IMDB project
Algorithms for Massive Datasets

Alberto Boggio Tomasaz, Daniel Dissegna

2020-2021

Abstract

In this project we present some implementations of Apriori algorithm, PCY algorithm and SON algorithm for the problem of finding frequent itemsets in a given dataset.
At first, we show the Python code of the implementations (using Google Colab) and then we discuss the algorithm implementations by analysing their resource consuption.
In the end, we present a parameter tuning algorithm for the problem of finding a suitable support threshold.

We declare that this material, which We now submit for assessment, is entirely our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of our work. We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by us or any other person for assessment on this or any other course of study.

1 Introduction

This project uses the IMDB dataset available on `https://www.kaggle.com/ashirwadsangwan/imdb-dataset` to run the experiments, with the purpose of finding frequent sets of actors which occur in the same movies.
The part of the dataset we used is the one contained in the file *title.principals.tsv* where the indexes of movies and every kind of person who worked for that movie (actors, directors, ...) are stored. Obviously we are interested in the actors only so, in the pre-processing we will convert the file title.principals.tsv into a Python list of lists, each of whom representing a basket containing the ids of the actors appearing in the movie (which is the basket).
This way we store in main memory a list of lists instead of reading the file from mass memory.

The implementations of the pre-processing, the algorithms and the experiments can be found in the repository on the following link:
`https://github.com/dingodaniel/Boggio_Dissegna_AMD_project`
or in the Google Colaboratory environment via
`https://colab.research.google.com/drive/1kVYmdURB3lPWHiRInCDQYkdd02WkhKkp?usp=sharing`.

1.1 Method for counting pairs

Counting all the possible pairs to see wheter they are frequent or not is the most tedious part of finding the entire set of frequent items. Without having a good data structure to store these counts, we may go out of memory. There are two main methods to accomplish this:

- Triangular-Matrix Method
- Triples Method

The first method stores one integer for each possible (even if not in the dataset) pair while the second one stores three integers for any pair that needs to be counted at runtime (item1, item2, counter).
When the triangular matrix is sparse, there could be a lot of useless cells that won't be used because the pair associated is not in any basket, suggesting to use the triples method instead. Conversely, if the triangular matrix is dense, the overhead due to triples method might exceed the advantage.
In order to choose which one fits best in terms of memory usage, it's necessary to analyze the dataset. In the following table we show some interesting attributes of the previously mentioned dataset:

I	1868056	number of overall items
B	3697162	number of baskets
P	1744815675540	number of possible pairs (overall)
WP	31901860	worst case on the number of different pairs in baskets

We have that the triangular matrix method needs to store an interger for each possible pair, so P integers are needed.
On the other hand, the triples method, in the worst case, needs 3 integers multiplied by the maximal number of different pairs in baskets, which is $3 \cdot WP$.
We have that $3 \cdot WP = 95705580 < 1744815675540 = P$ so the triples method fits best.

We also added an improvement to the triples method, that is, instead of storing the pair as key, we use the same function used in the triangular matrix method to map the pair into a single integer.

Thanks to this improvement, this method takes 1/3 less memory than the classical triples method.

Another reason why we used triples method is that, since we are using Python to implement the algorithms, the most efficient data structure we can exploit is a dictionary, so that the triples method is easily implicitly implemented.

2 Apriori algorithm

In this chapter we describe the main points/steps of our Apriori algorithm implementation.

1. First pass
   Counts the singletons in every basket and store it in a set the frequent singletons

2. Second pass
   First, we remove from the baskets all the items that are not frequent, that is, the ones that are not in the set of the frequent singletons. Then, we generate every possible pairs in all the baskets (with Python's method it.combinations) and count them in a dictionary according to the method described in the previous section.

   Optimization notes: We also tested two other ways of counting pairs:

   - generate every possible pair in all the baskets (made only by frequent items): for every basket, two nested for in which we skip the non-frequent items.
   - generate the pairs iteratively using the it.combinations iterator and count them only if the items are frequent.

   However, both methods were slower than the first one. Furthermore, the chosen method frees some memory up and also allows a faster counting of the sets in the subsequent iterations (since there are less items).

   Ater that, we store the frequent pairs in a set.

3. $k \geq 3$ pass

   For passes of sets of cardinality ≥ 3, we devised 3 different methods of filtering the baskets before counting all the k-sets. They can be selected by a parameter named itemfilter and the specifications are the following:

   (a) itemfilter=2: this is the standard way of counting the k-sets, that is, for each basket and for each item in the basket, we remove the item if it is not contained in at least k-1 (k-1)-sets that have been found frequent and are contained in the current basket.
   Opt. note: we devised 2 ways of doing so, which are discriminated on the basis of the following condition $\#freq\_sets < \binom{\#basket}{k-1}$. See the code for implementation details.
   This method is the most efficient in terms of allocated memory because it erases the most items from the baskets, leading to a lighter counting of the k-sets in the next phase (but it's less efficient in terms of required time).

