

**RAJALAKSHMI ENGINEERING COLLEGE (Autonomous)**

**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 LAB**

**THIRD YEAR**

**FIFTH SEMESTER**

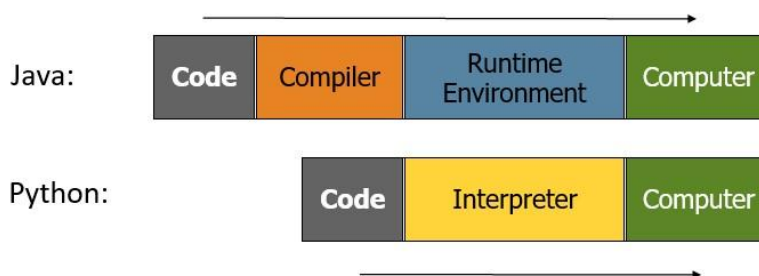
## INDEX

[illegible]

## Working Tools and Language

### PYTHON LANGUAGE

- Interpreter Languages
- Interpreted
- Not compiled like Java
- Code is written and then directly executed by an interpreter
- Type commands into interpreter and see immediate results



## Python Installation Steps

### Windows:

- Download Python from <http://www.python.org>
- Install Python.
- Run **Idle** from the Start Menu.

### Mac OS X:

- Python is already installed.
- Open a terminal and run `python` or run Idle from Finder.

### Linux:

- Chances are you already have Python installed. To check, run `python` from the terminal.
- If not, install from your distribution's package system.

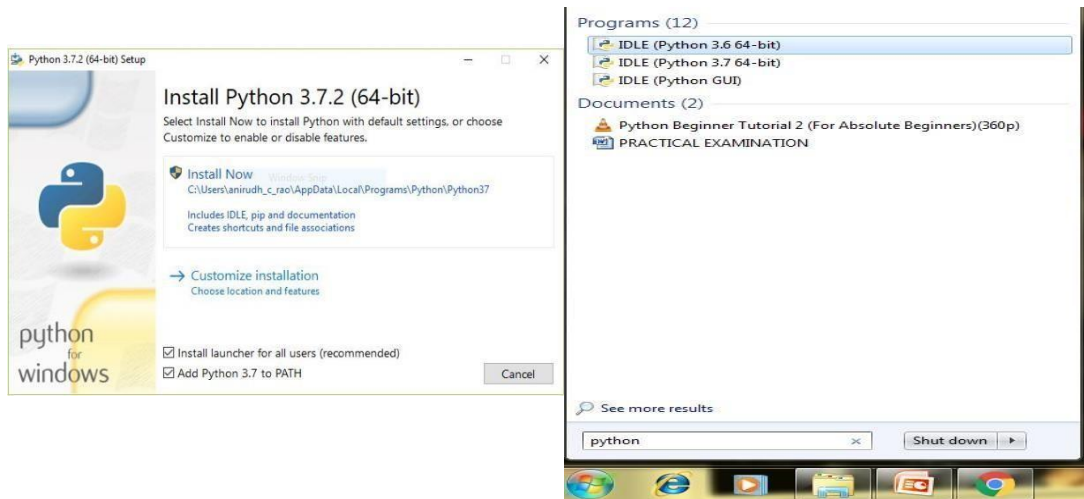
**Note:** For step by step installation instructions, see the course web site.

### Step 1: Download the Python 3 Installer

- Open a browser window and navigate to the Download page for Windows at python.org.
- Underneath the heading at the top that says Python Releases for Windows, click on the link for the Latest Python 3 Release – Python 3.x.x. (As of this writing, the latest version is Python 3.7.2.)
- Scroll to the bottom and select either Windows x86-64 executable installer for 64-bit or Windows x86 executable installer for 32-bit.

### Step 2: Run the Installer

- Once you have chosen and downloaded an installer, simply run it by double-clicking on the downloaded file. A dialog should appear that looks something like this:



## INSTALLATION STEPS ON PYTHON

Google search results for "python 3.6.2 download for 64 bit".

About 66,900 results (0.51 seconds)

<https://www.python.org/downloads/python-362/>  
**Python Release Python 3.6.2 | Python.org**  
 17-Jul-2017 — Python 3.6.2 was the second maintenance release of Python 3.6. ... the installer will download the needed software components at installation time: ... Mac OS X 64-bit/32-bit installer, Mac OS X, for Mac OS X 10.6 and later ...

<https://www.python.org/downloads/python-362rc2/>  
**Download Python | Python.org**  
 Download the latest version of Python. Download Python 3.9.4. Looking for Python with a different OS? Python for Windows, Linux/UNIX, Mac OS X, Other: 3.6.2 Python Releases for Windows Python 2.3.2 Python 2.2.3

<https://www.python.org/downloads/python-362rc2/>  
**Python Release Python 3.6.2rc2 | Python.org**

python.org/downloads/release/python-362/

Python 3.6.2

Release Date: July 17, 2017

Python 3.6.2 was the second maintenance release of Python 3.6.

**Note**

There are now **newer security-fix releases** of Python 3.6 that supersede 3.6.2 and Python 3.6 is now the **latest feature release** of Python 3. Get the latest releases of 3.6.x and 3.8.x here. Python 3.6.8 is planned to be the last bugfix release for 3.6.x. Following the release of 3.6.8, we plan to provide security fixes for Python 3.6 as needed through 2022, five years following its initial release.

### Files

Version	Operating System	Description	MD5 Sum	File Size	GPG
<a href="#">Gzipped source tarball</a>	Source release		e1a36bffd1d3a780b1825daf16e56c	22580749	<a href="#">SIG</a>
<a href="#">XZ compressed source tarball</a>	Source release		2c68846471994897278364fc18730dd9	16907204	<a href="#">SIG</a>
<a href="#">Mac OS X 64-bit/32-bit installer</a>	Mac OS X	for Mac OS X 10.6 and later	86e6193fd56b1e757fc8a5a2bb6c52ba	27561006	<a href="#">SIG</a>
<a href="#">Windows help file</a>	Windows		e520a5c1c3e3f02f68e3db23f74a7a90	8010498	<a href="#">SIG</a>
<a href="#">Windows x86-64 embeddable zip file</a>	Windows	for AMD64/EM64T/x64	0fdfe9f79e0991815d6fc1712871c17f	7047535	<a href="#">SIG</a>
<a href="#">Windows x86-64 executable installer</a>	Windows	for AMD64/EM64T/x64	4377e7d4e6877c248446f7cd6a1430cf	31434856	<a href="#">SIG</a>
<a href="#">Windows x86-64 web-based installer</a>	Windows	for AMD64/EM64T/x64	58ffad3d92a590a463908dfedbc68c18	1312496	<a href="#">SIG</a>
<a href="#">Windows x86 embeddable zip file</a>	Windows		2ca4768fdbadfe670e97857bfab83e8	6332409	<a href="#">SIG</a>
<a href="#">Windows x86 executable installer</a>	Windows		8d8e1711ef9a4b3d3d0ce21e4155c0f5	30507592	<a href="#">SIG</a>
<a href="#">Windows x86 web-based installer</a>	Windows		ccb7d66e3465eaf40ade05b76715b56c	1287040	<a href="#">SIG</a>

## WORKING WITH GOOGLE COLABORATORY

Google search results for "google colab" (About 1,43,00,000 results (0.46 seconds)).

<https://colab.research.google.com>

**Google Colab**

Colab notebooks allow you to combine executable code and rich text in a single document, along with images, HTML, LaTeX and more. When you create your ...

**Authorize with GitHub**  
Introduction to Colab and Python · Run in Google Colab ...

**Google Colab Notebook**  
Enter a search string to filter the list of notebooks shown below. Title ...

**Tensorflow with GPU**  
In this notebook you will connect to a GPU, and then run some ...

**Downloading files from GCS**  
Google Drive. You can access files in Drive in a number of ways ...

**Python Data Science Handbook**  
Python Data Science Handbook ... This is the Jupyter notebook ...

**Colaboratory**  
Colaboratory, or "Colab" for short, is a product from Google ...

More results from google.com »

**Welcome To Colaboratory**  
File Edit View Insert Runtime Tools Help

Table of contents

- Getting started
- Data science
- Machine learning
- More Resources
- Machine Learning Examples
- Section

**What is Colaboratory?**

Colaboratory, or "Colab" for short, allows you to write and execute Python in your browser, with

- Zero configuration required
- Free access to GPUs
- Easy sharing

Whether you're a **student**, a **data scientist** or an **AI researcher**, Colab can make your work easier. Watch [Introduction to Colab](#) to learn more, or just get started below!

