

**SRM INSTITUTE OF SCIENCE AND**

**TECHNOLOGY**

**FACULTY OF ENGINEERING & TECHNOLOGY**

(Formerly SRM University, Under section 3 of UGC Act, 1956)

S.R.M. NAGAR, KATTANKULATHUR –603 203,

KANCHEEPURAM DISTRICT

**SCHOOL OF COMPUTING**

**DEPARTMENT OF COMPUTER SCIENCE**

**18CSE305J - ARTIFICIAL INTELLIGENCE LAB MANUAL**

**Name: Anshika Maheshwari**

**Reg No: RA1911003010697**

**DEPARTMENT OF COMPUTATING TECHNOLIGIES COLLEGE OF ENGINEERING & TECHNOLOGY**

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**(Under Section 3 of UGC Act, 1956)**

**S.R.M. NAGAR, KATTANKULATHUR**

**BONAFIDE CERTIFICATE**

**Register No.: RA1911003010697**

Certified to be the bonafide record of work done by **Anshika Maheshwari** of **CSE, B.Tech.** Degree course in the Practical of **18CSC305J – Artificial Intelligence** in **SRM IST**, Kattankulathur during the academic year **2021 - 2022**.

**Staff In-Charge Head of the Department**

**Date:**

Submitted for University Examination held on \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ at **SRM IST**, Kattankulathur.

**Date:**

**Internal Examiner I Internal Examiner II**

# **COURSE CODE: 18CSC305J**

**COURSE TITLE: ARTIFICIAL INTELLIGENCE**

|  |  |  |
| --- | --- | --- |
| **Exp No.** | **Name of the Experiment** | **Page No** |
| 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 |  |

**Faculty In charge HOD**

**EXP\_1**

**Toy Problem-Bridge & Torch Problem**

Anshika Maheshwari

RA1911003010697

**Aim:**

To Solve bridge and torch problem using ai algorithm techniques

**Problem Statement**

The “Bridge and Torch” problem states that you are given an array of time a person needs to cross the bridge. Since it is time, it comprises positive integers. Along with the time we are given a bridge, which a person needs to cross. The bridge allows only two people at a time to cross. They carry a torch while crossing. And since there is a single torch. One of the people from the other side should return and take the torch back to the starting side. When two people cross the bridge, they can cross at the speed of a slower person. Find the minimum total time in which all persons can cross the bridge.

## Code

#include <bits/stdc++.h>

using namespace std;

int solveBridgeAndTorchProblem**(**int mask, bool direction, vector**<**int**>** &timeToCross, vector**<**vector**<**int**>>** &dp**)**

**{**

int n = timeToCross.size**()**;

if **(**!mask**)**

return 0;

if**(**dp**[**mask**][**direction**]**!=-1**)**

return dp**[**mask**][**direction**]**;

int transferredMask = **((**1**<<**n**)**-1**)**^mask;

int res = 0;

if **(**direction == 1**)** **{**

int minRow = INT\_MAX, person;

for **(**int i = 0; i **<** n; ++i**)** **{**

if **(**transferredMask & **(**1 **<<** i**))** **{**

if **(**minRow **>** timeToCross**[**i**])** **{**

person = i;

minRow = timeToCross**[**i**]**;

**}**

**}**

**}**

res = timeToCross**[**person**]** + solveBridgeAndTorchProblem**(**mask|**(**1 **<<** person**)**,direction^1, timeToCross, dp**)**;

**}**

else **{**

if **(**\_\_builtin\_popcount**(**mask**)** == 1**)** **{**

for **(**int i=0;i**<**n;++i**)** **{**

if **(**mask&**(**1**<<**i**))** **{**

res = timeToCross**[**i**]**;

break;

**}**

**}**

**}**

else **{**

res = INT\_MAX;

for**(**int i=0;i**<**n;++i**)** **{**

if**(**!**(**mask&**(**1**<<**i**)))**

continue;

for**(**int j=i+1;j**<**n;++j**)** **{**

if**(**mask&**(**1**<<**j**)){**

int val = max**(**timeToCross**[**i**]**, timeToCross**[**j**])**;

val += solveBridgeAndTorchProblem**(**mask^**(**1**<<**i**)**^**(**1**<<**j**)**, direction^1,timeToCross,dp**)**;

res = min**(**res, val**)**;

**}**

**}**

**}**

**}**

**}**

return dp**[**mask**][**direction**]** = res;

**}**

int main**()**

**{** int n;cin**>>**n;

vector**<**int**>** timeToCross**(**n**)**;

for**(**int i=0;i**<**n;i++**)**cin**>>**timeToCross**[**i**]**;

vector**<**vector**<**int**>>** dp**(**1**<<**20, vector**<**int**>(**2,-1**))**;

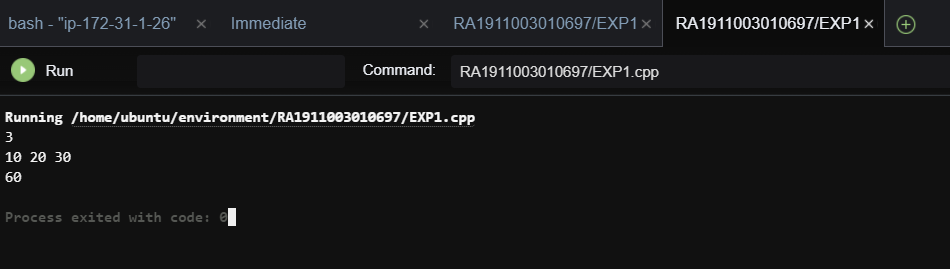
int mask = **(**1**<<**n**)**-1;

cout **<<** solveBridgeAndTorchProblem**(**mask, 0, timeToCross, dp**)**;

return 0;

**}**

Output



**Result:**

Hence the toy problem code was written and executed successfully

**EXPERIMENT-2**

**Agent Program-Travel Salesman Problem**

**Aim**

To illustrate agent program by writing code for travel salesman problem

**Problem Statement**

The challenge is to discover the shortest feasible route that visits each city precisely once and returns to the beginning point given a collection of cities and distances between each pair of cities.

Take note of the distinction between the Hamiltonian Cycle and the TSP. The Hamiltonian cycle issue is to determine whether a tour exists that visits each city precisely once. The challenge is to identify a least weight Hamiltonian Cycle. We know that Hamiltonian Tours exist (since the graph is complete), and there are many of them.

Take, For example, consider the graph shown in the figure on the right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.The problem is a famous NP-hard problem. There is no polynomial-time known solution for this problem.

**Code**

*from* sys *import* maxsize

v = 4

def travelling\_salesman\_function(*graph*, *s*):

    vertex = []

*for* i *in* range(v):

*if* i != s:

            vertex.append(i)

    min\_path = maxsize

*while* True:

        current\_cost = 0

        k = s

*for* i *in* range(len(vertex)):

            current\_cost += graph[k][vertex[i]]

            k = vertex[i]

        current\_cost += graph[k][s]

        min\_path = min(min\_path, current\_cost)

*if* not next\_perm(vertex):

*break*

*return* min\_path

def next\_perm(*l*):

    n = len(l)

    i = n-2

*while* i >= 0 and l[i] > l[i+1]:

        i -= 1

*if* i == -1:

*return* False

    j = i+1

*while* j < n and l[j] > l[i]:

        j += 1

    j -= 1

    l[i], l[j] = l[j], l[i]

    left = i+1

    right = n-1

*while* left < right:

        l[left], l[right] = l[right], l[left]

        left += 1

        right -= 1

*return* True

*#graph = [[0,10,15,20], [10,0,35,25], [15,35,0,30], [20,25,30,0]]*

list\_size = int(input("Enter the number "))

print("\n")

graph = [[int(input("Enter single number and press enter: ")) *for* \_ *in* range(list\_size)] *for* \_ *in* range(list\_size)]

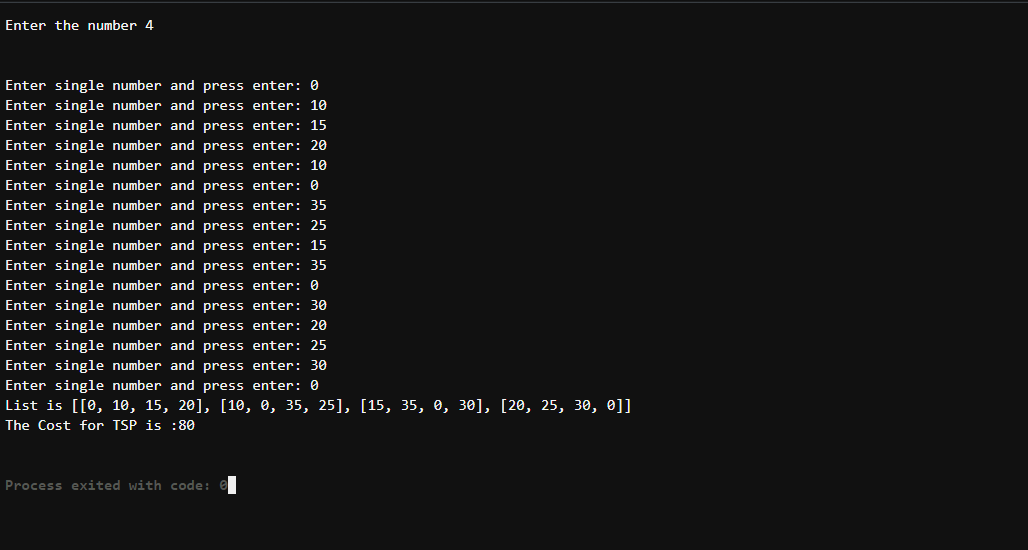
print("List is", graph)

s = 0

res = travelling\_salesman\_function(graph,s)

print("The Cost for TSP is :"+ str(res))

**Output**

****

**Result**

Hence the program for travel salesman problem was executed successfully.

**EXPERIMENT 3**

**Constraint Satisfactory problem**

**Anshika Maheshwari**

**RA1911003010697**

**Aim:**

Implementation of Constraint Satisfactory problem- Crypt Arithmetic Problem (‘SEND + MORE = MONEY’)

**Problem Statement:**

Given an expression where two words add to give a third word, assign some unique digit (0-9) to each letter where same letters cannot be assigned to different digit. The objective is to find out the digit represented by each letter that satisfies a given equation.

SEND

+ MORE

= MONEY

In this example, the solution to the puzzle is:  
O = 0, M = 1, Y = 2, E = 5, N = 6, D = 7, R = 8, and S = 9.

which gives us:

9567

+ 1085

= 10652

**Algorithm:**

1. Start.
2. Accept a expression. “SEND +MORE = MONEY”.
3. Extract the words SEND, MORE and MONEY.
4. Permute for different combination of values for S,E,N,D,M,O,R,Y.
5. Check if the sum of the left value i.e, SEND + MORE is equal to the right sum i.e, MONEY or NOT. If the sum value matches, print the mapping.
6. Continue for other permutation as well.
7. Stop.

