

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY DEPARTMENT OF NETWORKING AND COMMUNICATIONS

**18CSC305J – Artificial Intelligence Lab Record Notebook**

**2021-2022 / Even Semester**

|  |  |
| --- | --- |
| Student Name | Sakshil Verma |
| Register Number | RA1911031010071 |
| Year / Section | B.Tech – 3rd year (L1 Section) |
| Faculty Incharge | Dr.P.Supraja |

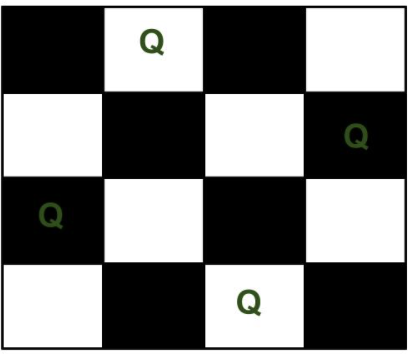
**18CSC305J - ARTIFICIAL INTELLIGENCE LIST OF LAB EXPERIMENTS**

|  |  |
| --- | --- |
| **S.No** | **Name of the Experiment** |
| 1 | Implementation of toy problems |
| 2 | Developing agent programs for real world problems |
| 3 | Implementation of constraint satisfaction problems |
| 4 | Implementation and Analysis of DFS and BFS for same application |
| 5 | Developing Best first search and A\* Algorithm for real world problems |
| 6 | Implementation of uncertain methods for an application |
| 7 | Implementation of unification and resolution for real world problems. |
| 8 | Implementation of learning algorithms for an application |
| 9 | Implementation of NLP programs |
| 10 | Applying deep learning methods to solve an application |

**EX1 : Toy Problem - N Queen Problem**

**Aim:** Solve the N queen toy problem using python.

**Problem Description:** The N Queen is the problem of placing N chess queens on an NxN chessboard so that no two queens attack each other. Given a 4 x 4 chessboard and number the rows and columns of the chessboard 1 through 4.



Since, we have to place 4 queens such as 91 92 93 and 94 on the chessboard, such that no two queens attack each other. In such a conditional each queen must be placed on a different row, i.e., we put queen "i" on row "i."

# Program:

global N N = 4

def printSolution(board): for i in range(N):

for j in range(N):

print (board[i][j], end = " ")

print()

def isSafe(board, row, col): for i in range(col):

if board[row][i] == 1:

return False

for i, j in zip(range(row, -1, -1),

range(col, -1, -1)):

if board[i][j] == 1:

return False

for i, j in zip(range(row, N, 1),

range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solveNQUtil(board, col): if col >= N:

return True for i in range(N):

if isSafe(board, i, col):

board[i][col] = 1

if solveNQUtil(board, col + 1) == True: return True

board[i][col] = 0

return False def solveNQ():

board = [ [0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0],

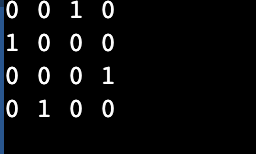
[0, 0, 0, 0] ]

if solveNQUtil(board, 0) == False: print ("Solution does not exist") return False

printSolution(board) return True

solveNQ()

# Output Screenshot:



**Result:** The program was executed successfully such that no two queens attack each other.

# EX2 : Agent Program for Real World Problems

**Aim:** To write an agent program for a Vacuum cleaner world problem.

**Problem Description:** Develop a simple reflex agent program in Python for the vacuum-cleaner world problem. This program defines the States, Goal State, Goal Test, Actions, Transition Model, and Path Cost. For each possible initial state, the program returns a sequence of actions that leads to the goal state, along with the path cost. Generates two test cases.

# Program:

#INSTRUCTIONS

#Enter LOCATION A/B in captial letters

#Enter Status O/1 accordingly where 0 means CLEAN and 1 means DIRTY

def vacuum\_world(): goal\_state = {'A': '0', 'B': '0'}

cost = 0

location\_input = input("Enter Location of Vacuum") status\_input = input("Enter status of " + location\_input) status\_input\_complement = input("Enter status of other room") print("Initial Location Condition" + str(goal\_state))

if location\_input == 'A':

print("Vacuum is placed in Location A") if status\_input == '1':

print("Location A is Dirty.")

goal\_state['A'] = '0'

cost += 1 #cost for suck

print("Cost for CLEANING A " + str(cost)) print("Location A has been Cleaned.")

if status\_input\_complement == '1': print("Location B is Dirty.") print("Moving right to the Location B. ") cost += 1

print("COST for moving RIGHT" + str(cost)) goal\_state['B'] = '0'

cost += 1 #cost for suck print("COST for SUCK " + str(cost)) print("Location B has been Cleaned. ")

else:

print("No action" + str(cost))

print("Location B is already clean.")

if status\_input == '0':

print("Location A is already clean ")

if status\_input\_complement == '1':# if B is Dirty print("Location B is Dirty.")

print("Moving RIGHT to the Location B. ") cost += 1

print("COST for moving RIGHT " + str(cost))

goal\_state['B'] = '0'

cost += 1 #cost for suck print("Cost for SUCK" + str(cost)) print("Location B has been Cleaned. ")

else:

else:

print("No action " + str(cost)) print(cost)

print("Location B is already clean.")

print("Vacuum is placed in location B") if status\_input == '1':

print("Location B is Dirty.") goal\_state['B'] = '0'

cost += 1 # cost for suck

print("COST for CLEANING " + str(cost)) print("Location B has been Cleaned.")

if status\_input\_complement == '1': print("Location A is Dirty.") print("Moving LEFT to the Location A. ") cost += 1 # cost for moving right

print("COST for moving LEFT" + str(cost))

else:

goal\_state['A'] = '0'

cost += 1 # cost for suck print("COST for SUCK " + str(cost)) print("Location A has been Cleaned.")

print(cost)

print("Location B is already clean.")

if status\_input\_complement == '1': # if A is Dirty print("Location A is Dirty.")

print("Moving LEFT to the Location A. ") cost += 1 # cost for moving right

print("COST for moving LEFT " + str(cost)) goal\_state['A'] = '0'

cost += 1 # cost for suck print("Cost for SUCK " + str(cost))

print("Location A has been Cleaned. ") else:

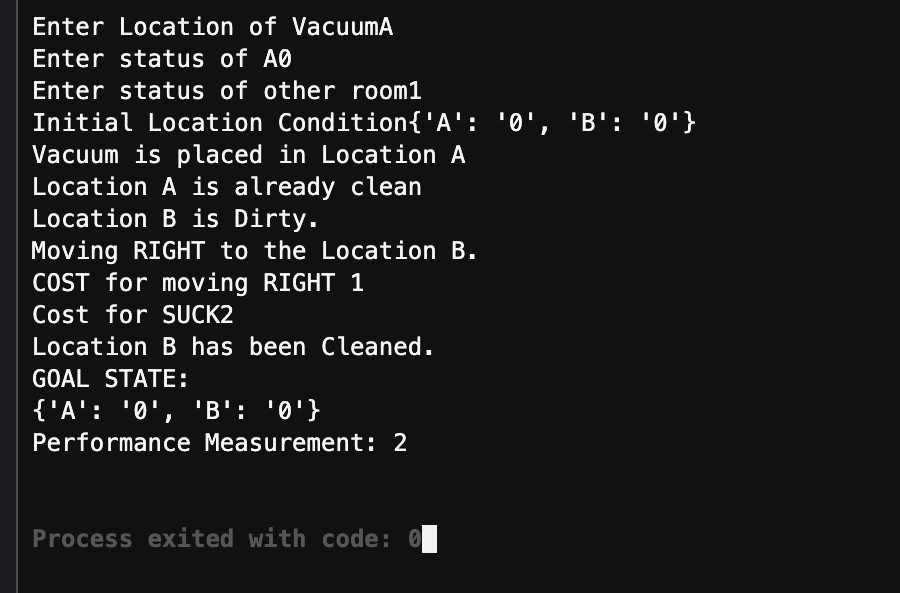
print("No action " + str(cost)) print("Location A is already clean.")

print("GOAL STATE: ")

print(goal\_state)

print("Performance Measurement: " + str(cost)) vacuum\_world()

# Output Screenshot:



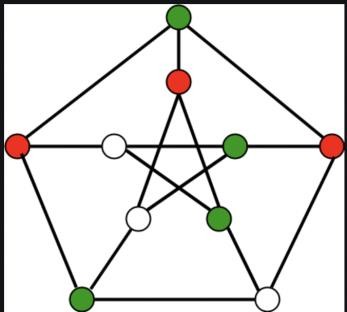
**Result:** The program was executed successfully and it returned a sequence of actions that lead to the goal state, along with the path cost.

