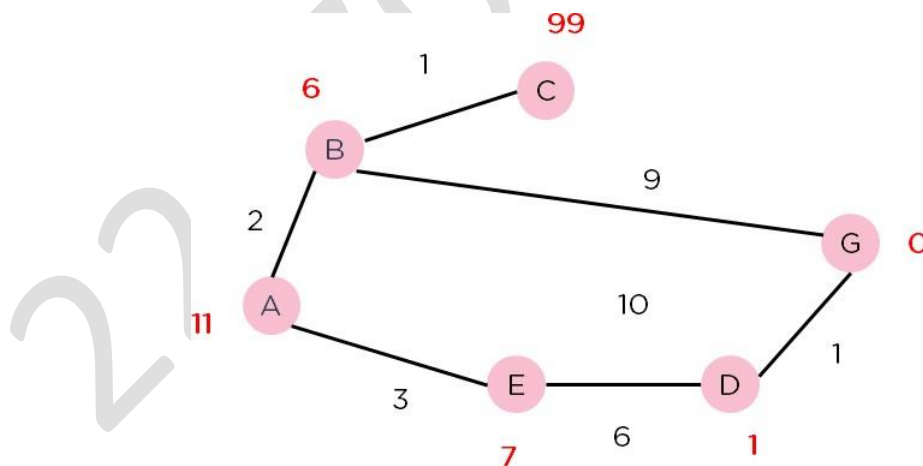
Initially, the Algorithm calculates the cost to all its immediate neighboring nodes, n , and chooses the one incurring the least cost. This process repeats until no new nodes can be chosen and all paths have been traversed. Then, you should consider the best path among them. If $f(n)$ represents the final cost, then it can be denoted as :

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

where:

$g(n)$ = cost of traversing from one node to another. This will vary from node to node

$h(n)$ = heuristic approximation of the node's value. This is not a real value but an approximation cost.

**PROGRAM:**

```

import heapq
class Node:
    def __init__(self, name, parent=None, g=0, h=0):
        self.name = name
        self.parent = parent
  
```

```

    self.g = g
    self.h = h
    self.f = g + h
def __lt__(self, other):
    return self.f < other.f
def a_star(graph, start, goal, h_values):
    open_list = []
    heapq.heappush(open_list, Node(start, None, 0, h_values[start]))
    closed_list = set()
    while open_list:
        current_node = heapq.heappop(open_list)
        if current_node.name == goal:
            path = []
            while current_node:
                path.append(current_node.name)
                current_node = current_node.parent
            return path[::-1]
        closed_list.add(current_node.name)
        for neighbor, cost in graph.get(current_node.name, []):
            if neighbor in closed_list:
                continue
            g_new = current_node.g + cost
            h_new = h_values[neighbor]
            f_new = g_new + h_new
            neighbor_node = Node(neighbor, current_node, g_new, h_new)
            heapq.heappush(open_list, neighbor_node)
    return None
graph = {}
h_values = {}
print("Enter heuristic values for each node. Type 'nil' to stop.")
while True:
    node = input("Enter node name (or 'nil' to finish): ")
    if node.lower() == 'nil':
        break
    h_value = int(input(f"Enter heuristic value for {node}: "))
    h_values[node] = h_value
print("Enter edges and their costs. Type 'nil' to stop.")
while True:

```

```

node1 = input("Enter the start node (or 'nil' to finish): ")
if node1.lower() == 'nil':
    break
node2 = input("Enter the end node: ")
cost = int(input(f"Enter the cost from {node1} to {node2}: "))
if node1 not in graph:
    graph[node1] = []
graph[node1].append((node2, cost))
start_node = input("Enter the start node: ")
goal_node = input("Enter the goal node: ")
path = a_star(graph, start_node, goal_node, h_values)
if path:
    print(f"Path found: {path}")
else:
    print("No path found")

