

# **18CSC305J – ARTIFICIAL INTELLIGENCE**

**SEMESTER – VI**

**2021 – 2022 (EVEN)**

**Name : Naga Kasi Sai Ram**  
**Register No : RA1911003010696**  
**Branch : CSE**  
**Section : C2**



**SRM**  
INSTITUTE OF SCIENCE & TECHNOLOGY  
*Deemed to be University u/s 3 of UGC Act, 1956*

**DEPARTMENT OF COMPUTING TECHNOLOGIES**

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

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

**S.R.M. NAGAR, KATTANKULATHUR – 603 203**  
**CHENGALPATTU DISTRICT**

**DEPARTMENT OF COMPUTING TECHNOLOGIES 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.: RA1911003010696**

Certified to be the bonafide record of work done by **Naga Kasi Sai Ram** 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**

# INDEX

Ex. No.	Title	Marks	Signature
1	Implementation of Toy Problems		
2	Developing agent programs for real world problems		
3	Implementation of constraint satisfaction problems		
4	Uninformed Search		
5	Informed Search		
6	Implementation of unification and resolution for real world problems.		
7	Implementation of uncertain methods for an application using Fuzzy logic/ Dempster Shafer theory		
8	Implementation of learning algorithms for an application		
9	Implementation of NLP programs		
10	Applying deep learning methods to solve an application		

## Experiment 1: Implementation of toy problems

### **Problem Statement:**

Given an array of ratings for n books. Find the minimum cost to buy all books with below conditions:

1. Cost of every book would be at-least 1 dollar.
2. A book has higher cost than an adjacent (left or right) if rating is more than the adjacent.

### **Aim:**

To find the minimum cost to buy all books.

### **Procedure:**

1. Make two arrays left2right and right2left and fill 1 in both of them.
2. Traverse left to right and fill left2right array and update it by seeing previous rating of given array. Do not care about next rating of given array.
3. Traverse right to left and fill right2left array and update it by seeing next rating of given array. Do not care about previous rating of given array.
4. Find maximum value of ith position in both array (left2right and right2left) and add it to result

### **Program:**

```
def costToBuy(ratings, n):
    left2right = [1]*n
    right2left = [1]*n

    for i in range(1, n):
        if ratings[i] > ratings[i-1]:
            left2right[i] = left2right[i-1] + 1

    for i in range(n-2, -1, -1):
        if ratings[i] > ratings[i+1]:
            right2left[i] = right2left[i+1] + 1

    res = list(map(lambda x, y: max(x, y), left2right, right2left))
    print("Cost of all books: ", res)
    return sum(res)
```

```
ratings = list(map(int, input("Enter the ratings of 'n'  
bananas:\n").strip().split()))  
print("\nCost to buy all books is: ", costToBuy(ratings, len(ratings)))
```

**Output:**

```
RA1911003010696:~/environment/RA1911003010696/Exp 1 $ python3 "Exp 1.py"  
Enter the ratings of 'n' bananas:  
1 3 4 3 7 1  
Cost of all books:  [1, 2, 3, 1, 2, 1]  
  
Cost to buy all books is:  10
```

**Result:**

Hence, minimum cost to buy all books is found.

## Experiment 2: Developing agent programs for real world problems

### Minimum Spanning Tree (Kruskal's Algorithm)

#### Aim:

Given a connected and undirected graph, find the spanning tree with a weight less than or equal to the weight of every other spanning tree.

#### Procedure:

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are  $(V-1)$  edges in the spanning tree.

#### Algorithm:

```
class UnionFind:
    def __init__(self, sz):
        self.root = [i for i in range(sz)]

    def find(self, x):
        return self.root[x]

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)
        if rootX != rootY:
            for i in range(len(self.root)):
                if self.root[i] == rootY:
                    self.root[i] = rootX

    def connected(self, x, y):
        return self.find(x) == self.find(y)

class Graph:
    def __init__(self, V):
        self.V = V
        self.adj = []

    def addEdge(self, x, y, w):
        self.adj.append((x, y, w))

    def kruskals(self):
```

```

    res = []
    cost = 0
    uf = UnionFind(self.V)

    for edge in sorted(self.adj, key = lambda e: e[2]): if
        not uf.connected(edge[0], edge[1]):
            cost += edge[2]
            res.append(edge)
            uf.union(edge[0], edge[1])

    return res, cost

v = int(input("Enter no. of vertices: "))
g = Graph(v)
e = int(input("Enter no. of edges: "))

for i in range(e):
    inp = input("Enter the vertices and weight of edge {}: ".format(i+1))
    edge = list(map(int, inp.split()))
    g.addEdge(edge[0], edge[1], edge[2])

r = g.kruskals()
print("Edges in minimum spanning tree: ", r[0]) print("Cost:
", r[1])

```

**Result:**

```

RA1911003010696:~/environment/RA1911003010696/Exp 2 $ python3 Exp2.py
Enter no. of vertices: 4
Enter no. of edges: 6
Enter the vertices and weight of edge 1: 0 1 1
Enter the vertices and weight of edge 2: 1 3 3
Enter the vertices and weight of edge 3: 3 2 4
Enter the vertices and weight of edge 4: 2 0 2
Enter the vertices and weight of edge 5: 0 3 2
Enter the vertices and weight of edge 6: 1 2 2
Edges in minimum spanning tree: [(0, 1, 1), (2, 0, 2), (0, 3, 2)]
Cost: 5

```

## Experiment 3 – Implementation of Constraint

### Satisfaction Problem

#### (Crypt Arithmetic Puzzle)

#### **Aim:**

Implementation of Constraint Satisfactory problem- Crypt Arithmetic Problem

#### **Procedure:**

1. Start.
2. Accept the expression given as input.
3. Extract the words from the input.
4. Permute for different combination of values for Left side of equal sign.
5. Check if the sum of the left value is equal to the right sum 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):
    # split equation in left and right
    left, right = equation.lower().replace(' ', '').split('=')
```



```

# split words in left part
left = left.split('+')

# create list of used letters
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)

```

**Result:**

```

RA1911003010699:~/environment/RA1911003010696/Exp 3 $ python3 Exp3.py
Enter an equation: LEFT + LEE = ALL
189 + 11 = 200 (mapping: {'e': 1, 't': 9, 'a': 2, 'f': 8, 'l': 0})
278 + 22 = 300 (mapping: {'e': 2, 't': 8, 'a': 3, 'f': 7, 'l': 0})
367 + 33 = 400 (mapping: {'e': 3, 't': 7, 'a': 4, 'f': 6, 'l': 0})
634 + 66 = 700 (mapping: {'e': 6, 't': 4, 'a': 7, 'f': 3, 'l': 0})
723 + 77 = 800 (mapping: {'e': 7, 't': 3, 'a': 8, 'f': 2, 'l': 0})
812 + 88 = 900 (mapping: {'e': 8, 't': 2, 'a': 9, 'f': 1, 'l': 0})

```

## Experiment 4:

### Uninformed Search

#### Shortest Path using

#### BFS

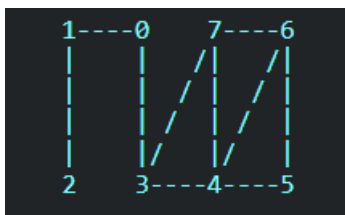
#### Aim:

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:

```
RA1911003010696:~/environment/RA1911003010696/Exp 4 $ python3 BFS.py
Enter the no. of Vertices: 8
Enter the no. of Edges: 10
Enter the vertices edge 1: 1 2
Enter the vertices edge 2: 1 0
Enter the vertices edge 3: 0 3
Enter the vertices edge 4: 3 7
Enter the vertices edge 5: 3 4
Enter the vertices edge 6: 4 7
Enter the vertices edge 7: 7 6
Enter the vertices edge 8: 4 6
Enter the vertices edge 9: 4 5
Enter the vertices edge 10: 5 6
Enter the source: 0
Enter the destination: 5
Shortest path: [0, 3, 4, 5]
Time taken by BFS: 1.0011601448059082
```

### Shortest Path using DFS

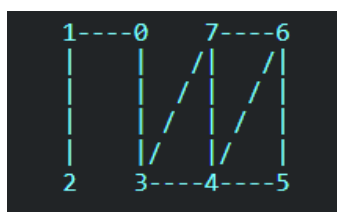
#### Aim:

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:

```

RA1911003010696:~/environment/RA1911003010696/Exp 4 $ python3 DFS.py
Enter the no. of Vertices: 8
Enter the no. of Edges: 10
Enter the vertices of edge 1: 1 2
Enter the vertices of edge 2: 1 0
Enter the vertices of edge 3: 0 3
Enter the vertices of edge 4: 3 7
Enter the vertices of edge 5: 3 4
Enter the vertices of edge 6: 4 7
Enter the vertices of edge 7: 7 6
Enter the vertices of edge 8: 4 5
Enter the vertices of edge 9: 4 6
Enter the vertices of edge 10: 5 6
Enter the source: 0
Enter the destination: 5
Shortest Path: (0, 3, 4, 5)
Time taken by BFS: 1.0012683868408203

```

#### 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:

### Informed Search

#### **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):
```

```
        parent = {}
```

```
        success = False
```

```
        open = [(s, h1)]
```

```
        closed = []
```

```
        parent[s] = None
```

```
        while open and not success:
```

```
            t = open.pop(0)
```

```
            print(t[0])
```

```
            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)
        parent[neighbor[0]] = t[0]
        open.sort(key = lambda t: t[1])

    if success:
        path = []
        n = d
        while parent[n] != None:
            path.append(n)
            n = parent[n]
        path.append(s)
        print("Path found: {}".format(path[::-1]))
    else:
        print("No path found!!!")

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

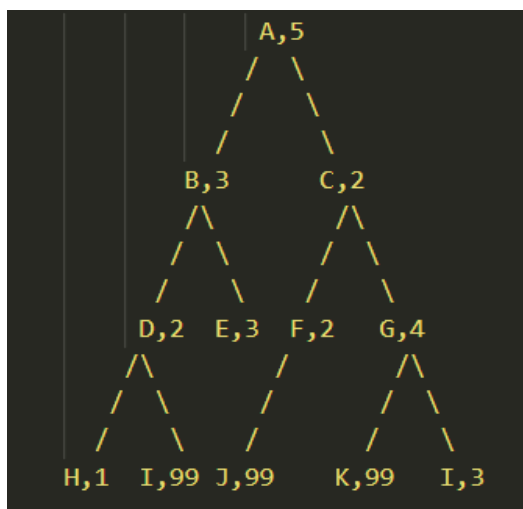
heuristics = dict()
for i in range(v):
    ver_h = input("Enter vertex {} and its heuristic: ".format(i+1)).strip().split()
    heuristics[ver_h[0]] = int(ver_h[1])
    # print(ver_h[0], int(ver_h[1]))

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:

