Market-basket Analysis

Algorithms for Massive Datasets Project

Camilla Gotta 945522
Laura Ciurca 942318
Università degli Studi di Milano
Data Science and Economics

Contents

1 **Problem Definition** **3**
1.1 Dataset . . . 3
1.2 Data Cleaning and Preprocessing . . . 5

2 **Algorithm and Implementation** **9**
2.1 Apriori Algorithm . . . 9
2.2 FPGrowth Algorithm . . . 9

3 **Analysis and Scaling Solution** **11**

4 **Conclusions** **15**

5 **Appendix** **18**

List of Figures

1.1 Kaggle API Setup 5
1.2 Dataset needed for analysis derived from an inner join on three main datasets 6
1.3 Dataset filtered for actor/actress profession 6
1.4 Dataset filtered for movie titleType 7
1.5 Baskets 7
1.6 Basket code 8

3.1 Apriori run on a sample of 70.000 11
3.2 Apriori algorithm runtime on samples of different sizes 12
3.3 FP-Growth results after being run on a sample 13
3.4 FP-Growth results after being run on entire dataset 14

4.1 Apriori and FP-Growth with different support values 16

Problem Definition

The aim of the project is to implement a scalable solution for finding frequent itemsets, which has been applied to a movies' dataset. This kind of analysis is also known as 'market-basket analysis', which deals with combinations of items that occur together frequently in baskets. The IMDB dataset taken from Kaggle has been analyzed in order to find the actors and actresses, considered as 'items', who occur together more frequently in the 'baskets' of movies. Considering that the dataset contains millions of records, an accurate pre-processing analysis has been applied after having set up Apache Spark environment, which is an open-source analytics engine focused on speed, ease in use, and distributed system necessary to analyze massive data sets. The solution has been written through Python 3 using Google Colab for a better reproducibility of the results.

1.1 Dataset

The dataset used in this analysis is the IMDB dataset[1], published on Kaggle under IMDb non-commercial licensing. It consist of 5 different datasets which provide details of the American cinematography, more in detail:

1. title.akas.tsv.gz - Contains the following information for titles:

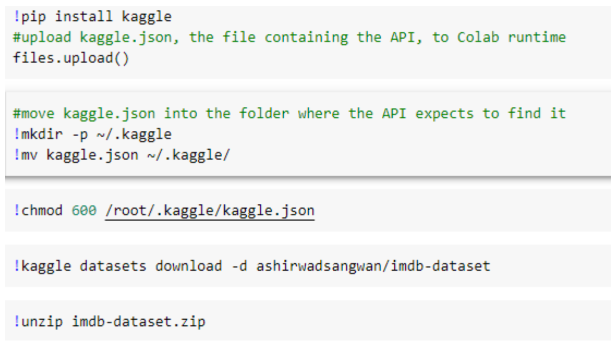
    - titleId (string): an alphanumeric unique identifier of the title.
    - ordering (integer): a number to uniquely identify rows for a given titleId.
    - title (string): the localized title.
    - region (string):the region for this version of the title.
    - language (string): the language of the title.
    - types (array): Enumerated set of attributes for this alternative title. One or more of the following: "alternative", "dvd", "festival", "tv", "video", "working", "original", "imdbDisplay". New values may be added in the future without warning.
    - attributes (array):Additional terms to describe this alternative title, not enumerated.

- isOriginalTitle (boolean): 0: not original title; 1: original title.

2. title.basics.tsv.gz - Contains the following information for titles:
    - tconst (string) - alphanumeric unique identifier of the title.
    - titleType (string) – the type/format of the title (e.g. movie, short, tvseries, tvepisode, video, etc).
    - primaryTitle (string) – the more popular title / the title used by the filmmakers on promotional materials at the point of release.
    - originalTitle (string) - original title, in the original language.
    - isAdult (boolean) - 0: non-adult title; 1: adult title.
    - startYear (YYYY) – represents the release year of a title. In the case of TV Series, it is the series start year.
    - endYear (YYYY) – TV Series end year. for all other title types.
    - runtimeMinutes – primary runtime of the title, in minutes.
    - genres (string array) – includes up to three genres associated with the title.
3. title.principals.tsv.gz – Contains the principal cast/crew for titles:
    - tconst (string) - alphanumeric unique identifier of the title.
    - ordering (integer) – a number to uniquely identify rows for a given titleId.
    - nconst (string) - alphanumeric unique identifier of the name/person.
    - category (string) - the category of job that person was in.
    - job (string) - the specific job title if applicable, else.
    - characters (string) - the name of the character played if applicable, else.
4. title.ratings.tsv.gz – Contains the IMDb rating and votes information for titles:
    - tconst (string) - alphanumeric unique identifier of the title.
    - averageRating – weighted average of all the individual user ratings.
    - numVotes - number of votes the title has received.
5. name.basics.tsv.gz – Contains the following information for names:
    - nconst (string) - alphanumeric unique identifier of the name/person.
    - primaryName (string)– name by which the person is most often credited.
    - birthYear – in YYYY format.
    - deathYear – in YYYY format if applicable.
    - primaryProfession (array of strings)– the top-3 professions of the person.
    - knownForTitles (array of tconsts) – titles the person is known for.

