In [1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount
("/content/drive", force_remount=True).
```

In [2]:
```
#Importing labraires
import csv
import pandas as pd
from itertools import combinations
```

In [3]:
```
#Read data from CSV
data = pd.read_csv('/content/drive/MyDrive/Datasets/marketbasket.csv')
```

In [4]:
```
data.head()
```

Out[4]:

	Hair Conditioner	Lemons	Standard coffee	Frozen Chicken Wings	98pct. Fat Free Hamburger	Sugar Cookies	Onions	Deli Ham	Dishwasher Detergent	Beets	40 Watt Lightbulb	Cre
0	0	0	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	0	0	0	0	0	
3	0	0	0	0	0	0	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	0	0	0	

5 rows × 255 columns

In [5]:
```
#Checking unique values does a column have
data.nunique()
```

Out[5]:
```
Hair Conditioner             2
Lemons                       2
Standard coffee              2
Frozen Chicken Wings         2
98pct. Fat Free Hamburger    2
                            ..
Souring Pads                 2
Tuna Spread                  2
Toilet Paper                 2
White Wine                   2
Columbian Coffee             2
Length: 255, dtype: int64
```

In [6]:
```
#Checking dataset info
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1361 entries, 0 to 1360
Columns: 255 entries,  Hair Conditioner to  Columbian Coffee
dtypes: int64(255)
memory usage: 2.6 MB
```

In [7]:

```
data.describe()
```

Out[7]:

	Hair Conditioner	Lemons	Standard coffee	Frozen Chicken Wings	98pct. Fat Free Hamburger	Sugar Cookies	Onions	Deli Ham	Disl De
count	1361.00000	1361.000000	1361.000000	1361.000000	1361.000000	1361.000000	1361.000000	1361.000000	136
mean	0.05878	0.030125	0.008817	0.026451	0.093314	0.055107	0.080088	0.010287	
std	0.23530	0.170994	0.093519	0.160532	0.290979	0.228272	0.271529	0.100937	
min	0.00000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	0.00000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
50%	0.00000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
75%	0.00000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
max	1.00000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	

8 rows × 255 columns

In [8]:
```
#Parameters
minsup=float(input("Enter Support-Threshold: "))
minsup=minsup*len(data)
minconf=float(input("Enter Confidence-Threshold: "))
```

```
Enter Support-Threshold: 0.25
Enter Confidence-Threshold: 0.1
```

In [9]:
```
#Add all data in a list of lists
items = []
for i in range(0, len(data)):
    items.append([str(data.values[i,j]) for j in range(0, len(data.values[0]))])
```

In [10]:
```
#Creating a list of dictionaries
count = [dict() for x in range(len(data.values[0])+1)]
```

In [11]:
```
#Count support for each individual items
s=[]
for i in items:
    for j in i:
        s.append(j)
for i in s:
    #If item is present in dictionary, increment its count by 1
    if i in count[1]:
        count[1][i] = count[1][i] + 1
    #If item is not present in dictionary, set its count to 1
    else:
        count[1][i] = 1
```

In [12]:
```
#Checking null values in our dataset
data.isnull().sum()
```

Out[12]:
```
Hair Conditioner            0
Lemons                      0
Standard coffee             0
Frozen Chicken Wings        0
```

```
98pct. Fat Free Hamburger    0
                           ..
Souring Pads                 0
Tuna Spread                  0
Toilet Paper                 0
White Wine                   0
Columbian Coffee             0
Length: 255, dtype: int64
```

Implementing Apriori Algo

In [13]:
```
#Generate frequent two item sets
slist=[list() for x in range(33)]
a=[]
a=combinations(count[1],2)
for j in a:
    slist[2].append(tuple(sorted(j)))
```

In [14]:
```
# slist[2]=list(combinations(count[1],2))
candidates=[]
for i in slist[2]:
    candidates.append(i)
for i in candidates:
    for k in items:
        f=0
        for l in i:
            if(k.__contains__(l)==0):
                f=1
                break
        if(f==0):
            if i in count[2]:
                count[2][i]=count[2][i]+1
            else:
                count[2][i]=1
for i in count[2].copy():
    if(count[2][i]<minsup):
        count[2].pop(i)
```

In [15]:
```
#Generate frequent itemsets of length z from z-1
def freq(z):
    for i in count[z-1]:
        for j in count[z-1]:
            a=set(i)
            b=set(j)
            #Generate z length itemsets from z-1 length frequent itemsets which have z-2 i
            if(len(a.intersection(b))==z-2):
                #Check if all subsets are in frequent z-1 itemsets, otherwise cannot be fi
                t=list(combinations(sorted(a.union(b)), z-1))
                c=0
                for n in t:
                    for m in count[z-1]:
                        if((n)==m):
                            c=c+1
                if(c==z):
                    flag=0
                    for h in slist[z]:
                        if(sorted(list(a.union(b)))==sorted(h)):
                            flag=1
                    if(flag==0):
                        slist[z].append(tuple(sorted(list(a.union(b)))))
    #Calculate support
    candidates=[]
```

