Market-basket Analysis on IMDB

Giorgia Mazzi[1]

[1]Algorithms for Massive Datasets - Data Science and Economics, Università degli studi di Milano
[1]e-mail: giorgia.mazzi@studenti.unimi.it
[1]matricola: 942268

ABSTRACT

This paper will deal with one of the most relevant techniques for charactering data: the discovery of frequent itemsets. This is possible by searching for the «association rules» between items. In this paper, I used the IMDB dataset from Kaggle that I studied to find the most frequent actors and actress and the most frequent pairs of actors and actress. I applied a market basket model where I used, as items, the actors and actress and, as baskets, the movies in which the actors and actress played. To find solutions, I applied two different algorithms: Apriori and Fp Growth. Due to the massive dataset, it was essential to adopt Apache Spark: an open-source analytics engine focused on distributed system that helps me with the goal of the project. The analysis is available in Github, using Google Colab.
Keywords: Market Basket Analysis, Apriori, FP Growth, Apache Spark;

Declaration of authorship: *"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."*

Outline

The goal of the project was to implement the right algorithms in order to find frequent itemsets in the «IMDB» dataset, published on Kaggle. I will explore the market-basket model of data by adopting different algorithms: Fp Growth and Apriori.

1 Introduction

Market-basket model of data is a way to represent and describe relationships between objects. I will explain this model with the «IMDB» dataset that you can find in Kaggle: `https://www.kaggle.com/ashirwadsangwan/imdb-dataset`. The model was composed by baskets (movies) and items (actors). The dataset is made up a huge sequence of baskets (movies), while each movie contains less actors than the total number of items(actors) that you can find in the whole dataset. The aim of the project is to understand the actors, but even more the pairs, triples, quadruples of actors that appear to be frequent in the dataset.

2 Dataset and Methodology

2.1 Dataset

IMBD dataset is published on Kaggle and released under the public domain license (CC0). The dataset presents different csv (listed below) that report different details and information about movies and about the different roles that each person could played in at least one of them.

- title.akas.tsv.gz, with the information for titles
- title.basics.tsv.gz, with other information for titles
- title.principals.tsv.gz, with the principal cast/crew for titles
- title.ratings.tsv.gz, with the IMDB rating and votes information for titles
- name.basics.tsv.gz, with the information for names of actors

For the goal of the project, I selected some features that I deemed to be most relevant for the analysis and I used just three of the csv that I renamed:

- title.basics.tsv.gz as **infotitlebasics**
- title.principals.tsv.gz, as **castpertitles**
- name.basics.tsv.gz as **infofornames**

From data presented in those datasets, I applied a pre processing step where I decided to filter features according to the purpose of the project. I decided to retrieve in the castpertitles file the lines where the category of the players in a movie was just actor and actress and in infotitlebasics, I selected the rows when the type of the videos was a movie. The dataset of the paper was composed by the three listed csv by using the SQL's language in Spark. So, I joined the features that will help me to retrieve the most useful information of movies and actors. I decided to apply an inner join between infotitlebasics and castpertitles, so to retrieve the information that are simultaneously shared - that are presented in all the two csv. Due to the missing presence of the name of the actors presented in the dataset, I decided also to apply to my dataset an inner join with infofornames: this was useful to retrieve the name of the actors and actress for the dataset.

The dataset used presented 1.692.939 rows and 6 columns.

```
+---------+--------------------+--------------+---------+------------+------------+
| ID_movie|         TITLE_movie|category_movie|ID_actors| NAME_actors|Role_in_movie|
+---------+--------------------+--------------+---------+------------+------------+
|tt0077621|         Goin' South|         movie|nm0000004|John Belushi|       actor|
|tt0082801|           Neighbors|         movie|nm0000004|John Belushi|       actor|
|tt0077975|National Lampoon'...|         movie|nm0000004|John Belushi|       actor|
|tt0078723|                1941|         movie|nm0000004|John Belushi|       actor|
|tt0080455|  The Blues Brothers|         movie|nm0000004|John Belushi|       actor|
+---------+--------------------+--------------+---------+------------+------------+
only showing top 5 rows
```

Figure 1. Dataset, only 5 rows

I will detail from which file I selected the features presented in the dataset:

- from infotitlebasics, I selected *tconst* that I renamed as **ID movie**
  (by the description provided on Kaggle, *tconst* (string) is an alphanumeric unique identifier of the title)
- from infotitlebasics, I took also *primaryTitle* that i called **TITLE movie**
  (*primaryTitle* (string) details the more popular title: the title used by the filmmakers on promotional materials at the point of release.)
- from infotitlebasics, I chose *titleType*, as **category movie**
  (*titleType* (string) explains the type/format of the title: e.g. movie, short, tvseries, tvepisode, video, etc.)
- from castpertitles, I retrieved *nconst*, as **ID actors**
  (*nconst* (string) is an alphanumeric unique identifier of the name/person.)
- from infofornames, I took *primaryName* for the name of the actors that I called **NAME actors**
  (*primaryName* (string) is the name by which the person is most often credited.)
- from castpertitles, I took *category*, as **Role in movie**
  (*category* (string) explains the category of job that person was in.)

I chose to select actors and actress presented in the castpertitles rather than in infofornames after an exploratory analysis on the different csv. Infofornames file presents the role that different persons played in different movies. It is possible to see that different actors and actress played also in other film different roles as assistant director, director, producer, production manager etc. To be sure that the role played, that made this person frequent or not, was actor or actress I used the castpertitles file that present a clear list of roles played in different movies. I preferred to use this data even if the actors presented are less than into the other file: in castpertitles, there are 1.867.043 actors and actress, while infofornames had 2.914.287 actors and actress.

Once I preprocessed the dataset and checked if the dataset collected only movies, actors and actress, I examined that there was no null values in the dataset and then I organized data in baskets. The baskets are the movies presented in the dataset. Each basket contained different actors and actress that played that film. To compose the baskets, I used ID features, both for movies and actors.

```
+---------+--------------------+
| ID_movie|              actors|
+---------+--------------------+
|tt0000335|[nm1012612, nm067...|
|tt0000502|[nm0252720, nm021...|
|tt0000630|         [nm0624446]|
|tt0000676|[nm0140054, nm009...|
|tt0000793|         [nm0691995]|
+---------+--------------------+
only showing top 5 rows
```

Figure 2. Baskets, only 5 rows

2.1.1 Exploratory Analysis on the dataset

Before starting to apply algorithms that will help me in the market basket analysis, I decided to answer different questions.

- **1.** Which is the number of actors or actress per film? How many movies have the same number of participants in the cast?
- **2.** Who are the TOP ten actors that played more movies?

To solve the first question I tried to retrieve, by the movies, the number of participants that each film has. In the analysis, we can see that the majority of movies in the dataset contains just four actors and actress in the cast. Infact in the dataset we had 221.462 movies with four figures that played the role of actor.
In the order of 22.000-29.000 number of movies, we can see (in a descending order) that: 29.313 movies had 6 actors, 28.288 films only 1 actor, 25.577 movies 5 actors, 23.617 movies 7 actors and 22.102 3 actors.
Between 15.000 and 18.000 amount of movies, the dataset presents 18.880 movies with 2 actors and 15.239 movies with 8 actors.
It is really unfrequent to find a cast composed by a lot of actors or actress in a cast: in fact, in the dataset, there were 8.017 with 9 actors and 1.159 with 10 actors or actress in the crew.

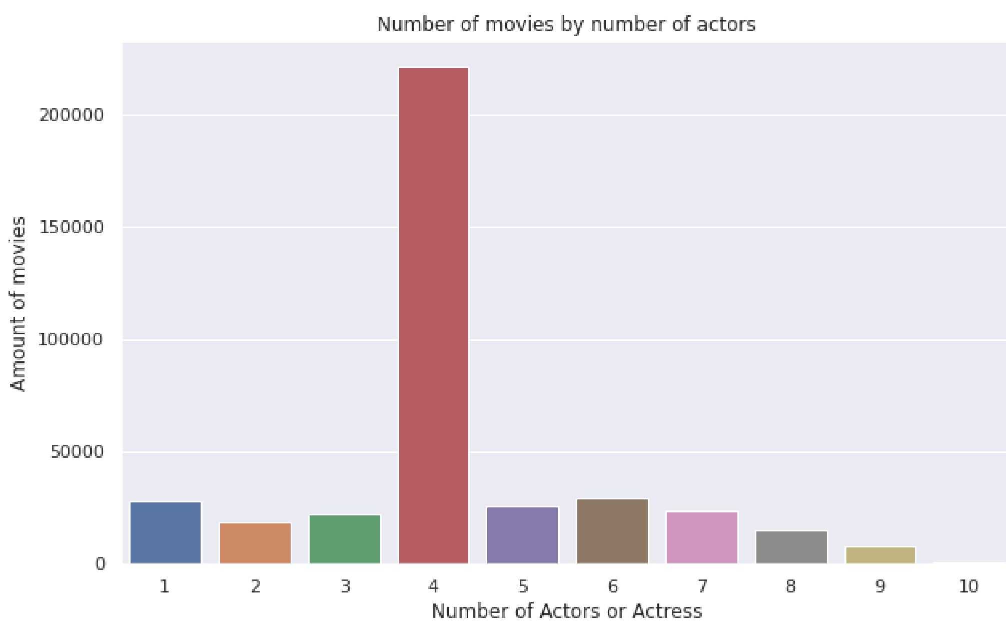


Figure 3. Amount of movies by the number of actors in the cast

The second question will bring us closer to the next analysis. I searched for ten most frequent actor or actress of movies in the dataset. By the query I imposed I found that the most frequent actors are, in order from the most frequent to the less: Brahmanandam, Adoor Bhasi,Matsunosuke Onoe, Eddie Garcia, Prem Nazir, Sung-il Shin, Paquito Diaz, Masayoshi Nogami, Mammootty, Aachi Manorama, Bahadur.

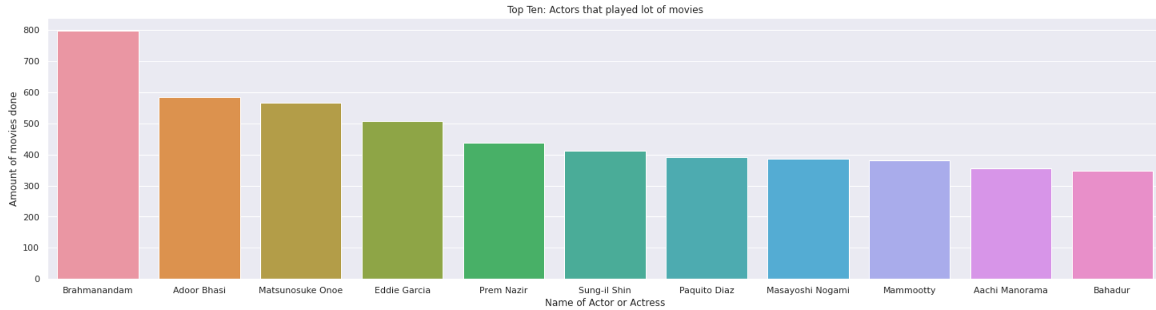


Figure 4. Top Ten: Actors that played lot of movies

2.2 Methodology

To reach the aim of this project I implemented two different algorithms: the FP growth and the Apriori algorithm.

