Market Basket Analysis on IMDb Dataset

Ivan Lamperti
Università degli Studi di Milano
Email: ivan.lamperti@studenti.unimi.it
Registration Number: 960494

Jaspreet Kaur
Università degli Studi di Milano
Email: jaspreet.kaur@studenti.unimi.it
Registration Number: 980691

1. Introduction

Market basket analysis is a data mining technique used to describe a form of relationship between two types of objects, that is, items and baskets: each item is a single object and each basket consists of a set of items tied together by a certain relationship. In particular, the aim of market basket analysis is to find itemsets that tend to appear frequently in baskets. More formally, an itemset I is considered frequent if its support, the number of baskets in which it appears, is at least a specific value, fixed at the beginning, called support threshold.

This technique can be applied in various fields, such as marketing and bioinformatics. For example, in the former, items are single purchasable products and baskets are sets of items purchased together: market basket analysis is used to understand the behaviour of the buyers, by identifying products that are often bought together. This information can be used in order to perform recommendations, offer coupons, organize the products in the store, etc.

The problem of finding frequent itemsets falls into the domain of big data, because, although the number of items considered can be manageable, the number of itemsets is approximately $\frac{n^k}{k!}$, where n is the number of items and k is the size of itemsets considered.

The Apriori algorithm and its variants, the SON algorithm and the Toivonen algorithm have been proposed to perform market basket analysis. These are the algorithms that have been implemented in our project. This paper will start by describing the dataset on which we have worked and will continue by explaining the algorithms applied and our implementations of them. Finally, we will evaluate our solution and compare the algorithms applied.

2. IMDb dataset

The dataset used is the IMDb dataset published on Kaggle. It is formed by five datasets, each of them contained in a gzipped, TSV-formatted file in the UTF-8 character set. The datasets can be described as follows:

1) title.akas.tsv.gz: contains the following information about titles:
   - titleId (string): alphanumeric unique identifier of the title.
   - ordering (integer): numeric unique identifier of the row for a given titleId.
   - title (string): title name.
   - region (string): region for the current version of the title.
   - language (string): language of the title.
   - types (array): enumerated set of attributes for the title; it can be "alternative", "dvd", "festival", "tv", "video", "working", "original", "imdbDisplay".
   - attributes (array): additional terms to describe the title.
   - isOriginalTitle (boolean): has value of 0 if it is not an original title, 1 if it is.
2) title.basics.tsv.gz: contains the following additional information about titles:
   - tconst (string): alphanumeric unique identifier of the title.
   - titleType (string): type/format of the title (e.g. movie, short, TV series, TV episode, video, etc).
   - primaryTitle (string): the most popular title.
   - originalTitle (string): original title, in the original language.
   - isAdult (boolean): has value of 0 if it is a non-adult title, 1 if it is an adult title.
   - startYear (YYYY): release year of a title.
   - endYear (YYYY): TV series end year.
   - runtimeMinutes: primary runtime of the title, in minutes.
   - genres (string array): genres associated with the title.
3) title.principals.tsv.gz: contains the principal cast/crew for titles described by:
   - tconst (string): alphanumeric unique identifier of the title.
   - ordering (integer): numeric unique identifier for rows for a given titleId.
   - nconst (string): alphanumeric unique identifier of the name/person.

- category (string): category of job that the person was in.
- job (string): specific job title if applicable.
- characters (string): name of the character played if applicable.

4) title.ratings.tsv.gz: contains the IMDb rating and votes information for titles described by:
    - 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 about names:
    - nconst (string): alphanumeric unique identifier of the name/person.
    - primaryName (string): name by which the person is most often credited.
    - birthYear (YYYY)
    - deathYear (YYYY)
    - primaryProfession (array of strings): top-3 professions of the person.
    - knownForTitles (array of tconsts): titles the person is known for.

In our project, items are actors and baskets are movies. The aim of market basket analysis applied on this dataset is finding groups of actors that have frequently played in movies together. Movies are taken from the dataset in *title.basics.tsv.gz* by filtering on the attribute *titleType*, that has to be equal to "movie", whereas actors are taken from the dataset in *title.principals.tsv.gz* by filtering on the attribute *category*, that has to be equal to "actor" or "actress". Actors are joined with movies on the attribute *tconst* and the result is joined to the dataset in *name.basics.tsv.gz* on the attribute *nconst*. From the result, we select only the attributes *tconst*, that identifies the movies, *nconst*, that identifies the actors, *primaryName*, that represents the name of actors. The last attribute is used only to print frequent itemsets in an easily readable way.

