
# Market Basket Analysis on the IMDB dataset in a Distributed Framework
# Algorithms for Massive Dataset Project

Nicole Maria Formenti - DSE

April 2021

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/our work. Iunderstand 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.

# Market basket analysis on the IMDB dataset in a distributed framework

## Introduction

This project focuses on the implementation of a market basket analysis on the *IMDb Dataset* from *Kaggle* repository, which contains information about movies and their ratings, along with the crew that took part in the movies. The market basket analysis consists in finding the frequent itemsets, which are items appearing together in the same baskets a sufficiently large number of times defined in advance. For this analysis, the movies are considered as baskets and the actors as items.
The problem tackled by the analysis is that of handling a large dataset in a distributed framework, which is a current issue in a world where an ever-increasing amount of data is available from all possible sources. The algorithms used are the *Apriori*, which works by generating candidate itemsets formed by frequent items and then matching them against the dataset, and the *FP-Growth*, which instead makes use of a compressed tree-like structure and develop an efficient mining method. The performance of the two in dealing with the task at hand is compared. In addition to the frequent itemsets, the association rules are retrieved. They provide information about which item is to be expected in a basket if other items are present.

## Dataset and data preparation

The dataset used for the analysis is the *IMDb Dataset* from *Kaggle repository*, which includes information about movies, their ratings and the corresponding crew. It is formed by five different tables, of which only three have been considered:

- *name.basics*: it contains information about the crew members. The variables are the following:
  - *nconst*: unique ID of the crew member
  - *PrimaryName*: name of the crew member
  - *birthYear*: it has been dropped
  - *deathYear*: it has been dropped

  - *PrimaryProfession*: different professions. Only the crew members which are actors have been selected.
  - *KnownForTitles*: list of movie IDs. It has been dropped

The final version of the table contains 9.706.922 actors.

- *title.principals*: it contains the correspondence between titles and crew members. Its features are:
  - *tconst*: unique ID of the title
  - *ordering*: a number to identify the different rows for each title ID. It has been dropped
  - *nconst*: unique ID of the crew member
  - *category*: role of the crew member for a particular title
  - *job*: the specific job title
  - *characters*: the name of the character played by the actor. It has been dropped

Crew members which are neither in the *category* actor nor in the *job* actor have been filtered out.

- *title.basics*: it contains information about moviesxf. Its variables are as follows:
  - *tconst*: unique ID of the title
  - *titleType*: format of the title. It has been dropped since it provides redundant information
  - *primaryTitle*: more popular title
  - *originalTitle*: title in the original language
  - *isAdult*: if the title has been conceived for an adult public. It has been dropped
  - *startYear*: the year of release. It has been dropped
  - *endYear*: the year of ending for series. It has been dropped
  - *runtimeMinutes*: the duration of the title. It has been dropped
  - *genres*: it contains up to three genres associated with the title

All the genres of titles have been considered.

The observations containing missing values for the variables *movie ID*, *actor ID*, *actor name* and *movie title* have been removed from all the tables. The number of observations after data cleansing for the different tables is the following:

- *actors*: 9.706.922 actors
- *movies to actors*: 14.818.830 correspondences between movies and actors

- *movies*: 6.321.302 movies

From these datasets two additional tables have been built. The first table *join data* contains the correspondence between movies' IDs and actors' IDs and the corresponding titles of the movies and names of the actors. It will be used later on to convert IDs of frequent itemsets into the corresponding actors' names. It is organised as follows:

```
movie id | movie title | actor id | actor name
```

The second table *baskets movies* is the one fed to the algorithm, which associates to each movie the list of actors that played in it in the format:

```
movie | [actor1, actor2, ...]
```

The variables considered for the analysis are the ID of the movies and the ID of the actors, since they are unique identifiers. Actually, there is a consistent number of actors having the same name and of movies having the same title.
All the datasets have been partitioned in 10 partitions to better exploit the distributed framework of *Apache Spark* for handling large data collections. ?

# Methods

The algorithms used to carry out the market basket analysis are the *Apriori* and the *FP-Growth*.

The *Apriori* algorithm is the basic implementation for finding frequent itemsets and it exploits the anti-monotone property, which says that the support of an itemset is never greater than the support of its subsets. Hence, all the subsets that can be derived from a frequent itemset must be frequent themselves. Whether an itemset is frequent or not is defined in terms of its support, which is as follows:

$$supp(j) = \frac{freq(j)}{total\ n.\ baskets}$$

An itemset is frequent only if its support is larger than a certain threshold, which is fixed in advance.
During the first pass the textitApriori counts the occurrences of each item and it filters only the frequent singletons. Then it builds a set of candidate pairs formed by frequent singletons and it scans again the dataset to retrieve the frequent pairs. This process is iterated.
The steps of the algorithm are:

1. Generate the candidates' set formed by frequent subsets
2. Count the occurrences of the candidates
3. Filter frequent candidates

The project contains an implementation from scratch of the *Apriori* algorithm, which generates the candidates' set in a slightly different way with respect to the original one. Instead of building candidate itemsets whose subsets are all frequent, it builds candidate itemsets where only some of the subsets are frequent. The ratio is that it is much more efficient to build a slightly larger set of candidates rather than checking if all the subsets of a candidate are frequent. That is because the generation of candidates is a costly task. Along with the frequent itemsets, the association rules are derived. They provide information about which item is likely to find in a basket given the same basket contains other items. They are expressed as:

$$\texttt{antecedent} \rightarrow \texttt{consequent}$$

For each association rule, some metrics are calculated in order to define how strong is that rule, namely the support, the confidence and the lift. The confidence is measured as:

$$conf(I \rightarrow j) = \frac{supp(I, \{j\})}{supp(I)}$$

Having a high confidence means that most of the times in which itemset $I$ appears, also item $j$ does. However, it doesn't provide information about how frequent is $j$ in all the baskets. In fact, it might be the case that item $j$ is very frequent in all the baskets, so the association rule has low significance.
The lift measure takes into account the popularity of item $j$ and it is derived as:

$$lift(I \rightarrow j) = \frac{supp(I, \{j\})}{supp(I)supp(\{j\})}$$

It measures the deviation of the observed rule from the rule under the assumption of independency between the body and the head of the rule, where the probability of two independent itemsets appearing together is equal to the product of the probability of each itemset to appear in a basket.
Unfortunately, the biggest drawback of the *Apriori* algorithm is that it is quite inefficient, especially when dealing with large datasets and long patterns of items or when using a low support threshold. The main bottlenecks of the algorithm are the generation of candidates, especially when their number is large, and the use of pattern matching to find frequent itemsets, which requires to pass the dataset iteratively. The latter is a problem when the dataset is particularly big and when the pattern to be matched is long.

The second algorithm used is the *FP-Growth*, which is an efficient alternative to the *Apriori*. It has been presented by *J.Han et al.* in their paper *Mining Frequent Patterns without Candidate Generation*. The algorithm takes a completely different approach, reducing the cost of computations in two ways:

1. It builds a compact data structure which is called ***frequent pattern tree***, where only the frequent singletons and their count are stored. The items are ordered in decreasing frequency and each item is represented as a node. The tree starts from an empty root and different branches represent different patterns of itemsets, called

*item prefix subtrees*. Every time a pattern is found, the corresponding items' count is increased. If two different transactions share an itemset or a subset of it, then they can be stored along the same item prefix subtree, saving space.
Another interesting feature of the *FP-tree* is the *head of node-links*, which connects the nodes referring to the same item in order to retrieve information about their frequency much faster.
To build the tree, the database needs to be scanned only twice, one for counting the frequency of singletons and another one to build the connections in the tree by exploring all the transactions. Moreover, it has been proved that this structure contains all the information needed to retrieve the complete set of frequent itemsets.

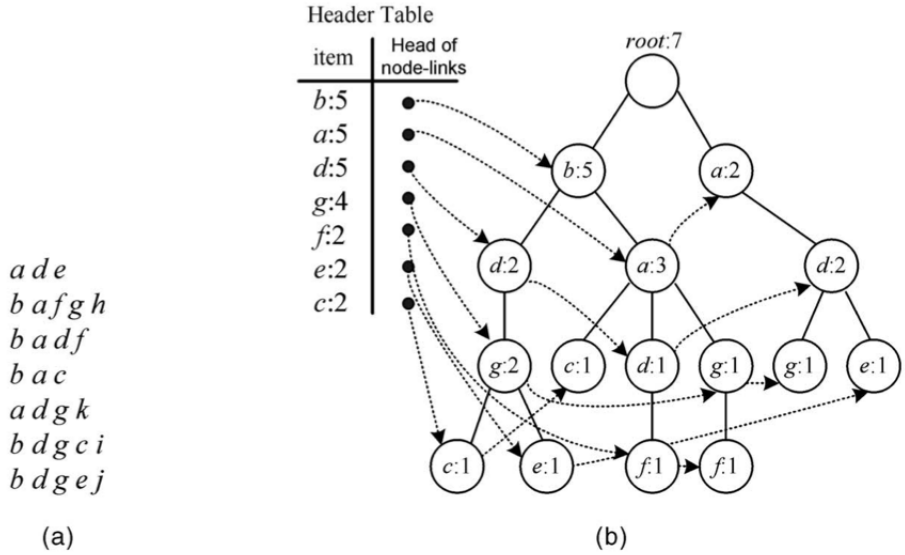


Figure 1: *(a) itemsets in the database (b) Frequent-pattern tree*

2. An efficient frequent pattern mining method called ***pattern fragment growth*** is used. This avoids the creation of the candidates' set, which is a costly process, moreover it is a *divide-and-conquer* method which divides the task into smaller subtasks.
   The mining starts from the last node of the header table, namely the less frequent node. If the frequency of this node is above the threshold, then it is considered, otherwise it is removed from the tree. If the node is frequent, the ***conditional pattern base*** is retrieved by taking all the prefix paths of the node in the different branches. This set of nodes is considered as a new smaller database used to build a new tree structure called ***conditional FP-tree*** containing only the frequent nodes, which will then be mined in the same way as before. The mining is iterated for all the frequent nodes in the *frequency pattern tree* in increasing frequency order.
   The process is illustrated below.