**Getting started**

The document you are reading is not a static web page, but an interactive environment called a **Colab notebook** that lets you write and execute code.

For example, here is a **code cell** with a short Python script that computes a value, stores it in a variable, and prints the result:

**Welcome To Colaboratory**  
File Edit View Insert Runtime Tools Help

+ Code + Text Copy to Drive

**What is Colaboratory?**

Colaboratory, or "Colab" for short, allows you to write and execute Python in your browser, with

- Zero configuration required
- Free access to GPUs
- Easy sharing

Whether you're a **student**, a **data scientist** or an **AI researcher**, Colab can make your work easier. Watch [Introduction to Colab](#) to learn more, or just get started below!

**Getting started**

The document you are reading is not a static web page, but an interactive environment called a **Colab notebook** that lets you write and execute code.

For example, here is a **code cell** with a short Python script that computes a value, stores it in a variable, and prints the result:

**Google sign-in required**

You must be logged in with a Google Account to continue.

CANCEL SIGN IN

**Welcome To Colaboratory**  
File Edit View Insert Runtime Tools Help

+ Code + Text Copy to Drive

**What is Colaboratory?**

Colaboratory, or "Colab" for short, allows you to write and execute Python in your browser, with

- Zero configuration required
- Free access to GPUs
- Easy sharing

Whether you're a **student**, a **data scientist** or an **AI researcher**, Colab can make your work easier. Watch [Introduction to Colab](#) to learn more, or just get started below!

**Getting started**

The document you are reading is not a static web page, but an interactive environment called a **Colab notebook** that lets you write and execute code.

For example, here is a **code cell** with a short Python script that computes a value, stores it in a variable, and prints the result:

**Welcome To Colaboratory**  
File Edit View Insert Runtime Tools Help

+ Code + Text Copy to Drive

**What is Colaboratory?**

Colaboratory, or "Colab" for short, allows you to write and execute Python in your browser, with

- Zero configuration required
- Free access to GPUs
- Easy sharing

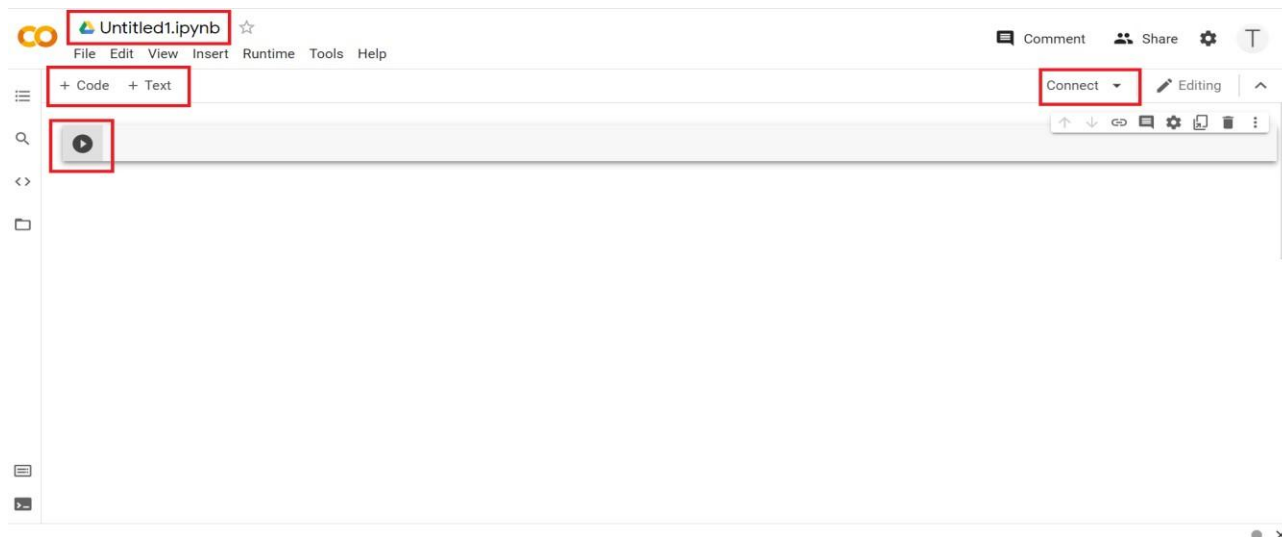
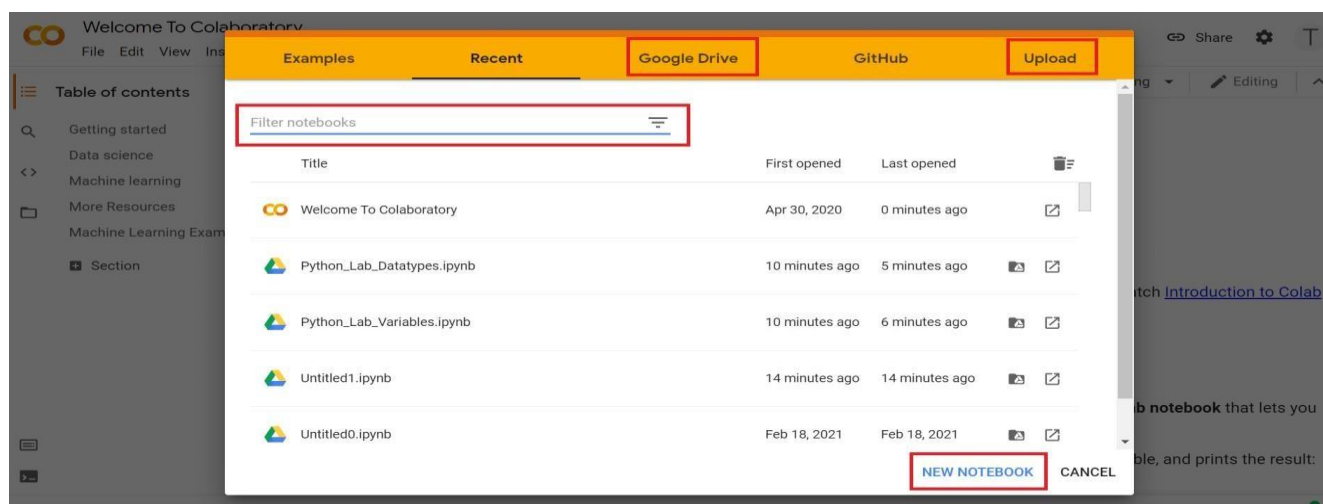
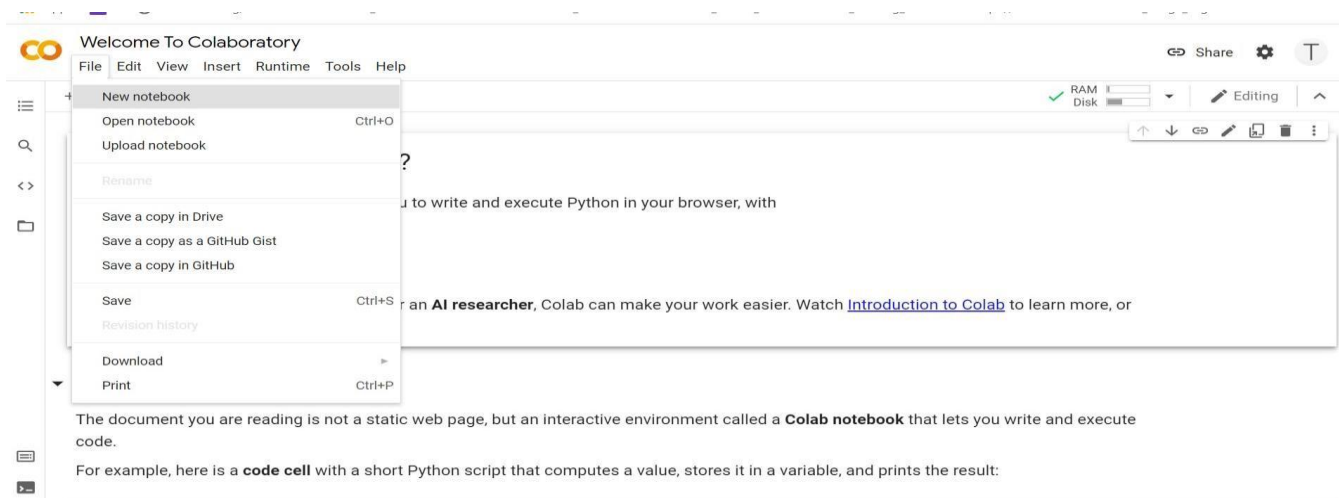
Whether you're a **student**, a **data scientist** or an **AI researcher**, Colab can make your work easier. Watch [Introduction to Colab](#) to learn more, or just get started below!