1.2 Data Cleaning and Preprocessing

At first, SparkContext has been created to determine each record of RDDs and to read the CSV file. Once data have been imported directly on Google Colab thanks to the Kaggle's API they are finally unzipped.



```
!pip install kaggle
#upload kaggle.json, the file containing the API, to Colab runtime
files.upload()
```

```
#move kaggle.json into the folder where the API expects to find it
!mkdir -p ~/.kaggle
!mv kaggle.json ~/.kaggle/
```

```
!chmod 600 /root/.kaggle/kaggle.json
```

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

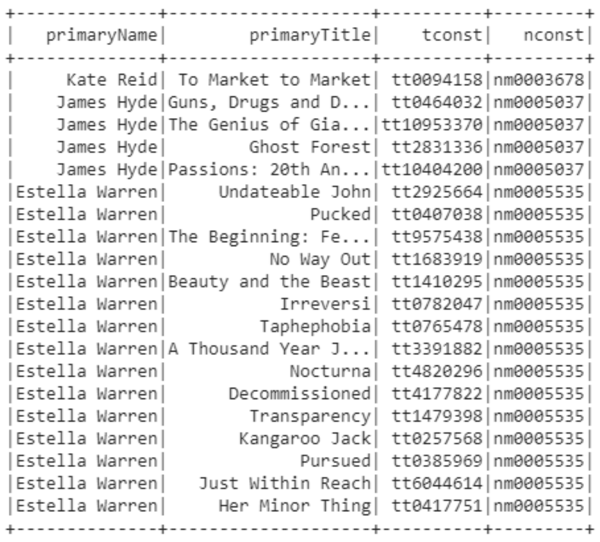
```
!unzip imdb-dataset.zip
```

Figure 1.1: Kaggle API Setup

For the purpose of this project and baskets' creation, the following files from the downloaded datasets are used:

- title.basics.tsv
- title.principals.tsv
- name.basics.tsv

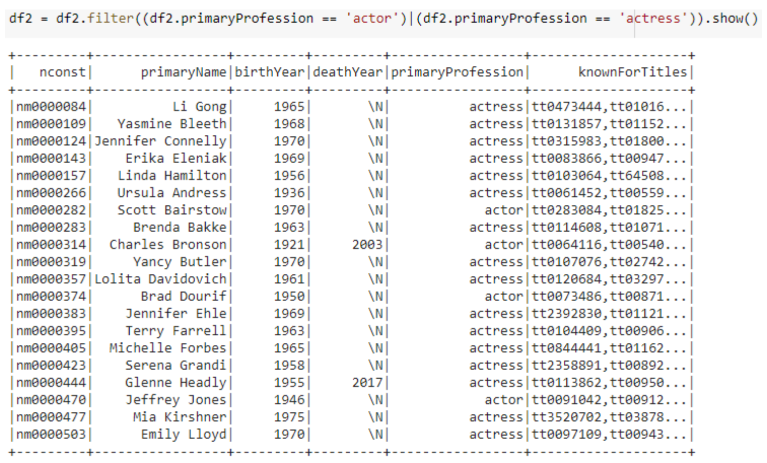
Since the task of the project is considering movies as baskets and actors as items, these two categories have been retrieved from the above mentioned datasets through an SQL inner join, resulting in the following new dataset displayed in Fig.1.2.



```
+-------------+------------------+----------+---------+
|  primaryName|      primaryTitle|    tconst|   nconst|
+-------------+------------------+----------+---------+
|    Kate Reid| To Market to Market| tt0094158|nm0003678|
|   James Hyde|Guns, Drugs and D...| tt0464032|nm0005037|
|   James Hyde|The Genius of Gia...|tt10953370|nm0005037|
|   James Hyde|        Ghost Forest| tt2831336|nm0005037|
|   James Hyde|Passions: 20th An...|tt10404200|nm0005037|
|Estella Warren|     Undateable John| tt2925664|nm0005535|
|Estella Warren|              Pucked| tt0407038|nm0005535|
|Estella Warren|The Beginning: Fe...| tt9575438|nm0005535|
|Estella Warren|          No Way Out| tt1683919|nm0005535|
|Estella Warren|Beauty and the Beast| tt1410295|nm0005535|
|Estella Warren|          Irreversi| tt0782047|nm0005535|
|Estella Warren|         Taphephobia| tt0765478|nm0005535|
|Estella Warren|A Thousand Year J...| tt3391882|nm0005535|
|Estella Warren|            Nocturna| tt4820296|nm0005535|
|Estella Warren|      Decommissioned| tt4177822|nm0005535|
|Estella Warren|        Transparency| tt1479398|nm0005535|
|Estella Warren|       Kangaroo Jack| tt0257568|nm0005535|
|Estella Warren|             Pursued| tt0385969|nm0005535|
|Estella Warren|   Just Within Reach| tt6044614|nm0005535|
|Estella Warren|     Her Minor Thing| tt0417751|nm0005535|
+-------------+------------------+----------+---------+
```