```
Enter the no. vertices: 12
Enter vertex 1 and its heuristic: A 5
Enter vertex 2 and its heuristic: B 3
Enter vertex 3 and its heuristic: C 2
Enter vertex 4 and its heuristic: D 2
Enter vertex 5 and its heuristic: E 3
Enter vertex 6 and its heuristic: F 2
Enter vertex 7 and its heuristic: G 4
Enter vertex 8 and its heuristic: H 1
Enter vertex 9 and its heuristic: I 99
Enter vertex 10 and its heuristic: J 99
Enter vertex 11 and its heuristic: K 99
Enter vertex 12 and its heuristic: I 3
Enter the no. edges: 11
Enter the vertices of edge 1: A B
Enter the vertices of edge 2: A C
Enter the vertices of edge 3: B D
Enter the vertices of edge 4: B E
Enter the vertices of edge 5: C F
Enter the vertices of edge 6: C G
Enter the vertices of edge 7: D H
Enter the vertices of edge 8: D I
Enter the vertices of edge 9: F J
Enter the vertices of edge 10: G K
Enter the vertices of edge 11: G I
Enter the source: A
Enter the destination: H
A
C
F
B
D
H
Path found: ['A', 'B', 'D', 'H']
```

### 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 = openSet[0]
        if len(openSet) > 1:
            for v in openSet[1:]:
                if g[v] + heuristic[v] < g[n] + heuristic[n]:
                    n = v
            if n == des or not Graph[n]:
                pass
            print(n)
            # 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])

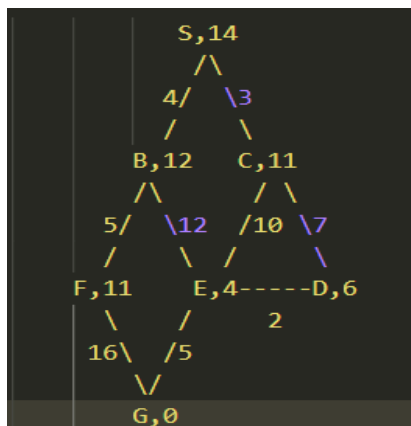
e = int(input("Enter the no. edges: "))
for i in range(e):
    edge = input("Enter the vertices of edge {} along with the weight: ".format(i+1)).strip().split()
    Graph[edge[0]].append((edge[1], int(edge[2])))

# print(Graph)

s = input("Enter the source: ")
d = input("Enter the destination: ")
aStar(s, d)

```

**Input Graph:**



### Output:

```
RA1911003010699:~/environment/RA1911003010696/Exp 5 $ python3 A*.py
Enter the no. vertices: 7
Enter vertex 1 and its heuristic: S 14
Enter vertex 2 and its heuristic: B 12
Enter vertex 3 and its heuristic: C 11
Enter vertex 4 and its heuristic: F 11
Enter vertex 5 and its heuristic: E 4
Enter vertex 6 and its heuristic: D 6
Enter vertex 7 and its heuristic: G 0
Enter the no. edges: 9
Enter the vertices of edge 1 along with the weight: S B 4
Enter the vertices of edge 2 along with the weight: S C 3
Enter the vertices of edge 3 along with the weight: B F 5
Enter the vertices of edge 4 along with the weight: B E 12
Enter the vertices of edge 5 along with the weight: C E 10
Enter the vertices of edge 6 along with the weight: C D 7
Enter the vertices of edge 7 along with the weight: D E 2
Enter the vertices of edge 8 along with the weight: F G 16
Enter the vertices of edge 9 along with the weight: E G 5
Enter the source: S
Enter the destination: G
S
C
B
D
E
G
Path found: ['S', 'C', 'D', 'E', 'G']
```

### Result:

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

**Experiment 6: Implementation of unification**  
**and resolution for real world problems**

**Aim:**

Implementation of unification and resolution for real world problems.

**Algorithm (unification):**

Step. 1: If  $\Psi_1$  or  $\Psi_2$  is a variable or constant, then:

- a) If  $\Psi_1$  or  $\Psi_2$  are identical, then return NIL.
- b) Else if  $\Psi_1$  is a variable,
  - a. then if  $\Psi_1$  occurs in  $\Psi_2$ , then return FAILURE
  - b. Else return  $\{ (\Psi_2 / \Psi_1) \}$ .
- c) Else if  $\Psi_2$  is a variable,
  - a. If  $\Psi_2$  occurs in  $\Psi_1$  then return FAILURE,
  - b. Else return  $\{ (\Psi_1 / \Psi_2) \}$ .
- d) Else return FAILURE.

Step.2: If the initial Predicate symbol in  $\Psi_1$  and  $\Psi_2$  are not same, then return FAILURE.

Step. 3: IF  $\Psi_1$  and  $\Psi_2$  have a different number of arguments, then return FAILURE. Step.

4: Set Substitution set(SUBST) to NIL.

Step. 5: For  $i=1$  to the number of elements in  $\Psi_1$ .

- a) Call Unify function with the  $i$ th element of  $\Psi_1$  and  $i$ th element of  $\Psi_2$ , and put the result into S.
- b) If S = failure then returns Failure
- c) If  $S \neq \text{NIL}$  then do,
  - a. Apply S to the remainder of both L1 and L2.
  - b. SUBST= APPEND(S, SUBST).

Step.6: Return SUBST.

**Algorithm (resolution):**

1. Conversion of facts into first-order logic.
2. Convert FOL statements into CNF

- ### Traugott Rules:

likes(bob, X)  $\leftarrow$  likes(X, logic)  
likes(bob, logic)

likes(U, V)  $\leftrightarrow$  (U = bob  $\wedge$  likes(V, logic))  
 $\vee$  (U = bob  $\wedge$  V = logic)  
 $\neg$  logic = bob  
 $\neg$  bob = logic

$\leftarrow$  likes(logic, logic)

disjunction  
of alternatives in the  
search space

unification of  
logic and bob  
fails

unification of  
logic and bob  
fails

$\neg$  likes(logic, logic)

$\wedge$

$\neg$  logic = bob  $\vee$   
 $\neg$  likes(logic, logic)

$\neg$  logic = bob

$\neg$  logic = bob  $\vee$   
 $\neg$  logic = logic

$\neg$  logic = bob

### 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 = input("Enter first exp: ") #'Q(a, g(x, a), f(y))'
    f2 = input("Enetr second exp: ") #'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:

```

Enter first exp: Q(a, g(x, a), f(y))
Enetr second exp: Q(a, g(f(b), a), x)
The process of Unification successful!
['f(b)/x', 'f(y)/x']

```



## 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 sentenceldx in range(len(self.inputSentences)):
            # Do negation of the Premise and add them as literal
            if "=>" in self.inputSentences[sentenceldx]:
                self.inputSentences[sentenceldx] = negateAntecedent(
                    self.inputSentences[sentenceldx])

    def askQueries(self, queryList):

```

[illegible]

```

        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):
        return 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("Exp 6\input.txt")
    knowledgeBase = KB(inputSentences_)
    knowledgeBase.prepareKB()
    results_ = knowledgeBase.askQueries(inputQueries_)
    printOutput("Exp 6\output.txt", results_)

```

### Input:

6

F(Joe)

H(John)

~H(Alice)

~H(John)

G(Joe)

G(Tom)

14

~F(x) | G(x)

~G(x) | H(x)

~H(x) | F(x)

~R(x) | H(x)

~A(x) | H(x)

~D(x,y) | ~H(y)

~B(x,y) | ~C(x,y) | A(x)

B(John,Alice)

B(John,Joe)

$\sim D(x,y) \mid \sim Q(y) \mid C(x,y)$

D(John,Alice)

Q(Joe)

D(John,Joe)

R(Tom)

**Output:**

```
[ 'FALSE', 'TRUE', 'TRUE', 'FALSE', 'FALSE', 'TRUE' ]
```

**Result:** Unification and resolution of expression was done and the conversion set was printed and the result of

all queries in input file were printed.

**Experiment 7: Implementation of**  
**uncertain methods for an application**  
**using Fuzzy logic**

**Aim:** Implementation of uncertain methods for an application using Fuzzy logic/ Dempster Shafer Theory.

**Algorithm:**

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

**Program:**

```
news = ["B01","B02","B03","B04","B05","B06","B07","B08","B09","B10","B11","B12","B13","B14","B15","B16","B17","B18","B19","B20","B21","B22","B23","B24","B25","B26","B27","B28","B29","B30"]
```

```
emotion =  
[97,36,63,82,71,79,55,57,40,57,77,68,60,82,40,80,60,50,100,11,58,68,64,57,77,98,91,50,95,27]
```

```
provoke =  
[74,85,43,90,25,81,62,45,65,45,70,75,70,90,85,68,72,95,18,99,63,70,66,77,55,64,59,95,55,79]
```

