**Exp.no.1 IMPLEMENTATION OF BFS**

**Aim:**

To implement a program to execute the breadth first search algorithm.

**Algorithm:**

1.Start

2.From the given graph,pick any node.

3.Visit the adjacent unvisited vertex, mark it as visited and display it

4. Insert the node displayed in a queue to keep track of the visited nodes.

5. If there are no remaining adjacent vertices left, remove the first vertex from the queue.

6.Repeat from step2 until the queue is empty or the desired node is found.

7.Stop

**Program:**

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

visited = []

queue = []

def bfs(visited, graph, node):

visited.append(node)

queue.append(node)

while queue:

s = queue.pop(0)

print (s, end = " ")

for neighbour in graph[s]:

if neighbour not in visited:

visited.append(neighbour)

queue.append(neighbour)

bfs(visited, graph, 'A')

**Output:**

A B C D E F

**Result:**

Thus the program has been executed successfully.

**Ex.no.2 IMPLEMENTATION OF DFS**

**Aim:**

To implement a program to execute the depth first search algorithm.

**Algorithm:**

1.Start

2.From the given graph, pick any node.

3.Visit the adjacent unvisited vertex, mark it as visited and recur on all its adjacent nodes.

4. Insert the node displayed in a queue to keep track of the visited nodes.

5. If there are no remaining adjacent vertices left, remove the first vertex from the queue.

6.Repeat from step2 until the queue is empty or the desired node is found.

**Program:**

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

visited = set()

def dfs(visited, graph, node):

if node not in visited:

print (node)

visited.add(node)

for neighbour in graph[node]:

dfs(visited, graph, neighbour)

dfs(visited, graph, 'A')

**Output:**

A B D E F C

**Result:**

Thus the program has been executed successfully.

**Exp.no 3 WATER JUG PROBLEM**

**Aim:**

To implement a program to solve the water jug problem using python programming language.

**Algorithm:**

1. Start
2. Initially both the jugs are empty and we need to reach the goal state of having just 2 liters of water in one jug.
3. We use a queue to the maintain the states until the length of the queue is greater than 0 do
4. Pop the topmost element as current state and check whether this state is already visited or not
   1. If it is visited we continue
   2. It does not meet jug constraints then we continue
5. If it is not then we append that to the solution and mark the state as visited
6. If we reach the solution then fill the final state otherwise continue the process for intermediate states
7. Print the solution path

**Program:**

from collections import deque

def BFS(a, b, target):

m = {}

isSolvable = False

path = []

q = deque()

q.append((0, 0))

while (len(q) > 0):

u = q.popleft()

if ((u[0], u[1]) in m):

continue

if ((u[0] > a or u[1] > b or u[0] < 0 or u[1] < 0)):

continue

path.append([u[0], u[1]])

m[(u[0], u[1])] = 1

if (u[0] == target or u[1] == target):

isSolvable = True

if (u[0] == target):

if (u[1] != 0):

path.append([u[0], 0])

else:

if (u[0] != 0):

path.append([0, u[1]])

sz = len(path)

for i in range(sz):

print("(", path[i][0], ",", path[i][1], ")")

break

q.append([u[0], b])

q.append([a, u[1]])

for ap in range(max(a, b) + 1):

c = u[0] + ap

d = u[1] - ap

if (c == a or (d == 0 and d >= 0)):

q.append([c, d])

c = u[0] - ap

d = u[1] + ap

if ((c == 0 and c >= 0) or d == b):

q.append([c, d])

q.append([a, 0])

q.append([0, b])

if (not isSolvable):

print("No solution")

Jug1, Jug2, target = 4, 3, 2

print("Path from initial state to solution state :")

BFS(Jug1, Jug2, target)

**Output:**

Path from initial state to solution state :

( 0 , 0 )

( 0 , 3 )

( 4 , 0 )

( 4 , 3 )

( 3 , 0 )

( 1 , 3 )

( 3 , 3 )

( 4 , 2 )

( 0 , 2 )

**Result:** Thus we have successfully found the solution for the water jug problem.

**Exp.no.4 N-QUEEN PROBLEM SOLUTIONS**

**Aim:**

To write a program to find the solutions of n queens problem.