**Getting started**

The document you are reading is not a static web page, but an interactive environment called a **Colab notebook** that lets you write and execute code.

For example, here is a **code cell** with a short Python script that computes a value, stores it in a variable, and prints the result:

RAM Disk



**EX.NO:**

**DATE:**

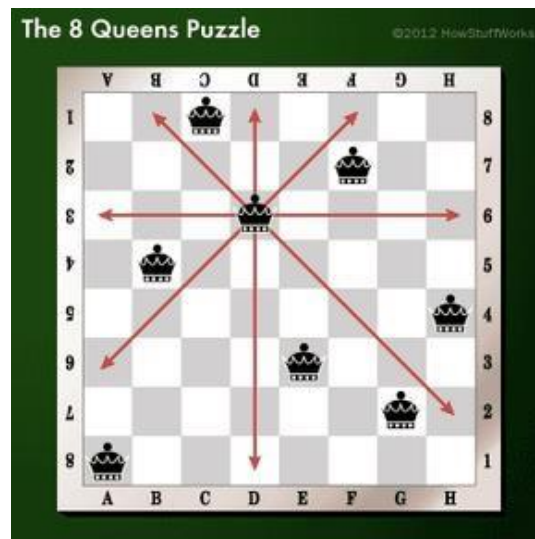
### **8- QUEENS PROBLEM**

**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.







220701053



File Edit View Insert Runtime Tools Help

```

# Function to print the board configuration
def print_board(board):
    for row in board:
        print(" ".join("Q" if col else "." for col in row))
    print("\n")

# Function to check if a queen can be placed at board[row][col]
def is_safe(board, row, col):
    # Check the current row on left side
    for i in range(col):
        if board[row][i]:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j]:
            return False

    # Check lower diagonal on left side
    for i, j in zip(range(row, len(board)), range(col, -1, -1)):
        if board[i][j]:
            return False

    return True

# Recursive function to solve the 8-Queens problem
def solve_8_queens(board, col):
    # Base case: If all queens are placed, print the board
    if col >= len(board):
        print_board(board)
        return True

    # Consider this column and try placing queens in all rows one by one
    res = False
    for i in range(len(board)):
        if is_safe(board, i, col):
            # Place this queen in board[i][col]
            board[i][col] = "Q"
            res = solve_8_queens(board, col + 1)
            if res:
                return True
            board[i][col] = "."
    return res

```



**OUTPUT:**

```
Enter the number of queens (N): 8
Q . . . . .
. . . . Q . .
. . . . . Q .
. . . . . Q .
. . . . . Q .
. . Q . . . .
. . . . . Q .
. Q . . . . .
. . . Q . . .
```

**RESULT:**

Thus ,To implement an 8-Queens problem using Python has been executed and verified.

**EX.NO:**

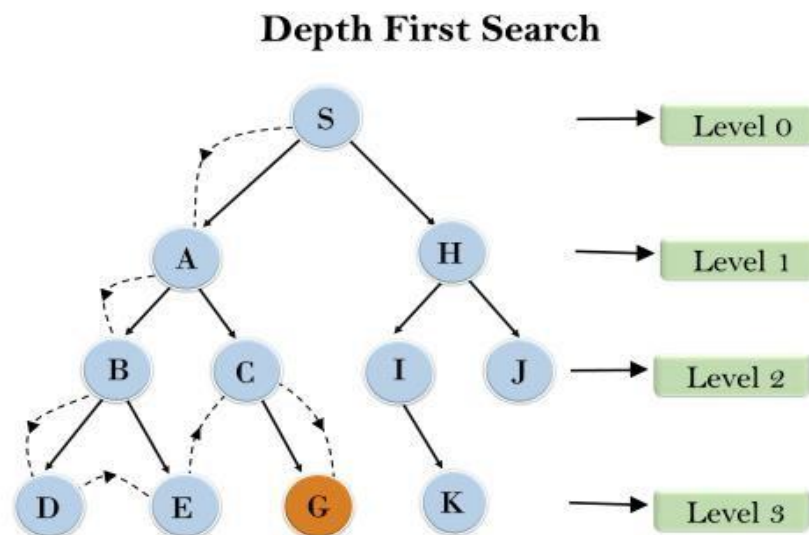
**DATE:**

### **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:**



220701053 

File Edit View Insert Runtime Tools Help

Code + Text

True

```

# Define the graph as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

# DFS function using recursion
def dfs_recursive(graph, node, visited):
    if node not in visited:
        print(node, end=" ")
        visited.add(node)

        # Recursively visit all the unvisited neighbors
        for neighbor in graph[node]:
            dfs_recursive(graph, neighbor, visited)

# DFS function using an explicit stack
def dfs_stack(graph, start):
    visited = set()
    stack = [start]

    while stack:
        node = stack.pop()

        if node not in visited:
            print(node, end=" ")
            visited.add(node)

            # Add neighbors to the stack in reverse order to maintain DFS order
            for neighbor in reversed(graph[node]):
                stack.append(neighbor)

```

**OUTPUT:**A screenshot of a terminal window with a dark background. On the left, there is a light gray icon of a square with two arrows pointing in opposite directions. To the right of the icon, the text "DFS:" is displayed. Below "DFS:", the letters "A", "B", "D", "E", "F", and "C" are listed vertically, one per line.

```
DFS:  
A  
B  
D  
E  
F  
C
```

**RESULT:**

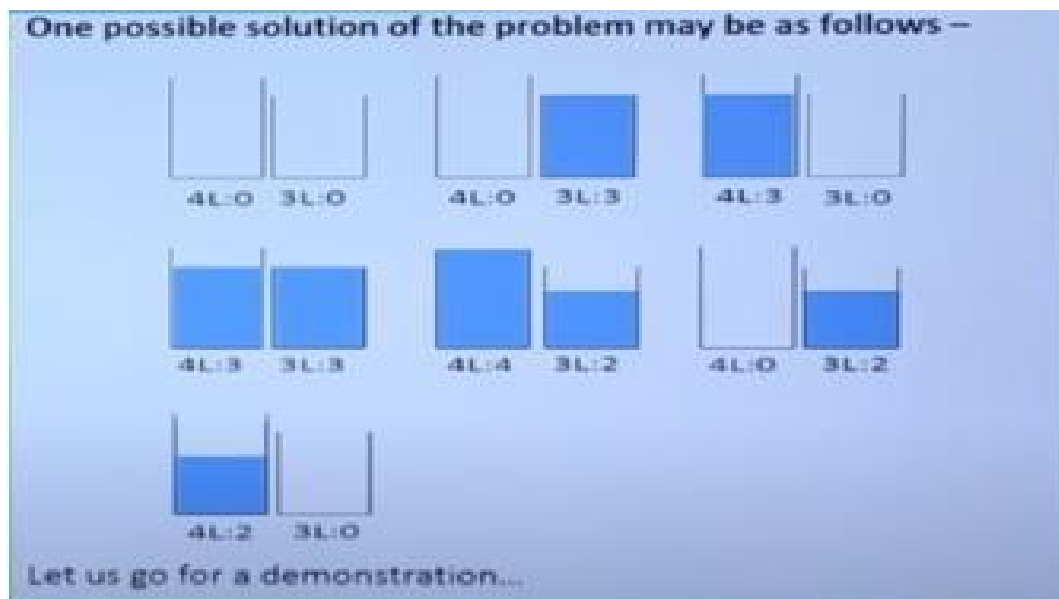
Therefore, To implement a depth-first search problem using Python has been executed and verified.