```

OUTPUT:

```

220701034 - POAI - LAB EXPERIMENTS
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Enter heuristic values for each node. Type 'nil' to stop.
Enter node name (or 'nil' to finish): A
Enter heuristic value for A: 11
Enter node name (or 'nil' to finish): B
Enter heuristic value for B: 6
Enter node name (or 'nil' to finish): C
Enter heuristic value for C: 99
Enter node name (or 'nil' to finish): D
Enter heuristic value for D: 1
Enter node name (or 'nil' to finish): E
Enter heuristic value for E: 7
Enter node name (or 'nil' to finish): G
Enter heuristic value for G: 0
Enter node name (or 'nil' to finish): NIL
Enter edges and their costs. Type 'nil' to stop.
Enter the start node (or 'nil' to finish): A
Enter the end node: B
Enter the cost from A to B: 2
Enter the start node (or 'nil' to finish): B
Enter the end node: C
Enter the cost from B to C: 1
Enter the start node (or 'nil' to finish): B
Enter the end node: G
Enter the cost from B to G: 9
Enter the start node (or 'nil' to finish): D
Enter the end node: E
Enter the cost from D to E: 6
Enter the start node (or 'nil' to finish): G
Enter the end node: D
Enter the cost from G to D: 1
Enter the start node (or 'nil' to finish): E
Enter the end node: A
Enter the cost from E to A: 3
Enter the start node (or 'nil' to finish): NIL
Enter the start node: A
Enter the goal node: G
Path found: ['A', 'B', 'G']

```

RESULT:

Thus , the heuristic algorithm successfully identifies an efficient, least-cost path in the graph by evaluating node weights and heuristic estimates.

INTRODUCTION TO PROLOG

AIM:

To learn PROLOG terminologies and write basic programs.

TERMINOLOGIES:

1. Atomic Terms:

Atomic terms are usually strings made up of lower- and uppercase letters, digits, and the underscore, starting with a lowercase letter.

Ex:

dog
ab_c_321

2. Variables:

Variables are strings of letters, digits, and the underscore, starting with a capital letter or an underscore.

Ex:

Dog
Apple_420

3. Compound Terms:

Compound terms are made up of a PROLOG atom and a number of arguments (PROLOG terms, i.e., atoms, numbers, variables, or other compound terms) enclosed in parentheses and separated by commas.

Ex:

is_bigger(elephant,X)
f(g(X,_),7)

4. Facts:

A fact is a predicate followed by a dot.

Ex:

bigger_animal(whale).
life_is_beautiful.

5. Rules:

A rule consists of a head (a predicate) and a body (a sequence of predicates separated by commas).

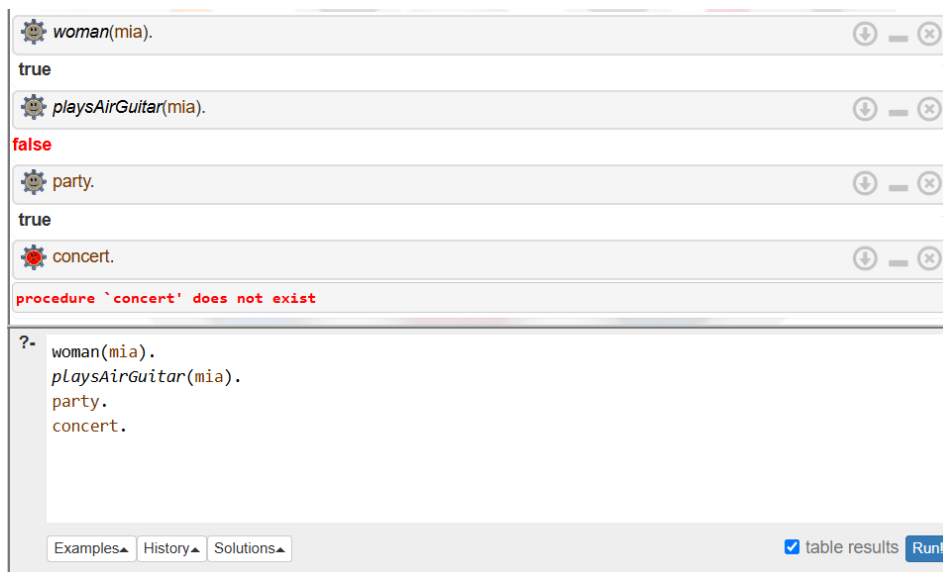
Ex:

is_smaller(X,Y):-is_bigger(Y,X).
aunt(Aunt,Child):-sister(Aunt,Parent),parent(Parent,Child).

SOURCE CODE:

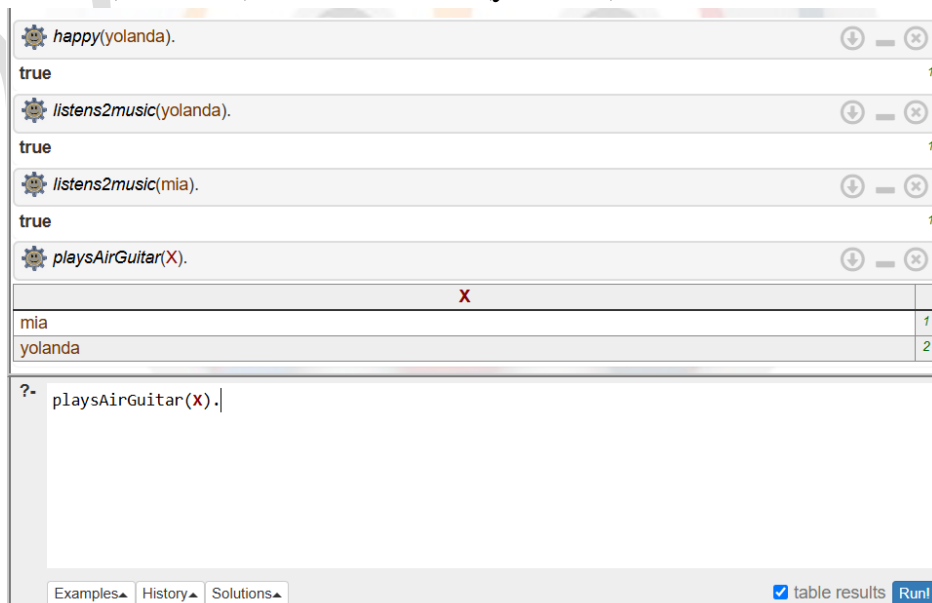
KB1:

woman(mia).
woman(jody).
woman(yolanda).
playsAirGuitar(jody).
party.
Query 1: ?-woman(mia).
Query 2: ?-playsAirGuitar(mia).
Query 3: ?-party.
Query 4: ?-concert.



KB2:

happy(yolanda).
listens2music(mia).
listens2music(yolanda):-happy(yolanda).
playsAirGuitar(mia):-listens2music(mia).
playsAirGuitar(Yolanda):-listens2music(yolanda).



KB3:

likes(dan,sally).
likes(sally,dan).
likes(john,brittney).
married(X,Y) :- likes(X,Y) , likes(Y,X).
friends(X,Y) :- likes(X,Y) ; likes(Y,X).

The screenshot shows a Prolog interpreter window with the following content:

- Query: `married(dan, sally).`
- Result: `true`
- Query: `likes(dan,X)`
- Result: A table with one row containing `sally`.
- Query: `married(john, brittney).`
- Result: `false`
- Query: `?- married(dan, sally). likes(dan,X) married(john, brittney).`
- Result: A table with one row containing `sally`.

The bottom of the window has tabs for "Examples", "History", and "Solutions", and a "Run!" button.

KB4:

food(burger).
food(sandwich).
food(pizza).
lunch(sandwich).
dinner(pizza).
meal(X):-food(X).

