Alessandro Foffo 950124

# Algorithms For Massive Datasets Market Basket Analysis-IMDB Movies Dataset

06/2021

*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 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.*

# Contents

# Chapter 1

# Introduction

In the project, the data mining technique of Market-Basket analysis was applied on the IMDB-movies dataset.

The algorithms were implemented using the Apache Spark computational framework, while the code was run on Google Colab.

In particular, the movies were considered as baskets and the actors as items, and the purpose of the analysis was to retrieve only the frequent pairs of actors. This choice was motivated on one hand by semplicity, but on the other also by the fact that it is often rare that an intemset with cardinality larger than 2 is found to be frequent if the threshold is set high enough.

In particular, the A-Priori and the SON algorithms were used to mine the dataset and find frequent pairs of actors.

After finding the frequent pairs and their frequency (of course obtaining the same result with both algorithms) the support of the frequent items were inspected, in order to output the confidence and the interest of the respective association rules.

In the design of the algorithms, the most typical Spark transformations and actions (e.g. map(), flatMap(), reduce(), reduceByKey() etc.) were used.

In the next section, a deeper description of the techniques used and their implementation will follow.

# Chapter 2

# Dataset descritption, data organization and pre-processing

As it was anticipated in the introducion, the dataset considered was the IMDB movies dataset.

Since this dataset is available on Kaggle (and moreover to attain to the prescriptions of the project) the choice was to download the data during code execution, using the Kaggle API.

## 2.1 Dataset descritption

The dataset contains 5 tables: title akas, title principals, title basics,title ratings and name basics. Among them, only 3 were taken into consideration: title principals, title basics and name basics. All the tables were initially stored as a dataframe through the Spark SQLContext.

The title principals table contains 6 attributes: tconst (alphanumeric unique identifier of the title); ordering (a number to uniquely identify rows for a given tconst); nconst (alphanumeric unique identifier of the name/person); category (the category of job that person was in); job (the specific job title); characters (the name of the character played).

The title basics table contains 9 attributes: tconst (alphanumeric unique identifier of the title); titleType (the type/format of the title, e.g. movie, short, tvseries, tvepisode, video, etc); primaryTitle (the title used by the filmmakers when the movie was released); originalTitle (original title, in the original language); isAdult (0: non-adult title; 1: adult title); startYear (the release year of a title); endYear (TV Series end year); runtimeMinutes (primary runtime of the title, in minutes); genres (includes up to three genres associated with the title).

The name basics table contains 6 attributes: nconst (alphanumeric unique identifier of the name/person); primaryName (name by which the person is most often credited); birthYear; deathYear (if applicable); primaryProfession (the top-3 professions of the person); knownForTitles (titles the person is known for).

## 2.2 Data Organization

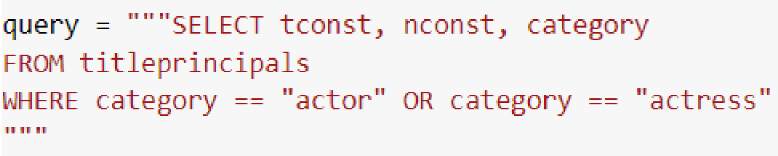
In order to have the data ready for the analysis, some preliminary operations on the tables were performed.

In particular, the purpose of these operations was to obtain an RDD where each entry represented a basket, namely a tuple of two elements: the first one representing the titleId of the movie, and the second one a list containing all the Ids - given by the attribute 'nconst' - of the actors playing in that movie.

This information was not directly available from the tables, for two reasons: first of all, the word 'title' stands for every possible audio-visual representation (e.g. Tv series, short, video..) and not just movies; furthermore, the dataset contains information about every professional figure (e.g. director, cinematographer, producer, editor..) and not only actors.

