UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

Master's Degree in Computer Science

Market Basket Analysis for the IMDB Dataset

Professor: Malchiodi Dario
Course: Algorithms for Massive Datasets

Students:
Facchinetti Nicolas 961648
Belotti Antonio 960822

Academic year 2020/2021

Contents

Chapter 1

Introduction

Disclaimer 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 we 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.

Objective The aim of this project is to implement from scratch some algorithms for finding frequent itemsets, a process better know as market-basket analysis. In particular the chosen techniques to analyze the data have to scale up to larger datasets.

For the development of this work, we used the IMDb dataset [1], realesd under IMDb non-commercial licensing [3]. To perform frequent itemset discovery, we considering movies as baskets and actors as items. The structure and characteristics of the dataset will be described in depth in the next chapter.

Market Basket Analysis Among the principal tasks of feature extraction we have the problem of finding frequent item sets. It's better know as market-basket analysis, since is carried out on data that consists of "baskets" of small items. In particular we are looking for small sets of items that appear together in many baskets, which characterize our data as "frequent itemsets".
The most classical example of this type of analysis is the one applied to true market baskets, like reported in figure 1. We are interested in finding the sets of items, like egg and milk, that people tend to buy together at a supermarket in order to track the customers habits and take profit from this knowledge, for

[1] https://www.kaggle.com/ashirwadsangwan/imdb-dataset

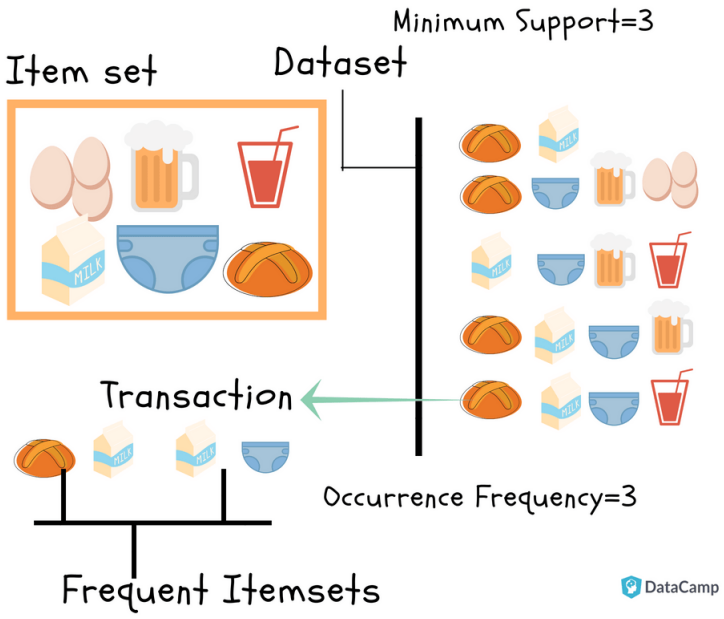

Figure 1.1: An example of market basket analysis applied to true market baskets.

example by better organizing discount campaigns.

The market-basket model of data is used to describe a common form of many-to-many relationship between two kinds of object: *items* and *baskets* (or transactions). Each basket contains a set of items (or itemset) which is usually much smaller than the total number of items. The number of buckets on the other hand is assumed to be very high.

Intuitively, a set of items is assumed to be frequent if it appears in many baskets. We define the *support* of an itemset as the number of baskets that contains this itemset. We consider *frequent itemsets* all the itemsets having support grater than a minimum support value we fix. This lower bound is called *support threshold*.

To carry out our experiments we decided to implement from scratch a distributed version of the famous A-Priori algorithm using a map-reduce framework. We then compared it to the classical sequential version of a A-Priori. We also used A-Priori as a subrutine of our SON algorithm implementation.

Distributed environment Since in the objective of the project is clearly stated that the developed algorithms have to scale up to larger datasets we decided to implement them in a distributed fashion. We decided to use Apache

Spark, a distributed processing framework used for big data workloads which provides an object-oriented library for processing data on clusters using a map-reduce model.

As required, the project is implemented in Python 3. We developed exclusively on a Jupyter Notebook hosted on a Colab environment. We chose this platform so we could focus on the algorithms and experiments and not worry about Spark configurations. However, all Colab instances run on a single node and not on a real clustered environment, so we don't expect to obtain mind-blowing results from the distributed algorithms.
For a real world application it's better to use paid cloud solutions such as Databricks or Amazon Web Services because they can scale on multiple nodes.

Structure of this work In chapter 2 we show the chosen dataset by clearly explaining each of its fields. Next we illustrate all the preprocessing techniques applied to properly rearrange the data for a market-basket analysis.
In chapter 3 we introduce the chosen algorithms to do the analysis and some notes on their implementation.
In chapter 4 we compare the results obtained from each algorithms.
Finally in chapter 5 we derive the conclusions on the work done.

Chapter 2

Data

In this chapter we are going to describe the chosen dataset, the parts of the latter which have been considered, how data have been organized and the applied preprocessing techniques.

