Apriori Algorithm Code Implementation

Hey everyone! Welcome to the doc on the implementation of the Apriori algorithm. Just a reminder, it is highly important that you brush up your concepts on how the algorithm works before you start implementing the algorithm in Python. Make sure you are familiar with the following keywords, as we will be using these terms frequently.

1. Support
2. Confidence
3. Frequent Itemsets
4. Rule Generation technique
5. Candidate Generation
6. Candidate Pruning

Note: It is highly recommended that you first try to implement the algorithm on your own from scratch. We understand how tempting it must be to scroll down and jump to the solution (which you might have already done by now). Keep in mind that implementing the algorithm from scratch will not only improve your ability to write complex algorithms but also help you realise the areas in Apriori that you might be weak in.

You can go through the concepts again from the *Introduction to Machine Learning: Lecture Notes* where we have explained the Apriori algorithm in detail.

You can download the dataset from the link below
(https://www.kaggle.com/roshansharma/market-basket-optimization).

The following libraries are used for the implementation of the algorithm:-
a) Numpy, b) Pandas.

Note: Just a head's up. Many of us will find it difficult to follow the code. As a result, we have tried to simplify the explanation as much as we could. Make sure you follow this doc line by line, so you are not confused at any point.

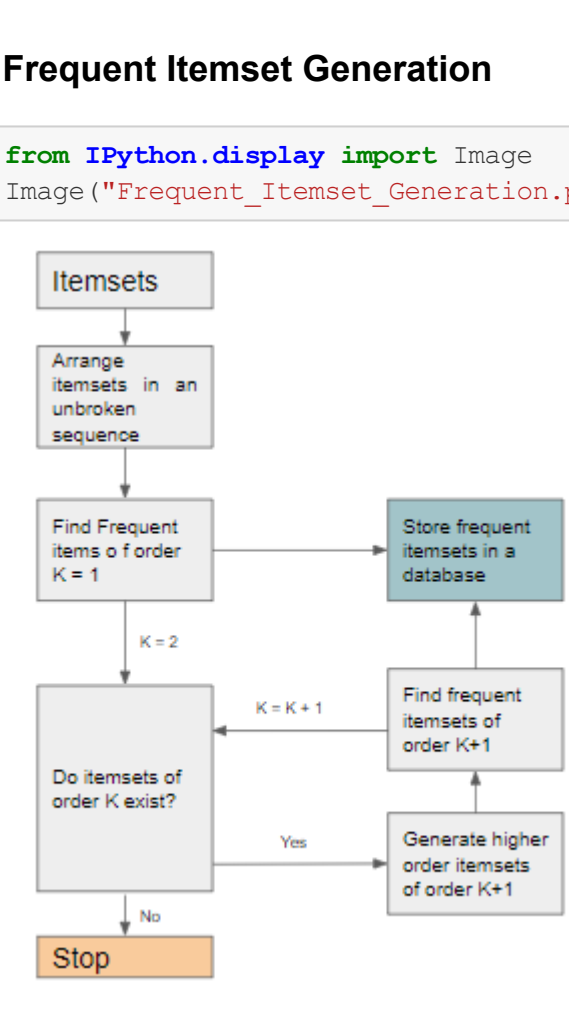
We have divided the code implementation into the following two parts:
Part A: Frequent Itemset Generation
Part B: Rule Generation

You may refer to the flow diagrams given below

Frequent Itemset Generation

In [1]:
```
from IPython.display import Image
Image("Frequent_Itemset_Generation.png")
```