**Program:**

*import* itertools

def get\_value(*word*, *substitution*):

    s = 0

    factor = 1

*for* letter *in* reversed(word):

        s += factor \* substitution[letter]

        factor \*= 10

*return* s

def solve(*equation*):

    left, right = equation.lower().replace(' ', '').split('=')

    left = left.split('+')

    letters = set(right)

*for* word *in* left:

*for* letter *in* word:

            letters.add(letter)

    letters = list(letters)

    digits = range(10)

*for* perm *in* itertools.permutations(digits, len(letters)):

        sol = dict(zip(letters, perm))

*if* sum(get\_value(word, sol) *for* word *in* left) == get\_value(right, sol):

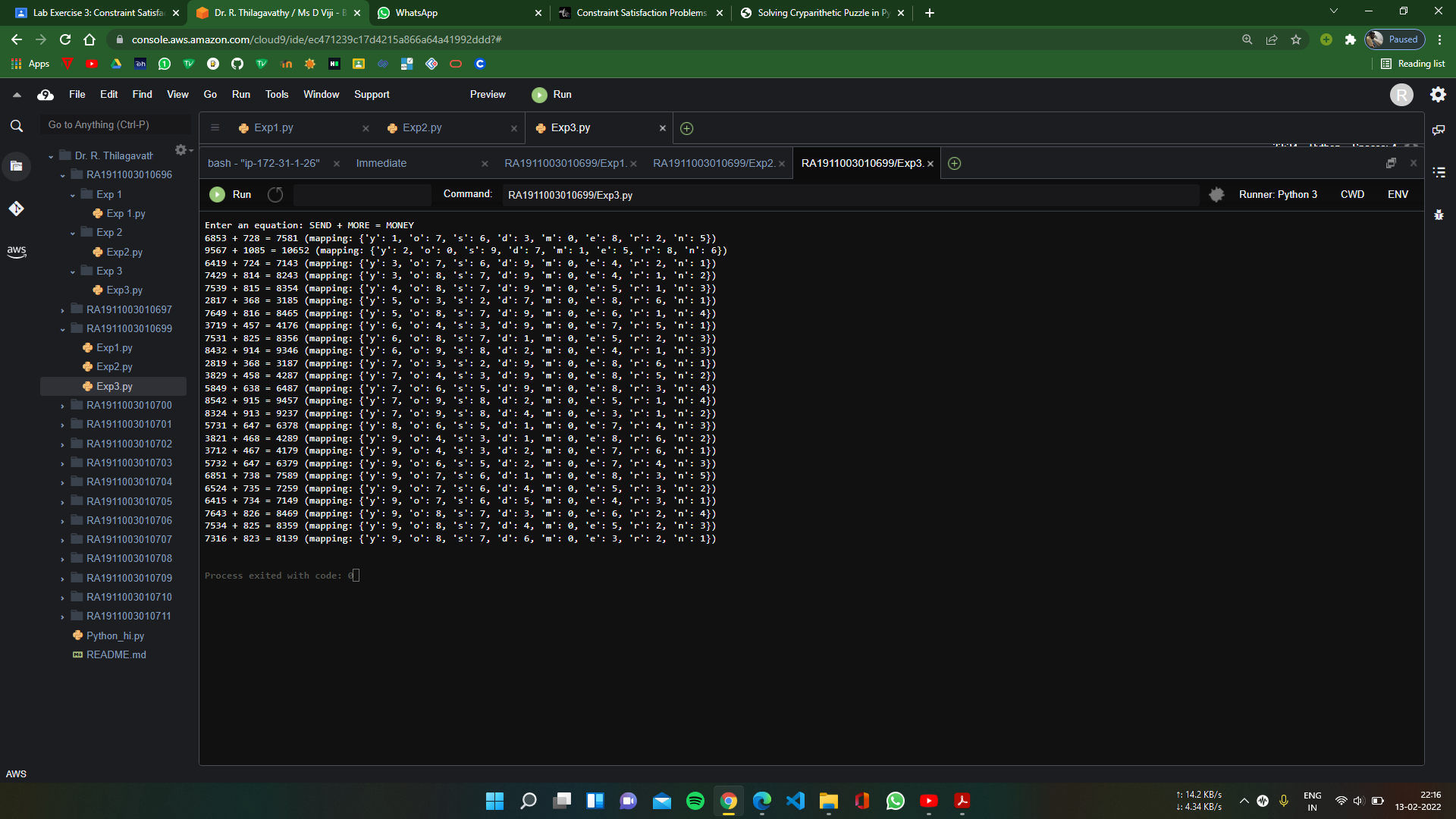
            print(' + '.join(str(get\_value(word, sol)) *for* word *in* left) + " = {} (mapping: {})".format(get\_value(right, sol), sol))

*if* \_\_name\_\_ == '\_\_main\_\_':

    eq = input("Enter an equation: ")

    solve(eq)

**Output:**

****

**RESULT:-**

Hence, the implementation of the Cryptarithmetic Problem is done successfully.

**Experiment – 4**

**Shortest Path using Breadth First Search & Depth First Search**

**Shortest Path using BFS**

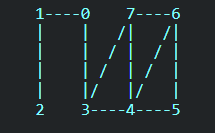
**Aim:**

Given an unweighted graph, a source, and a destination, we need to find the shortest path from source to destination in the graph in the most optimal way using BFS.

**Procedure:**

1. Initialize graph with vertices and edges.
2. Take a visited array, all initialized to false, to keep track of visited vertices.
3. Start searching from the source to find destination.
4. Take a previous array to keep track of previous vertices.
5. For a current vertex, add all its neighbors to the queue to traverse the vertices breadth wise.
6. Break the iteration when destination is found.
7. Track the shortest path from source to destination from previous array.

**Graph:**



**Program:**

import time

*class* Graph:

*def* \_\_init\_\_(*self*, *V*):

        self.V = V

        self.adj = [[] for i in range(V)]

        self.visited = [False]\*V

*def* addEdge(*self*, *u*, *v*):

        self.adj[u].append(v)

        self.adj[v].append(u)

*def* BFS(*self*, *s*, *d*):

        prev = *dict*()

        queue = [s]

        self.visited[s] = True

        while queue:

            t = queue.pop(0)

            # print(t, end = " ")

            for ver in self.adj[t]:

                if not self.visited[ver]:

                    prev[ver] = t

                    queue.append(ver)

                    self.visited[ver] = True

                    if ver == d:

                        break

        return prev

*def* shortestPath(*self*, *s*, *d*):

        path = []

        prev = self.BFS(s, d)

        at = d

        while at != s:

            path.append(at)

            at = prev[at]

        path.append(s)

        print("Shortest path: ", path[::-1])

v = *int*(input("Enter the no. of Vertices: "))

e = *int*(input("Enter the no. of Edges: "))

g = Graph(v)

for i in range(e):

    inp = input("Enter the vertices edge {}: ".format(i+1))

    edge = *list*(map(*int*, inp.split()))

    g.addEdge(edge[0], edge[1])

s = *int*(input("Enter the source: "))

d = *int*(input("Enter the destination: "))

begin = time.time()

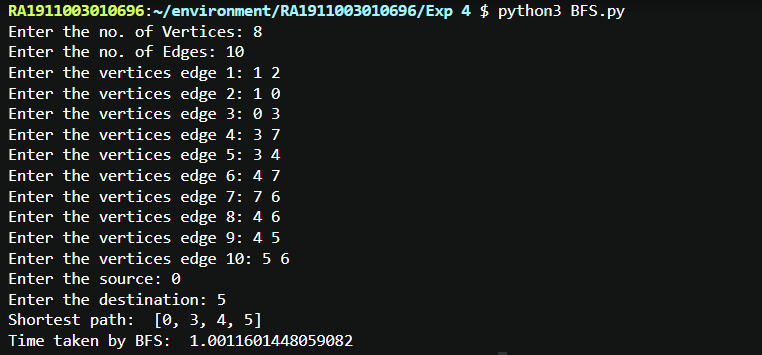
g.shortestPath(s, d)

time.sleep(1)

end = time.time()

print("Time taken by BFS: ", end - begin)

**Output:**



**Shortest Path using DFS**

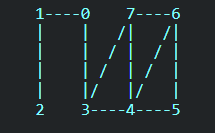
**Aim:**

Given an unweighted graph, a source, and a destination, we need to find the shortest path from source to destination in the graph in the most optimal way using BFS.

**Procedure:**

1. Initialize graph with vertices and edges.
2. Take a visited array, all initialized to false, to keep track of visited vertices.
3. Start searching from the source to find destination.
4. Keep a stack to search depth wise.
5. Pop an element from the stack, mark it visited and store all its neighbors in the stack and continue search from there along depth.
6. Store every path whenever the destination is reached.
7. Pick a path of shortest length to get the shortest path.

**Graph:**



**Program:**

import time

from collections import defaultdict

class Graph:

def \_\_init\_\_(self, V):

self.V = V

self.adj = defaultdict(list)

self.visited = [False]\*V

def addEdge(self, u, v):

self.adj[u].append(v)

self.adj[v].append(u)

def DFS(self, s, d, path, paths):

self.visited[s] = True

path.append(s)

if s == d:

paths[tuple(path)] = len(path)

for ver in self.adj[s]:

if not self.visited[ver]:

self.DFS(ver, d, path, paths)

path.pop(-1)

self.visited[s] = False

def shortestPath(self, s, d):

path = []

allPaths = dict()

self.DFS(s, d, path, allPaths)

short = float('inf')

shortest\_path = []

for p, l in allPaths.items():

if short > l:

short = l

shortest\_path = p

print("Shortest Path: ", shortest\_path)

v = int(input("Enter the no. of Vertices: "))

e = int(input("Enter the no. of Edges: "))

g = Graph(v)

for i in range(e):

inp = input("Enter the vertices of edge {}: ".format(i+1))

edge = list(map(int, inp.split()))

g.addEdge(edge[0], edge[1])

s = int(input("Enter the source: "))

d = int(input("Enter the destination: "))

begin = time.time()

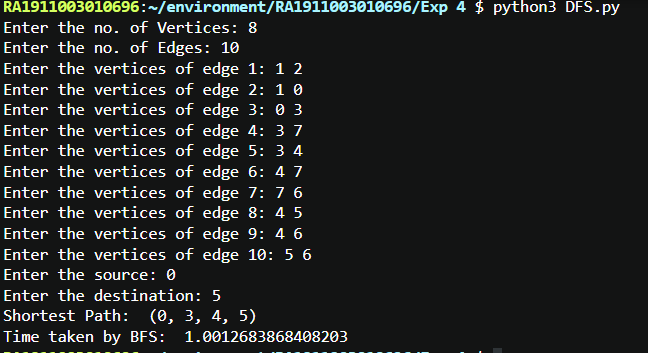
g.shortestPath(s, d)

time.sleep(1)

end = time.time()

print("Time taken by DFS: ", end - begin)

**Output:**



**Result:**

From the above outputs it was clear that DFS takes more time to find shortest path from a source to destination as the DFS has to search all the possible paths from the source to destination to find the shortest path. In case of BFS, destination is reached using the shortest path as it traverses along breadth.

