Discovery of Frequent Itemsets using SON Algorithm

Riccardo Conforto Galli, mat 17521A

September 4, 2024

1 Introduction

The aim of this project is the implementation from scratch of a system to find frequent item-sets in a movie database. A frequent item-set is a set of items that is present in at least a chosen number of baskets s, called the support threshold. In the so called "market-basket analysis", frequent-itemsets are used to learn not only the items that are commonly together but also association rules between them. An association rule between an set of items I and an item j is denoted $I \rightarrow j$ and implies that if all the element of I are present in the basket then there's a strong probability that j is also present in the same basket. In this specific scenario each movie is considered as a basket and each actor as an item, therefore the goal is to find groups of actors that frequently star together. In order to find these frequent item-sets I decided to implement the SON algorithm using SPARK

2 Data

The dataset used to perform this analysis is `letterboxd` (2024 version), which is a collection of movies and tv shows obtained from the website `letterboxd.com`. In this dataset are included the following files:

- `actors.csv`
- `countries.csv`
- `crew.csv`
- `genres.csv`
- `languages.csv`
- `movies.csv`
- `releases.csv`
- `studios.csv`
- `themes.csv`

In order to conduct a market-basket analysis the only data of interest are the items (actors) and their baskets (movie) both of which are present in the `actors.csv` file. This is composed by two columns:

- `id`: identifier referring to a movie (`int`)
- `name`: The name of the actor starring in it (`string`)

Each actor is present once for each of their movie and each movie is present once for each actor starring in it.

2.1 Preprocessing

The algorithms that will be used on this dataset assume data to be organized in baskets: one for each movie and containing the respective actors. It is therefore necessary to transform the data grouping based on the movie id and, for each of those groups, obtain the list of all actors.

3 Algorithms

3.1 A-priori

The A-priori algorithm exploits the monotonicity of item-sets, which is the property that if a set is frequent then all of its subsets are also frequent, to reduce the number of tuples to check. With this property is also possible to find a stop condition for the algorithm: if no frequent tuples of size k are found, then there will not be frequent tuples of size k+1. The algorithm requires one pass for each set size and starts with the set containing all the singletons of the items. The frequency of elements of this set, called C_1, is counted in a single pass, the elements present at least s time are frequent and so become part of the set L_1. In general for each set size there are two sets: the set C_k which contains all the candidates of size k and the set L_k which is the set of items of size k that are really frequent. For each step, C_k is the set of the tuples of size k such that each element of the tuple is in L_{k-1}. After this the algorithm counts all the occurrences of the tuples in C_k and if this number is at least s the tuple is added to the set L_k. It's not necessary to generate C_k explicitly, instead for each basket the element not present in L_{k-1} are removed and the tuples generated by the combination of the remaining items are all part of C_k.

3.2 SON

The Idea behind this algorithm is to divide the file in chunks such that each chunk can easily fit in main memory and run on it an algorithm of our choice to find frequent-itemsets (in our case A-priori) adjusting the support threshold to be $s \cdot \frac{\text{chunk size}}{\text{total size}}$. The output will be a set of candidate frequent item-sets for each of the input chunk. If an item-set is not present in these sets then it means that its support in each chunk is less than $s \cdot \frac{\text{chunk size}}{\text{total size}}$ so its total support will be less than $\sum_{n\_chunks} s \cdot \frac{\text{chunk size}}{\text{total size}} = s$ thus the item-set is not frequent. Then in a second pass the algorithm checks the support over the full dataset for the candidates present in at least one of the output sets and if it's lower than s they are discarded. The output are the frequent item-sets. SON lends itself to parallelization and can be implemented using a sequence of map and reduce functions:

- a first map function takes as input the chunk, applies a-priori on it and outputs a set of key value pair (c,1) where c is a frequent item-set
- a first reduce function simply outputs all key that appears at least once

- a second map function takes as input the candidate item-sets and a chunk of the file and outputs a set of key-value pair (c,s') where c is an item-set and s' its support inside the chunk
- a second reduce function sums the partial supports for each key and outputs only the keys with at least support s