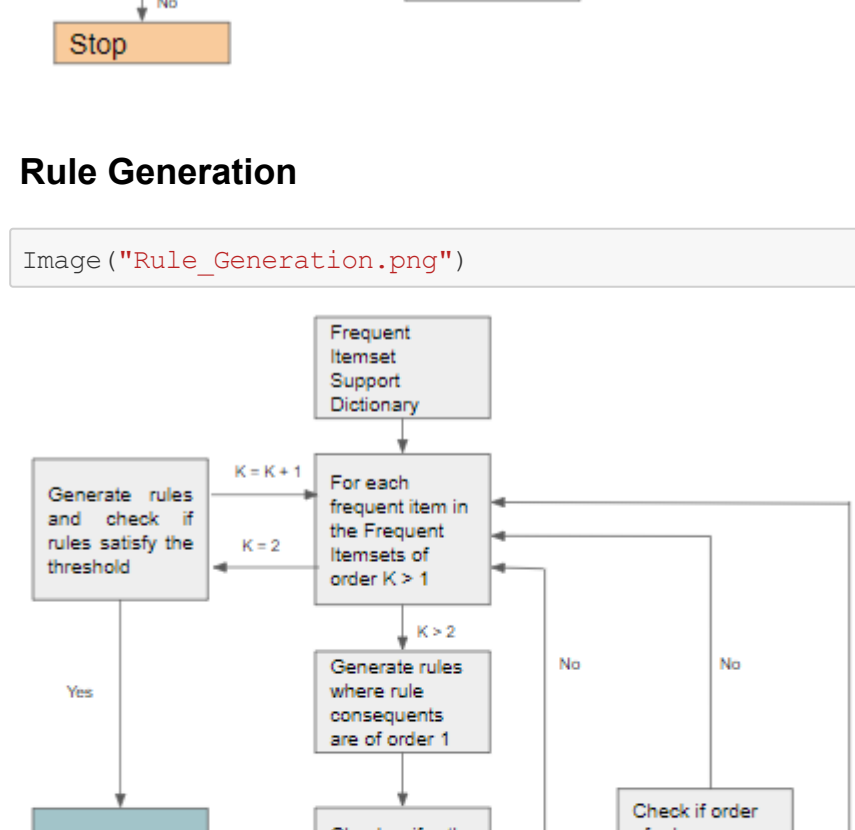
Out[1]:



Rule Generation

In [2]:
```
Image("Rule_Generation.png")
```

Out[2]:



Let's begin!

PART A: Frequent Itemset Generation

In [3]:
```
# Importing the libraries and the dataset

import numpy as np
import pandas as pd

Market_Data = pd.read_csv('Market_Basket_Optimisation.csv',index_col=None, header = None ) # Use you
r local path here

Market_Data.head(10)
```

Out[3]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
0	shrimp	almonds	avocado	vegetables mix	green grapes	whole weat flour	yams	cottage cheese	energy drink	tomato juice	low fat yogurt	green tea	honey	salad	mine wa
1	burgers	meatballs	eggs	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
2	chutney	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
3	turkey	avocado	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
4	mineral water	milk	energy bar	whole wheat rice	green tea	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
5	low fat yogurt	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
6	whole wheat pasta	french fries	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
7	soup	light cream	shallot	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
8	frozen vegetables	spaghetti	green tea	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
9	french fries	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N

In [4]:
```
# Converting the Market Dataset into a nested list
Dataset = []

for index, transaction in Market_Data.iterrows():
    cleaned_transaction = transaction[~transaction.isnull()].tolist()
    Dataset.append(cleaned_transaction)
```

In [5]:
```
# First 5 transactions of the dataset
Dataset[:5]
```

Out[5]:
```
[['shrimp',
  'almonds',
  'avocado',
  'vegetables mix',
  'green grapes',
  'whole weat flour',
  'yams',
  'cottage cheese',
  'energy drink',
  'tomato juice',
  'low fat yogurt',
  'green tea',
  'honey',
  'salad',
  'mineral water',
  'salmon',
  'antioxydant juice',
  'frozen smoothie',
  'spinach',
  'olive oil'],
 ['burgers', 'meatballs', 'eggs'],
 ['chutney'],
 ['turkey', 'avocado'],
 ['mineral water', 'milk', 'energy bar', 'whole wheat rice', 'green tea']]
```

1. The createItem function

Each transaction consists of various items, and the entire dataset consists of 7,501 transactions. With the help of this function, you can generate all the unique items from the entire dataset.