**Experiment – 5**

**Finding shortest path through best first search and A\***

**Best First Search**

**Aim:** To find a path from source to destination using Best first search algorithm

**Procedure:**

1. Create 2 empty lists: OPEN and CLOSED
2. Start from the initial node (say N) and put it in the ‘ordered’ OPEN list
3. Repeat the next steps until GOAL node is reached
   1. If OPEN list is empty, then EXIT the loop returning ‘False’
   2. Select the first/top node (say N) in the OPEN list and move it to the CLOSED list. Also capture the information of the parent node
   3. If N is a GOAL node, then move the node to the Closed list and exit the loop returning ‘True’. The solution can be found by backtracking the path
   4. If N is not the GOAL node, expand node N to generate the ‘immediate’ next nodes linked to node N and add all those to the OPEN list
   5. Reorder the nodes in the OPEN list in ascending order according to an evaluation function f(n)

**Program:**

from collections import defaultdict

*class* Graph:

*def* \_\_init\_\_(*self*, *V*):

        self.V = V

        self.adj = defaultdict(*list*)

*def* addEdge(*self*, *u*, *v*, *h2*):

        self.adj[u].append((v, h2))

*def* bestFirst(*self*, *s*, *d*, *h1*):

        success = False

        open = [(s, h1)]

        closed = []

        while open and not success:

            t = open.pop(0)

            print(t[0], *end* = " ")

            if t[0] == d:

                success = True

                closed.append(t)

            else:

                closed.append(t)

                for neighbor in self.adj[t[0]]:

                    if neighbor not in open and neighbor not in closed:

                        open.append(neighbor)

                open.sort(*key* = *lambda*  *t*: t[1])

v = *int*(input("Enter the no. vertices: "))

g = Graph(v)

heuristics = *dict*()

for i in range(v):

    ver\_h = input("Enter vertex {} and its heuristic: ". format(i+1)).strip().split()

    heuristics[ver\_h[0]] = *int*(ver\_h[1])

    # print(ver\_h[0], int(ver\_h[1]))

e = *int*(input("Enter the no. edges: "))

for i in range(e):

    edge = input("Enter the vertices of edge {}: ". format(i+1)).strip().split()

    # print(heuristics[edge[0]], heuristics[edge[1]])

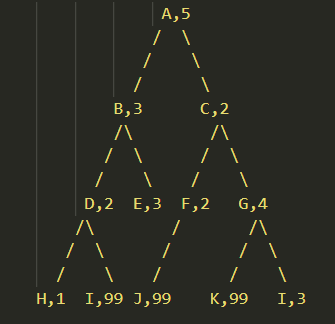
    g.addEdge(edge[0], edge[1], heuristics[edge[1]])

s = input("Enter the source: ")

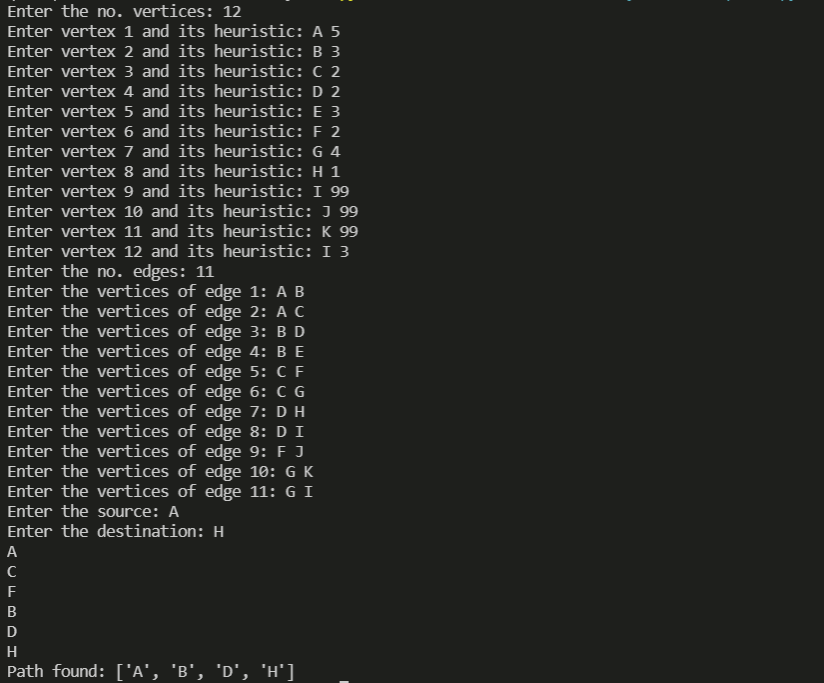
d = input("Enter the destination: ")

g.bestFirst(s, d, heuristics[s])

**Input Graph:**

****

**Output:**



**Result:**

Hence, best first search algorithm is implemented to find a path from source to destination

**A\* Search**

**Aim:** To find a path from source to destination using Best first search algorithm

**Procedure:**

1. Create 2 empty lists: OPEN and CLOSED
2. Start from the initial node (say N) and put it in the ‘ordered’ OPEN list
3. Repeat the next steps until GOAL node is reached
   1. If OPEN list is empty, then EXIT the loop returning ‘False’
   2. Select the first/top node (say N) in the OPEN list and move it to the CLOSED list. Also capture the information of the parent node
   3. If N is a GOAL node, then move the node to the Closed list and exit the loop returning ‘True’. The solution can be found by backtracking the path
   4. If N is not the GOAL node, expand node N to generate the ‘immediate’ next nodes linked to node N and add all those to the OPEN list
   5. Reorder the nodes in the OPEN list in ascending order according to an evaluation function f(n)

**Program:**

from collections import defaultdict

heuristic = dict()

Graph = defaultdict(list)

*def* aStar(*start*, *des*):

    openSet = [*start*]

    closedSet = []

    g = {}

    parent = {}

    g[*start*] = 0

    parent[*start*] = None

    while openSet:

        n = None

        for v in openSet:

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

                n = v

                print(n)

            if n == d or not Graph[n]:

                pass

            else:

                for m, w in Graph[n]:

                    if m not in openSet and m not in closedSet:

                        openSet.append(m)

                        parent[m] = n

                        g[m] = g[n] + w

                    else:

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

                            g[m] = g[n] + w

                            parent[m] = n

                            if m in closedSet:

                                closedSet.remove(m)

                                openSet.append(m)

        if n == None:

            print("Path doesn't exist!!!")

            return

        if n == *des*:

            path = []

            while parent[n] != None:

                path.append(n)

                n = parent[n]

            path.append(s)

            print("Path found: {}".format(path[::-1]))

            # print(parent)

            return

        openSet.remove(n)

        closedSet.append(n)

v = int(input("Enter the no. vertices: "))

for i in range(v):

    ver\_h = input("Enter vertex {} and its heuristic: ". format(i+1)).strip().split()

    heuristic[ver\_h[0]] = int(ver\_h[1])

e = int(input("Enter the no. edges: "))

for i in range(e):

    edge = input("Enter the vertices of edge {} along with the weight: ". format(i+1)).strip().split()

    Graph[edge[0]].append((edge[1], int(edge[2])))

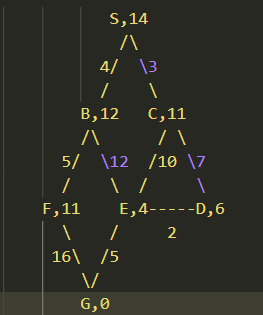
# print(Graph)

s = input("Enter the source: ")

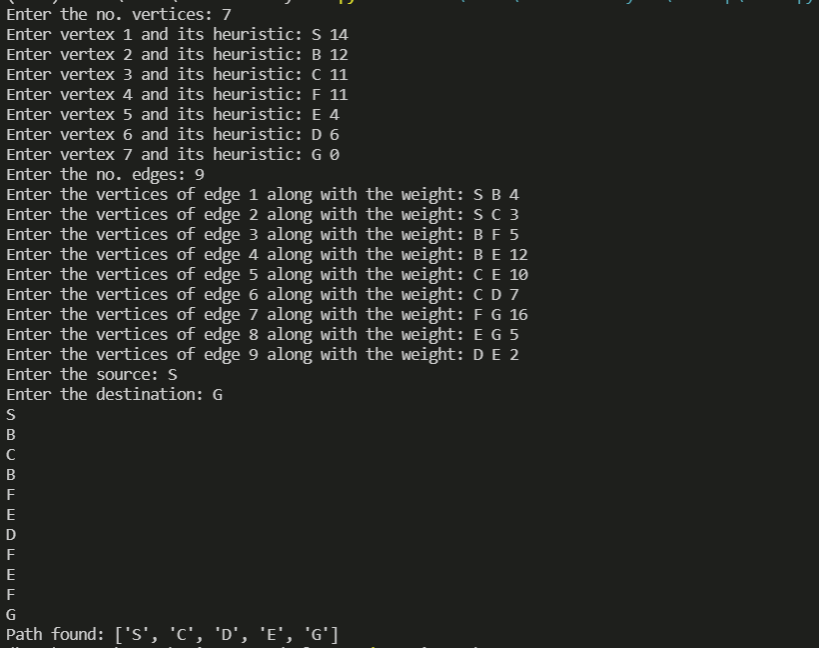
d = input("Enter the destination: ")

aStar(s, d)

**Input Graph:**

****

**Output:**

****

**Result:**

Hence, A\* search algorithm is implemented to find a path from source to destination.

**Exp No: 6**

**IMPLEMENTATION OF UNIFICATION AND RESOLUTION**

**AIM**

To write a code to illustrate 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 *for* 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) *for* 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*:

sub\_list.append(tmp)

*# Step 6*

*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:

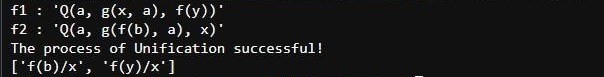
print('The process of Unification failed!')

*else*:

print('The process of Unification successful!')

print(result)

**OUTPUT UNIFICATION :**



**Result:**

Thus the code for unification and resolution was successfully executed.

**Experiment No : 7** **IMPLEMENTATION OF UNCERTAIN METHODS OF AN APPLICATION**

**Aim:**

To implement uncertain methods of an application.

**Problem Statement:**

To implement Fuzzy logic using matplotlib in python and find the graph of temperature, humidity and speed in different conditions.

**Algorithm:**

1. Locate the input, output, and state variables of the plane under consideration.
2. Split the complete universe of discourse spanned by each variable into a number of fuzzy subsets, assigning each with a linguistic label. The subsets include all the elements in the universe.
3. Obtain the membership function for each fuzzy subset.
4. Assign the fuzzy relationships between the inputs or states of fuzzy subsets on one side and the output of fuzzy subsets on the other side, thereby forming the rule base.
5. Choose appropriate scaling factors for the input and output variables for normalizing the variables between [0, 1] and [-1, I] interval.
6. Carry out the fuzzification process.
7. Identify the output contributed from each rule using fuzzy approximate reasoning.
8. Combine the fuzzy outputs obtained from each rule.
9. Finally, apply defuzzification to form a crisp output.

**Optimization Technique:**

1. Decomposing the large-scale system into a collection of various subsystems.
2. Varying the plant dynamics slowly and linearizing the nonlinear plane dynamics about a set of operating points.
3. Organizing a set of state variables, control variables, or output features for the system under consideration.
4. Designing simple P, PD, PID controllers for the subsystems. Optimal controllers can also be designed.