   (b) itemfilter=1: this is a relaxation of the previous technique. In fact, we relax the last condition on the membership of the frequent (k-1)-sets to the current basket. In practice, we just count how many times some item appears in the frequent (k-1)-sets and, in the end, we delete every item which appear less then k-1 times. This leads to a faster filtering but uses more memory than the previous itemfilter. Doing so, the k-sets counting phase will be worse because itemfilter=1 filters less items than itemfilter=2, which means that there will be more candidate k-sets in the counting phase.

   (c) itemfilter=0: it simply skips the filtering phase and directly counts all the k-sets in the baskets. Remember that the non-frequent singletons have already been removed from the baskets, so the set of candidate k-sets will be less populated anyway. This approach could be used to significantly speed up the algorithm at the cost of using more memory to count the k-sets in the next phase.

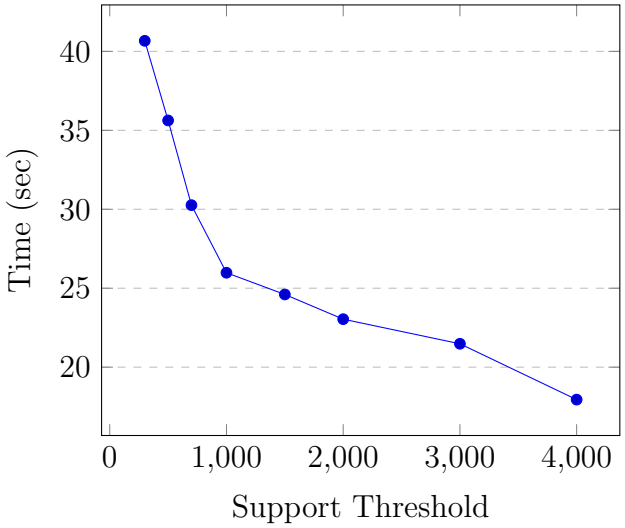
   Opt. note: we also tested the method in which, instead of first filtering the items from the baskets, we just check on the generated k-sets that the condition in (a) holds for each item in the current k-set and, only then, count it. However this approach took not only more memory for the baskets but also more time to count the k-sets.

3 Performance of Apriori algorithm

As the threshold decreases, the number of frequent itemsets will increase leading to a bigger time and memory consumption for the Apriori algorithm. In this section we will study the behaviour of the algorithm in different resource aspects.

Here follows the time consumption of the Apriori algorithm (without item filtering for k-tuples with k $\geq$ 3) as the support threshold grows:

Dependency between support threshold and time



Support Threshold	Time (sec)
4000	17.944
3000	21.481
2000	23.039
1500	24.604
1000	25.980
700	30.260
500	35.624
300	40.668

For what concerns the memory consumption we analyze the occupied memory (for counting) on each pass of the Apriori algorithm with respect to a support threshold of 300.

The following datas are the output printed by the algorithm with the verbose flag set to *True* (and itemfilter implicitly set to 0).

Memory used to count singletons: 83.886 MB	Memory used to count pairs: 5.242 MB
Number of candidate singletons: 1868056	Number of candidate pairs: 121151
Number of frequent singletons: 6014	Number of frequent pairs: 3892
Time to count singletons: 11.592 sec	Time to count pairs: 12.617 sec
Memory used to count 3sets: 10.485 MB	Memory used to count 4sets: 10.485 MB
Number of candidate 3sets: 262902	Number of candidate 4sets: 319465
Number of frequent 3sets: 2773	Number of frequent 4sets: 1774
Time to count 3sets: 3.527 sec	Time to count 4sets: 3.415 sec
Memory used to count 5sets: 10.485 MB	Memory used to count 6sets: 10.485 MB
Number of candidate 5sets: 285349	Number of candidate 6sets: 198122
Number of frequent 5sets: 1066	Number of frequent 6sets: 541
Time to count 5sets: 2.949 sec	Time to count 6sets: 2.153 sec
Memory used to count 7sets: 5.242 MB	Memory used to count 8sets: 1.310 MB
Number of candidate 7sets: 103075	Number of candidate 8sets: 36704
Number of frequent 7sets: 192	Number of frequent 8sets: 41
Time to count 7sets: 1.628 sec	Time to count 8sets: 1.193 sec
Memory used to count 9sets: 0.295 MB	Memory used to count 10sets: 0.036 MB
Number of candidate 9sets: 7773	Number of candidate 10sets: 731
Number of frequent 9sets: 4	Number of frequent 10sets: 0
Time to count 9sets: 1.152 sec	Time to count 10sets: 1.107 sec

We can notice that the most consuming pass is the first because of the memory consumption due to the singletons counting. Obviously when it comes to counting singletons there is no problem because it's not a big data problem.