In [6]:
```
# For the given dataset writing a function to return the list of distinct items in the dataset

def createItem(dataSet):

    """
    This function extracts all the unique items from the entire dataset and sorts them (alphabetical
ly)

    Attributes
    ----------
    dataSet : list
        Market dataset

    Return Types
    ---------
    frozen Itemlist : list
        frozen list of unique items
    """

    ItemList = []

    for transaction in dataSet:
        for item in transaction:
            if not [item] in ItemList:

                # creating unique single lists in Itemlist. ie list of all items
                ItemList.append([item])

    ItemList.sort()

    return list(map(frozenset, ItemList))
```

In [7]:
```
# This function searches for all the items and sorts them (by default they are sorted alphabeticall
y):
temp_data = createItem(Dataset)

# Displaying 15 unique items from our Dataset
temp_data[:15]
```

Out[7]:
```
[frozenset({' asparagus'}),
 frozenset({'almonds'}),
 frozenset({'antioxydant juice'}),
 frozenset({'asparagus'}),
 frozenset({'avocado'}),
 frozenset({'babies food'}),
 frozenset({'bacon'}),
 frozenset({'barbecue sauce'}),
 frozenset({'black tea'}),
 frozenset({'blueberries'}),
 frozenset({'body spray'}),
 frozenset({'bramble'}),
 frozenset({'brownies'}),
 frozenset({'bug spray'}),
 frozenset({'burger sauce'})]
```

Why have we used frozenset?

Frozenset simply freezes an iterable object, such as the one above, so that the order is immutable. Frozensets cannot be modified further. This property will later be applied while joining different itemset lists. We want the order to be maintained.

2. The scanData function

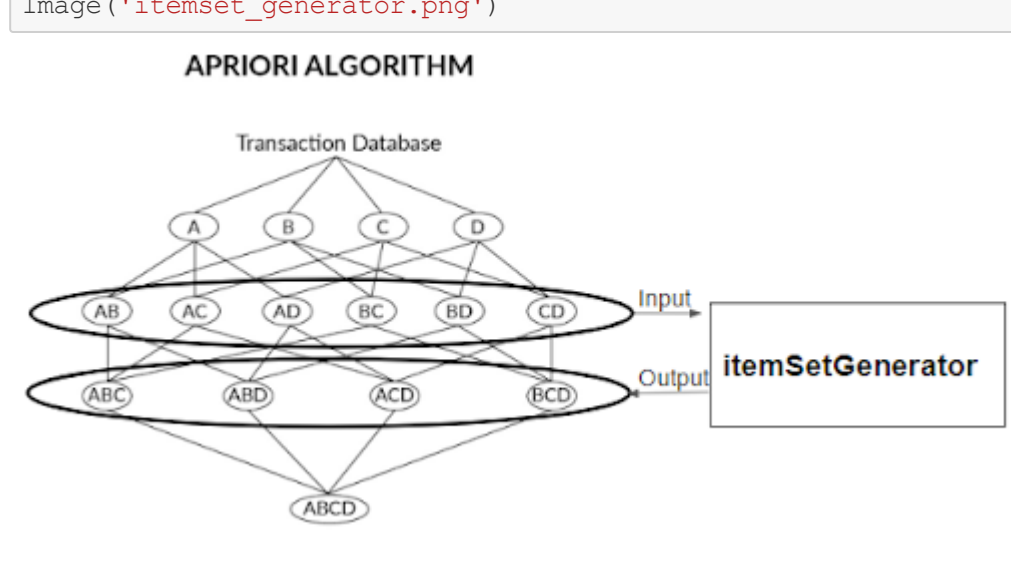
In [8]:
```
# Returns an Itemset and dictionary with Support values

def scanData(data, itemsetk, minSupport):

    """
    As you know, support is the frequency of a particular item or itemset.
    In order to calculate the support, we have created a supportDict dictionary.
    This stores the support value for each item set.
    If the support value is greater than minsupport, we store the itemset in freqItemset.
    Finally the function returns the 'freqItemset' and the 'supportDict'.

    Attributes
    ----------
    data : list
    List of frozensets

    itemsetk : Frozen set
    Note they can also be 2 item sets such as [a, b] or 3 item sets like []

    minSupport : float
    minimum support set by the user

    Return Types
    ---------
    freqItemset : Frozenset
    list of itemsets that satisfy the support threshold

    supportDict : dict
    Support values of each itemset

    """

    tempDict = {}

    for transaction in data:
        for item in itemsetk:
            if item.issubset(transaction):
                if not item in tempDict: tempDict[item]=1 # tempDict contains number of all items
                else: tempDict[item] += 1

    numItems = float(len(data))
    freqItemset = []
    supportDict = {}

    for key in tempDict:
        support = tempDict[key]/numItems

        if support >= minSupport:
            freqItemset.insert(0,key) # freqItemset contains all frequent items
        supportDict[key] = support # contains support of all items

    return freqItemset, supportDict
```