# EX3 : Constraint Satisfaction Problem

**Aim:** To write a python program for graph coloring problem using backtracking.

**Problem Description:** Given an undirected graph and a number m, determine if the graph can be coloured with at most m colors such that no two adjacent vertices of the graph are colored with the same color. Here coloring of a graph means the assignment of colors to all vertices.

Following is an example of a graph that can be coloured with 3 different colours.



# Program:

class Graph():

def init (self, vertices): self.V = vertices

self.graph = [[0 for column in range(vertices)]\

for row in range(vertices)]

def isSafe(self, v, colour, c): for i in range(self.V):

if self.graph[v][i] == 1 and colour[i] == c: return False

return True

def graphColourUtil(self, m, colour, v): if v == self.V:

return True

for c in range(1, m + 1):

if self.isSafe(v, colour, c) == True: colour[v] = c

if self.graphColourUtil(m, colour, v + 1) == True: return True

colour[v] = 0 def graphColouring(self, m):

colour = [0] \* self.V

if self.graphColourUtil(m, colour, 0) == None: return False

print ("Solution exist and Following are the assigned colours:") for c in colour:

print (c,end=' ') return True

g = Graph(4)

g.graph = [[0, 1, 1, 1], [1, 0, 1, 0], [1, 1, 0, 1], [1, 0, 1, 0]]

m = 3 g.graphColouring(m)

# Output Screenshot:



**Result:** The program was executed successfully and it returned the colors assigned to the vertices.

# EX4 : Implementation and Analysis of DFS and BFS for an application

**Aim:** Given the root node of a binary search tree, return the sum of values of all nodes with a value in the range [low, high] using depth first and then breadth first search.

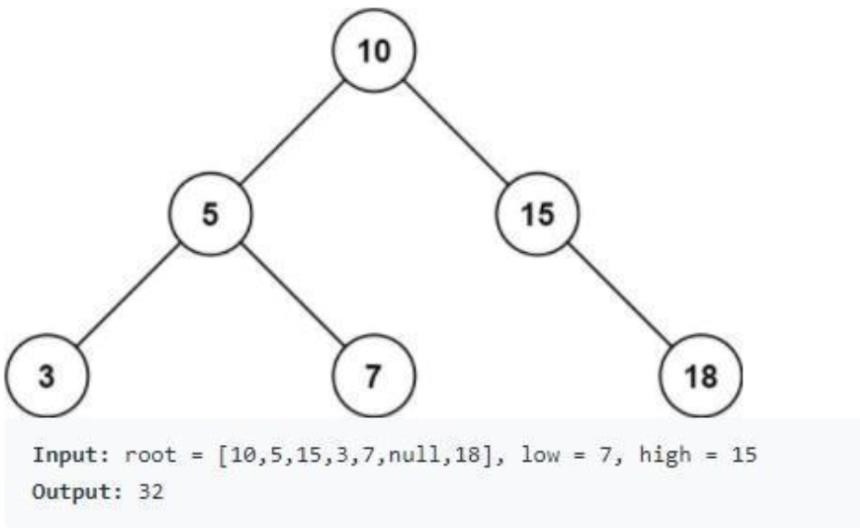
While :

The number of nodes in the tree is in the range [1, 2 \* 104].

1 <= Node.val <= 105

1 <= low <= high <= 105

All Node.value are unique.



## ALGORITHM #1 DFS

## Traverse the tree using a depth first search. Create a stack to store accessed nodes.

If node.value falls outside the range [L, R]

Then only the right branch could have nodes with value inside [L, R]. If Left <= node.value <= Right then Result[0] + = node.value

Else, recursively call the function until all nodes are visited.

ALGORITHM #2 BFS :

Traverse the tree using the breadth first search approach. Maintain a queue and ptr to point toward the current node. If node == None then continue.

If Left <= node.value <= Right the result + = node.value If Left > node.value then queue.append(node.right)

If R < node.value then queue.append(node.left) Repeat till all nodes are visited.

OPTIMIZATION TECHNIQUE:

Time Complexity: O(N), where N is the number of nodes in the tree. Space Complexity: O(N). For the recursive implementation, the recursion will consume additional space in the function call stack. In the worst case, the tree is of chain shape, and we will reach all the way down to the leaf node. For the iterative implementation, essentially we are doing a BFS (Breadth-First Search) traversal, where the stack will contain no more than two levels of the nodes. The maximal number of nodes in a binary tree is N/2.

Therefore, the maximal space needed for the stack would be O(N).

## DEPTH FIRST SEARCH (DFS) CODE :

*#ITERATIVE APPROACH*

class Solution(object):

def rangeSumBST(self, root, L, R): def dfs(node):

if node:

if L <= node.val <= R: self.ans += node.val

if L < node.val: dfs(node.left)

if node.val < R: dfs(node.right)

self.ans = 0 dfs(root) return self.ans

# RECURSIVE APPROACH

def rangeSumBST(root, L, R): ans = 0

stack = [root] while stack:

node = stack.pop() if node:

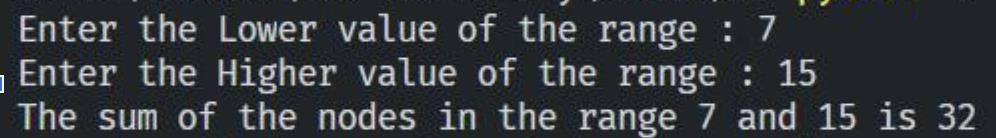
if L <= node.val <= R:

ans += node.val if L < node.val:

stack.append(node.left) if node.val < R:

stack.append(node.right) return ans

## DEPTH FIRST SEARCH OUTPUT :



**BREADTH FIRST (BFS) CODE:**

class TreeNode:

def init (self, val=0, left=None, right=None): self.val = val

self.left = left self.right = right

class Solution(object):

def rangeSumBST(self, root, L, R): if root == None:

return 0

res = 0

q = [root] while q:

next = []

for node in q:

if L <= node.val <= R: res += node.val

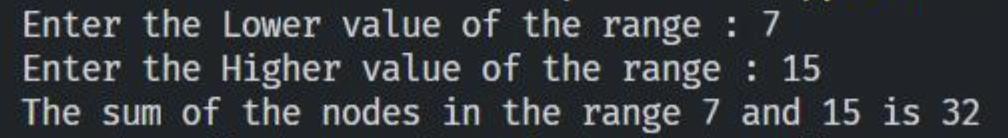
if node.left: next.append(node.left)

if node.right: next.append(node.right)

q = next return res

bst = TreeNode(10, 7, 15) Solution().rangeSumBST(bst,10,7,15)

## BREADTH FIRST OUTPUT:



**RESULT :** Successfully found the sum of nodes in a binary search tree between any given range (min, max) using both depth first search and breadth first search approach.

# EX5 : Implementation and Analysis of BFS and A\* Search

**Aim:** To implement Best First Algorithm and A\* Algorithm using python.

**BEST FIRST SEARCH**

## Description:

In BFS and DFS, when we are at a node, we can consider any of the adjacent as next node. So both BFS and DFS blindly explore paths without considering any cost function. The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore.

## Algorithm:

Define a list, OPEN, consisting solely of a single node, the start node, s. IF the list is empty, return failure.

Remove from the list the node n with the best score (the node where f is the minimum), and move it to a list, CLOSED.

Expand node n.

IF any successor to n is the goal node, return success and the solution (by tracing the path from the goal node to s).

FOR each successor node: 1.apply the evaluation function, f, to the node.

1. IF the node has not been in either list, add it to OPEN.

looping structure by sending the algorithm back to the second step.