EX.NO:

DATE:

**DEPTH-FIRST SEARCH – WATER JUG PROBLEM**

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:**



220701053



File Edit View Insert Runtime Tools Help

+ Code + Text

```
from collections import deque

# Function to solve the water jug problem
def water_jug_problem():
    # Initialize the queue for BFS
    queue = deque()
    queue.append((0, 0)) # Starting state with both jugs empty
    visited = set()      # Set to track visited states
    visited.add((0, 0))

    # BFS Loop
    while queue:
        x, y = queue.popleft()

        # If we achieve the goal, print the state and return
        if y == 2:
            print(f"Solution found: (3-gallon jug: {x}, 4-gallon jug: {y})")
            return

        # Print the current state
        print(f"Current state: (3-gallon jug: {x}, 4-gallon jug: {y})")

        # Possible operations
        next_states = [
            (3, y), # Fill the 3-gallon jug
            (x, 4), # Fill the 4-gallon jug
            (0, y), # Empty the 3-gallon jug
            (x, 0), # Empty the 4-gallon jug
            (x - min(x, 4 - y), y + min(x, 4 - y)), # Pour water from 3-gallon jug to 4-gallon jug
            (x + min(y, 3 - x), y - min(y, 3 - x)), # Pour water from 4-gallon jug to 3-gallon jug
        ]

        # Process each next possible state
        for next_state in next_states:
            if next_state not in visited:
                queue.append(next_state)
                visited.add(next_state)

    print("No solution found.")
```

**OUTPUT:**

```
↔ [ ('fill', 2), ('pour', 2, 1), ('fill', 2), ('pour', 2, 1)]
```

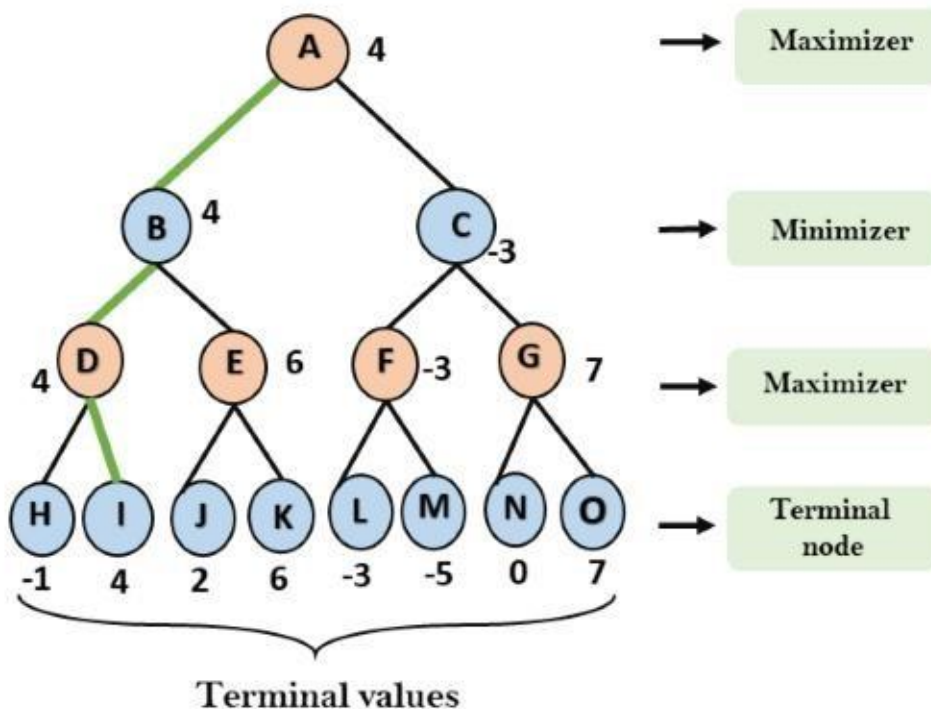
**RESULT:**




Thus, DEPTH-FIRST SEARCH – WATER JUG PROBLEM has been executed and verified.



EX.NO:DATE:MINIMAX ALGORITHM

- 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:**


220701053


File Edit View Insert Runtime Tools Help

+ Code + Text

```

▶ # Minimax function to evaluate a game tree
def minimax(depth, node_index, is_maximizer, scores, max_depth):
    # Base case: If we've reached the terminal node
    if depth == max_depth:
        return scores[node_index]

    # If it's Maximizer's turn, choose the maximum value
    if is_maximizer:
        best = float('-inf')
        # Recur for left and right children
        best = max(best, minimax(depth + 1, node_index * 2, False, scores, max_depth))
        best = max(best, minimax(depth + 1, node_index * 2 + 1, False, scores, max_depth))
        return best

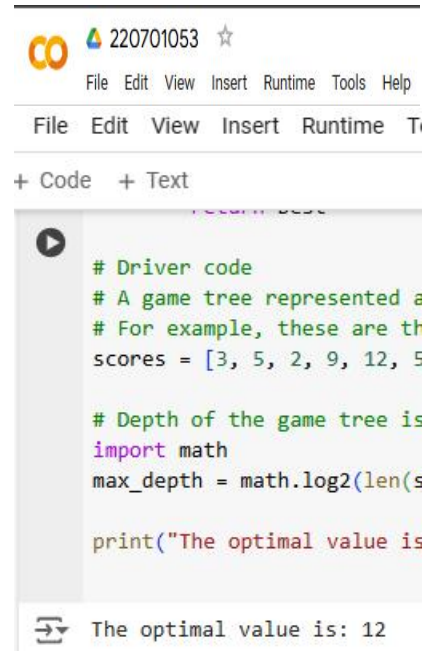
    # If it's Minimizer's turn, choose the minimum value
    else:
        best = float('inf')
        # Recur for left and right children
        best = min(best, minimax(depth + 1, node_index * 2, True, scores, max_depth))
        best = min(best, minimax(depth + 1, node_index * 2 + 1, True, scores, max_depth))
        return best

# Driver code
# A game tree represented as leaf nodes
# For example, these are the terminal nodes and their scores
scores = [3, 5, 2, 9, 12, 5, 23, 23]

# Depth of the game tree is log2(len(scores))
import math
max_depth = math.log2(len(scores))

print("The optimal value is:", minimax(0, 0, True, scores, int(max_depth)))

```

**OUTPUT:**

The screenshot shows a code editor interface with a menu bar (File, Edit, View, Insert, Runtime, Tools, Help) and a toolbar with '+ Code' and '+ Text' buttons. The code is as follows:

```
# Driver code
# A game tree represented a
# For example, these are th
scores = [3, 5, 2, 9, 12, 5

# Depth of the game tree is
import math
max_depth = math.log2(len(s

print("The optimal value is
```

Below the code editor, the output is displayed: "The optimal value is: 12".

**RESULT:**

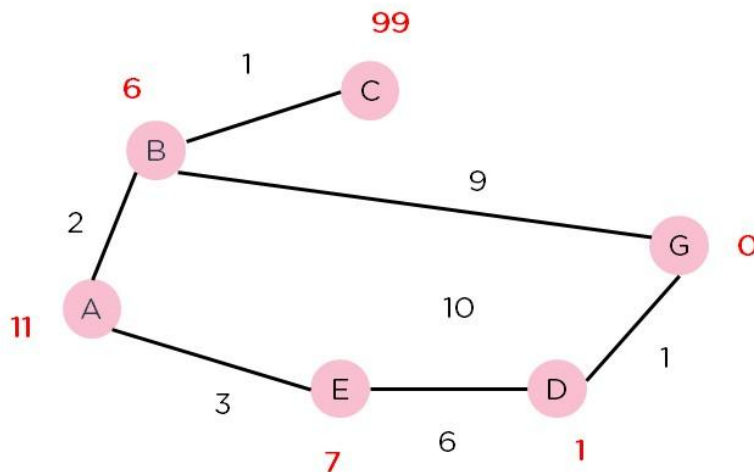
Thus, the program to implement **MINIMAX ALGORITHM** has been successfully executed and verified




