Ecole Polytechnique de Louvain
2020-2021
LINGI2364 - Mining Patterns in Data

Project report : frequent itemsets mining

Authors: Group 39
LEBLANC Stéphane 81972000
MOUADEN Badr-Ali 98022000

March 12, 2021

1 Introduction

For this project, we were asked to implement at least two frequent itemset miner algorithms. We decided to implement the Apriori algorithm based on boolean matrix and the depth-first search Eclat algorithm.

2 Apriori

This version of the Apriori algorithm is based on boolean matrix. First of all, we have to translate the database into a such matrix. In this one, each row represents an item and each column represents a transaction of the database. Then each element of the matrix is filled with 1 if an item is present in a transaction. For example, in the picture below, the item I1 is present in the three first transactions but not in the last one. This allows to scan the database only once and so to reduce the I/O operations.

Item	T1	T2	T3	T4
I0	0	0	1	1
I1	1	1	1	0
I2	1	0	0	0

Table 1: Example of boolean matrix

Then, we can remove the rows for which the total number of 1 is less than the minimum support. This means that this only one item is present in a number of transactions that is less than the minimum support. So this item is infrequent and any larger set with this item will also be infrequent thanks to the anti-monotonicity property. For example, for a minimum support of 2, the I2 row will be deleted. By this operation, we can reduce the size of the matrix.

Once our matrix is ready, we can start to compute the frequent itemsets. For each one, we reduce the view of the matrix to its items. For example, for a matrix of 10 items and a itemset {I0, I7, I9}, the matrix below is used to compute the frequency of this itemset. We can simply apply a logical AND between the rows of the view and count the number of 1 to compute the support. In our example, the support of {I0, I7, I9} is 1 and so the itemset is infrequent for a minimum support of 2. This allows to reduce the size of the matrix and so the computation time for each frequency calculation of an itemset.

Item	T1	T2	T3	T4
I0	1	0	1	0
I7	1	1	0	0
I9	1	0	0	1

Table 2: View of the matrix

This translation of the database in a matrix makes the computation time for the frequencies pretty faster but, nevertheless, the generation of candidates takes time. It's made by combining the frequent itemsets that are the same except for the last item. This operation becomes long when there a lot of frequent itemsets which have a pretty big size. One way to optimize that is to stop the comparison between two itemsets when the first, second or third elements are different. It allows to not traverse all the itemset for each comparison. For example, by doing that, we gained 30 seconds for the algorithm to finish on the chess dataset with a minimum frequency of 0.7.

3 Depth First Search Algorithm : ECLAT

As the algorithm ECLAT uses a different structure of the data. The first implementation that was made was their conversion into a more vertical structure. Instead of having the each transaction with the item composing them, we modify that in order to have a structure where we identify in which transaction each of the item appear. In that way, the generation of candidates can be made by just intersecting the sets of transactions of two, three or more items.
This is the role of the function *to_ vertical(data)* which will output a dictionary where the keys are the items and values are a set composed of every transaction where the item is present. Computing the support of an itemset becomes very easy, it is just the length of the set associated with the itemset.

TIDs	Items
1	1 2 3
2	2 3 4
3	3 4 5
4	2 3
5	3 4
6	1 2 3 4
7	1 2 4
8	5

Table 3: Classical representation

$\Longrightarrow$

Items	TIDs
1	1 6 7
2	1 2 4 6 7
3	1 2 3 4 5 6
4	2 3 5 6 7
5	3 8

Table 4: Vertical representation

For the main part of the implementation, an object oriented one was selected. It allows us to have a cleaner and more comprehensible code.
The class has three attributes which are the total number of transactions in the dataset, the minimum frequency and the dictionary that will contain the frequent itemsets and their frequency. The main class method is runner which is a recursive method that will go through the modified dataset (vertical approach) as shown in table 4. In this method, we will first compute the frequency of an itemset and compare it to the minimum one, then in the second part, through the method *get_ children*, we search candidates in the children of an itemset if it is frequent, the generated candidates becomes the new dataset that will be tested. As it is a recursive algorithm, we develop the network in a depth first search manner. Finally the frequent itemsets are printed and stored in a dictionary where the keys are tuples containing the items in a frequent itemset and the values are the frequencies.
An approach was taken in the order that was used to search for frequent itemsets. It was decides that an goof solution was to attack the lower support/frequency itemsets first in order to increase the speed of the algorithm. Indeed, let's take the example laid out in table 3 (the toy dataset). If we take, as the first itemset which will be explored, the itemset {4}, we see that the generation of candidates if the minimum frequency is 0.125, will generate, at the first level, the itemset {4 , 5} among a big quantity of other itemsets. This itemset would have not been generated while working on the children of {4}, if it was already considered while generating the children of {5}.
As {5} will have less children than {4}, if we start by generating children of {5} before {4}, we reduce the number of children of {4} and thus we reduce the branch which starts with {4}. As the algorithm is a recursive one, the less we use the function by recursion, the quicker the computation will be.