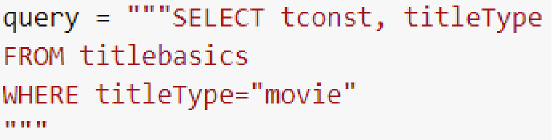
3 simple SQL queries (shown below and implemented via the Spark.sql function) allowed to create one final table, which was then transformed in the RDD containing the baskets:

- from the 'title.principals' table, only the rows where the attribute 'category' was either 'actor' or 'actress' were selected;



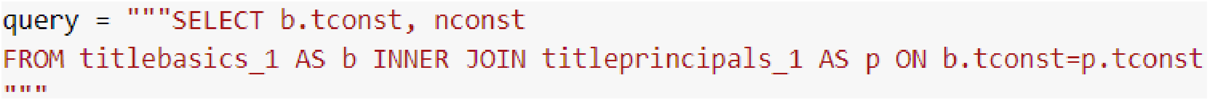
```
query = """SELECT tconst, nconst, category
FROM titleprincipals
WHERE category == "actor" OR category == "actress"
"""
```

- from the 'title.basics' table, only the rows where the attrbute 'titleType' was 'movie' were picked;



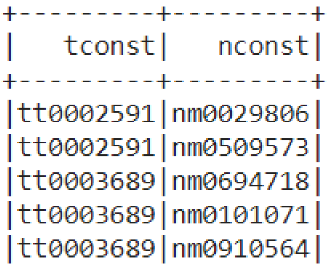
```
query = """SELECT tconst, titleType
FROM titlebasics
WHERE titleType="movie"
"""
```

- from the table obtained through an INNER JOIN operation between the two filtered tables, the attributes 'tconst' and 'nconst' were chosen, so as to obtain a table where there is an entry for every actor appearing in any movie.



```
query = """SELECT b.tconst, nconst
FROM titlebasics_1 AS b INNER JOIN titleprincipals_1 AS p ON b.tconst=p.tconst
"""
```

The final table obtained is reported below. Notice that each entry of the attribute 'tconst' represents the titleId of a **movie**, and each entry of the attribute 'nconst' is the Id of an **actor**.



```
+---------+---------+
|   tconst|   nconst|
+---------+---------+
|tt0002591|nm0029806|
|tt0002591|nm0509573|
|tt0003689|nm0694718|
|tt0003689|nm0101071|
|tt0003689|nm0910564|
```

Finally, the RDD containing the baskets was obtained from this table. In partcular, the *rdd* method was applied to the dataframe in which the table was stored and the *'groupByKey'* transformation was enforced in order to group together in the same basket all the actors appearing in the same movie.

# Chapter 3

# A-Priori Algorithm

A function was defined to implement the A-priori algorithm. This takes as input the RDD of baskets previously defined and a threshold set by the user, and outputs the list of frequent pairs of actors; the term 'frequent' in this context denotes a number of occurences larger than the threshold.

## 3.1 A-Priori - 1st pass

To implement the 1st pass of the algorithm, a series of transformations was applied to the RDD containing baskets: first, applying *'flatMap'*, a list containing all the actors appearing in all movies was obtained; then each actor occurency was sided with the value 1 through a *'map'* transformation to get a pair RDD, with keys given by the Id of the actor and values all equal to 1; the final RDD storing the counts for each single actor was obtained applying a *'reduceByKey'* transformation summing all the values (1) associated with each key (actor).

Enforcing the action *'collect'*, this distributed file was stored in main memory as a list.

## 3.2 A-Priori - Between the 2 passes

Between the 1st and the 2nd pass of the algorithm, the list storing the counts of each actor was filtered, to retrieve only the counts of the actors that appear a number of times larger than the threshold. In this way a list with all the frequent singleton items and their count was obtained.

Finally, another list containing only the Ids of the frequent actors, without their count was created.

