# IMDB - Market Basket Analysis: a Spark Project

Simone Giachetta - 956536

October 22, 2021

# Contents

# 1 Introduction and Preliminaries

The following document shows a market-basket analysis on a dataset containing information about movies and actors. The goal of this type of analysis is to find frequent itemsets: it means find items (in this case actors) that are frequently in the same basket (in our case movie). This kind of analysis allow also to predict which item can be included inside the basket, given certain items already in the basket.

## 1.1 Tools and Packages

The programming languages used is Python and the tools used are the following.

#### Tools

**Google Colab** Provided by Google, it is a Jupiter Notebook with a ready to use computing resources that allows to write and execute Python.

**Kaggle** Kaggle is a competition platform that also provide a dataset repository easily accessible through API.

**Spark** Part of the Apache Foundation, it is an open-source analytics framework that allow to process large amount of data. His main feature is the in-memory cluster computing: thanks to that is possible to higher the speed of data processing. Furthermore, using the Resilient DIstributed Dataset (RDD) technique helps in managing failures: this because if a cluster fail to execute his job, the master can rebalance the work to the other nodes (which have the data in it). The 5 main components of Spark are:

- Spark Core: it contains the basic functionality and define the RDDs;
- Spark SQL: it is the package to work with structured data using SQL;
- Spark Streaming: allows processing of live stream data;
- MLib machine learning: provide machine learning algorithms and functionalities;
- GraphX: is a library for manipulating graphs and allow Spark to manage the nodes.

Image

**Python packages** Here the main packages used:

**pySpark** pySpark is an interface for Spark in Python. Allows to use the features (for example, SQL and ML) providing also distribute environment functions.

**Pandas** Pandas is (probably) the most famous tool for data manipulation and analysis in Python.

### 1.2 Dataset

The dataset exploited in this analysis is the IMDBb Dataset (available on Kaggle). The dataset offer 5 tables:

- title.akas: contain secondary movie characteristics (such as title, region, language);
- title.basics: contain primary movie informations (id, title, primary title, original title, genre);
- title.principals: link each movie to people who had been part of it (movie id, people id, role, job);
- title.rating: show the rating for each movie;
- name.basics: contain information about people and the main movie (people id, name, profession, knowsfortitles)

### 1.3 Goal

The goal is to build a market-basket analysis where movies are the baskets and actors are the items, resulting in a creation of frequent itemsets and suggesting which actors could be part of a certain team.

## 2 Data Preparation

### 2.1 Cleaning and Preprocessing

To begin, we start filtering the movie based on ratings. Being one of the goal the suggestion of possible actors in a certain team, it's better to create this suggestion on movie that had at least a mediocre rating: it's not a great advice to suggest an actor in a movie that went bad. Of course, it can be that the fault was of the director, budget or bad times, but anyway actors has an important role in a good doing of a movie. Using the rating table, it is possible to filter the movie with a rating of at least 4.5. Furthermore, the number of votes should be at least of 100 to be significant.

```
# Selecting only the film at least mediocre that has at least 100 votes
ratings_df = ratings.filter(ratings['averageRating'] > 4.4).filter(ratings['numVotes'] > 100)
ratings_df.createOrReplaceTempView('ratings_df')
ratings_df.show(n = 10)
```

```
+---------+-------------+--------+
|   tconst|averageRating|numVotes|
+---------+-------------+--------+
|tt0000001|          5.6|    1550|
|tt0000002|          6.1|     186|
|tt0000003|          6.5|    1207|
|tt0000004|          6.2|     113|
|tt0000005|          6.1|    1934|
|tt0000006|          5.2|     102|
|tt0000007|          5.5|     615|
|tt0000008|          5.4|    1667|
|tt0000010|          6.9|    5545|
|tt0000011|          5.2|     236|
+---------+-------------+--------+
```

The next step is to filter the principals table to retrieve only the actors: in doing that, it's also a good idea to check which approach is better to query the data, so it's possible to follow the most efficient.

```
q = 0
for i in range(10000):
  start = time.time()
  actors = principals.filter(principals['category'] == 'actor')
  q += (time.time() - start)
print(q)
```

```
13.940341711044312
```

```
q = 0
for i in range(10000):
  start = time.time()
  query = 'SELECT * from principals WHERE category = "actor" '
  actors_1 = spark.sql(query)
  q += (time.time() - start)
print(q)
```

```
13.715858221054077
```

It seems there is no winner.