**Uncertainty In this problem :** Fuzzy Logic - Temperature, Humidity and Speed.

**CODE :**

*from* fuzzy\_system.fuzzy\_variable\_output *import*

FuzzyOutputVariable *from* fuzzy\_system.fuzzy\_variable\_input *import*

FuzzyInputVariable *# from fuzzy\_system.fuzzy\_variable import*

FuzzyVariable *from* fuzzy\_system.fuzzy\_system *import* FuzzySystem temp

= FuzzyInputVariable('Temperature', 10, 40, 100)

temp.add\_triangular('Cold', 10, 10, 25) temp.add\_triangular('Medium',

15, 25, 35) temp.add\_triangular('Hot', 25, 40, 40) humidity

= FuzzyInputVariable('Humidity', 20, 100, 100)

humidity.add\_triangular('Wet', 20, 20, 60) humidity.add\_trapezoidal('Normal', 30, 50, 70, 90) humidity.add\_triangular('Dry', 60, 100, 100) motor\_speed

= FuzzyOutputVariable('Speed', 0, 100, 100) motor\_speed.add\_triangular('Slow',

0, 0, 50) motor\_speed.add\_triangular('Moderate', 10, 50, 90) motor\_speed.add\_triangular('Fast', 50, 100, 100)

system = FuzzySystem() system.add\_input\_variable(temp) system.add\_input\_variable(humidity) system.add\_output\_variable(motor\_speed)

system.add\_rule(

{ 'Temperature':'Cold',

'Humidity':'Wet' }, { 'Speed':'Slow'})

system.add\_rule(

{ 'Temperature':'Cold',

‘'Humidity':'Normal' },  { 'Speed':'Slow'})

system.add\_rule(

{ 'Temperature':'Medium',

'Humidity':'Wet' },

{ 'Speed':'Slow'})

system.add\_rule(

{ 'Temperature':'Medium',

'Humidity':'Normal' },

{ 'Speed':'Moderate'})

system.add\_rule(

{ 'Temperature':'Cold',

'Humidity':'Dry' },

{ 'Speed':'Moderate'})

system.add\_rule(

{ 'Temperature':'Hot',

'Humidity':'Wet' },

{ 'Speed':'Moderate'})

system.add\_rule(

{ 'Temperature':'Hot',

'Humidity':'Normal' },

{ 'Speed':'Fast'})

system.add\_rule(

{ 'Temperature':'Hot',

'Humidity':'Dry' },

{ 'Speed':'Fast'}) system.add\_rule(

{ 'Temperature':'Medium',

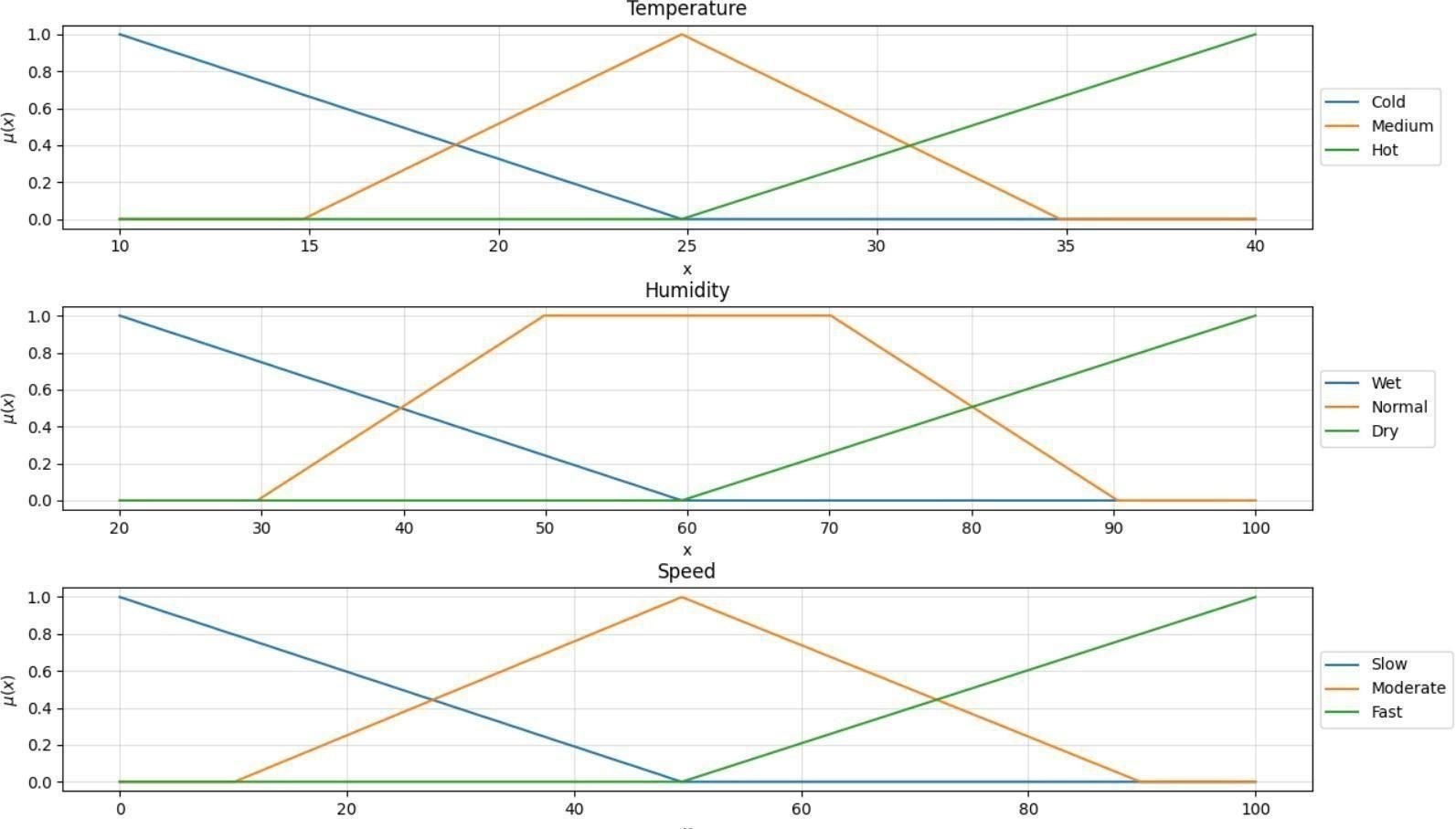
'Humidity':'Dry' }, { 'Speed':'Fast'})

output = system.evaluate\_output({

'Temperature':18,  'Humidity':60 })

print(output) system.plot\_system()

**OUTPUT :**



**Result:**

We have successfully implemented fuzzy uncertainty problem using matplotlib and output is received.

**EXPERIMENT 8: Implementation of learning algorithms for an application**

**8A: Linear regression**

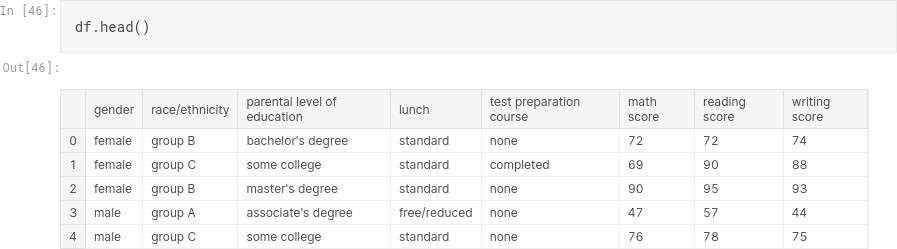
**Aim:** To write a program to implement linear regression on student score dataset

**Algorithm:**

The main function to calculate values of coefficients

1. Initialize the parameters.
2. Predict the value of a dependent variable by given an independent variable.
3. Calculate the error in prediction for all data points.
4. Calculate partial derivatives w.r.t a0 and a1.
5. Calculate the cost for each number and add them.
6. Update the values of a0 and a1.

**Dataset:**



**Code:**

import pandas as pd pd.set\_option('display.max\_columns', None) import numpy as np

df = pd.read\_csv('../input/students-performance-in-exams/StudentsPerformance.csv') df.shape df.columns df.info() df.describe() df.head() df.isnull().sum() df.rename(columns = {'race/ethnicity':'race'}, inplace = True) df.rename(columns = {'parental level of education':'parent\_education'}, inplace = True) df.rename(columns = {'test preparation course':'prep\_course'}, inplace = True) df.rename(columns = {'math score':'math\_score'}, inplace = True) df.rename(columns = {'reading score':'reading\_score'}, inplace = True) df.rename(columns = {'writing score':'writing\_score'}, inplace = True) df['total\_score'] = df['math\_score'] + df['reading\_score'] + df['writing\_score'] df.columns

import seaborn as sns import matplotlib.pyplot as plt

plt.figure(figsize = (6,8)) sns.set(style = 'darkgrid', font = 'sans-serif', font\_scale = 1.25, palette = 'Set2') ax = sns.countplot( x = 'gender', data = df, edgecolor = 'black') ax.set\_title('Distribution of Student Genders', fontsize = 15) ax.set(xlabel = 'Gender', ylabel = 'Frequency')

plt.figure(figsize = (9,6)) sns.set(style = 'darkgrid', font = 'sans-serif', font\_scale = 1.25, palette = 'deep') ax = sns.countplot( x = 'race', data = df, edgecolor = 'black') ax.set\_title('Distribution of Student Race/Ethnicity', fontsize = 15) ax.set(xlabel = 'Race/Ethnicity', ylabel = 'Frequency')

plt.figure(figsize = (13,6)) sns.set(style = 'darkgrid', font = 'sans-serif', font\_scale = 1.25, palette = 'deep') ax = sns.countplot( x = 'parent\_education', data = df, edgecolor = 'black') ax.set\_title('Distribution of Parent Education Level', fontsize = 20) ax.set(xlabel = 'Parental Education Level', ylabel = 'Frequency')

plt.figure(figsize = (6,8)) sns.set(style = 'darkgrid', font = 'sans-serif', font\_scale = 1.25, palette = 'deep') ax = sns.countplot( x = 'lunch', data = df, edgecolor = 'black') ax.set\_title('Distribution of Lunch Options', fontsize = 15) ax.set(xlabel = 'Lunch Option', ylabel = 'Frequency')

plt.figure(figsize = (6,8)) sns.set(style = 'darkgrid', font = 'sans-serif', font\_scale = 1.25, palette = 'deep') ax = sns.countplot( x = 'prep\_course', data = df, edgecolor = 'black') ax.set\_title('Distribution of Prep Course', fontsize = 15) ax.set(xlabel = 'Prep Course', ylabel = 'Frequency')

sns.set(style = 'darkgrid', font = 'sans-serif', font\_scale = 1.25) plt.figure(figsize = (10,6)) plt.hist(df['math\_score'], bins = 20, color = 'cornflowerblue') plt.xlabel('Math Score', fontsize = 13) plt.ylabel('Frequency', fontsize = 13) plt.title('Distribution of Math Scores', fontsize = 13) plt.show()

sns.set(style = 'darkgrid', font = 'sans-serif', font\_scale = 1.25) plt.figure(figsize = (10,6)) plt.hist(df['reading\_score'], bins = 20, color = 'lightcoral') plt.xlabel('Reading Score', fontsize = 13) plt.ylabel('Frequency', fontsize = 13) plt.title('Distribution of Reading Scores', fontsize = 13) plt.show()

sns.set(style = 'darkgrid', font = 'sans-serif', font\_scale = 1.25) plt.figure(figsize = (10,6)) plt.hist(df['writing\_score'], bins = 20, color = 'goldenrod') plt.xlabel('Writing Score', fontsize = 13) plt.ylabel('Frequency', fontsize = 13) plt.title('Distribution of Writing Score', fontsize = 13) plt.show()

sns.set(style = 'darkgrid', font = 'sans-serif', font\_scale = 1.25) plt.figure(figsize = (10,6)) plt.hist(df['total\_score'], bins = 20, color = 'darkorchid') plt.xlabel('Total Score', fontsize = 13) plt.ylabel('Frequency', fontsize = 13) plt.title('Distribution of Total Score', fontsize = 13) plt.show()

df1 = df[['gender', 'race', 'parent\_education', 'prep\_course', 'lunch']]

X = pd.get\_dummies(df1, columns = ['gender', 'race', 'parent\_education', 'prep\_course', 'lunch'], dtype = int) y = df['total\_score']

from sklearn.linear\_model import LinearRegression import statsmodels.api as sm

X\_constant = sm.add\_constant(X) lin\_reg = sm.OLS(y, X\_constant).fit() lin\_reg.summary()

import statsmodels.stats.api as sms sns.set\_style('darkgrid')

sns.mpl.rcParams['figure.figsize'] = (15.0, 9.0)

def linearity\_test(model, y): fitted\_vals = model.predict() resids = model.resid

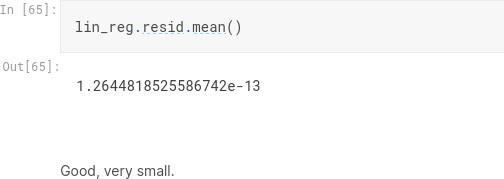
fig, ax = plt.subplots(1,2)

sns.regplot(x = fitted\_vals, y = y, lowess = True, ax = ax[0], line\_kws = {'color': 'red'}) ax[0].set\_title('Observed vs. Predicted Values', fontsize = 16) ax[0].set(xlabel = 'Predicted', ylabel = 'Observed')

sns.regplot(x = fitted\_vals, y = resids, lowess = True, ax = ax[1], line\_kws = {'color' : 'red'}) ax[1].set\_title('Residuals vs. Predicted Values', fontsize = 16) ax[1].set(xlabel = 'Predicted', ylabel = 'Residuals')

linearity\_test(lin\_reg, y) lin\_reg.resid.mean()

**Output:**



**Result:** Linear regression was trained and tested on students’ scores dataset.

**8B: Support Vector Machine (SVM)**

**Aim:** To write a program to implement support vector machine on breast cancer detection dataset

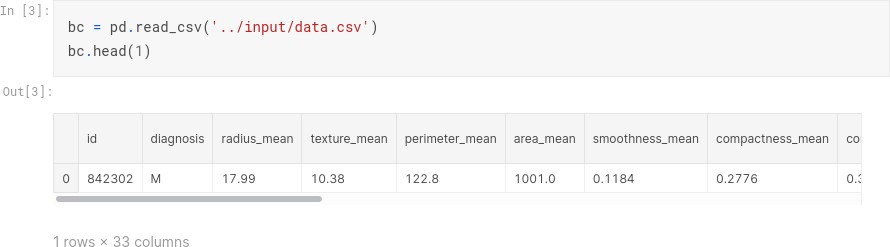
**Algorithm:**

Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.