## Code:

from queue import PriorityQueue v = 14 graph = [[] for i in range(v)]

def best\_first\_search(source, target, n): visited = [0] \* n visited[0] = True

pq = PriorityQueue() pq.put((0, source)) while pq.empty() == False: u = pq.get()[1]

print(u, end=" ") if u == target:

break

for v, c in graph[u]:

if visited[v] == False: visited[v] = True pq.put((c, v)) print()

def addedge(x, y, cost): graph[x].append((y, cost)) graph[y].append((x, cost))

addedge(0, 1, 3)

addedge(0, 2, 6)

addedge(0, 3, 5)

addedge(1, 4, 9)

addedge(1, 5, 8)

addedge(2, 6, 12)

addedge(2, 7, 14)

addedge(3, 8, 7)

addedge(8, 9, 5)

addedge(8, 10, 6)

addedge(9, 11, 1)

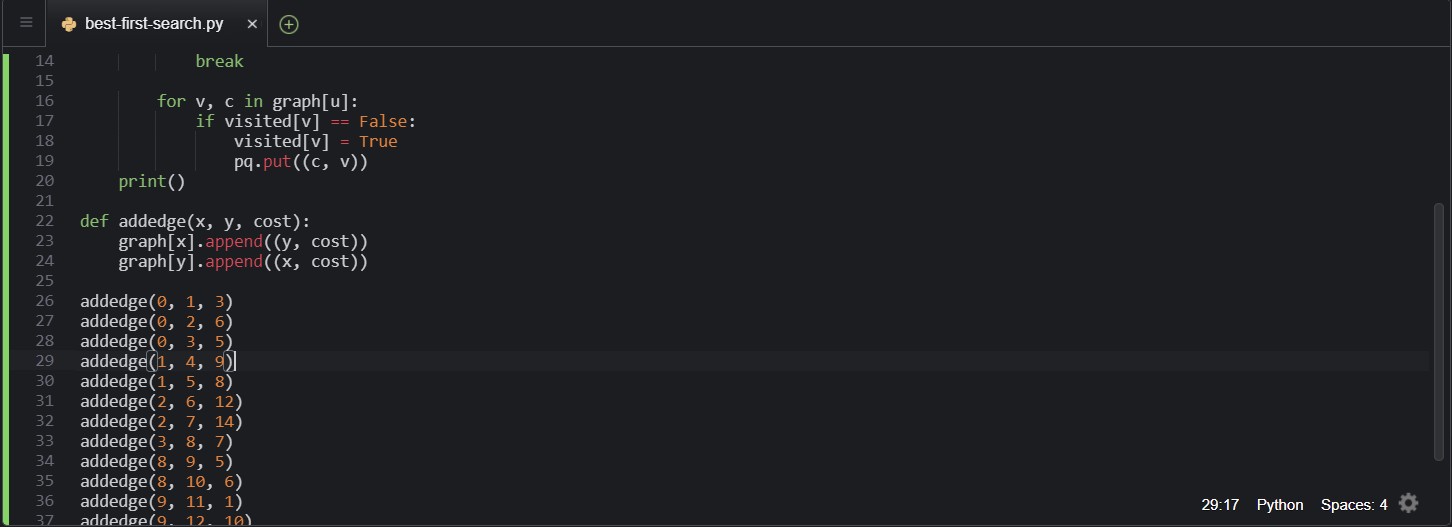
addedge(9, 12, 10)

addedge(9, 13, 2)

source = 0

target = 9 best\_first\_search(source, target, v)

## Output:



## A\* Best First Search

**Description:**

A\* is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

## Code:

def aStarAlgo(start\_node, stop\_node):

open\_set = set(start\_node) closed\_set = set() g = {} #store distance from starting node

parents = {}# parents contains an adjacency map of all nodes

#ditance of starting node from itself is zero g[start\_node] = 0

#start\_node is root node i.e it has no parent nodes #so start\_node is set to its own parent node parents[start\_node] = start\_node

while len(open\_set) > 0: n = None

#node with lowest f() is found for v in open\_set:

if n == None or g[v] + heuristic(v) < g[n] + heuristic(n): n = v

first

if n == stop\_node or Graph\_nodes[n] == None: pass else:

for (m, weight) in get\_neighbors(n):

#nodes 'm' not in first and last set are added to

#n is set its parent

if m not in open\_set and m not in closed\_set: open\_set.add(m) parents[m] = n

g[m] = g[n] + weight

i.e g(m) to the

#for each node m,compare its distance from start

#from start through n node else:

if g[m] > g[n] + weight: #update g(m)

g[m] = g[n] + weight #change parent of m to n parents[m] = n

#if m in closed set,remove and add to open if m in closed\_set: closed\_set.remove(m) open\_set.add(m)

if n == None:

print('Path does not exist!') return None

# if the current node is the stop\_node

# then we begin reconstructin the path from it to the start\_node if n == stop\_node:

path = []

while parents[n] != n: path.append(n)

n = parents[n] path.append(start\_node) path.reverse() print('Path found: {}'.format(path)) return path

# remove n from the open\_list, and add it to closed\_list # because all of his neighbors were inspected open\_set.remove(n)

closed\_set.add(n)

print('Path does not exist!') return None

#define fuction to return neighbor and its distance #from the passed node

def get\_neighbors(v): if v in Graph\_nodes: return Graph\_nodes[v] else:

return None

#for simplicity we ll consider heuristic distances given #and this function returns heuristic distance for all nodes def heuristic(n):

H\_dist = { 'A': 11,

'B': 6,

'C': 99,

‘D’:1,

'E': 7,

'G': 0,

}

return H\_dist[n]

#Describe your graph here Graph\_nodes = { 'A': [('B', 2), ('E', 3)],

'B': [('C', 1),('G', 9)],

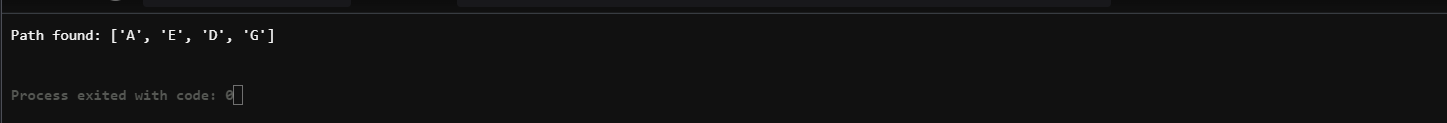
'C': None, 'E': [('D', 6)],

'D': [('G', 1)],

}

aStarAlgo('A', 'G')

## Output:

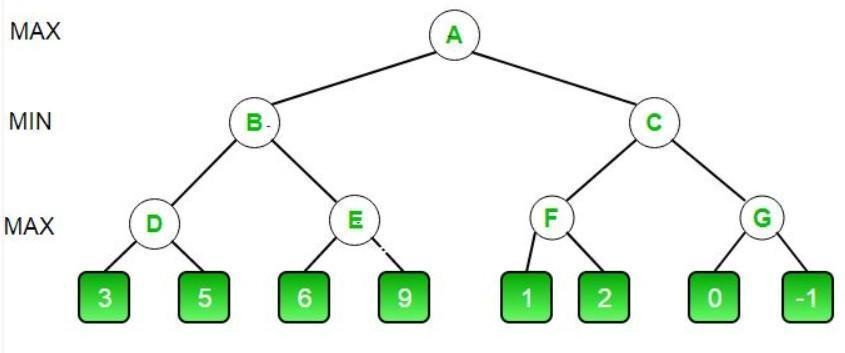


**Result:** Best First Search and A\* algorithm were successfully implemented and executed in AWS using Python3.

# EX6 : MINIMAX ALGORITHM IN ALPHA BETA PRUNING

**Aim:** Developing a mini max algorithm for real world problems.

**PROBLEM :** Find the optimal value in the given tree of integer values in the most optimal way possible under the time complexity O(B^D).