4 Implementation

4.1 A-priori

The algorithm takes as input one iterator containing a chunk of the baskets, the support threshold and the number of total baskets. Since for SON the chunk size is chosen to be much less than the available memory it's possible to keep all its content in a list. In the first pass the algorithm counts the occurrence of each actor using a dictionary. At the same time a counter keeps track of the number of basket processed. All actors with support at least $\frac{s \cdot \text{baskets processed}}{\text{total baskets}}$ are put in a frequent singletons list. Counting the number of baskets processed and dividing it for the total basket is necessary since the fraction of basket in each chunk is decided by spark and could be different from one chunk to another. After this step for each size of itemsets k the content of each basket is filtered in order to keep only the element present in the last frequent itemsets and the frequency of each of their possible combination is increased. In the end the next set of frequent itemsets is constructed with all the tuples with support at least $s \cdot$ baskets processed/total baskets.

4.2 SON

In this project has been implement the parallel version of SON, using a distributed dataset (a spark RDD) and applying the map reduce step described above.

```
#FIRST STEP
c = rdd.mapPartitions(lambda x: apriori(x,s,tot))
c = c.reduceByKey(lambda a,b: a).keys().collect()
```

For the chunk subdivision it has been used the `mapPartition()` function, this allows to naturally subdivide the load across the SPARK cluster. `reduceByKey(lambda a,b: a)` is applied in order to merge values for each key by simply keeping one of the two values. After this operation only the keys are collected ignoring the values.

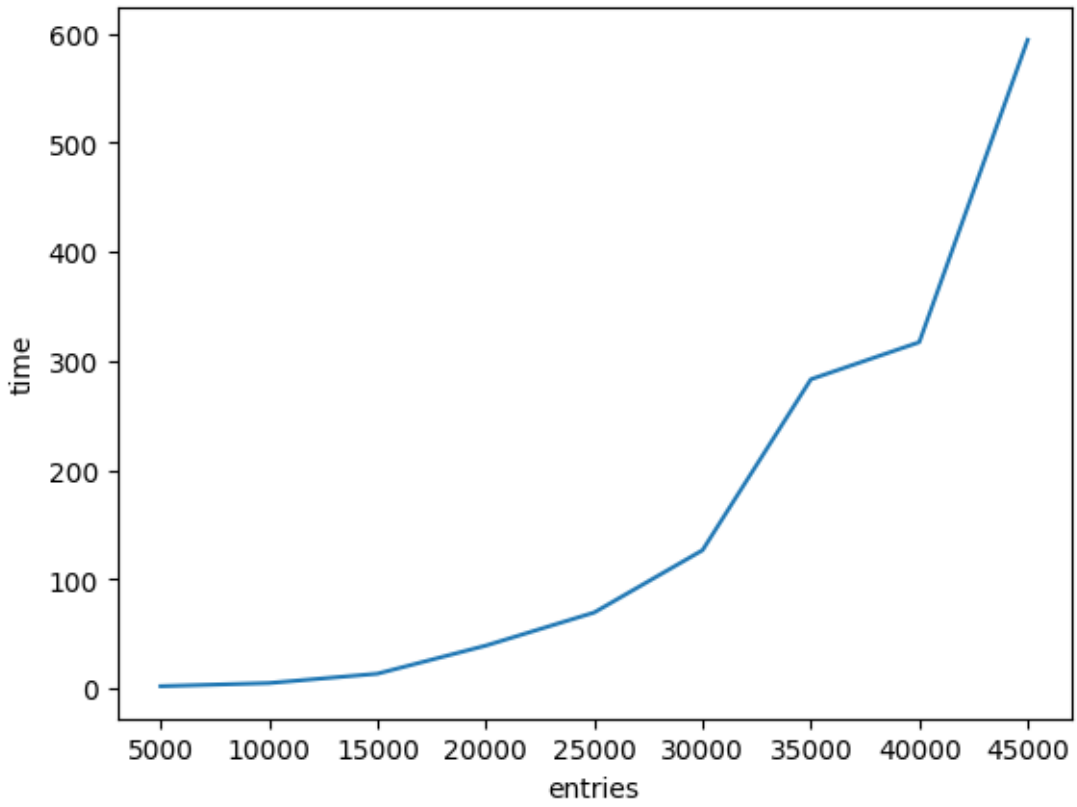
```
#SECOND STEP
out = rdd.mapPartitions(lambda x: chunk_support(x, c))
out = out.reduceByKey(lambda a,b: a+b)
out = out.filter(lambda a: a[1] >= s).keys().collect()
```