```
def checkEmotion(x):
```

```
    eLow, eMedium, eHigh = 0,0,0
```

```
    if x >= 0 and x <= 35:
```

```
        eLow = 1
```

```
    elif x > 35 and x < 39:
```

```
        eLow = (-1*((x-39)/(39-35)))
```

```
        eMedium = ((x-35)/(39-35))
```

```
    elif x >= 39 and x <= 61:
```

```
        eMedium = 1
```

*elif*  $x > 61$  and  $x < 65$ :

$eMedium = (-1 * ((x-65)/(65-61)))$

$eHigh = ((x-61)/(65-61))$

*elif*  $x \geq 65$ :

$eHigh = 1$

*return*  $eLow, eMedium, eHigh$

*def* *checkProvoke*( $x$ ):

$pLow, pMedium, pHigh = 0, 0, 0$

*if*  $x \geq 0$  and  $x \leq 55$ :

$pLow = 1$

*elif*  $x > 55$  and  $x < 60$ :

$pLow = (-1 * (x-60)/(60-55))$

$pMedium = ((x-55)/(60-55))$

*elif*  $x \geq 60$  and  $x \leq 85$ :

$pMedium = 1$

*elif*  $x > 85$  and  $x < 87$ :

$pMedium = (-1 * (x-87)/(82-87))$

$pHigh = ((x-85)/(87-85))$

*elif*  $x \geq 87$ :

$pHigh = 1$

*return*  $pLow, pMedium, pHigh$

*def* *inference*( $eLow, eMedium, eHigh, pLow, pMedium, pHigh$ ):

$Y1, Y2, Y3, Y4, Y5 = 0, 0, 0, 0, 0$

$N1, N2, N3, N4 = 0, 0, 0, 0$

$Y, N = 0, 0$

*if*  $eHigh \neq 0$  and  $pHigh \neq 0$ :

$Y1 = \min(eHigh, pHigh)$

*if*  $eHigh \neq 0$  and  $pMedium \neq 0$ :

$Y2 = \min(eHigh, pMedium)$

*if*  $eHigh \neq 0$  and  $pLow \neq 0$ :

$N1 = \min(eHigh, pLow)$



*if eMedium != 0 and pHigh != 0:*

*Y3 = min(eMedium, pHigh)*

*if eMedium != 0 and pMedium != 0:*

*N2 = min(eMedium, pMedium)*

*if eMedium != 0 and pLow != 0:*

*N3 = min(eMedium, pLow)*

*if eLow != 0 and pHigh != 0:*

*Y4 = min(eLow, pHigh)*

*if eLow != 0 and pMedium != 0:*

*Y5 = min(eLow, pMedium)*

*if eLow != 0 and pLow != 0:*

*N4 = min(eLow, pLow)*

*Y = max(Y1, Y2, Y3, Y4, Y5)*

*N = max(N1, N2, N3, N4)*

*return Y, N*

*def defuzzification(Y, N):*

*if Y != 0 and N != 0:*

*return ((Y\*60)+(N\*40))/(Y+N)*

*elif Y != 0:*

*return (Y\*60)/Y*

*elif N != 0:*

*return (N\*40)/N*

*count = 0*

*while count < 30:*

*eLow, eMedium, eHigh = checkEmotion(emotion[count])*

*pLow, pMedium, pHigh = checkProvoke(provoke[count])*

*Ya, Tidak = inference(eLow, eMedium, eHigh, pLow, pMedium, pHigh)*

*hasil = defuzzification(Ya, Tidak)*

*if hasil < 55:*

*hoax = "No"*

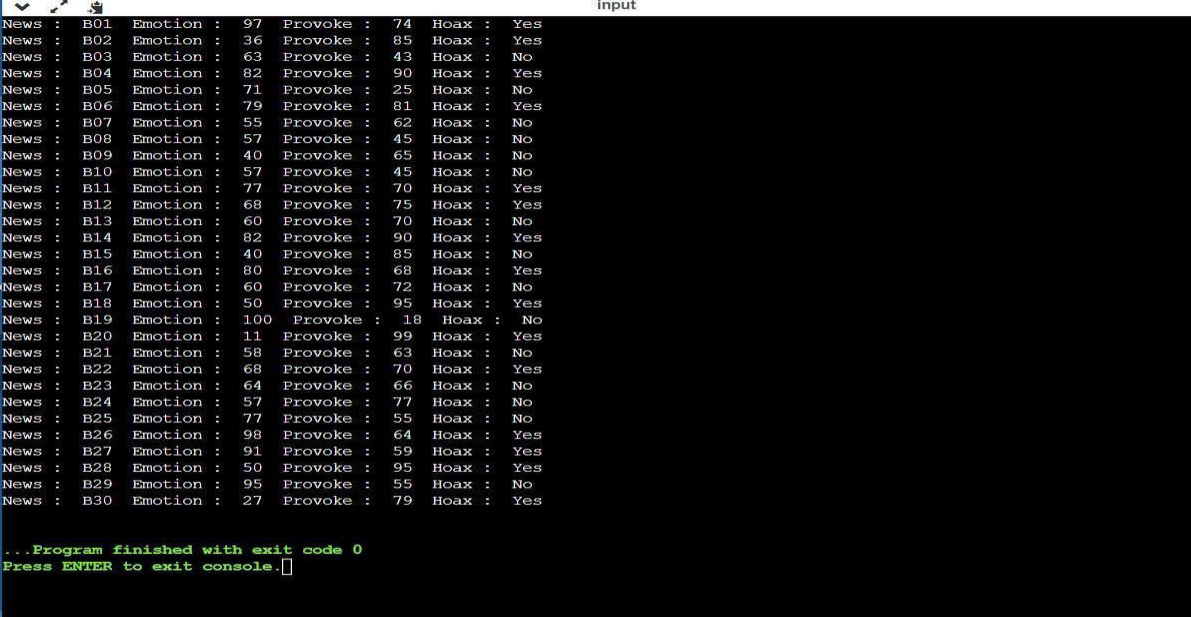
*elif hasil >= 55:*

```
hoax = "Yes"
```

```
print("News : ",news[count]," Emotion : ",emotion[count]," Provoke : ",provoke[count]," Hoax : ", hoax)
```

```
count += 1
```

### Output:



```
Input
News : B01 Emotion : 97 Provoke : 74 Hoax : Yes
News : B02 Emotion : 36 Provoke : 85 Hoax : Yes
News : B03 Emotion : 63 Provoke : 43 Hoax : No
News : B04 Emotion : 82 Provoke : 90 Hoax : Yes
News : B05 Emotion : 71 Provoke : 25 Hoax : No
News : B06 Emotion : 79 Provoke : 81 Hoax : Yes
News : B07 Emotion : 55 Provoke : 62 Hoax : No
News : B08 Emotion : 57 Provoke : 45 Hoax : No
News : B09 Emotion : 40 Provoke : 65 Hoax : No
News : B10 Emotion : 57 Provoke : 45 Hoax : No
News : B11 Emotion : 77 Provoke : 70 Hoax : Yes
News : B12 Emotion : 68 Provoke : 75 Hoax : Yes
News : B13 Emotion : 60 Provoke : 70 Hoax : No
News : B14 Emotion : 82 Provoke : 90 Hoax : Yes
News : B15 Emotion : 40 Provoke : 85 Hoax : No
News : B16 Emotion : 80 Provoke : 68 Hoax : Yes
News : B17 Emotion : 60 Provoke : 72 Hoax : No
News : B18 Emotion : 50 Provoke : 95 Hoax : Yes
News : B19 Emotion : 100 Provoke : 18 Hoax : No
News : B20 Emotion : 11 Provoke : 99 Hoax : Yes
News : B21 Emotion : 58 Provoke : 63 Hoax : No
News : B22 Emotion : 68 Provoke : 70 Hoax : Yes
News : B23 Emotion : 64 Provoke : 66 Hoax : No
News : B24 Emotion : 57 Provoke : 77 Hoax : No
News : B25 Emotion : 77 Provoke : 55 Hoax : No
News : B26 Emotion : 98 Provoke : 64 Hoax : Yes
News : B27 Emotion : 91 Provoke : 59 Hoax : Yes
News : B28 Emotion : 50 Provoke : 95 Hoax : Yes
News : B29 Emotion : 95 Provoke : 55 Hoax : No
News : B30 Emotion : 27 Provoke : 79 Hoax : Yes

...Program finished with exit code 0
Press ENTER to exit console.
```

### Result:

Implementation of uncertain methods for an application is successfully implemented.

## Experiment 8: Implementation of learning algorithms for an application

### Linear Regression

#### Aim:

Implementation of Linear Regression algorithm to predict using the given dataset.

#### Procedure:

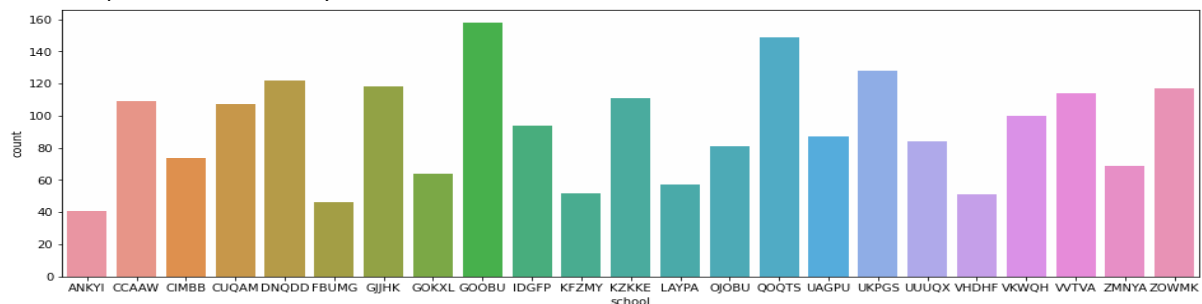
```
# importing libraries import numpy
as np import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns import plotly
as px import warnings
warnings.filterwarnings("ignore") df =
pd.read_csv("test_scores.csv") df.info()

<class 'pandas.core.frame.DataFrame'> RangeIndex:
2133 entries, 0 to 2132 Data columns (total 11 columns):
#      Column      Non-Null Count  Dtype
---  -
0     school      2133 non-null    object
1     school_setting  2133 non-null    object
2     school_type    2133 non-null    object
3     classroom      2133 non-null    object
4     teaching_method  2133 non-null    object
5     n_student      2133 non-null    float64
6     student_id     2133 non-null    object
7     gender         2133 non-null    object
8     lunch          2133 non-null    object
9     pretest        2133 non-null    float64
10    posttest       2133 non-null    float64
dtypes: float64(3), object(8) memory usage: 183.4+
KB
```

#### data analysis and visualisation :

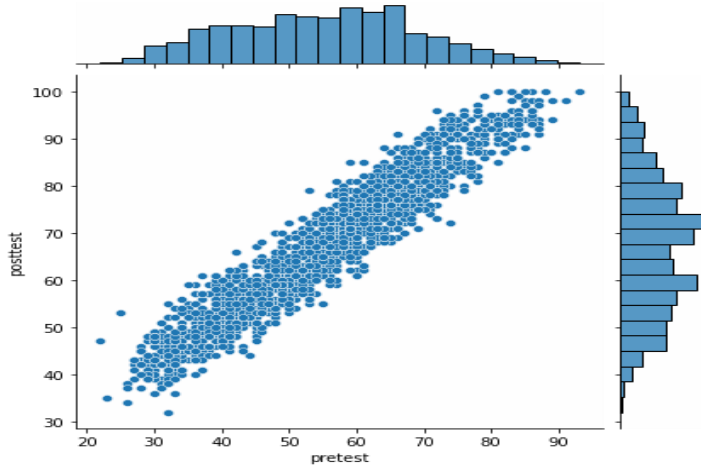
```
plt.figure(figsize=(16,5))
sns.countplot(data=df,x="school")

<AxesSubplot:xlabel='school',ylabel='count'>
```



```
sns.jointplot(df["pretest"],df["posttest"])
```

```
<seaborn.axisgrid.JointGrid at 0x2397d4b35e0>
```



```
sns.boxplot(df["school"],df["posttest"])
```

## Data Cleaning :

```
from sklearn.model_selection import train_test_split X =
```

```
df["pretest"].values.reshape(-1,1)
```

```
y = df["posttest"]
```

```
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.3)
```

```
from sklearn.linear_model import LinearRegression svr = LinearRegression()
```

```
svr.fit(X_train,y_train) pred =
```

```
svr.predict(X_test)
```

```
from sklearn.metrics import mean_absolute_error,mean_squared_error
```

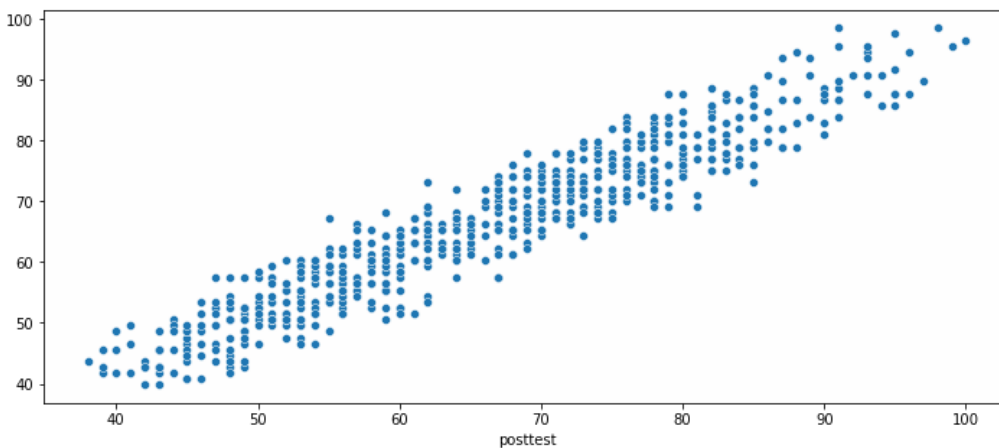
```
print(mean_absolute_error(y_test,pred)) print(np.sqrt(mean_squared_error(y_test,pred)))
```

