Frequent itemsets for IMDb datasets

Rumi Alberto
University of Milan,
Algorithms for massive datasets.

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.

1 Introduction

In this project I've developed and analyzed different algorithms we discussed during lecture which aim is to find frequent itemsets from a dataset. This study is applied to the IMDb dataset published on Kaggle [1] under IMDb non-commercial licensing and describes how the A-Priori, PCY, SON and Toivonen algorithms are implemented and how my implementation performs. Finding frequent pairs is the most memory and time consuming step in all the algorithms, so I decided to focus the implementations on this step.

2 Datasets description

As described by the project specifications, I used the IMDB dataset. This dataset is divided into 5 different TSV files, each one containing different information about different films. The file I am most interested in is the one containing the `principal` cast for each film. In this file [1] I'm interested in:

- **tconst** (string) - alphanumeric unique identifier of the title.
- **nconst** (string) - alphanumeric unique identifier of the name/person.
- **category** (string) - the category of job that person was in.

In order to make some analysis of the results the `film` dataset is also important, where there is:

- **tconst** (string) - alphanumeric unique identifier of the title.
- **titleType** (string) – the type/format of the title (short, movie, tvMovie, tvSeries, tvEpisode, tvShort, tvMiniSeries, tvSpecial, video, videoGame).

3 Data organization and preprocessing

Usually in the frequent itemsets analysis the data is represented as:

$$Basket \rightarrow SetOfItems$$

In our case the data is represented in a different way. Each row of the dataset represents a film in which an actor worked.

In addition to this the data is not sorted by film so we can end up in a situation like:

Good Will Hunting $\rightarrow$ *Matt Damon*
Good Will Hunting $\rightarrow$ *Ben Affleck*
Gone Baby Gone $\rightarrow$ *Matt Damon*
Gone Baby Gone $\rightarrow$ *Ben Affleck*
Good Will Hunting $\rightarrow$ *Robin Williams*

In order to deal with this kind of data I decided to sort the dataset by titleID. This allows not to reshape the whole dataset, but is enaught to keep just the current set of actors for each film while iterating over the whole dataset.

To make a complete analysis of the results 3 main datasets preprocessing teqnique are applyed:

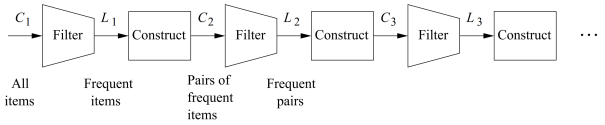
1. Principal dataset without any filtering.
2. Principal dataset where films are filtered in order to have $titleType = movie$.
3. Principal dataset where films are filtered in order to have $titleType = movie$ and principal $category = actor \vee actress$.

4 Algorithms and implementation

The algorithms to solve the frequent itemsets problem take advantage of the monotonicity property of frequent itemsets, which states that if a set I of items is frequent, then so is every subset of I. The first algorithm that we saw is the A-Priori algorithm, so I've started from this one.

4.1 A-Priori

This algorithm is divided into different phases schematized in the following schema [2]:



In my implementation I've focused on the frequent pairs and triplets without going any further, but to implement the functionality to find even also bigger subsets would be done still following this schema.

The algorithm proceeds as follows:

1. **First scan** of the dataset, finding singleton frequency and creating the mapping from the actor ID (string) to integer ID and it's corresponding inverse.
2. **Second Scan** of the dataset, finding frequency for pairs considering only actors whose singleton frequency is more than the support threshold.
3. `-if requested-` **Third scan** similar to the second scan in order to find frequent triplets of actors, considering only actors already frequent as pairs.

4.2 PCY

This algorithm has a similar structure to the previous one, but optimize the use of the memory in the first pass using a *hash function* for the pairs and a *hash table*:

1. **First Pass**, for each basket:
   (a) Compute singleton frequency table and create the mapping from the actor ID (string) to integer ID and it's corresponding inverse,
   (b) For each possible pair, compute the hash function of the pair and increment the respective value in the hash table.
2. Compress the hash table into a `bitarray` on which 1 denotes a frequent hashed pair and 0 a not frequent hashed pair.
3. **Second Pass**, for each basket:
   (a) Consider only actors which singleton frequency is higher than the support threshold.
   (b) For each pair, consider only the ones which $bitmap[hashfunction(pair)] = 1$
   (c) Count frequency of the selected candidates pairs
4. Filter frequent pairs

The effectiveness of this approach is based on how many pairs we can filter with the bit map we create between the scans. For this reason this algorithm is run with different dimensions of the hash table measuring how this parameter affect the dimension of the set of the candidate pairs.

As hash function for the pairs I used a simple function:

$$hash\_fun((a_1,a_2)) = (a_1+a_2)\%hash\_table\_dimension$$

Which is the sum of the integer IDs of the pair module the dimension of the hash table used.

