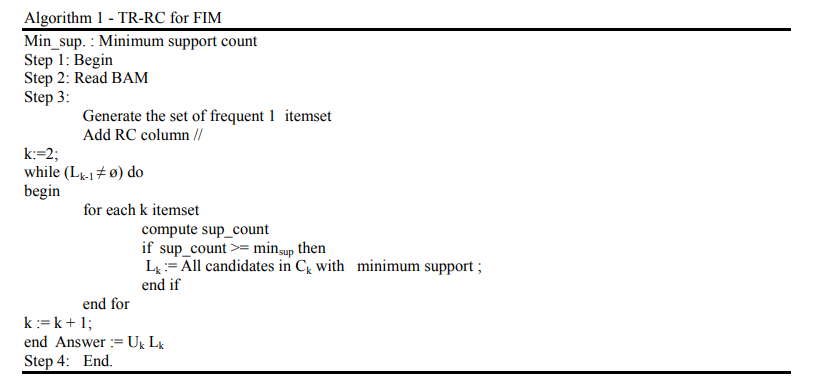
Association Rule Mining Project

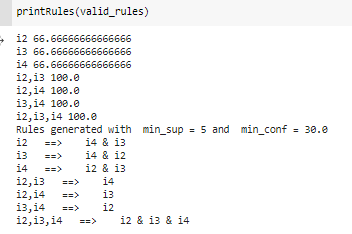
# Apriori Algorithm

Association rule mining finds interesting associations and relationships among large sets of data items. This rule shows how frequently a itemset occurs in a transaction. Given a set of transactions, the goal is to find rules that will predict the occurrence of an item based on the occurrences of other items in the transaction.

## Transaction Reduction

A transaction that does not contain any frequent k-itemsets cannot contain any frequent (k+1)-itemsets. Therefore, such a transaction can be marked or removed from further consideration. Implemented the below algorithm from scratch per the [paper](https://www.sciencedirect.com/science/article/pii/S1877050915004512). Refer transaction\_reduction.py for implementation.



Rules are generated for toy data as below with minimum support of 2 and confidence of 30%.

## Hash based Technique

When scanning each transaction in the database to generate the frequent 1-itemsets, L1, we can generate all the 2-itemsets for each transaction, hash (i.e., map) them into the different buckets of a hash table structure, and increase the corresponding bucket counts. A 2-itemset with a corresponding bucket count in the hash table that is below the support threshold cannot be frequent and thus should be removed from the candidate set.

We used itemset of size 2 for frequent set generation using the below hash function.

**H(x,y)= ((Order of first)\*10 + (Order of second)) mod 7**

Refer hashing.py for implementation details.

## Rule generation using WEKA

**Data Source:**

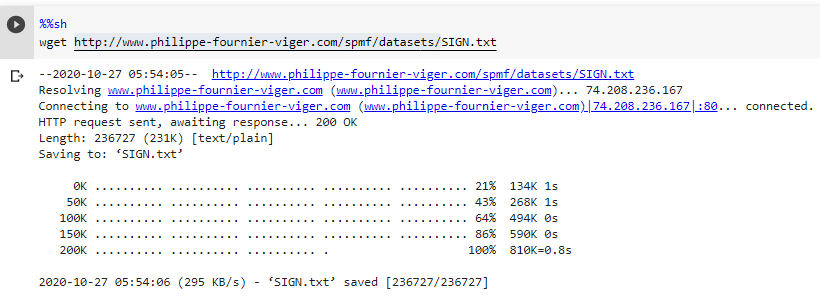
We used transactions data available at **SPFM** website<http://www.philippe-fournier-viger.com/spmf/index.php> **.** SPMF is an open-source softwareanddata mining librarywritten in Java, specialized in pattern mining(the discovery of patterns in data)

**Library Used:** WEKA library is used for rule generation using SPFM data.

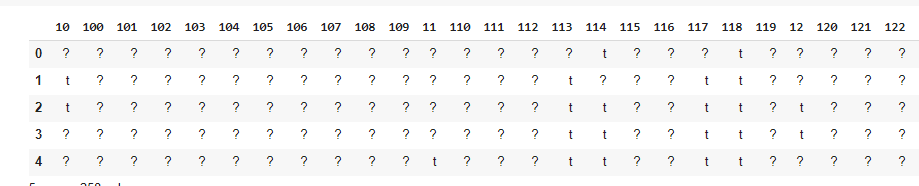
**Script**: SPFM2WEKA parser is written to convert data in SPFM format to WEKA format. Refer the script convertspfm2weka.py

Example#1: - SIGN data

A dataset of sign language utterance containing approximately 800 sequences and 267 items. The original dataset file in another format can be obtained [here](http://cs-people.bu.edu/panagpap/Research/asl_mining.htm) with more details on this dataset.