The screenshot shows a Prolog interpreter window with the following content:

- Query: `food(pizza)`
- Result: `true`
- Query: `meal(X),lunch(X)`
- Result: A table with one row containing `sandwich`.
- Query: `dinner(sandwich)`
- Result: `false`
- Query: `?- food(pizza) meal(X),lunch(X) dinner(sandwich)`
- Result: A table with one row containing `sandwich`.

The bottom of the window has tabs for "Examples", "History", and "Solutions", and a "Run!" button.

KB5:

owns(jack,car(bmw)).
owns(john,car(chevy)).
owns(olivia,car(civic)).
owns(jane,car(chevy)).
sedan(car(bmw)).
sedan(car(civic)).
truck(car(chevy)).

The screenshot displays a Prolog interpreter window with several query results:

- Query 1:** `owns(John,X)`

John		X
jack		car(bmw)
john		car(chevy)
olivia		car(civic)
jane		car(chevy)
- Query 2:** `owns(John,_)`

John	
jack	
john	
olivia	
jane	
- Query 3:** `owns(Who,car(chevy))`

Who	
john	
jane	
- Query 4:** `owns(jane,X),sedan(X)`

Result: **false**
- Query 5:** `owns(jane,X),truck(X)`

X	
car(chevy)	

At the bottom, a list of loaded clauses is shown:

```
?- owns(John,X)  
owns(John,_)  
owns(Who,car(chevy))  
owns(jane,X),sedan(X)  
owns(jane,X),truck(X)
```

Buttons at the bottom include "Examples", "History", "Solutions", "table results" (checked), and "Run!".

RESULT:

Thus, we have written basic programs to learn prolog terminologies.

PROLOG FAMILY TREE**AIM:**

To develop a family tree program using PROLOG with all possible facts, rules, and queries.

SOURCE CODE:**KNOWLEDGE BASE:**

```
/*FACTS :: */
```

```
male(peter).  
male(john).  
male(chris).  
male(kevin).
```

```
female(betty).  
female(jeny).  
female(lisa).  
female(helen)
```

```
parentOf(chris,peter).  
parentOf(chris,betty).  
parentOf(helen,peter).  
parentOf(helen,betty).  
parentOf(kevin,chris).  
parentOf(kevin,lisa).  
parentOf(jeny,john).  
parentOf(jeny,helen).
```

```
/*RULES :: */
```

```
/* son,parent
```

```
son,grandparent*/
```

```
father(X,Y):- male(Y), parentOf(X,Y).
```

```
mother(X,Y):- female(Y), parentOf(X,Y).
```

```
grandfather(X,Y):- male(Y),parentOf(X,Z),parentOf(Z,Y).
```

```
grandmother(X,Y):- female(Y),parentOf(X,Z),parentOf(Z,Y).
```

```
brother(X,Y):- male(Y), father(X,Z), father(Y,W),Z==W.
```

```
sister(X,Y):- female(Y), father(X,Z),father(Y,W),Z==W.
```

OUTPUT:

365 users online			
Search			
male(Y), parentOf(X,Y).			
Y		X	
peter		chris	1
peter		helen	2
john		jeny	3
chris		kevin	4
false			
female(Y), parentOf(X,Y).			
Y		X	
betty		chris	1
betty		helen	2
lisa		kevin	3
helen		jeny	4
male(Y),parentOf(X,Z),parentOf(Z,Y).			
Y		X	
		Z	
peter	kevin	chris	1
peter	jeny	helen	2
false			
female(Y),parentOf(X,Z),parentOf(Z,Y).			
Y		X	
		Z	
betty	kevin	chris	1
betty	jeny	helen	2
false			
male(Y), father(X,Z), father(Y,W),Z==W.			
procedure `father(A,B)` does not exist			
female(Y), father(X,Z), father(Y,W),Z==W.			
procedure `father(A,B)` does not exist			
?- Examples History Solutions			
table results Run!			