For what concerns pairs, we can see that, for this particular dataset, we don't have any problem storing the pair-counting data structure because the memory consumption is even lower than in the first pass.

In the next passes, we can notice that the time consumption decreases, whilst the memory used for counting triples, quadruples... is initially higer than in the pair pass, but then decreases. This behaviour is in accordance with the theoretical previsions because we actually used no filters on the items before counting the k-sets.

However, that doesn't mean that for other (bigger or more complex) datasets, the behaviour will be the same. In order to address the problem of a possible growing of the number of candidate k-tuples (which can be exponential in k), we might want to use the item filtering techniques to decrease the memory consumption (increasing the time complexity).

The following datas are the output printed by the algorithm with the verbose flag set to $True$ and itemfilter set to 1.

Memory used to count singletons: 83.886 MB	Memory used to count pairs: 5.242 MB
Number of candidate singletons: 1868056	Number of candidate pairs: 121151
Number of frequent singletons: 6014	Number of frequent pairs: 3892
Time to count singletons: 11.170 sec	Time to count pairs: 12.399 sec
Memory used to count 3sets: 0.589 MB	Memory used to count 4sets: 0.295 MB
Number of candidate 3sets: 16622	Number of candidate 4sets: 5766
Number of frequent 3sets: 2773	Number of frequent 4sets: 1774
Time to count 3sets: 4.289 sec	Time to count 4sets: 2.240 sec
Memory used to count 5sets: 0.073 MB	Memory used to count 6sets: 0.036 MB
Number of candidate 5sets: 2142	Number of candidate 6sets: 1342
Number of frequent 5sets: 1066	Number of frequent 6sets: 541
Time to count 5sets: 1.277 sec	Time to count 6sets: 0.992 sec
Memory used to count 7sets: 0.018 MB	Memory used to count 8sets: 0.004 MB
Number of candidate 7sets: 558	Number of candidate 8sets: 153
Number of frequent 7sets: 192	Number of frequent 8sets: 41
Time to count 7sets: 0.806 sec	Time to count 8sets: 0.812 sec
Memory used to count 9sets: 0.0002 MB	Memory used to count 10sets: 0.0002 MB
Number of candidate 9sets: 4	Number of candidate 10sets: 0
Number of frequent 9sets: 4	Number of frequent 10sets: 0
Time to count 9sets: 0.619 sec	Time to count 10sets: 0.608 sec

And the following are the output printed by the algorithm with the verbose flag set to *True* and itemfilter set to 2.

Memory used to count singletons: 83.886 MB	Memory used to count pairs: 5.242 MB
Number of candidate singletons: 1868056	Number of candidate pairs: 121151
Number of frequent singletons: 6014	Number of frequent pairs: 3892
Time to count singletons: 10.755 sec	Time to count pairs: 12.745 sec
Memory used to count 3sets: 0.295 MB	Memory used to count 4sets: 0.147 MB
Number of candidate 3sets: 6126	Number of candidate 4sets: 3515
Number of frequent 3sets: 2773	Number of frequent 4sets: 1774
Time to count 3sets: 30.960 sec	Time to count 4sets: 4.136 sec
Memory used to count 5sets: 0.073 MB	Memory used to count 6sets: 0.036 MB
Number of candidate 5sets: 1903	Number of candidate 6sets: 1229
Number of frequent 5sets: 1066	Number of frequent 6sets: 541
Time to count 5sets: 2.105 sec	Time to count 6sets: 0.950 sec
Memory used to count 7sets: 0.018 MB	Memory used to count 8sets: 0.004 MB
Number of candidate 7sets: 557	Number of candidate 8sets: 153
Number of frequent 7sets: 192	Number of frequent 8sets: 41
Time to count 7sets: 0.450 sec	Time to count 8sets: 0.185 sec
Memory used to count 9sets: 0.0002 MB	Memory used to count 10sets: 0.0002 MB
Number of candidate 9sets: 4	Number of candidate 10sets: 0
Number of frequent 9sets: 4	Number of frequent 10sets: 0
Time to count 9sets: 0.136 sec	Time to count 10sets: 0.015 sec

It is possible to see that the memory consumption is lower using $itemfilter = 1$ and even lower using $itemfilter = 2$ but the time complexity increases (actually it reaches 30 seconds in the 3rd pass using $itemfilter = 2$).
However, the memory consumption in the second pass (pairs) is not affected because the item filter we used does not apply to the pairs (or rather, it is applied but our implementation differenciates the two in order to gain efficiency).