Next step will be to filter actors who take part only in the "at-least-mediocre" movies, using the following query.

```
'''
    SELECT actors_1.tconst, actors_1.nconst
     FROM ratings_df
     INNER JOIN actors_1 ON ratings_df.tconst = actors_1.tconst
     '''
```

Ultimately, it's a good idea to delete the actors who take part only in one movie: it's not possible that actors with just one film recorded can be part of a frequent itemset.

```
query = '''SELECT *
          FROM (SELECT nconst, count(tconst) as counter FROM good_film_actors GROUP BY nconst)
          WHERE counter > 1'''
more_than_one = spark.sql(query)
more_than_one.show(n = 5)
```

```
+---------+-------+
|   nconst|counter|
+---------+-------+
|nm0709856|      7|
|nm0124236|      9|
|nm0706926|     11|
|nm0018091|      2|
|nm0564706|      2|
+---------+-------+
```

And now, using the IDs to inner join the actors table.

```
query = '''SELECT good_film_actors.nconst, good_film_actors.tconst
          FROM more_than_one
          INNER JOIN good_film_actors ON good_film_actors.nconst = more_than_one.nconst'''
last_actors = spark.sql(query)
last_actors.createOrReplaceTempView('last_actors')
last_actors.show()
```

```
+---------+---------+
|   nconst|   tconst|
+---------+---------+
|nm0000086|tt0045293|
|nm0000086|tt0049877|
|nm0000086|tt0050260|
|nm0000086|tt0051989|
|nm0000086|tt0052276|
```

Let's see how many rows we filter out.

```
print(actors_1.count())
print(last_actors.count())
```

```
8493701
435265
```

Reducing the dataframe from 8.5 million to 435k could seem too much but is also true that now the dataset is more significant that a huge one with lots of useless data

## 2.2 Exploratory Data Analysis

To have a better overview of our data, can be useful to run a quick data analysis.

Number of movie

```
query = ' SELECT count(distinct tconst) FROM last_actors'
qr = spark.sql(query)
qr.show()
```

```
+----------------------+
|count(DISTINCT tconst)|
+----------------------+
|                171270|
+----------------------+
```

Number of actors

```
query = ' SELECT count(distinct nconst) FROM last_actors'
qr = spark.sql(query)
qr.show()
```

```
+----------------------+
|count(DISTINCT nconst)|
+----------------------+
|                 41827|
+----------------------+
```

Average Movies per actor

```
query = ''' SELECT count(distinct tconst)/count(distinct nconst) as average
         FROM last_actors'''
qr = spark.sql(query)
qr.show()
```

```
+-----------------+
|          average|
+-----------------+
|4.094723503956774|
+-----------------+
```

Number of actors per number of movies

```
query = ''' SELECT counter, count(nconst)
          FROM more_than_one
          GROUP BY counter
          ORDER BY count(nconst) DESC'''
qr = spark.sql(query)
qr.show()
```

```
+-------+-------------+
|counter|count(nconst)|
+-------+-------------+
|      2|        14426|
|      3|         6619|
|      4|         3903|
|      5|         2617|
|      6|         1874|
|      7|         1355|
|      8|         1118|
|      9|          931|
|     10|          752|
|     11|          690|
|     12|          573|
|     13|          526|
|     14|          407|
|     15|          362|
|     16|          321|
```

Movie with the largest actors partecipation

```
query = '''SELECT tconst, count(nconst)
          FROM last_actors
          GROUP BY tconst
          ORDER BY count(nconst) DESC'''
qr = spark.sql(query)
qr.show()
```

```
+---------+-------------+
|   tconst|count(nconst)|
+---------+-------------+
|tt2221405|           10|
|tt2205635|           10|
|tt2606492|           10|
|tt0278228|           10|
```

From this quick exploration we got some information. For example:

- the maximum number of actors who play in the same movie, so the top of a possible frequent itemsets;
- the average number of actors in a movie, could be possibile that the more insightful frequent itemsets are around that size (so between 3 and 5);
- most of the actors played in 2 to 10 film, so probably the majority of the itemsets will have a size of 2/3.