3. The itemSetGenerator function

From the pictorial representation given below, we can see that the function takes a list of k-order itemsets and returns all the possible combinations of the k+1 order.

In [9]:
```
Image('itemset_generator.png')
```

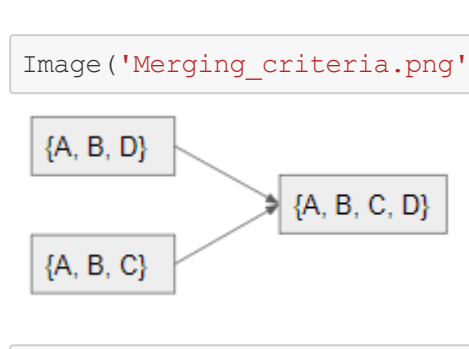
Out[9]:



The function uses candidate generation to merge those k order frequent itemsets whose first (k-1) elements are identical.

In [10]:
```
Image('Merging_criteria.png')
```

Out[10]:



In [11]:
```
# Creating Higher order Itemsets

def itemSetGenerator(itemsetk, k):

    """
    This function creates itemsets of order k+1 by merging the input itemsets of order k

    Attributes
    ----------
    itemsetk: list
        contains a frozen list of itemsets of (k-1)th order
    k: int
        order of itemset we want to generate

    Return Types
    ---------
    higherOrderitemset : list
        Merged itemsets

    """

    higherOrderitemset = []
    lenitemsetk = len(itemsetk)


    for i in range(lenitemsetk):
        for j in range(i+1, lenitemsetk):
            L1 = list(itemsetk[i])[:k-2]
            L2 = list(itemsetk[j])[:k-2]
            L1.sort()
            L2.sort()

            # Two frequent itemsets of order k are merged only if their k-1 itemsets are identical
            if L1 == L2:
                higherOrderitemset.append(itemsetk[i] | itemsetk[j]) # Performing set union creates
 itemset with n+1 items

    return higherOrderitemset
```

4. The Apriori function

All the functions that we have created until now will come into play in the Apriori function. All the possible k-order itemsets will be generated, and the support will be calculated for each itemset. Consequently, the freqItemsets of all orders will be stored and returned.

Description:

We use the 'createItem' function to extract all the unique items and store them as a frozenset. Next we scan these 1-itemsets, generate all the frequent itemsets and store them in 'freqItemsets'. Following this, we generate higher order itemsets until we are no longer able to find any itemsets of the kth order.

In [12]:
```
def apriori(dataSet, minSupport):

    """"
    The apriori function generates all the frequent itemsets and the support values for each possibl
e itemset.
    The reason for storing all the support values is that it will be used later for Rule Generation

    Attributes
    ----------
    dataSet: list
        The list of all transactions created before creating the functions.
    minSupport: float
        Since minSupport is completed upto the user, we have assumed the minSupport as 0.01 for the
 Market_Data dataset.

    Return Types
    ---------
    freqItemsets : list
         All possible frequent itemsets (for all values of order k)
    supportDict : dict
        support of all the itemsets

    """
    itemList = createItem(dataSet) # Creating frozenset of items


    # Generating all the frequent 1-itemsets and the support of those items
    freqItemset1, supportDict = scanData(dataSet, itemList, minSupport)
    freqItemsets = [freqItemset1]
    k = 2

    while (len(freqItemsets[k-2]) > 0): # Incrementing k until we no longer find any kth order items
ets

        itemsetk = itemSetGenerator(freqItemsets[k-2], k) # Generating itemsets of order k

        # Generating the frequent itemset for the kth order and support for each of these itemsets
        freqItemsetk, supportDictk = scanData(dataSet, itemsetk, minSupport)

        supportDict.update(supportDictk)
        freqItemsets.append(freqItemsetk)

        k += 1
    return freqItemsets, supportDict
```