4.3 Distributed approach with SON algorithm

This algorithm is the only one I've implemented which is based on a distributed approach. In this algorithm the input is divided into chunks in order to process them in parallel. Due to the data representation, when creating the partitions there is the possibility that a bucket is split between different partitions. In order to avoid this kind of problems I've used the `partitionBy` function which allows to use a user specific hash function in order to distribute the data between the partitions, focusing this hash function on the title ID.

Then I've implemented the algorithm using the MapReduce model, following the schema suggested in the book [2]:

1. **First Map**: create candidates of frequent pairs for each chunk.
2. **First Reduce**: remove duplicates.
3. **Second Map**: for each basket compute pairs and count their frequency only if they are candidates
4. **Second Reduce**: sum up the frequencies of the same pairs over the chunks and filter out the not frequent ones.

4.4 Toivonen algorithm

The last algorithm implemented in this project is an algorithm based on the sampling technique. Starting from a portion of the dataset, the frequent itemsets are found using the A-Priori algorithm. Then it should be computed the *negative border* as the collection of itemsets that are not frequent in the sample, but all of their immediate subsets are frequent.

In my implementation I decided to build the negative border while doing the last scan, so in this **full** scan of the dataset the false positives are removed and the frequency of each item which belongs to the negative border is computed. If some member of the negative border is frequent, then we cannot be sure that there are not some even larger frequent sets so we give no result and the algorithm should be repeated.

Summarizing the computational overload of this algorithm is based on two scans (A-Priori) on a *smaller* partition plus a single full scan of the dataset.

5 Scale to really big datasets

In order to scale up to bigger datasets there is the possibility to run the A-Priori and PCY algorithms loading only a sample of the data in main memory. There is the *sampled version* of these two algorithms which allows to set two principal parameters:

- p : is the probability that each film has to be taken into account.
- *adjust_ st* (default is 0.9): is the factor of adjustment for the support threshold in order to lower the amount of false negatives.

$$support\_threshold = adjust\_st \cdot p \cdot original\_support\_threshold$$

The steps followed in the sampled version are:

1. Draw a random number for each film and if it is less or equal to p include the title in the sample loading it from memory,
2. Perform the algorithm in order to find frequent pairs on the sample,
3. Given the solution, count the absolute frequencies of the sampled solution loading consecutive partitions of the original dataset in order to remove false positives.

The technique used to remove false positives is specific for an hypothetic case where the full dataset does not fit into main memory. A partition is loaded into main memory in order to count absolute frequencies of the sampled result, then this partition is removed in order to load the next one increasing the count of the absolute frequencies. This process is then repeated till the end of the dataset file. At the end the frequencies we obtained are compared with the original support threshold in order to take only the frequent ones. In an hypothetical case like this one I decided not to deal with the sorting problem, allowing a slight higher amount of false negatives.

6 Experimental results and discussion

6.1 A-Priori

For the full dataset we have a total of 5.706.001 different titles, for the second dataset 522.758 and for the third a total of 393.656 titles. I've started with the smaller dataset analyzing different threshold and once obtained a reasonable result I used a proportion in order to get the support thresholds for the other datasets.

st	avg t_{pairs}	nPairs	avg t_{tripl}	nTriplets
Dataset movies-actors				
20	4.5s	1029	8.3s	214
40	3.9s	175	6.5s	46
60	3.7s	49	5.8s	14
Dataset movies				
28	10.9s	1700	23.7s	399
53	8.9s	287	15.3s	61
80	8.1s	83	12.6s	12
Dataframe full				
290	1min 38s	26693	5min 36s	46606
580	1min 28s	7887	3min 41s	13661
870	1min 19s	3648	2min 53s	6476

Table 1. A-Priori results

In Table 1 is possible to see a weird result: more frequent triplets in the full dataset than frequent pairs. This is caused by the TV Series. In the full dataset there are really long TV Series with a lot of episodes with the same crew. So a really frequent cast (basket) for all the episodes of a TV Series makes more frequent triplets than pairs and this causes this kind of result.

6.2 PCY

Also in this case we have different number of unique actors in the three different data sets. This leads to a different number of possible candidate pairs. So I made different test with different dimensions for the number of buckets we hash to in the first pass for the different data sets. Also here the number of buckets is tested empirically and the same proportion is used into the different data sets.

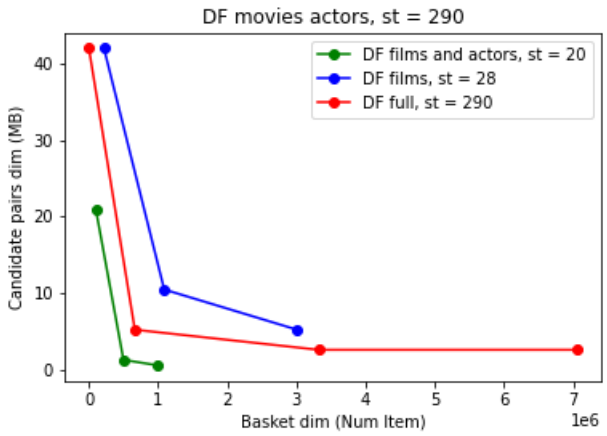


Fig. 1. PCY results

From the graph in Figure 1 we can see that we get a benefit in terms of memory needed for the second pass with a higher number of baskets.