The development of the FP growth algorithm is related to the concept of association rules. It is possible to find the algorithm in the machine learning Spark library for extracting frequent itemsets and its documentation. On 16 May 2000, in "Mining frequent patterns without candidate generation", J. Han, J. Pei, Y. Yin proposed this innovative method to understand the association rules between objects: "the frequent pattern tree (FP-tree) structure, which is an extended prefix-tree structure for storing compressed, crucial information about frequent patterns, and develop an efficient FP-tree-based mining method, FP-growth, for mining the complete set of frequent patterns by pattern fragment growth". FP growth algorithm starts by generating the frequent itemsets according to the minimum support defined by the user. After selecting those itemsets that result frequent according to the hyperparameter imposed, it starts the construction of the FP Tree. According to the support of each items in the transactions, we can built the tree by the root from the leaves, adding the items from the most frequent to the less, as in all transactions are presented. In the second phase, we need to construct the frequent pattern generation by studying the conditional pattern base and the conditional FP tree, which consider the support of the elements and the path to achieve it in the tree. The FP growth exploits the frequent patterns in the item sets of dataset rather than considering all the combinations of frequent items. By building this fundamental tree, it is possible to compress the information and to scale up with data size.

The Apriori algorithm was proposed by Agrawal and Srikant in 1994. It can be used to operate on databases containing lots of transactions. It proceeds by identifying the frequent items in the dataset, extracting and extending them to pairs sets (then triples etc) as long as those item sets appear sufficiently frequent. The approach is a "bottom up": once it is performed the candidate generation (and its support is above that imposed by the user) then the most frequent items will form groups of candidates that will be tested. After constructing candidates, the algorithm prunes the candidates which have a support minor that imposed by the user. So by a number (according to goal to achieve) of alternation of constructing and filtering candidates, it is possible to find the most frequent singletons, pairs, triples.. If the FP growth is already available in the Spark environment, it is useful to build the Apriori algorithm from scratch to exploits its advantages.

By the implementation of these two algorithms is possible to manage large dataset by scaling us its size, even if the two are different. In the first approach, you can reach the goal by following the frequent pattern, while in the second approach with Apriori, it is possible by selecting the most frequent items to understand which are the other pairs, triples ect to be considered frequent (candidates generation).

2.2.1 Implementation of the environment

In this section I will try to detail the settings that I had to define to implement the two algorithms. The code of this project is written in Google colab, that I deposited on Github. Google Colab is an interesting platform that allows us to execute code directly on the Cloud. I started the code by loading the dataset from Kaggle.

```
from google.colab import files #upload kaggle.json, containing API
files.upload()
```

```
Scegli file kaggle.json
• kaggle.json(application/json) - 68 bytes, last modified: 7/11/2021 - 100% done
Saving kaggle.json to kaggle.json
{'kaggle.json': b'{"username":"giorgiamazzi","key":"54d405d836593c93344595c2ee691c6e"}'}
```

```
! pip install -q kaggle
```

```
! mkdir -p ~/.kaggle #make directory with name Kaggle
```

```
! mv kaggle.json ~/.kaggle/ #move json file into the directory created
```

```
! chmod 600 ~/.kaggle/kaggle.json #give permission to this file
```

```
! kaggle datasets download -d ashirwadsangwan/imdb-dataset
```

```
!unzip imdb-dataset.zip
```

Figure 5. Load dataset from Kaggle into Google Colab

It is advisable to work with Apache Spark in order to manage massive dataset. I get the open source from the Apache Software Foundation. Apache Spark is an open-source unified analytics engine for large-scale data processing. Whereas Hadoop reads and writes files to HDFS, Spark processes data in RAM using RDD (Resilient Distributed Dataset). A starting point into all functionalities in Spark is the Spark Session and Spark Context.

```
!apt-get install openjdk-8-jdk-headless -qq > /dev/null  #install Java
!wget -q https://apache.osuosl.org/spark/spark-3.2.0/spark-3.2.0-bin-hadoop2.7.tgz #download spark3.0 with hadoop
!tar xf spark-3.2.0-bin-hadoop2.7.tgz #unzip folder
```

```
!pip install -q findspark #locate Spark on the system
import findspark
findspark.init("spark-3.2.0-bin-hadoop2.7")
```

```
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64" #set the environmental path that will enable to run Pyspark
os.environ["SPARK_HOME"] = "/content/spark-3.2.0-bin-hadoop2.7"
```

```
!pip install pyspark
import pyspark
from pyspark import SparkConf, SparkContext
from pyspark.sql import SparkSession
```

```
Requirement already satisfied: pyspark in /usr/local/lib/python3.7/dist-packages (3.2.0)
Requirement already satisfied: py4j==0.10.9.2 in /usr/local/lib/python3.7/dist-packages (from pyspark) (0.10.9.2)
```

```
spark = SparkSession.builder.getOrCreate()
```

```
sc = spark.sparkContext
```

Figure 6. Setup of Spark

In Appendix section you con find the code for the implementation of the two algorithms.

3 Empirical Results

I decided to evaluate the performance of the two algorithms in their ability to scale up with data size and to produce the desired results. To run FP-growth, I had to specify as hyper-parameter the minSupport, which is the minimum support for an itemset to be identified as frequent. As I had in baskets 393.654 movies, I started by considering frequent an actor or a pair that played at least 10 movies. This little threshold allows to retrieve single frequent actors and also pairs, triples, quadruples of actors that should be considered frequent together.
Building a FP tree and considering conditional patterns, the algorithm was able to identify as frequent: 24.054 actors and 3.443 pairs of actors.

NAME_actors	items	freq
Brahmanandam	nm0103977	798
Adoor Bhasi	nm0006982	585
Matsunosuke Onoe	nm0648803	565
Eddie Garcia	nm0305182	506
Prem Nazir	nm0623427	438
Sung-il Shin	nm0793813	411
Paquito Diaz	nm0246703	391
Masayoshi Nogami	nm0619107	387
Mammootty	nm0007123	381
Aachi Manorama	nm7390393	355

Figure 7. Frequent actors - FP Growth algorithm

In the picture, it is possible to identify the names of the 10 most frequent actors. The FP Growth algorithm was able to find the same actors that we can see in the picture for the solution of Question 2, in the same order.

items	freq
[nm0623427, nm0006982]	237
[nm0046850, nm0006982]	169
[nm0419653, nm0006982]	162
[nm2082516, nm0648803]	147
[nm2373718, nm0648803]	126
[nm0619779, nm0006982]	122
[nm2373718, nm2082516]	113
[nm2077739, nm0648803]	113
[nm0419653, nm0046850]	109
[nm0659173, nm1006879]	103

only showing top 10 rows

NAME_actors	items	freq
Prem Nazir	nm0623427	237
Adoor Bhasi	nm0006982	237
Adoor Bhasi	nm0006982	169
Bahadur	nm0046850	169
Jayabharati	nm0419653	162
Adoor Bhasi	nm0006982	162
Matsunosuke Onoe	nm0648803	147
Kijaku Ôtani	nm2082516	147
Matsunosuke Onoe	nm0648803	126
Kitsuraku Arashi	nm2373718	126
Adoor Bhasi	nm0006982	122
Thikkurisi Sukuma...	nm0619779	122
Suminojo Ichikawa	nm2077739	113
Kitsuraku Arashi	nm2373718	113
Kijaku Ôtani	nm2082516	113
Matsunosuke Onoe	nm0648803	113
Jayabharati	nm0419653	109
Bahadur	nm0046850	109
Panchito	nm0659173	103
Dolphy	nm1006879	103

Figure 8. Frequent pairs actors - FP Growth algorithm

In the most frequent pairs, I found few of the frequent singletons identified before. The algorithm was able to identify these pairs, listed from the most frequent one: 1. Prem Nazir - Adoor Bhasi, 2. Adoor Bhasi - Bahadur, 3. Jayabharati - Adoor Bhasi, 4. Matsunosuke Onoe - Kijaku Ôtani, 5. Matsunosuke Onoe - Kitsuraku Arashi etc.
With this threshold, it was possible also easily to retrieve the most frequent triples and quadruples. Undoubtedly the number of quadruples identified is less than the triples of frequent actors: 658 triples (example of frequent triple: Kitsuraku Arashi - Matsunosuke Onoe - Kijaku Ôtani), while 315 quadruples (example of frequent quadruple: Matsunosuke Onoe - Kijaku Ôtani - Kitsuraku Arashi -Suminojo Ichikawa).

items	freq
[nm2373718, nm2082516, nm0648803]	112
[nm2077739, nm2082516, nm0648803]	100
[nm2077739, nm2373718, nm0648803]	95
[nm2077739, nm2373718, nm2082516]	87
[nm1770187, nm2082516, nm0648803]	80
[nm0419653, nm0046850, nm0006982]	75
[nm0619779, nm0623427, nm0006982]	74
[nm1770187, nm2373718, nm0648803]	70
[nm2384746, nm1698868, nm2366585]	69
[nm2077739, nm1770187, nm0648803]	64

only showing top 10 rows

NAME_actors	items	freq
Kitsuraku Arashi	nm2373718	112
Matsunosuke Onoe	nm0648803	112
Kijaku Ôtani	nm2082516	112
Suminojo Ichikawa	nm2077739	100
Kijaku Ôtani	nm2082516	100
Matsunosuke Onoe	nm0648803	100
Matsunosuke Onoe	nm0648803	95
Kitsuraku Arashi	nm2373718	95
Suminojo Ichikawa	nm2077739	95
Kijaku Ôtani	nm2082516	87
Kitsuraku Arashi	nm2373718	87
Suminojo Ichikawa	nm2077739	87
Kijaku Ôtani	nm2082516	80
Matsunosuke Onoe	nm0648803	80
Sen'nosuke Nakamura	nm1770187	80
Adoor Bhasi	nm0006982	75
Jayabharati	nm0419653	75
Bahadur	nm0046850	75
Adoor Bhasi	nm0006982	74
Thikkurisi Sukuma...	nm0619779	74

only showing top 20 rows

Figure 9. Frequent triples actors - FP Growth algorithm

items	freq
[nm2077739, nm2373718, nm2082516, nm0648803]	86
[nm1770187, nm2373718, nm2082516, nm0648803]	62
[nm2077739, nm1770187, nm2082516, nm0648803]	54
[nm2077739, nm1770187, nm2373718, nm0648803]	51
[nm2367854, nm2384746, nm1698868, nm2366585]	51
[nm2373151, nm2373718, nm2082516, nm0648803]	48
[nm1283907, nm2373718, nm2082516, nm0648803]	46
[nm2077739, nm1770187, nm2373718, nm2082516]	45
[nm2373151, nm2077739, nm2082516, nm0648803]	45
[nm2373151, nm2077739, nm2373718, nm0648803]	44

only showing top 10 rows

NAME_actors	items	freq
Matsunosuke Onoe	nm0648803	86
Kijaku Ôtani	nm2082516	86
Kitsuraku Arashi	nm2373718	86
Suminojo Ichikawa	nm2077739	86
Kitsuraku Arashi	nm2373718	62
Matsunosuke Onoe	nm0648803	62
Kijaku Ôtani	nm2082516	62
Sen'nosuke Nakamura	nm1770187	62
Sen'nosuke Nakamura	nm1770187	54
Suminojo Ichikawa	nm2077739	54
Matsunosuke Onoe	nm0648803	54
Kijaku Ôtani	nm2082516	54
Matsunosuke Onoe	nm0648803	51
Hôshô Bandô	nm2384746	51
Shôzô Arashi	nm2367854	51
Kitsuraku Arashi	nm2373718	51
Sen'nosuke Nakamura	nm1770187	51
Ritoku Arashi	nm2366585	51
Suminojo Ichikawa	nm2077739	51
Enshô Jitsukawa	nm1698868	51

only showing top 20 rows

Figure 10. Frequent quadruples actors - FP Growth algorithm

FPGrowth allows us to understand the relationships between frequent items by providing "associationRules". By identifying "antecedent" and "consequent" items, it is possible to retrieve the estimates that characterize a relationship: confidence, lift, support. To analyze the association between items, we should consider theory behind two measures of goodness:

- **Confidence** $Conf(I \to j) = \frac{Support(I \cup (j))}{Support(I)}$
- **Interest** $Int(I \to j) = Conf(I \to j) - \frac{Support(j)}{Totalbaskets}$

Briefly, the confidence measures the conditional probability of occurrence of consequent given the antecedent item, while the interest or the lift controls the support of consequent item, calculating the conditional probability of occurrence of consequent given antecedent.
The goal of the project was to retrieve the pairs of actors that should be considered frequent and their presence should have a positive effect on their collegues (lift > 1). An example of the pair identified by the FP growth, with the hyperparameter described previously, that has the highest confidence and lift is Moe Howard and Larry Fine.

antecedent	consequent	confidence	lift	support
[nm0002935]	[nm0004310]	1.0	32804.5	3.0483622673718545E-5

- Moe Howard and Larry Fine -

Figure 11. Results FP Growth algorithm with a minSupport at 0.00003

Looking also to the results, with lift in ascending order, it is possible to see that all couples retrieved by the algorithm present a positive lift.
Take in mind that working with FP growth helps us to preserve all the information of the dataset, by the compression FP's tree. This could be a disadvantage when dealing with massive database, while the algorithm will not fit in the shared memory.

I decided also to evaluate the FP growth algorithm with a more restrictive threshold. This threshold will be similar to that used in the Apriori algorithm, that I present later. By setting as at 0.00033 the minsupport, I decided to consider frequent an actor or a pair that played at least 130 movies. Due to this restrictive threshold, the algorithm was able only to retrieve just a single frequent pair. I take for granted that, for the case of frequent singleton, the result of FP growth algorithm also with this restrictive threshold is the same identified previously (in Figure 7).

antecedent	consequent	confidence	lift	support
[nm2082516]	[nm0648803]	0.9130434782608695	636.1472874182377	3.7342437775305217E-4

- Matsunosuke Onoe and Kijaku Ôtani -

Figure 12. Results FP Growth algorithm with a minSupport at 0.00033

The algorithm was able to evaluate the relationship between these two actors, above listed, thought their confidence and lift. The probability of Matsunosuke Onoe to be present, given the presence of Kijaku Ôtani is 0.91. The lift of this pair is at 636.

To evaluate the findings that FP growth retrieved, I applied the Apriori algorithm. The Apriori algorithm exploits the monotonicity property: if an itemset is frequent, then all of its subsets must also be frequent. By the anti-monotone property of support, we can perform support-based pruning and it is possible to retrieve frequent candidates, pairs, triples and quadruples by adding steps into the process. We start by selecting the right candidates, which have the supports higher than the threshold we impose, then we need to filter and to produce frequent singleton. By the frequent singleton it is possible to create the candidates pairs, then filtering... and so on. By exploiting the advantages of Spark, I was able with the map and reduce step to scaling up the data size and to retrieve the desired results. To make a comparison with the FP growth model, I decided to impose as a threshold (min support) at 130, so I considered frequent actors or pairs that played at least 130 movies. The Apriori algorithm was able to identify 324 single frequent items. The following listed actors are the result of the algorithm, from the most frequent to the less.

actor	frequent
[nm0103977]	798
[nm0006982]	585
[nm0648803]	565
[nm0305182]	506
[nm0623427]	438
[nm0793813]	411
[nm0246703]	391
[nm0619107]	387
[nm0007123]	381
[nm7390393]	355

only showing top 10 rows

NAME_actors	actor
Brahmanandam	nm0103977
Adoor Bhasi	nm0006982
Matsunosuke Onoe	nm0648803
Eddie Garcia	nm0305182
Prem Nazir	nm0623427
Sung-il Shin	nm0793813
Paquito Diaz	nm0246703
Masayoshi Nogami	nm0619107
Mammootty	nm0007123
Aachi Manorama	nm7390393
Bahadur	nm0046850
Mohanlal	nm0482320
Mithun Chakraborty	nm0149822
Shivaji Ganesan	nm0304262
Sultan Rahi	nm0706691
Nagesh	nm0619309
Shakti Kapoor	nm0007106
Pandharibai	nm0659250
Tom Byron	nm0001000
Jayabharati	nm0419653

only showing top 20 rows

Figure 11. Frequent actors - Apriori algorithm

The frequent actors, identified by Apriori algorithm, are the same and in the same order to the answer of Question 2 in the exploratory analysis and to the frequent singleton results identified by FP growth algorithm. Due to the threshold imposed, the Apriori algorithm was able to identify only one pair as frequent. It was no possible to retrieve more than this frequent pair, listed in Figure 12.

```
+------------------------+--------+
|pairs                   |frequent|
+------------------------+--------+
|[{nm2082516, nm0648803}]|146     |
+------------------------+--------+
```

```
+----------------+---------+
|     NAME_actors|ID_actors|
+----------------+---------+
|Matsunosuke Onoe|nm0648803|
|    Kijaku Ôtani|nm2082516|
+----------------+---------+
```

Figure 12. Frequent pairs - Apriori algorithm

The most frequent pair retrived by the Apriori is the same that was identified in the FP growth model: Matsunosuke Onoe and Kijaku Ôtani. After this analysis, I decided to estimate the association rule with the common estimate listed before. The confidence of the pair, so the probability of Matsunosuke Onoe to be present, given the presence of Kijaku Ôtani is 0,91. The interest was estimated 634,78, a positive relationship: this means that the occurrence of the Kijaku Ôtani has a positive effect on the occurrence of the Matsunosuke Onoe.
To check if the pair is truly frequent and to check if the two algorithms were able to achieve the goal of the project, I retrieved from the dataset the movies played by the two actors. In the dataset it was possible to understand that the pair have played together 146 movies (the same frequent value for the pair retrieved by the two algorithms) and below, it is possible to have a list of movies done.

```
+----------------------+
|TITLE_movie           |
+----------------------+
|Nidaime jiraiya       |
|Onigoroshi juzô       |
|Gôsho Kingôro         |
|Moyuru uzumaki sanbu  |
|Araki Mataemon        |
|Kantô shichinin otoko |
|Chûshingura           |
|Yakko no kôsan        |
|Meitô takada matabei  |
|Benten kozo           |
|Go henge kikûmatsu    |
|Yoshioka Kanefusa     |
|Gôsho no Gorozô       |
|Kana tehon chûsinghura|
|Takenâka hanbei       |
+----------------------+
only showing top 15 rows
```

Figure 13. List of movies played by Matsunosuke Onoe and Kijaku Ôtani, only 15 rows

If you are curious about the movies or the actors, it is possible to retrieve lots of other information by joining this results with the other csv presented in the beginning of the analysis.

4 Conclusions

Based on performing market basket analysis, this paper analyzes how to find frequent sets of items appearing in several baskets applying two different algorithms. I started the analysis retrieving the dataset on movies and actors from IMBD dataset, on Kaggle. After a pre processing step and an exploratory analysis, I applied the Apriori and FPGrowth algorithms to achieve the goal. By exploiting Apache Spark environment, I was able to identify the most frequent actors, pairs, triples and quadruples of actors in mostly movies of the dataset used. Both algorithms achieve the same results in different ways by managing to scale up the massive dataset used. In conclusion, even if the two algorithms are different, they found the same results when we impose a restrictive threshold. When we imposed it, a pair should be considered frequent when played together at least 130 movies; the two algorithms identified as frequent pair: Matsunosuke Onoe and Kijaku Ôtani, that played together 146 movies. The confidence and their lift better explain the positive relationship between the two actors and their estimates in the two algorithms confirmed the result. In the project, it is underlined how crucial is the choice of the threshold to consider an item as frequent or not. The two algorithms behave in two different ways and their difference need to be taken into account: Apriori finds the frequent itemsets with candidate generation, while FP Growth algorithm discovers the frequent itemsets without generating candidates. In the beginning of the analysis I started with a less restrictive threshold (frequent was an item, pair, triples that played 10 movies) and then I used a threshold of 130 movies. Due to the choice of a more restrictive threshold or less can let us achieve different results: it is possible to achieve more triples, quadruples if the threshold is less restrictive. By a less restrictive threshold the two algorithms can achieve different results. For the goal of the project, I checked the quality of the frequent pair, retrieved with a restrictive threshold by the two algorithms and I confirmed the goodness of result of this Market Basket Analysis.

5 Appendix

In the next page, you will find the whole code of the project.