```
3.586352028437999
```

```
4.342623648767632
```

```
plt.figure(figsize=(12,5)) sns.scatterplot(x=y_test,y=pred)
```

```
<AxesSubplot:xlabel='posttest'>
```



**Result:** Linear regression has been implemented successfully.

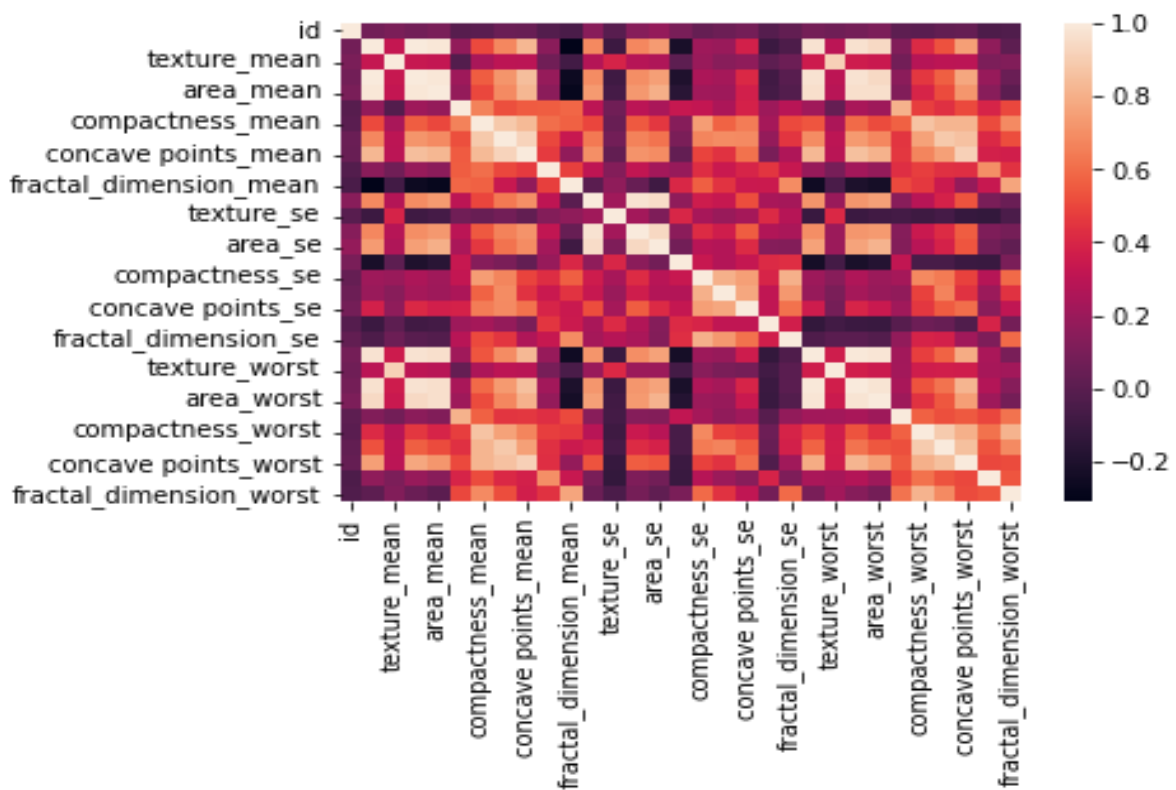
## Support Vector Classification

### Aim:

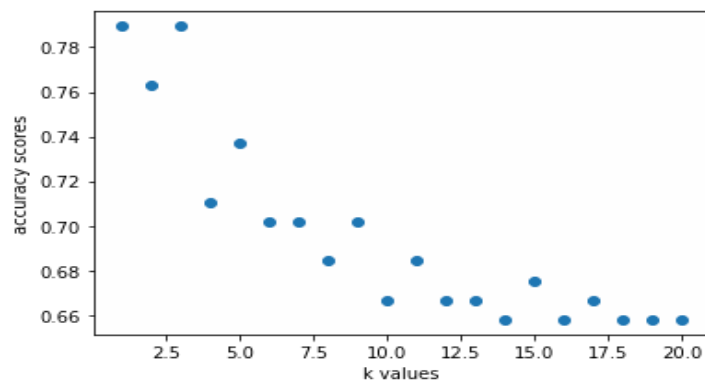
Implementation of Support Vector Classification algorithm to classify like the cases of breast Cancer

### Procedure:

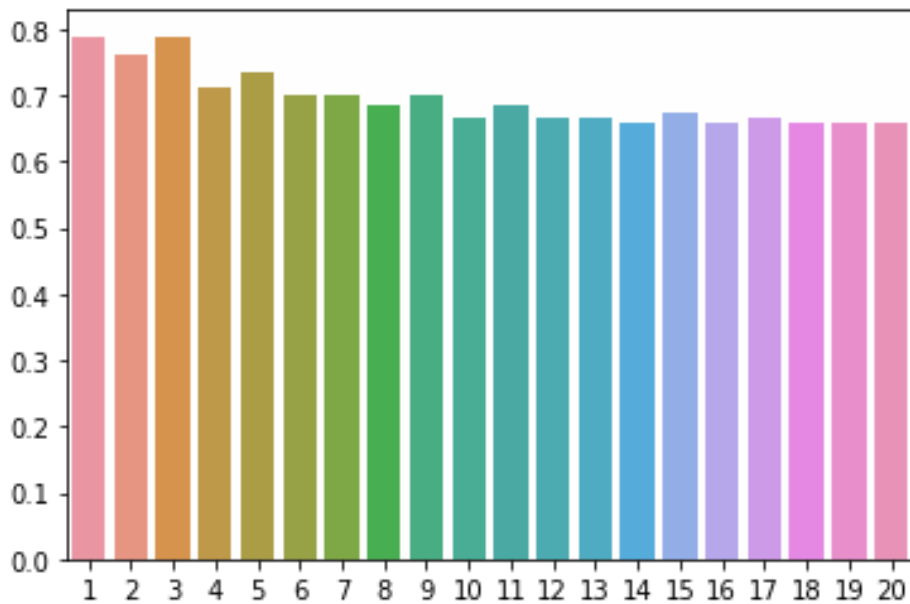
```
import numpy as np
import pandas as pd df=pd.read_csv('data.csv')
df=df.drop(['Unnamed:32'],axis=1) sns.heatmap(df.corr())
```



```
plt.scatter(range(1,21),list_1) plt.xlabel('k values')
plt.ylabel('accuracy scores') plt.show()
```



```
sns.barplot(x=list(range(1,21)),y=list_1)
```



### Result:

Hence, support vector classifier has tested and implemented successfully.

## K-Means Clustering

### Aim:

Implementation of K-means clustering algorithm to group the customers based on the demographic.

### Procedure:

KMeans Clustering comes in handy when you want to group things together based on the similar qualities they share.

In this dataset, each row represents a customer who has certain qualities like Annual Income and Spending Score. In this notebook we will not be using sklearn. Instead we will code the steps involved in Clustering and visualise what's happening.

### Main Concept

Two data points (or two customers) are considered similar if they are closer together in mathematical space. We can use Euclidean distance as the formula to measure this distance.

### Import Libraries

```
import numpy as np
import pandas as pd
```

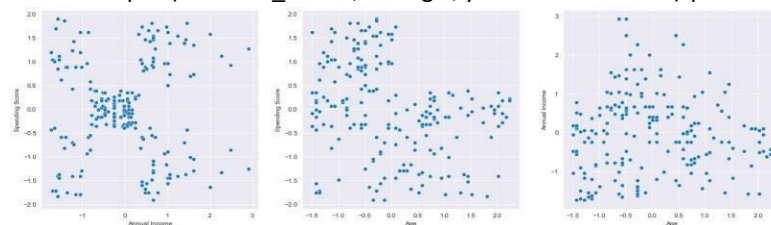
```
import matplotlib.pyplot as plt
import seaborn as sns
```

```
sns.set_style('darkgrid')
df=pd.read_csv('Mall_Customers.csv')
from sklearn.preprocessing import StandardScaler ss = StandardScaler()
df_scaled = pd.DataFrame(ss.fit_transform(df), columns = df.columns) df_scaled.head()
```

	Age	Annual Income	Spending Score
0	-1.424569	-1.738999	-0.434801
1	-1.281035	-1.738999	1.195704
2	-1.352802	-1.700830	-1.715913
3	-1.137502	-1.700830	1.040418
4	-0.563369	-1.662660	-0.395980

## Plotting the data

```
plt.figure(figsize = (20,6)) plt.subplot(1,3,1)
sns.scatterplot(data = df_scaled, x = 'Annual Income', y = 'Spending Score')
plt.subplot(1,3,2)
sns.scatterplot(data = df_scaled, x = 'Age', y = 'Spending Score') plt.subplot(1,3,3)
sns.scatterplot(data = df_scaled, x = 'Age', y = 'Annual Income') plt.show()
```



## Choose a value for K

```
centroids = df_scaled.sample(5, random_state = 10)
```

Here are the coordinates of our randomly chosen centroids. Let's plot them.