EX.No :DATE :A\* SEARCH ALGORITHM

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:**


220701053



---

File Edit View Insert Runtime Tools Help

---

+ Code + Text

---

```

import heapq

# A class to represent the graph
class Graph:
    def __init__(self):
        self.edges = {}          # Stores edges as adjacency lists
        self.h = {}              # Stores heuristic values

    # Add an edge to the graph
    def add_edge(self, from_node, to_node, cost):
        if from_node not in self.edges:
            self.edges[from_node] = []
        self.edges[from_node].append((to_node, cost))

    # Set heuristic value for each node
    def set_heuristic(self, node, heuristic_value):
        self.h[node] = heuristic_value

    # A* algorithm implementation
    def a_star(self, start, goal):
        # Priority queue to store nodes with their f(n) values
        open_list = []
        heapq.heappush(open_list, (0, start))

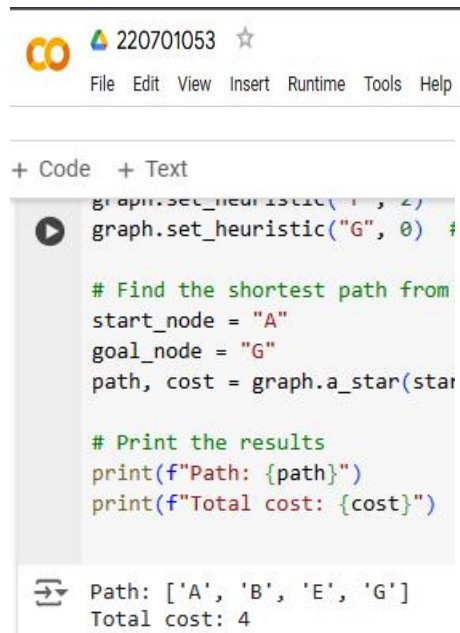
        # Dictionaries to store g(n) values and the optimal path
        g = {start: 0}
        came_from = {start: None}

        while open_list:
            # Pop the node with the lowest f(n) value
            _, current = heapq.heappop(open_list)

            # If the goal is reached, construct the path
            if current == goal:
                path = []
                while current:
                    path.append(current)
                    current = came_from[current]

```

## OUTPUT:



The screenshot shows a code editor interface with a menu bar (File, Edit, View, Insert, Runtime, Tools, Help) and a toolbar with a play button. The code is as follows:

```
graph.set_heuristic("A", 4)
graph.set_heuristic("G", 0)

# Find the shortest path from
start_node = "A"
goal_node = "G"
path, cost = graph.a_star(start_node, goal_node)

# Print the results
print(f"Path: {path}")
print(f"Total cost: {cost}")
```

Below the code, the output is displayed:

```
Path: ['A', 'B', 'E', 'G']
Total cost: 4
```

## RESULT:

Thus, the program implementing A\* SEARCH ALGORITHM has been executed and verified

**EX.NO :****DATE:****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.

**OUTPUT: -**

```
?- woman(mia).
true.

?- playsAirGuitar(mia).
false.

?- party.
true.

?- concert.
ERROR: Unknown procedure: concert/0 (DWIM could not correct goal)
?- ■
```

**KB2:**

happy(yolanda). listens2music(mia).

Listens2music(yolanda):-happy(yolanda). playsAirGuitar(mia):-listens2music(mia).

playsAirGuitar(Yolanda):-listens2music(yolanda).

**OUTPUT: -**

```
?- playsAirGuitar(mia).
true.

?- playsAirGuitar(yolanda).
true.

?- ■
```

**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).

**OUTPUT: -**

```
?- likes(dan,X).
X = sally.

?- married(dan,sally).
true.

?- married(john,brittney).
false.
```

**KB4:** food(burger).

food(sandwich).

food(pizza).

lunch(sandwich).

dinner(pizza).

meal(X):-food(X).

**OUTPUT:**

```
?-
|   food(pizza).
true.

?- meal(X), lunch(X).
X = sandwich ,

?- dinner(sandwich).
false.

?-
```

**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)).

**OUTPUT:**

```
?-
|   owns(john,X).
X = car(chevy).




?- owns(john,_).
true.

?- owns(Who,car(chevy)).
Who = john ,

?- owns(jane,X),sedan(X).
false.

?- owns(jane,X),truck(X).
X = car(chevy).
```

**PROGRAM:**



220701053



---

File Edit View Insert Runtime Tools Help


---

Code
Text

```

KB1 = {
    "woman": ["mia", "jody", "yolanda"],
    "playsAirGuitar": ["jody"],
    "party": True
}

def query_kb1(query):
    if query[0] == "woman" and query[1] in KB1["woman"]:
        return True
    elif query[0] == "playsAirGuitar" and query[1] in KB1["playsAirGuitar"]:
        return True
    elif query[0] == "party" and KB1["party"]:
        return True
    else:
        return False

# Queries for KB1
print("KB1 Query Results:")
print(query_kb1(["woman", "mia"])) # Expected True
print(query_kb1(["playsAirGuitar", "mia"])) # Expected False
print(query_kb1(["party"])) # Expected True
print(query_kb1(["concert"])) # Expected False

# Define KB2 with facts and rules
KB2 = {
    "happy": ["yolanda"],
    "listens2music": ["mia"],
}

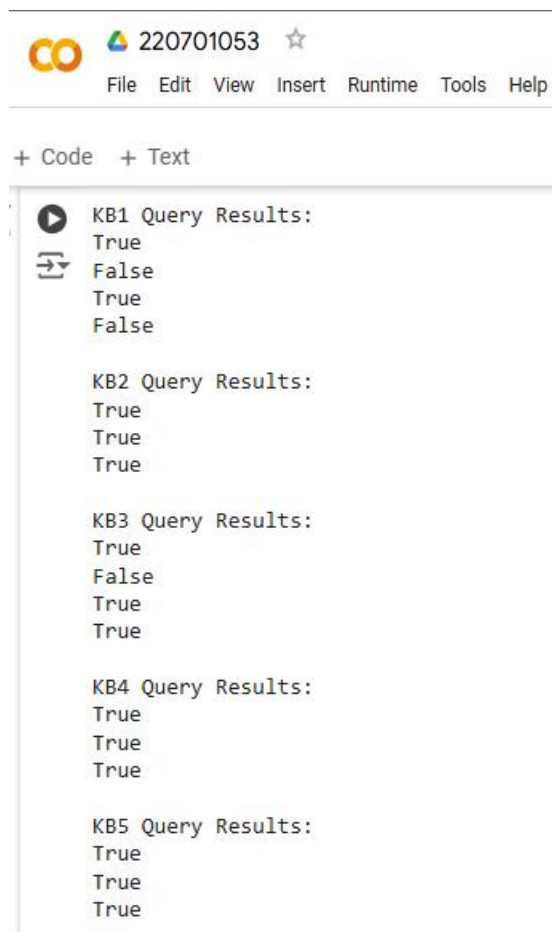
def listens2music(person):
    return person in KB2["listens2music"] or person in KB2["happy"]

def playsAirGuitar(person):
    return listens2music(person)

# Queries for KB2
print("\nKB2 Query Results:")
print(listens2music("mia")) # Expected True
print(playsAirGuitar("mia")) # Expected True

```

## **OUTPUT:**



The screenshot shows a Prolog IDE interface. At the top, there is a logo with the letters 'CO' in orange and a user identifier '220701053' next to a star icon. Below this is a menu bar with the following items: File, Edit, View, Insert, Runtime, Tools, and Help. Under the menu bar, there are two tabs: '+ Code' and '+ Text'. The '+ Text' tab is active, displaying the output of several Prolog queries. The output is organized into five sections, each starting with a query label followed by its results:

- KB1 Query Results:**
  - True
  - False
  - True
  - False
- KB2 Query Results:**
  - True
  - True
  - True
- KB3 Query Results:**
  - True
  - False
  - True
  - True
- KB4 Query Results:**
  - True
  - True
  - True
- KB5 Query Results:**
  - True
  - True
  - True

## **RESULT:**

Thus ,the program implementing Prolog has been executed successfully.