Algorithms for Massive Dataset Project

Giorgia Mazzi

Load the **dataset** from **Kaggle**

```
from google.colab import files #upload kaggle.json, containing API
files.upload()
```

Scegli file kaggle.json

- **kaggle.json**(application/json) - 68 bytes, last modified: 2/12/2021 - 100% done

```
Saving kaggle.json to kaggle.json
{'kaggle.json': b'{"username":"giorgiamazzi","key":"3e4bdd6972758babbb3327722d0f1f60
```

```
! pip install -q kaggle
```

```
! mkdir -p ~/.kaggle #make directory with name Kaggle
```

```
! mv kaggle.json ~/.kaggle/ #move json file into the directory created
```

```
! chmod 600 ~/.kaggle/kaggle.json #give permission to this file
```

```
! kaggle datasets download -d ashirwadsangwan/imdb-dataset
```

```
Downloading imdb-dataset.zip to /content
100% 1.44G/1.44G [00:15<00:00, 125MB/s]
100% 1.44G/1.44G [00:15<00:00, 100MB/s]
```

```
!unzip imdb-dataset.zip
```

```
Archive:  imdb-dataset.zip
  inflating: name.basics.tsv.gz
  inflating: name.basics.tsv/name.basics.tsv
  inflating: title.akas.tsv.gz
  inflating: title.akas.tsv/title.akas.tsv
  inflating: title.basics.tsv.gz
  inflating: title.basics.tsv/title.basics.tsv
  inflating: title.principals.tsv.gz
  inflating: title.principals.tsv/title.principals.tsv
  inflating: title.ratings.tsv.gz
  inflating: title.ratings.tsv/title.ratings.tsv
```

Spark Setup

```
!apt-get install openjdk-8-jdk-headless -qq > /dev/null  #install Java
!wget -q https://apache.osuosl.org/spark/spark-3.2.0/spark-3.2.0-bin-hadoop2.7.tgz #downlo
!tar xf spark-3.2.0-bin-hadoop2.7.tgz #unzip folder
```

```
!pip install -q findspark #locate Spark on the system
import findspark
findspark.init("spark-3.2.0-bin-hadoop2.7")
```

```
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64" #set the environmental path
os.environ["SPARK_HOME"] = "/content/spark-3.2.0-bin-hadoop2.7"
```

```
!pip install pyspark
import pyspark
from pyspark import SparkConf, SparkContext
from pyspark.sql import SparkSession
```

```
Collecting pyspark
  Downloading pyspark-3.2.0.tar.gz (281.3 MB)
     |████████████████████████████████| 281.3 MB 40 kB/s
Collecting py4j==0.10.9.2
  Downloading py4j-0.10.9.2-py2.py3-none-any.whl (198 kB)
     |████████████████████████████████| 198 kB 64.5 MB/s
Building wheels for collected packages: pyspark
  Building wheel for pyspark (setup.py) ... done
  Created wheel for pyspark: filename=pyspark-3.2.0-py2.py3-none-any.whl size=281805
  Stored in directory: /root/.cache/pip/wheels/0b/de/d2/9be5d59d7331c6c2a7c1b6d1a4f4
Successfully built pyspark
Installing collected packages: py4j, pyspark
Successfully installed py4j-0.10.9.2 pyspark-3.2.0
```

```
spark = SparkSession.builder.getOrCreate()
```

```
sc = spark.sparkContext
```

```
!pip install python-utils
import numpy as np
import pandas as pd
import seaborn as sns
```

```
Requirement already satisfied: python-utils in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (from p
```

Dataset Inspection

Tables' contents detailed [url of dataset]

```
infofornames = spark.read.csv("/content/name.basics.tsv", sep=r'\t', header=True)
infotitleakas = spark.read.csv("/content/title.akas.tsv", sep=r'\t', header=True)
infotitlebasics = spark.read.csv("/content/title.basics.tsv", sep=r'\t', header=True)
```

```
castpertitles = spark.read.csv("/content/title.principals.tsv", sep=r'\t', header=True)
ratingandvotes = spark.read.csv("/content/title.ratings.tsv", sep=r'\t', header=True)
```

```
infofornames.createOrReplaceTempView("infofornames")
infotitleakas.createOrReplaceTempView("infotitleakas")
infotitlebasics.createOrReplaceTempView("infotitlebasics")
castpertitles.createOrReplaceTempView("castpertitles")
ratingandvotes.createOrReplaceTempView("ratingandvotes")
```

```
infofornames.show(5)
```

```
+---------+---------------+---------+---------+--------------------+----------------
|   nconst|    primaryName|birthYear|deathYear|   primaryProfession|      knownForTi
+---------+---------------+---------+---------+--------------------+----------------
|nm0000001|   Fred Astaire|     1899|     1987|soundtrack,actor,...|tt0050419,tt0053
|nm0000002|  Lauren Bacall|     1924|     2014|  actress,soundtrack|tt0071877,tt0117
|nm0000003|Brigitte Bardot|     1934|       \N|actress,soundtrac...|tt0054452,tt0049
|nm0000004|   John Belushi|     1949|     1982|actor,writer,soun...|tt0077975,tt0072
|nm0000005| Ingmar Bergman|     1918|     2007|writer,director,a...|tt0069467,tt0050
+---------+---------------+---------+---------+--------------------+----------------
only showing top 5 rows
```

```
infotitleakas.show(5)
```

```
+---------+--------+--------------------+------+--------+-----------+----------+----
|  titleId|ordering|               title|region|language|      types|attributes|isOr
+---------+--------+--------------------+------+--------+-----------+----------+----
|tt0000001|       1|Carmencita - span...|    HU|      \N|imdbDisplay|        \N|
|tt0000001|       2|          Καρμενσίτα|    GR|      \N|         \N|        \N|
|tt0000001|       3|          Карменсита|    RU|      \N|         \N|        \N|
|tt0000001|       4|          Carmencita|    US|      \N|         \N|        \N|
|tt0000001|       5|          Carmencita|    \N|      \N|   original|        \N|
+---------+--------+--------------------+------+--------+-----------+----------+----
only showing top 5 rows
```

```
infotitlebasics.show(5)
```

```
+---------+---------+--------------------+--------------------+-------+---------+---
|   tconst|titleType|        primaryTitle|       originalTitle|isAdult|startYear|end
+---------+---------+--------------------+--------------------+-------+---------+---
|tt0000001|    short|          Carmencita|          Carmencita|      0|     1894|
|tt0000002|    short|Le clown et ses c...|Le clown et ses c...|      0|     1892|
|tt0000003|    short|      Pauvre Pierrot|      Pauvre Pierrot|      0|     1892|
|tt0000004|    short|         Un bon bock|         Un bon bock|      0|     1892|
|tt0000005|    short|    Blacksmith Scene|    Blacksmith Scene|      0|     1893|
+---------+---------+--------------------+--------------------+-------+---------+---
only showing top 5 rows
```

```
castpertitles.show(5)
```

```
+---------+--------+---------+---------------+--------------------+-----------+
|   tconst|ordering|   nconst|       category|                 job| characters|
+---------+--------+---------+---------------+--------------------+-----------+
|tt0000001|       1|nm1588970|           self|                  \N|["Herself"]|
|tt0000001|       2|nm0005690|       director|                  \N|         \N|
|tt0000001|       3|nm0374658|cinematographer|director of photo...|         \N|
|tt0000002|       1|nm0721526|       director|                  \N|         \N|
|tt0000002|       2|nm1335271|       composer|                  \N|         \N|
+---------+--------+---------+---------------+--------------------+-----------+
only showing top 5 rows
```

```
ratingandvotes.show(5)
```

```
+---------+-------------+--------+
|   tconst|averageRating|numVotes|
+---------+-------------+--------+
|tt0000001|          5.6|    1550|
|tt0000002|          6.1|     186|
|tt0000003|          6.5|    1207|
|tt0000004|          6.2|     113|
|tt0000005|          6.1|    1934|
+---------+-------------+--------+
only showing top 5 rows
```

Exploratory Data and Data Cleaning

For my project, i'd like to implement a system that helps us to find frequent itemsets. Market-basket model of data is a way to represent and describe relationships between objects. In this project I'll use as items (actors) and as baskets (movies). So i want to discover the most frequent actors or actress in movies. To do the analysis I selected only three csv provided : **infofornames**, **castpertitles**, **infotitlebasics**.

```
infofornames = infofornames.filter((infofornames.primaryProfession == 'actor')|(infofornam
infofornames.show(5)
```

```
+---------+----------------+---------+---------+-----------------+-----------------
|   nconst|     primaryName|birthYear|deathYear|primaryProfession|      knownForTit
+---------+----------------+---------+---------+-----------------+-----------------
|nm0000084|         Li Gong|     1965|       \N|          actress|tt0473444,tt01016
|nm0000109|   Yasmine Bleeth|     1968|       \N|          actress|tt0131857,tt01152
|nm0000124|Jennifer Connelly|     1970|       \N|          actress|tt0315983,tt01800
|nm0000143|    Erika Eleniak|     1969|       \N|          actress|tt0083866,tt00947
|nm0000157|   Linda Hamilton|     1956|       \N|          actress|tt0103064,tt64508
+---------+----------------+---------+---------+-----------------+-----------------
only showing top 5 rows
```

```
infofornames = infofornames.select(['primaryName', 'nconst', 'primaryProfession'])
infofornames = infofornames.withColumnRenamed("primaryName", "Name_actors") \
```

```
    .withColumnRenamed("nconst","Id_actors") \
    .withColumnRenamed("primaryProfession","role")
infofornames.printSchema()
infofornames.show(5)
```

```
root
 |-- Name_actors: string (nullable = true)
 |-- Id_actors: string (nullable = true)
 |-- role: string (nullable = true)

+-----------------+---------+-------+
|      Name_actors|Id_actors|   role|
+-----------------+---------+-------+
|          Li Gong|nm0000084|actress|
|   Yasmine Bleeth|nm0000109|actress|
|Jennifer Connelly|nm0000124|actress|
|    Erika Eleniak|nm0000143|actress|
|   Linda Hamilton|nm0000157|actress|
+-----------------+---------+-------+
only showing top 5 rows
```

```
spark.sql('''SELECT count(distinct(infofornames.nconst))
             FROM infofornames
             WHERE (infofornames.primaryProfession = 'actor') OR (infofornames.primaryPro
```

```
+----------------------+
|count(DISTINCT nconst)|
+----------------------+
|               2914287|
+----------------------+
```