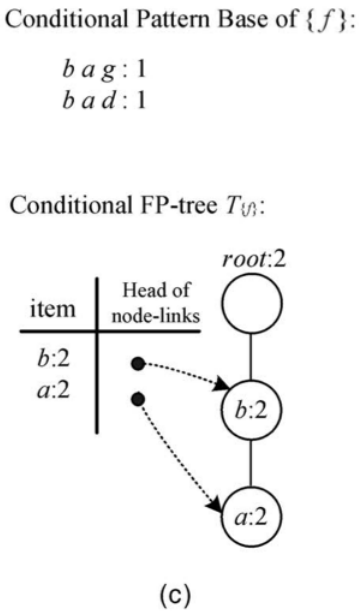


Figure 2: *(c) conditional pattern base and conditional FP-tree for the node f*

Compared to the *Apriori* algorithm, the *FP-Growth* is far less computationally expensive. The only step they have in common is the retrieval of frequent singletons, however the *FP-Growth* only requires to pass the dataset twice, while *Apriori* scans the whole dataset at each iteration. In addition, the latter has to generate the set of candidates at each iteration, which is costly in terms of computation and memory, whereas the former simply recursively mines the frequent pattern tree built at the beginning. These two are the main reasons why *FP-Growth* is way more efficient. This difference in performance gets larger as the dimension of the dataset grows and the minimum support threshold decreases, hence the *FP-Growth* scales much better. In fact, the authors of the paper state that this algorithm is an order of magnitude faster than the *Apriori*, especially for big datasets.

The analysis and data manipulation has been carried out with the *Apache Spark* analytics engine, which is a distributed framework used to efficiently handle large datasets, providing high scalability.
It relies on the use of RDD (resilient distributed datasets), which distributes the data to a cluster of workers that will perform their jobs in parallel. It uses a master/slave schema, where the workers or executors are coordinated by a central machine, called driver. An advantage of *Apache Spark* is that it makes use of *In-memory computing*, which stores the frequently used data in the main memory and avoid retrieving the data from the mass memory, whose access is costly in terms of time.
The parallelised operations executed by the system are of two types:

1. Transformation: the RDD is mapped into another RDD. Transformations are lazily evaluated

2. Action: the RDD is processed, however the result is not anymore an RDD. This operation triggers the transformation operations

Another important property of *Apache Spark* is the fault tolerance, which allows the system to keep running even if some of its components fail their task.

# Experiment

The goal of the experiment is the retrieval of frequent itemsets and of association rules along with their support, confidence and lift. Furthermore, the performance and scalability of the *Apriori* and the *FP-Growth* is compared for different dimensions of the dataset. Due to the large computational cost of the *Apriori* when dealing with large datasets, the algorithm has been run only on a subset of baskets formed by movies belonging to the *Musical* genre. The subset contains 35.111 baskets, as opposed to the total number of baskets in the dataset which is about 100 times bigger, namely 3.531.063.
Both algorithms have also been tested for subsets of data containing 15.000, 5.000 and 1.000 baskets in order to see how their performance changes as the number of baskets increases. The minimum support threshold is equal to 0.0006 for the *Apriori* and 0.00001 for the *FP-Growth* for computational time reasons and it has been kept constant for the different experiments. The resulting runtime trends have been plotted.

# Results

As expected, there is a large difference in the performance of the two algorithms. The runtime of the *Apriori* algorithm for the subset of 35.111 baskets is approximately 4.5 hours if the generation of frequent itemsets is taken into account and 1.5 hours only for retrieving the frequent itemsets, while that of the *FP-Growth* for the whole dataset composed by 3.531.063 baskets is on average less than 1 minute. The total number of frequent itemsets retrieved by the former is 10.244 with 40.610 association rules, while the latter retrieved a total of 603.39 frequent itemsets and 2.203.194 association rules. The minimum support selected is 0.0006 in the case of *Apriori* and 0.00001 in the case of *FP-Growth*. In both cases the longest frequent itemsets are composed by at most 10 items.