Getting the dataset We started by downloading the dataset archive in our Colab environment using the Python Kaggle API [2].
In order to download data from Kaggle, we have to use our personal API credentials. Those are contained in a file *kaggle.json*. After uploading the file we move it where the library expects it and we give it the right permissions flags.

```
$ mkdir -p ~/.kaggle/
$ mv kaggle.json ~/.kaggle/
$ chmod 600 ~/.kaggle/kaggle.json
```

Launching the command *datasets download <owner>/<dataset-name>* will download *dataset-name* of Kaggel's user *owner*. These parameters can be obtained either on Kaggle website or by using the cli command *datasets list*.

```
// downloads the 1.44GB imbd dataset as a zip file.
$ kaggle datasets download 'ashirwadsangwan/imdb-dataset'

// unzip and delete
$ unzip imdb-dataset.zip
$ rm imdb-dataset.zip
```

The unzipped dataset is composed by 5 different tsv [6] files, UTF-8 encoded, plus the same files again but compressed in a gzip [1] archive.

Dataset description The first line of each tsv file is the header, containing the names of all columns. By convention for this dataset, a '\N' denotes a missing or null field.
As mentioned before, the dataset is split in five different files. We will now list the schema for all of them.

1. title.akas.tsv.gz contains the following information for titles:
    - titleId (string) - a tconst, an alphanumeric unique identifier of the title.
    - ordering (integer) – a number to uniquely identify rows for a given titleId.
    - title (string) – the localized title.
    - region (string) - the region for this version of the title.
    - language (string) - the language of the title.
    - types (array) - Enumerated set of attributes for this alternative title. One or more of the following: "alternative", "dvd", "festival", "tv", "video", "working", "original", "imdbDisplay".
    - attributes (array) - Additional terms to describe this alternative title, not enumerated.
    - isOriginalTitle (boolean) – 0: not original title; 1: original title.
2. title.basics.tsv.gz contains the following information for titles:
    - tconst (string) - alphanumeric unique identifier of the title.
    - titleType (string) – the type/format of the title (e.g. movie, short, tvseries, tvepisode, video, etc).
    - primaryTitle (string) – the more popular title / the title used by the filmmakers on promotional materials at the point of release.
    - originalTitle (string) - original title, in the original language.
    - isAdult (boolean) - 0: non-adult title; 1: adult title.
    - startYear (YYYY) – represents the release year of a title. In the case of TV Series, it is the series start year.
    - endYear (YYYY) – TV Series end year. for all other title types.
    - runtimeMinutes – primary runtime of the title, in minutes. genres (string array) – includes up to three genres associated with the title.
3. title.principals.tsv.gz contains the principal cast/crew for titles:
    - tconst (string) - alphanumeric unique identifier of the title.
    - ordering (integer) – a number to uniquely identify rows for a given titleId.
    - nconst (string) - alphanumeric unique identifier of the name/person.
    - category (string) - the category of job that person was in.
    - job (string) - the specific job title if applicable, else. characters (string) - the name of the character played if applicable, else.
4. title.ratings.tsv.gz contains the IMDb rating and votes information for titles:

- tconst (string) - alphanumeric unique identifier of the title.
- averageRating – weighted average of all the individual user ratings.
- numVotes - number of votes the title has received.

5. name.basics.tsv.gz contains the following information for names:
    - nconst (string) - alphanumeric unique identifier of the name/person.
    - primaryName (string)– name by which the person is most often credited.
    - birthYear – in YYYY format.
    - deathYear – in YYYY format if applicable, else .
    - primaryProfession (array of strings)– the top-3 professions of the person.
    - knownForTitles (array of tconsts) – titles the person is known for.

2.1 Preprocessing

The assigned task is to do an analysis considering movies as baskets and actors as items. By inspecting the fields of the shown datasets it's clear we need only a small sub set of the available data

Since the analysis has to be done only on actors/actresses, we will use the dataset *title.principals.tsv*. In particular we are interested in the following columns:

- 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.

We start by directly loading the file *title.principals.tsv* in a Spark DataFrame [4] *df_principals* by using the method *read.csv()* called on a *SparkContext* object, specifying only the needed columns. We then retrieved the distinct values for field *category* of *df_principals*. The result is shown in table 2.1. As we can see there are a lot of different roles involved in the production of a movie. Using this data we can select only the people who have worked as *actor* or *actress*, as indicated in the assignment .
Manipulation of spark DataFrames is done using *pyspark.sql*, a SQL-like api for DataFrames. The number of rows dropped from 36468817 to 14818798.
A sample of the resulting DataFrame is reported in table 2.2.

Having all the information about the actors, we can now extract the information we need about the movies, which can be found in the file *title.basics.tsv*. In particular we are only interested in fields:

- tconst (string) - alphanumeric unique identifier of the title.