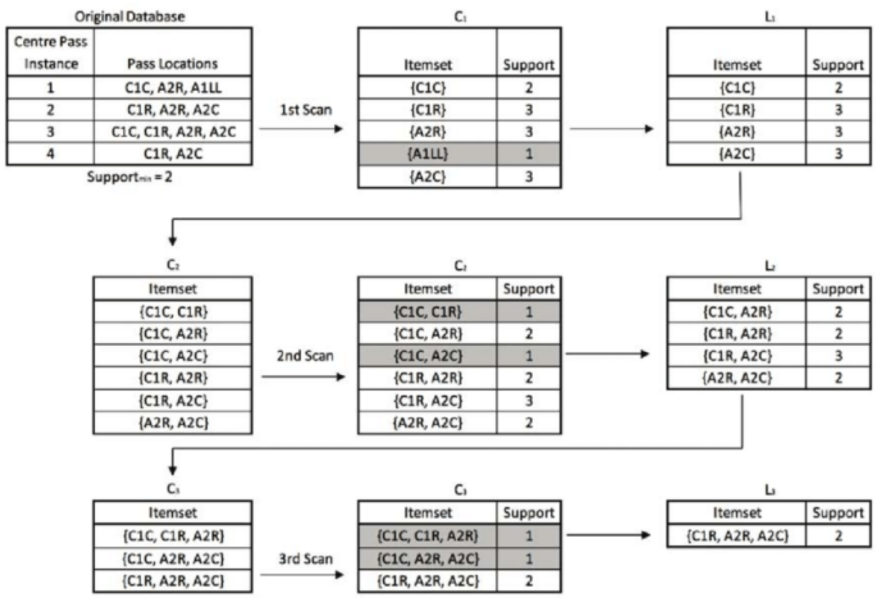



Figure 1: Source: ResearchGate, pubblication of Samuel J Robertson

# 3 MBA and FTGrowth

One of the most famous algorithms for the Market Basket Analysis in the Apriori. This algorithm use two steps to reduce the space used by pairs. During the first step each singleton is counted and during the second phase the singleton are filtered using a threshold (minimum support): if the counter is higher the singleton is accepted as candidate. This process is looped and at each level the size of the itemset grow by one: at the second level there are tuples, at the third are triples and so on. The algorithm is easy to understand and deploy but requires high computational resources if the itemsets are large and the support is low; also, the entire set of data needs to be scanned.

[Figure 1]

To deal with the limitations, an improvement of the Apriori algorithm have been designed.

The Frequent Pattern Growth algorithm represent the dataset in a tree structure that keep association between itemsets. Creating leaves using a choosing itemset as root allow to reduce the time needed.