## ALGORITHM MINIMAX APPROACH :

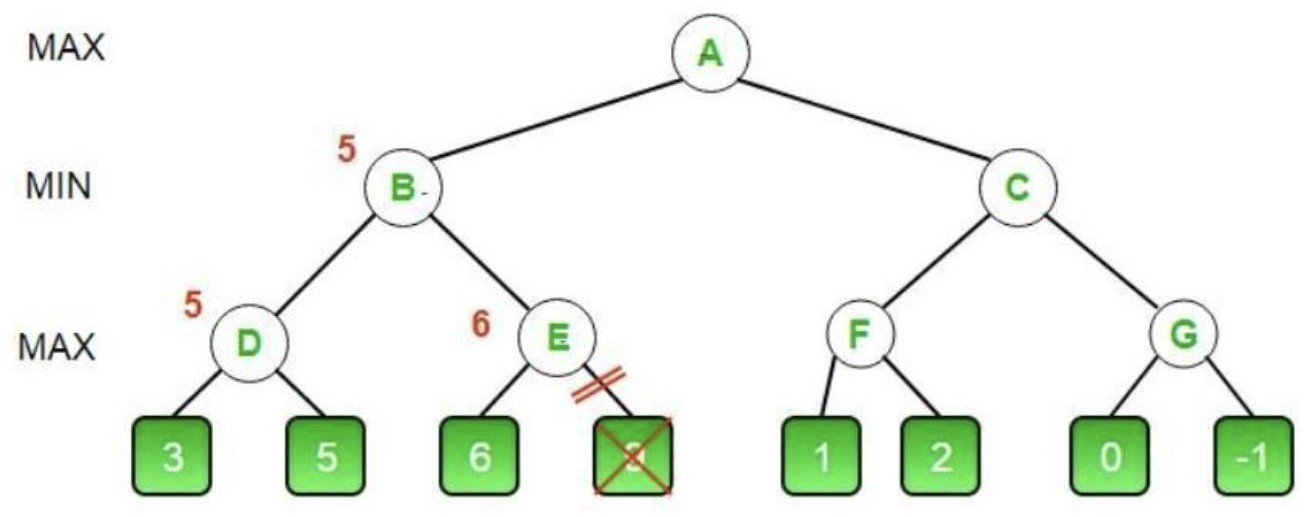
* 1. Start traversing the given tree in top to bottom manner.
  2. If node is a leaf node then return the value of the node.
  3. If isMaximizingPlayer exist then bestVal = -INFINITY
  4. For each child node, value = minimax(node, depth+1, false, alpha, beta)
  5. bestVal = max( bestVal, value) and alpha = max( alpha, bestVal)
  6. If beta <= alpha then stop traversing and return bestVal
  7. Else, bestVal = +INFINITY
  8. For each child node, value = minimax(node, depth+1, true, alpha, beta)
  9. bestVal = min( bestVal, value) and beta = min( beta, bestVal)
  10. if beta <= alpha the stop traversing and return bestVal

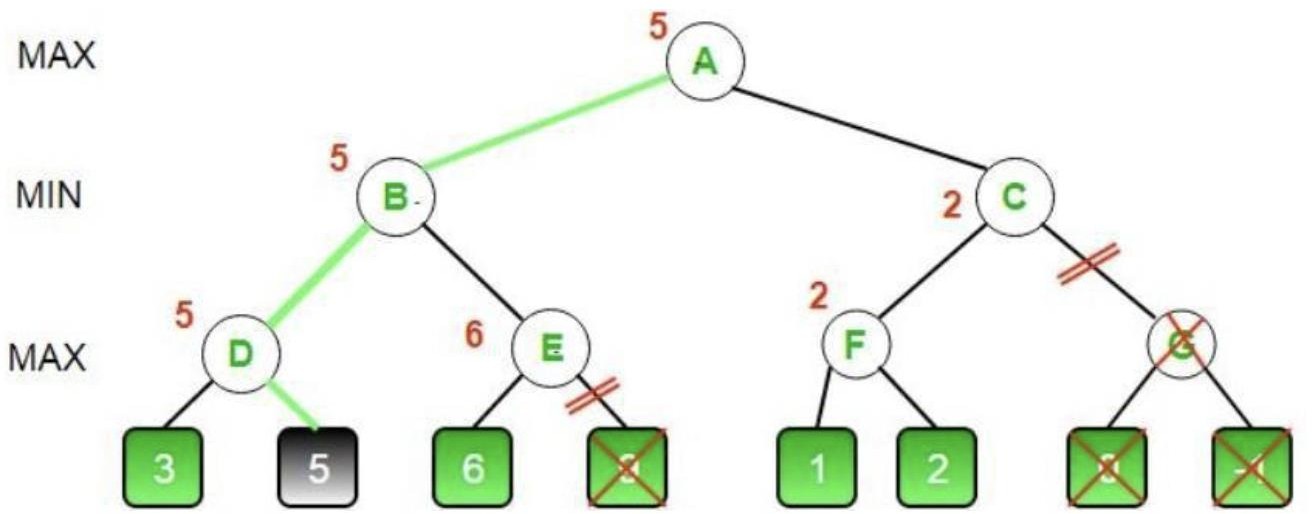
## OPTIMIZATION TECHNIQUE :

Alpha-Beta pruning is not actually a new algorithm, rather an optimization technique for minimax algorithms. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree. It cuts off branches in the game tree which need not be searched because there already exists a better move available. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta. Let’s define the parameters alpha and beta.

**Alpha** is the best value that the **maximizer** currently can guarantee at that level or above.

**Beta** is the best value that the **minimizer** currently can guarantee at that level or above.





## CODE (MINIMAX ALGORITHM) :

MAX, MIN = 1000, -1000

def minimax(depth, nodeIndex, maximizingPlayer,

values, alpha, beta):

if depth == 3:

return values[nodeIndex] if maximizingPlayer:

best = MIN

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i,

False, values, alpha, beta)

best = max(best, val) alpha = max(alpha, best) if beta <= alpha:

break

else:

return best

best = MAX

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i, True, values, alpha, beta)

best = min(best, val) beta = min(beta, best) if beta <= alpha:

break

return best

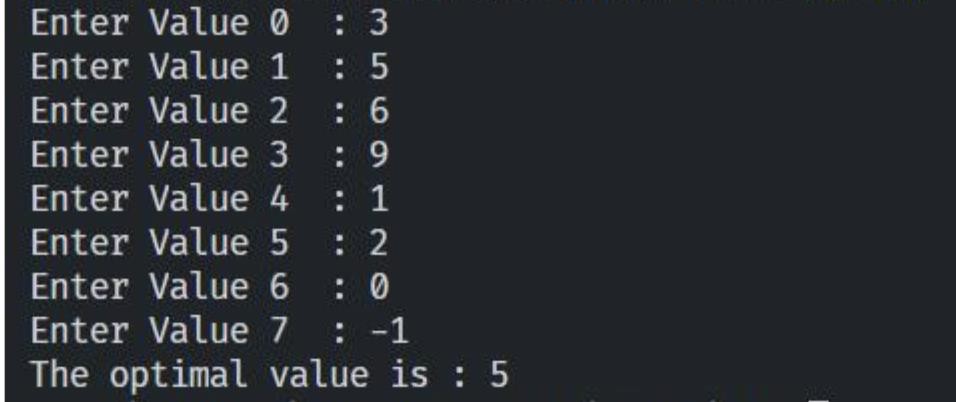
if name == " main ": values = []

for i in range(0, 8):

x = int(input(f"Enter Value {i} : ")) values.append(x)

print ("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))

## OUTPUT :



**RESULT :** The Optimal value of the given tree successfully found using Minimax Algorithm with Alpha Beta Pruning in time complexity O(B^D).

# EX7 : IMPLEMENTATION OF UNIFICATION AND RESOLUTION

**PROBLEM STATEMENT :** Developing an optimized technique using an appropriate artificial intelligence algorithm to solve the Unification and Resolution.