Category
actress
producer
writer
composer
director
self
actor
editor
cinematographer
archive_sound
production_designer
archive_footage

Table 2.1: Unique values contained in the field *category* of the dataset *title.principals.tsv*

tconst	nconst	category
tt0000005	nm0443482	actor
tt0000005	nm0653042	actor
tt0000007	nm0179163	actor
tt0000007	nm0183947	actor
tt0000008	nm0653028	actor

Table 2.2: Top 5 rows of the DataFrame *df_principals*

- titleType (string) – the type/format of the title (e.g. movie, short, tvseries, tvepisode, video, etc).

As we did before, we load the file *title.basics.tsv* in a Spark DataFrame *df_basics*. In table 2.3 we show the unique values for field *titleType*. We are only interested in rows labelled as *movie*, so we drop all the rows with a different *titleType*, reducing the number of records from 6321302 to 536034.
A sample of the resulting DataFrame *df_basics* is reported in table 2.4.

We now have two Spark DataFrame objects:

- *df_principals* containing the *nconst* identifiers of people who played as actor/actress in a given *tconst* title,
- *df_basics* containing only the *tconst* identifiers of movies.

In order to perform the desired maket-basket analysis, we have to reorganize our data in a specific manner. We need a table in which each row is composed by the identifier of the title (*tconst*) and the list of identifiers of the actors that played in it (*nconst*).

TitleType
tvSeries
tvMiniSeries
tvMovie
tvEpisode
movie
tvSpecial
video
videoGame
tvShort
short

Table 2.3: Unique values contained in the field *titleType* of the dataset *title.basics.tsv*

tconst	titleType
tt0000009	movie
tt0000147	movie
tt0000335	movie
tt0000502	movie
tt0000574	movie

Table 2.4: Top 5 rows of the DataFrame *df_basics*

The first step to obtain this result is to join *df_principals* with *df_basics* using the unique identifiers of the titles *tconst*. This operation is achieved using the method *join()* of the Spark DataFrame object. We also specified to return only the columns *tconst* and *nconst* since we are no longer interested in the content of *category* and *titleType*. The resulting DataFrame *basket_data* now contains a series of tuples *[tconst, nconst]*, as shown by the sample in table 2.5.

tconst	nconst
tt0000009	nm0183823
tt0000009	nm1309758
tt0000009	nm0063086
tt0000335	nm0675239
tt0000335	nm1010955

Table 2.5: Top 5 rows of *basket_data* obtained after the join operation

We then called method *dropDuplicates()* to remove eventual duplicates followed by a call to method *groupBy()* on column *tconst*.

After this operation *basket_data* contains 393656 buckets, corresponding to

the unique identifier of a title, followed by a list of identifers of the actors that played in it, like shown in table 2.6.

tconst	nconsts
tt0000009	[nm0063086, nm0183823, nm1309758]
tt0000335	[nm1010955, nm1012612, nm1011210, nm1012621, nm0675239, nm0675260]
tt0000502	[nm0215752, nm0252720]
tt0000574	[nm0846887, nm0846894, nm3002376, nm0170118]
tt0000615	[nm3071427, nm0581353, nm0888988, nm0240418, nm0346387, nm0218953]

Table 2.6: Top 5 rows of *basket_data* after grouping the data on *tconst*

As we can see in the above sample we now have a suitable DataFrame to do the desired analysis. The final step is to alphabetic sort the entry in every basket and convert *basket_data* into an RDD [5]. RDDs expose a different API compared to DataFrames, making functional-style map-reduce code easier to write for us. To obtain an RDD from a DataFrame simply access the *.rdd* field on the DataFrame object.

Chapter 3

Algorithms

In this chapter we describe the algorithms we used for our experiments: Apriori and SON. We recall how they work, their implementation and how they handle (or do not handle) a big dataset.

3.1 A-priori algorithm

The a-priori algorithm is the simplest frequent itemsets discovery algorithm in data mining. To reduce the itemsets feasible space it leverages on the monotonicity of the apriori property of frequent itemsets: *if a set I of items is frequent, then so is every subset of I.*
By considering the negation of this predicate, if a set is infrequent also any of its supersets will be an infrequent and can therefore be excluded *a priori*. The support of an itemset is the number of occurrences of that itemset divided by the total number of transactions. An itemset is frequent if its support is grater then a fixed support threshold we fix by the start. Figure 3.1 shows how this property allows the reduction of the search space.