**Algorithm:**

1. Start

2. Get the number of queens to be placed on the chessboard from the user.

3. Create a n\*n matrix with all elements set to 0.

4. If a queen can be placed in the row safely then mark the row and column number as the solution and return true else unmark the row and try with other rows.

5. If there are no possible ways to place the queens then print the same n\*n matrix with all elements 0.

6. Display the output.

7. Stop

**Program:**

print ("Enter the number of queens")

N = int(input())

board = [[0]\*N for \_ in range(N)]

def is\_attack(i, j):

for k in range(0,N):

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

return True

for k in range(0,N):

for l in range(0,N):

if (k+l==i+j) or (k-l==i-j):

if board[k][l]==1:

return True

return False

def N\_queen(n):

if n==0:

return True

for i in range(0,N):

for j in range(0,N):

if (not(is\_attack(i,j))) and (board[i][j]!=1):

board[i][j] = 1

if N\_queen(n-1)==True:

return True

board[i][j] = 0

return False

N\_queen(N)

for i in board:

print (i)

**Output:**

Enter the number of queens

8

[1, 0, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 1, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 0, 1]

[0, 0, 0, 0, 0, 1, 0, 0]

[0, 0, 1, 0, 0, 0, 0, 0]

[0, 0, 0, 0, 0, 0, 1, 0]

[0, 1, 0, 0, 0, 0, 0, 0]

[0, 0, 0, 1, 0, 0, 0, 0]

**Result:**

Thus , we have successfully executed the program to find the n queens problem.

**Exp.no.5 A\* ALGORITHM**

**Aim:**

To implement a program to execute the A\* algorithm.

**Algorithm:**

1.Start

2.Define a list OPEN.Initially, OPEN consists solely of a single node, the start node S.

3.If the list is empty, return failure and exit.

4.Remove node n with the smallest value of f(n) from OPEN and move it to list CLOSED.If node n is a goal state, return success and exit.

5. Expand node n

6. If any successor to n is the goal node, return success and the solution by tracing the path from goal node to S.Otherwise, go to Step-07.

7. For each successor node, Apply the evaluation function f to the node.If the node has not been in either list, add it to OPEN.

8. Go back to Step-03

**Program:**

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

g[start\_node] = 0

parents[start\_node] = start\_node

while len(open\_set) > 0:

n = None

for v in open\_set:

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

n = v

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

pass

else:

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

if m not in open\_set and m not in closed\_set:

open\_set.add(m)

parents[m] = n

g[m] = g[n] + weight

else:

if g[m] > g[n] + weight:

g[m] = g[n] + weight

parents[m] = n

if m in closed\_set:

closed\_set.remove(m)

open\_set.add(m)

if n == None:

print('Path does not exist!')

return None

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

open\_set.remove(n)

closed\_set.add(n)

print('Path does not exist!')

return None

def get\_neighbors(v):

if v in Graph\_nodes:

return Graph\_nodes[v]

else:

return None

**Output:**

Path found: ['A', 'F', 'G', 'I', 'J']

**Result:**

Thus the program has been executed successfully.

**Exp No : 6 MINIMAX ALGORITHM**

**Aim :**

To perform minimax algorithm on a tree using python

**Algorithm :**

1. Start
2. If the current depth is equal to the target depth, return the scores of the Index.
3. If maxTurn is true, return the maximum value of the current values to each upper node respectively. Else go to 4
4. Return the minimum value of the current values to each upper node respectively.
5. This function goes recursively until the end is reached.
6. Stop

**Program :**

import math

def minimax (curDepth, nodeIndex,maxTurn, scores,targetDepth):

if (curDepth == targetDepth):

return scores[nodeIndex]

if (maxTurn):

return max(minimax(curDepth + 1, nodeIndex \* 2,False, scores, targetDepth),

minimax(curDepth + 1, nodeIndex \* 2 + 1,False, scores, targetDepth))

else:

return min(minimax(curDepth + 1, nodeIndex \* 2,True, scores, targetDepth),

minimax(curDepth + 1, nodeIndex \* 2 + 1,True, scores, targetDepth))

scores = [3, 5, 2, 9, 12, 5, 23, 23]

treeDepth = math.log(len(scores), 2)

print("The optimal value is : ", end = "")

print(minimax(0, 0, True, scores, treeDepth))

**Output:**

The optimal value is: 12

**Result :**

Thus we have successfully implemented the minimax algorithm in python.