```
plt.figure(figsize = (20,6)) plt.subplot(1,3,1)
```

```
sns.scatterplot(data = df_scaled, x = 'Annual Income', y = 'Spending Score', alpha = 0.2)
```

```
for i in range(5):
```

```
plt.scatter(x = centroids['Annual Income'].iloc[i], y =
centroids['Spending Score'].iloc[i], marker = '$%d$' % (i), s=60,color =
'Black') plt.subplot(1,3,2)
```

```
sns.scatterplot(data = df_scaled, x = 'Age', y = 'Spending Score', alpha =
0.2)
```

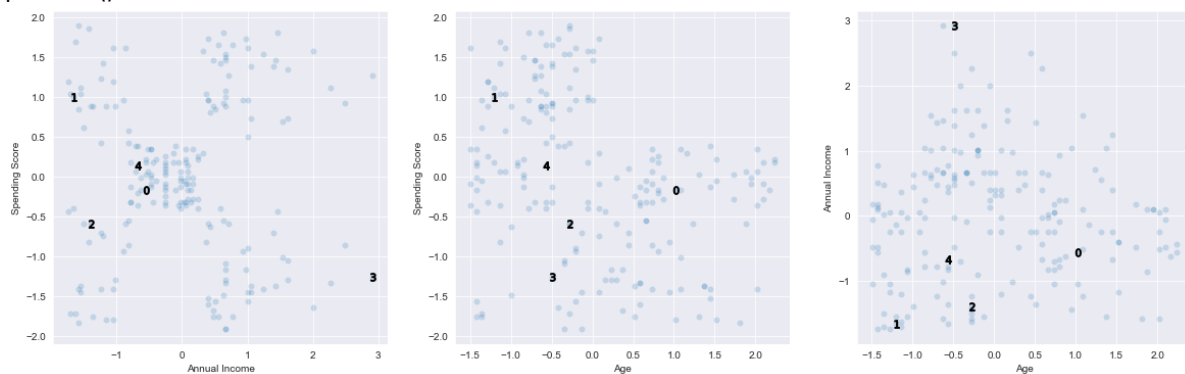
```
for i in range(5):
    plt.scatter(x = centroids['Age'].iloc[i], y = centroids['Spending Score'].iloc[i], marker = '$%d$' % (i),
s=60,color = 'Black')
```

```
plt.subplot(1,3,3)
```

```
sns.scatterplot(data = df_scaled, x = 'Age', y = 'Annual Income', alpha =
0.2)
```

```
for i in range(5):
    plt.scatter(x = centroids['Age'].iloc[i], y = centroids['Annual Income'].iloc[i], marker = '$%d$' % (i),
s=60,color = 'Black')
```

```
plt.show()
```



```
from scipy.spatial.distance import cdist
```

```
distances = pd.DataFrame(cdist(df_scaled, centroids,'euclidean'), columns =
['Distance From C1', 'Distance From C2', 'Distance From C3', 'Distance
From C4', 'Distance From C5'])
```

### Assign cluster labels to each point

We assign the label to each point based on whichever centroid it is closest to.

```
cluster_labels = pd.Series(np.argmin(distances.values, axis = 1))
```

```
# displaying labels assigned to the first five rows
cluster_labels.head()
```

```
import matplotlib.cm as cm plt.figure(figsize =
(20,6)) plt.subplot(1,3,1)
sns.scatterplot(data = df_scaled, x = 'Annual Income', y = 'Spending Score', alpha = 1,hue = cluster_labels,
palette='Spectral' )
```

```
for i in range(5):
```



```
plt.scatter(x = centroids['Annual Income'].iloc[i], y =
centroids['Spending Score'].iloc[i], s=200, color = 'White') plt.scatter(x = centroids['Annual Income'].iloc[i], y
=
centroids['Spending Score'].iloc[i], marker = '$%d$' % (i), s=40, color =
'black') plt.subplot(1,3,2)
```

```
sns.scatterplot(data = df_scaled, x = 'Age', y = 'Spending Score', alpha =
1, hue = cluster_labels, palette='Spectral')
```

```
for i in range(5):
```

```
    plt.scatter(x = centroids['Age'].iloc[i], y = centroids['Spending Score'].iloc[i], s=200, color = 'White')
```

```
    plt.scatter(x = centroids['Age'].iloc[i], y = centroids['Spending Score'].iloc[i], marker = '$%d$' % (i),
s=40, color = 'Black')
```

```
plt.subplot(1,3,3)
```

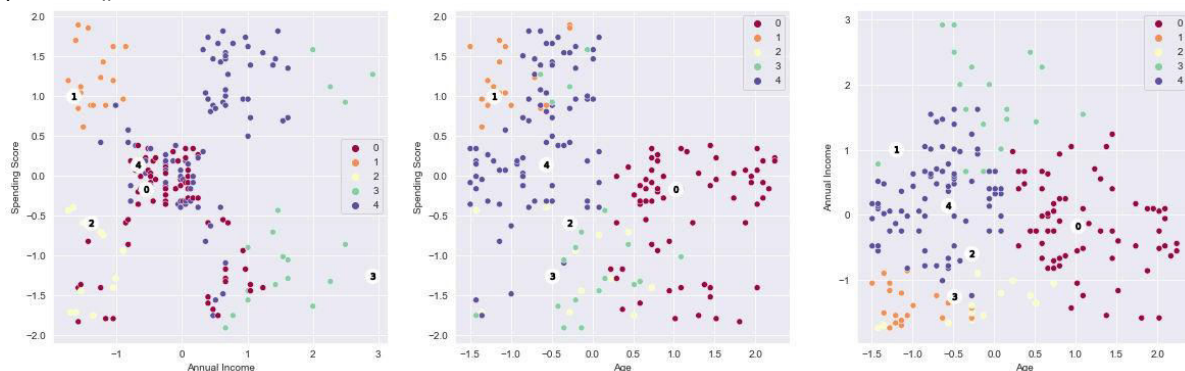
```
sns.scatterplot(data = df_scaled, x = 'Age', y = 'Annual Income', alpha =
1, hue = cluster_labels, palette='Spectral')
```

```
for i in range(5):
```

```
    plt.scatter(x = centroids['Age'].iloc[i], y = centroids['Spending Score'].iloc[i], s=200, color = 'White')
```

```
    plt.scatter(x = centroids['Age'].iloc[i], y = centroids['Spending Score'].iloc[i], marker = '$%d$' % (i),
s=40, color = 'Black')
```

```
plt.show()
```



Again look at the leftmost plot. We see it clearly. The centroids are not optimal representations of the centre of the cluster. The centroid of cluster 4 is quite far from most of it's data points.

## Update the centroid and identify the new clusters

```
new_centroids = []
```

```
for i in range(K): new_centroids.append(list(df_scaled[cluster_labels==
i].mean().values))
```

```
new_centroids = pd.DataFrame(new_centroids, columns = df_scaled.columns, index = ['C1',
'C2', 'C3', 'C4', 'C5'])
```

```
plt.figure(figsize = (20,6))
```

```
plt.subplot(1,3,1)
```

```
sns.scatterplot(data = df_scaled, x = 'Annual Income', y = 'Spending Score', alpha = 1,hue = cluster_labels,
palette='Spectral' )
```

```
for i in range(5):
```

```
    plt.scatter(x = new_centroids['Annual Income'].iloc[i], y =
new_centroids['Spending Score'].iloc[i], s=200, color = 'White') plt.scatter(x = new_centroids['Annual
Income'].iloc[i], y =
new_centroids['Spending Score'].iloc[i], marker = '$%d$' %(i), s=40, color
= 'black') plt.subplot(1,3,2)
```

```
sns.scatterplot(data = df_scaled, x = 'Age', y = 'Spending Score', alpha =
1,hue = cluster_labels, palette='Spectral')
```

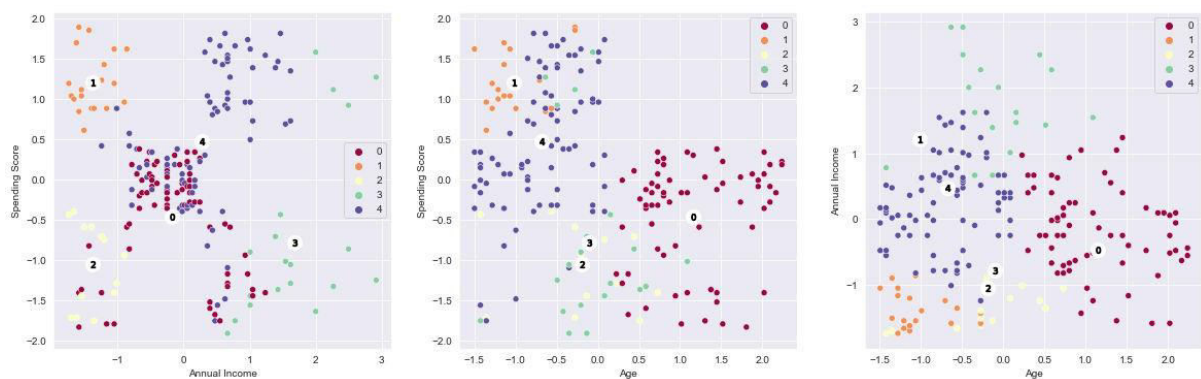
```
for i in range(5):
```

```
    plt.scatter(x = new_centroids['Age'].iloc[i], y =
new_centroids['Spending Score'].iloc[i], s=200, color = 'White') plt.scatter(x = new_centroids['Age'].iloc[i], y =
new_centroids['Spending Score'].iloc[i], marker = '$%d$' %(i), s=40, color
= 'Black') plt.subplot(1,3,3)
```

```
sns.scatterplot(data = df_scaled, x = 'Age', y = 'Annual Income', alpha =
1,hue = cluster_labels, palette='Spectral')
```

```
for i in range(5):
```

```
    plt.scatter(x = new_centroids['Age'].iloc[i], y =
new_centroids['Spending Score'].iloc[i], s=200, color = 'White') plt.scatter(x = new_centroids['Age'].iloc[i], y =
new_centroids['Spending Score'].iloc[i], marker = '$%d$' %(i), s=40,color =
'Black') plt.show()
```



**Result:**

Hence, the k means clustering has been implemented successfully.

## Apriori Algorithm

**Aim:**

Implementation of Apriori algorithm to find the frequent products sets and relevant association rules from a sales database.

### Procedure:

### Reading and Cleaning Data

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

df = pd.read_csv("bread basket.csv")

# Count of unique customers
df['Transaction'].nunique() df['date'] =
df['date_time'].dt.date

#Extracting time
df['time'] = df['date_time'].dt.time

# Extracting month and replacing it with text
df['month'] = df['date_time'].dt.month
df['month'] = df['month'].replace((1,2,3,4,5,6,7,8,9,10,11,12),

('January','February','March','April','May','June','July','August', 'September','October','November','December'))

# Extracting hour
df['hour'] = df['date_time'].dt.hour
# Replacing hours with text
hour_in_num=(1,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23)
hour_in_obj=('1-2','7-8','8-9','9-10','10-11','11-12','12-13','13-
14','14-15',
                '15-16','16-17','17-18','18-19','19-20','20-21','21-22','22-
23','23-24')
df['hour'] = df['hour'].replace(hour_in_num, hour_in_obj)

# Extracting weekday and replacing it with text df['weekday'] =
df['date_time'].dt.weekday df['weekday'] = df['weekday'].replace((0,1,2,3,4,5,6),

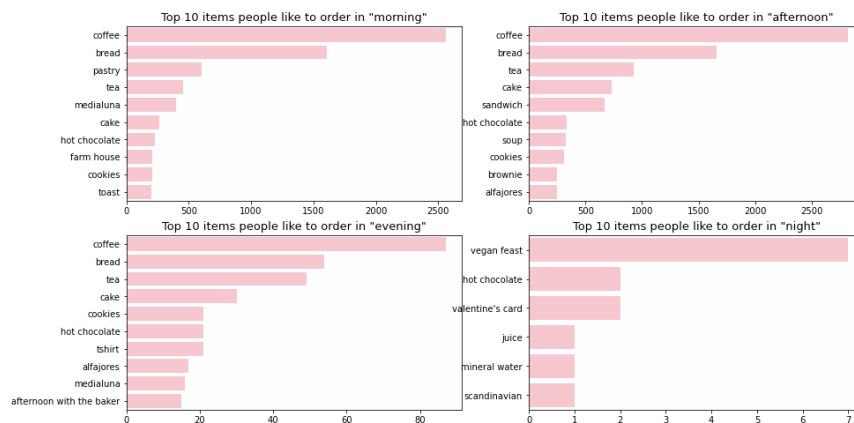
('Monday','Tuesday','Wednesday','Thursday','Friday','Saturday','Sunday'))

# dropping date_time column
df.drop('date_time', axis = 1, inplace = True)

dates = df.groupby('date')['Transaction'].count().reset_index() dates =
dates[dates['Transaction']>=200].sort_values('date').reset_index(drop = True)
data =
df.groupby(['period_day','Item'])['Transaction'].count().reset_index().sort
_values(['period_day','Transaction'],ascending=False) day =
['morning','afternoon','evening','night']
```

```
plt.figure(figsize=(15,8))
for i,j in enumerate(day):
    plt.subplot(2,2,i+1)
    df1 = data[data.period_day==j].head(10)
    sns.barplot(data=df1,y=df1.Item,x=df1.Transaction,color='pink') plt.xlabel("")
    plt.ylabel("")
    plt.title('Top 10 items people like to order in "{}".format(j), size=13)

plt.show()
```



```
import mlxtend
from mlxtend.frequent_patterns import apriori
from mlxtend.frequent_patterns import association_rules

transactions_str = df.groupby(['Transaction', 'Item'])['Item'].count().reset_index(name='Count')
transactions_str

my_basket = transactions_str.pivot_table(index='Transaction', columns='Item', values='Count',
aggfunc='sum').fillna(0)
def encode(x):
    if x <= 0:
        return 0
    if x >= 1:
        return 1
```