Algorithm Notation remarks:

1. I it the itemset
2. $T = \{t \subseteq I\}$ is the set of all transactions
3. C_k is the set of *candidate* frequent itemsets of size k
4. L_k is the set of *true* frequent itemsets of size k
5. $support(e) = \frac{\text{number of occurrences of e in T}}{|T|}$ where itemset $e \subseteq I$

The algorithm is iterative, for a k ranging from 1 to *maximum size of frequent itemsets*: it scans the database for frequent itemsets of size k by analyzing sets which are a combinations of frequent itemsets of size $k-1$. If no frequent

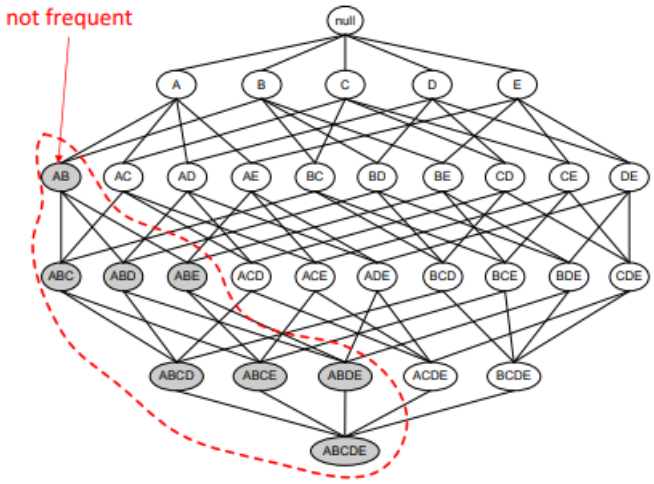

Figure 3.1: An example of the antimonotonicity of the apriori property. The indicated node in the lattice is a not frequent itemset and therefore also its supersets are marked as infrequent.

itemsets of size k are found then monotonicity tells us there can be no larger frequent itemsets and so we can stop. Figure 12 shows the steps of the algorithm while algorithm 1 the pseudocode.

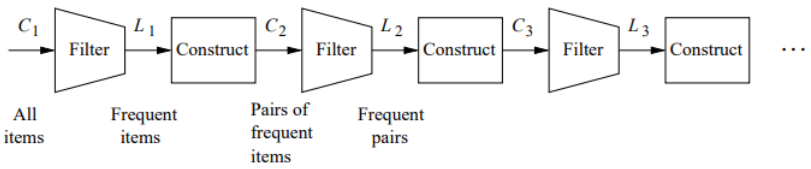

Figure 3.2: Various steps of the Apriori algorithm

Considerations The apriori algorithm is simple is effective; however the performances will not be good enough on a sufficiently large dataset.
The biggest problem is the memory footprint: even counting the most frequent singletons can be problematic to do on a single machine. Generating the candidate sets C_k can be even more costly: the process can quickly lead to memory trashing. To reduce this problem, it is possible to avoid generating all the candidate itemset explicitly by using the monotonicity property. Testing for membership in C_k is done by testing whether all the members of size $k-1$ of a k-tuple are frequent in L_1.

Algorithm 1: Apriori

```
1  k = 1
2  C_1 = I
3  L_1 = {e ∈ C_1 : support(e) ≥ support_threshold}
4  while L_k ≠ ∅ do
5  |  k = k + 1
6  |  C_k = L_{k-1} ⋈ L_{k-1}
7  |  for each transaction t do
8  |  |  increment by one the count of all candidates C_k that appear in t
9  |  end
10 |  L_k = {e ∈ C_k : support(e) ≥ support_threshold}
11 end
12 return all L_k
```

Some apriori variants have been proposed to utilize the memory in a smarter way, like PCY. The core idea is the same, but PCY uses the free memory available during the first pass to store a hash table. As we examine each basket we generate all its pairs of items and add 1 to the bucket in which they are hashed. Then we keep only the buckets with support greater than the threshold. This information is compressed in a bitmap (to save space) and used to have an advantage in the second pass, since a candidate i, j must have both i and j as frequent items but also i, j hashed to a frequent bucket.
Note that the Apriori algorithm needs an entire pass of the datasets for each k. There are some random based algorithm which return an exact answer with only two complete pass. However the base version of the algorithm is still practical for small instances and may be used as a sub procedure for this types of algorithms, like SON which is presented later.

Implementation In this project we implemented two versions for the Apriori algorithm: the first is the basic flavor while the latter is implemented using a map-reduce framework, as an experimental exercise. In both implementations we stop at finding itemsets until the set C_k is empty.

The basic implementation of apriori is straight forward. There are only a couple details we would like to point out.
Firstly, we used a hash-map to store the count of the itemsets instead of an array of integers, therefore removing the need to map every element to an integer identifier. The algorithm starts by counting, for each bucket, the occurrences of every item in the itemset and then stores the value in an dictionary by using the item itself as key. Then we extract the singleton frequent itemsets from this dictionary by taking the elements that have support greater than threshold.
Secondly, we explicitly generate the candidate pairs by checking to not make duplicates. We use again a dictionary to count the occurrences of the tuples in the buckets and after we filter out those that have support lower than thresh-

old. The same process is repeated until the candidate itemset is empty; the algorithm then returns the union of all the frequent items.