4 Comparison of the performance

This section compares the performance of the 2 algorithms on many datasets with a variable frequency.

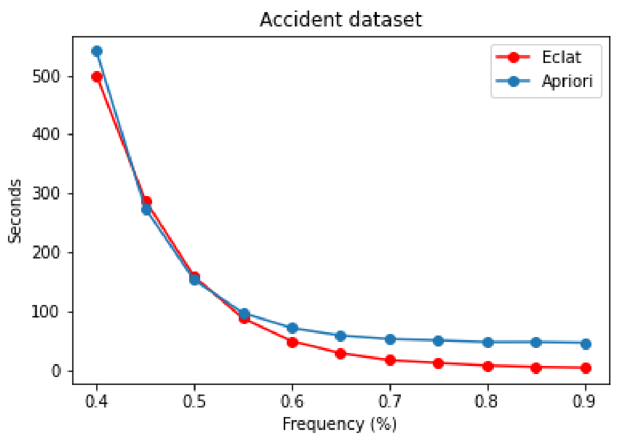

Figure 1: Performance on accident dataset

Here, there is no big difference between the algorithms in performance. It's due to the nature of the dataset. This one has not a big number of different items and so the generation of candidates for Apriori doesn't take much time. As the dataset has a lot of transactions, the initialization of the matrix takes most of the time for Apriori for a frequency greater than 0.6 (like 99 %). Once it's done, the computation becomes very fast. Below 0.6, the number of candidates grows and their generation is hugely impacted.

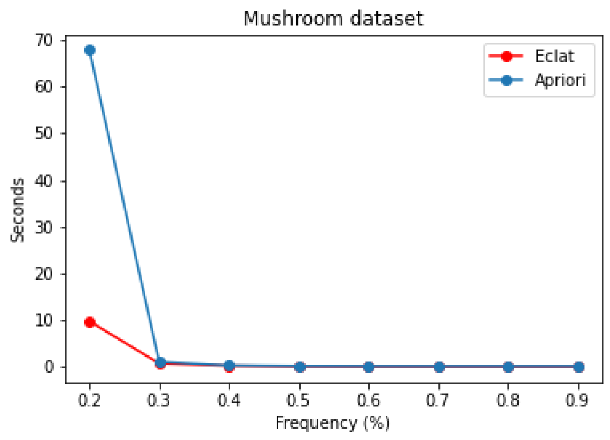

Figure 2: Performance on mushroom dataset

The situation is a bit different for this dataset. We can see that at higher frequencies, the two algorithm have a similar behaviour. Below a frequency of 0,3, the number of frequent itemsets explodes and so, the candidates generation of Apriori is hugely impacted while Eclat handles the situation better by the nature of the search.

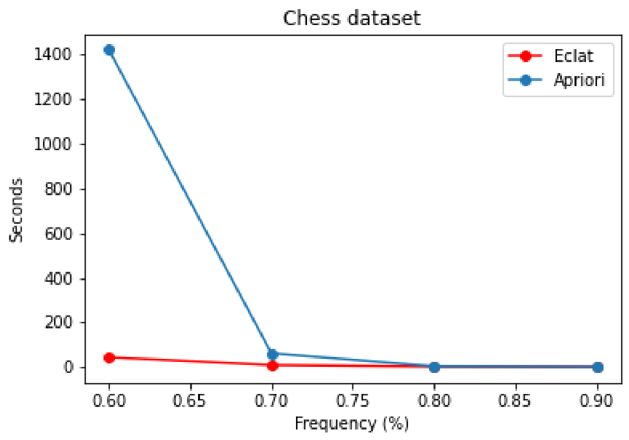

Figure 3: Performance on chess dataset

This dataset has not a lot of different items but a lot of frequent itemsets. Therefor, the generation of candidates for Apriori is once more hugely impacted for a frequency below 0,7.

5 Conclusion

For a large number of frequent itemsets and/or frequent itemsets which have a lot of items, the Apriori algorithm is poorly performing due to the nature of candidates generation. Another data structure like a trie for this operation would be wise. Eclat handles the situation better thanks to the depth first search. Nevertheless, when the number of frequent itemsets is reasonable, the two algorithms behave similarly. In addition, for a large number of transactions, the translation of the database into a matrix takes a larger time for Apriori. It's because each column represents a transaction. But once it's done, the computation becomes pretty fast because, as for Eclat, we don't have to browse the database again.