In this way I see that lots of actors that I retrieve (the actors that have played at least one time as actors), they did also other jobs in other films.

```
spark.sql('''SELECT distinct(infofornames.primaryProfession)
             FROM infofornames''').show(20, False)
```

```
+---------------------------------------------------------------+
|primaryProfession                                              |
+---------------------------------------------------------------+
|actress,producer,production_manager                            |
|producer,miscellaneous,location_management                     |
|art_department,miscellaneous,assistant_director                |
|actor,producer,assistant_director                              |
|assistant_director,camera_department,transportation_department|
|assistant_director,actress,director                            |
|miscellaneous,actor,location_management                        |
|miscellaneous,casting_department,actress                       |
|cinematographer,camera_department,talent_agent                 |
|make_up_department,actor,costume_designer                      |
|miscellaneous,camera_department,script_department              |
|camera_department,special_effects,visual_effects               |
|production_designer,sound_department                           |
|director,writer,casting_director                               |
```

```
|art_department,art_director,assistant_director                |
|casting_department,miscellaneous,make_up_department           |
|casting_department,actor,camera_department                    |
|editor,producer,camera_department                             |
|producer,director,animation_department                        |
|sound_department,producer,assistant_director                  |
+--------------------------------------------------------------+
only showing top 20 rows
```

```
spark.sql('''SELECT distinct(castpertitles.category)
              FROM castpertitles''').show()
```

```
+-------------------+
|           category|
+-------------------+
|            actress|
|           producer|
|             writer|
|           composer|
|           director|
|               self|
|              actor|
|             editor|
|    cinematographer|
|      archive_sound|
|production_designer|
|    archive_footage|
+-------------------+
```

```
castpertitles = castpertitles.filter((castpertitles.category == 'actor')|(castpertitles.ca
```

```
castpertitles = castpertitles.select(['tconst', 'nconst', 'category'])
castpertitles = castpertitles.withColumnRenamed("nconst", "Id_actors") \
    .withColumnRenamed("tconst","Id_movie") \
    .withColumnRenamed("category","role")
castpertitles.printSchema()
castpertitles.show(5)
```

```
root
 |-- Id_movie: string (nullable = true)
 |-- Id_actors: string (nullable = true)
 |-- role: string (nullable = true)

+---------+---------+-----+
| Id_movie|Id_actors| role|
+---------+---------+-----+
|tt0000005|nm0443482|actor|
|tt0000005|nm0653042|actor|
|tt0000007|nm0179163|actor|
|tt0000007|nm0183947|actor|
|tt0000008|nm0653028|actor|
+---------+---------+-----+
only showing top 5 rows
```

```
spark.sql('''SELECT count(distinct(castpertitles.nconst))
              FROM castpertitles
              WHERE (castpertitles.category = 'actor') OR (castpertitles.category = 'actre
```

```
+----------------------+
|count(DISTINCT nconst)|
+----------------------+
|               1867043|
+----------------------+
```

```
infotitlebasics = infotitlebasics.filter(infotitlebasics.titleType == 'movie')
```

```
infotitlebasics = infotitlebasics.select(['primaryTitle', 'tconst', 'titleType'])
infotitlebasics = infotitlebasics.withColumnRenamed("primaryTitle", "Name_movie") \
    .withColumnRenamed("tconst","Id_movie") \
    .withColumnRenamed("titleType","movie")
infotitlebasics.printSchema()
infotitlebasics.show(5)
```

```
root
 |-- Name_movie: string (nullable = true)
 |-- Id_movie: string (nullable = true)
 |-- movie: string (nullable = true)

+--------------------+---------+-----+
|          Name_movie| Id_movie|movie|
+--------------------+---------+-----+
|          Miss Jerry|tt0000009|movie|
|The Corbett-Fitzs...|tt0000147|movie|
|Soldiers of the C...|tt0000335|movie|
|            Bohemios|tt0000502|movie|
|The Story of the ...|tt0000574|movie|
+--------------------+---------+-----+
only showing top 5 rows
```

Dataset for the Market Basket Analysis

```
infotitle = spark.sql('''SELECT infotitlebasics.tconst AS ID_movie, infotitlebasics.primar
                                FROM infotitlebasics
                                WHERE 1=1
                                AND (infotitlebasics.titleType == 'movie')''')
cast = spark.sql('''SELECT castpertitles.tconst as ID_movie, castpertitles.nconst AS ID_ac
                              FROM castpertitles
                              WHERE 1=1
                              AND (castpertitles.category = 'actor') OR (castpertitles.cat
```

```
infotitle.createOrReplaceTempView("infotitle")
cast.createOrReplaceTempView("cast")
```

```
data = spark.sql('''SELECT infotitle.ID_movie as ID_movie, infotitle.TITLE_movie as TITLE_
                    FROM (infotitle INNER JOIN cast ON infotitle.ID_movie = cast.ID_movie)
                    INNER JOIN infofornames on cast.ID_actors = infofornames.nconst''')
```

```
data.createOrReplaceTempView("data")
```

```
data.show(5)
```

```
+---------+--------------------+--------------+---------+------------+-------------+
| ID_movie|         TITLE_movie|category_movie|ID_actors| NAME_actors|Role_in_movie|
+---------+--------------------+--------------+---------+------------+-------------+
|tt0077621|         Goin' South|         movie|nm0000004|John Belushi|        actor|
|tt0082801|           Neighbors|         movie|nm0000004|John Belushi|        actor|
|tt0077975|National Lampoon'...|         movie|nm0000004|John Belushi|        actor|
|tt0078723|                1941|         movie|nm0000004|John Belushi|        actor|
|tt0080455|  The Blues Brothers|         movie|nm0000004|John Belushi|        actor|
+---------+--------------------+--------------+---------+------------+-------------+
only showing top 5 rows
```

```
print(data.count()) #1.692.939 rows
```

```
1692939
```

```
spark.sql('''SELECT category_movie, COUNT(*)
             FROM data
             GROUP BY category_movie''').show() #Check: is there only movies?
```

```
+--------------+--------+
|category_movie|count(1)|
+--------------+--------+
|         movie| 1692939|
+--------------+--------+
```

```
spark.sql('''SELECT Role_in_movie, COUNT(*)
             FROM data
             GROUP BY Role_in_movie''').show() #Check: how many actors and actress?
```

```
+-------------+--------+
|Role_in_movie|count(1)|
+-------------+--------+
|      actress|  637347|
|        actor| 1055592|
+-------------+--------+
```

Checking Null Values

```
spark.sql('''SELECT COUNT(*)
```

```
            FROM data
            WHERE (ID_movie = '/N') OR (TITLE_movie = '/N') OR (ID_actors = '/N') OR (NA
```

```
+--------+
|count(1)|
+--------+
|       0|
+--------+
```

Question 1: *Which is the number of actors or actress per film? How many movies have the same number of participants in the cast?*

```
groupbymovies = spark.sql('''SELECT ID_movie, TITLE_movie, COUNT(ID_actors) AS number_acto
                    FROM data
                    GROUP BY ID_movie, TITLE_movie
                    ORDER BY number_actors DESC''')
```

```
groupbymovies.show(20)
```

```
+----------+--------------------+-------------+
|  ID_movie|         TITLE_movie|number_actors|
+----------+--------------------+-------------+
| tt2543584|        One Day Here|           10|
| tt0160071|Bottom Dweller 5:...|           10|
| tt6735094|           Sathriyan|           10|
| tt3813084|         Jalta Badan|           10|
| tt7153466|sti xagi kai sti ...|           10|
| tt6446750|     Khalik Fe Hallak|          10|
| tt7128038|Exterminator. Fac...|           10|
| tt7440016|           With Love|           10|
| tt7618402|   Change of Gangster|          10|
| tt7162426|     Giafka portokali|           10|
| tt0280055|Putus sudah kasih...|           10|
| tt6764552| Permanent Reminders|           10|
| tt5789676|              Stab 7|           10|
| tt5878312|CHIC: The One Yea...|           10|
| tt0393168|Como agua pa' lon...|           10|
| tt7909340|             Witness|           10|
| tt1641918|       All or Nothing|           10|
|tt11046112|                Haul|           10|
| tt0422886|      Onyong Majikero|           10|
| tt0349853|   The Myster General|           10|
+----------+--------------------+-------------+
only showing top 20 rows
```

```
from pyspark.sql import functions as f
```

```
moviesbycast = groupbymovies.groupBy('number_actors').agg(f.collect_set('ID_movie').alias(
```

```
moviesbycast.show()
```

```
+-------------+--------------------+
|number_actors|              movies|
+-------------+--------------------+
|           10|[tt7368790, tt361...|
|            9|[tt5134236, tt352...|
|            8|[tt5204858, tt039...|
|            7|[tt0069205, tt007...|
|            6|[tt5229070, tt019...|
|            5|[tt1330991, tt032...|
|            4|[tt8917774, tt154...|
|            3|[tt11011054, tt23...|
|            2|[tt9489044, tt341...|
|            1|[tt0292703, tt933...|
+-------------+--------------------+
```

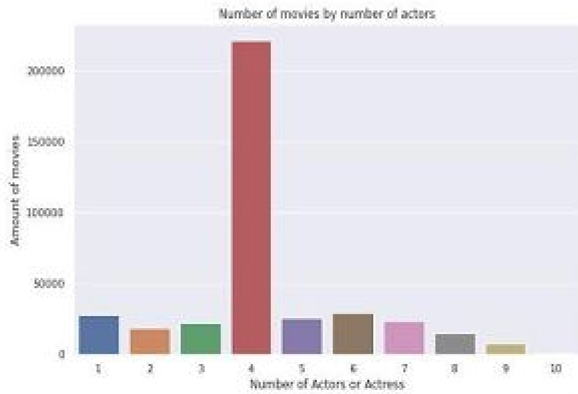
```
from pyspark.sql.functions import size, col
```

```
moviesbycast = moviesbycast.withColumn("movies",size(col("movies")))
```

```
moviesbycast.show()
```

```
+-------------+------+
|number_actors|movies|
+-------------+------+
|           10|  1159|
|            9|  8017|
|            8| 15239|
|            7| 23617|
|            6| 29313|
|            5| 25577|
|            4|221462|
|            3| 22102|
|            2| 18880|
|            1| 28288|
+-------------+------+
```

```
sns.set_style("darkgrid")
sns.set(rc={"figure.figsize":(10,6)})
ax = sns.barplot(x = "number_actors", y = "movies", data = moviesbycast.toPandas())
ax.set(title = "Number of movies by number of actors", xlabel = "Number of Actors or Actre
```

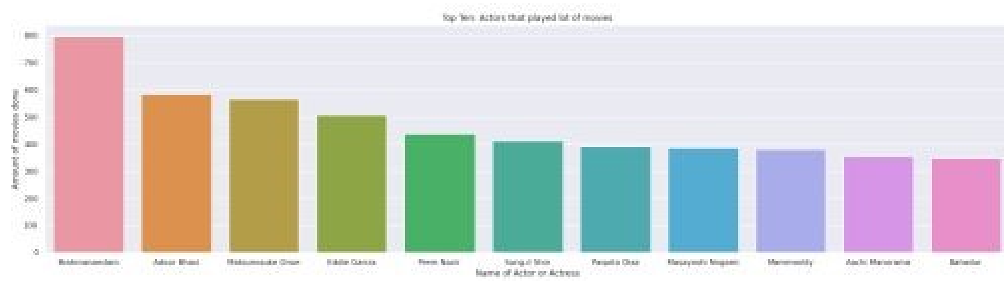


The dataset presents a huge amount of movies with four actors in the cast, while very little number of films with ten actors.