Regarding the map-reduce implementation of apriori, it is important to note that there is a costly broadcast operation after the first pass. This operation is reasonable as long as the number of frequent singletons is relatively small. The number of nodes and the bandwidth between them is also a limiting factor. We therefore know that this implementation is not applicable for a very massive dataset, even if we cannot quantify what massive is.
The computation is carried out in several steps, one for each candidate set C_k. The first map operation takes all the buckets in the RDD and with *flatMap()* returns all the items in the buckets. We apply a map operation to the elements returning the key-value pair $(I, 1)$, where I is an item and 1 to take note of its occurrence. Next we apply a reduce operation on the results obtained, by summing up the values of the same keys to get the support of the items in the dataset. We then filter out the elements having support lower than the threshold.
The next iterations starts by computing the candidate doubletons from the frequent singletons. The *flatMap()* operation returns for each bucket the itemsets which are in the candidate doubletons. Next, a *map()* operation takes all those itemsets and maps them to $(D, 1)$, where D is the itemset and 1 to take note of its occurrence. In the end, as done for the first step, we apply a reduce operation to sum up the values of the keys to obtain the support of the itemsets and we keep the element which have count greater than the threshold. The same process is repeated until the filter step returns no more frequent itemsets; the algorithm then returns the union of all frequent itemsets.

3.2 SON algorithm

The Apriori algorithm needs one whole pass of the dataset for each size of the frequent itemsets we want to extract. If the dataset is very big we can encounter the problem of not being able to store in main memory both the data and the itemsets count. The idea to tackle this problem is not to use the entire file of baskets, but instead work on a smaller portion of it, a chunk. Is important to shuffle the dataset (or to pick the baskets for the chunk with a certain probability) to have a sample which is uniformly distributed in the file.

Savasere, Omiecinski, and Navathe proposed an algorithm which uses only two passes of the dataset, gets the exact answer and lends itself very well to a distributed implementation. The SON algorithm (named with the initials of the authors) works by dividing the dataset in multiple chunks (both in the sense of a chunk in a distributed environment or a piece of file), finding frequent itemsets local to each chunk using a known algorithm, merging the results obtaining a candidate itemset and afterwards removing false positives with a second pass of the dataset.

Algorithm As mentioned above we can imagine the algorithm as a two-step processing of data; the idea behind the algorithm is to work with small portions of the dataset.

In the first step we treat each chunk as a normal dataset. We apply to each chunk a known algorithm for finding frequent itemsets, like Apriori or one of its variant. In this way we find all the frequent itemsets in the given chunk. However we must pay attention to the support threshold: if s is the threshold defined on the entire dataset we must reduce it to $p * s$, where p is the fraction of the dataset in the chunk. Once all the chunks are processed, we merge all the chunk-frequent itemsets obtained to produce the *candidate itemsets*.

In the second phase, we work on these candidate itemsets. The candidate itemsets may contain some false positives, i.e. itemsets frequent in a chunk but not in the whole dataset. To remove them is compulsory to do a second pass on the entire dataset, by counting the occurrences of all the elements in the candidate itemsets and discard the entries that have a support lower than the support threshold.

The SON algorithm is well suited for a parallel implementation because each chunk in the first step can be processed in parallel. We divide the dataset in chunks and assign every single one of them to a computing unit, which will find the local frequent itemsets. After combining the results of the first step we can again distribute the found candidate itemsets to many processors, each working on a portion of the data. Now each processor counts the support of each candidate itemset in its partition of data. Finally we can sum up the results obtained by all the computing units and filter out the itemsets with support lower than the threshold.
In this way the algorithm has a natural way of expressing the two steps as two map-reduce operations, summarized in figure 3.2. In particular, the four

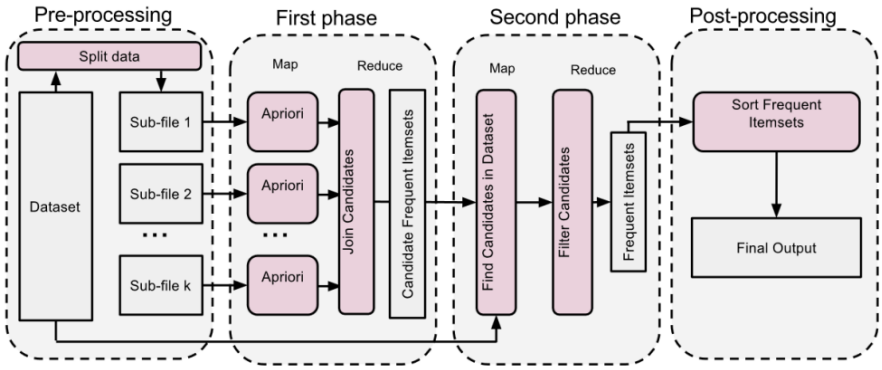

Figure 3.3: Two map-reduce operations for the SON algorithm

functions do the following operations:

1. **Fist map function** Take the assigned chunk of baskets and find the frequent itemsets using an algorithm like apriori (remember to lower the support threshold of the local search on the chunk). Then map each frequent itemset F to $(F, 1)$, where value 1 is irrelevant and only placed to be able to reduce the keys.

2. **Fist reduce function** Each reduce task takes a set of keys (itemsets) and reduce them to produce the candidate itemset. The value can be ignored since is irrelevant to know in how many partitions an itemset is frequent.