In [13]:
```
freqItemsets, supportDict = apriori(Dataset, minSupport = 0.01)
```

In [14]:
```
# Frequent Itemsets of order 3
# Similarly you can extract the other itemsets as well
freqItemsets[2]
```

Out[14]:
```
[frozenset({'french fries', 'mineral water', 'spaghetti'}),
 frozenset({'eggs', 'milk', 'mineral water'}),
 frozenset({'eggs', 'ground beef', 'mineral water'}),
 frozenset({'chocolate', 'eggs', 'spaghetti'}),
 frozenset({'chocolate', 'milk', 'mineral water'}),
 frozenset({'chocolate', 'mineral water', 'spaghetti'}),
 frozenset({'chocolate', 'ground beef', 'mineral water'}),
 frozenset({'chocolate', 'milk', 'spaghetti'}),
 frozenset({'mineral water', 'olive oil', 'spaghetti'}),
 frozenset({'frozen vegetables', 'milk', 'mineral water'}),
 frozenset({'frozen vegetables', 'mineral water', 'spaghetti'}),
 frozenset({'chocolate', 'eggs', 'mineral water'}),
 frozenset({'milk', 'mineral water', 'spaghetti'}),
 frozenset({'ground beef', 'milk', 'mineral water'}),
 frozenset({'ground beef', 'mineral water', 'spaghetti'}),
 frozenset({'mineral water', 'pancakes', 'spaghetti'}),
 frozenset({'eggs', 'mineral water', 'spaghetti'})]
```

In [15]:
```
# Displaying support values of 10 random itemsets from the supportDict dictionary

import random

supportDict_list = list(supportDict.items())
num_keys = len(supportDict_list)

[supportDict_list[random.randint(0, num_keys-1)] for i in range(10)]
```

Out[15]:
```
[(frozenset({'frozen vegetables', 'ham'}), 0.0029329422743634183),
 (frozenset({'cider', 'fresh tuna'}), 0.0009332089054792694),
 (frozenset({'cookies', 'shrimp'}), 0.0041327822956939075),
 (frozenset({'herb & pepper', 'mushroom cream sauce'}), 0.0009332089054792694),
 (frozenset({'fromage blanc', 'turkey'}), 0.0014664711371817091),
 (frozenset({'soup', 'vegetables mix'}), 0.0011998400213304892),
 (frozenset({'fresh tuna', 'yogurt cake'}), 0.0009332089054792694),
 (frozenset({'cake', 'chicken'}), 0.006532462338354886),
 (frozenset({'light cream', 'shrimp'}), 0.0025329956005865884),
 (frozenset({'chocolate', 'cider'}), 0.0021330489268097585)]
```

Congratulations! You now have your frequent itemsets. Let's move on to Rule Generation. Hope you had no trouble following the implementation until now. If at any point you felt that your concepts were weak or you were not able to follow the code or text, you can revisit the lecture notes and revise the Apriori algorithm.

PART B: Rule Generation

5. The 'calcConf' function

As you may have guessed, this function takes in a frequent itemset and calculates the confidence for each rule that it generates.

As you know, a rule is made of an antecedent and a consequent. The general structure of a rule is given below:
(Rule antecedent) ----> (Rule consequent)

Recall the definition of confidence. For a given rule (a, b)--->(c, d), the confidence is calculated as follows::
conf((a, b)--->(c, d)) = support(a, b, c, d)/ support(a, b)

In other words, it is (support of frequent itemset)/ (support of (frequent itemset - consequent))

Notice how the same formula is used in the code?