Figure 1.2: Dataset needed for analysis derived from an inner join on three main datasets

After having checked values of primary profession column from name.basics, only those who have as primary profession actor and/or actress have been selected, as one can see in Fig.1.3

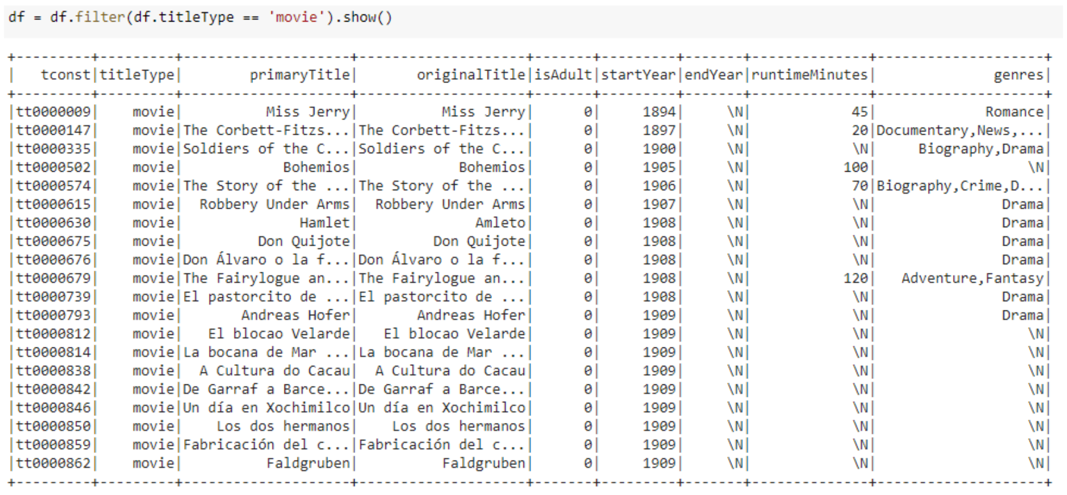


```
df2 = df2.filter((df2.primaryProfession == 'actor')|(df2.primaryProfession == 'actress')).show()
```

```
+---------+----------------+---------+---------+-----------------+-------------------+
|   nconst|     primaryName|birthYear|deathYear|primaryProfession|     knownForTitles|
+---------+----------------+---------+---------+-----------------+-------------------+
|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...|
|nm0000266|   Ursula Andress|     1936|       \N|          actress|tt0061452,tt00559...|
|nm0000282|   Scott Bairstow|     1970|       \N|            actor|tt0283084,tt01825...|
|nm0000283|     Brenda Bakke|     1963|       \N|          actress|tt0114608,tt01071...|
|nm0000314|  Charles Bronson|     1921|     2003|            actor|tt0064116,tt00540...|
|nm0000319|     Yancy Butler|     1970|       \N|          actress|tt0107076,tt02742...|
|nm0000357|Lolita Davidovich|     1961|       \N|          actress|tt0120684,tt03297...|
|nm0000374|      Brad Dourif|     1950|       \N|            actor|tt0073486,tt00871...|
|nm0000383|    Jennifer Ehle|     1969|       \N|          actress|tt2392830,tt01121...|
|nm0000395|    Terry Farrell|     1963|       \N|          actress|tt0104409,tt00906...|
|nm0000405|  Michelle Forbes|     1965|       \N|          actress|tt0844441,tt01162...|
|nm0000423|    Serena Grandi|     1958|       \N|          actress|tt2358891,tt00892...|
|nm0000444|    Glenne Headly|     1955|     2017|          actress|tt0113862,tt00950...|
|nm0000470|    Jeffrey Jones|     1946|       \N|            actor|tt0091042,tt00912...|
|nm0000477|     Mia Kirshner|     1975|       \N|          actress|tt3520702,tt03878...|
|nm0000503|      Emily Lloyd|     1970|       \N|          actress|tt0097109,tt00943...|
+---------+----------------+---------+---------+-----------------+-------------------+
```