For the second step another `mapPartition()` is applied, this time to count the support in each partition of the candidate set. Again the result is reduced using `reduceByKey()` but this time summing the partial supports. A filter function is then applied to eliminate all the elements where the value, which is the support, is lower than s and the filtered keys are collected.

5 Experiments and result

5.1 Execution time as the dataset size increases

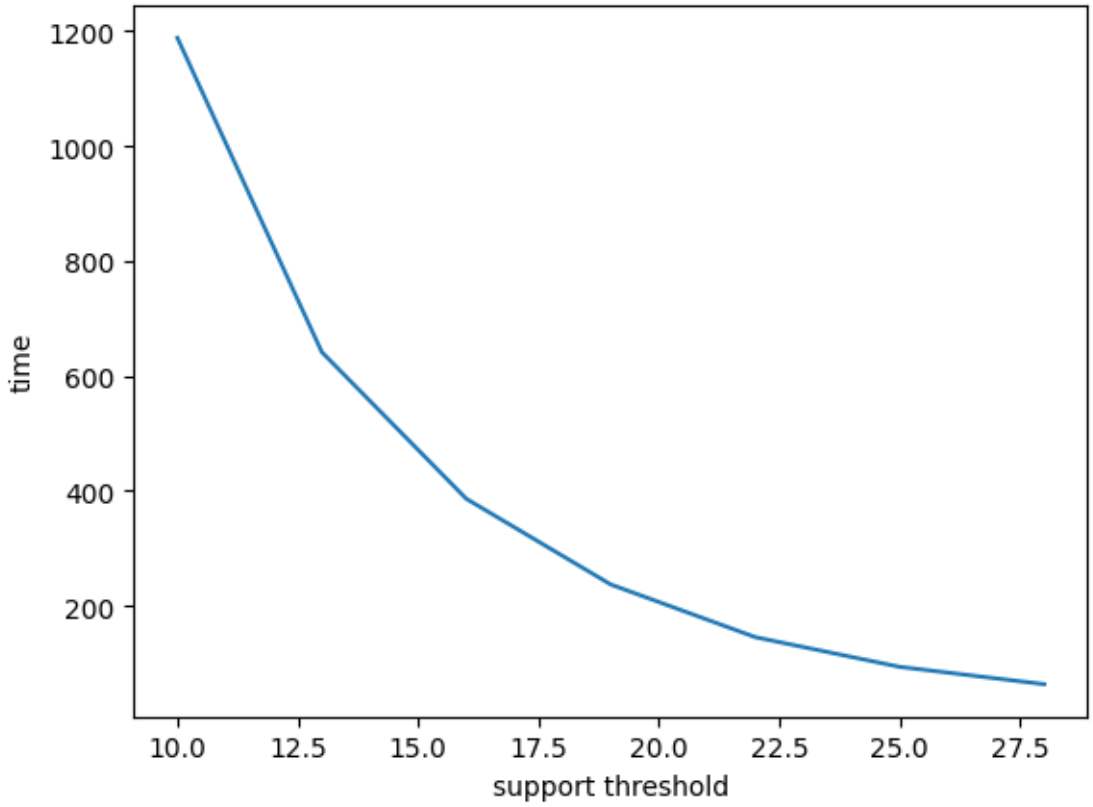
Experiments are run over sorted datasets of varying size from 10.000 to 45.000 while the support threshold remains fixed at 6. As expected more elements implies more possible frequent item-sets, as a result the execution time highly decreases with dataset's size.



tuple size \ elements in dataset	5000	10000	15000	20.000	25.000	30.000	35.000	40.000	45.000
2	-	4	6	11	61	100	198	219	299
3	-	1	1	1	26	32	93	65	65
4	-	-	-	-	8	17	61	25	23
5	-	-	-	-	1	6	28	6	6
6	-	-	-	-	-	1	8	1	1
7	-	-	-	-	-	-	1	-	-