## ALGORITHM :

* + 1. function PL-RESOLUTION (KB, Q) returns true or false inputs: KB,
    2. the knowledge base, group of sentences/facts in propositional logic
    3. Q, the query, a sentence in propositional logic
    4. clauses → the set of clauses in the CNF representation of KB ^ Q new → { }
    5. loop do for each Ci, Cj in clauses do
    6. resolvents → PL-RESOLVE (Ci, Cj)
    7. if resolvents contains the empty clause the return true
    8. new → new union resolvents
    9. if new is a subset of clauses then return false
    10. clauses → clauses union true

## OPTIMIZATION TECHNIQUE:

Resolution basically works by using the principle of proof by contradiction. To find the conclusion we should negate the conclusion. Then the resolution rule is applied to the resulting clauses. Each clause that contains complementary literals is resolved to produce a2. new clause, which can be added to the set of facts (if it is not already present). This process continues until one of the two things happen:•There are no new clauses that can be added. An application of the resolution rule derives the empty clauseAn empty clause shows that the negation of the conclusion is a complete

contradiction, hence the negation of the conclusion is invalid or false or the assertion is completely valid or true.

1. Convert the given statements in Predicate/Propositional Logic
2. Convert these statements into Conjunctive Normal Form
3. Negate the Conclusion (Proof by Contradiction)
4. Resolve using a Resolution Tree (Unification)

## CODE UNIFICATION :

def get\_index\_comma(string): index\_list = list() par\_count = 0

for i in range(len(string)):

if string[i] == ',' and par\_count == 0: index\_list.append(i)

elif string[i] == '(':

par\_count += 1 elif string[i] ==

')':

par\_count -= 1 return index\_list

def is\_variable(expr): for i in expr:

if i == '(' or i == ')': return False

return True

def process\_expression(expr): expr = expr.replace(' ', '')

index = None

or i in range(len(expr)): if expr[i] == '(':

index = i break

predicate\_symbol = expr[:index]

expr = expr.replace(predicate\_symbol, '') expr = expr[1:len(expr) - 1]

arg\_list = list()

indices = get\_index\_comma(expr)

if len(indices) == 0: arg\_list.append(expr)

else:

arg\_list.append(expr[:indices[0]]) for i, j in zip(indices, indices[1:]): arg\_list.append(expr[i + 1:j])

arg\_list.append(expr[indices[len(indices) - 1] + 1:])

return predicate\_symbol, arg\_list

def get\_arg\_list(expr):

\_, arg\_list = process\_expression(expr)

flag = True while flag:

flag = False

for i in arg\_list:

if not is\_variable(i): flag = True

\_, tmp = process\_expression(i) or j in tmp:

if j not in arg\_list: arg\_list.append(j)

arg\_list.remove(i) return arg\_list

def check\_occurs(var, expr): arg\_list = get\_arg\_list(expr) if var in arg\_list:

return True return False

def unify(expr1, expr2):

if is\_variable(expr1) and is\_variable(expr2): if expr1 == expr2:

return 'Null' else:

return False

elif is\_variable(expr1) and not is\_variable(expr2): if check\_occurs(expr1, expr2):

return False else:

tmp = str(expr2) + '/' + str(expr1) return tmp

elif not is\_variable(expr1) and is\_variable(expr2): if check\_occurs(expr2, expr1):

return False

else:

tmp = str(expr1) + '/' + str(expr2) return tmp

else:

predicate\_symbol\_1, arg\_list\_1 = process\_expression(expr1) predicate\_symbol\_2, arg\_list\_2 = process\_expression(expr2)

# Step 2

if predicate\_symbol\_1 != predicate\_symbol\_2: return False

# Step 3

elif len(arg\_list\_1) != len(arg\_list\_2): return False

else:

# Step 4: Create substitution list sub\_list = list()

# Step 5:

for i in range(len(arg\_list\_1)):

tmp = unify(arg\_list\_1[i], arg\_list\_2[i])

if not tmp: return False

elif tmp == 'Null': pass

else:

if type(tmp) == list: for j in tmp:

sub\_list.append(j)

else:

Stepsub\_list.append(tmp)

return sub\_list

if name == ' main ': # f1 = 'Q(a, g(x, a), f(y))'

# f2 = 'Q(a, g(f(b), a), x)' f1 = input('f1 : ')

f2 = input('f2 : ') result = unify(f1, f2)

if not result:

print('The process of Unification failed!') else:

print('The process of Unification successful!') print(result)

## OUTPUT UNIFICATION :

## CODE RESOLUTION :

import copy import time

class Parameter: variable\_count = 1

def init (self, name=None): if name:

self.type = "Constant" self.name = name

else:

self.type = "Variable"

self.name = "v" + str(Parameter.variable\_count)

Parameter.variable\_count += 1

def isConstant(self):

return self.type == "Constant"

def unify(self, type\_, name): self.type = type\_ self.name = name

def eq (self, other):

return self.name == other.name

def str (self): return self.name

class Predicate:

def init (self, name, params): self.name = name

self.params = params

def eq (self, other):

return self.name == other.name and all(a == b for a, b in zip(self.params, other.params))

def str (self):

return self.name + "(" + ",".join(str(x) for x in self.params) + ")"

def getNegatedPredicate(self):

return Predicate(negatePredicate(self.name), self.params)

class Sentence: sentence\_count = 0

def init (self, string):

self.sentence\_index = Sentence.sentence\_count Sentence.sentence\_count += 1

self.predicates = [] self.variable\_map = {} local = {}

for predicate in string.split("|"):

name = predicate[:predicate.find("(")] params = []

for param in predicate[predicate.find("(") + 1: predicate.find(")")].split(","): if param[0].islower():

if param not in local: # Variable local[param] = Parameter()

self.variable\_map[local[param].name] = local[param] new\_param = local[param]

else:

new\_param = Parameter(param) self.variable\_map[param] = new\_param

params.append(new\_param) self.predicates.append(Predicate(name, params))

def getPredicates(self):

return [predicate.name for predicate in self.predicates]

def findPredicates(self, name):

return [predicate for predicate in self.predicates if predicate.name == name]

def removePredicate(self, predicate): self.predicates.remove(predicate)

for key, val in self.variable\_map.items(): if not val:

self.variable\_map.pop(key)

def containsVariable(self):

return any(not param.isConstant() for param in self.variable\_map.values())

def eq (self, other):

if len(self.predicates) == 1 and self.predicates[0] == other: return True

return False

def str (self):

return "".join([str(predicate) for predicate in self.predicates]

class KB:

def init (self, inputSentences):

self.inputSentences = [x.replace(" ", "") for x in inputSentences] self.sentences = []

self.sentence\_map = {}

def prepareKB(self): self.convertSentencesToCNF()

for sentence\_string in self.inputSentences: sentence = Sentence(sentence\_string) for predicate in sentence.getPredicates():

self.sentence\_map[predicate] = self.sentence\_map.get( predicate, []) + [sentence]

def convertSentencesToCNF(self):

for sentenceIdx in range(len(self.inputSentences)):

# Do negation of the Premise and add them as literal if "=>" in self.inputSentences[sentenceIdx]:

self.inputSentences[sentenceIdx] = negateAntecedent( self.inputSentences[sentenceIdx])

def askQueries(self, queryList): results = []

for query in queryList:

negatedQuery = Sentence(negatePredicate(query.replace(" ", ""))) negatedPredicate = negatedQuery.predicates[0] prev\_sentence\_map = copy.deepcopy(self.sentence\_map) self.sentence\_map[negatedPredicate.name] = self.sentence\_map.get(

negatedPredicate.name, []) + [negatedQuery] elf.timeLimit = time.time() + 40

try:

result = self.resolve([negatedPredicate], [

False]\*(len(self.inputSentences) + 1))

except:

result = False

self.sentence\_map = prev\_sentence\_map if result:

results.append("TRUE") else:

results.append("FALSE") return results

def resolve(self, queryStack, visited, depth=0): if time.time() > self.timeLimit:

raise Exception if queryStack:

query = queryStack.pop(-1)

negatedQuery = query.getNegatedPredicate() queryPredicateName = negatedQuery.name

if queryPredicateName not in self.sentence\_map: return False

else:

queryPredicate = negatedQuery

for kb\_sentence in self.sentence\_map[queryPredicateName]:

if not visited[kb\_sentence.sentence\_index]:

for kbPredicate in kb\_sentence.findPredicates(queryPredicateName):

anUnify, substitution = performUnification( copy.deepcopy(queryPredicate),

copy.deepcopy(kbPredicate))

if canUnify:

newSentence = copy.deepcopy(kb\_sentence) newSentence.removePredicate(kbPredicate) newQueryStack = copy.deepcopy(queryStack)

if substitution:

for old, new in substitution.items():

if old in newSentence.variable\_map:

parameter = newSentence.variable\_map[old] newSentence.variable\_map.pop(old) parameter.unify(