3. Implementation of algorithms

Each algorithm has been implemented like an Iterator on State, where State is a dictionary that contains all the information necessary for one run of the algorithm, for example the initial dataset, the size of itemsets, the support threshold and the frequent itemsets of the previous run. Each time the size of the frequent itemsets needed is increased, another iteration of the algorithm is executed by calling the method *next*.

The initial dataset has been organized in two ways: the first one is a dataframe cointaining the extracted information, as described in section 2; the second one is a CSV file, where each row is in the form of *tconst, nconst1* | *nconst2* | ..., by associating each movie to the list of actors that have played in it, separated by a pipe.

In the following subsections, we describe in more detail the implementation of each algorithm.

3.1. Apriori algorithm

The Apriori algorithm is able to find frequent itemsets of size k by performing k passes over the baskets' file, i.e., one pass is performed to find frequent singletons, another one is performed to find frequent pairs and so on. The purpose is to reduce the number of itemsets that have to be counted and, as a consequence, the number of counters that have to be memorized. As said above, the number of possible itemsets is very big, namely $\frac{n^k}{k!}$, where n is the number of items and k is the size of itemsets considered; therefore, memorizing one counter for each possible itemset could consume all the main memory available and each increase of each counter could require loading a page from the disk. As a result, the algorithm would be very slow and very inefficient. The Apriori algorithm solves this problem, by discarding *a priori* counters of itemsets that will certainly be non-frequent, memorizing only the counters of itemsets that are likely to be frequent. In order to reach this goal, one pass for each cardinality of the itemsets has to be performed and during each new pass the monotonicity property has to be applied.

Definition 3.1 (Monotonicity property)**.** If a set I of items is frequent, then each subset of I is itself frequent.

Suppose we want to find frequent itemsets of size k. The algorithm starts by finding frequent singletons. It creates an array of counts, whose i-th element counts the occurrences of the i-th item, and performs the first pass on the baskets' file: as it reads baskets, it looks at each item in each basket and increments its counter in the array of counts. At the end of the pass, it compares the counters with the support threshold: the i-th entry of the array of count is set to 0 if the i-th item is not frequent, otherwise it is set to an unique integer from 1 to the total number of frequent singletons.

In order to find frequent pairs, the algorithm performs a second pass on the baskets' file and it generates all the pairs for each basket. Each pair is analyzed: if at least one of the singletons that form the pair is not frequent, so it has its counter set to 0, the pair is not processed further, otherwise the pair is considered a candidate pair and its counter is incremented. This reasoning is derived directly form the monotonicity property [3.1]: if at least one of the singletons that form a pair is not frequent, the pair will definitely not be frequent. At the end of the pass, it compares the pairs' counters with the support threshold: if the counter of a pair exceeds the support threshold, the pair is frequent, otherwise it is not.

In order to find frequent triples, the algorithms performs a third pass on the baskets' file and the same reasoning described above is applied: if the triple is formed by frequent pairs, its occurrences are counted, otherwise the triple is automatically discarded. At the end, the counters are compared with the support threshold, in order to get frequent triples. The computation proceeds similarly until it reaches frequent itemsets of size k, as wanted.

In our project, we have implemented two versions of the Apriori algorithm. The first version is executed inside the distributed environment offered by Spark and it is formed by three functions:

- *get_ck*: it works on the dataframe that contains the data described in section 2 and it returns a new dataframe which contains all the possible candidate itemsets of size k, formed by the frequent itemsets of size k-1 found in the previous run.
- *get_lk*: it works on the candidate itemsets given by *get_ck* and it counts all the occurrences of each candidate itemset. It returns a dataframe, with only the candidate itemsets whose count is at least the support threshold, so the frequent itemsets of size k.
- *apriori_algorithm*: it is an Iterator on State, that calls the function *get_lk* on the current State and at the end it updates the State. This function starts the algorithm itself.

The second version is executed in memory and it works on the CSV version of the baskets' file. The implementation works on three functions:

- *get_ck*: it scans the baskets' file and forms all the combinations of dimension k of actors. If the monotonicity filter is satisfied, the counter of the combination is increased.
- *get_lk*: it calls the previous function *get_ck* in order to get candidate itemsets of dimension k. For each of the candidate itemsets, it compares its support with the support threshold. It returns the frequent itemsets, whose support is at least the support threshold.
- *apriori_algorithm*: it is an Iterator on State, that calls the function *get_lk* on the current State and at the end it updates the State. This function starts the algorithm itself.