Figure 1.3: Dataset filtered for actor/actress profession

Another important check has been done as concerns the type of the title, selecting just those with value “movie”, as shown in Fig.1.4

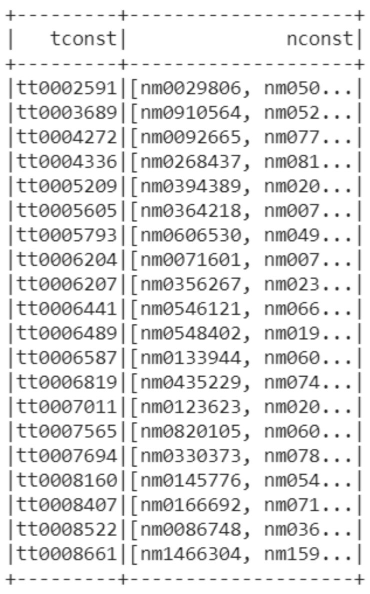


```
df = df.filter(df.titleType == 'movie').show()
```

```
+---------+---------+--------------------+--------------------+-------+---------+-------+--------------+--------------------+
|   tconst|titleType|        primaryTitle|       originalTitle|isAdult|startYear|endYear|runtimeMinutes|              genres|
+---------+---------+--------------------+--------------------+-------+---------+-------+--------------+--------------------+
|tt0000009|    movie|          Miss Jerry|          Miss Jerry|      0|     1894|     \N|            45|             Romance|
|tt0000147|    movie|The Corbett-Fitzs...|The Corbett-Fitzs...|      0|     1897|     \N|            20|Documentary,News,...|
|tt0000335|    movie|Soldiers of the C...|Soldiers of the C...|      0|     1900|     \N|            \N|     Biography,Drama|
|tt0000502|    movie|            Bohemios|            Bohemios|      0|     1905|     \N|           100|                  \N|
|tt0000574|    movie|The Story of the ...|The Story of the ...|      0|     1906|     \N|            70|Biography,Crime,D...|
|tt0000615|    movie|  Robbery Under Arms|  Robbery Under Arms|      0|     1907|     \N|            \N|               Drama|
|tt0000630|    movie|              Hamlet|              Amleto|      0|     1908|     \N|            \N|               Drama|
|tt0000675|    movie|         Don Quijote|         Don Quijote|      0|     1908|     \N|            \N|               Drama|
|tt0000676|    movie|Don Álvaro o la f...|Don Álvaro o la f...|      0|     1908|     \N|            \N|               Drama|
|tt0000679|    movie|The Fairylogue an...|The Fairylogue an...|      0|     1908|     \N|           120|   Adventure,Fantasy|
|tt0000739|    movie|El pastorcito de ...|El pastorcito de ...|      0|     1908|     \N|            \N|               Drama|
|tt0000793|    movie|       Andreas Hofer|       Andreas Hofer|      0|     1909|     \N|            \N|               Drama|
|tt0000812|    movie|   El blocao Velarde|   El blocao Velarde|      0|     1909|     \N|            \N|                  \N|
|tt0000814|    movie|La bocana de Mar ...|La bocana de Mar ...|      0|     1909|     \N|            \N|                  \N|
|tt0000838|    movie|  A Cultura do Cacau|  A Cultura do Cacau|      0|     1909|     \N|            \N|                  \N|
|tt0000842|    movie|De Garraf a Barce...|De Garraf a Barce...|      0|     1909|     \N|            \N|                  \N|
|tt0000846|    movie|Un día en Xochimilco|Un día en Xochimilco|      0|     1909|     \N|            \N|                  \N|
|tt0000850|    movie|    Los dos hermanos|    Los dos hermanos|      0|     1909|     \N|            \N|                  \N|
|tt0000859|    movie|Fabricación del c...|Fabricación del c...|      0|     1909|     \N|            \N|                  \N|
|tt0000862|    movie|          Faldgruben|          Faldgruben|      0|     1909|     \N|            \N|                  \N|
+---------+---------+--------------------+--------------------+-------+---------+-------+--------------+--------------------+
```