**Exp No : 7 INTRODUCTION TO PROLOG**

**Aim:**

To understand the working and implementation of prolog

**Working:**

Prolog (programming in logic) is one of the most widely used programming languages in artificial

intelligence research. As opposed to imperative languages such as C or Java (which also happens to be

object-oriented) it is a declarative programming language. That means, when implementing the solution

to a problem, instead of specifying how to achieve a certain goal in a certain situation, we specify what

the situation (rules and facts) and the goal (query) are and let the Prolog interpreter derive the solution

for us. Prolog is very useful in some problem areas, such as artificial intelligence, natural language

processing, databases, . . . , but pretty useless in others, such as graphics or numerical algorithms.

**Prolog Syntax:**

Terms

The central data structure in Prolog is that of a term. There are terms of four kinds:

atoms, numbers, variables, and compound terms. Atoms and numbers are sometimes grouped together

and called atomic terms.

Atoms. Atoms are usually strings made up of lower- and uppercase letters, digits, and the underscore,

starting with a lowercase letter. The following are all valid Prolog atoms:

elephant, b, abcXYZ, x\_123, another\_pint\_for\_me\_please

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

underscore. Examples: X, Elephant, \_4711, X\_1\_2, MyVariable, \_

The last one of the above examples (the single underscore) constitutes a special case. It is called the

anonymous variable and is used when the value of a variable is of no particular interest. Multiple

occurrences of the anonymous variable in one expression are assumed to be distinct, i.e. their values

don’t necessarily have to be the same.

Compound terms. Compound terms are made up of a functor (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. The following are some examples for compound terms:

is\_bigger(horse, X), f(g(X, \_), 7), ’My Functor’(dog)

It’s important not to put any blank characters between the functor and the opening parentheses, or

Prolog won’t understand what you’re trying to say. In other places, however, spaces can be very helpful

for making programs more readable. The sets of compound terms and atoms together form the set of

Prolog predicates. A term that doesn’t contain any variables is called a ground term.

Clauses, Programs and Queries

In the introductory example we have already seen how Prolog programs are made up of facts and rules.

Facts and rules are also called clauses.

Facts. A fact is a predicate followed by a dot. Examples:

bigger(whale, \_).

life\_is\_beautiful.

The intuitive meaning of a fact is that we define a certain instance of a relation as being true.

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

commas). Head and body are separated by the sign :- and, like every Prolog expression, a rule has to be

terminated by a dot. Examples:

is\_smaller(X, Y) :- is\_bigger(Y, X).

aunt(Aunt, Child) :- sister(Aunt, Parent),parent(Parent, Child).

The intuitive meaning of a rule is that the goal expressed by its head is true, if we (or

rather the Prolog system) can show that all of the expressions (subgoals) in the rule’s

body are true.

Programs. A Prolog program is a sequence of clauses.

Queries. After compilation a Prolog program is run by submitting queries to the interpreter. A query has

the same structure as the body of a rule, i.e. it is a sequence of predicates separated by commas and

terminated by a dot. They can be entered at the Prolog prompt, which in most implementations looks

something like this: ?-. When writing about queries we often include the ?-. Examples:

?- is\_bigger(elephant, donkey).

?- small(X), green(X), slimy(X).

Intuitively, when submitting a query like the last example, we ask Prolog whether all its predicates are

provably true, or in other words whether there is an X such that small(X), green(X), and slimy(X) are all

True

**Example 1:**

likes.pl

delicious(cakes).

delicious(pickles).

likes(priya,Food) :-delicious(Food).

Queries:

1. ?- likes(priya,cakes).

true.