Question 2: *Who are the TOP ten actors that played more movies?*

```
groupbyactors = spark.sql('''SELECT NAME_actors, COUNT(ID_movie) AS number_movies
                           FROM data
                           GROUP BY NAME_actors
                           ORDER BY number_movies DESC
                           LIMIT 11''')
```

```
sns.set_style("darkgrid")
sns.set(rc={"figure.figsize":(25,6)})
ax = sns.barplot(x = "NAME_actors", y = "number_movies", data = groupbyactors.toPandas())
ax.set(title = "Top Ten: Actors that played lot of movies", xlabel = "Name of Actor or Act
```



Baskets for Market Basket Analysis

```
baskets_id= data.groupBy('ID_movie').agg(f.collect_set('ID_actors').alias('actors'))
baskets_id.createOrReplaceTempView('baskets_id')
```

```
baskets_id.show(5)
```

```
+---------+--------------------+
| ID_movie|              actors|
+---------+--------------------+
|tt0000335|[nm1012612, nm067...|
|tt0000502|[nm0252720, nm021...|
|tt0000630|         [nm0624446]|
|tt0000676|[nm0140054, nm009...|
|tt0000793|         [nm0691995]|
+---------+--------------------+
only showing top 5 rows
```

```
spark.sql('''SELECT COUNT('ID_movie')
              FROM baskets_id ''').show() #393.654 movies
```

```
+---------------+
|count(ID_movie)|
+---------------+
|         393654|
+---------------+
```

FP growth

The FP-growth algorithm - explored in the paper Han et al., Mining frequent patterns without candidate generation - as Apriori algorithm tries to calculate item frequencies and identify frequent items. Different from it with the FP-tree structure that encode transactions without generating candidate sets explicitly.

```
from pyspark.ml.fpm import FPGrowth
```

```
FP = FPGrowth(itemsCol="actors", minSupport=0.00003)  #to be frequent more than 10 movies
```

```
model = FP.fit(baskets_id) #it takes 3:40 minutes
```

```
model.freqItemsets.show()
items_set = model.freqItemsets
```

```
spark-3.2.0-bin-hadoop2.7/python/pyspark/sql/context.py:127: FutureWarning: Deprecat
  FutureWarning
+-----------+----+
|      items|freq|
+-----------+----+
```

```
|[nm1700980]|  20|
|[nm0203836]|  46|
|[nm0878546]|  31|
|[nm0226773]|  33|
|[nm0014122]|  20|
|[nm1816849]|  51|
|[nm1954434]|  36|
|[nm1181575]|  15|
|[nm5999008]|  12|
|[nm2761937]|  13|
|[nm0839293]|  15|
|[nm1293253]|  15|
|[nm1122888]|  12|
|[nm0435919]|  15|
|[nm2846617]|  16|
|[nm3156758]|  13|
|[nm0756956]|  12|
|[nm0564691]|  14|
|[nm0952104]|  23|
|[nm0599973]|  20|
+-----------+----+
only showing top 20 rows
```

```
items_set.createOrReplaceTempView("items_set")
```

```
spark.sql('''SELECT items, freq
             FROM items_set
             WHERE size(items) = 1
             ORDER BY freq DESC''').show(10, False)
```

```
+-----------+----+
|items      |freq|
+-----------+----+
|[nm0103977]|798 |
|[nm0006982]|585 |
|[nm0648803]|565 |
|[nm0305182]|506 |
|[nm0623427]|438 |
|[nm0793813]|411 |
|[nm0246703]|391 |
|[nm0619107]|387 |
|[nm0007123]|381 |
|[nm7390393]|355 |
+-----------+----+
only showing top 10 rows
```

```
spark.sql('''SELECT count(items)
             FROM items_set
             WHERE size(items) = 1''').show()
```

```
+------------+
|count(items)|
+------------+
```

```
|       24054|
+------------+
```

```
singleactors = spark.sql('''SELECT items, freq
                            FROM items_set
                            WHERE size(items) = 1
                            ORDER BY freq DESC
                            LIMIT 10''')
singleactors.printSchema()
```

```
root
 |-- items: array (nullable = false)
 |    |-- element: string (containsNull = false)
 |-- freq: long (nullable = false)
```

```
namesingleactors_data = spark.sql('''SELECT DISTINCT(ID_actors), NAME_actors
                                     FROM data''')
```

```
from pyspark.sql.functions import concat_ws
```

```
singleactors = singleactors.withColumn("items", concat_ws(" ",col("items")))
singleactors.printSchema()
singleactors.createOrReplaceTempView("singleactors")
```

```
root
 |-- items: string (nullable = false)
 |-- freq: long (nullable = false)
```

```
namesingleactors_data.createOrReplaceTempView("namesingleactors_data")
```

```
spark.sql('''SELECT namesingleactors_data.NAME_actors, singleactors.items, singleactors.fr
              FROM singleactors INNER JOIN namesingleactors_data ON singleactors.items = r
              ORDER BY singleactors.freq DESC''').show()
```

```
+----------------+---------+----+
|     NAME_actors|    items|freq|
+----------------+---------+----+
|    Brahmanandam|nm0103977| 798|
|     Adoor Bhasi|nm0006982| 585|
|Matsunosuke Onoe|nm0648803| 565|
|    Eddie Garcia|nm0305182| 506|
|      Prem Nazir|nm0623427| 438|
|    Sung-il Shin|nm0793813| 411|
|    Paquito Diaz|nm0246703| 391|
|Masayoshi Nogami|nm0619107| 387|
|       Mammootty|nm0007123| 381|
|  Aachi Manorama|nm7390393| 355|
+----------------+---------+----+
```

