Algorithm for massive datasets : Counting frequent actors pairs

Théotim Barbier

Abstract

This project take place in the course of Algorithm for massive datasets conducted by D. Malchiodi In the Università degli Studi di Milano Statale. This course aims to devellop method to deal with massive datasets. In this project, we will see an application of this course in the problem of counting frequent itemsets. You can find a the github repository here.

I declare that this material, which I now submit for assessment, is entirely my 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 my work. I 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 me or any other person for assessment on this or any other course of study.

Contents

Part I

Introduction

The goal of this project is to count the pair of frequent actors that appears in movies. But, since the quantity of movies that have to be processed can be huge, we need algorithm that can be used over massive dataset. Therefore, we relieve to the problem of counting frequent itemsets in dataset, or in other words, the market basket analysis. The aim of this subject, as its name suggests, is to count frequent set of items considering that the items are contained in batches. For exemple, considering a market and more precisely all the basket bought in the market, it can be used to find wich items are frequently buy together. In our case, we consider our movies as batches of actors and we want to find the most frequents pairs of actors in these movies.

All this project will be implementing in a jupyter notebook available here. Everything is coded in python3 and the main libraries used are numpy, pandas and pyspark.

Part II

Dataset

In this project we will use the «Letterboxd» dataset, which is published on Kaggle and released under the GPL 3 license. This dataset regroup informations about a huge amount of movies coming directly from the website `letterboxd.com`. This dataset was used with the version of the 27th may 2024.

Loading the data To load the data, I use the kaggle API to download the csv file and load it using panda or pyspark considering the methods used.

organisation For our purpose, this data contains a list of actors associated to movies in the dataset (indicated with their movie identifier in the dataset) in the actors.csv file of the dataset. Therefore, we just have to regroup this list by movies to get our batches.

Pre-processing Since the data are provided as a list of pair of movie-actor, we do not need to care about error in datasets. In fact, the only errors that can occurs are that an *id* of a movie or a name of an actors is miswritten. But this will not have a real impact in our algorithms if we consider that we are processing a huge amount of the data. We will also need to index every actor to map them to integer. But this will be made during the preleminary step of the algorithms.

Part III

Algorithms

To solve this problem, we will see the use of two differents algorithm : the PCY and the SON algorithm. These algorithm can be used on massive dataset. In this project, I will propose you an implementation of the PCY and SON algorithm using only numpy array and I will also use spark and Map-Reduce for a second implementation of SON algortihm.

1 Preliminary step

For both of the algorithms, we need first to recover our frequent actors as single set. To do so, we just count our actors occurencies considering our basic dataset. Then we get our list of actors and frequencies. Now, we can also use this list to create our mapping of the actor's name to integer. Then we apply this map to our basic dataset and then we create our batches of actors index corresponding to each movie.

We now also have to create our treshold s. To do that, we will collect in our frequent actors the value that countains 1% of the most frequent actors. This value of 1% is in some sense arbitrary, this should be choosed regarding the problem we are facing. Recall that the list of actors can be handle in one machine, then it is quite easy to sort this frequency and recover the 1% most frequent actors.

Finally, for reasons of conveniences, we also create a boolean list that map the index of every actor to the test if its frequency is above the treshold s.

2 Main algorithm

In this part, we will focus on the main part of each algorithms used in this project.

Let's recall that counting the frequency of each pair of actor is obviously impossible because it would result in storing a counter for every possible pair of actor in the main memory. Considering n actors, this could result in the worst case to around 2^n counter which can not be handle in a single machine when n is too big. Therefore we need clever algorithms to avoid the storage price of storing all these counter.

Main idea of a priori algorithm and monotonicity assumption

The main idea of all those algorithm resides in the monotonicity assumption : If an item is not frequent then the pair containing this items can not be frequent. Then to check if a pair is frequent, an idea is to check if each of its item are frequent as a single set. This is called the a priori algorithm.

We apply this idea in two different algorithm :

2.1 PCY algorithm

2.1.1 Idea

This algorithm is based on the clever idea to use hash function to distribute our pair. As a matter of fact, in a first phase, we hash every possible pair in bucket. But we don't keep the pair, we only add 1 to the bucket. Therefore, it is evident that the larger the value of the bucket, the more frequent the pair hashed in this bucket. With this, we can only keep the most frequents buckets. Then, in a second phase, we do a second check for every pairs and create/update a counter only if the pair is hash in a frequent bucket and has two frequent actors. At the end, theoritically, we get all the frequent pair without False Positive (FP) and False Negative (FN).