The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane.

SVM chooses the extreme points/vectors that help in creating the hyperplane. These extreme cases are called support vectors, and hence the algorithm is termed as Support Vector Machine.

**Dataset:**



**Code:**

import numpy as np import matplotlib.pyplot as plt from sklearn.svm import SVC from sklearn.model\_selection import train\_test\_split from sklearn.metrics import confusion\_matrix from sklearn import preprocessing import pandas as pd import random import itertools import seaborn as sns sns.set(style = 'darkgrid')

def plot\_confusion\_matrix(cm, classes, normalize=False, title='Confusion matrix', cmap=plt.cm.Blues):

plt.imshow(cm, interpolation='nearest', cmap=cmap) plt.title(title) plt.colorbar() tick\_marks = np.arange(len(classes)) plt.xticks(tick\_marks, classes, rotation=45) plt.yticks(tick\_marks, classes)

if normalize:

cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis] print("Normalized confusion matrix") else: print('Confusion matrix, without normalization') print(cm)

thresh = cm.max() / 2. for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):

plt.text(j, i, cm[i, j], horizontalalignment="center",

color="white" if cm[i, j] > thresh else "black")

plt.tight\_layout() plt.ylabel('True label') plt.xlabel('Predicted label')

bc = pd.read\_csv('../input/data.csv') bc.head(1)

bcs = pd.DataFrame(preprocessing.scale(bc.iloc[:,2:32])) bcs.columns = list(bc.iloc[:,2:32].columns) bcs['diagnosis'] = bc['diagnosis']

from pandas.plotting import scatter\_matrix p = sns.PairGrid(bcs.iloc[:,20:32], hue = 'diagnosis', palette = 'Reds') p.map\_upper(plt.scatter, s = 20, edgecolor = 'w')

p.map\_diag(plt.hist)

p.map\_lower(sns.kdeplot, cmap = 'GnBu\_d')

p.add\_legend()

p.figsize = (30,30)

mbc = pd.melt(bcs, "diagnosis", var\_name="measurement") fig, ax = plt.subplots(figsize=(10,5)) p = sns.violinplot(ax = ax, x="measurement", y="value", hue="diagnosis", split = True, data=mbc, inner = 'quartile', palette =

'Set2');

p.set\_xticklabels(rotation = 90, labels = list(bcs.columns));

sns.swarmplot(x = 'diagnosis', y = 'concave points\_worst',palette = 'Set2', data = bcs);

sns.jointplot(x = bc['concave points\_worst'], y = bc['area\_mean'], stat\_func=None, color=" 4CB391", edgecolor = 'w', size = 6); X = bcs.iloc[:,0:30]

y = bcs['diagnosis'] class\_names = list(y.unique())

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.33, random\_state=42)

svc = SVC(kernel = 'linear',C=.1, gamma=10, probability = True) svc.fit(X,y) y\_pred = svc.fit(X\_train, y\_train).predict(X\_test) t = pd.DataFrame(svc.predict\_proba(X\_test))

svc.score(X\_train,y\_train), svc.score(X\_test, y\_test)

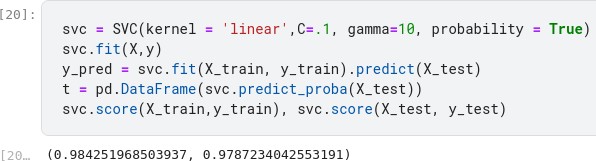
mtrx = confusion\_matrix(y\_test,y\_pred) np.set\_printoptions(precision = 2)

plt.figure()

plot\_confusion\_matrix(mtrx,classes=class\_names,title='Confusion matrix, without normalization')

plt.figure() plot\_confusion\_matrix(mtrx, classes=class\_names, normalize = True, title='Normalized confusion matrix') plt.show()

**Output:**



**Result:** Support Vector Machine was trained and tested on breast cancer detection.

**8C: K-means clustering**

**Aim:** To write a program to implement k-means clustering on customer demographic dataset

**Algorithm:**

The working of the K-Means algorithm is explained in the below steps:

Step-1: Select the number K to decide the number of clusters.

Step-2: Select random K points or centroids. (It can be other from the input dataset).

Step-3: Assign each data point to their closest centroid, which will form the predefined K clusters.

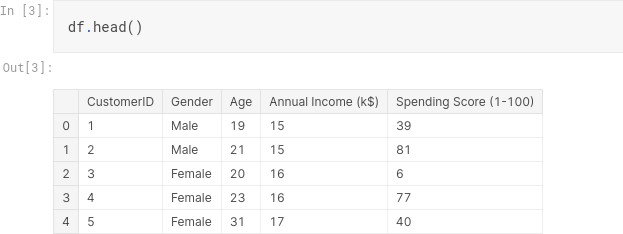
Step-4: Calculate the variance and place a new centroid of each cluster.

Step-5: Repeat the third steps, which means reassign each datapoint to the new closest centroid of each cluster.

Step-6: If any reassignment occurs, then go to step-4 else go to FINISH.

Step-7: The model is ready.

**Dataset:**



**Code:**

df.head()

df.drop('CustomerID', axis=1, inplace = True) df.head()

df.shape df.info() df.isnull().sum()

df.describe() cor = df.corr() sns.set(font\_scale=1.4) plt.figure(figsize=(9,8)) sns.heatmap(cor, annot=True, cmap='plasma') plt.tight\_layout() plt.show() plt.figure(figsize=(16,12),facecolor='9DF08E')



Spending Score

plt.subplot(3,3,1) plt.title('Spending Score\n', color=' FF000B') sns.distplot(df['Spending Score (1-100)'], color='orange')

 Age plt.subplot(3,3,2) plt.title('Age\n', color=' FF000B') sns.distplot(df['Age'], color=' 577AFF')

 Annual Income plt.subplot(3,3,3) plt.title('Annual Income\n', color=' FF000B') sns.distplot(df['Annual Income (k$)'], color='black')

plt.suptitle(' Distribution Plots\n', color=' 0000C1', size = 30) plt.tight\_layout()

 Before-After Label Encoder

from sklearn.preprocessing import LabelEncoder print('\033[0;32m' + 'Before Label Encoder\n' + '\033[0m' + '\033[0;32m', df['Gender'])

le = LabelEncoder()

df['Gender'] = le.fit\_transform(df.iloc[:,0])

print('\033[0;31m' + '\n\nAfter Label Encoder\n' + '\033[0m' + '\033[0;31m', df['Gender']) spending\_score\_male = 0 spending\_score\_female = 0

for i in range(len(df)): if df['Gender'][i] == 1:

spending\_score\_male = spending\_score\_male + df['Spending Score (1-100)'][i] if df['Gender'][i] == 0:

spending\_score\_female = spending\_score\_female + df['Spending Score (1-100)'][i]

print('\033[1m' + '\033[93m' + f'Males Spending Score : {spending\_score\_male}') print('\033[1m' + '\033[93m' + f'Females Spending Score: {spending\_score\_female}')

plt.figure(figsize=(16,16),facecolor=' 54C6C0') plt.subplot(3,3,1)

plots = sns.barplot(x=['Female','Male'], y=df['Gender'].value\_counts(), data=df)

for bar in plots.patches: plots.annotate(format(bar.get\_height(), '.0f'), (bar.get\_x() + bar.get\_width() / 2, bar.get\_height()), ha='center', va='center', size=13, xytext=(0, 8),

textcoords='offset points',color='red')

plt.xlabel("Gender", size=14) plt.ylabel("Number", size=14) plt.yticks(np.arange(0,116,10),size='14') plt.grid(False)

plt.title("Number of Genders\n", color="red", size='22')

 Gender & Total Spending Score

list\_genders\_spending\_score = [int(spending\_score\_female),int(spending\_score\_male)] series\_genders\_spending\_score = pd.Series(data = list\_genders\_spending\_score)

plt.subplot(3,3,2)

plots = sns.barplot(x=['Female','Male'], y=series\_genders\_spending\_score, palette=['yellow','purple'])

for bar in plots.patches: plots.annotate(format(bar.get\_height(), '.0f'), (bar.get\_x() + bar.get\_width() / 2, bar.get\_height()), ha='center', va='center', size=13, xytext=(0, 8),

textcoords='offset points',color='red')

plt.xlabel("Gender", size=14) plt.ylabel("Total Spending Score", size=14) plt.yticks(np.arange(0,6001,1000),size='14') plt.grid(False)

plt.title("Gender & Total Spending Score\n", color="red", size='22')

 Gender & Mean Spending Score

list\_genders\_spending\_score\_mean =

[int(spending\_score\_female/df['Gender'].value\_counts()[0]),int(spending\_score\_male/df['Gender'].value\_counts()[1])] series\_genders\_spending\_score\_mean = pd.Series(data = list\_genders\_spending\_score\_mean)

plt.subplot(3,3,3)

plots = sns.barplot(x=['Female','Male'], y=series\_genders\_spending\_score\_mean, palette='hsv')

for bar in plots.patches: plots.annotate(format(bar.get\_height(), '.0f'), (bar.get\_x() + bar.get\_width() / 2, bar.get\_height()), ha='center', va='center', size=13, xytext=(0, 8),

textcoords='offset points',color='red')

plt.xlabel("Gender", size=14) plt.ylabel("Mean Spending Score", size=14) plt.yticks(np.arange(0,71,10),size='14') plt.grid(False) plt.title("Gender & Mean Spending Score\n", color="red", size='22') plt.tight\_layout() plt.show() plt.figure(figsize=(12,8)) sns.scatterplot(x = df['Age'], y = df['Spending Score (1-100)']) plt.title('Age - Spending Score', size = 23, color='red') plt.figure(figsize=(12,8)) sns.scatterplot(x = df['Annual Income (k$)'], y = df['Spending Score (1-100)'], palette = "red") plt.title('Annual Income - Spending Score', size = 23, color='red') x = df.iloc[:,0:].values print("\033[1;31m" + f'X data before PCA:\n {x[0:5]}')

 standardization before PCA

from sklearn.preprocessing import StandardScaler sc = StandardScaler()

X = sc.fit\_transform(x)

