LINGI2364 - Mining Patterns
Project 1 - Implementing Apriori

Group 10
de Voghel Brieuc - 5910 1600
Mersch-Mersch Severine - 3298 1600

Wednesday 11th March, 2020

1 Introduction

For this project, 2 implementations of frequent itemset mining algorithms had to be implemented and compared: a basic version of the apriori algorithm, called `apriori()`, and a second, more complete and optimized version, called `alternative_miner()`.
In this report, we detail our implementations and the added optimisations. Also, performance tests were run on different datasets in order to compare both implementations.

2 Implementations and Optimisations

2.1 First implementation : `apriori()`

An object is created with the given method that reads the dataset and stores every transaction, and every item in those transactions (see section 2.1.2 for the changes we later added to this initial, given method).

This object is then used to iteratively determine all frequent itemsets, level by level. Each level is a dictionary with the key being a candidate itemset, and the value being the itemset's frequency. Only the previous level needs to be kept in order to determine the next one.

At every level, all occurrences of the candidates are counted by going through all the transactions, in order to determine their support. Then, only the frequent ones (the ones whose support is higher than the minimum support, ie. the minimum frequency multiplied by the number of transactions) are printed with their corresponding frequency.
Finally, those frequent itemsets are combined to form the itemsets of the next level.

2.1.1 Optimisation 1 : Combining itemsets to create the next level

Combining the itemsets to create the candidates of the next level is done by taking the itemsets that have a common prefix (all but one item in common), and adding the two different suffixes. This thus creates a new itemset with a size larger by one.
This optimisation improved compared to naively combining two itemset.

2.1.2 Optimisation 2 : Counting the first level appearances when creating the dataset

When creating the object representing the dataset, we go through all transactions to collect the different items. To use this first pass through the dataset in a useful way, we decided not only to memorise the different items, but also to count their support directly.
This spares us a lot of time when the dataset is composed of lots of different items compared to the number of transactions.

2.2 Second implementation : `alternative_miner()`

Our second frequent itemset mining algorithm is an implementation of the depth first search ECLAT algorithm.

Here too, an object representing the dataset is first created; but this time in the vertical representation with a given projection (starting with an empty projection, of course). The recursive ECLAT algorithm is called with this object.

The vertical representation is a list of transaction identifiers (TID list). It is stored as an ordered dictionary with as key the item and as value a set of the indexes of the transactions where the item appears. The size of the set is the support of the item. In addition, the itemset under which the projection of the vertical representation is based on, is stored as a list.

For each frequent item in this vertical representation, the completed itemset is printed with its frequency. An item is frequent when its support is at least as large as the minimum support. The completed itemset is the itemset under which the projection is done, plus the item itself.
Then, ECLAT is recursively called with the projection of the vertical representation under this frequent item.

2.2.1 Optimisation 1 : Ordered dictionary

The data structure used to store the TID list has to be ordered in order to have a linear (or constant) time complexity when computing the projection. Since a list is way to slow and a standard dictionary is not ordered, we chose to use an ordered dictionary.
This is what really optimised ECLAT.

2.2.2 Optimisation 2 : Sets for the TID lists

When projecting the TID list under an item, the intersection of the items' TID lists have to be computed. This is most easily done with sets.
Originally, lists were used, but this appeared to be a bad implementation choice.

3 Experimental performance tests

3.1 Comparison of both implementations

Even though the implementation of `apriori()` (with it's counting optimisation) passed all performance tests on Inginious, implementing `alternative_miner()` let us really appreciate the efficiency of a depth first search. Even more so, as `alternative_miner()` really has a simple and straightforward implementation.
But how does it compare performance wise ?

3.2 Performance in terms of time

In order to provide reliable performance tests, we developed a testing script. This script executes on 4 different datasets, both algorithms with different minimum frequencies. If an algorithm took more than (in this case) 2 minutes to compute a frequency on a dataset, it will not try to compute the next (smaller) frequency on that dataset.
These are the raw results (mean of 10 different iterations of the script) :

	apriori()				alternative_miner()			
minimum frequency \datasets	retail	mushroom	chess	accidents	retail	mushroom	chess	accidents
0.9	1.04	0.24	12.91	19.72	1.56	0.21	0.61	18.86
0.8	1.03	0.50	187.48	171.25	1.48	0.23	5.02	26.17
0.7	1.02	0.47			1.49	0.34	27.63	42.41
0.6	1.02	1.19			1.47	0.26	145.64	106.25
0.5	1.03	4.98			1.57	0.42		305.03
0.4	1.10	26.15			1.66	0.86		
0.3	1.11	99.92			1.65	2.63		
0.2	1.10	1675.65			1.67	31.66		
0.1	1.65				1.83	299.16		