3.1.1. PCY with multihash algorithm.

PCY is a variant of the Apriori algorithm. It exploits the memory left free after one pass of the Apriori algorithm, in order to form another filter on the possible candidate itemsets, that can be used in combination of the monotonicity filter. In this way, the number of possible candidate itemsets is decreased and so is the number of counters that have to be maintained in main memory, hence obtaining a general decrease of the memory usage. However, the memory used to maintain the additional filter has to be considered.

The additional filter is formed by a hash function, that hashes all the possible candidate itemsets of higher cardinality than the cardinality of the frequent itemsets that the algorithm is creating in the current pass. During the aforementioned pass, perfomed to count the candidate itemsets of size k, each possible candidate itemset of size k+1 is hashed to an integer number, a bucket, where the total number of buckets is selected in such a way that all the free memory is used. Each bucket has an associated counter and, whenever a candidate itemset is hashed to the bucket, its counter is increased. At the end of the pass, not only the counters of the candidate itemsets of size k are compared to the support threshold, in order to get frequent itemsets of size k, but also the counters of the buckets are comparred to the support threshold: if the counter of one bucket is at least the support threshold, it is considered a frequent bucket. In the next pass, where the algorithm works to find candidate itemsets of size k+1, the information about frequent buckets is used: an itemset is considered a candidate itemset of size k+1, hence its counter is increased, if the monotonicity filter is satisfied and if the itemset hashes to a frequent bucket.

In this explanation, only one hash function has been considered, but the number of hash functions applied can be increased, by obtaining a bigger number of filters on the candidate itemsets. This idea is implemented in the multihash algorithm, a variant of the PCY algorithm. In this version, a set of hash functions is applied to the candidate itemsets of size k+1 during the same pass performed to count candidate itemsets of size k. After the usage of all hash functions, different groups of frequent buckets are obtained, therefore they can be used to form different filters. The free memory is divided uniformly between the hash functions, so that the number of buckets for each of them is selected in such a way that all the free memory assigned to each hash function is used completely.

In our project, we have implemented the multihash variant of the PCY algorithm, where an arbitrary big number of hash functions can be defined and passed to the algorithm. The execution runs three functions:

- *get_ck*: it scans the baskets' file and it forms all the possible candidate itemsets of size k. The counter associated to each of them is increased only if the monotonicity filter is satisfied and if the buckets at which the candidate itemset is hashed by each of the hash functions is a frequent bucket. Another loop on data is performed to consider all the possible candidate itemsets of size k+1. Each hash function is applied and the counter of the obtained bucket is increased. It returns the counters of the candidate itemsets of size k and the frequent buckets for the itemsets of size k+1.
- *get_lk*: it calls the previous function *get_ck* in order to get candidate itemsets of dimension k. For each of the candidate itemsets, it compares its support with the support threshold. It returns the frequent itemsets, whose support is at least the support threshold, and the frequent buckets of size k+1 unmodified.
- *pcy_algorithm*: it is an Iterator on State, that computes the free memory for each hash function, calls the function *get_lk* on the current State and at the end updates the State. This function starts the algorithm itself.

3.2. SON algorithm

The SON algorithm is perfectly suited to be executed in a distributed environment with the baskets' file memorized in a distributed file system. It is based on the idea of dividing the baskets' file in chunks and applying an algorithm that can be executed in memory on each chunk, in order to find the frequent itemsets inside. The fraction of total baskets in a chunk is p, so the total number of chunks is $\frac{1}{o}$. The support threshold has to be lowered down considering the size of each chunk, so it will become $p * s$, where s is the original support threshold on the whole baskets' file.

Once all the chunks have been processed independently, the total set of frequent itemsets found is obtained by performing the union of frequent itemsets found in each chunk. Inside these frequent itemsets, there can be false positive, that is, itemsets considered frequent in the chunk, but not frequent in the whole dataset. These itemsets have to be discarded. For this reason, a final full scan of the baskets' file is perfomed, that allows to count the occurrences of all the frequent itemsets found in chunks. Itemsets whose support is below the original support threshold s are false positive and are discarded from the result.

The algorithm performs two scans of the baskets' file: the first one to read all the chunks and the second one to remove false positive. It is an exact algorithm, meaning that it does not give false positives nor false negatives.