Figure 1.4: Dataset filtered for movie titleType

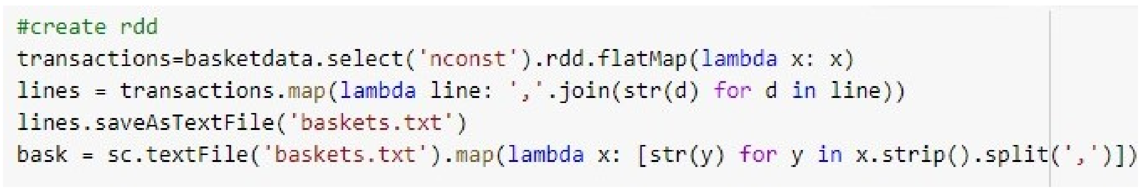
Once data has been cleaned, baskets have been created, in order to group all actors for each movie. In order to have a more precise and quick analysis, unique Ids for both actors and movies have been used, which are respectively defined as ”nconst” and ”tconst” from the original dataset.



```
+---------+--------------------+
|   tconst|              nconst|
+---------+--------------------+
|tt0002591|[nm0029806, nm050...|
|tt0003689|[nm0910564, nm052...|
|tt0004272|[nm0092665, nm077...|
|tt0004336|[nm0268437, nm081...|
|tt0005209|[nm0394389, nm020...|
|tt0005605|[nm0364218, nm007...|
|tt0005793|[nm0606530, nm049...|
|tt0006204|[nm0071601, nm007...|
|tt0006207|[nm0356267, nm023...|
|tt0006441|[nm0546121, nm066...|
|tt0006489|[nm0548402, nm019...|
|tt0006587|[nm0133944, nm060...|
|tt0006819|[nm0435229, nm074...|
|tt0007011|[nm0123623, nm020...|
|tt0007565|[nm0820105, nm060...|
|tt0007694|[nm0330373, nm078...|
|tt0008160|[nm0145776, nm054...|
|tt0008407|[nm0166692, nm071...|
|tt0008522|[nm0086748, nm036...|
|tt0008661|[nm1466304, nm159...|
+---------+--------------------+
```

Figure 1.5: Baskets

After having defined the baskets, textFile() method has been applied to read file line by line, so that each line in our CSV file will be a value in RDD.



```
#create rdd
transactions=basketdata.select('nconst').rdd.flatMap(lambda x: x)
lines = transactions.map(lambda line: ','.join(str(d) for d in line))
lines.saveAsTextFile('baskets.txt')
bask = sc.textFile('baskets.txt').map(lambda x: [str(y) for y in x.strip().split(',')])
```

Figure 1.6: Basket code

Algorithm and Implementation

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

2.1 Apriori Algorithm

Apriori algorithm is the first algorithm that has been proposed for frequent itemset mining. After being improved by R. Agarwal and R. Srikant, it came to be known as Apriori. This data mining technique follows two main steps to reduce the search space iteratively until the most frequent itemset is achieved:

1. In the first iteration of the algorithm, each item is taken as a 1-itemsets candidate, more precisely the algorithm finds frequencies by considering how many times the items occur in the data-set. It depends on the frequencies of the itemset: frequencies, or “support value”, are obtained for every single item, by extracting every item in RDDs and calculating each unique item’s frequency.
2. In the second step, the algorithm counts all the candidates that consist of frequent items and checks which have counts that are equal to or greater than the support threshold. If the candidates do not meet the minimum support, then they are regarded as infrequent and thus removed.