2.1.2 Implementation

This algorithm can be implemented using several hash function but in this project, we will only see the case of 1 hash function.

The implementation is quite easy since it only consist on 2 scan of every pairs. Each scan is simply a double loop over all the batches to catch every pairs.

For the hash function, I used the basic hash function implemented in python that I rescale in the range of the bucket. Then the bitmap consists only in a array list containing a counter for each bucket. Note that the number of bucket can be chosen in order to be able to store all the hash counter in the main computer.

For the second phase, the counters are now stored in a dictionary mapping pair of actor to their frequency in all of the batches. Note that we do not care about the order in the pairs, $\forall a_1, a_2$ actors :

$$(a_1, a_2) = (a_2, a_1)$$

I have chose here to always take in the first position of the pair the minimum reffering to the index of the actors.

2.1.3 Treshold and number of bucket

I said before that theoritically, the algorithm is exact, but, in fact, it only depends on the treshold you chose. This will be a recurrent problem in all the algorithm we will see.

Here we need to chose a treshold to keep only the bucket that we can consider frequent. The theory say that the hash function map uniformly the pair over the bucket. We can also see that each pair is counted at least once and frequent pair are counted at least s times (the treshold for single actors) . Then, considering a bucket with at least one frequent pair, at the end of the first phase, this bucket has a counter of at least : average number of distinct pair hashed in a bucket + s (the minimum value is that there is one frequent pair counted s times and every other pair is counted 1 time). The average number of distinct pair hashed in a bucket is equal to the total number of distinct pair divided by the number of bucket. The problem is that we don't know the total number of distinct pair. To compute this value we need to count every pair which is impossible regarding to our problem of storing all these pairs.

The solutions proposed to solve this problem of treshold in this project are the following :

- We could just use the usual treshold s without rescale it. This is an easy option that surely avoid creating FN. This works well when the number of bucket is rescale considering the number of data processed in order to keep a limited amount of pair hashed in each bucket. However, by doing that on big dataset, the treshold become usually too low and then every bucket result to be considered as frequent. This is not a problem for mistake but then the PCY algorithm is no more usefull and we could just have run an a-priori algorithm.
- A second option that can be used is to keep only a part of the bucket without consider the treshold. For exemple, we could keep only $\frac{1}{2}$ of the most frequent bucket. This surely could result in creating FP. But experimentaly, it worked quite well on the amount of data I processed in my computer.

2.2 SON Algorithm

2.2.1 Idea

This algorithm is based on the idea of instead process all the data in a row, we can process data chunk by chunk. In fact, by partitioning our dataset we can perform an a-priori algorithm on each partition of the dataset. To do so, we also need to rescale our treshold considering the number of batches contained in the chunk. After that, we get candidate pair for each chunk, then we just have to take the union of all the candidates on every chunk. After doing that it only remains to check for FP by counting the frequency of every candidate pair over the whole dataset. Recall that the number of candidate should be very lower than the number of possible pair, so the counter could be stored in the main memory.

We will see 2 implementation of this algorithm using only classical libraries (as numpy) and using pyspark library.

2.2.2 Implementation with numpy

First, we need to implement the a-priori algorithm. Actually, in the jupyter notebook I implemented a more general algorithm, called *simple random algorith*m, which perform the same computation as "a priori" algorithm but only on a proportion p of the data. With that, we just need to split our array of batches and perform the algorithm on every chunk of the data. It returns an array list of candidate for each chunk and then we can join the candidates to make an array list of all the possible candidates. After that we perform a scan of all the pair with a double loop to recover the counters of our candidates pairs that we stock in a dictionary. Finally, it remains to filter out the FP which counters are lower than the treshold.

2.2.3 Implementation with spark

What is spark? Spark is a framework used for computation over large dataset available in scala, Java, R and python. In our case, this framework is very usefull in order to process data coming from

any type of data storage (for example any Distributed File System). In fact, it is not dependant of how the data are stored, which make it a really powerfull tool. Spark can also perform classical big dataset method such as Map-Reduce.

First, we need to start a spark session in order to use spark. To do so, I have used the same method as seen in the lecture of "algorithm for massive dataset". We also need to import pyspark which is the python version of spark.

Data loading and pre-processing with spark In our case, data are contained in a csv file. Then we just have to use the *read* function of spark which will return a dataframe, a class of spark which theoriticaly contain data. In practice, spark does not compute any function until it is really needed, probably to maintain its adaptability to any data storage. After that, we limit our data processed using the *limit* function of spark dataframe. Now we need to recover everything that we need to perform our algorithm :

- Single frequency of actors
- Index of actors
- batches
- create treshold

Actually, considering the assumption that the single frequency of actors can be stored in a single computer, I could have only used numpy and the *groupby* method of spark to compute all this variables. But, I chose to do it with several spark methods instead. In all the case the methods are almost the same used previously without spark.