In [16]:
```
def calcConf(freqSet, H, supportDict, bigRuleList, minConf):

    """"
    For each conseq in H, this function generates rules and calculates its confidence
    If the confidence is greater than minconf, we store the rule. We also store their rule consequen
ts in prunedH.

    Attributes
    ----------
    freqSet: list
        frequent itemset (We have already generated the frequent itemsets in PART A)
        eg. frozenset({'french fries', 'mineral water', 'spaghetti'})

    H: list
        Combinations of items stored in freqset
        eg. For the freqSet example shown above, H can be
            [frozenset({'french fries'}), frozenset({'spaghetti'}), frozenset({'mineral water'})]
            or
            [frozenset({'french fries', 'spaghetti'}),
            frozenset({'french fries', 'mineral water'}),
            frozenset({'spaghetti', 'mineral water'})]

    supportDict: dict
        Support Values of all the possible generated until now

    bigRuleList: list
        All our rules will be stored in this list and will be updated every time the function is exe
cuted

    minConf: float
        minimum confidence


    Return Types
    ---------
    prunedH : list
          Contains those consequents whose rules had a confidence higher than minconf

    """

    prunedH = []

    for conseq in H:

        conf = supportDict[freqSet]/supportDict[freqSet - conseq] # calculate confidence

        if conf >= minConf:
            bigRuleList.append((freqSet-conseq, conseq, conf))
            print(freqSet-conseq, '--->', conseq, 'confidence = ', conf)
            prunedH.append(conseq)

    return prunedH
```

Use the demonstration shown below to help you understand better

In [17]:
```
freqSet = frozenset({'french fries', 'mineral water', 'spaghetti'})
H = [frozenset({'french fries'}), frozenset({'spaghetti'}), frozenset({'mineral water'})]
minConf = 0.2

Hmp1 = calcConf(freqSet, H, supportDict, [], minConf)

# After generating the rules using the sets in H, we use H to generate higher order sets.
# After passing it through itemSetGenerator, we get,
print('\nHmp1')
print(Hmp1)
```

```
frozenset({'french fries', 'mineral water'}) ---> frozenset({'spaghetti'}) confidence =  0.30039
525691699603
frozenset({'french fries', 'spaghetti'}) ---> frozenset({'mineral water'}) confidence =  0.36714
975845410625

Hmp1
[frozenset({'spaghetti'}), frozenset({'mineral water'})]
```

If you notice, 'french fries' was removed from H. Since conseq = 'french fries', the confidence came to less than 0.2. Now, you must be wondering why is prunedH important? We will come back to this answer while explaining the 'rulesFromConseq' function.

6. The rulesFromConseq function

I'm sure that many of you would have had your doubts over here and opened the solution just to understand this particular function. Yes, we know function recursion has been used. Yes, this part is a little tricky. But don't worry about it; hopefully by the time you finish reading this, all your doubts will be cleared.

Description:
As you can see, H represents the consequents of the rules that we will be generating. If you notice carefully, the order of each consequent will always be less than the order of the frequent itemset. This condition has been established in the final part of the function. We begin with sets of first order consequent. The 'calcConf' function is used to generate the corresponding rules and store them in 'bigRuleList' if the confidence is greater than 'minconf'. The function 'calcConf' returns the same H with those consequents whose rules crossed the confidence threshold. Using these rule consequent sets, the 'itemSetGenerator' function is used to create higher order itemsets.

Consider the following example,
We start with,

In [18]:
```
freqSet = frozenset({'french fries', 'mineral water', 'spaghetti'})
H = [frozenset({'french fries'}), frozenset({'spaghetti'}), frozenset({'mineral water'})]
```

After generating the rules using the sets in H, we use H to generate higher order sets. After passing it through itemSetGenerator, we get,

In [19]:
```
Hmp1 = itemSetGenerator(H, 2)
Hmp1
```

Out[19]:
```
[frozenset({'french fries', 'spaghetti'}),
 frozenset({'french fries', 'mineral water'}),
 frozenset({'mineral water', 'spaghetti'})]
```

Later on, we use the condition that the order of each consequent will always be less than the order of the frequent itemset. If True, we want to generate new rules where the itemsets in Hmp1 are used as rule consequents.