RESULT:

Thus, we have developed a family tree program using PROLOG with all possible facts, rules, and queries.

IMPLEMENTING ARTIFICIAL NEURAL NETWORKS FOR AN APPLICATION USING PYTHON – REGRESSION

Regression using Artificial Neural Networks

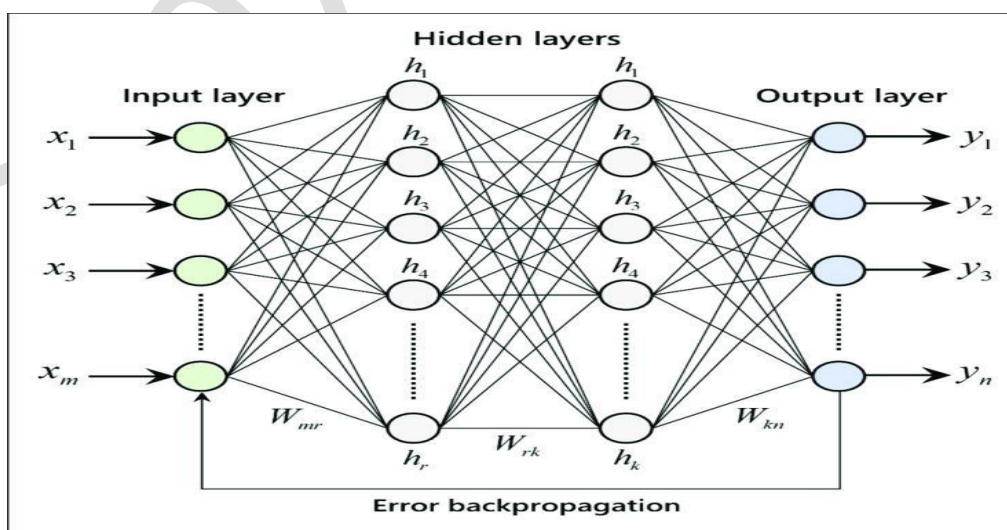
Why do we need to use Artificial Neural Networks for Regression instead of simply using Linear Regression?

The purpose of using Artificial Neural Networks for Regression over Linear Regression is that the linear regression can only learn the linear relationship between the features and target and therefore cannot learn the complex non-linear relationship. In order to learn the complex non-linear relationship between the features and target, we are in need of other techniques. One of those techniques is to use Artificial Neural Networks. Artificial Neural Networks have the ability to learn the complex relationship between the features and target due to the presence of activation function in each layer. Let's look at what are Artificial Neural Networks and how do they work.

Artificial Neural Networks

Artificial Neural Networks are one of the deep learning algorithms that simulate the workings of neurons in the human brain. There are many types of Artificial Neural Networks, Vanilla Neural Networks, Recurrent Neural Networks, and Convolutional Neural Networks. The Vanilla Neural Networks have the ability to handle structured data only, whereas the Recurrent Neural Networks and Convolutional Neural Networks have the ability to handle unstructured data very well. In this post, we are going to use Vanilla Neural Networks to perform the Regression Analysis.

Structure of Artificial Neural Networks



The Artificial Neural Networks consists of the Input layer, Hidden layers, Output layer. The hidden layer can be more than one in number. Each layer consists of n number of neurons. Each layer will be having an Activation Function associated with each of the neurons. The

activation function is the function that is responsible for introducing non-linearity in the relationship. In our case, the output layer must contain a linear activation function. Each layer can also have regularizers associated with it. Regularizers are responsible for preventing overfitting.

Artificial Neural Networks consists of two phases,

- Forward Propagation
- Backward Propagation

Forward propagation is the process of multiplying weights with each feature and adding them. The bias is also added to the result.

Backward propagation is the process of updating the weights in the model. Backward propagation requires an optimization function and a loss function.