**EX.NO****DATE\_:****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.

**PROGRAM:**



The screenshot shows a Python IDE with a menu bar (File, Edit, View, Insert, Runtime, Tools, Help) and a toolbar with icons for CO, a file icon, and a star. The code is written in a light-themed editor with a dark border. It defines facts for male and female individuals, parent relationships, and functions for family relationships like father, mother, grandfather, grandmother, and brother.

```

# Facts
male = {"peter", "john", "chris", "kevin"}
female = {"betty", "jeny", "lisa", "helen"}

# Parent relationships
parent_of = {
    "chris": ["peter", "betty"],
    "helen": ["peter", "betty"],
    "kevin": ["chris", "lisa"],
    "jeny": ["john", "helen"]
}

# Functions for family relationships

def father(child, person):
    return person in male and person in parent_of and child in parent_of[person]

def mother(child, person):
    return person in female and person in parent_of and child in parent_of[person]

def grandfather(grandchild, person):
    if person in male:
        for parent in parent_of.get(person, []):
            if grandchild in parent_of.get(parent, []):
                return True
    return False

def grandmother(grandchild, person):
    if person in female:
        for parent in parent_of.get(person, []):
            if grandchild in parent_of.get(parent, []):
                return True
    return False

def brother(sibling1, sibling2):
    for parent, children in parent_of.items():
        if sibling1 in children and sibling2 in children and sibling1 != sibling2:
            return sibling1 in male and sibling2 in male
    return False
  
```



**OUTPUT:**

---

 220701053 ☆  
File Edit View Insert Runtime Tools Help  
Code + Text  

```
# Example Queries
print("Father of Peter:", father("peter", "ch
print("Mother of Peter:", mother("peter", "he
print("Grandfather of Peter:", grandfather("p
print("Grandmother of Peter:", grandmother("p
print("Is Chris the brother of Betty?", broth
print("Is Helen the sister of Jeny?", sister(
```

  
 Father of Peter: True  
Mother of Peter: True  
Grandfather of Peter: True  
Grandmother of Peter: True  
Is Chris the brother of Betty? False  
Is Helen the sister of Jeny? False

---

**RESULT:**

Thus, the program implementing prolog family tree has been executed successfully.

**EX.NO :****DATE :**

## **IMPLEMENTING ARTIFICIAL NEURAL NETWORKS FOR AN APPLICATION USING PYTHON - REGRESSION**

### **Regression using Artificial Neural Networks**

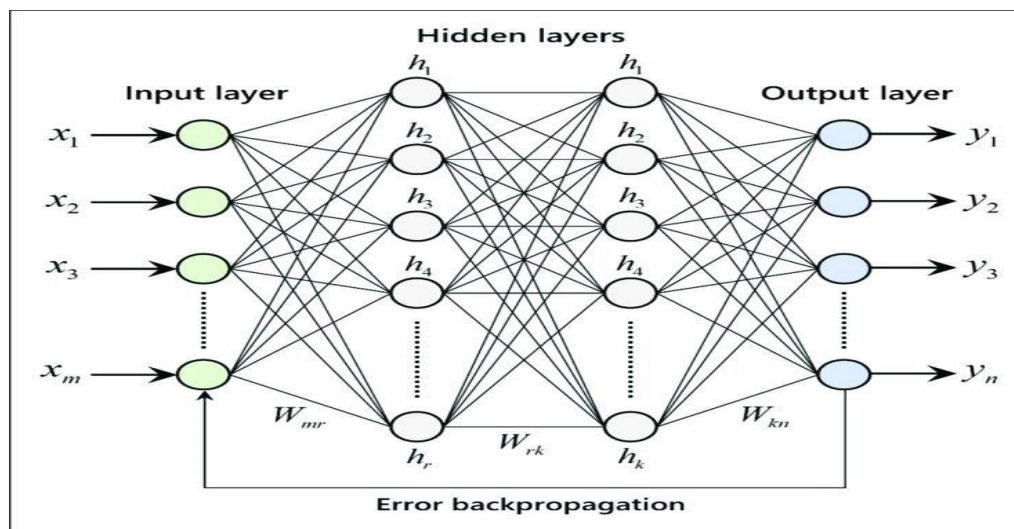
Why do we need to use Artificial Neural Networks for Regression instead of simply using Linear Regression?

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

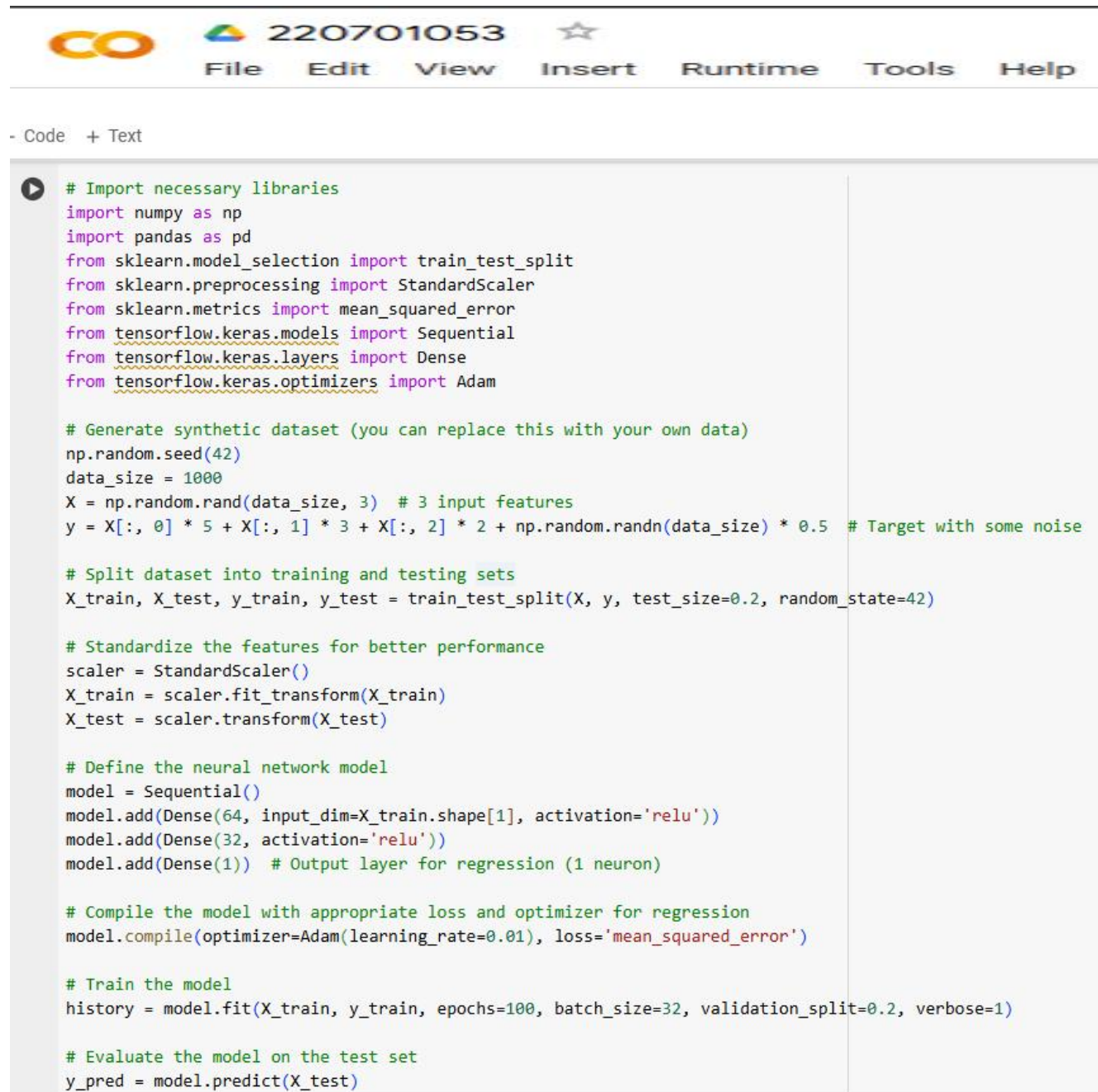
Forward propagation is the process of multiplying weights with each feature and adding them. The bias is also added to the result. Backward propagation is the process of updating the weights in the model.

Backward propagation requires an optimization function and a loss function.