Q) Now why does calcConf return prunedH and not H?

Here, we have used confidence-based pruning.
Let's consider generating rules for the itemset {a, b, c, d}

Suppose the confidence for {b, c, d} ---> {a} is less than the threshold. We can eliminate all the rules in which the rule consequent consists of higher-order itemsets containing {a}

Thus, we can eliminate {b, d} → {a, c}, {c, d} → {a, b}, {b, c} → {a, d}, {d} → {a, b, c}

Now suppose {b, c, d} ---> {a} does not satisfy the threshold. In this case, we eliminate {a} from the rule consequent, so that our algorithm does not have to create higher order itemsets of {a}, and check their thresholds as well.

In [20]:
```
def rulesFromConseq(freqSet, H, supportDict, bigRuleList, minConf):

    """"
    This function generates rules for itemsets of order > 2

    Attributes
    ----------
    freqSet: list
        frequent itemset

    H: list
        Combinations of items stored in freqset

    supportDict: dict
        Support Values of all the possible generated until now

    bigRuleList: list
        All our rules will be stored in this list and will be updated every time the function is exe
cuted

    minConf: float
        minimum confidence

    """

    m = len(H[0]) # Order of the consequent while generating the rules

    H = calcConf(freqSet, H, supportDict, bigRuleList, minConf)
    if len(H)>1:
        # creating higher order candidates
        Hmp1 = itemSetGenerator(H, m+1)

        if Hmp1 == []:
            # Hmp1 will be an empty list if the itemsets in H don't satisfy the condition for mergin
g
            return 0

        if (len(Hmp1[0]) < len(freqSet)):
            # Generate rules while the order of the itemsets in Hmp1 is less than the number of item
s in the frequent itemset
            rulesFromConseq(freqSet, Hmp1, supportDict, bigRuleList, minConf)
```

7. The generateRules function

Consider this the final function for Rule Generation. We have our functions and all we need to do is extract the frequent itemsets and generate the rules with the help of our functions.

Description:
We will begin by initialising the bigRuleList to an empty list. Now, we will form a nested loop runs through all the frequent itemsets in the 'freqItemsets' created in PART A. As explained earlier, we start with H1 consisting of all the order one combinations of the items in 'freqSet'. If the order of 'freqSet' is two, i.e., i = 1, then there is no need to create higher order itemsets from its subsets. Hence, we directly generate the rules by calling 'calcConf'. Once i>1, we need to break the freqSet into its subsets and generate different combinations of the rules. We have already shown the working in the explanation of the 'rulesFromConseq' function. Once you have all your rules in the 'bigRuleList', you can print them.

In [21]:
```
def generateRules(freqItemsets, supportDict, minConf):  #supportDict is a dictionary coming from sca
nData

    """"
    For each conseq in H, this function creates a rule and the confidence is calculated.
    If the confidence is greater than minconf, we store the rule. We also store the rule consequents
 in prunedH.

    Attributes
    ----------
    freqItemsets: list
        Frequent Itemsets generated from PART A
        eg. frozenset({'french fries', 'mineral water', 'spaghetti'})
    supportDict: dict
        Support Dictionary created in PART A
    minConf: float
        minimum confidence set by the user

    Return Types
    ---------
    bigRuleList : list
          Contains all the rules whose confidence is greater than minConf

    """
    bigRuleList = []
    for i in range(1, len(freqItemsets)): # Only get the sets with two or more items
        for freqSet in freqItemsets[i]:
            H1 = [frozenset([item]) for item in freqSet]
            if (i > 1):
                rulesFromConseq(freqSet, H1, supportDict, bigRuleList, minConf)
            else:
                calcConf(freqSet, H1, supportDict, bigRuleList, minConf)
    return bigRuleList
```