Between the two steps of the A-Priori, the count of the items is examined to determine which of them are frequent as singletons, in order to set a threshold sufficiently high that does not return too many frequent sets, namely 1% of the baskets.[2]

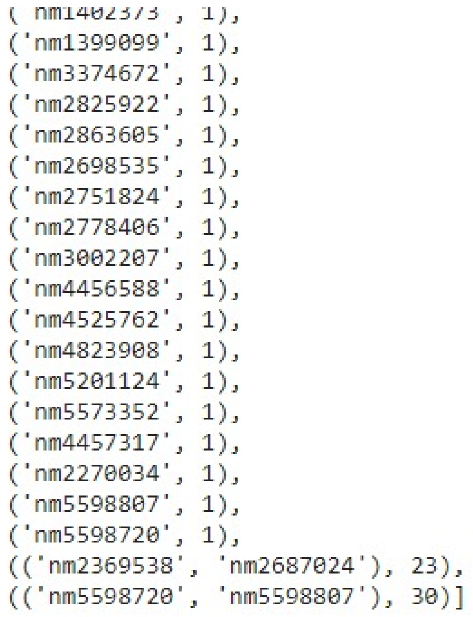
2.2 FPGrowth Algorithm

FP-Growth is an algorithm available in the machine learning Spark library for extracting frequent itemsets and it is a popular alternative to the basic Apriori algorithm. In general, the algorithm has been designed to operate on databases containing baskets. As for the Apriori algorithm, the itemset is considered as ”frequent” if it meets a user-specified support threshold. In particular and what makes it different from

Apriori frequent pattern mining algorithm, FP-Growth is a frequent pattern mining algorithm that does not require candidate generation. Internally, it uses a so-called FP-tree (frequent pattern tree) data structure without generating the candidate sets explicitly, which makes it particularly usefull for large datasets.[3]

Analysis and Scaling Solution

The performance of the two algorithms has been evaluated in terms of execution time. Due to the enormous computational cost in the generation of frequent itemsets, the Apriori algorithm has been implemented for a sample size of 70.000 transactions with support value of 0.0003.
Apriori works as expected, with a runtime value of 1719.518134355545 seconds, approximately 28 minutes, finding the frequent itemsets from transactions. Most of them are singletons as well as it shows two pairs of frequent itemset which appear in the baskets respectively 23 and 30 times, as shown in Fig.3.1.



```
('nm1402373', 1),
('nm1399099', 1),
('nm3374672', 1),
('nm2825922', 1),
('nm2863605', 1),
('nm2698535', 1),
('nm2751824', 1),
('nm2778406', 1),
('nm3002207', 1),
('nm4456588', 1),
('nm4525762', 1),
('nm4823908', 1),
('nm5201124', 1),
('nm5573352', 1),
('nm4457317', 1),
('nm2270034', 1),
('nm5598807', 1),
('nm5598720', 1),
(('nm2369538', 'nm2687024'), 23),
(('nm5598720', 'nm5598807'), 30)]
```

Figure 3.1: Apriori run on a sample of 70.000

Apriori algorithm has been applied to samples of different size in such a way to look at its scalability. It demonstrates how much time is needed for Apriori algorithm for mining frequent itemsets as one increases the sample size of transactions: more data to be processed, more candidate itemsets to generate, thus more time to find the maximum frequent itemset. As expected, to a bigger sample size corresponds more time (in seconds) of execution with a more than proportional relation, as one can see in Fig.3.2.

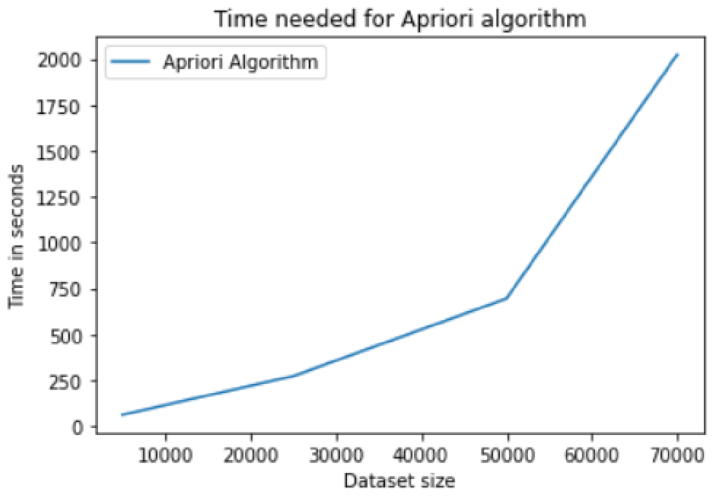
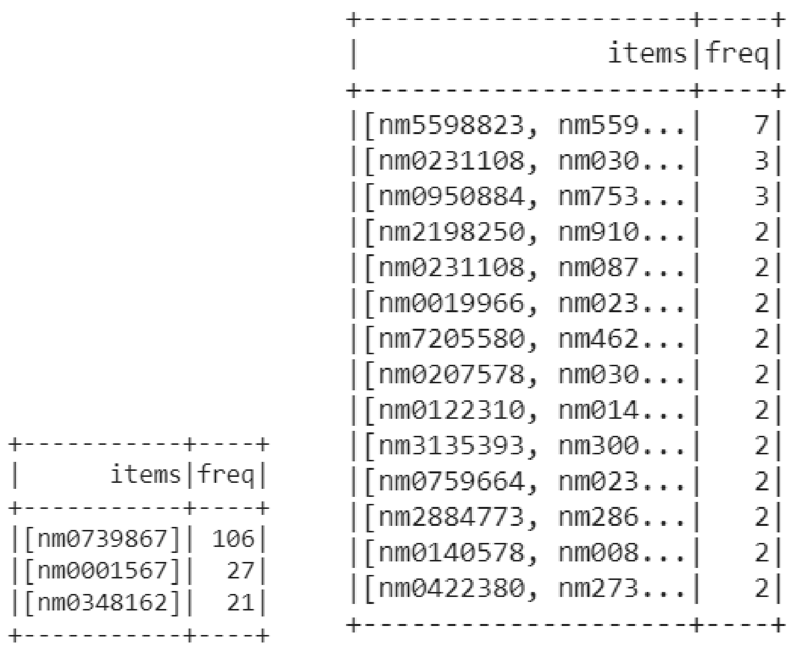