2. ?- delicious(Food).

Food = cakes ;

Food = pickles.

**Example 2:**

likes(mary,food).

likes(mary,wine).

likes(john,wine).

likes(john,mary).

**Queries:**

| ?- likes(mary,food).

yes.

| ?- likes(john,wine).

yes.

| ?- likes(john,food).

no.

**Result:**

Prolog’s understanding and implementation have been successfully understood.

**EXP NO 8 FAMILY TREE GENERATION USING PROLOG**

**Aim :**

To generate a family tree using prolog

**Algorithm :**

1. Start.
2. Define the gender of each person.
3. Assign parents for every person
4. Define all the relationships in a family using the parents assignment and individual definition of each persson.
5. Check the relationships.
6. Stop

**Program :**

male(dev).

male(jonathan).

male(karthik).

male(gowtham).

male(manas).

male(madin).

female(ziya).

female(diya).

female(ishitha).

female(jessy).

female(mia).

parent(mia,dev).

parent(mia,ishitha).

parent(madin,dev).

parent(madin,ishitha).

parent(jonathan,madin).

parent(jonathan,jessy).

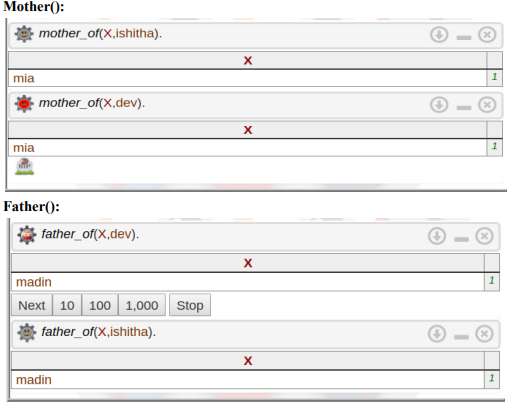
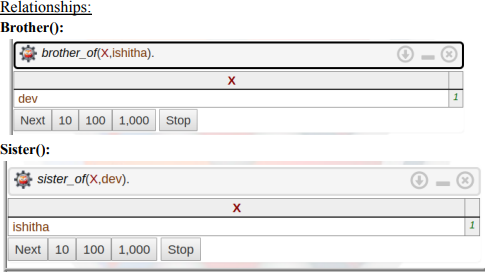
parent(diya,madin).

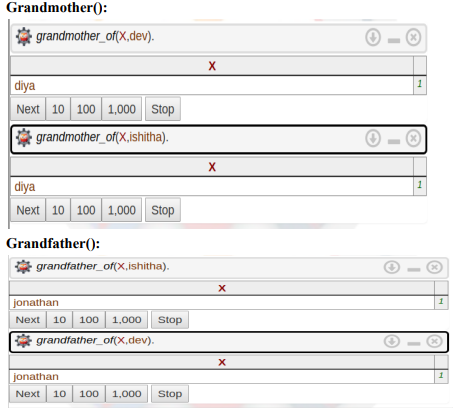
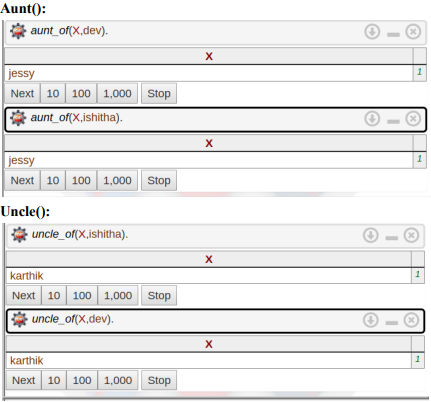
parent(diya,jessy).

parent(jessy,ziya).

parent(karthik,ziya). brother\_of(X,Y):-male(X),parent(F,X),parent(M,X),parent(F,Y),parent(M,Y). sister\_of(X,Y):-female(X),parent(F,X),parent(M,X),parent(F,Y),parent(M,Y). mother\_of(X,Y):-female(X),parent(X,Y). father\_of(X,Y):-male(X),parent(X,Y). aunt\_of(X,Y):-female(X),parent(Z,Y),sister\_of(X,Z). uncle\_of(X,Y):-male(X),parent(Z,Y),sister\_of(F,Z),parent(F,C),parent(X,C). grandmother\_of(X,Y):-female(X),parent(X,Z),parent(Z,Y). grandfather\_of(X,Y):-male(X),parent(X,Z),parent(Z,Y).