## 3.3 A-Priori - 2nd pass

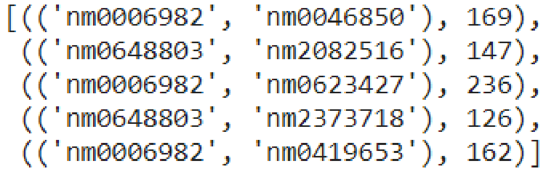
To carry out the 2nd pass of the algorithm, the baskets RDD was scanned again. First of all a *'map'* transformation was applied to discard the Id of the movies and only consider the list containing the Id of the actors appearing in them. Then, to detect the pairs that consist of two frequent actors inside each list, one *'map'*, one *'filter'*, one *'flatMap'* and another *'map'* transformations were performed. These are described below:

- The first transformation, for every list of actors in the RDD, only kept an item in the list if it was inside the list of frequent single actors.
- The second one was a little more subtle: since the occurrence of a pair where both actors are frequent can only be detected in a list with two or more elements, and since from the previous transformation many empty lists or with only one item were produced, with this transformation only the lists with length at least 2 were kept.
- The *'flatMap'* transformation instead applied to each list of the RDD (now containing only the Ids of frequent actors) the 'itertool' function 'combination', which, given a list, outputs a tuple for every possible combination of 2 elements in the list with no repetition.
- To count together the inverted tuples (containing the same pair but in inverted order) a simple alphabetic sorting was applied through a map transformation to every tuple composing the RDD.

Finally, the pairs composed of two frequent actors were counted in the same way as before: each occurrence was sided with a 1 through a *'map'* transformation; a *'reduceByKey'* transformation with the sum function was finally enforced to perform the count. The resulting RDD was stored in main memory as a list of key-value tuples containing as key a tuple representing a pair, and as value the frequency of the pair. This list was again filtered using Python's 'filter' function to only obtain pairs with frequency larger than the threshold. This list of frequent pairs represented the output of the function.

## 3.4 A-Priori - Results

The algorithm was implemented with a threshold of 125. The pairs output as frequent with this threshold were 5. They are shown below with the respective frequency.



```
[(('nm0006982', 'nm0046850'), 169),
 (('nm0648803', 'nm2082516'), 147),
 (('nm0006982', 'nm0623427'), 236),
 (('nm0648803', 'nm2373718'), 126),
 (('nm0006982', 'nm0419653'), 162)]
```

As it is possible to see, the most frequent pair's frequency exceeded 200.

# Chapter 4

# SON Algorithm

After having defined a function implementing the A-Priori algorithm, it was easy to implement the SON Algorithm as well: this, in fact, is based on obtaining the candidates by recursively applying the A-Priori on n equally sized chunks of the basket file, and then scanning the dataset to filter false positives.

## 4.1 Partitioning the baskets file

The choice was to create 10 equally sized chunks from the original baskets file. This was done in 2 steps:

- the first one involved storing, through the *'collect'* action, the RDD with the baskets as a list;
- The second step was giving that list as input to a function that, taking as input a list, partitions it in n equally sized chunks, returning a list of lists.

## 4.2 SON - 1st pass

A function implementing the 1st pass of the SON-Algorithm was defined. This, taking as input a list of chunks of the baskets file and a threshold, returned as output the list of candidate pairs output by the SON Algorithm. In particular, the function first recursively applied the A-Priori algorithm on each chunk of the list, storing the results (in the form of 2-tuples containing a pair and the respective frequency) in a list.

Then, through a list comprehension, the frequency of the pairs was discarded and, using the function set, duplicate pairs were removed, obtaining the equivalent of a union operations.

## 4.3 SON - 2nd pass

Once again, a function was defined implementing the 2nd pass of the algorithm. This took as input the RDD containing the baskets, the list of candidates output in the first pass and the threshold and returned the list of frequent pairs with their respective frequency.

The function operated on the RDD of baskets, applying first a *'map'* transformation to discard the Ids of the movies; then a *'flatMap'* trnsformation was enforced, which, for every list of actors, applied the 'combinations' function defined before.