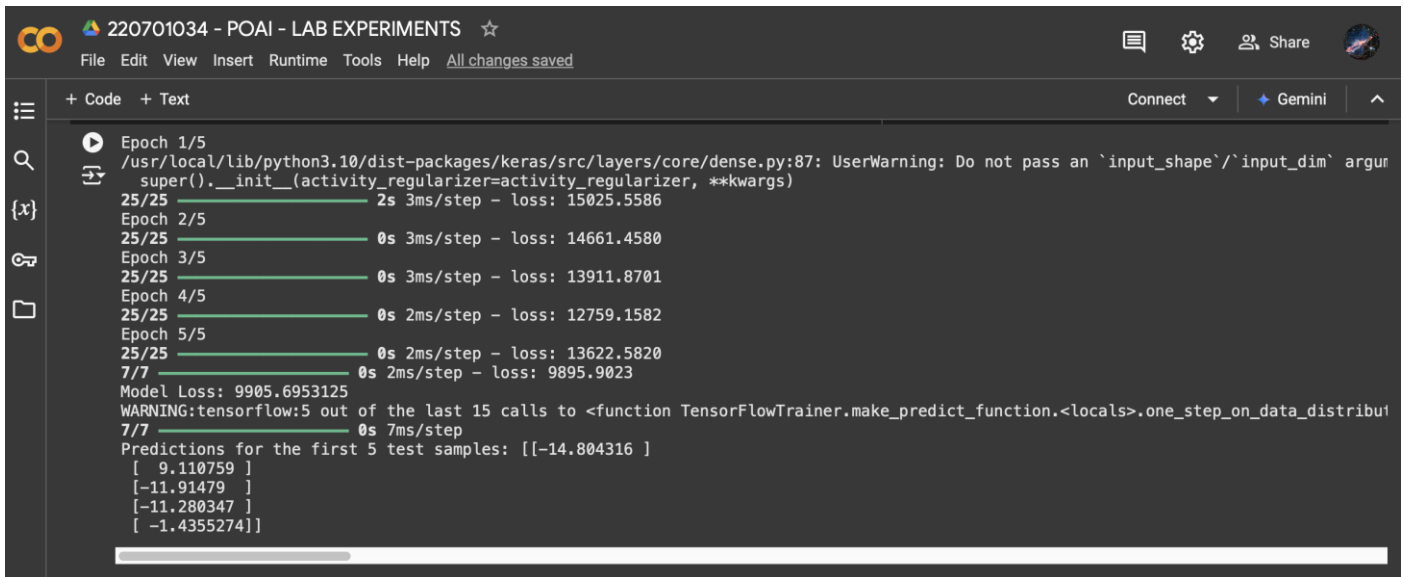
AIM:

To implement artificial neural networks for an application in Regression using python.

PROGRAM:

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_regression
X, y = make_regression(n_samples=1000, n_features=5, noise=0.1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
model = Sequential()
model.add(Dense(units=64, activation='relu', input_dim=X_train.shape[1]))
model.add(Dense(units=32, activation='relu'))
model.add(Dense(units=1, activation='linear'))
model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(X_train, y_train, epochs=5, batch_size=32, verbose=1)
loss = model.evaluate(X_test, y_test)
print(f'Model Loss: {loss}')
y_pred = model.predict(X_test)
print("Predictions for the first 5 test samples:", y_pred[:5])
```

OUTPUT:



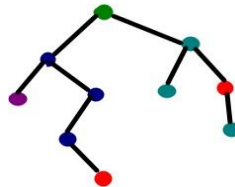
```
Epoch 1/5
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to `Dense` layer.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
25/25 — 2s 3ms/step — loss: 15025.5586
Epoch 2/5
25/25 — 0s 3ms/step — loss: 14661.4580
Epoch 3/5
25/25 — 0s 3ms/step — loss: 13911.8701
Epoch 4/5
25/25 — 0s 2ms/step — loss: 12759.1582
Epoch 5/5
25/25 — 0s 2ms/step — loss: 13622.5820
7/7 — 0s 2ms/step — loss: 9895.9023
Model Loss: 9905.6953125
WARNING:tensorflow:5 out of the last 15 calls to <function TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distribut
7/7 — 0s 7ms/step
Predictions for the first 5 test samples: [[-14.804316 ]
[ 9.110759 ]
[-11.91479 ]
[-11.280347 ]
[-1.4355274]]
```

RESULT:

Thus , we have successfully implemented artificial neural networks for an application in regression using python.

IMPLEMENTATION OF DECISION TREE CLASSIFICATION TECHNIQUES

Decision Tree is one of the most powerful and popular algorithm. Decision-tree algorithm falls under the category of supervised learning algorithms. It works for both continuous as well as categorical output variables.



AIM:

To implement a decision tree classification technique for gender classification using python.

EXPLANATION:

- Import tree from sklearn.
- Call the function DecisionTreeClassifier() from tree
- Assign values for X and Y.
- Call the function predict for Predicting on the basis of given random values for each given feature.
- Display the output.

PROGRAM:

```
from sklearn import tree
X = [[150, 50, 37], [160, 60, 38], [170, 70, 39], [180, 80, 40], [165, 55, 36]]
Y = [0, 0, 1, 1, 0]
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X, Y)
prediction = clf.predict([[175, 75, 41]])
print("Predicted Gender (0 = Female, 1 = Male):", prediction[0])
```

OUTPUT:

```
220701034 - POAI - LAB EXPERIMENTS
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Predicted Gender (0 = Female, 1 = Male): 1
RAM
Disk
Gemini
```

RESULT: Thus, we have successfully implemented a decision tree classification techniques for gender classification.

IMPLEMENTATION OF CLUSTERING TECHNIQUES K – MEANS

The k-means clustering method is an unsupervised machine learning technique used to identify clusters of data objects in a dataset. There are many different types of clustering methods, but k-means is one of the oldest and most approachable. These traits make implementing k-means clustering in Python reasonably straightforward, even for novice programmers and data scientists.

If you're interested in learning how and when to implement k-means clustering in Python, then this is the right place. You'll walk through an end-to-end example of k-means clustering using Python, from preprocessing the data to evaluating results.

How does it work?

First, each data point is randomly assigned to one of the K clusters. Then, we compute the centroid (functionally the center) of each cluster, and reassign each data point to the cluster with the closest centroid. We repeat this process until the cluster assignments for each data point are no longer changing.