"Variable" if new[0].islower() else "Constant", new) newSentence.variable\_map[new] = parameter

for predicate in newQueryStack:

for index, param in enumerate(predicate.params): if param.name in substitution:

new = substitution[param.name] predicate.params[index].unify(

"Variable" if new[0].islower() else "Constant", new)

for predicate in newSentence.predicates: newQueryStack.append(predicate)

new\_visited = copy.deepcopy(visited)

if kb\_sentence.containsVariable() and len(kb\_sentence.predicates) > 1:

new\_visited[kb\_sentence.sentence\_index] = True

if self.resolve(newQueryStack, new\_visited, depth + 1): eturn True

return False return True

def performUnification(queryPredicate, kbPredicate): substitution = {}

if queryPredicate == kbPredicate: return True, {}

else:

for query, kb in zip(queryPredicate.params, kbPredicate.params):

if query == kb: continue

if kb.isConstant():

if not query.isConstant():

if query.name not in substitution: substitution[query.name] = kb.name

elif substitution[query.name] != kb.name: return False, {}

query.unify("Constant", kb.name) else:

return False, {}

else:

if not query.isConstant():

if kb.name not in substitution: substitution[kb.name] = query.name

elif substitution[kb.name] != query.name:

return False, {} kb.unify("Variable", query.name)

else:

if kb.name not in substitution: substitution[kb.name] = query.name

elif substitution[kb.name] != query.name:

return False, {} return True, substitution

def negatePredicate(predicate):

return predicate[1:] if predicate[0] == "~" else "~" + predicate

def negateAntecedent(sentence):

antecedent = sentence[:sentence.find("=>")] premise = []

for predicate in antecedent.split("&"): premise.append(negatePredicate(predicate))

premise.append(sentence[sentence.find("=>") + 2:]) return "|".join(premise)

def getInput(filename):

with open(filename, "r") as file: noOfQueries = int(file.readline().strip())

inputQueries = [file.readline().strip() for \_ in range(noOfQueries)] noOfSentences = int(file.readline().strip()) inputSentences = [file.readline().strip()

for \_ in range(noOfSentences)]

return inputQueries, inputSentences

def printOutput(filename, results): print(results)

with open(filename, "w") as file: for line in results:

file.write(line) file.write("\n")

file.close()

if name == ' main ':

inputQueries\_, inputSentences\_ = getInput('input.txt') knowledgeBase = KB(inputSentences\_) knowledgeBase.prepareKB()

results\_ = knowledgeBase.askQueries(inputQueries\_)

printOutput("output.txt", results\_)

## INPUT RESOLUTION :

6

A(Alice)

~A(Alice) Z(Zig)

~Z(Zig) G(Golf)

~G(Good) 10

A(x) => B(x)

B(x) => C(x)

C(x) => D(x)

D(x) => E(x)

E(x) => A(x) G(g)

H(h)

I(i)

J(j)

X(x)

## OUTPUT RESOLUTION :



**RESULT :** Developed Unification and Resolution Algorithm in Python for solving logical problems.

# EX 8 : IMPLEMENTATION OF LEARNING ALGORITHMS FOR AN APPLICATION

# Aim:

To implement a supervised learning algorithm.

# Problem Description:

Linear Regression is a machine learning algorithm based on supervised learning. It performs a regression task. Regression models a target prediction value based on independent variables. It is mostly used for finding out the relationship between variables and forecasting.

# Program:

import numpy as np

import matplotlib.pyplot as plt import pandas as pd

import seaborn as sns

%matplotlib inline

companies=pd.read\_csv('sample\_data/LinearRegressionExample.csv') x=companies.iloc[:,:-1].values

y=companies.iloc[:,4].values companies.head()

print(x) print(y)

sns.heatmap(companies.corr())

from sklearn.preprocessing import LabelEncoder, OneHotEncoder from sklearn.compose import ColumnTransformer

#Encode State Column labelencoder = LabelEncoder()

x[:,3] = labelencoder.fit\_transform(x[:,3])

ct = ColumnTransformer([("State", OneHotEncoder(), [3])], remainder = 'passthrough') x = ct.fit\_transform(x)

print(x)

x=x[:,1:]

print(x)

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test=train\_test\_split(x,y,test\_size=0.2,random\_state=0)

from sklearn.linear\_model import LinearRegression model\_fit=LinearRegression() model\_fit.fit(x\_train,y\_train)

y\_pred=model\_fit.predict(x\_test) print(y\_pred)

print(model\_fit.coef\_) print(model\_fit.intercept\_)

from sklearn.metrics import r2\_score r2\_score(y\_test, y\_pred)

x\_state = companies.State y\_profit = companies.Profit

from sklearn.preprocessing import LabelEncoder labelencoder = LabelEncoder()

x\_state = labelencoder.fit\_transform(x\_state) print(x\_state)

x\_state = np.array(x\_state).reshape(-1,1) y\_profit = np.array(y\_profit).reshape(-1,1)

print(x\_state.shape) print(y\_profit.shape)

X\_train\_1, X\_test\_1, Y\_train\_1, Y\_test\_1 = train\_test\_split(x\_state, y\_profit, test\_size = 0.2, random\_state=5)

print(X\_train\_1.shape) print(X\_test\_1.shape) print(Y\_train\_1.shape) print(Y\_test\_1.shape)

from sklearn.metrics import mean\_squared\_error reg\_1 = LinearRegression()

reg\_1.fit(X\_train\_1, Y\_train\_1)

y\_train\_predict\_1 = reg\_1.predict(X\_train\_1)

rmse = (np.sqrt(mean\_squared\_error(Y\_train\_1, y\_train\_predict\_1))) r2 = round(reg\_1.score(X\_train\_1, Y\_train\_1),2)

print("The model performance for training set") print('RMSE is {}'.format(rmse))

print('R2 score is {}'.format(r2)) print("\n")

y\_pred\_1 = reg\_1.predict(X\_test\_1)

rmse = (np.sqrt(mean\_squared\_error(Y\_test\_1, y\_pred\_1))) r2 = round(reg\_1.score(X\_test\_1, Y\_test\_1),2)

print("The model performance for testing set") print(" ") print("Root Mean Squared Error: {}".format(rmse)) print("R^2: {}".format(r2))

print("\n")