In our project, the SON algorithm has been applied in conjunction with the Apriori algorithm, inside the distributed environment offered by Spark. In particular, we have decided to divide the baskets' file in five partitions, obtaining in this way five chunks. The sequence of operations can be described by two MapReduce jobs performed by the following three functions:

- *get_ck*: it scans one chunk to obtain frequent itemsets of size k inside the chunk, by applying the Apriori algorithm with a lower support threshold. This is the first map function. On its output, the first reduce function is applied, that discards duplicates, because one itemset can be frequent in more than one chunk. The final output is the set of frequent itemsets.
- *get_lk*: the result of the previous MapReduce job is broadcasted. This functions performs the last full scan in order to discard false positives: the second map function computes the support of each frequent itemset found considering a portion of the whole dataset and the second reduce function sums all the partial supports in order to get the whole support for each frequent itemset. If the support is lower that the original threshold, then the frequent itemset is discarded from the result. It returns the frequent itemsets, without false positives.
- *son_algorithm*: it is an Iterator on State, that calls the function *get_lk* on the current State and at the end updates the State. This function starts the algorithm itself.

3.3. Toivonen algorithm

The Toivonen algorithm is based on sampling baskets, in order to perform Market Basket Analysis only on some baskets and not the whole baskets' file. In this way, the size of the dataset is decreased. The algorithm performs one pass over a small sample and one final full pass over the whole dataset, becoming less demanding than the SON algorithm, but with a price to pay: there is a small, yet nonzero probability that the algorithm will fail to produce any output at all. In this case, it can be repeated until it gives an answer: as the sample is selected randomly from the dataset, a new execution could change the sample and give a successful output.

The fraction of baskets in the samples is p, so the baskets in the sample are $p * n$, where n is the total number of baskets. The support threshold has to be adjusted and it will become $0.9/0.8 * p * s$: the smaller the threshold, the more memory is used, the more likely the algorithm will give a successful output.

After selecting the sample, any algorithm that runs in memory can be used to find frequent itemsets. After this phase, the algorithm constructs the negative border.

Definition 3.2 (Negative border)**.** An itemset is in the negative border if it is not deemed frequent in the sample, but all its immediate subsets are.

After forming the negative border, the algorithm performs a full scan of the baskets' file, in order to determine the support of the frequent itemsets found in the sample and of the itemsets put in the negative border. There can be two possible outcomes:

- if no itemset in the negative border is frequent in the whole dataset, the algorithm outputs the frequent itemsets in the sample, that are frequent also in the dataset, discarding in this way the false positives.
- if there exist at least one itemsets in the negative border that is frequent in the whole dataset, the algorithm fails to produce an output, because there could exist some itemsets not in the negative border nor the set of frequent itemsets in the sample that are frequent in the whole.

The Toivonen algorithm is an exact algorithm, meaning that it does not give false positives nor false negatives.

In our project, we have applied the Toivonen algorithm, starting by creating the sample on which it has to be applied. In order to select its size, we have relied on the following table, taken from the scientific paper that presents the Toivonen algorithm, cited in the bibliography [2]:

ϵ	δ	$\|s\|$
0.01	0.01	27.000
0.01	0.001	38.000
0.01	0.0001	50.000
0.001	0.01	2.700.000
0.001	0.001	3.800.000
0.001	0.0001	5.000.000

ϵ is the acceptable error, δ is the probability that the error is bigger that ϵ and $|s|$ is the size of the sample. In the aforementioned paper, it is said that a sample of size 50.000 is sufficient in many applications. The Toivonen algorithm is then applied in conjunction with the Apriori algorithm and by using three functions:

- *get_ck*: it scans the sample and counts the occurencies of candidate itemsets of size k that pass the monotonicity filter.
- *get_lk*: calls the previous function *get_ck* in order to get candidate itemsets of dimension k. For each of the candidate itemsets, it compares its support with the support threshold, in order to get frequent itemsets of size k. It computes the negative border and performs a full scan of the baskets' file. If no itemsets of the negative border are frequent in the dataset, it returns the frequent itemsets by discarding the false positive, otherwise it returns None.
- *toivonen_algorithm*: it is an Iterator on State, that calls the function *get_lk* on the current State. If the output is None, it stops, otherwise it updates the State. This function starts the algorithm itself.

4. Evaluation of the algorithms

4.1. Constants

The constants used in the executions of the algorithms are:

- support threshold = 30
- number of chunks in the SON algorithm = 5
- adjustment of the support threshold for the Toivonen algorithm = 0.8
- Toivonen sample size = 5

4.2. Results

All the algorithms applied find the same number of frequent itemsets of each size, as expected. In particular, they find 7453 frequent singletons, 373 frequent doubletons, 90 frequent triples, 34 frequent quadruples, 5 frequent quintuples and no frequent sextuplets.