K-means clustering requires us to select K, the number of clusters we want to group the data into. The elbow method lets us graph the inertia (a distance-based metric) and visualize the point at which it starts decreasing linearly. This point is referred to as the "elbow" and is a good estimate for the best value for K based on our data.

AIM:

To implement a K - Means clustering technique using python language.

EXPLANATION:

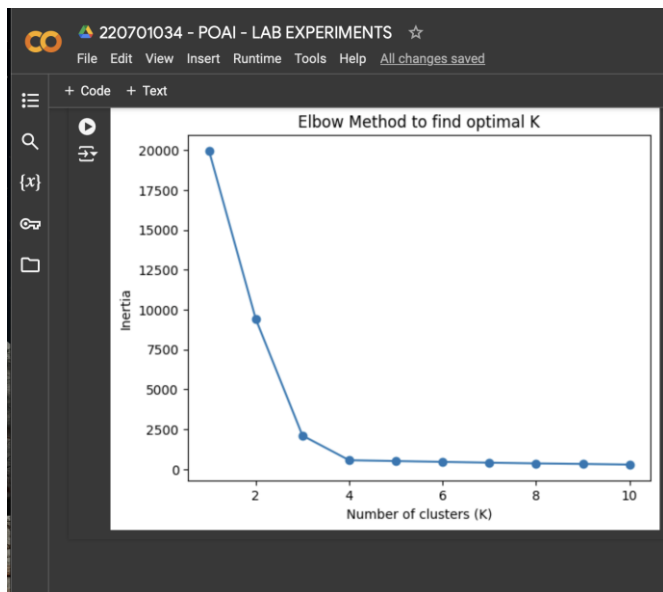
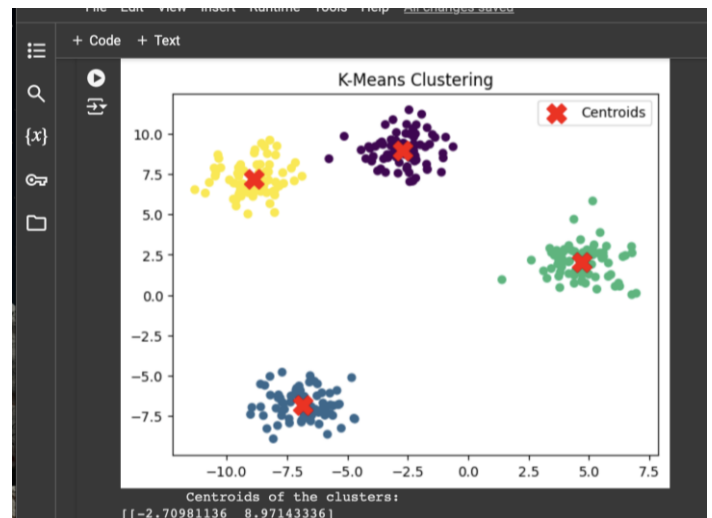
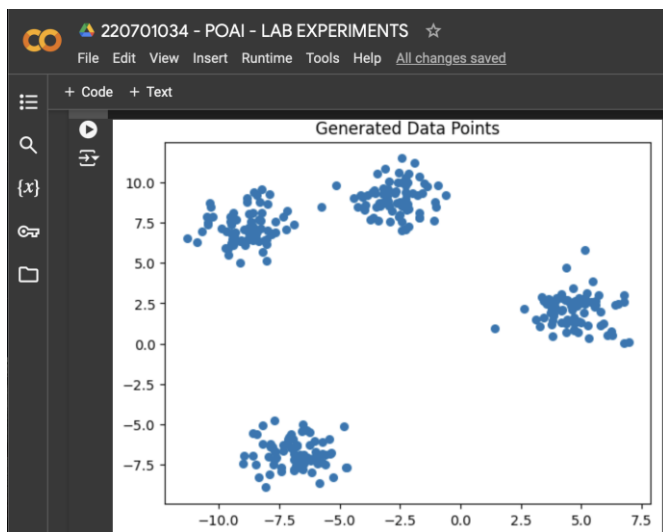
- Import KMeans from sklearn.cluster
- Assign X and Y.
- Call the function KMeans().
- Perform scatter operation and display the output.

PROGRAM:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
X, _ = make_blobs(n_samples=300, centers=4, random_state=42)
220701034
```

```
plt.scatter(X[:, 0], X[:, 1], s=30)
plt.title("Generated Data Points")
plt.show()
kmeans = KMeans(n_clusters=4, random_state=42)
kmeans.fit(X)
centroids = kmeans.cluster_centers_
labels = kmeans.labels_
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=30)
plt.scatter(centroids[:, 0], centroids[:, 1], c='red', marker='X', s=200, label='Centroids')
plt.title("K-Means Clustering")
plt.legend()
plt.show()
print(end="\n")
print("Centroids of the clusters:")
print(centroids, end="\n\n")
inertia = []
for k in range(1, 11):
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(X)
    inertia.append(kmeans.inertia_)
plt.plot(range(1, 11), inertia, marker='o')
plt.title("Elbow Method to find optimal K")
plt.xlabel("Number of clusters (K)")
plt.ylabel("Inertia")
plt.show()
```

OUTPUT:



RESULT:

Thus, we have successfully implemented a K-Means clustering technique using python language.