The FP Growth algorithm was able to find the same name that we can see in the graph as solution of Question 2, in the same order.

```
spark.sql('''SELECT items, freq
             FROM items_set
             WHERE size(items) = 2
             ORDER BY freq DESC''').show(10, False)

     +----------------------+----+
     |items                 |freq|
     +----------------------+----+
     |[nm0623427, nm0006982]|237 |
     |[nm0046850, nm0006982]|169 |
     |[nm0419653, nm0006982]|162 |
     |[nm2082516, nm0648803]|147 |
     |[nm2373718, nm0648803]|126 |
     |[nm0619779, nm0006982]|122 |
     |[nm2373718, nm2082516]|113 |
     |[nm2077739, nm0648803]|113 |
     |[nm0419653, nm0046850]|109 |
     |[nm0659173, nm1006879]|103 |
     +----------------------+----+
     only showing top 10 rows
```

```
spark.sql('''SELECT count(items)
             FROM items_set
             WHERE size(items) = 2''').show()

     +------------+
     |count(items)|
     +------------+
     |        3443|
     +------------+
```

```
pairsactors = spark.sql('''SELECT items, freq
                           FROM items_set
                           WHERE size(items) = 2
                           ORDER BY freq DESC
                           LIMIT 10''')


from pyspark.sql.functions import explode


pairsactors = pairsactors.select(explode(pairsactors.items).alias("items"),"freq")


pairsactors = pairsactors.withColumn("items", concat_ws(" ", col("items")))
pairsactors.createOrReplaceTempView("pairsactors")
pairsactors.show()

     +--------+----+
```

```
|    items|freq|
+---------+----+
|nm0623427| 237|
|nm0006982| 237|
|nm0046850| 169|
|nm0006982| 169|
|nm0419653| 162|
|nm0006982| 162|
|nm2082516| 147|
|nm0648803| 147|
|nm2373718| 126|
|nm0648803| 126|
|nm0619779| 122|
|nm0006982| 122|
|nm2373718| 113|
|nm2082516| 113|
|nm2077739| 113|
|nm0648803| 113|
|nm0419653| 109|
|nm0046850| 109|
|nm0659173| 103|
|nm1006879| 103|
+---------+----+
```

```
spark.sql('''SELECT namesingleactors_data.NAME_actors, pairsactors.items, pairsactors.frec
             FROM pairsactors INNER JOIN namesingleactors_data ON pairsactors.items = nam
             ORDER BY pairsactors.freq DESC''').show()
```

```
+--------------------+---------+----+
|          NAME_actors|    items|freq|
+--------------------+---------+----+
|         Adoor Bhasi|nm0006982| 237|
|          Prem Nazir|nm0623427| 237|
|             Bahadur|nm0046850| 169|
|         Adoor Bhasi|nm0006982| 169|
|         Adoor Bhasi|nm0006982| 162|
|         Jayabharati|nm0419653| 162|
|    Matsunosuke Onoe|nm0648803| 147|
|        Kijaku Ôtani|nm2082516| 147|
|    Kitsuraku Arashi|nm2373718| 126|
|    Matsunosuke Onoe|nm0648803| 126|
|         Adoor Bhasi|nm0006982| 122|
|Thikkurisi Sukuma...|nm0619779| 122|
|    Kitsuraku Arashi|nm2373718| 113|
|   Suminojo Ichikawa|nm2077739| 113|
|    Matsunosuke Onoe|nm0648803| 113|
|        Kijaku Ôtani|nm2082516| 113|
|         Jayabharati|nm0419653| 109|
|             Bahadur|nm0046850| 109|
|            Panchito|nm0659173| 103|
|              Dolphy|nm1006879| 103|
+--------------------+---------+----+
```

```
spark.sql('''SELECT items, freq
             FROM items_set
```

```
            WHERE size(items) = 3
            ORDER BY freq DESC''').show(10, False)
```

```
    +--------------------------------+----+
    |items                           |freq|
    +--------------------------------+----+
    |[nm2373718, nm2082516, nm0648803]|112 |
    |[nm2077739, nm2082516, nm0648803]|100 |
    |[nm2077739, nm2373718, nm0648803]|95  |
    |[nm2077739, nm2373718, nm2082516]|87  |
    |[nm1770187, nm2082516, nm0648803]|80  |
    |[nm0419653, nm0046850, nm0006982]|75  |
    |[nm0619779, nm0623427, nm0006982]|74  |
    |[nm1770187, nm2373718, nm0648803]|70  |
    |[nm2384746, nm1698868, nm2366585]|69  |
    |[nm2077739, nm1770187, nm0648803]|64  |
    +--------------------------------+----+
    only showing top 10 rows
```

```
spark.sql('''SELECT count(items)
            FROM items_set
            WHERE size(items) = 3''').show()
```

```
    +------------+
    |count(items)|
    +------------+
    |         658|
    +------------+
```

```
triplesactors = spark.sql('''SELECT items, freq
            FROM items_set
            WHERE size(items) = 3
            ORDER BY freq DESC
            LIMIT 10''')
triplesactors = triplesactors.select(explode(triplesactors.items).alias("items"),"freq")
triplesactors = triplesactors.withColumn("items", concat_ws(" ", col("items")))
triplesactors.createOrReplaceTempView("triplesactors")
```

```
spark.sql('''SELECT namesingleactors_data.NAME_actors, triplesactors.items, triplesactors.
            FROM triplesactors INNER JOIN namesingleactors_data ON triplesactors.items =
            ORDER BY triplesactors.freq DESC''').show()
```

```
    +-------------------+---------+----+
    |         NAME_actors|    items|freq|
    +-------------------+---------+----+
    |    Matsunosuke Onoe|nm0648803| 112|
    |    Kitsuraku Arashi|nm2373718| 112|
    |        Kijaku Ôtani|nm2082516| 112|
    |   Suminojo Ichikawa|nm2077739| 100|
    |        Kijaku Ôtani|nm2082516| 100|
    |    Matsunosuke Onoe|nm0648803| 100|
    |    Matsunosuke Onoe|nm0648803|  95|
    |    Kitsuraku Arashi|nm2373718|  95|
    |   Suminojo Ichikawa|nm2077739|  95|
    |        Kijaku Ôtani|nm2082516|  87|
```

```
|    Kitsuraku Arashi|nm2373718|  87|
|   Suminojo Ichikawa|nm2077739|  87|
|    Matsunosuke Onoe|nm0648803|  80|
| Sen'nosuke Nakamura|nm1770187|  80|
|        Kijaku Ôtani|nm2082516|  80|
|         Adoor Bhasi|nm0006982|  75|
|             Bahadur|nm0046850|  75|
|         Jayabharati|nm0419653|  75|
|         Adoor Bhasi|nm0006982|  74|
|Thikkurisi Sukuma...|nm0619779|  74|
+--------------------+---------+----+
only showing top 20 rows
```

```
spark.sql('''SELECT items, freq
              FROM items_set
              WHERE size(items) = 4
              ORDER BY freq DESC''').show(10, False)
```

```
+--------------------------------------------+----+
|items                                       |freq|
+--------------------------------------------+----+
|[nm2077739, nm2373718, nm2082516, nm0648803]|86  |
|[nm1770187, nm2373718, nm2082516, nm0648803]|62  |
|[nm2077739, nm1770187, nm2082516, nm0648803]|54  |
|[nm2077739, nm1770187, nm2373718, nm0648803]|51  |
|[nm2367854, nm2384746, nm1698868, nm2366585]|51  |
|[nm2373151, nm2373718, nm2082516, nm0648803]|48  |
|[nm1283907, nm2373718, nm2082516, nm0648803]|46  |
|[nm2077739, nm1770187, nm2373718, nm2082516]|45  |
|[nm2373151, nm2077739, nm2082516, nm0648803]|45  |
|[nm2373151, nm2077739, nm2373718, nm0648803]|44  |
+--------------------------------------------+----+
only showing top 10 rows
```

```
spark.sql('''SELECT count(items)
              FROM items_set
              WHERE size(items) = 4''').show()
```

```
+------------+
|count(items)|
+------------+
|         315|
+------------+
```

```
quadruplesactors = spark.sql('''SELECT items, freq
              FROM items_set
              WHERE size(items) = 4
              ORDER BY freq DESC
              LIMIT 10''')
quadruplesactors = quadruplesactors.select(explode(quadruplesactors.items).alias("items"),
quadruplesactors = quadruplesactors.withColumn("items", concat_ws(" ", col("items")))
quadruplesactors.createOrReplaceTempView("quadruplesactors")
```

```
spark.sql('''SELECT namesingleactors_data.NAME_actors, quadruplesactors.items, quadruplesa
            FROM quadruplesactors INNER JOIN namesingleactors_data ON quadruplesactors.i
            ORDER BY quadruplesactors.freq DESC''').show()
```

```
+------------------+---------+----+
|        NAME_actors|    items|freq|
+------------------+---------+----+
|       Kijaku Ôtani|nm2082516|  86|
|  Suminojo Ichikawa|nm2077739|  86|
|   Matsunosuke Onoe|nm0648803|  86|
|   Kitsuraku Arashi|nm2373718|  86|
|   Kitsuraku Arashi|nm2373718|  62|
|   Matsunosuke Onoe|nm0648803|  62|
|       Kijaku Ôtani|nm2082516|  62|
|Sen'nosuke Nakamura|nm1770187|  62|
|Sen'nosuke Nakamura|nm1770187|  54|
|   Matsunosuke Onoe|nm0648803|  54|
|  Suminojo Ichikawa|nm2077739|  54|
|       Kijaku Ôtani|nm2082516|  54|
|Sen'nosuke Nakamura|nm1770187|  51|
|      Ritoku Arashi|nm2366585|  51|
|        Hôshô Bandô|nm2384746|  51|
|  Suminojo Ichikawa|nm2077739|  51|
|   Matsunosuke Onoe|nm0648803|  51|
|       Shôzô Arashi|nm2367854|  51|
|    Enshô Jitsukawa|nm1698868|  51|
|   Kitsuraku Arashi|nm2373718|  51|
+------------------+---------+----+
only showing top 20 rows
```

```
model.associationRules.show()
rules = model.associationRules
```

```
spark-3.2.0-bin-hadoop2.7/python/pyspark/sql/context.py:127: FutureWarning: Deprecat
  FutureWarning
+--------------------+-----------+------------------+------------------+------------
|          antecedent| consequent|        confidence|              lift|
+--------------------+-----------+------------------+------------------+------------
|[nm0931054, nm041...|[nm0001889]|               0.8|           9841.35|3.0483622673
|[nm1283907, nm064...|[nm2373718]|0.8909090909090909|2739.9213068181816|1.2447479258
|[nm1283907, nm064...|[nm2082516]|0.9272727272727272|2267.2336533032185|1.2955539636
|[nm0799982, nm080...|[nm0482121]|               1.0| 8946.681818181818|7.1128452905
|[nm2679281, nm029...|[nm2394215]|             0.875|3785.1346153846157|3.5564226452
|[nm2679281, nm029...|[nm0945427]|               1.0|1805.7522935779816|4.0644830231
|[nm4050725, nm167...|[nm0314775]|            0.9375|16775.028409090908|3.8104528342
|[nm4050725, nm167...|[nm2846621]|            0.9375| 19423.71710526316|3.8104528342
|         [nm3252391]|[nm3252185]|0.8571428571428571|28118.142857142855|3.0483622673
|[nm2077739, nm064...|[nm2373718]|0.8407079646017699|2585.5316648230087|2.4132867950
|[nm2077739, nm064...|[nm2082516]|0.8849557522123894| 2163.766283735503|2.5403018894
|         [nm0457112]|[nm0257951]|0.9230769230769231| 4910.444906444906|3.0483622673
|[nm0120381, nm081...|[nm0344655]|0.8421052631578947|10045.397129186604|4.0644830231
|[nm1283907, nm236...|[nm0648803]|               1.0| 696.7327433628318|4.8265735900
|[nm1283907, nm236...|[nm2082516]|0.8947368421052632|2187.6815952925795|4.3185132121
|         [nm0204068]|[nm0709315]|0.9473684210526315| 7171.834008097165|4.5725434010
|[nm2687024, nm242...|[nm0648803]|               1.0| 696.7327433628318|3.3023924563
|         [nm0909040]|[nm1259779]|0.8571428571428571| 9372.714285714286|7.6209056684
```

```
|         [nm0909040]|[nm0485226]|0.9142857142857143| 4863.678764478764|8.1289660463
|[nm2373151, nm242...|[nm0648803]|               1.0| 696.7327433628318|3.3023924563
+--------------------+-----------+------------------+------------------+------------
only showing top 20 rows
```

```
rules.createOrReplaceTempView("rules")
rules.sort(rules.lift.desc()).show(10, False)
descrules = rules.sort(rules.lift.desc())
descrules.createOrReplaceTempView("descrules")
```

```
+---------------------------------------------+-----------+------------------+-------
|antecedent                                   |consequent |confidence        |lift
+---------------------------------------------+-----------+------------------+-------
|[nm0004310]                                  |[nm0002935]|1.0               |32804.5
|[nm0002935]                                  |[nm0004310]|1.0               |32804.5
|[nm6774606]                                  |[nm6774610]|1.0               |28118.1
|[nm6774608, nm2811639, nm6774609, nm6774610]|[nm6774607]|1.0               |28118.1
|[nm2811639, nm6774610]                       |[nm6774607]|1.0               |28118.1
|[nm2811639, nm6774606]                       |[nm6774610]|1.0               |28118.1
|[nm2811639, nm6774610]                       |[nm6774606]|1.0               |28118.1
|[nm6774608, nm2811639, nm6774609, nm6774610]|[nm6774606]|1.0               |28118.1
|[nm2811639, nm6774610]                       |[nm6774608]|0.9285714285714286|28118.1
|[nm6774606]                                  |[nm2811639]|1.0               |28118.1
+---------------------------------------------+-----------+------------------+-------
only showing top 10 rows
```

```
firstrow_descrules = spark.sql('''SELECT *
                                  FROM descrules
                                  LIMIT 1''')
firstrow_descrules.show(1,False)
```

```
+-----------+-----------+----------+-------+---------------------+
|antecedent |consequent |confidence|lift   |support              |
+-----------+-----------+----------+-------+---------------------+
|[nm0002935]|[nm0004310]|1.0       |32804.5|3.0483622673718545E-5|
+-----------+-----------+----------+-------+---------------------+
```

```
spark.sql('''SELECT namesingleactors_data.NAME_actors, namesingleactors_data.ID_actors
              FROM namesingleactors_data
              WHERE namesingleactors_data.ID_actors IN ('nm0002935', 'nm0004310')''').show
```

```
+-----------+---------+
|NAME_actors|ID_actors|
+-----------+---------+
| Moe Howard|nm0002935|
| Larry Fine|nm0004310|
+-----------+---------+
```

```
rules.sort(rules.lift.asc()).show(10, False)
```

```
ascrules = rules.sort(rules.lift.asc())
ascrules.createOrReplaceTempView("ascrules")
```

```
+--------------------------------+-----------+-----------------+-----------------+
|antecedent                      |consequent |confidence       |lift             |
+--------------------------------+-----------+-----------------+-----------------+
|[nm0648300, nm0619779, nm0623427]|[nm0006982]|0.8              |538.3302564102564|
|[nm2133977, nm1588355]          |[nm0006982]|0.8              |538.3302564102564|
|[nm0451005, nm1588355]          |[nm0006982]|0.8              |538.3302564102564|
|[nm0024301, nm0623427]          |[nm0006982]|0.8181818181818182|550.565034965035 |
|[nm1588355, nm0623427]          |[nm0006982]|0.8275862068965517|556.8933687002652|
|[nm2414317, nm2369538, nm2077739]|[nm0648803]|0.8              |557.3861946902655|
|[nm0619779, nm0046850, nm0623427]|[nm0006982]|0.8333333333333334|560.7606837606838|
|[nm1770187]                     |[nm0648803]|0.8151260504201681|567.9250092957537|
|[nm0080246, nm0623427]          |[nm0006982]|0.8444444444444444|568.2374928774929|
|[nm4236498]                     |[nm0006982]|0.85             |571.9758974358974|
+--------------------------------+-----------+-----------------+-----------------+
only showing top 10 rows
```

FP Growth with a **restrictive threshold**

```
FP_minsup = FPGrowth(itemsCol="actors", minSupport=0.00033) #to be frequent 130 movies
model_minsup = FP_minsup.fit(baskets_id) #it takes 3 min
```

```
rules_minsup = model_minsup.associationRules
rules_minsup.createOrReplaceTempView("rules_minsup")
rules_minsup.sort(rules_minsup.lift.desc()).show(20, False)
```

```
spark-3.2.0-bin-hadoop2.7/python/pyspark/sql/context.py:127: FutureWarning: Deprecat
  FutureWarning
+-----------+-----------+------------------+-----------------+---------------------+
|antecedent |consequent |confidence        |lift             |support              |
+-----------+-----------+------------------+-----------------+---------------------+
|[nm2082516]|[nm0648803]|0.9130434782608695|636.1472874182377|3.7342437775305217E-4|
+-----------+-----------+------------------+-----------------+---------------------+
```

```
spark.sql('''SELECT namesingleactors_data.NAME_actors as FPgrowth_pair
             FROM namesingleactors_data
             WHERE namesingleactors_data.ID_actors IN ('nm2082516', 'nm0648803')''').show
```

```
+----------------+
|   FPgrowth_pair|
+----------------+
|Matsunosuke Onoe|
|    Kijaku Ôtani|
+----------------+
```