*# applying the function to the dataset*

```
my_basket_sets = my_basket.applymap(encode)
my_basket_sets.head()
```

In [ ]:

```
# using the 'apriori algorithm' with min_support=0.01 (1% of 9465)
# It means the item should be present in atleast 94 transaction out of 9465 transactions only when we
considered that item in
# frequent itemset
frequent_items = apriori(my_basket_sets, min_support = 0.01, use_colnames = True)
frequent_items

rules = association_rules(frequent_items, metric = "lift", min_threshold =
1)
rules.sort_values('confidence', ascending = False, inplace = True) rules
```

Out[ ]:									
	antecedent s	consequent s	antecedent support	consequent support	support	confidence	lift	leverage	conviction
31	(toast) 0.478394	(coffee)	0.033597		0.02366 6 3	0.70440	1.47243 1	0.00759 3	1.764582
29	(spanish brunch) 0.478394	(coffee)	0.018172		0.01088 2 7	0.59883	1.25176 6	0.00218 9	1.300235
19	(medialuna ) 0.478394	(coffee)	0.061807		0.03518 2 1	0.56923	1.18987 8	0.00561 4	1.210871
23	(pastry) 0.478394	(coffee)	0.086107		0.04754 4 7	0.55214	1.15416 8	0.00635 1	1.164682
1	(alfajores)	(coffee)	0.036344	0.478394	0.01965 1	0.540698	1.13023 5	0.00226 4	1.135648
17	(juice) 0.478394	(coffee)	0.038563		0.02060 2 7	0.53424	1.11675 0	0.00215 4	1.119919
25	(sandwich) 0.478394	(coffee)	0.071844		0.03824 6 3	0.53235	1.11279 2	0.00387 7	1.115384
7	(cake)	(coffee)	0.103856	0.478394	0.05472 8	0.526958	1.10151 5	0.00504 4	1.102664
26	(scone) 0.478394	(coffee)	0.034548		0.01806 7 6	0.52293	1.09310 7	0.00153 9	1.093366
13	(cookies) 0.478394	(coffee)	0.054411		0.02820 9 7	0.51844	1.08372 3	0.00217 9	1.083174
1	(hot				5 chocolate)	(coffee) 0.02958	0.058320 3	0.508296	0.508296

				1.06031 1	0.00168 3	1.058553			
5	(brownie)	(coffee)	0.040042	0.478394	0.01965 1	0.490765	1.02586 0	0.00049 5	1.024293
2 0	(muffin) 0.478394	(coffee)	0.038457		0.01880 6 1	0.48901	1.02219 3	0.00040 8	1.020777
3	(pastry)	(bread)	0.086107	0.327205	0.02916 0	0.338650	1.03497 7	0.00098 5	1.017305

	antecedent s	consequent s	antecedent support	consequent support	support	confidence	lift	leverage	conviction
10	(cake) 0.142631	(tea)	0.103856		0.02377 2 1	0.22889	1.60478 1	0.00895 9	1.111865
39	(coffee, tea) 0.103856	(cake)	0.049868		0.01003 7 1	0.20127	1.93797 7	0.00485 8	1.121962
32	(sandwich) 0.142631	(tea)	0.071844		0.01436 9 0	0.20000	1.40222 2	0.00412 2	1.071712
9	(hot chocolate)	(cake) 0.103856	0.058320		0.01141 0 2	0.19565	1.88387 4	0.00535 4	1.114125
38	(coffee, cake) 0.142631	(tea)	0.054728		0.01003 7 8	0.18339	1.28582 2	0.00223 1	1.049923
11	(tea) 0.103856	(cake)	0.142631		0.02377 2 7	0.16666	1.60478 1	0.00895 9	1.075372
37	(pastry)	(coffee, bread)	0.086107 0.090016		0.01119 9 1	0.13006	1.44487 2	0.00344 8	1.046033
36	(coffee, bread) 0.086107	(pastry)	0.090016		0.01119 9 3	0.12441	1.44487 2	0.00344 8	1.043749
6	(coffee)	(cake)	0.478394	0.103856	0.05472 8	0.114399	1.10151 5	0.00504 4	1.011905
34	(coffee, bread) 0.103856	(cake)	0.090016		0.01003 7 2	0.11150	1.07362 1	0.00068 8	1.008606

		(cake) (hot chocolate)	0.103856	0.058320	0.011410	0.109868	1.883874	0.005354	1.057910
3					0.014369		1.402222	0.004122	1.032134
3	(tea)	(sandwich)	0.142631		0.100741				
			0.071844						
2					0.047544		1.154168	0.006351	1.014740
2	(coffee)	(pastry)	0.478394		0.099382				
			0.086107						
3		(coffee,			0.010037		1.073621	0.000688	1.007336
5	(cake)	bread)	0.103856		0.096643				
			0.090016						



	antecedent s	consequent s	antecedent support	consequent support	support	confidence	lift	leverage	conviction
40	(cake) 0.049868	(coffee, tea)	0.103856		0.01003 7 3	0.09664	1.93797 7	0.00485 8	1.051779
2	(bread)	(pastry)	0.327205	0.086107	0.02916 0	0.089119	1.03497 7	0.00098 5	1.003306
24	(coffee) 0.071844	(sandwich)	0.478394		0.03824 6 7	0.07994	1.11279 2	0.00387 7	1.008807
18	(coffee) 0.061807	(medialuna)	0.478394		0.03518 2 2	0.07354	1.18987 8	0.00561 4	1.012667
41 )	(tea	(coffee, cake)	0.142631 0.054728		0.01003 7 0	0.07037	1.28582 2	0.00223 1	1.016827
14	(coffee)	(hot chocolate)	0.478394 0.058320		0.02958 3 7	0.06183	1.06031 1	0.00168 3	1.003749
12	(coffee) 0.054411	(cookies)	0.478394		0.02820 9 6	0.05896	1.08372 3	0.00217 9	1.004841
30	(coffee) 0.033597	(toast)	0.478394		0.02366 6 0	0.04947	1.47243 1	0.00759 3	1.016699
16	(coffee) 0.038563	(juice)	0.478394		0.02060 2 5	0.04306	1.11675 0	0.00215 4	1.004705
0	(coffee)	(alfajores)	0.478394	0.036344	0.01965 1	0.041078	1.13023 5	0.00226 4	1.004936
4	(coffee)	(brownie)	0.478394	0.040042	0.01965 1	0.041078	1.02586 0	0.00049 5	1.001080

<b>2</b>				0.01880		1.02219	0.00040	
<b>1</b>	(coffee)	(muffin)	0.478394	6		3	8	1.000888
	0.038457				0.03931			
				1				
<b>2</b>				0.01806		1.09310	0.00153	
<b>7</b>	(coffee)	(scone)	0.478394	7		7	9	1.003343
	0.034548				0.03776			
				5				
<b>2</b>		(spanish		0.01088		1.25176	0.00218	
<b>8</b>	(coffee)	brunch)	0.478394	2		6	9	1.004682
			0.018172		0.02274			
				7				

**Result:** Hence, the apriori algorithm has been implemented successfully.

## Experiment 9: Implementation of NLP programs

**Aim:**

To a Program for sentiment analysis for the given statements

### Procedure:

```
import re
import string
import numpy
as np
import random
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

Exploratory Data Analysis

import nltk
from nltk.stem import WordNetLemmatizer
lemma = WordNetLemmatizer()
nltk.download('stopwords')
from nltk.corpus import stopwords

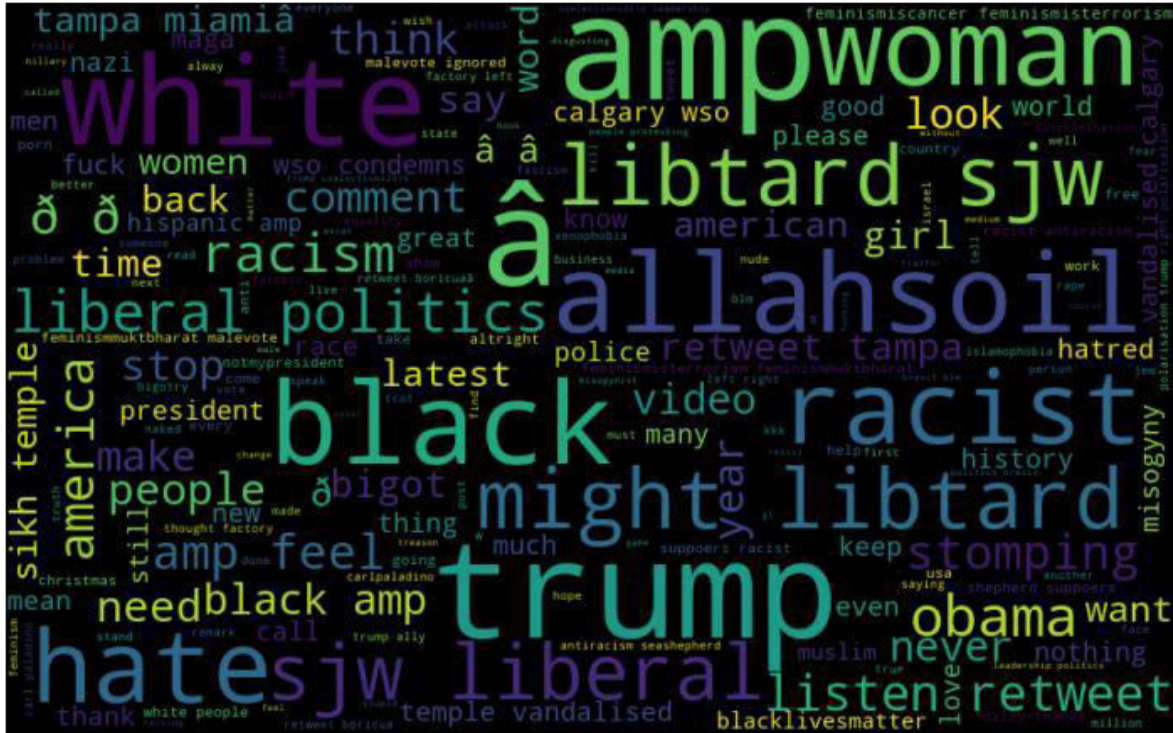
[nltk_data] Downloading package stopwords to C:\Users\Sai Ram [nltk_data]
[nltk_data]   Pendyala\AppData\Roaming\nltk_data... [nltk_data]   Package
stopwords is already up-to-date!
all_words = " ".join(sent for sent in train['clean_text'])
from wordcloud import WordCloud
wordcloud = WordCloud(width=800, height=500, random_state=42,
max_font_size=100).generate(all_words)
plt.figure(figsize=(15,8))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.show()
```