The main thing which change is that the batch dataset is now store in a Resilient Distributed Dataset (RDD). This is a spark object that can perform map and reduce function efficiently. In fact, this object is like a DFS where the data are separated in chunk. Once again, this object is not really computed until a function is called on this object. Here we use the *repartition* function on our dataframe to simulate a DFS. This function separate our data in several partition, which can be considered as chunk in a DFS. Therefore, we will be able to use the mapPartitions function of spark to recreate the real computation of a map function on a DFS.

Map-Reduce on SON algorithm The transcription of the SON algorithm in a Map-Reduce way is really easy. As a matter of fact, we have to consider our previous chunk of the algorithm as the chunk in our DFS. Then, the algorithm works as following :

1. A first *map* function recover all candidate for each chunk and return them as a key value pair :
$$(1, candidate), \forall candidate$$
2. The first *reduce* function just merge all the key value pair in the list of all the candidates over all the batches.

3. The second *map* function return the count of every candidate in each pair :

$$(candidate, count), \forall candidate$$

4. The last *reduce* function return then the sum of every count for each candidate which becomes our final counters.

Implementation with spark The *mapPartitions* function of spark works as follow : it gives an iterator over a chunk to a function in the input and return the result of the function for each chunk. Then for the first *map* function, we can use almost the same *simple random algorithm* function as before. The second *map* function is simple cause it consist to count the candidate in each chunk. So I implemented first as a double loop over the batches which check if the pair is in the candidates. But we will see later that it can result to some problems of time in the computation. The *reduce* functions are really classical function as it's only adding values or joining values. After all the Map-Reduce, it remains only to check if the candidates are above the treshold.

In the both implementations, the algorithm return an exact result.

2.2.4 Problem of treshold and number of candidate

Once again, in practice with have a problem with the treshold or a problem with the process of the second map. In the first implementation of SON algorithm, I used the classical reduce treshold to the size of the chunk. But when I started the computation it's take so much time that it has never finished. The reason why is that the number of candidate could be very large depending on how the data is splitted into chunks. Then, when iterating the double loop to produce every pair of each batches, it has to check if the pair is in the set of candidate. But it has to do that for every possible pairs in our batches. Then, in the worst case (not real but an estimation) you could have to do $nb_{candidate} * 2^{actors}$ check in total. This is for sure reduced using the parrallelization of the Map-Reduce function, so it's not really a problem when handling real big dataset. But it remains a problem on my own computer.

Therefore, my first idea was to increase a bit the treshold of the chunk in order to produce way less candidate. Doing that, I assure that the algorithm will take a reasonnable time to process. Nevertheless, it makes the algorithm become no more exact. This is the implementation in the

Then the second idea was to process the second loop in a clever way. In fact we can avoid to do all those check by removing the non-candidate of the batches. To do so, we extract a list of single candidates from the list of pair candidate. Then, for each batches we remove all the actors that are not in our single candidates. And now we can do a double loop on the new batches which contains only the singles candidates. Therefore, the number of check is reduced a lot. This is more efficient because the average number of actors in each batch is quite huge (around 40 I think) otherwise the classical check is probably better.

In this project, both of the solution are implemented on the SON algorithm without spark. But for the spark algorithm, only the efficient way is implemented.

Part IV

Experiments

The experiments are conduct on google colab using my computer which has a processor Intel® Core™ i5-7200U CPU @ 2.50GHz × 4 and 12,0 Gio of RAM. I used exactly the same jupyter notebook as present joined to this report or in the github depository.

parameters We have several parameters on that we can change in the jupyter notebook :

- **data_size** or **sp_data_size** : this parameters fix the number of data we want to process for the algorithm. This cut the data taken in the csv file of actors, then it is largely possible that we cut the data in a single batch. But we consider that this is not really important, considering the amount of data processed.
- **nb_bucket** : this parameters fix the number of bucket used in the PCY algorithm. A reasonable choice is to take a number of bucket equals to 1% of the batches.
- **treshold** : As mentionned before, the treshold is the parameters fixing if a candidate is frequent or not. Usually we take the treshold such as 1% of the single actors are frequents, but we could consider other treshold for other purpose.
- **n** (and **p**) : this parameters is fixing the amount of data we want in each chunk for the SON algorithm (without spark). Then we can calculate p such as : $p = \frac{n}{nb_{batches}}$, which is the number such as we get $\frac{1}{p}$ chunk. Recall that p is used to reduce the treshold when processing chunk.
- **nb_chunk** : this number correspond to the number of chunk we want in the SON algorithm with spark. In reality, this number is fixed by the number of chunk in the DFS. But here to emulate this, we use the *repartition* function of spark which separate a dataframe into partition to compute the *mapPartition* function.
- **proba** (not used) : this is the parameter of the probability for the *simple_Random_Algorithm*. It determines the percentage of data processed during the algorithm.