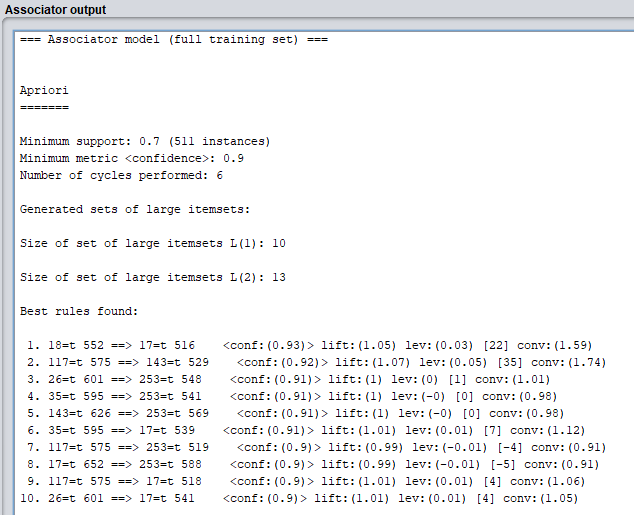
This raw data is converted to WEKA format, where Binary matrix of transactions and items is created.



Here each cell[i,j] represents if the transaction ‘i’ has item ‘j’ . Presence is represented as ‘t’ and absence as ‘?’.

**Rules**:

Below Rules generated in the WEKA tool with given support and confidence.



# FP-Growth Algorithm

Apriori Algorithm has slow performance and has below drawbacks

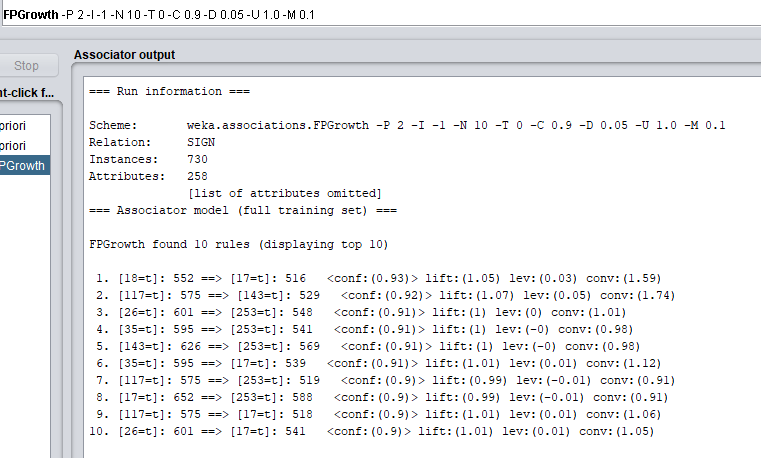
* At each step, candidate sets must be built.
* To build the candidate sets, the algorithm must repeatedly scan the database.

These two properties inevitably make the algorithm slower. To overcome these redundant steps, a new association-rule mining algorithm was developed named Frequent Pattern Growth Algorithm. It overcomes the disadvantages of the Apriori algorithm by storing all the transactions in a Tree Data Structure.

## Rule generation using WEKA

**Rules**:

Below Rules generated in the WEKA tool with given support and confidence.



## Rule generation using implementation from scratch

**FP growth Algorithm:**

FP algorithm is implemented from scratch in python. Generated Condition FP trees using Bottom UP and Top Down approach:

**Example 1:**

test\_data = [['I1','I2','I5'],  
 ['I2','I3','I4'],  
 ['I3','I4'],  
 ['I1','I2','I3','I4']]  
  
**Bottom up Approach:**Condtional FPTree Root on I1 : ['Null 1', ['I2 2']]  
Condtional FPTree Root on I4 : ['Null 1', ['I3 3', ['I2 2']]]  
Condtional FPTree Root on I3 : ['Null 1', ['I2 2']]