**Output :**





**Result :**

Thus we have successfully generated a family tree using prolog.

**Ex.no.9 Unification and Resolution using Prolog**

**Unification:**

Unification is a process of making two different logical atomic expressions identical by finding a substitution. Unification depends on the substitution process. It takes two literals as input and makes them identical using substitution.In logic and computer science, unification is an algorithmic process of solving equations between symbolic expressions. ... If higher-order variables, that is, variables representing functions, are allowed in an expression, the process is called higher-order unification, otherwise first-order unification.

**Resolution:**

Resolution yields a complete inference algorithm when coupled with any complete search algorithm.Resolution makes use of the inference rules. Resolution performs deductive inference.Resolution uses proof by contradiction. One can perform Resolution from a Knowledge Base. A Knowledge Base is a collection of facts or one can even call it a database with all facts.

**Algorithm:**

**Unification:**

1.It takes two literals as input and makes them identical using substitution.

2.Let Ψ1 and Ψ2 be two atomic sentences and 𝜎 be a unifier such that, **Ψ1𝜎 = Ψ2𝜎**, then it can be expressed as **UNIFY(Ψ1, Ψ2)**.

**Resolution:**

1.Conversion of facts into first-order logic.

2.Convert FOL statements into CNF

3.Negate the statement which needs to prove (proof by contradiction)

4.Draw resolution graph (unification).

**Program:**

**Unification:**

?- X = X.

?- bob = X.

?- bob = robert.

**Resolution:**

q :- p, z

q ∨~(p∧z)

q ∨~p∨~z

**Result:**

Hence we have successfully implemented unification and resolution using prolog.

**Exp No. 10 Decision Tree Implementation**

**Aim:**

To implementDecision Tree using python

**Algorithm:**

1. Start at the start node
2. On each iteration, iterate through the very unused attribute of the set S and calculates Entropy(H) and Information gain(IG) of this attribute.
3. Then select the attribute which has the smallest Entropy or Largest Information gain.
4. The set S is then split by the selected attribute to produce a subset of the data.
5. The algorithm continues to recur on each subset, considering only attributes never selected before.

**Program:**

import pandas

from sklearn import tree

from sklearn.tree import DecisionTreeClassifier

import matplotlib.pyplot as plt

df = pandas.read\_csv("data.csv")

d = {'UK': 0, 'USA': 1, 'N': 2}

df['Nationality'] = df['Nationality'].map(d)

d = {'YES': 1, 'NO': 0}

df['Go'] = df['Go'].map(d)

features = ['Age', 'Experience', 'Rank', 'Nationality']

X = df[features]

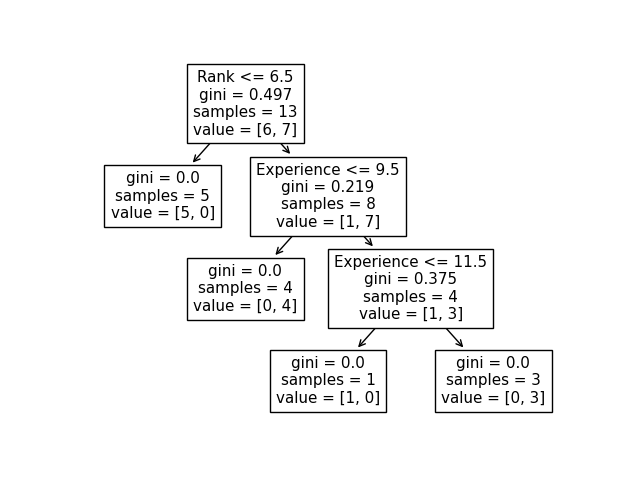
y = df['Go']

dtree = DecisionTreeClassifier()

dtree = dtree.fit(X, y)

tree.plot\_tree(dtree, feature\_names=features)