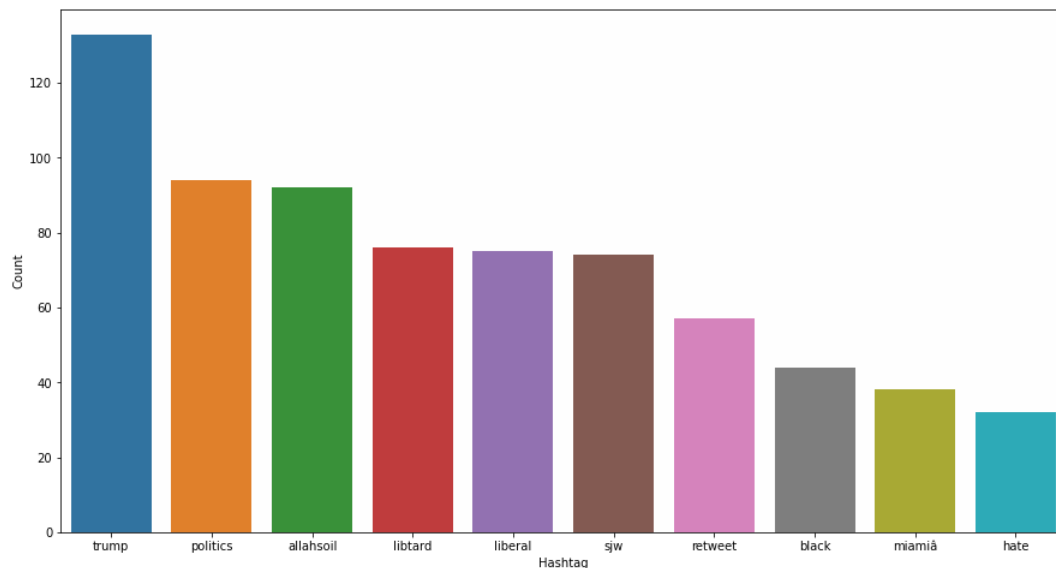
```
neg_words = "".join(sent for sent in train['clean_text'][train['label'] == 1])
```

```
wordcloud = WordCloud(width=800, height=500, random_state=42,  
max_font_size=100).generate(neg_words)
```

```
plt.figure(figsize=(15,8)) plt.imshow(wordcloud,  
interpolation='bilinear') plt.axis('off')  
plt.show()
```



```
neg = neg.nlargest(columns='Count', n=10)
plt.figure(figsize=(15,8)) sns.barplot(data=neg, x='Hashtag',
y='Count') plt.show()
```



```
from sklearn.feature_extraction.text import CountVectorizer
bow_vectorizer = CountVectorizer(max_df=0.90, min_df=2, max_features=1000, stop_words='english')
```

```

bow=bow_vectorizer.fit_transform(train['clean_text'])
from sklearn.model_selection import train_test_split
xtrain, xtest, ytrain, ytest = train_test_split(bow, train['label'], random_state=42, test_size=0.25)
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import f1_score, accuracy_score

# training
model = LogisticRegression()
model.fit(xtrain, ytrain)

LogisticRegression()
# testing
pred = model.predict(xtest) f1_score(ytest,
pred)

0.4835965978128798
accuracy_score(ytest, pred)

0.9468151670629458
# use probability to get output pred_prob =
model.predict_proba(xtest) pred = pred_prob[:, 1] >= 0.3
pred = pred.astype(np.int)

f1_score(ytest, pred) 0.532520325203252
accuracy_score(ytest, pred)

0.9424352396446002

```

### **Result:**

A program for sentiment analysis for the given statements is executed successfully.

## Experiment 10: Applying deep learning methods to solve an application

### **Aim:**

Applying deep learning methods to solve an application.

### **Procedure:**

```
import numpy as np
import pandas as pd
from pathlib import Path
import os.path
import matplotlib.pyplot as plt
import tensorflow as tf

# Create a list with the filepaths for training and testing
train_dir = Path('../input/fruit-and-vegetable-image-recognition/train')
train_filepaths = list(train_dir.glob(r'**/*.jpg'))

test_dir = Path('../input/fruit-and-vegetable-image-recognition/test')
test_filepaths = list(test_dir.glob(r'**/*.jpg'))

val_dir = Path('../input/fruit-and-vegetable-image-recognition/validation')
val_filepaths = list(test_dir.glob(r'**/*.jpg'))

def proc_img(filepath):
    """ Create a DataFrame with the filepath and the labels of the pictures
    """

    labels = [str(filepath[i]).split("/")[-2] \
               for i in range(len(filepath))]

    filepath = pd.Series(filepath, name='Filepath').astype(str)
    labels = pd.Series(labels, name='Label')

    # Concatenate filepaths and labels
    df = pd.concat([filepath, labels], axis=1)

    # Shuffle the DataFrame and reset index
    df = df.sample(frac=1).reset_index(drop = True)

    return df

train_df = proc_img(train_filepaths)
test_df = proc_img(test_filepaths)
val_df = proc_img(val_filepaths)

print('-- Training set --\n')
print(f'Number of pictures: {train_df.shape[0]}\n')
print(f'Number of different labels: {len(train_df.Label.unique())}\n')
print(f'Labels: {train_df.Label.unique()}')
```

In [ ]:

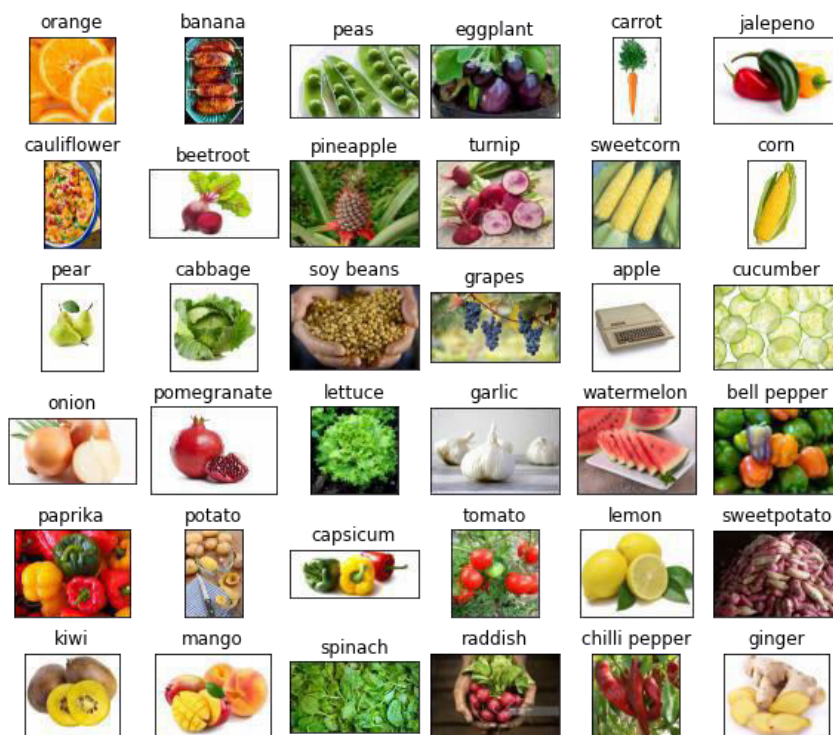
-- Training set --

Number of pictures: 3193

Number of different labels: 36

Labels: ['orange' 'banana' 'peas' 'eggplant' 'carrot' 'jalepeno' 'cauliflower'  
'beetroot' 'pineapple' 'turnip' 'sweetcorn' 'corn' 'pear' 'cabbage'  
'soy beans' 'grapes' 'apple' 'cucumber' 'onion' 'pomegranate' 'lettuce'  
'garlic' 'watermelon' 'bell pepper' 'paprika' 'potato' 'capsicum'  
'tomato' 'lemon' 'sweetpotato' 'kiwi' 'mango' 'spinach' 'raddish'  
'chilli pepper' 'ginger']

```
for i, ax in enumerate(axes.flat):  
    ax.imshow(plt.imread(df_unique.Filepath[i]))  
    ax.set_title(df_unique.Label[i], fontsize = 12)  
plt.tight_layout(pad=0.5)  
plt.show()
```



## 2. Load the Images with a generator and Data Augmentation

```
train_generator = tf.keras.preprocessing.image.ImageDataGenerator(  
    preprocessing_function=tf.keras.applications.mobilenet_v2.preprocess_input  
)  
test_generator = tf.keras.preprocessing.image.ImageDataGenerator(  
    preprocessing_function=tf.keras.applications.mobilenet_v2.preprocess_input  
)  
pretrained_model = tf.keras.applications.MobileNetV2(  
    input_shape=(224, 224, 3),  
    include_top=False,  
    weights='imagenet',
```

```

    pooling='avg'
)
pretrained_model.trainable = False