**Top Down Approach:**Condtional FPTree Root on I1: None  
Condtional FPTree Root on I4: ['I1:1']  
Condtional FPTree Root on I3: [['I1:1'], 'I4:2']  
Condtional FPTree Root on I2: [[['I1:1'], 'I4:2'], 'I3:2', 'I1:1']

**Example 2:**

test\_data = [['I1','I2','I5'],  
 ['I2','I4'],  
 ['I2','I3'],  
 ['I1','I2','I4'],  
 ['I1','I3'],  
 ['I2','I3'],  
 ['I1','I3'],  
 ['I1','I2','I3','I5'],  
 ['I1','I2','I3']]

**Bottom up Approach:**Condtional FPTree Root on I1:['Null 1', ['I2 2']]  
Condtional FPTree Root on I4:['Null 1', ['I3 3', ['I2 2']]]  
Condtional FPTree Root on I3:['Null 1', ['I2 2']]

**Top Down Approach:**Condtional FPTree Root on I1: None  
Condtional FPTree Root on I4: ['I1:1']  
Condtional FPTree Root on I3: [['I1:1'], 'I4:2']  
Condtional FPTree Root on I2: [[['I1:1'], 'I4:2'], 'I3:2', 'I1:1']

**Example 3:**

test\_data = [['A','B'],  
 ['B','C','D'],  
 ['A','C','D','E'],  
 ['A','D','E'],  
 ['A','B','C']]  
 **Bottom up Approach:**Condtional FPTree Root on E: ['Null 1', ['A 2', ['D 2']]]  
Condtional FPTree Root on D: ['Null 1', ['C 1'], ['A 2', ['C 1']]]  
Condtional FPTree Root on C: ['Null 1', ['B 1'], ['A 2', ['B 1']]]  
Condtional FPTree Root on B: ['Null 1', ['A 2']]  
  
**Top Down Approach:**Condtional FPTree Root on E : None  
Condtional FPTree Root on D : ['E:1']  
Condtional FPTree Root on C : [['E:1'], 'D:1']  
Condtional FPTree Root on B : ['C:1']  
Condtional FPTree Root on A : [['E:1'], 'D:1', [['E:1'], 'D:1'], 'C:1', ['C:1'], 'B:2']

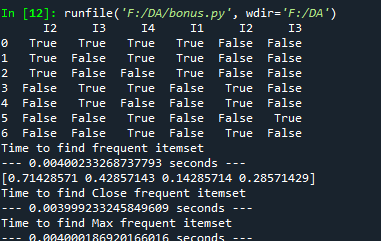
**Optimization and Comparison of performance:**Below is the approach followed in implementation of Fp-growth algorithm Traditional one (Bottom up approach):

* 1: Find the ordered frequent items. For items with the same frequency, order is given  
  based on the alphabetical order.
* 2: Develop the FP-tree from the above data
* 3: From FP-tree, deduce the FP-conditional tree for each item
* 4: Identify the frequent patterns

In top bottom approach: Traversed the item from top to bottom (until leaf node). For finding the Conditional FP patterns, we need to traverse the tree for all its children from top to bottom for each branch. Also, while developing FP tree, each node is linked to its another instance if they are placed in another branch. So, for traversing the children, multiple branches are traversed and selected only required items, whereas from Bottom to Top approach, where we do traverse from leaf to root, only one time traversal is fine.

## Bonus

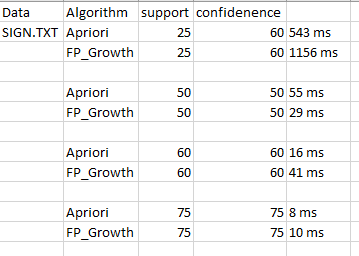
This part is implemented using library. For detailed steps refer bonus\_using\_library.py



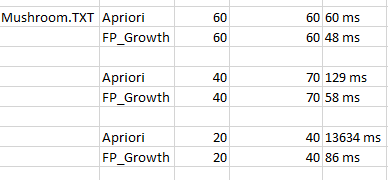
# Comparative case study

We used SIGN.txt from SPFM library to compare the timing using Apriori and FP\_Growth algorithms.

Below table indicates the result at various support and confidence levels. Results of the experiments are attached in SIGN\_Timings.txt



Results of the experiments are attached in Mushroom\_Timings.txt . In this case FP\_Growth is faster than Apriori.



Results of the experiments are attached in chess\_Timings.txt . In this case FP\_Growth is faster than Apriori by 1000x times.