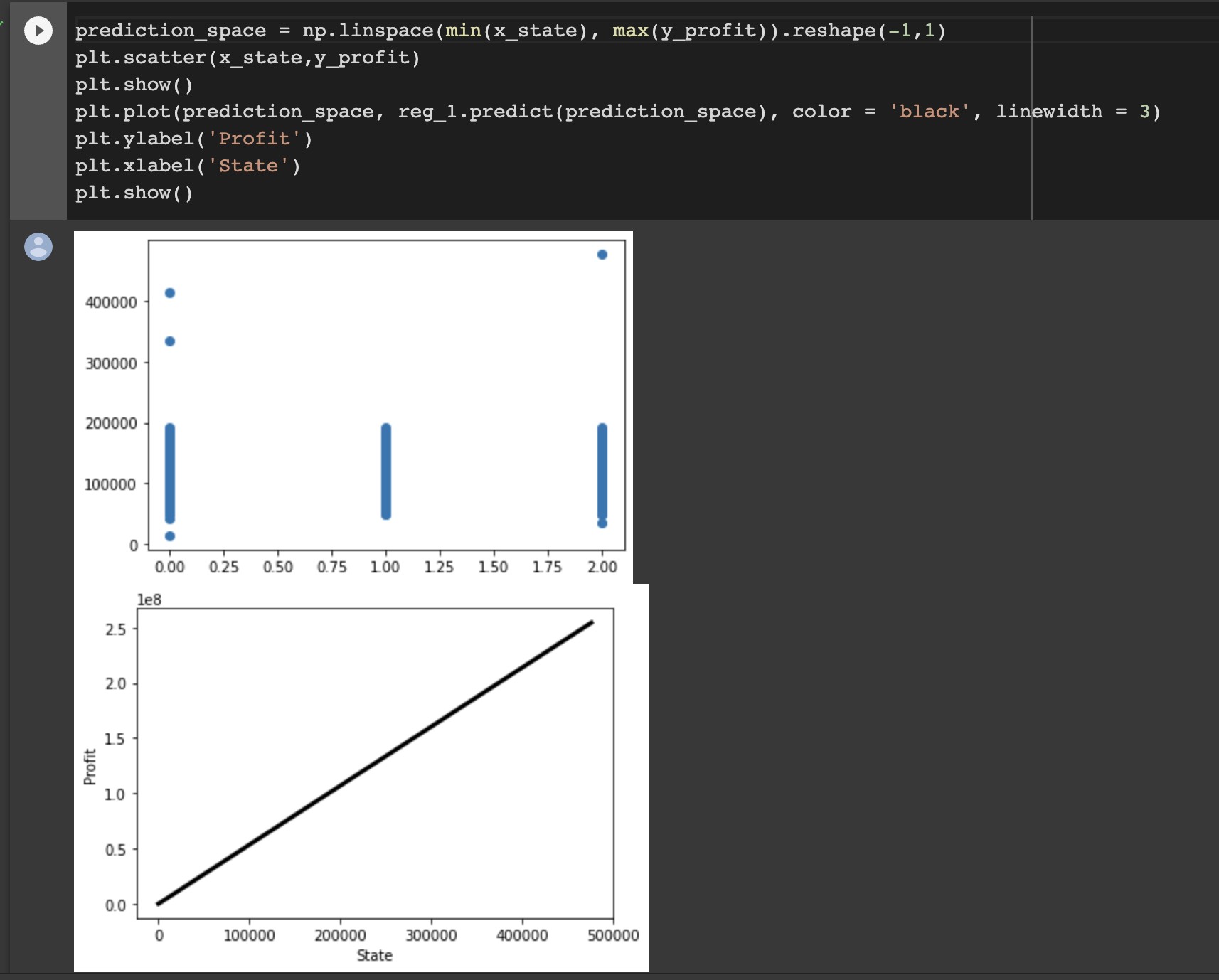
prediction\_space = np.linspace(min(x\_state), max(y\_profit)).reshape(-1,1) plt.scatter(x\_state,y\_profit)

plt.show()

plt.plot(prediction\_space, reg\_1.predict(prediction\_space), color = 'black', linewidth = 3) plt.ylabel('Profit')

plt.xlabel('State') plt.show()

# Output Screenshot:



**Result:** The program was executed successfully.

# EX 9 : IMPLEMENTATION OF NLP PROGRAM TEXT CLASSIFICATION

**Aim:**

Solve NLP program for text classification

## Problem Description:

Text clarification is the process of categorizing the text into a group of words. By using NLP, text classification can automatically analyze text and then assign a set of predefined tags or categories based on its context. NLP is used for sentiment analysis, topic detection, and language detection.

# Program:

import pandas as pd import numpy as np

from nltk.tokenize import word\_tokenize from nltk import pos\_tag

from nltk.corpus import stopwords

from nltk.stem import WordNetLemmatizer from sklearn.preprocessing import LabelEncoder from collections import defaultdict

from nltk.corpus import wordnet as wn

from sklearn.feature\_extraction.text import TfidfVectorizer from sklearn import model\_selection, naive\_bayes, svm from sklearn.metrics import accuracy\_score np.random.seed(500)

Corpus = pd.read\_csv(r"https://raw.githubusercontent.com/Gunjitbedi/Text- Classification/master/corpus.csv",encoding='latin-1')

# Step - a : Remove blank rows if any.

Corpus['text'].dropna(inplace=True)

# Step - b : Change all the text to lower case. This is required as python interprets 'dog' and 'DOG' differently

Corpus['text'] = [entry.lower() for entry in Corpus['text']]

# Step - c : Tokenization : In this each entry in the corpus will be broken into set of words print(Corpus['text'])

import nltk nltk.download('punkt')

# Step - c : Tokenization : In this each entry in the corpus will be broken into set of words Corpus['text']= [word\_tokenize(entry) for entry in Corpus['text']]

print(Corpus['text']) import nltk nltk.download('wordnet')

# Step - d : Remove Stop words, Non-Numeric and perfom Word Stemming/Lemmenting.

# WordNetLemmatizer requires Pos tags to understand if the word is noun or verb or adjective etc. By default it is set to Noun

tag\_map = defaultdict(lambda : wn.NOUN) tag\_map['J'] = wn.ADJ

tag\_map['V'] = wn.VERB tag\_map['R'] = wn.ADV import nltk

nltk.download('averaged\_perceptron\_tagger') import nltk

nltk.download('stopwords')

for index,entry in enumerate(Corpus['text']):

# Declaring Empty List to store the words that follow the rules for this step Final\_words = []

# Initializing WordNetLemmatizer() word\_Lemmatized = WordNetLemmatizer()

# pos\_tag function below will provide the 'tag' i.e if the word is Noun(N) or Verb(V) or something else.

for word, tag in pos\_tag(entry):

# Below condition is to check for Stop words and consider only alphabets if word not in stopwords.words('english') and word.isalpha():

word\_Final = word\_Lemmatized.lemmatize(word,tag\_map[tag[0]]) Final\_words.append(word\_Final)

# The final processed set of words for each iteration will be stored in 'text\_final' Corpus.loc[index,'text\_final'] = str(Final\_words)

print(Corpus['text\_final'])

Train\_X, Test\_X, Train\_Y, Test\_Y = model\_selection.train\_test\_split(Corpus['text\_final'],Corpus['label'],test\_size=0.3)

Encoder = LabelEncoder()

Train\_Y = Encoder.fit\_transform(Train\_Y) Test\_Y = Encoder.fit\_transform(Test\_Y) Tfidf\_vect = TfidfVectorizer(max\_features=5000) Tfidf\_vect.fit(Corpus['text\_final'])

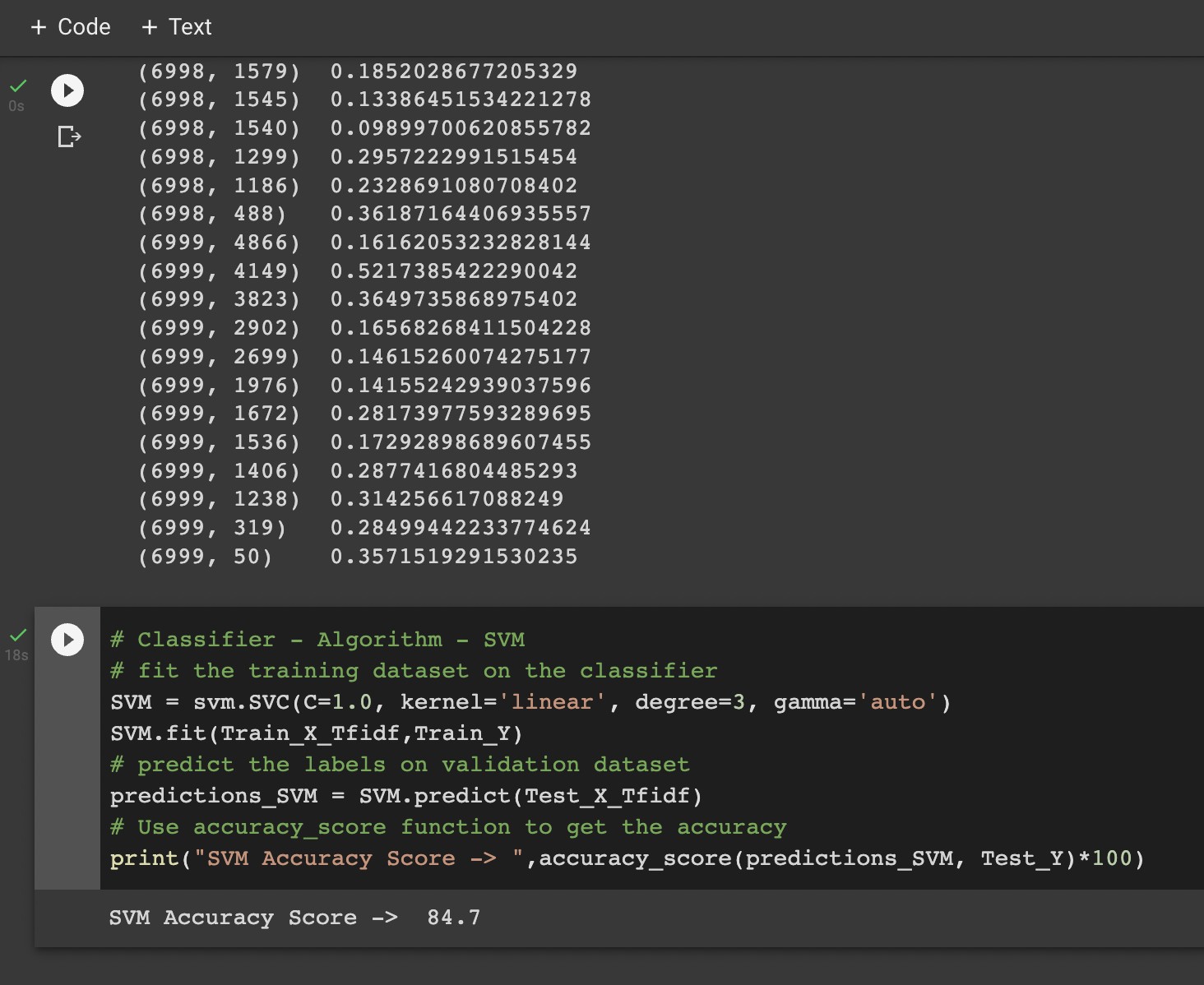
Train\_X\_Tfidf = Tfidf\_vect.transform(Train\_X) Test\_X\_Tfidf = Tfidf\_vect.transform(Test\_X) print(Tfidf\_vect.vocabulary\_) print(Train\_X\_Tfidf)