In [22]:
```
final_rules = generateRules(freqItemsets, supportDict, 0.3)
```

```
frozenset({'cooking oil'}) ---> frozenset({'spaghetti'}) confidence =  0.31070496083550914
frozenset({'red wine'}) ---> frozenset({'spaghetti'}) confidence =  0.36492890995260663
frozenset({'fresh bread'}) ---> frozenset({'mineral water'}) confidence =  0.30959752321981426
frozenset({'herb & pepper'}) ---> frozenset({'spaghetti'}) confidence =  0.32884097035040427
frozenset({'grated cheese'}) ---> frozenset({'mineral water'}) confidence =  0.3333333333333333
frozenset({'grated cheese'}) ---> frozenset({'spaghetti'}) confidence =  0.3155216284987277
frozenset({'herb & pepper'}) ---> frozenset({'mineral water'}) confidence =  0.34501347708894875
frozenset({'herb & pepper'}) ---> frozenset({'ground beef'}) confidence =  0.3234501347708895
frozenset({'soup'}) ---> frozenset({'mineral water'}) confidence =  0.45646437994722955
frozenset({'olive oil'}) ---> frozenset({'spaghetti'}) confidence =  0.3481781376518219
frozenset({'cereals'}) ---> frozenset({'mineral water'}) confidence =  0.3989637305699482
frozenset({'cake'}) ---> frozenset({'mineral water'}) confidence =  0.33881578947368424
frozenset({'red wine'}) ---> frozenset({'mineral water'}) confidence =  0.38862559241706157
frozenset({'frozen vegetables'}) ---> frozenset({'mineral water'}) confidence =  0.3748251748251
7486
frozenset({'chocolate'}) ---> frozenset({'mineral water'}) confidence =  0.3213995117982099
frozenset({'ground beef'}) ---> frozenset({'mineral water'}) confidence =  0.41655359565807326
frozenset({'ground beef'}) ---> frozenset({'spaghetti'}) confidence =  0.3989145183175034
frozenset({'soup'}) ---> frozenset({'milk'}) confidence =  0.3007915567282322
frozenset({'pancakes'}) ---> frozenset({'mineral water'}) confidence =  0.3548387096774194
frozenset({'spaghetti'}) ---> frozenset({'mineral water'}) confidence =  0.3430321592649311
frozenset({'salmon'}) ---> frozenset({'spaghetti'}) confidence =  0.3166144200626959
frozenset({'chicken'}) ---> frozenset({'mineral water'}) confidence =  0.38000000000000006
frozenset({'tomatoes'}) ---> frozenset({'mineral water'}) confidence =  0.3567251461988304
frozenset({'tomatoes'}) ---> frozenset({'spaghetti'}) confidence =  0.3060428849902534
frozenset({'turkey'}) ---> frozenset({'mineral water'}) confidence =  0.30703624733347548
frozenset({'turkey'}) ---> frozenset({'eggs'}) confidence =  0.31130063965884863
frozenset({'cooking oil'}) ---> frozenset({'mineral water'}) confidence =  0.39425587467362292
frozenset({'milk'}) ---> frozenset({'mineral water'}) confidence =  0.3703703703703704
frozenset({'whole wheat rice'}) ---> frozenset({'mineral water'}) confidence =  0.34396355353075
17
frozenset({'burgers'}) ---> frozenset({'eggs'}) confidence =  0.33027522935779813
frozenset({'avocado'}) ---> frozenset({'mineral water'}) confidence =  0.348
frozenset({'frozen smoothie'}) ---> frozenset({'mineral water'}) confidence =  0.32
frozenset({'honey'}) ---> frozenset({'mineral water'}) confidence =  0.31741573033707865
frozenset({'low fat yogurt'}) ---> frozenset({'mineral water'}) confidence =  0.313588850174216
frozenset({'olive oil'}) ---> frozenset({'mineral water'}) confidence =  0.4190283400809717
frozenset({'salmon'}) ---> frozenset({'mineral water'}) confidence =  0.4012539184952978
frozenset({'shrimp'}) ---> frozenset({'mineral water'}) confidence =  0.3302238805970149
frozenset({'french fries', 'spaghetti'}) ---> frozenset({'mineral water'}) confidence =  0.36714
975845410625
frozenset({'french fries', 'mineral water'}) ---> frozenset({'spaghetti'}) confidence =  0.30039
```