 PCA from sklearn.decomposition import PCA pca = PCA(n\_components = 2) X\_2D = pca.fit\_transform(X)

print("\033[0;32m" + f'\nX data after PCA:\n {X\_2D[0:5,:]}')

 finding optimum number of clusters

from sklearn.cluster import KMeans wcss\_list = []

for i in range(1,11): kmeans\_test = KMeans(n\_clusters = i, init ='k-means++', random\_state=88) kmeans\_test.fit(X\_2D)

wcss\_list.append(kmeans\_test.inertia\_)

plt.figure(figsize=(9,6)) plt.plot(range(1, 11), wcss\_list) plt.title('The Elbow Method', color='red',fontsize='23') plt.xlabel('Number of clusters') plt.xticks(np.arange(1,11)) plt.ylabel('WCSS') plt.show() kmeans = KMeans(n\_clusters = 4, init ='k-means++', random\_state=88) y\_kmeans = kmeans.fit\_predict(X\_2D)

plt.figure(1 , figsize = (16 ,9)) plt.scatter(X\_2D[y\_kmeans == 0, 0], X\_2D[y\_kmeans == 0, 1], s = 80, c = 'orange', label = 'Cluster-1') plt.scatter(X\_2D[y\_kmeans == 1, 0], X\_2D[y\_kmeans == 1, 1], s = 80, c = 'red', label = 'Cluster-2') plt.scatter(X\_2D[y\_kmeans == 2, 0], X\_2D[y\_kmeans == 2, 1], s = 80, c = 'green', label = 'Cluster-3') plt.scatter(X\_2D[y\_kmeans == 3, 0], X\_2D[y\_kmeans == 3, 1], s = 80, c = 'purple', label = 'Cluster-4') plt.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], s = 375, c = 'brown', label = 'Centroids') plt.title("Customers' Clusters") plt.xlabel('PCA Variable-1', color='red') plt.ylabel('PCA Variable-2', color='red') plt.legend() plt.show() x = df[['Age','Annual Income (k$)','Spending Score (1-100)']].values x\_df = df[['Age','Annual Income (k$)','Spending Score (1-100)']]  this line for 3d scatter plot wcss\_list = []

for i in range(1,11): kmeans\_test = KMeans(n\_clusters = i, init ='k-means++', random\_state=88) kmeans\_test.fit(x)

wcss\_list.append(kmeans\_test.inertia\_)

plt.figure(figsize=(9,6)) plt.plot(range(1, 11), wcss\_list) plt.title('The Elbow Method', color='red',fontsize='23') plt.xlabel('Number of clusters') plt.xticks(np.arange(1,11)) plt.ylabel('WCSS') plt.show()

 KMeans kmeans = KMeans(n\_clusters = 2, init ='k-means++', random\_state=88) y\_kmeans = kmeans.fit\_predict(x)  clusters visualization

plt.figure(1 , figsize = (16 ,9)) plt.scatter(x[y\_kmeans == 0, 0], x[y\_kmeans == 0, 1], s = 80, c = ' 13DB8C', label = 'Cluster-1') plt.scatter(x[y\_kmeans == 1, 0], x[y\_kmeans == 1, 1], s = 80, c = ' 72BAFF', label = 'Cluster-2') plt.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], s = 350, c = 'brown', label = 'Centroids') plt.title("Customers' Clusters") plt.xlabel('Age', color='red') plt.ylabel('Annual Income (k$)', color='red') plt.legend() plt.show()

x = df[['Annual Income (k$)','Spending Score (1-100)']].values

 finding optimum number of clusters

wcss\_list = [] for i in range(1,11): kmeans\_test = KMeans(n\_clusters = i, init ='k-means++', random\_state=88) kmeans\_test.fit(x)

wcss\_list.append(kmeans\_test.inertia\_)

plt.figure(figsize=(9,6)) plt.plot(range(1, 11), wcss\_list) plt.title('The Elbow Method', color='red',fontsize='23') plt.xlabel('Number of clusters') plt.xticks(np.arange(1,11)) plt.ylabel('WCSS') plt.show() kmeans = KMeans(n\_clusters = 5, init ='k-means++', random\_state=88) y\_kmeans = kmeans.fit\_predict(x)

 clusters visualization

plt.figure(1 , figsize = (16 ,9)) plt.scatter(x[y\_kmeans == 0, 0], x[y\_kmeans == 0, 1], s = 80, c = 'orange', label = 'Cluster-1') plt.scatter(x[y\_kmeans == 1, 0], x[y\_kmeans == 1, 1], s = 80, c = 'red', label = 'Cluster-2') plt.scatter(x[y\_kmeans == 2, 0], x[y\_kmeans == 2, 1], s = 80, c = 'purple', label = 'Cluster-3') plt.scatter(x[y\_kmeans == 3, 0], x[y\_kmeans == 3, 1], s = 80, c = 'lime', label = 'Cluster-4') plt.scatter(x[y\_kmeans == 4, 0], x[y\_kmeans == 4, 1], s = 80, c = 'blue', label = 'Cluster-5') plt.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], s = 375, c = 'brown', label = 'Centroids') plt.title("Customers' Clusters") plt.xlabel('Annual Income (k$)', color='red') plt.ylabel('Spending Score', color='red') plt.legend() plt.show()

**Output:**



**Result:** K-means clustering was trained and tested on customer demographic data

**8D: Apriori**

**Aim:** To write a program to implement apriori on sales dataset

**Algorithm:**

Step 1. Computing the support for each individual item

Step 2. Deciding on the support threshold

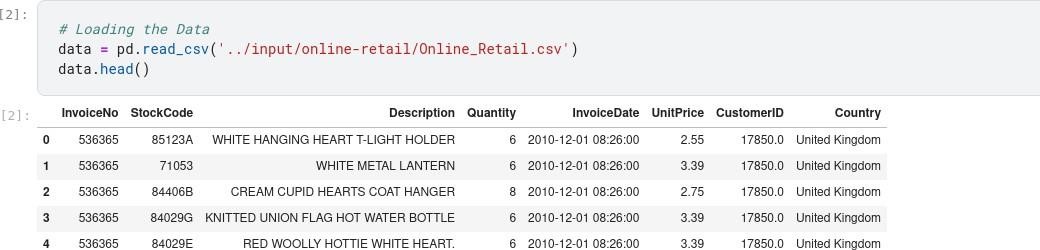
Step 3. Selecting the frequent items

Step 4. Finding the support of the frequent itemsets

Step 5. Repeat for larger sets

Step 6. Generate Association Rules and compute confidence Step 7. Compute lift

**Dataset:**



**Code:**

import numpy as np

import pandas as pd

from mlxtend.frequent\_patterns import apriori, association\_rules

 Loading the Data data = pd.read\_csv('../input/online-retail/Online\_Retail.csv') data.head()

 Exploring the columns of the data data.columns

 Exploring the different regions of transactions data.Country.unique()

 Stripping extra spaces in the description data['Description'] = data['Description'].str.strip()

 Dropping the rows without any invoice number data.dropna(axis = 0, subset =['InvoiceNo'], inplace = True) data['InvoiceNo'] = data['InvoiceNo'].astype('str')

 Dropping all transactions which were done on credit data = data[~data['InvoiceNo'].str.contains('C')]

 Transactions done in France basket\_fra = (data[data['Country'] =="France"]

.groupby(['InvoiceNo', 'Description'])['Quantity']

.sum().unstack().reset\_index().fillna(0) .set\_index('InvoiceNo'))

 Transactions done in the Brazil basket\_bra = (data[data['Country'] =="Brazil"]

.groupby(['InvoiceNo', 'Description'])['Quantity']

.sum().unstack().reset\_index().fillna(0) .set\_index('InvoiceNo'))

 Transactions done in Portugal basket\_por = (data[data['Country'] =="Portugal"]

.groupby(['InvoiceNo', 'Description'])['Quantity']

.sum().unstack().reset\_index().fillna(0) .set\_index('InvoiceNo'))

 Transactions done in Portugal basket\_swe = (data[data['Country'] =="Sweden"]

.groupby(['InvoiceNo', 'Description'])['Quantity']

.sum().unstack().reset\_index().fillna(0) .set\_index('InvoiceNo'))

 Defining the hot encoding function to make the data suitable  for the concerned libraries

def hot\_encode(x):

if(x<= 0): return 0

if(x>= 1):

return 1

 Encoding the datasets basket\_encoded = basket\_fra.applymap(hot\_encode) basket\_fra = basket\_encoded

basket\_encoded = basket\_bra.applymap(hot\_encode) basket\_bra = basket\_encoded

basket\_encoded = basket\_por.applymap(hot\_encode) basket\_por = basket\_encoded

basket\_encoded = basket\_swe.applymap(hot\_encode) basket\_swe = basket\_encoded

France

 Building the model frq\_items = apriori(basket\_fra, min\_support = 0.05, use\_colnames = True)

 Collecting the inferred rules in a dataframe rules = association\_rules(frq\_items, metric ="lift", min\_threshold = 1) rules = rules.sort\_values(['confidence', 'lift'], ascending =[False, False]) print(rules.head())

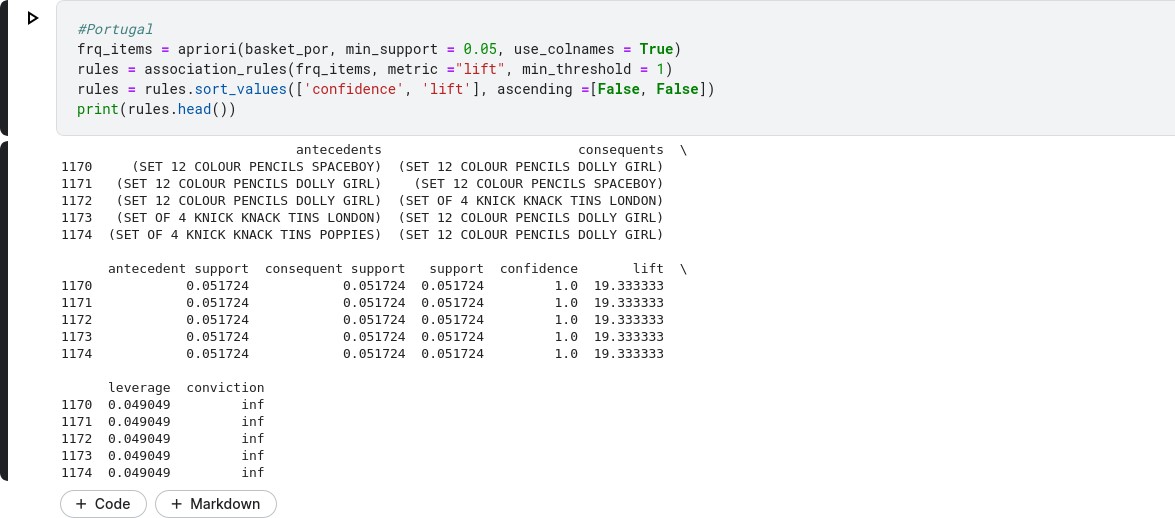
Brazil frq\_items = apriori(basket\_bra, min\_support = 0.05, use\_colnames = True) rules = association\_rules(frq\_items, metric ="lift", min\_threshold = 1) rules = rules.sort\_values(['confidence', 'lift'], ascending =[False, False]) print(rules.head())

Portugal frq\_items = apriori(basket\_por, min\_support = 0.05, use\_colnames = True) rules = association\_rules(frq\_items, metric ="lift", min\_threshold = 1) rules = rules.sort\_values(['confidence', 'lift'], ascending =[False, False]) print(rules.head())

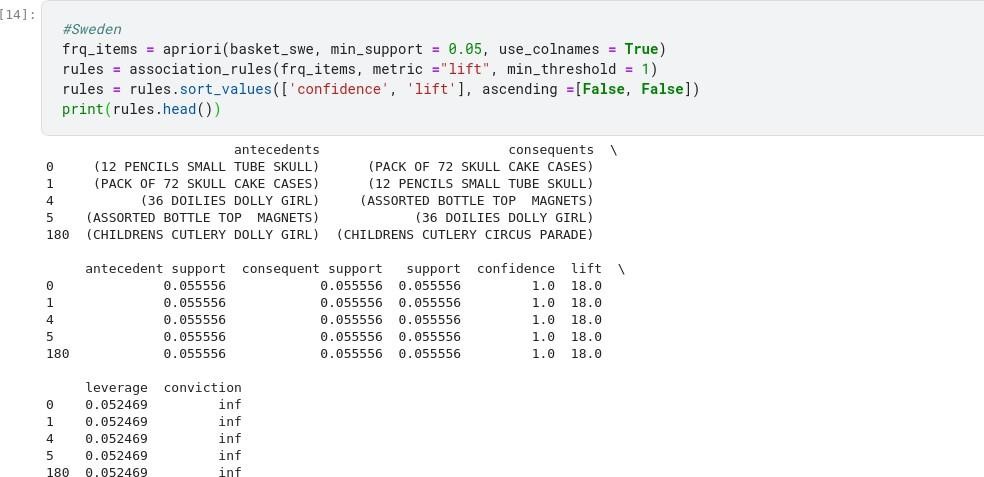
Sweden frq\_items = apriori(basket\_swe, min\_support = 0.05, use\_colnames = True) rules = association\_rules(frq\_items, metric ="lift", min\_threshold = 1) rules = rules.sort\_values(['confidence', 'lift'], ascending =[False, False]) print(rules.head())

**Output:**

Rules for items of Portugal



Rules for items of Sweden.