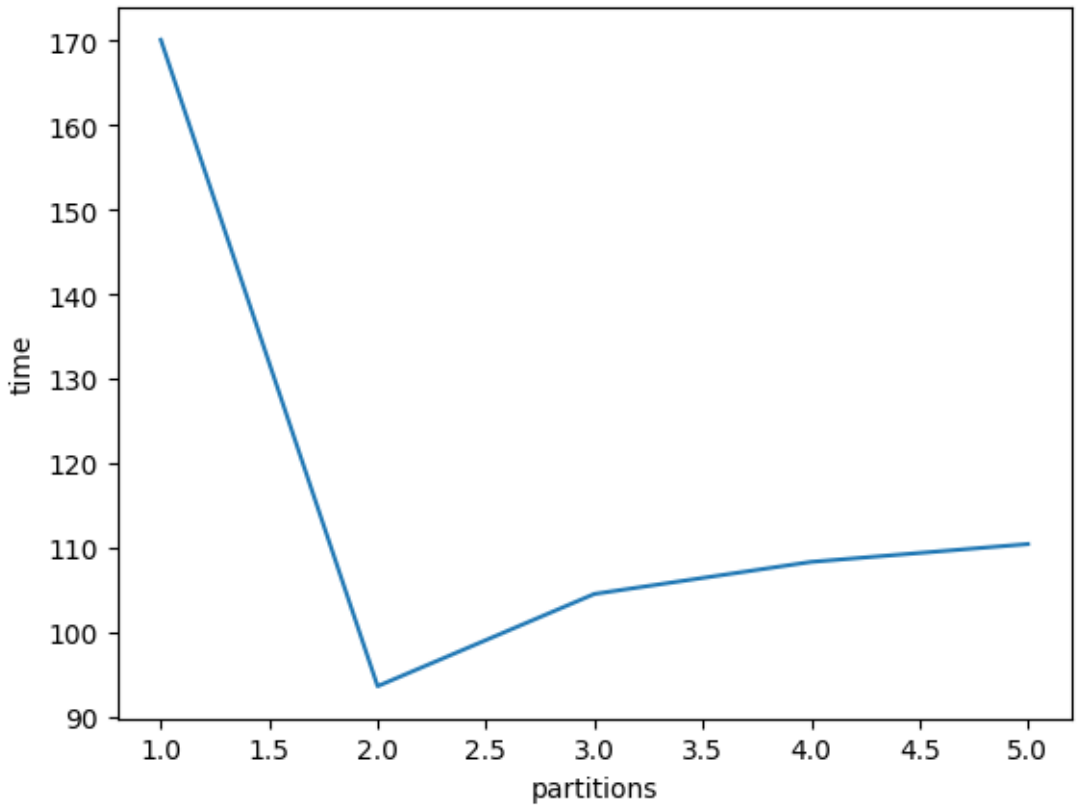
5.2 Execution time as the support threshold increases

In order to observe the variation of time in relation to the support threshold the experiments are run over the same set of 100.000 baskets and the support threshold is ranging from 10 to 29 with steps of 3. As we expect an higher support threshold highly reduces the number of frequent tuples and consequently the number of their combination to check in the next step. As a result the execution time highly decreases.



5.3 Execution time as the partition number increases

Here we analyze the variation of time of execution in relation to the number of Spark partition. Experiments are run over the same set of 99.000 baskets using a support threshold of 25 with 1 to 5 partitions. The result could be surprising at first since the performance of the algorithm not only doesn't get better with more than 2 partitions but instead worsen with each increase. The reason is that this test was executed using a single Spark node on Google Colab with a free account, thus with a CPU with only 2 cores. It is therefore clear how performances get better as long as there are enough cores to take care of new partitions, but instead decrease for the parallelization overhead once their number is bigger than the number of cores. With more nodes we expect a decrease in execution time.


170
160
150
140
130
120
110
100
90
time
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
partitions

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.