The resulting RDD of tuples, after having gone through another *'map'* transformation sorting the tuples to count together inverted pairs, was filtered, keeping only the tuples appearing in the candidates set output in the first pass.

The remaining transformations mirrored the ones enforced also in the A-Priori algorithm to perform the count of the pairs: in particular, the pairs were sided with the number 1 through a textit'map' transformation and then a *'reduceByKey'* transformation counted the number of occurrencies of each pair.

Finally, a filtering operation was performed to selet only the candidate pairs which had a frequency larger than the threshold, and the resulting RDD was stored in main memory as a list containing the frequent pairs with their respective frequency.

## 4.4 SON - Results

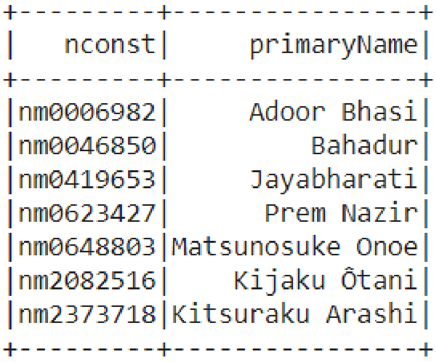
Not surprisingly, applying the SON Algorithm the same results obtained with A-Priori were achieved; the SON-Algorithm in fact does not output neither false positive nor false negatives.

# Chapter 5

# Analysis of the association rules

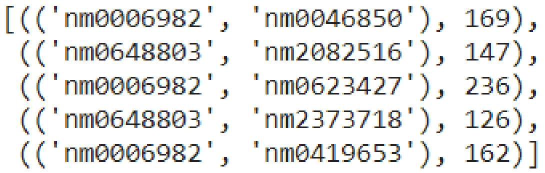
To conclude the assignment, the confidence of the association rules were analyzed for some of the frequent pairs.

In particular, as it was stated previously, 5 frequent pairs of actors were found. First of all, for the sake of completeness, the names of the actors were retrieved by querying the 'name.basics' table. The result of the query is shown below.



```
+---------+----------------+
|   nconst|     primaryName|
+---------+----------------+
|nm0006982|     Adoor Bhasi|
|nm0046850|         Bahadur|
|nm0419653|     Jayabharati|
|nm0623427|      Prem Nazir|
|nm0648803|Matsunosuke Onoe|
|nm2082516|    Kijaku Ôtani|
|nm2373718|Kitsuraku Arashi|
+---------+----------------+
```

Then, 2 examples of association rules were considered: the ones related to the most frequent and the least frequent pairs.



```
[(('nm0006982', 'nm0046850'), 169),
 (('nm0648803', 'nm2082516'), 147),
 (('nm0006982', 'nm0623427'), 236),
 (('nm0648803', 'nm2373718'), 126),
 (('nm0006982', 'nm0419653'), 162)]
```

As it can be seen in the set of frequent pairs output by the algorithms reported again here, the most frequent pair is the one formed by Adoor

Bhasi and Prem Nazir. In this analysis, the confidence of the association rule Bhasi→Nazir was computed; this is equal to the probability that Nazir appears in a movie where Bhasi played.

In general, the confidence of the association rule $I \to j$ is given by the ratio between the support for $I \cup j$ and the support for $I$.

In this case, the support for $I \cup j$ is the frequency of the pair, that is 236. The support for I instead, retrieved from the list of frequent actors output by the A-Prioi algorithm, was 585. The resulting confidence therefore was 236/585=0.4: this means that if Bhasi plays in a movie, then with probability 0.4 Nazir will also do it.

The pair with lowest frequency instead is the one formed by Onoe and Arashi. The support for $I \cup j$ in this case is 126, while the support for $I$ (retrieved by the list of frequent single actors) is 565.

The resulting confidence is 126/565=0.22: with probability 0.22, if Onoe plays in a movie, then Arashi will also do it.