**Output:**

****

**Result**:

Hence, Decision Tree is implemented using python

**Exp No 11 Linear Regression Implementation**

**Aim:**

To implement Linear Regression using python

**Algorithm:**

1. Start

2. Read Number of Data (n)

3. For i=1 to n:

Read Xi and Yi

Next i

4. Initialize:

sumX = 0

sumX2 = 0

sumY = 0

sumXY = 0

5. Calculate Required Sum

For i=1 to n:

sumX = sumX + Xi

sumX2 = sumX2 + Xi \* Xi

sumY = sumY + Yi

sumXY = sumXY + Xi \* Yi

Next i

6. Calculate Required Constant a and b of y = a + bx:

b = (n \* sumXY - sumX \* sumY)/(n\*sumX2 - sumX \* sumX)

a = (sumY - b\*sumX)/n

7. Display value of a and b

8. Stop

**Program:**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import load\_diabetes

from sklearn.linear\_model import LinearRegression

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

from sklearn.metrics import mean\_squared\_error, r2\_score

X, Y = load\_diabetes(return\_X\_y=True)

X = X[:,8].reshape(-1,1)

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split( X, Y, test\_size = 0.3, random\_state = 10 )

lr =LinearRegression()

lr.fit( X\_train, Y\_train )

Y\_pred = lr.predict( X\_test )

print("Value of the oefficients: \n", lr.coef\_)

print(f"Mean square error: {mean\_squared\_error( Y\_test, Y\_pred)}")

print(f"Coefficient of determination: {r2\_score( Y\_test, Y\_pred )}")

plt.scatter(X\_test, Y\_test, color = "black", label = "original data")

plt.plot(X\_test, Y\_pred, color = "blue", linewidth=3, label = "regression line")

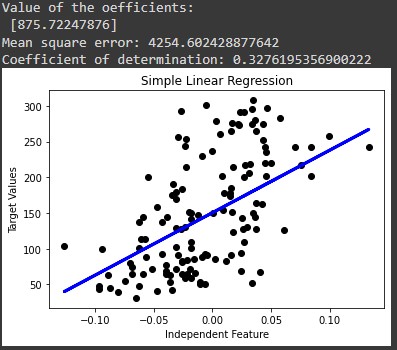
plt.xlabel("Independent Feature")

plt.ylabel("Target Values")

plt.title("Simple Linear Regression")

plt.show()

**Output:**



**Result:**

Hence, Linear Regression has been implemented using python.

**Exp No 12 K-Mean Clustering**

**Aim:**

To perform K-means clustering using python.

**Algorithm:**

1. Start

2. Get Data points D, Number of clusters k

3. Initialize k centroids randomly

4. Associate each data point in D with the nearest centroid. This will divide the data points into k clusters.

5. Recalculate the position of centroids.

6. Repeat steps 4 and 5 until there are no more changes in the membership of the data points

7. Output Data points with cluster memberships

**Program:**

import matplotlib.pyplot as plt

from sklearn.datasets import make\_blobs

# create dataset

X, y = make\_blobs(

n\_samples=150, n\_features=2,

centers=3, cluster\_std=0.5,

shuffle=True, random\_state=0

)

# plot

plt.scatter(

X[:, 0], X[:, 1],

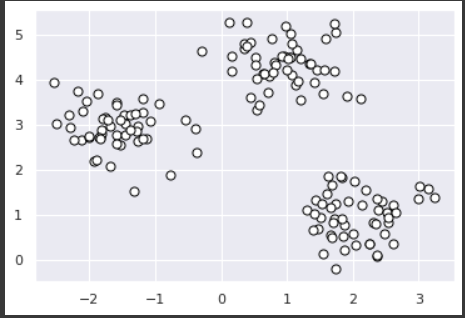
c='white', marker='o',

edgecolor='black', s=50

)

plt.show()

**Output:**

****

**Result:**

Hence, K-mean is implemented using python.