4.3. Execution times

The algorithms can be compared by their execution time, as shown in the table below, where Ti is the execution time in seconds required to find frequent itemsets of size i.

Algorithm	T1	T2	T3	T4	T5
Apriori Spark	4	11	3	3	3
Apriori	3	5	6	6	6
PCY multihash	22	32	29	23	16
SON	11	11	9	9	8
Toivonen	ND	ND	ND	ND	ND

The values have been computed as the averages of the execution times of three consecutive executions, by running the algorithms on Google Colab.

In Apriori implemented with Spark the slowest operation is the join in order to create candidate itemsets. In Apriori in memory the slowest operation is the scan of the baskets' file, that is the slowest operation in the PCY algorithm too. In the latter, during the read operation, all the bitmaps created from the hash functions must be checked and new bitmaps must be created for candidate itemsets of higher dimensionality. In the SON algorithm, the slowest and most dangerous operation is the method collect, called twice in our implementation. In the Toivonen algorithm, the slowest operation is the last scan of the baskets' file.

The execution times of the Toivonen algorithm are not defined, because our implementation does not behave as expected: the algorithm almost always ends with no output, because it finds frequent itemsets in the negative border. The only case in which it works is when we select more than the 50% of the original dataset with a very low support threshold. Therefore it is clear that there are some issues in our implementation.

4.4. Memory usage

The Apriori algorithm implemented with Spark memorizes the dataframe, which contains the data on which we are working, and another small dataframe needed to perform the join operation, in order to create candidate itemsets of size k+1 from frequent itemsets of size k. It creates also another dataframe, that contains the support of each candidate itemset. This implementation is scalable by definition, as it is executed in a distributed environment.

The algorithms presented below store the supports of each candidate itemset. Each support is represented by a (key, value) pair, where keys are tuples of k integers and values are integers, so each support occupies $4 * (k + 1)$ bytes, where k is the size of the candidate itemset.

The Apriori algorithm executed in memory stores the support of each candidate itemset of size k, in addition to the bitmap stored to identify frequent itemsets of size k-1. As the amount of RAM used at each step is low, the algorithm scales up linearly with the size of data. The execution time will increase linearly with the size of data.

The PCY algorithm with the multihash variant uses the memory to store the support of each candidate itemset of size k and the bitmaps for itemsets of size k+1, in addition to the bitmap stored to identify the frequent itemsets of size k-1. As the amount of RAM used at each step is low, the algorithm scales up linearly with the size of data. If there is no memory left after storing the counters, the buckets of the hash functions used, as they are selected to occupy all the free memory, will be 0, so the algorithm will essentially become an Apriori algorithm. The execution time will increase linearly with the size of data.

The SON algorithm stores the RDD, which contains the data on which we are working, and the frequent itemsets found in each chunk. The memory usage of each worker depends on the algorithm chosen to compute frequent itemsets; in our case the Apriori algorithm is applied, so the memory usage is the same as described above. It is scalable, as it is executed in a distributed environment. However, in our implementation, the method collect is called twice: data is moved to a single node and this operation could lead to a failure and decreases the scalability of the implementation.

The memory usage of the Toivonen Algorithm depends on the algorithm chosen to compute frequent itemsets. In our case, the Apriori algorithm is applied, so the memory usage is the same as described above, but limited to the size of the sample selected. In addition, the Toivonen algorithm requires memory to store the sample selected, the negative border with the support of each itemset contained in it and the actual frequent itemsets with their support, all computed by the last full scan of the baskets' file. As the amount of RAM used at each step is low, the algorithm scales up linearly with the size of data. The execution time will increase linearly with the size of data.

5. Conclusion

Our dataset consists of 393.759 movies. As the size of the dataset is very limited, the best algorithm can not be identified. In this context, the Apriori algorithm executed in memory seems to be the best solution; however, with a significant increase in the size of the initial dataset, it is very likely that the SON algorithm or the Apriori algorithm executed in Spark would have better performances, because of the distributed environment they reside in.

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

References

[1] JURE LESKOVEC, ANAND RAJARAMAN, JEFF ULLMAN, 2011, *Mining of Massive Datasets*, Cambridge University Press, Cambridge

[2] HANNU TOIVONEN, 1996, *Sampling Large Databases for Association Rules* In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, (Taj Mahal Hotel, Mumbai, India), Morgan Kaufmann Publishers Inc., 134-145.