**Result:** Apriori algorithm was successfully trained and tested on a sales dataset.

# **EXPERIMENT 9: Implementation of NLP programs**

**AIM:**

To write a program to perform sentiment analysis on given statements.cd

**PROCEDURE:**

## 1. Load the CSV dataset into Pandas DataFrame 2. Remove the unnecessary columns from the DataFrame 3. Splitting the dataset into train and test set 4. Removing neutral sentiments 5. Filter the *stopwords* from the text entries 6. Define *word features* and their extraction from text 7. Train Naive Bayes classifier for classification of text into positive or negative 8. Test the trained classifier on our test-set

**DATASET:**



**CODE:**

import numpy as np  linear algebra import pandas as pd  data processing, CSV file I/O

(e.g. pd.read\_csv) from sklearn.model\_selection import train\_test\_split function for splitting data to train and test sets

import nltk from nltk.corpus import stopwords

from nltk.classify import SklearnClassifier

from wordcloud import WordCloud,STOPWORDS import matplotlib.pyplot as plt get\_ipython().run\_line\_magic('matplotlib', 'inline')

from subprocess import check\_output

data = pd.read\_csv('../input/Sentiment.csv')

 Keeping only the necessary columns data = data[['text','sentiment']]

 First of all, splitting the dataset into a training and a testing set. The test set is the 10% of the original dataset. For this particular analysis I dropped the neutral tweets, as my goal was to only differentiate positive and negative tweets.

 Splitting the dataset into train and test set train, test = train\_test\_split(data,test\_size = 0.1)  Removing neutral sentiments train = train[train.sentiment != "Neutral"]

 As a next step I separated the Positive and Negative tweets of the training set in order to easily visualize their contained words. After that I cleaned the text from hashtags, mentions and links. Now they were ready for a WordCloud visualization which shows only the most emphatic words of the Positive and Negative tweets.

train\_pos = train[ train['sentiment'] == 'Positive'] train\_pos = train\_pos['text'] train\_neg = train[ train['sentiment'] == 'Negative'] train\_neg = train\_neg['text']

def wordcloud\_draw(data, color = 'black'): words = ' '.join(data) cleaned\_word = " ".join([word for word in words.split()

|  |  |
| --- | --- |
| word.startswith('@')  word.startswith(' ') | if 'http' not in word and not    and not    and word != 'RT' |

])

wordcloud = WordCloud(stopwords=STOPWORDS,

background\_color=color, width=2500, height=2000

).generate(cleaned\_word)

plt.figure(1,figsize=(13, 13)) plt.imshow(wordcloud) plt.axis('off')

plt.show()

print("Positive words") wordcloud\_draw(train\_pos,'white') print("Negative words") wordcloud\_draw(train\_neg)

 Interesting to notice the following words and expressions in the positive word set:

 \*\*truth\*\*, \*\*strong\*\*, \*\*legitimate\*\*,

\*\*together\*\*, \*\*love\*\*, \*\*job\*\*



 In my interpretation, people tend to believe that their ideal candidate is truthful, legitimate, above good and bad.

 At the same time, negative tweets contains words like:

 \*\*influence\*\*, \*\*news\*\*, \*\*elevator music\*\*, \*\*disappointing\*\*, \*\*softball\*\*, \*\*makeup\*\*, \*\*cherry picking\*\*, \*\*trying\*\*



 In my understanding people missed the decisively action and considered the scolded candidates too soft and cherry picking.

 After the visualization, I removed the hashtags, mentions, links and stopwords from the training set.



 \*\*Stop Word:\*\* Stop Words are words which do not contain important significance to be used in Search Queries. Usually these words are filtered out from search queries because they return vast amounts of unnecessary information. ( the, for, this etc. )

tweets = []

stopwords\_set = set(stopwords.words("english"))

for index, row in train.iterrows():

words\_filtered = [e.lower() for e in

row.text.split() if len(e) >= 3] words\_cleaned = [word for word in words\_filtered if 'http' not in word and not word.startswith('@') and not word.startswith(' ') and word != 'RT'] words\_without\_stopwords = [word for word in

words\_cleaned if not word in stopwords\_set] tweets.append((words\_without\_stopwords,

row.sentiment))

test\_pos = test[ test['sentiment'] == 'Positive'] test\_pos = test\_pos['text'] test\_neg = test[ test['sentiment'] == 'Negative'] test\_neg = test\_neg['text']

 As a next step I extracted the so called features with nltk lib, first by measuring a frequent distribution and by selecting the resulting keys.

 Extracting word features def get\_words\_in\_tweets(tweets): all = [] for (words, sentiment) in tweets:

all.extend(words)

return all

def get\_word\_features(wordlist): wordlist = nltk.FreqDist(wordlist) features = wordlist.keys() return features

w\_features =

get\_word\_features(get\_words\_in\_tweets(tweets))

def extract\_features(document):

document\_words = set(document) features = {} for word in w\_features:

features['contains(%s)' % word] = (word in

document\_words) return features

 Hereby I plotted the most frequently distributed words. The most words are centered around debate nights.

wordcloud\_draw(w\_features)

 Using the nltk NaiveBayes Classifier I classified the extracted tweet word features.

 Training the Naive Bayes classifier training\_set = nltk.classify.apply\_features(extract\_features,tweets) classifier =

nltk.NaiveBayesClassifier.train(training\_set)

 Finally, with my selected metrics, I tried to measure how the classifier algorithm scored.

neg\_cnt = 0 pos\_cnt = 0 for obj in test\_neg:

res =

classifier.classify(extract\_features(obj.split())) if(res == 'Negative'):

neg\_cnt = neg\_cnt + 1

for obj in test\_pos:

res =

classifier.classify(extract\_features(obj.split())) if(res == 'Positive'): pos\_cnt = pos\_cnt + 1

print('[Negative]: %s/%s ' % (len(test\_neg),neg\_cnt))

print('[Positive]: %s/%s ' % (len(test\_pos),pos\_cnt))

**OUTPUT:**





**RESULTS:**

## Sentiment analysis was performed successfully on the given statements

**Experiment-10**

**Implementation of Natural Language Problem – Text to Speech**

**Aim:**

To implement natural language problem programs- Text to Speech.

**Algorithm:**

1. Import, install, download all the required modules / packages/ libraries required to perform natural language processing activities.
2. Perform tokenization and display the output.
3. Convert the given input into bi-grams, tri-grams and n- grams as required.
4. Perform stemming for the given input.
5. Perform part-of-speech tagging and display the output for the given input.
6. Implement Named entity recognition on the given input.
7. Perform text-to-speech with the help of gTTS module.

**Code:**

!pip install gTTS import nltk import nltk.corpus

#Tokenization

from nltk.tokenize import word\_tokenize

chess = "Samay Raina is the best chess streamer in the world" nltk.download('punkt') word\_tokenize(chess)

#sentence tokenizer

from nltk.tokenize import sent\_tokenize

chess2 = "Samay Raina is the best chess streamer in the world. Sagar Sh ah is the best chess coach in the world"

sent\_tokenize(chess2)

#Checking the number of tokens len(word\_tokenize(chess))

#bigrams and n-grams

astronaut = "Can anybody hear me or am I talking to myself? My mind is running empty in the search for someone else"

astronaut\_token=(word\_tokenize(astronaut)) list(nltk.bigrams(astronaut\_token)) list(nltk.trigrams(astronaut\_token)) list(nltk.ngrams(astronaut\_token,5))

#Stemming

from nltk.stem import PorterStemmer my\_stem = PorterStemmer() my\_stem.stem("eating") my\_stem.stem("going") my\_stem.stem("shopping")

#pos-tagging

tom ="Tom Hanks is the best actor in the world" tom\_token = word\_tokenize(tom) nltk.download('averaged\_perceptron\_tagger') nltk.pos\_tag(tom\_token)

#Named entity recognition from nltk import ne\_chunk

president = "Barack Obama was the 44th President of America" president\_token = word\_tokenize(president)

president\_pos = nltk.pos\_tag(president\_token)

nltk.download('maxent\_ne\_chunker') nltk.download('words') print(ne\_chunk(president\_pos))

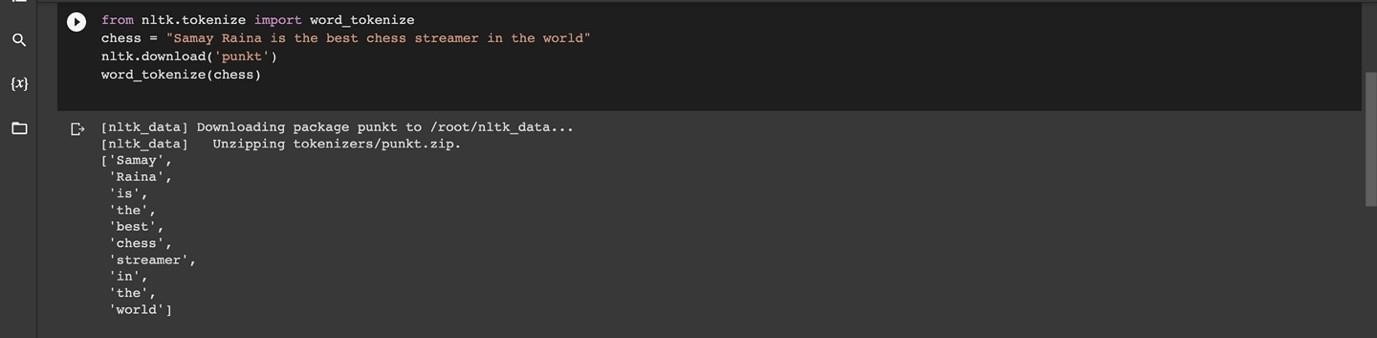
from gtts import gTTS from IPython.display import Audio

tts = gTTS('Hello Atul, How are you') tts.save('1.wav')