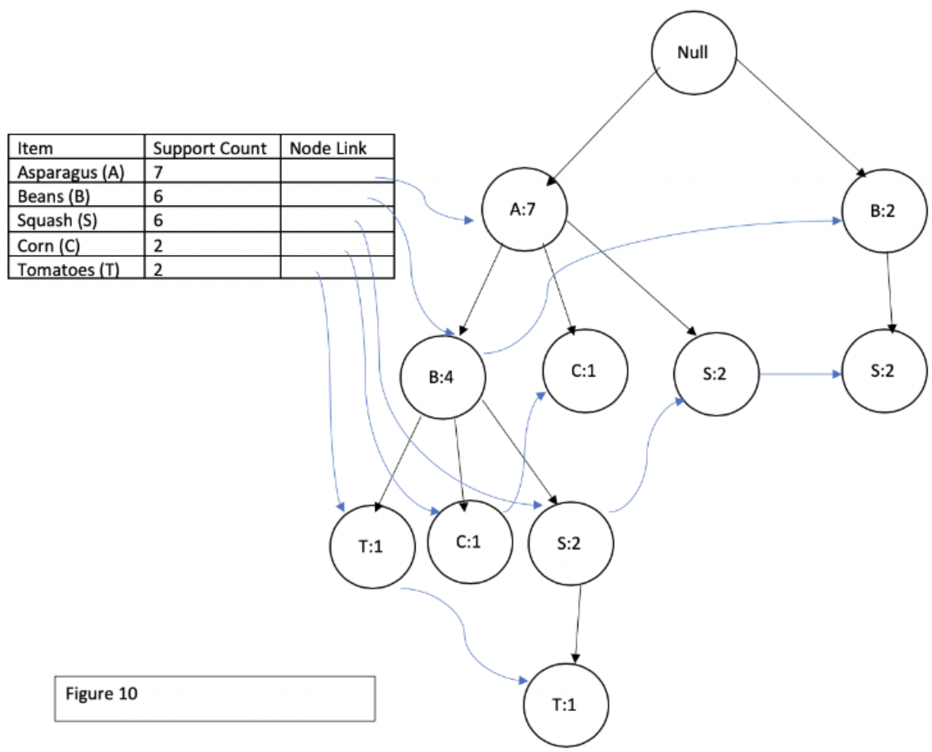



Figure 2: Source: www.mygreatlearning.com

# 4 Results

FT Growth is the package already build-in Spark that allows to exploit the Frequent Pattern Growth algorithm. The first step to take is create the baskets and generate the model. From here, it's possible to display the frequent itemsets.

```
baskets = last_actors.groupBy('tconst').agg(collect_set('nconst').alias('name_id'))
baskets.createOrReplaceTempView('baskets')
fpGrowth = FPGrowth(itemsCol="name_id", minSupport=0.001)
model = fpGrowth.fit(baskets)
```

```
mostPopularItemInABasket = model.freqItemsets
mostPopularItemInABasket.createOrReplaceTempView("mostPopularItemInABasket")
mostPopularItemInABasket.show()
```

```
+--------------------+----+
|               items|freq|
+--------------------+----+
|         [nm0000305]| 856|
|         [nm0144657]| 720|
|         [nm0532235]| 659|
|         [nm0642145]| 516|
|         [nm0224007]| 501|
|         [nm0444786]| 442|
|         [nm0001293]| 439|
|[nm0001293, nm053...| 341|
|         [nm0299192]| 434|
|         [nm0915762]| 425|
|         [nm0001101]| 424|
|         [nm0001832]| 421|
|         [nm0001319]| 419|
|[nm0001319, nm091...| 307|
```

Then print the table of associations

```
associationRules = model.associationRules
associationRules.createOrReplaceTempView("associationRules")
associationRules.show()
```

```
+--------------------+-----------+------------------+------------------+
|          antecedent| consequent|        confidence|              lift|
+--------------------+-----------+------------------+------------------+
|[nm0004898, nm049...|[nm0000494]|0.9942857142857143| 901.0122448979592|
|         [nm0004310]|[nm0002935]|               1.0| 835.4634146341463|
|         [nm0552509]|[nm0001832]|0.9019607843137255| 366.9330725164175|
|         [nm0427489]|[nm0001083]|0.9955947136563876| 615.5794462380127|
|[nm0000996, nm000...|[nm0000408]|               1.0| 882.8350515463918|
|[nm2625816, nm064...|[nm0000563]|               1.0| 658.7307692307692|
|[nm0183417, nm000...|[nm0756114]|0.9948453608247423| 851.9358247422681|
|         [nm0005380]|[nm0004951]|0.8647540983606558| 617.1101434426229|
|         [nm0333410]|[nm0005194]|0.8695652173913043| 559.8888525661981|
|         [nm0333410]|[nm0005110]|0.8647342995169082|  423.151552795031|
|         [nm0813812]|[nm0913587]|0.9347826086956522| 870.1098771266542|
|         [nm0394438]|[nm0000994]| 0.974169741697417|507.13085611099274|
|         [nm0394438]|[nm0848251]|0.8081180811808119| 604.3946889250552|
|         [nm0001652]|[nm0001101]|0.9004524886877828|363.72758900367114|
|         [nm0137230]|[nm0005531]|0.8942731277533039| 705.8163990336792|
|         [nm0137230]|[nm0261678]|0.8942731277533039| 729.3436123348017|
|         [nm0374865]|[nm0301959]|0.9688581314878892| 568.2751102052424|
|         [nm0374865]|[nm1433588]|0.9688581314878892| 578.1753734492362|
|         [nm0647638]|[nm0000563]|0.9322033898305084| 614.0710560625814|
```