```
    for i in slist[z]:
        candidates.append(tuple(i))
    for i in candidates:
        for k in items:
            f=0
            for l in i:
                if(k.__contains__(l)==0):
                    f=1
                    break
            #If the complete item is present in the transaction, we increase it's support
            if(f==0):
                #If already present in dictionary then increment by 1
                if i in count[z]:
                    count[z][i]=count[z][i]+1
                #Else add it to dictionary
                else:
                    count[z][i]=1
    for i in count[z].copy():
        if(count[z][i]<minsup):
            count[z].pop(i)
```

In [16]:

```
#Call function to generate frequent itemssets
i=3
while(len(count[i-1])!=0):
    freq(i)
    i=i+1
q=i-2
```

In [17]:

```
#Function to extract single item set from a tuple
def value(a):
    a=str(a)
    a=a[:-2]
    a=a[2:]
    n=a[:-1]
    return n
```

In [18]:

```
#Find maximal and closed itemsets
maximal=[]
closed=[]
for i in range(1,q):
    for j in count[i]:
        fm=0
        fc=0
        for k in count[i+1]:
            a=set(list([j]))
            b=set(list(k))
            #Set is maximal if no immediate superset is frequent
            if(a.intersection(b)==a):
                fm=1
                #Set is closed if none of its immediate supersets have equal support
                if(count[i][j]==count[i+1][k]):
                    fc=1
        if(fm==0):
            maximal.append(j)
        if(fc==0):
            closed.append(j)
```

In [19]:

```
#All sets at the top of the tree are automatically maximal and closed
for i in count[q]:
```

```
    maximal.append(i)
    closed.append(i)
```

In [20]:
```
#Find Association Rules
print("ASSOCIATION RULES")
c=0
ant=count.copy()
for i in range(q,0,-1):
    for j in ant[i]:
        for k in range(i-1,0,-1):
            s=list(combinations(list(j),k))
            #Traverse through list of all combinations of antecedants
            for n in s:
                #Sorting to prevent duplicate itemsets
                r=tuple(sorted(set(j).difference(set(n))))
                l=len(n)
                #Check if len(n)==1 to be able to extract key to search in the support dic
                if(l==1):
                    n=value(n)
                    l=1
                if(len(r)==1):
                    r2=value(r)
                if(n!=None):
                    #If rule's confidence is greater than minconfidence, then print the ru
                    if((ant[len(j)][j]/ant[l][n])>=minconf):
                        #Rule is only significant if it is present in CLOSED, otherwise it
                        if(closed.__contains__((n))):
                            c=c+1
                            if(len(r)==1):
                                print(n,"(",ant[l][n],")","--->",r2,"(",ant[len(r)][r2],")
                            else:
                                print(n,"(",ant[l][n],")","--->",r,"(",ant[len(r)][r],")",
```

```
ASSOCIATION RULES
1 ( 11032 ) ---> 0 ( 336023 )  confidence =  0.11412255257432923
```

Result of Apriori

**Support=0.0045, Confidence=0.1 ASSOCIATION RULES 1 (11032) ---> 0 (336023) confidence = 0.11412255257432923

**Support=0.05, Confidence=0.25 {}

**Support=0.05, Confidence=0.05 ASSOCIATION RULES 1 (11032) ---> 0 (336023) confidence = 0.11412255257432923

Implementing FP Growth

In [21]:
```
#Read data from CSV again because kernal dead again and again
fp_data = pd.read_csv('/content/drive/MyDrive/Datasets/marketbasket.csv')
```

In [22]:
```
0#Parameters
minsup=float(input("Enter Support-Threshold: "))
minsup=minsup*len(fp_data)
minconf=float(input("Enter Confidence-Threshold: "))
```

```
Enter Support-Threshold: 0.025
Enter Confidence-Threshold: 0.1
```

In [23]:
```
#Add all fp_data in a list of lists
items = []
for i in range(0, len(fp_data)):
    items.append([str(fp_data.values[i,j]) for j in range(0, len(fp_data.values[0]))])
```

In [24]:
```
#Creating a list of dictionaries
count = [dict() for x in range(len(fp_data.values[0])+1)]
```

In [25]:
```
#Count support for each individual items
s=[]
for i in items:
    for j in i:
        s.append(j)
for i in s:
    #If item is present in dictionary, increment its count by 1
    if i in count[1]:
        count[1][i] = count[1][i] + 1
    #If item is not present in dictionary, set its count to 1
    else:
        count[1][i] = 1
```

In [26]:
```
#Storing transactions as lists without infrequent items
a=list(count[1])
item=[list() for i in range(len(fp_data))]
c=0
for i in range(0,len(items)):
    for j in range(len(items[i])):
        if(a.__contains__(items[i][j])!=0):
            item[i].append(items[i][j])
```

In [27]:
```
#Function to sort list to support
def sort(a):
    for i in range(len(a)-1):
        for j in range(len(a)-i-1):
            if(count[1][a[j]]<count[1][a[j+1]]):
                a[j],a[j+1]=a[j+1],a[j]
```

In [28]:
```
#Call function to sort all transactions in descending order of their support
for i in range(0,len(fp_data)):
    if(len(item[i])>1):
        sort(item[i])
```

In [29]:
```
#Tree class for FP-Tree
class tree:
    def __init__(self, name, sup, parent):
        self.name = name
        self.sup = sup
        self.nodeLink = None
        self.parent = parent
        self.children = []
```

In [30]:
```
#Function to check if the node is present is a child of the current node
def ispresent(node,name):
    f=-1
    for i in node.children:
        f=f+1
```