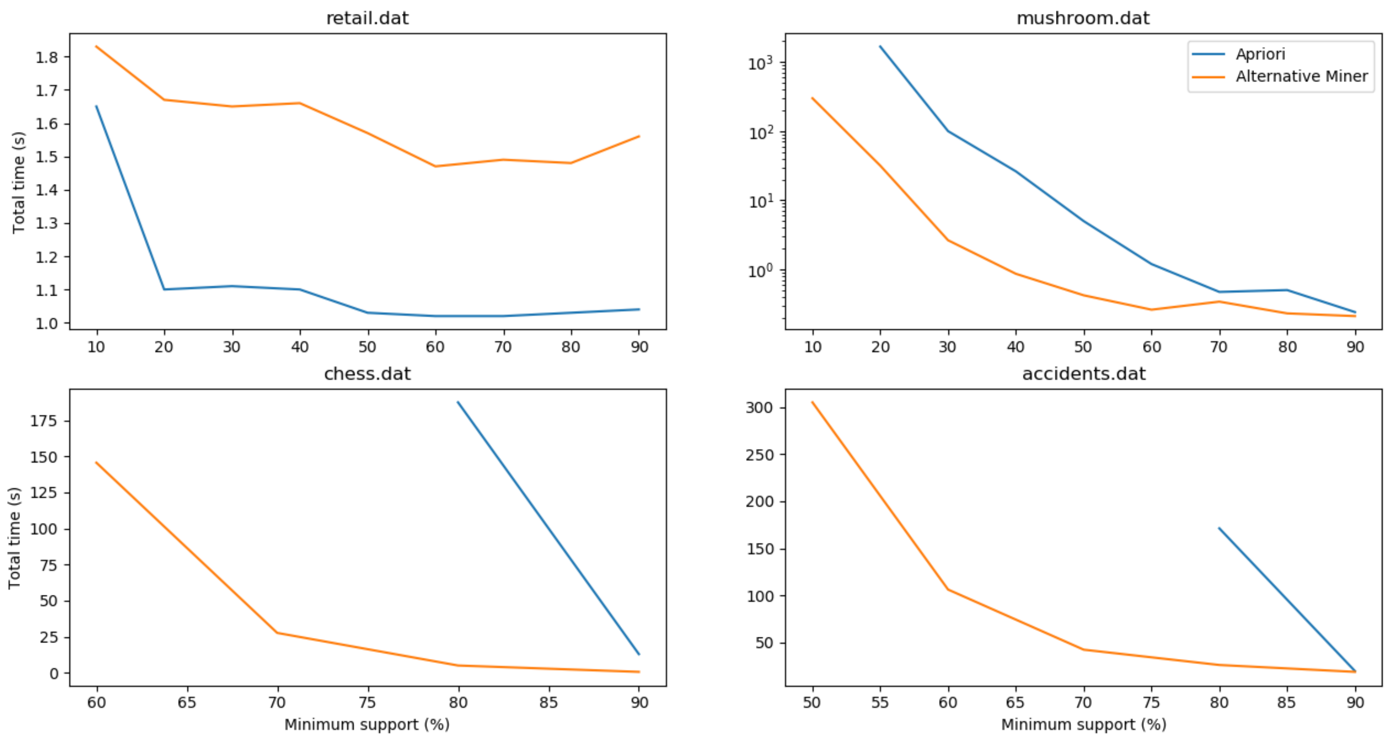

Figure 1: Performance tests in terms of time [1]

As you can see, in nearly all datasets, the ECLAT algorithm (alternative miner) is faster than apriori. When the minimum support is high, this difference is quite small but when it get lower, the difference can become enormous for bigger datasets.

The only exception is the retail dataset. This comes from the fact that it is a dataset that has a lot of infrequent items and thus the "first pass" (see section 2.1.2) counting the support for the apriori algorithm gives a big time gain.

The exponential nature of the curves implies that our apriori algorithm will practically never be able to compute low frequencies on large datasets.

When executing `apriori()` with `-m cProfile`, we notice that the parts of the code taking most of the time is computing the support of all the itemsets at every level. This could be improved by using better adapted data structures for storing the itemsets and the transactions.
The initialisation of the data structure object (reading through the input file) also takes a reasonable amount of time but allows to save time later on for the first level (see above : exception for the retail dataset).

3.3 Performance in terms of memory

This analysis is only theoretical.

Because `apriori()` has to save all candidates of the previous level in order to compute the next one, large datasets consume lots of memory, even for high minimum support.
`alternative_miner()`, on the other hand, is a DFS and thus has a space complexity of $\mathcal{O}(bd)$, where b is the branching factor (number of frequent itemsets) and d the depth (length of longest frequent itemset).

[1] Be aware that for the graph in the upper right corner, its y scale is logarithmic to more easily notice the smaller time difference for large minimum supports.

4 Difficulties encountered

When beginning the implementation design we had some difficulties to understand the difference between the basic apriori algorithm that does not have any optimisation and the more advanced one with candidate pruning.
This confusion was rapidly solved when we dove deeper into the implementation itself.

5 Conclusion

This project let us develop two frequent itemset mining algorithms : apriori and DFS ECLAT.
We analysed the difference in implementation that lead to a big gap in performances, both time and memory wise. ECLAT really stands out because of its optimised depth first approach and the appropriate data structures.