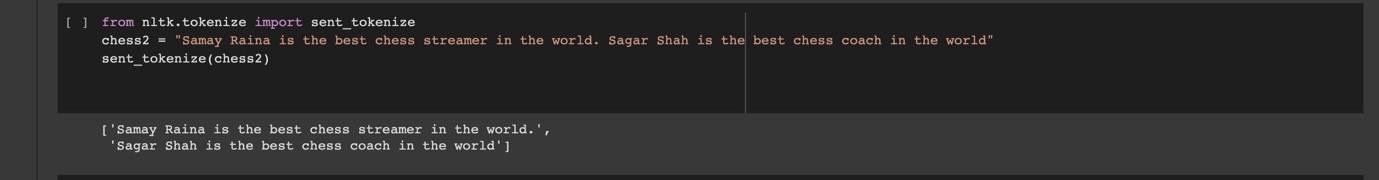
sound\_file = '1.wav' Audio(sound\_file, autoplay=True)

**Output:**

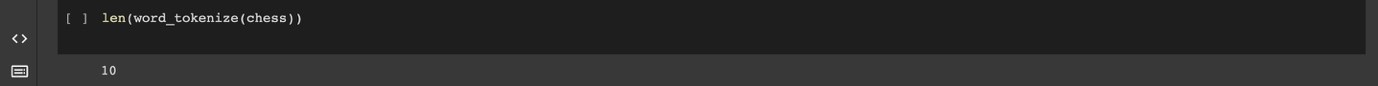
1. **Tokenization:**



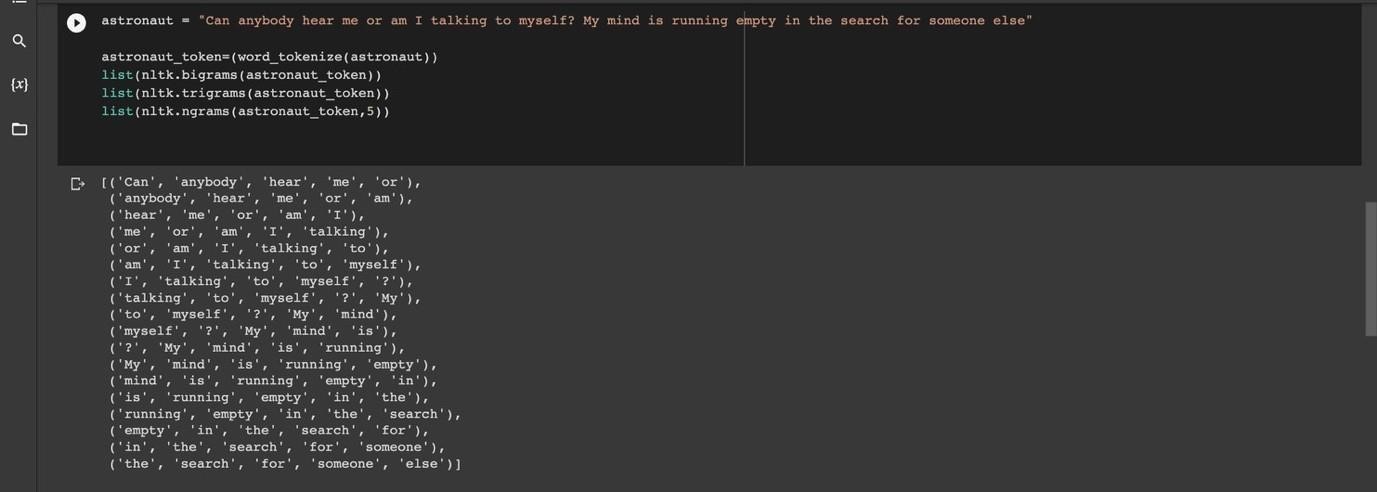
1. **Sentence tokenizer:**



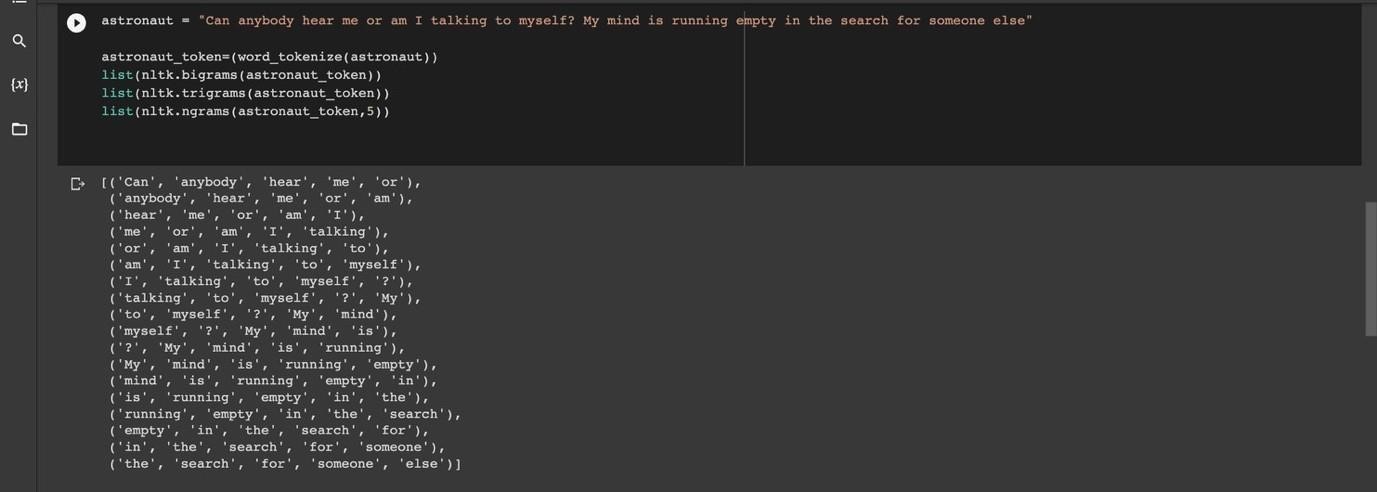
1. **Number of tokens** (for chess = "Samay Raina is the best chess streamed in the world”)



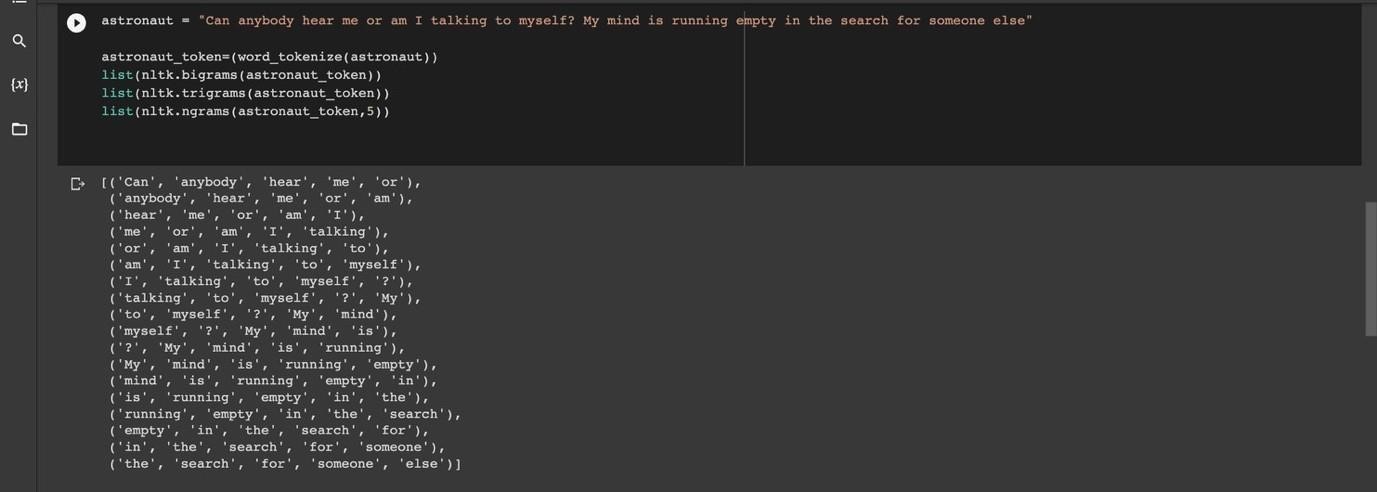
1. **Bigrams:**



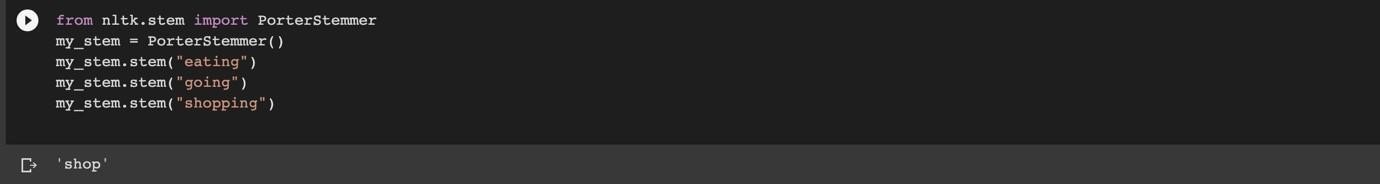
1. **Trigrams:**



1. **N-grams:**



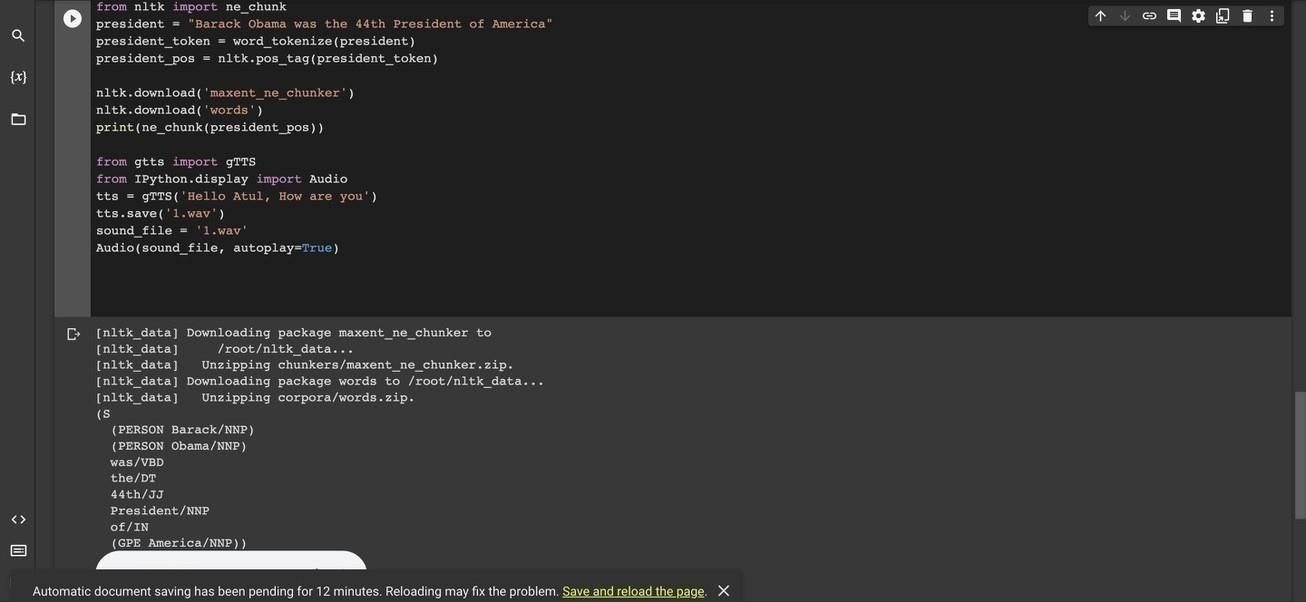
1. **Stemming:**



1. **Pos-tagging:**



1. **Named entity recognition:**



**Result:**

Thus, google text to speech has been performed along with other language processing successfully.