As for the comparison of the trend of the two algorithms when the size of the dataset varies, the time required by the *Apriori* algorithm to generate the association rules has not been taken into account. This is because this additional time probably heavily depends on the specific implementation at hand, hence only the time required to retrieve the frequent itemsets have been considered. The same is not true for the *FP-Growth* since, on the one hand, the already available implementation in the *Spark* framework computes both frequent itemsets and association rules by default, and on the other hand, the time required to retrieve the association rules is negligible. The plot of the trends of the runtime for *APriori* and *FP-Growth* is shown below.

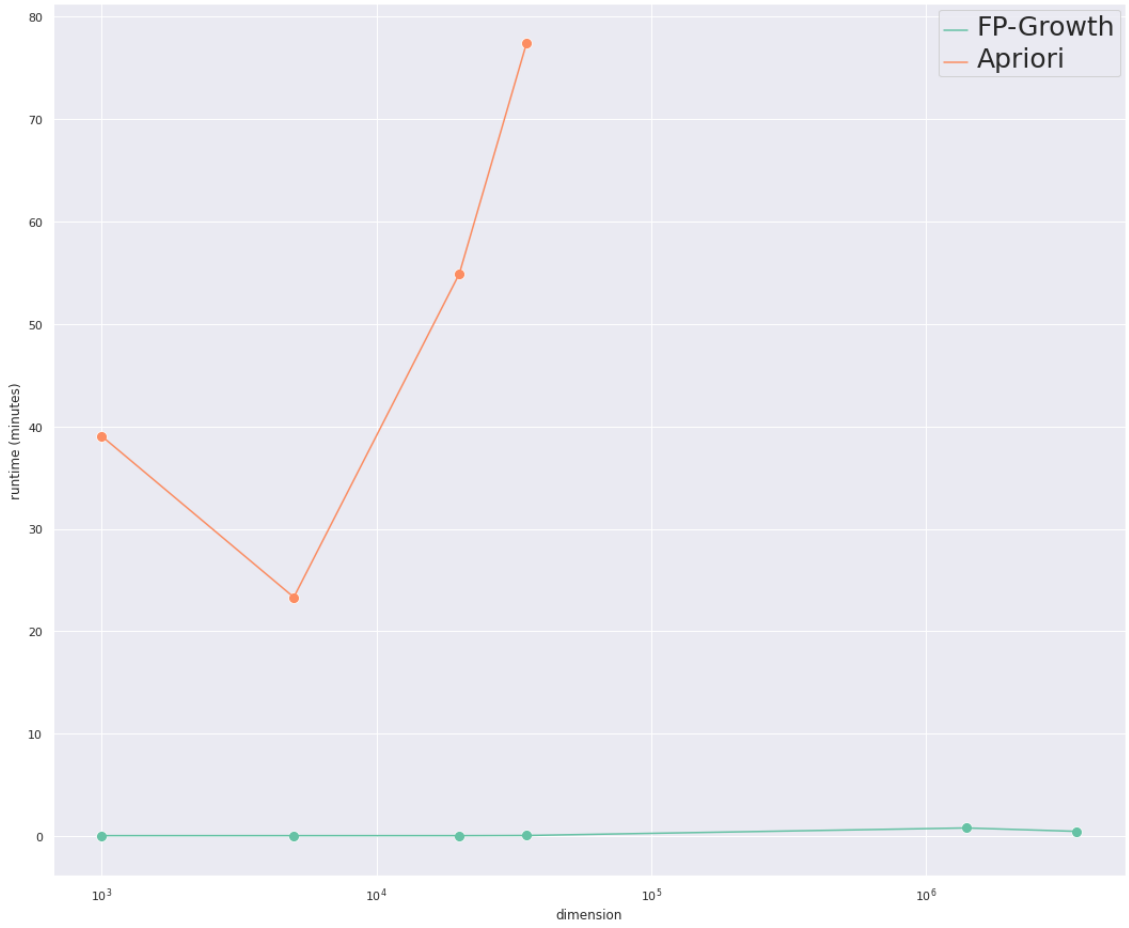


Figure 3: *FP-Growth vs Apriori trends as the dataset size varies. The x axis is in log-scale format*

The runtime of *FP-Growth* slightly increases with the size of the dataset, but overall its total runtime is stable. On the contrary, the performance of *Apriori* algorithm sensibly worsen as the dataset's dimension grows, even if the growth is contained as in this case.

In conclusion, when working with large datasets one should always choose an efficient and scalable solution, whose performance isn't excessively affected by the dimension of the dataset. Concerning the implementation of a system for the retrieval of frequent itemsets, the natural choice would be an algorithm such as *FP-Growth*, which took a remarkably lower amount of time, with respect to *Apriori*, to analyse a dataset 100 times larger and with a minimum support threshold 60 times smaller. More precisely, the ratio between the runtimes is 1 to 90 if the generation of association rules is not accounted for and 1 to 270 otherwise.