SVM = svm.SVC(C=1.0, kernel='linear', degree=3, gamma='auto') SVM.fit(Train\_X\_Tfidf,Train\_Y)

# predict the labels on validation dataset predictions\_SVM = SVM.predict(Test\_X\_Tfidf) # Use accuracy\_score function to get the accuracy

print("SVM Accuracy Score -> ",accuracy\_score(predictions\_SVM, Test\_Y)\*100)

# Output Screenshot:



**Result:** The program was executed successfully and accuracy was found.

# EX10 : APPLYING DEEP LEARNING METHODS TO SOLVE AN APPLICATION - CNN ALGORITHM

**Aim:** To write a convolutional neural network algorithm program using deep learning.

**Problem Description: I**n deep learning, a convolutional neural network (CNN/ConvNet) is a class of deep neural networks, most commonly applied to analyze visual imagery.

# Program:

import tensorflow as tf

from tensorflow.keras import datasets, layers, models import matplotlib.pyplot as plt

(train\_images, train\_labels), (test\_images, test\_labels) = datasets.cifar10.load\_data()

# Normalize pixel values to be between 0 and 1

train\_images, test\_images = train\_images / 255.0, test\_images / 255.0

class\_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

plt.figure(figsize=(10,10)) for i in range(25):

plt.subplot(5,5,i+1) plt.xticks([])

plt.yticks([]) plt.grid(False)

plt.imshow(train\_images[i])

# The CIFAR labels happen to be arrays, # which is why you need the extra index plt.xlabel(class\_names[train\_labels[i][0]])

plt.show()

model = models.Sequential()

model.add(layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(32, 32, 3)))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.summary()

model.add(layers.Flatten()) model.add(layers.Dense(64, activation='relu')) model.add(layers.Dense(10))

model.summary()

model.compile(optimizer='adam', loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True), metrics=['accuracy'])

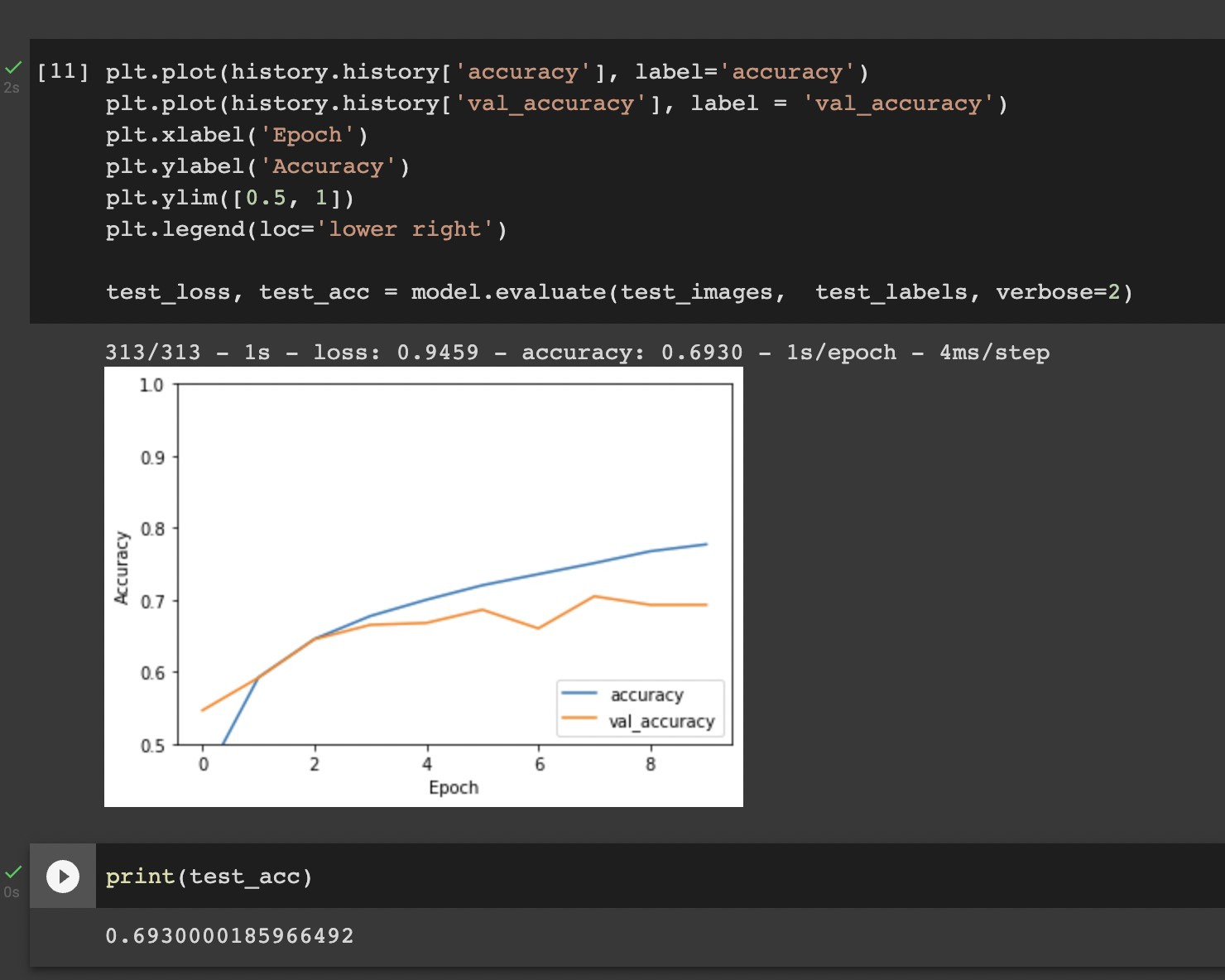
history = model.fit(train\_images, train\_labels, epochs=10, validation\_data=(test\_images, test\_labels))

plt.plot(history.history['accuracy'], label='accuracy') plt.plot(history.history['val\_accuracy'], label = 'val\_accuracy') plt.xlabel('Epoch')

plt.ylabel('Accuracy') plt.ylim([0.5, 1]) plt.legend(loc='lower right')

test\_loss, test\_acc = model.evaluate(test\_images, test\_labels, verbose=2) print(test\_acc)

# Output Screenshot:



**Result:** The program was executed successfully and accuracy was found.