Again, our dataset produces a limited number of pairs, so we don't actually have any problem counting them, but this problem might occur with bigger and more complex datasets so we addressed this problem using PCY.

4 PCY

Our PCY Algorithm is exactly the same as the Apriori with some slight modifications.
In the first pass, we do not only count the singletons but we also use two hashing functions to hash every pair of the baskets in a bucket. After doing so, we create two bitmaps in which we store whether the buckets are frequent or not.

Then, in the second pass, before counting the pairs, we also check wheter the selected pair is mapped in a frequent basket or not for both the hash functions. If so, we can count it, otherwise, we can discard it.

For implementation reasons, instead of using bitmaps, which are not simple to handle in Python, we exploited two Python sets containing just the frequent bucket, which is obviously a worse solution in terms of speed and memory consumption, but we think it is the simplest and most efficient way to emulate a bitmap.

4.1 Performance of PCY

In section 3 we analyzed the time and memory consumption of Apriori algorithm, we now compare it with the PCY resource management.
Follows the output printed by PCY with support threshold 300, 100000 buckets, verbose flag set to $True$ and itemfilter set to 2.

Memory used to count singletons: 83.886 MB	Memory used to count pairs: 1.310 MB
Memory used to count buckets 1: 5.242984 MB	Number of candidate pairs: 39094
Memory used to count buckets 2: 5.242984 MB	Number of frequent pairs: 3892
Number of candidate singletons: 1868056	Time to count pairs: 18.753 sec
Number of frequent singletons: 6014	
Number of frequent buckets 1: 41255 / 100000	
Number of frequent buckets 2: 41249 / 100000	
Time to count singletons: 94.528 sec	
Memory used to count 3sets: 0.295 MB	Memory used to count 4sets: 0.147 MB
Number of candidate 3sets: 6126	Number of candidate 4sets: 3515
Number of frequent 3sets: 2773	Number of frequent 4sets: 1774
Time to count 3sets: 31.370 sec	Time to count 4sets: 4.121 sec
Memory used to count 5sets: 0.073 MB	Memory used to count 6sets: 0.036 MB
Number of candidate 5sets: 1903	Number of candidate 6sets: 1229
Number of frequent 5sets: 1066	Number of frequent 6sets: 541
Time to count 5sets: 2.147 sec	Time to count 6sets: 0.936 sec
Memory used to count 7sets: 0.018 MB	Memory used to count 8sets: 0.004 MB
Number of candidate 7sets: 557	Number of candidate 8sets: 153
Number of frequent 7sets: 192	Number of frequent 8sets: 41
Time to count 7sets: 0.414 sec	Time to count 8sets: 0.170 sec
Memory used to count 9sets: 0.0002 MB	Memory used to count 10sets: 0.0002 MB
Number of candidate 9sets: 4	Number of candidate 10sets: 0
Number of frequent 9sets: 4	Number of frequent 10sets: 0
Time to count 9sets: 0.135 sec	Time to count 10sets: 0.021 sec

As expected, the only passes that were affected by PCY are the first and second ones (singletons and pairs). In the first pass, we have to enumerate every pair in each basket (time consumption increases) and store a count for each bucket (memory consumption increases). However, in the second pass, the memory used to count the pairs decreases significantly due to the bucket filtering at the cost of keeping two bitmaps ($2 \cdot 100000$ bits).
The results for the $k \geq 3$ passes are the same as the Apriori for all the itemfilter versions, so no farther analysis are needed. Obviously, we could generalize PCY to perform the bucket count also for k-tuples with $k \geq 3$, but we think that it would not be necessary because the already used item filters are enough (but there might be some datasets in which it is necessary).

An unusual result is that we used more memory to store bucket counters than the memory we would have used in the second pass of Apriori but, again, this is because the dataset examined does not require PCY in the first place, so we just needed to show that using a reasonable number of buckets we can decrease memory usage in the second pass.

5 SON algorithm

All the algorithms we discussed until now assumed there is enough main memory to store:

- the entire datasets
- the counts for the candidate itemsets at any pass

However, one of the two assumptions may not hold, especially when we are dealing with big data.

For this reason, we also implemented the SON algorithm both in a standard and distibuted fashion.