### **AIM :**

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

## PROGRAM



The image shows a Google Colab interface. At the top, there is a header bar with the Colab logo, a user ID '220701053', a star icon, and a menu bar with options: File, Edit, View, Insert, Runtime, Tools, and Help. Below the header, there is a tab labeled '- Code' and a '+ Text' button. The main area contains a Python script for training a neural network model. The script includes imports for numpy, pandas, sklearn, and tensorflow.keras. It generates a synthetic dataset, splits it into training and testing sets, standardizes the features, defines a neural network model with three layers (64, 32, and 1 neurons), compiles it with Adam optimizer and mean squared error loss, trains it for 100 epochs, and finally evaluates it on the test set.

```
# Import necessary libraries
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

# Generate synthetic dataset (you can replace this with your own data)
np.random.seed(42)
data_size = 1000
X = np.random.rand(data_size, 3) # 3 input features
y = X[:, 0] * 5 + X[:, 1] * 3 + X[:, 2] * 2 + np.random.randn(data_size) * 0.5 # Target with some noise

# Split dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize the features for better performance
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

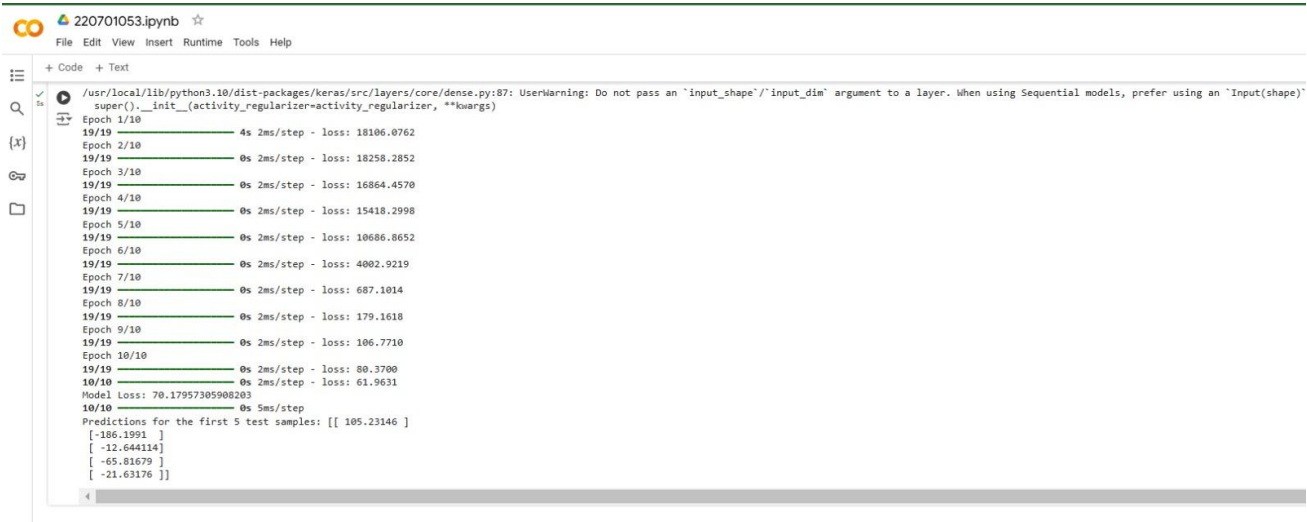
# Define the neural network model
model = Sequential()
model.add(Dense(64, input_dim=X_train.shape[1], activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(1)) # Output layer for regression (1 neuron)

# Compile the model with appropriate loss and optimizer for regression
model.compile(optimizer=Adam(learning_rate=0.01), loss='mean_squared_error')

# Train the model
history = model.fit(X_train, y_train, epochs=100, batch_size=32, validation_split=0.2, verbose=1)

# Evaluate the model on the test set
y_pred = model.predict(X_test)
```

## OUTPUT:



The image shows a Jupyter Notebook interface with a file named '220701053.ipynb'. The notebook contains a single code cell that displays the output of a Keras model training process. The output shows the training progress over 10 epochs, with a loss that decreases from approximately 18106.0762 to 61.9631. The final model loss is 70.17957305908203. The output also includes predictions for the first 5 test samples, which are: [-186.1991], [-12.644114], [-65.81679], and [-21.63176].

```
/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 a layer. When using Sequential models, prefer using an "Input(shape)"
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/10
19/19 ————— 4s 2ms/step - loss: 18106.0762
Epoch 2/10
19/19 ————— 0s 2ms/step - loss: 18258.2852
Epoch 3/10
19/19 ————— 0s 2ms/step - loss: 16864.4570
Epoch 4/10
19/19 ————— 0s 2ms/step - loss: 15418.2998
Epoch 5/10
19/19 ————— 0s 2ms/step - loss: 10686.8652
Epoch 6/10
19/19 ————— 0s 2ms/step - loss: 4002.9219
Epoch 7/10
19/19 ————— 0s 2ms/step - loss: 687.1014
Epoch 8/10
19/19 ————— 0s 2ms/step - loss: 179.1618
Epoch 9/10
19/19 ————— 0s 2ms/step - loss: 106.7710
Epoch 10/10
19/19 ————— 0s 2ms/step - loss: 80.3700
10/10 ————— 0s 2ms/step - loss: 61.9631
Model Loss: 70.17957305908203
10/10 ————— 0s 5ms/step
Predictions for the first 5 test samples: [[ 105.23146 ]
 [-186.1991 ]
 [-12.644114]
 [-65.81679 ]
 [-21.63176 ]]
```

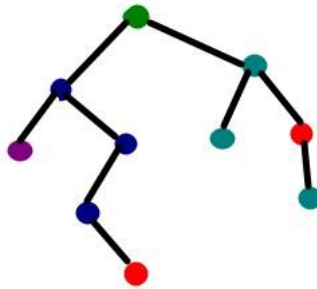
**RESULT:** Thus, the program implementing artificial neural networks for an AN application using python regression has been executed successfully.

**EX.NO :**

**DATE :**

### **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:**


220701053 

File Edit View Insert Runtime Tools Help

Code + Text

```

# Import necessary libraries
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Sample data for gender classification
# Features: [Height (cm), Weight (kg), Shoe Size (EU)]
X = [
    [170, 70, 42],
    [160, 60, 38],
    [180, 80, 44],
    [155, 48, 36],
    [165, 65, 40],
    [175, 75, 43],
    [160, 55, 37],
    [185, 85, 45]
]

# Labels (0 for Female, 1 for Male)
y = [1, 0, 1, 0, 0, 1, 0, 1]

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)

# Create Decision Tree Classifier
clf = DecisionTreeClassifier()

# Train the classifier
clf.fit(X_train, y_train)

# Make predictions on the test set
y_pred = clf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy on test data:", accuracy)

# Display predictions for test data
print("Predictions for test data:", y_pred)

```



**OUTPUT:**

The image shows a Jupyter Notebook interface. At the top, there is a logo and the text "220701053.ipynb" with a star icon. Below this is a menu bar with options: File, Edit, View, Insert, Runtime, Tools, Help, and a status bar indicating "All changes saved". The main area of the notebook is divided into two sections. The top section is labeled "+ Code" and contains a single line of code: "Predicted Gender (0 = Female, 1 = Male): 1". The bottom section is labeled "+ Text" and is currently empty.

**RESULT:**

Thus, the program implementing decision tree classification has been executed successfully.