It's also evident the trade off between time and space that we encounter (Figure 3). With a better usage of the memory space we can decrease the number of candidate pairs, but on the other hand the efficiency of the algorithm is performing worse than the A-Priori one.

6.3 SON

The distributed approach is harder to test not having an actual cluster to use, google colab allows me to use a 1 physical core machine. It is possible to see in Figure 3 that this algorithm is not performing as well as the other algorithms that I've implemented.

It is also possible to see in Figure 2 that using more than 4 partitions in the algorithm causes more inefficiency. This is caused by physical limitation that I have in the hardware available. I did not included in the graph the full dataset

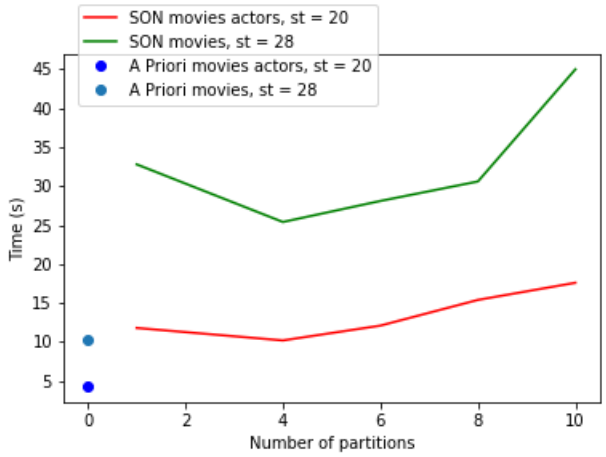


Fig. 2. SON results

experiment which takes in the best scenario (4 partitions) around 5 min 30s, performing as expected from the graph.

6.4 Toivonen

This approach is the only one which makes a single full scan of the dataset in order to give a solution. But on the other hand I've noticed that the partition of data taken into account needs to be rather small, and a small partition often leads to a `no response` solution of the algorithm. In the Table 2 I've reported different tests performed using different thresholds. The parameters that we want to analyze is the average time of the algorithm and the factor *r/t* which represents the number of times the algorithm produced a result over the number of times it is applied.

In the overall time spent for the algorithm it is also taken into account the time used to sample the data which is around 2 seconds for the dataset with only movies and actors, 2.5 seconds for the dataset with only movies and 27 seconds for the full data set.

As it is possible to see in the Table 2, I tried to run this algorithm using relatively high values for thresholds and low values for sample size. With also these adjustments this algorithm still performs worse than the A-Priori. I think this approach would work better with really big dataset where we want to search really frequent items in order to save time and space for computation. As it is possible to see I've set the threshold really high in the full dataset and using a really little partition I can get the result still without a good time performance, but using a really small partition saving up in space.

st	p	adj st	avg time	r/t
Dataset movies-actors				
80	0.6	0.8	9.5s	3/10
100	0.5	0.8	9.8s	5/10
120	0.4	0.8	8.5	1/10
Dataset movies				
100	0.6	0.8	23.4s	7/10
130	0.5	0.8	22.4s	6/10
160	0.4	0.8	21.2	2/10
Dataset full				
900	0.4	0.9	3min 28s	5/10
1200	0.3	0.9	3min 15s	4/10
1400	0.3	0.9	3min 15s	8/10

Table 2. Results for 10 runs of Toivonen

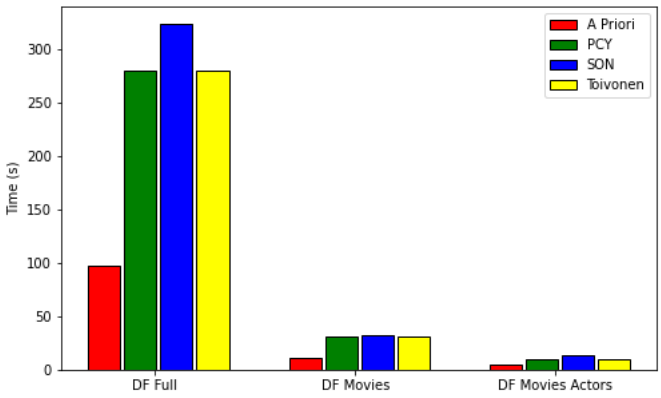


Fig. 3. Time results

References

[1] Imdb dataset published on kaggle under imdb non-commercial licensing. Avaiable at: https://www.kaggle.com/ashirwadsangwan/imdb-dataset.

[2] Jure Leskovec, Anand Rajaraman, J. U. *Mining of Massive Datasets*. Avaible at: `www.http://mmds.org`.