Apriori Algorithm

```
transactions = baskets_id.select('actors').rdd.flatMap(lambda x: x)
```

```
transactions.take(4)
```

```
[['nm1012612',
  'nm0675260',
  'nm1012621',
  'nm1010955',
  'nm0675239',
  'nm1011210'],
 ['nm0252720', 'nm0215752'],
 ['nm0624446'],
 ['nm0140054', 'nm0097421']]
```

```
list_flat =  transactions.flatMap(list)
singleitem = list_flat.map(lambda item: (item , 1))
```

```
singleitem.take(2)
```

```
[('nm1012612', 1), ('nm0675260', 1)]
```

```
def SumCount(x,y):
    return x+y
```

```
support = singleitem.reduceByKey(SumCount)
```

```
support.take(2)
```

```
[('nm1012621', 1), ('nm1011210', 1)]
```

```
supports = support.map(lambda item: item[1])
```

```
supports.take(2)
```

```
[1, 1]
```

```
minSupport = 130
```

```
support = support.filter(lambda item: item[1] >= minSupport )
```

```
support.take(2)
```

```
[('nm0225905', 151), ('nm0435229', 138)]
```

```
singlerdd = support.map(lambda item: ([item[0]] , item[1]))
```

```
singlerdd.take(2)

    [(['nm0225905'], 151), (['nm0435229'], 138)]


frequent_actors = support.map(lambda item: (item[0]))


frequent_actors.take(2)

    ['nm0225905', 'nm0435229']


columns = ["actor", "frequent"]
candidates_actors = singlerdd.toDF(columns)
candidates_actors.createOrReplaceTempView("candidates_actors")

spark.sql('''SELECT *
             FROM candidates_actors
             ORDER BY frequent DESC''').show(10, False)

    +-----------+--------+
    |actor      |frequent|
    +-----------+--------+
    |[nm0103977]|798     |
    |[nm0006982]|585     |
    |[nm0648803]|565     |
    |[nm0305182]|506     |
    |[nm0623427]|438     |
    |[nm0793813]|411     |
    |[nm0246703]|391     |
    |[nm0619107]|387     |
    |[nm0007123]|381     |
    |[nm7390393]|355     |
    +-----------+--------+
    only showing top 10 rows



candidates_actors = candidates_actors.withColumn("actor", concat_ws(" ",col("actor")))
candidates_actors.createOrReplaceTempView("candidates_actors")


spark.sql('''SELECT namesingleactors_data.NAME_actors, candidates_actors.actor, candidates
             FROM candidates_actors INNER JOIN namesingleactors_data ON candidates_actors
             ORDER BY candidates_actors.frequent DESC''').show()

    +------------------+---------+--------+
    |       NAME_actors|    actor|frequent|
    +------------------+---------+--------+
    |      Brahmanandam|nm0103977|     798|
    |       Adoor Bhasi|nm0006982|     585|
    |  Matsunosuke Onoe|nm0648803|     565|
    |      Eddie Garcia|nm0305182|     506|
    |        Prem Nazir|nm0623427|     438|
    |      Sung-il Shin|nm0793813|     411|
    |      Paquito Diaz|nm0246703|     391|
    |  Masayoshi Nogami|nm0619107|     387|
```

```
|         Mammootty|nm0007123|     381|
|    Aachi Manorama|nm7390393|     355|
|           Bahadur|nm0046850|     348|
|           Mohanlal|nm0482320|     344|
|Mithun Chakraborty|nm0149822|     330|
|   Shivaji Ganesan|nm0304262|     323|
|       Sultan Rahi|nm0706691|     315|
|            Nagesh|nm0619309|     313|
|     Shakti Kapoor|nm0007106|     310|
|       Pandharibai|nm0659250|     303|
|         Tom Byron|nm0001000|     303|
|       Jayabharati|nm0419653|     303|
+------------------+---------+--------+
only showing top 20 rows
```

The frequent single actors that Apriori find are the same that we found in FP growth, in the same order.

```
print(frequent_actors.count())
```

```
324
```

```
import itertools
```

```
 pairs_list = list(itertools.combinations(frequent_actors.toLocalIterator(),2))
```

```
def removef (rdd, list_items):
  for item in list_items:
    if set(list(item)).issubset(set(rdd)):
      return((item, 1))
```

```
support_pairs = transactions.map(lambda x: removef(x, pairs_list)).filter(lambda x: x is r
```

```
support_pairs.take(2)
```

```
[(('nm0392442', 'nm0369058'), 1), (('nm0392442', 'nm0001935'), 1)]
```

```
sum_support_pairs= support_pairs.reduceByKey(SumCount)
```

```
candidates_pair = sum_support_pairs.filter(lambda item: item[1] >= minSupport)
```

```
candidates_pairs = candidates_pair.map(lambda item: ([item[0]] , item[1]))
```

```
columnspairs = ["pairs", "frequent"]
pairs = candidates_pairs.toDF(columnspairs)
pairs.createOrReplaceTempView("pairs")
```

```
spark.sql('''SELECT *
              FROM pairs
              ORDER BY frequent DESC''').show(10, False) #it takes 4h
```

```
+------------------------+--------+
|pairs                   |frequent|
+------------------------+--------+
|[{nm2082516, nm0648803}]|146     |
+------------------------+--------+
```

```
spark.sql('''SELECT namesingleactors_data.NAME_actors, namesingleactors_data.ID_actors
              FROM namesingleactors_data
              WHERE ID_actors in ('nm2082516','nm0648803')''').show()
```

```
+----------------+---------+
|     NAME_actors|ID_actors|
+----------------+---------+
|Matsunosuke Onoe|nm0648803|
|    Kijaku Ôtani|nm2082516|
+----------------+---------+
```

```
spark.sql('''SELECT candidates_actors.actor, candidates_actors.frequent
              FROM candidates_actors
              WHERE candidates_actors.actor in ('nm2082516','nm0648803') ''').show()
```

```
+---------+--------+
|    actor|frequent|
+---------+--------+
|nm2082516|     161|
|nm0648803|     565|
+---------+--------+
```

Confidence and **Lift** of the Apriori frequent pair

```
Confidence_nm2082516_nm0648803 = round(146/161, 2)
```

```
print(Confidence_nm2082516_nm0648803)
```

```
0.91
```

```
value = round((161*565)/393654, 2)
lift_pair = round((146/value), 2)
print(lift_pair)
```

```
634.78
```

Check on **results**

```
movies_of_Matsunosuke_Onoe = spark.sql('''SELECT ID_movie, TITLE_movie
                                FROM data
                                WHERE (ID_actors = 'nm0648803')''')


movies_of_Kijaku_Otani = spark.sql('''SELECT ID_movie, TITLE_movie
                                FROM data
                                WHERE (ID_actors = 'nm2082516')''')


movies_of_Matsunosuke_Onoe.createOrReplaceTempView("movies_of_Matsunosuke_Onoe")
movies_of_Kijaku_Otani.createOrReplaceTempView("movies_of_Kijaku_Otani")
```

Movies of the Frequent pair

```
spark.sql('''SELECT a.TITLE_movie
              FROM movies_of_Matsunosuke_Onoe a INNER JOIN movies_of_Kijaku_Otani b on a.I
```

```
+----------------------+
|TITLE_movie           |
+----------------------+
|Nidaime jiraiya       |
|Onigoroshi juzô       |
|Gôsho Kingôro         |
|Moyuru uzumaki sanbu  |
|Araki Mataemon        |
|Kantô shichinin otoko |
|Chûshingura           |
|Yakko no kôsan        |
|Meitô takada matabei  |
|Benten kozo           |
|Go henge kikûmatsu    |
|Yoshioka Kanefusa     |
|Gôsho no Gorozô       |
|Kana tehon chûsinghura|
|Takenâka hanbei       |
+----------------------+
only showing top 15 rows
```

```
title_movies_frequentpair = spark.sql('''SELECT a.TITLE_movie
                                        FROM movies_of_Matsunosuke_Onoe a INNER JOIN movie
title_movies_frequentpair.createOrReplaceTempView("title_movies_frequentpair")
spark.sql('''SELECT count(distinct(TITLE_movie))
              FROM title_movies_frequentpair''').show()
```

```
+---------------------------+
|count(DISTINCT TITLE_movie)|
+---------------------------+
|                        146|
+---------------------------+
```

```
spark.sql('''SELECT b.TITLE_movie, a.*
```

```
        FROM infotitlebasics a INNER JOIN title_movies_frequentpair b on a.primaryTi
        WHERE (a.titleType == 'movie')''').show()
```

```
+--------------------+---------+---------+--------------------+--------------------+
|         TITLE_movie|   tconst|titleType|        primaryTitle|       originalTitle|
+--------------------+---------+---------+--------------------+--------------------+
|       Yurei hannôjô|tt1068870|    movie|       Yurei hannôjô|       Yurei hannôjô|
|         Nihon ginji|tt1095100|    movie|         Nihon ginji|         Nihon ginji|
|     Natsume sentarô|tt1075367|    movie|     Natsume sentarô|     Natsume sentarô|
|     Nichigetsû tarô|tt1066465|    movie|     Nichigetsû tarô|     Nichigetsû tarô|
|    Kairiki shinkchi|tt1093834|    movie|    Kairiki shinkchi|    Kairiki shinkchi|
|Sûrûga dainagôn t...|tt1075838|    movie|Sûrûga dainagôn t...|Sûrûga dainagôn t...|
| Moyuru uzumaki gobu|tt1086867|    movie| Moyuru uzumaki gobu| Moyuru uzumaki gobu|
|Kashûn toyamazaku...|tt1094644|    movie|Kashûn toyamazaku...|Kashûn toyamazaku...|
|Kana tehon chûsin...|tt1089695|    movie|Kana tehon chûsin...|Kana tehon chûsin...|
|Moyuru uzumaki sanbu|tt1085463|    movie|Moyuru uzumaki sanbu|Moyuru uzumaki sanbu|
|Oyashirazû no ada...|tt1070834|    movie|Oyashirazû no ada...|Oyashirazû no ada...|
|         Niôchô gôrô|tt1074996|    movie|         Niôchô gôrô|         Niôchô gôrô|
|      Nagaî gênaburô|tt1106829|    movie|      Nagaî gênaburô|      Nagaî gênaburô|
| Moyuru uzumaki nibu|tt1085462|    movie| Moyuru uzumaki nibu| Moyuru uzumaki nibu|
|       Hôncho kôachi|tt1082842|    movie|       Hôncho kôachi|       Hôncho kôachi|
|Gôketsu miyabe kû...|tt1081985|    movie|Gôketsu miyabe kû...|Gôketsu miyabe kû...|
|       Daîja no ocho|tt1096877|    movie|       Daîja no ocho|       Daîja no ocho|
|     Gôsho no Gorozô|tt1081986|    movie|     Gôsho no Gorozô|     Gôsho no Gorozô|
|Mûsashiya tatsugo...|tt1106828|    movie|Mûsashiya tatsugo...|Mûsashiya tatsugo...|
|       Ikaruga Heiji|tt1082001|    movie|       Ikaruga Heiji|       Ikaruga Heiji|
+--------------------+---------+---------+--------------------+--------------------+
only showing top 20 rows
```