3. **Second map function** The second map task takes all the output of the first reduce (candidate itemset) and again a portion of the baskets. Each map task then count the number of every single element in the candidate itemset in the assigned portion of data, producing a set of key-values pairs in the form (C, v) where C is one itemset and v its support in the assigned basket.

4. **Second reduce function** Finally the reduce function take all the key-value pairs of the above step and sum up the values for all the same keys, obtaining the total support of the itemsets in the baskets. Those itemsets whose value is greater than the support threshold are the ones which are frequent in the dataset and they are returned along with their support.

Considerations In this context we refer to *false positive* and *false negative* according to the following definitions:

Definition 1. ***False positive:*** *an itemset that is frequent in the chunk but not the whole.*

Definition 2. ***False negative:*** *an itemset that is frequent in the whole but not in the chunk.*

The false positive are removed from the candidate itemset obtained after the first phase by counting their occurrence in the dataset and checking that their count is above the support threshold.
SON is also not affected by false negatives. If an itemset is not frequent in every chunk, its support is less than s/p in every chunk. Therefore the total support j of this itemset is $j < \sum_{n=1}^{p} s/p = s$ hence no false negative is possible.
Since false negatives are not possible and false positives are removed by the algorithm, SON is a correct algorithm. The result of the computation of SON using Apriori as a sub procedure is equal to the result of Apriori by itself.

Implementation We implemented the algorithm following the above presented map-reduce approach.

We find the sweet-sport number of partitions for the data considering how many machines are available in the cluster and multiply this number by 4 assuming that each executor has 4 cores; then we re-partition the RDD of our basket data according to this number. Next we find the reduced support for the first step by dividing the global support by the number of partitions and round it to the lower integer.

In the first map step we decided to utilize the classic implementation of Apriori presented in the last chapter. In particular we use the method *mapPartitions()* of pyspark, which applies a function to each partition of the RDD on which is called. The result of this computation is the execution of Apriori on each partition independently from the others, finding the local frequent itemsets in them. Next we apply a map function to all the itemset to return the key-value pair $(C, 1)$, where C is the candidate itemset. The first reduce operation then simply take all this tuples and returns all candidate itemsets without repetitions.

Next we take all the key-value pairs and put only the keys in a Python dictionary using the key itself as value for the hash function and as value the boolean *True*, indicating that the itemset hashed is a frequent itemset. This is done to be able in future to check in constant time if an item in a bucket is in the frequent itemset without need to examine all the elements in the candidate frequent itemset. The dictionary is then broadcasted to all the workers nodes to reduce the amount of network transfer for the next computation.

In the second map operations we use again *mapPartions()*, but instead in this step we call a function which has as arguments the partition of basket data assigned to the node and the above introduced dictionary. This function, for each candidate itemset i, checks for each bucket b in the chunk whether i is a subset of the itemset of b. In case this is true we increase the value of a counter initialized to 0 and once all the buckets are examined we return a tuple consisting of the itemset checked and the value of the counter.
Once we have examined all the chunks we can pass to the second reduce function, which sum up the value of each itemset i with same key. In this way is obtained the global support of i and we filter out the key-value pairs with support lower than the original threshold value. The remaining ones are the real frequent itemsets in the baskets.

Chapter 4

Experiments

In this chapter we show the results obtained on the dataset elaborated as in chapter 2 with the three algorithms proposed in chapter 3, along with some considerations on the chosen support thresholds and time executions.

Correctness of the algorithms We start our experimental analysis by ensuring that the three developed algorithms are correct with a toy dataset. We extracted only 1000 rows from the elaborated datasets and compared the results with the Spark's implementation of FP-Growth, using as support threshold 1% of the considered buckets. All the four algorithms return the same itemsets along the same supports.

Support threshold Considering that a good rule of thumb is to start the analysis with 1% as support threshold and that the dataset has a total of 393656 buckets, the task is to find (taking in account only singletons) actors who played in at least 3936 movie. This number is intuitively to high and indeed all the algorithms return no frequent itemsets.
From our experiments, the maximum threshold for which the algorithms return

% of buckets	Support	% of buckets	Support
0.05	197	0.12	473
0.06	237	0.14	552
0.07	276	0.16	630
0.08	315	0.18	709
0.09	355	0.2	788
0.1	394		

Table 4.1: % of the buckets for the threshold support along with the corresponding transactions as support(round to the next integer)

at least one item is 0.2%: such value corresponds to at least 788 movies. We

run the algorithms with different support threshold values and inspected the number of returned itemsets.
As shown in table 4.1 we started with 0.2% and then decrease the values with a step of 0.02 to 0.1%. Then we decided to reduce the step to 0.1% until 0.05% to better represent how many itemsets are found. In the table, in addition to each percentage of the buckets considered as support threshold, we show the corresponding numeric minimum support threshold.