```

### 3. Train the model

```

inputs = pretrained_model.input

x = tf.keras.layers.Dense(128, activation='relu')(pretrained_model.output)
x = tf.keras.layers.Dense(128, activation='relu')(x)

outputs = tf.keras.layers.Dense(36, activation='softmax')(x)

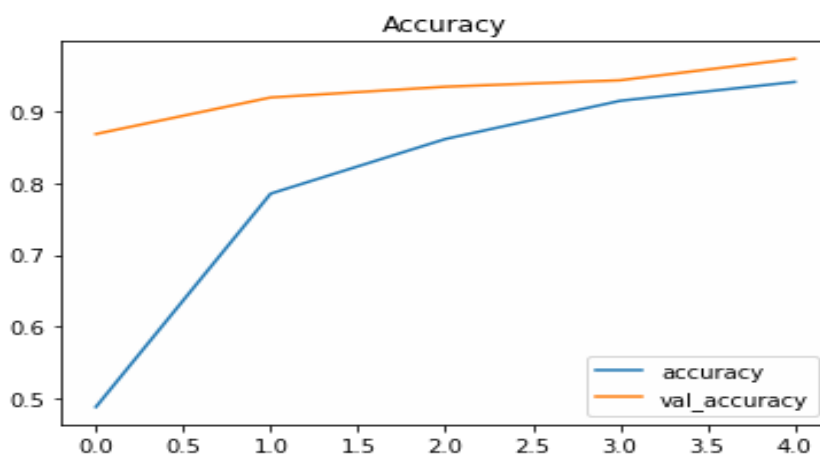
model = tf.keras.Model(inputs=inputs, outputs=outputs)

model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

history = model.fit(
    train_images,
    validation_data=val_images,
    batch_size = 32,
    epochs=5,
    callbacks=[
        tf.keras.callbacks.EarlyStopping(
            monitor='val_loss',
            patience=2,
            restore_best_weights=True
        )
    ]
)

pd.DataFrame(history.history)[['accuracy', 'val_accuracy']].plot()
plt.title("Accuracy")
plt.show()

```





#### 4. Visualize the result

```
# Predict the label of the test_images
pred = model.predict(test_images)
pred = np.argmax(pred,axis=1)
labels = (train_images.class_indices)
labels = dict((v,k) for k,v in labels.items())
pred = [labels[k] for k in pred]
```

```
y_test = [labels[k] for k in test_images.classes]
```

/opt/conda/lib/python3.7/site-packages/PIL/TiffImagePlugin.py:785: UserWarning: Corrupt EXIF dat

a. Expecting to read 4 bytes but only got 0.

```
warnings.warn(str(msg))
```

```
from sklearn.metrics import accuracy_score
```

```
acc = accuracy_score(y_test, pred)
```

```
print(f'Accuracy on the test set: {100*acc:.2f}%')
```

Accuracy on the test set: 97.31%

```
from sklearn.metrics import confusion_matrix
```

```
import seaborn as sns
```

```
cf_matrix = confusion_matrix(y_test, pred, normalize='true')
```

```
plt.figure(figsize = (15,10))
```

```
sns.heatmap(cf_matrix,
```

```
annot=True,
```

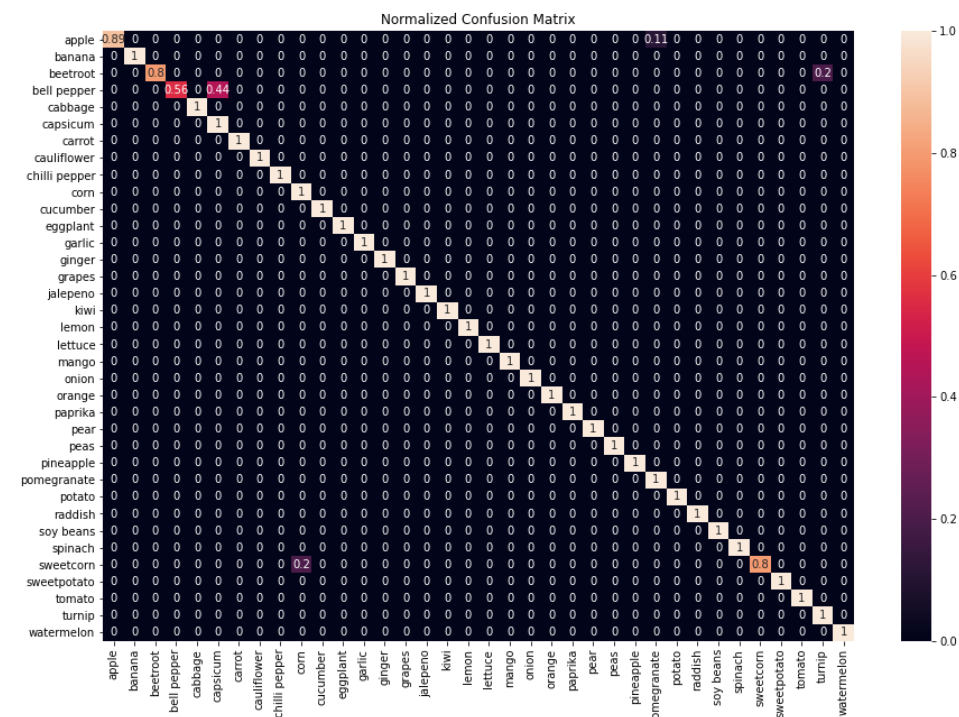
```
xticklabels = sorted(set(y_test)),
```

```
yticklabels = sorted(set(y_test)),
```

```
)
```

```
plt.title('Normalized Confusion Matrix')
```

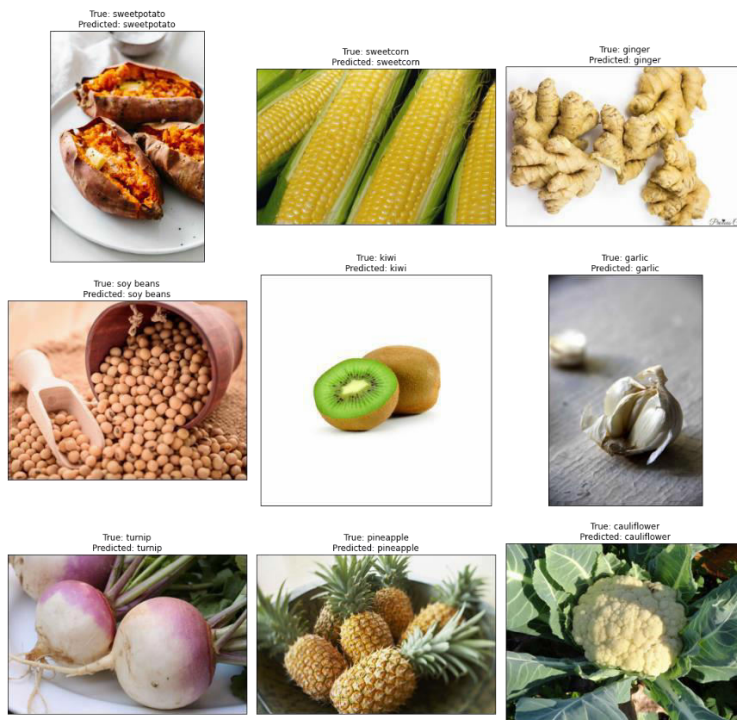
```
plt.show()
```



```
for i, ax in enumerate(axes.flat):
```

```
ax.imshow(plt.imread(test_df.Filepath.iloc[i]))
```

```
ax.set_title(f"True: {test_df.Label.iloc[i]}\nPredicted: {pred[i]}")
plt.tight_layout()
plt.show()
```



## 5. Class activation heatmap for image classification

```
import matplotlib.cm as cm
```

```
def get_img_array(img_path, size):
    img = tf.keras.preprocessing.image.load_img(img_path, target_size=size)
    array = tf.keras.preprocessing.image.img_to_array(img)
    # We add a dimension to transform our array into a "batch"
    # of size "size"
    array = np.expand_dims(array, axis=0)
    return array

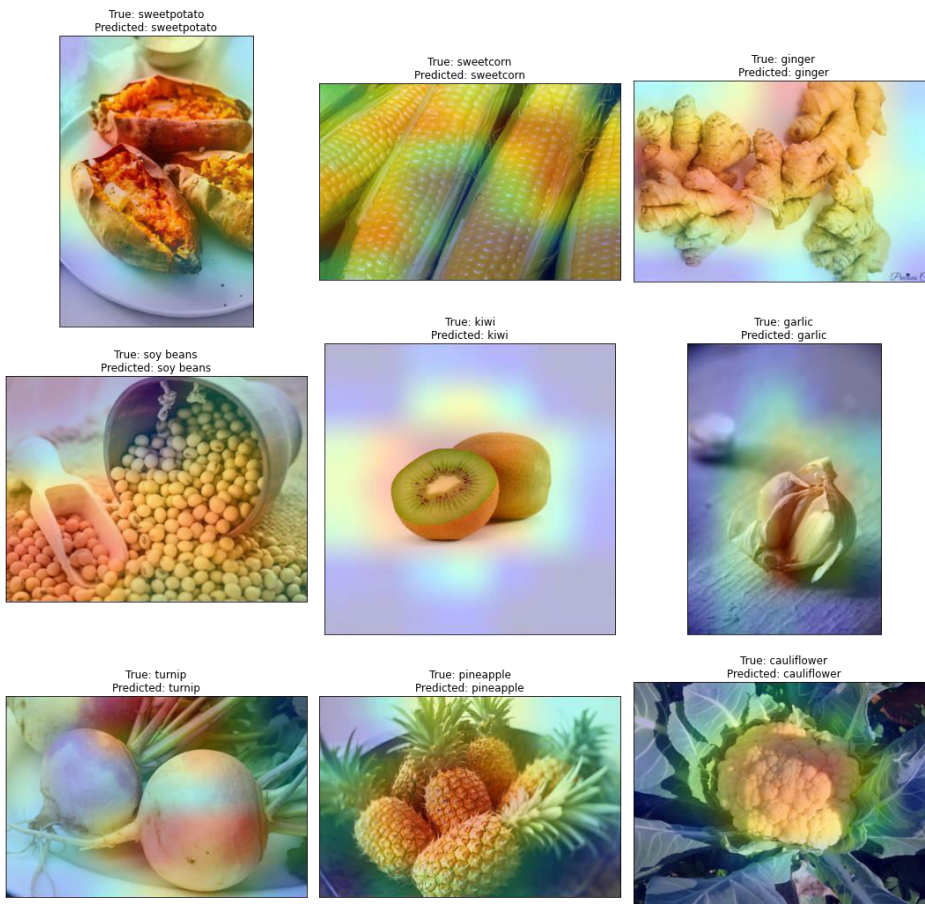
def make_gradcam_heatmap(img_array, model, last_conv_layer_name, pred_index=None):
    # First, we create a model that maps the input image to the activations
    # of the last conv layer as well as the output predictions
    grad_model = tf.keras.models.Model(
        [model.inputs], [model.get_layer(last_conv_layer_name).output, model.output]
    )

    # Then, we compute the gradient of the top predicted class for our input image
    # with respect to the activations of the last conv layer
    with tf.GradientTape() as tape:
        last_conv_layer_output, preds = grad_model(img_array)
        if pred_index is None:
            pred_index = tf.argmax(preds[0])
        class_channel = preds[:, pred_index]
```

```

for i, ax in enumerate(axes.flat):
    img_path = test_df.Filepath.iloc[i]
    img_array = preprocess_input(get_img_array(img_path, size=img_size))
    heatmap = make_gradcam_heatmap(img_array, model, last_conv_layer_name)
    cam_path = save_and_display_gradcam(img_path, heatmap)
    ax.imshow(plt.imread(cam_path))
    ax.set_title(f"True: {test_df.Label.iloc[i]}\nPredicted: {pred[i]}")
plt.tight_layout()
plt.show()

```



## Result:

Hence, deep learning methods have been applied to an application.