Last step, using the trasform fuction and taking as an input items against all the association rules and summarize the consequent as prediction.

```
# transform examines the input items against all the association rules and summarize the
# consequents as prediction
asso = model.transform(baskets)
asso.createOrReplaceTempView('asso')
query = 'SELECT * FROM asso WHERE size(prediction) > 0'
prediction = spark.sql(query)
prediction.show()
```

```
+---------+--------------------+--------------------+
|   tconst|             name_id|          prediction|
+---------+--------------------+--------------------+
|tt0040695|[nm0694090, nm000...|         [nm0394438]|
|tt0244000|[nm0268199, nm053...|         [nm1533927]|
|tt0302361|[nm0001305, nm000...|[nm0374865, nm143...|
|tt0380609|[nm0000321, nm033...|[nm0005194, nm000...|
|tt0720166|[nm0005194, nm000...|         [nm0005110]|
|tt0810922|[nm0283945, nm033...|[nm0005194, nm000...|
```

As a result, With the IDs inside the prediction we can retrieve names of actors. For example:

- a prediction for the group composed by Kyle Gallner, Matthew Gray Gubler, Adam Nee and Hannibal Burres is: Joe Mantegna.

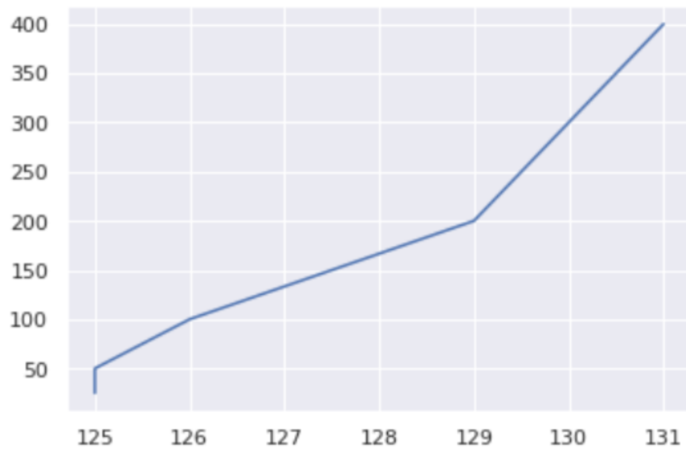
```
+------------+
| primaryName|
+------------+
|Kyle Gallner|
+------------+

+-------------------+
|        primaryName|
+-------------------+
|Matthew Gray Gubler|
+-------------------+

+-----------+
|primaryName|
+-----------+
|   Adam Nee|
+-----------+

+---------------+
|    primaryName|
+---------------+
|Hannibal Buress|
+---------------+
```

```
people.filter(people['nconst'] == 'nm0001505').select('primaryName').show()
```

```
+------------+
| primaryName|
+------------+
|Joe Mantegna|
+------------+
```

- a prediction for the group composed by Denis Leary, John Leguizano, Seann William Scott and Ray Romano is: Brad Garrett.

Closing the analysis, it's a good idea checking the time required based on different size of the dataset. As it's possible to see, the time required increase but not in a worrying way. Setting the size of the dataset of 25000, 50000, 100000, 200000, 400000 rows the time required goes from 125 to 131 seconds.




**Bibliography**


- https://www.oreilly.com/library/view/learning-spark/9781449359034/ch01.html
- https://spark.apache.org
- https://www.softwaretestinghelp.com/
- https://www.mygreatlearning.com/blog/understanding-fp-growth-algorithm/


**DECLARATION** *I declare that this material, which I/We now submit for assessment, is entirely my/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 my/our work. I/We 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/us or any other person for assessment on this or any other course of study*