Returned itemsets As can be seen from table 4.2, all the algorithms returned the same number of itemsets for each threshold value. It also shows in detail the number of singletons, doubletons and triplets found as frequent.
As can be seen no triplets are found and the only doubleton is discovered with

Support	Apriori	Apriori MR	SON	Singletons	Doubletons	Triplets
0.05	90	90	90	89	1	0
0.06	49	49	49	48	1	0
0.07	28	28	28	28	0	0
0.08	15	15	15	15	0	0
0.09	10	10	10	10	0	0
0.1	6	6	6	6	0	0
0.12	4	4	4	4	0	0
0.14	3	3	3	3	0	0
0.16	1	1	1	1	0	0
0.18	1	1	1	1	0	0
0.2	1	1	1	1	0	0

Table 4.2: On the left side, number of frequent itemsets found by support threshold value for the three algorithms. On the right side, the same number divided by itemset length

a support of at most 0.6%.
In figure 4 is reported the stacked bar chart of itemset counts for the various support levels.

Execution times In table 4 we show the measured executions times of the algorithms.

At a first sight is clear how the execution time increases as the support is lowered. This is due to the larger candidate set that must be generated and checked at every iteration.
One thing that pops out is the worse performance of distributed algorithms compared with sequential algorithm. This was expected, as we did our experiment using Colab and not a real distributed environment and the overhead introduced by Spark is considerable. However, we want to point out that the difference between Apriori implementation narrows as the threshold is lowered. In particular with support 0.05% the map-reduce implementation of Apriori

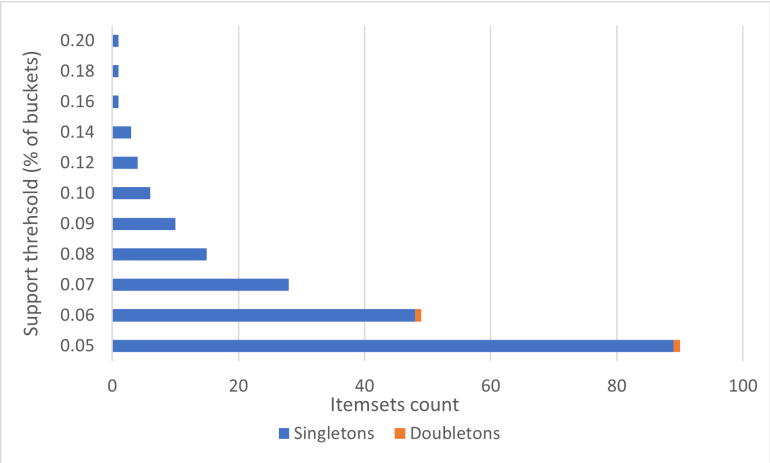

Figure 4.1: Stacked bar chart of itemsets count for the various support levels. In blue are reported singletons, in orange doubletons.

% of buckets	Apriori	Apriori MR	SON HM
0.05	981.462990	969.568024	1097.482405
0.06	281.688775	286.202729	344.297078
0.07	95.344404	101.994837	135.184267
0.08	27.547581	35.582046	47.452804
0.09	12.570376	21.252565	20.639722
0.10	5.134855	13.453004	11.166369
0.12	2.751391	11.393425	7.783163
0.14	2.037042	10.721066	5.835834
0.16	1.272782	10.475100	5.315095
0.18	1.216546	9.902270	4.742669
0.20	1.302487	10.021605	4.629177

Table 4.3: Execution times in seconds by support threshold values

performs better than the sequential one, so maybe the trend is inverted for even lower value of the threshold.
Note that SON performs better than map-reduce Apriori for greater value of the threshold, while performs worse when the support is lowered. This is probably due to the fact that the threshold itself is divided among the worker nodes (considering 0.05% of the buckets each worker has a support of 50 transactions) and this leads to find a much larger number of candidates.

To better represent the situation, figure 4.2 and 4.3 report the execution time of the implemented algorithms with respect to the support threshold, which is converted on a logarithmic scale to better visualize the data, respectively on a

histogram and on a line chart.

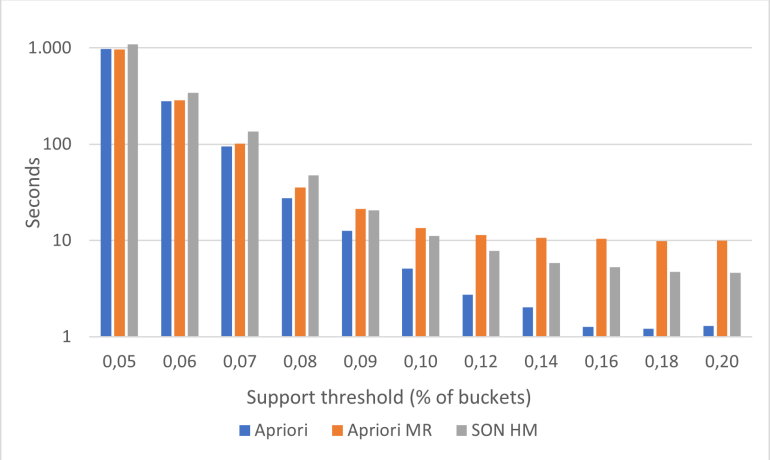

Figure 4.2: Histogram of the execution times for each support value.