The main steps of the algorithm are the following:

1. split the baskets in $\frac{1}{p}$ chunks according to a given fraction p.
2. run some algorithm to find the frequent itemsets (Apriori, PCY, ...) on each chunk with support threshold $p \cdot original\_supp\_thr$ and get the candidate frequent itemsets. It is possible to prove that there cannot be false negatives generated this way.
3. filter out the false positive itemsets by scanning all the baskets.

These steps lend themselves to a parallel computation, so we decided to implement them with a Map-Reduce framework.

The steps are the following:

1. First map function: takes the chunk and returns the candidate frequent itemsets by using the Apriori algorithm with support threshold $p \cdot original\_supp\_thr$. The output is a set of (key, value) pairs (kset, 1) where kset is a frequent itemset from the chunk.
2. First reduce function: simply ignores the value and produce the key of each itemset eliminating any double (that could have been produced from multiple chunks). The output is the set of candidate frequent itemsets which is the union of the ones produced by every chunk.
3. Second map function: takes in input the set of all candidate frequent itemsets and a chunk, then returns the number of occurrences of each of the candidate itemsets among the baskets in the chunk. The output is a set of key value pairs (kset, count) where kset is the candidate frequent itemset and count is the number of occurrencies in the chunk.
4. Second reduce function: returns the sum of the occurrencies of every given itemset.

5. Those itemsets whose count on all the baskets is $\geq$ than the original support threshold are returned as frequent itemsets (filtering of false positives).

This algorithm makes sense only when we cannot store the entire dataset in main memory, otherwise it would require a bigger amount of time (especially in the second map-reduce step). For this reason we won't investigate the time and memory complexity but just check that the result is correct.

6 Tuning of Support Threshold

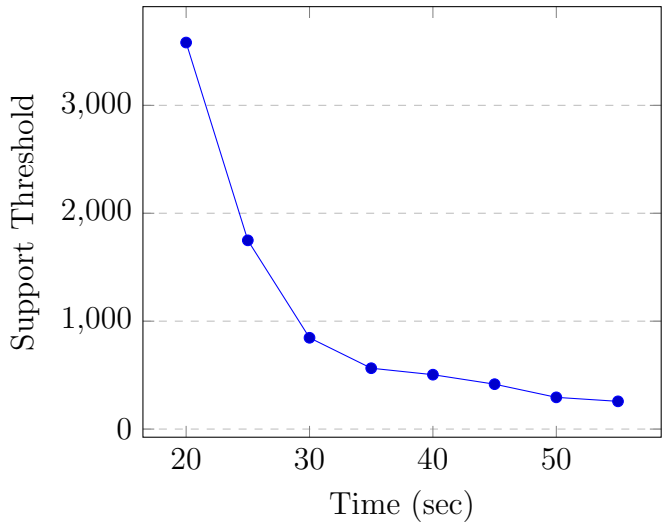
A very important problem to solve, when it comes to finding frequent itemsets, is the tuning of the support threshold. In fact, the whole output of the algorithms will depend on the choice of a suitable threshold.
Our approach to this problem is to find a support threshold value such that the computation of the Apriori algorithm doesn't take more than a given time to finish. This way we can use a dicotomic search techinque to find the minimum value such that the algorithm finishes the computation in less than a given number of seconds.
Obviously, the same reasoning could be done with the memory consumption but we addressed that problem earlier with the itemfilters and PCY algorithm. The reason why we find this criteria valid is that it is a tradeoff between performance and quality which gives us the opportunity to control the time consumption and at the same time use the best (minimal) support threshold for the Apriori, PCY or SON algorithm.
The algorithm works as follows:

- Set the extremes for dicotomic search to
  $left := 1$
  $right :=$ # of items.
- unless $left$ and $right$ differ for a significant amount (their relative gap is higher than 1%)
  - $mean := \lfloor \frac{right+left}{2} \rfloor$
  - Run Apriori algorithm, with $mean$ as support threshold, until it reaches the given maximal time
    - if the algorithm finished in time then $right := mean$
    - if the timeout occured then $left := mean$
- In the end, the value of $right$ corresponds to the minimal value of the support threshold for which the algorithm finished before the timeout, so we will use it as our optimal value.

Obviously, we could have used either PCY or SON instead of Apriori, but in our case there was not any memory consumption problem and the faster implementation was indeed the one for Apriori algorithm.

In the following figure we show some results of the parameter tuning for some required computational time.

Dependency between support threshold and time



Time (sec)	Support Threshold
20	3582
25	1749
30	846
35	564
40	504
45	416
50	294
55	257

7 Conclusions

In this work we explored some different techniques to solve the problem of finding frequent itemsets analysing multiple options that can be used depending on the instance we want to study, providing an explanation of when to use those options and why.