Figure 3.2: Apriori algorithm runtime on samples of different sizes

As concerns the FP-Growth algorithm, it has been chosen to retrieve the results performed on a sample of 70.000 transactions and then on the entire dataset to show the difference concerning scalability, runtime and performance with respect to Apriori results. Support value was fixed at 0.00003.
As shown in Fig.3.3, the most frequent item appears on film's basket 107 times and the maximum frequent pair has a frequency equal to 7 times.

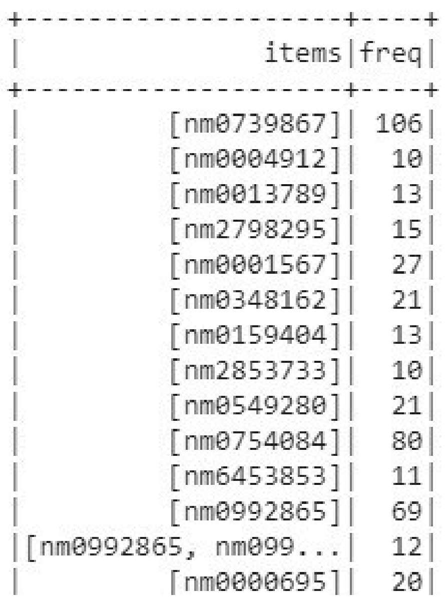


```
+-----------+----+
|      items|freq|
+-----------+----+
|[nm0739867]| 106|
|[nm0001567]|  27|
|[nm0348162]|  21|
+-----------+----+
```

```
+--------------------+----+
|               items|freq|
+--------------------+----+
|[nm5598823, nm559...|   7|
|[nm0231108, nm030...|   3|
|[nm0950884, nm753...|   3|
|[nm2198250, nm910...|   2|
|[nm0231108, nm087...|   2|
|[nm0019966, nm023...|   2|
|[nm7205580, nm462...|   2|
|[nm0207578, nm030...|   2|
|[nm0122310, nm014...|   2|
|[nm3135393, nm300...|   2|
|[nm0759664, nm023...|   2|
|[nm2884773, nm286...|   2|
|[nm0140578, nm008...|   2|
|[nm0422380, nm273...|   2|
+--------------------+----+
```

Figure 3.3: FP-Growth results after being run on a sample

FP-growth algorithm has been run on the whole dataset and with threshold 0.00003, same as before, and has resulted in performing in 135.223123 seconds (about 2 minutes). The outcome, displayed in Fig.3.4, presents frequent singletons and only one frequent pair with respect to the result obtained before. This is due to the fact of having the same support level but for a larger size of the dataset, meaning a decreasing number of items classifying as frequent: more frequent singletons rather than frequent itemsets.



```
+--------------------+----+
|               items|freq|
+--------------------+----+
|           [nm0739867]| 106|
|           [nm0004912]|  10|
|           [nm0013789]|  13|
|           [nm2798295]|  15|
|           [nm0001567]|  27|
|           [nm0348162]|  21|
|           [nm0159404]|  13|
|           [nm2853733]|  10|
|           [nm0549280]|  21|
|           [nm0754084]|  80|
|           [nm6453853]|  11|
|           [nm0992865]|  69|
|[nm0992865, nm099...|  12|
|           [nm0000695]|  20|
```