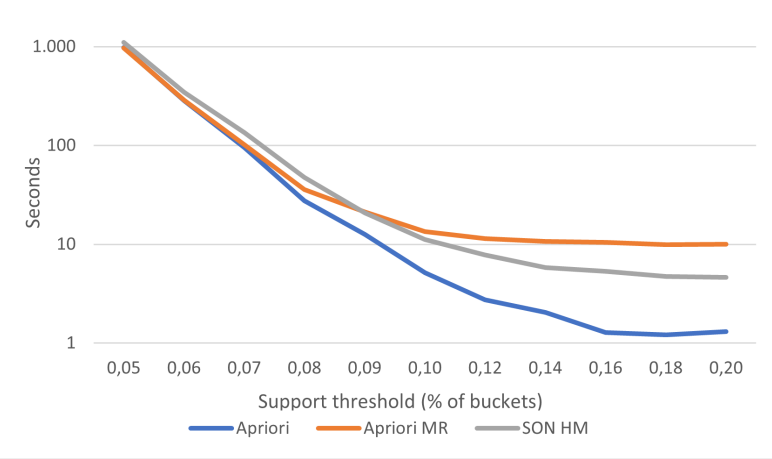

Figure 4.3: Line chart of the execution times for each support value.

Chapter 5

Conclusions

The aim of this project is to do a market basket analysis on the Kaggle's IMDb dataset and implement from scratch some solutions, which can also scale up to larger datasets. In particular we chose to use two algorithms: Apriori, of which we propose a sequential and distributed version, and SON, that lends itself well to a map reduce implementation. We started by describing the general task of finding frequent itemsets and then briefly explaining the development environment we used.

We then described the proposed dataset and clearly show the preprocessing phases in order to have movies as baskets and actors as items.

The first algorithm considered to solve the problem is Apriori. We gave a brief description of the method along with some considerations on the performances and the implementation details. We first focused on the basic sequential version and then we moved on to the distributed approach using map-reduce. We then presented the SON algorithm with its correctness and implementation notes. The latter, along with the distributed implementation of Apriori, represent a good solution to scale up also to very large dataset by being capable of running on a cluster of machines.
One possible limitation of our implementation is the broadcast we do of the candidate frequent itemsets. Broadcasting a large set of items has a network cost and could occupy too much memory on worker nodes. Creating the broadcast variable itself could be unfeasible because it requires a collect of the candidates and could fill up the memory of the master node.

From the conducted experiment the runtime of the algorithms increases as the threshold goes down. This behaviour can be expected, since a lower support means more returned itemset, as well as more candidate set to build and check at each iteration.
The algorithms implemented with Spark have performances comparable to the one of the sequential implementation of Apriori. This was expected, as we used

the free Colab environment running on a single node and not on a real cluster. From the conducted experiments, for greater values of threshold sequential Apriori return the best temporal results, probably due to the overhead introduced by Spark. However by lowering the support the difference with the distributed versions narrows: in particular with the lower tested support apriori map-reduce is the best performing algorithm.
Instead comparing the two distributed algorithms we can see something of interesting: for greater thresholds SON outperforms Apriori map-reduce but about at 0.09% the trend is reversed.

5.1 Further developments

A possible improvement of the work is try to implicitly check membership in the candidate itemset C_k during iterations of Apriori, instead of explicitly generating it from the set L_{k-1} to save some memory space.

Furhter development can be done on the distributed version of the two algorithms, by testing them on real Spark environment running on clusters to take advantage of more parallelism in computation.

Also investigating even lower support threshold values would be interesting, since from the conducted experiments with a support equal to the 0.05% of the buckets the trend of dominance of sequential Apriori is inverted and outperformed by the same algorithm but implemented with map-reduce.

Bibliography

[1] *GNU Gzip*. https://www.gnu.org/software/gzip/.

[2] Inc. IMDb.com. *How to Use Kaggle*. https://www.kaggle.com/docs/api.

[3] Inc. IMDb.com. *IMDb Conditions of Use*. https://www.imdb.com/conditions?pf_rd_m=A2FGELUUNOQJNL&pf_rd_p=3aefe545-f8d3-4562-976a-e5eb47d1bb18&pf_rd_r=K7VGSA5BY26HTH7KAGZV&pf_rd_s=center-1&pf_rd_t=60601&pf_rd_i=interfaces&ref_=fea_mn_lk2.

[4] *pyspark.sql.DataFrame*. https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.DataFrame.html.

[5] *RDD Programming Guide*. https://spark.apache.org/docs/latest/rdd-programming-guide.html.

[6] *TSV, Tab-Separated Values*. https://www.loc.gov/preservation/digital/formats/fdd/fdd000533.shtml.