**EX NO :**

**DATE :**

### **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:**


220701053


File Edit View Insert Runtime Tools Help

Code + Text

```

# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

# Generate synthetic data for clustering
X, y = make_blobs(n_samples=300, centers=4, cluster_std=0.6, random_state=0)

# Visualize the synthetic data
plt.scatter(X[:, 0], X[:, 1], s=30, color='gray')
plt.title("Data Points")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()

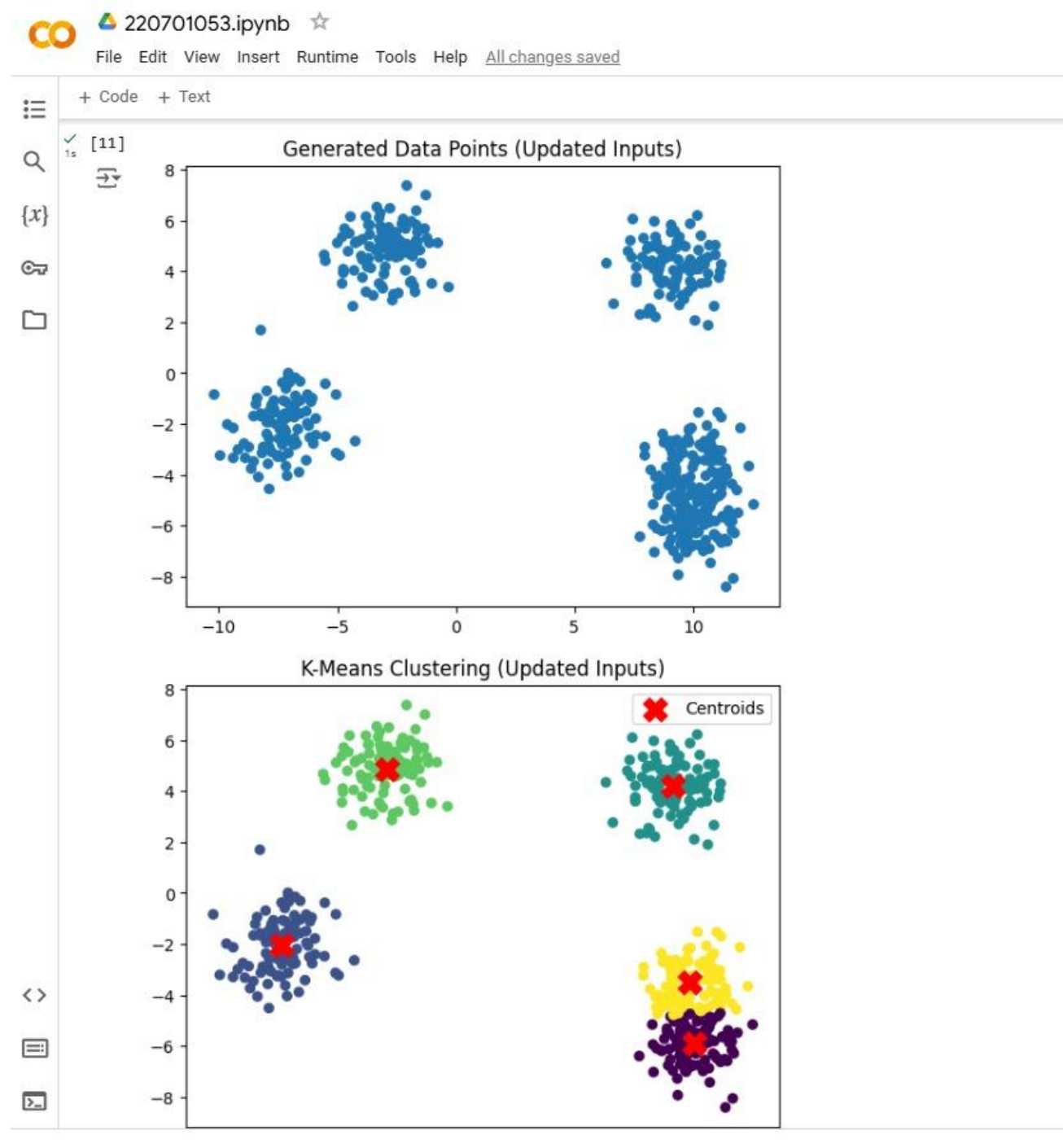
# Use the elbow method to find the optimal number of clusters (K)
inertia = []
K_values = range(1, 11)
for k in K_values:
    kmeans = KMeans(n_clusters=k, random_state=0)
    kmeans.fit(X)
    inertia.append(kmeans.inertia_)

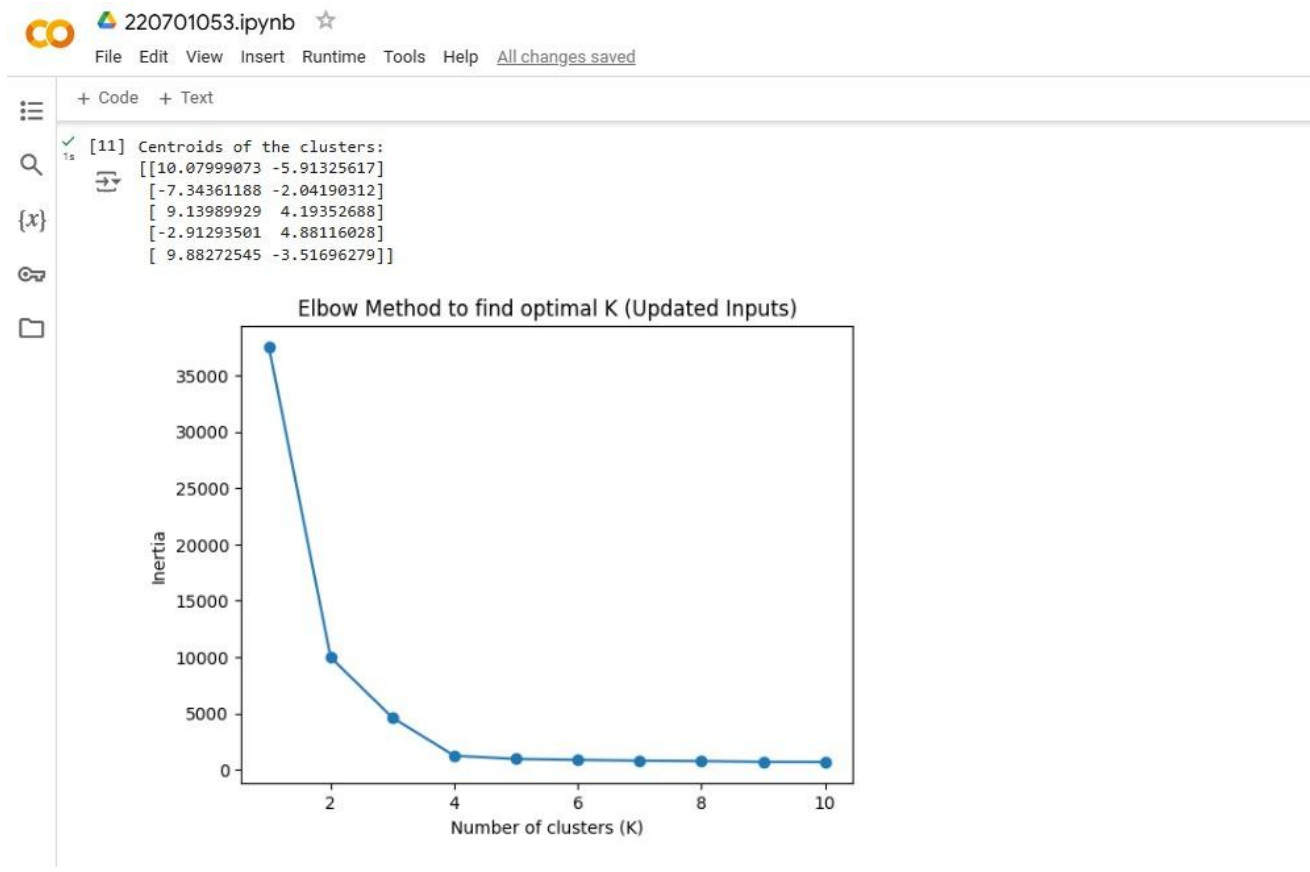
# Plot the elbow curve
plt.plot(K_values, inertia, marker='o')
plt.title("Elbow Method for Optimal K")
plt.xlabel("Number of clusters (K)")
plt.ylabel("Inertia")
plt.show()

# From the elbow plot, choose K=4 (optimal number of clusters for this example)
kmeans = KMeans(n_clusters=4, random_state=0)
y_kmeans = kmeans.fit_predict(X)

# Visualize the clusters
plt.scatter(X[y_kmeans == 0, 0], X[y_kmeans == 0, 1], s=50, c='red', label='Cluster 1')
plt.scatter(X[y_kmeans == 1, 0], X[y_kmeans == 1, 1], s=50, c='blue', label='Cluster 2')

```

**OUTPUT:**



## **RESULT:**

Thus, the program implementing Cluster tree K-means has been executed successfully.