```
        if(i.name==name):
            return f
    return -1
```

In [31]:
```
#HeaderTable which stores the reference of last/first occurence of an item. Used as a link
lastocc=count[1].copy()
for i in lastocc:
    lastocc[i]=None
```

In [32]:
```
#Function to create FP-tree
root = tree("root",-1,None)
z=0
for i in item:
    current=root
    for j in range(len(i)):
        if(ispresent(current,i[j])>=0):
            current=current.children[ispresent(current,i[j])]
            current.sup=current.sup+1
        else:
            child=tree(i[j],1,current)
            current.children.append(child)
            t=current
            current=current.children[ispresent(current,i[j])]
            current.parent=t
            if(lastocc[current.name]==None):
                lastocc[current.name]=current
            else:
                current.nodeLink=lastocc[current.name]
                lastocc[current.name]=current
```

In [33]:
```
#Function to extract single item set from a tuple
def value(a):
    a=str(a)
    a=a[:-2]
    a=a[2:]
    a=a[:-1]
    return a
```

In [34]:
```
#Function to get frequent itemsets with suffix 'node' and length n
def singlepath(node,n):
    c=0
    sup=node.sup
    path=[]
    pathname=[]
    current=node

    #Get the path from current node to root
    while(current.parent!=None):
        path.append(current)
        pathname.append(current.name)
        current=current.parent
    path.remove(node)
    pathname.remove(node.name)
    candidatepath=[]
    temp_candidatepath=[]
    #Generate combinations of length n in the path
    a =(list(combinations(pathname,n)))
    for j in a:
        temp_candidatepath.append(tuple(sorted(j)))
    #Append the suffix 'node.name' to the above paths
```

```
    for j in temp_candidatepath:
        j=list(j)
        j.append(node.name)
        candidatepath.append(sorted(j))
    #Update counts of the generated itemsets
    for j in candidatepath:
        j=tuple(j)
        if j in count[n+1]:
            count[n+1][j]=count[n+1][j]+sup
        else:
            count[n+1][j]=sup

    #Iterating in the candidate tree recursively
    if(node.nodeLink!=None):
        node=node.nodeLink
        singlepath(node,i)
```

In [35]:
```
#Check if itemset is frequent
def frequent(n):
    f=0
    for i in count[n]:
        if(count[n][i]>=minsup):
            f=1
    if(f==1):
        return 1
    else:
        return 0
```

In [36]:
```
0
```

Out[36]:
```
0
```

In [37]:
```
#Remove infrequent itemsets
for z in range(len(fp_data.values[0])+1):
    for i in count[z].copy():
            if(count[z][i]<minsup):
                count[z].pop(i)
```

In [38]:
```
#Get 'q', the length of the longest itemset
i=2
while(len(count[i-1])!=0):
    i=i+1
q=i-2

#Find maximal and closed itemsets
maximal=[]
closed=[]
for i in range(1,q):
    for j in count[i]:
        fm=0
        fc=0
        for k in count[i+1]:
            a=set(list([j]))
            b=set(list(k))
            #Set is maximal if no immediate superset is frequent
            if(a.intersection(b)==a):
                fm=1
                #Set is closed if none of its immediate supersets have equal support
                if(count[i][j]==count[i+1][k]):
                    fc=1
```

```
        if(fm==0):
            maximal.append(j)
        if(fc==0):
            closed.append(j)
```

In [39]:

```
#All sets at the top of the tree are automatically maximal and closed
for i in count[q]:
    maximal.append(i)
    closed.append(i)
```

In [40]:

```
#Find Association Rules
print("ASSOCIATION RULES")
ant=count.copy()
for i in range(q,0,-1):
    for j in ant[i]:
        for k in range(i-1,0,-1):
            s=list(combinations(list(j),k))
            #Traverse through list of all combinations of antecedants
            for n in s:
                #Sorting to prevent duplicate itemsets
                r=tuple(sorted(set(j).difference(set(n))))
                l=len(n)
                #Check if len(n)==1 to be able to extract key to search in the support dic
                if(l==1):
                    n=value(n)
                    l=1
                if(len(r)==1):
                    r2=value(r)
                if(n!=None):
                    #If rule's confidence is greater than minconfidence, then print the ru
                    if((ant[len(j)][j]/ant[l][n])>=minconf):
                        #Rule is only significant if it is present in CLOSED, otherwise it
                        if(closed.__contains__((n))):
                            c=c+1
                            if(len(r)==1):
                                print(n,"(",ant[l][n],")","--->",r2,"(",ant[len(r)][r2],")
                            else:
                                print(n,"(",ant[l][n],")","--->",r,"(",ant[len(r)][r],")",
```

```
ASSOCIATION RULES
```