In this part, we will see some test of the algorithms presented so far. We will test them only of a part of the data in order to make the computation process in a reasonable amount of time. Here for exemple, we only take 20% of the data. We can therefore produces result with all the 4 algorithms we have seen.

We have seen that every algorithm we use is exact. And, we see by these experiments that all the algorithms return the same frequent pairs (same as *figure 1*), which is a first good point.

Now we can compare the execution time of each algorithm. We perform every algorithm for 20% of the data after the pre-processing part. The *table 1* is presenting these results. We see that the most efficient in our case is the SON algorithm without spark. However, we are only processing a small amount of data here so the use of spark is not really usefull here. But if we want to process

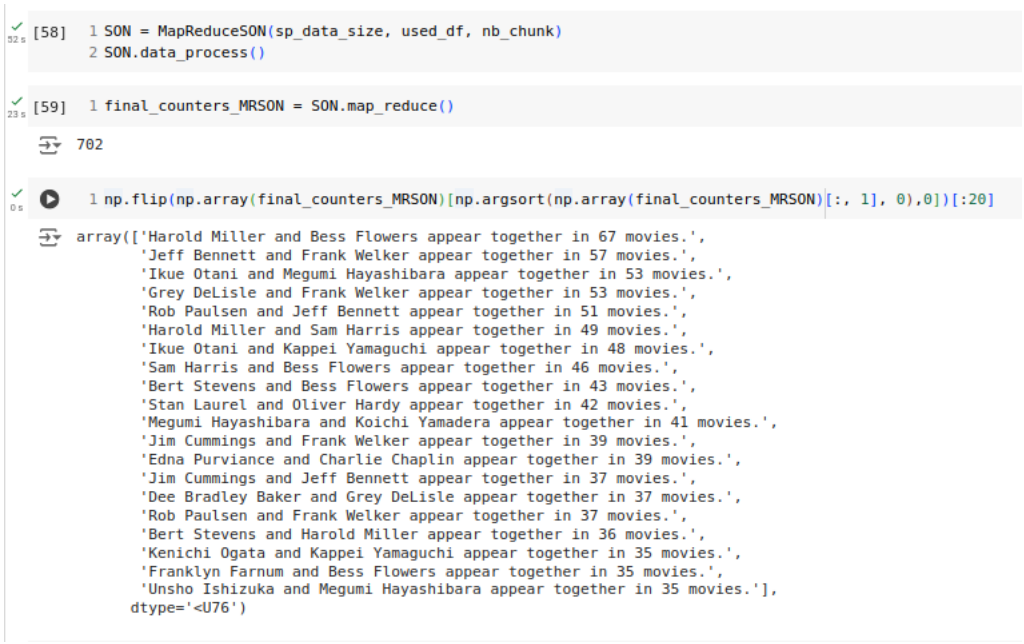

Figure 1: Output of SON algorithm with the parameters : 20% of data, nb_chunk = 3

	PCY	SON without spark	SON efficient without spark	SON spark
processing time (s)	35.776	20.846	5.088	23.961

Table 1: Processing time for every algorithm with 20% of the data. The number of bucket for PCY algorithm is 1% of the batches processed. The SON algorithms are processed over around 3 chunks of the data.

a huge amount of data stored in a DFS, it is for sure the best option we have there. Let's also add that the computation of SON with spark could be easily parrallelized which also reduces the amount of time.

Part V

Conclusion

To conclude this report, we have seen in this project several algorithm to manage the problem of counting frequent items with massive dataSet and their implementation in python3. Moreover, we have seen the use of spark in order to manage computation over massive dataset, with the Map-Reduce method.

Part VI

Bibliography

- Lecture of *Algorithm for Massive Dataset*, conducted by D. Malchiodi In the Università degli Studi di Milano Statale during 2023/2024 year.
- *Mining of Massive Datasets*, written by A. Rajaraman and J. Ullman