Figure 3.4: FP-Growth results after being run on entire dataset

As resulted from the experimental study, it is clear that the performance of FP-Growth algorithm is better than the one of the Apriori mainly because the first requires less execution time than the latter, meaning that the time to mine the frequent itemsets is extremely less. The efficiency of the Spark-based algorithms extensively depends on the way it is parallelized on Spark, and the underlying data structure used to store and compute frequent itemsets. Being highly iterative, this parallel and distributed version of algorithms have been developed to definitely avoid the computational problem of generating frequent itemsets for larger datasets: that encourages the usage of Spark that overcomes all problems of scalability, memory and speed.

Conclusions

The purpose of this work, based on performing market basket analysis, that is finding frequent sets of items appearing in many of the same baskets, has been accomplished. Firstly, the absolute number of films that contain a particular set of actors and/or actresses has been retrieved from the chosen IMBD dataset. Then, Apriori and FP-Growth algorithms have been implemented to achieve this goal. In fact, results demonstrate how many and which actors and actresses appear more frequently in which films. It is possible to conclude that both algorithms have reached the initial purpose, demonstrating how many and which actors and actresses appear more frequently individually and together in which films. Furthermore, they have managed to scale up massive quantities of data.
What can be decisive in evaluating the algorithms' performance are the time required to obtain these results and consequently the threshold decision for which a set of items can be defined as frequent or not. As shown in the Fig.4.1, maintaining the sample fixed at 70.000 baskets, the performance analysis of runtime for various support levels demonstrates a strong relation between the necessary time for running the algorithms and the selected threshold values: the time of execution decreases as the minimum support level increases.
For the Apriori, as the support becomes bigger, less time is needed to find the maximal frequent itemsets of the transactions but it has to be highlighted that as the support value goes towards 0.001, the algorithm prints out an empty list of frequent itemsets: the choice of the support have a critical role to obtain the expected results. If the support is too high, the result will be nor a list of frequent itemsets neither a list of frequent singletons. Just a meaningless empty list.

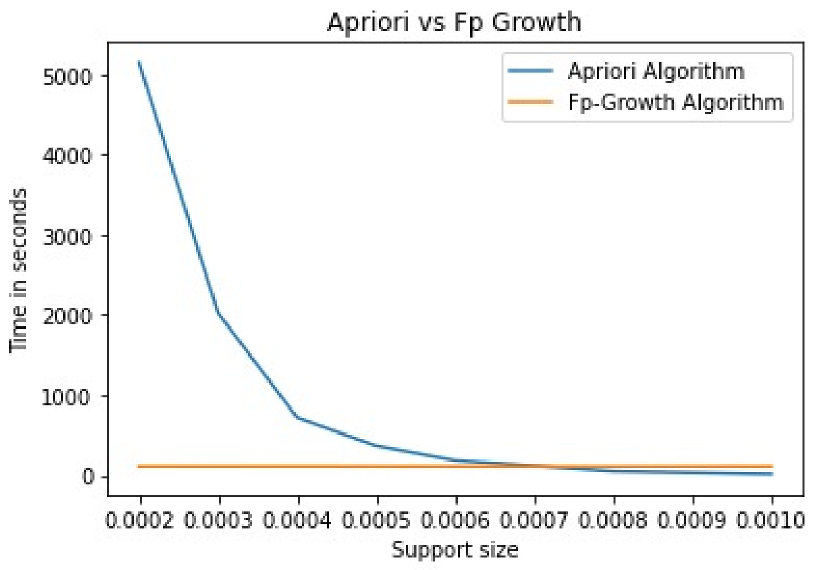


Figure 4.1: Apriori and FP-Growth with different support values

By other side, FP-growth algorithm's performance shows a constant trend: no matter the value of the fixed support, the time span is approximately two minutes maximum. Surely it comes with a considerable saving of time.

Bibliography

[1] Kaggle.com, *IMDb Dataset*. Available at: https://www.kaggle.com/ashirwadsangwan/imdb-dataset

[2] J. Leskovec, A. Rajaraman, J.Ullman, *Mining of Massive Datasets*, 2014

[3] A. Rakesh, R. Srikant, *Fast algorithms for mining association rules*, 1994

Appendix

```
# -*- coding: utf-8 -*-
"""Market_Basket_Analysis_AMD_project.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1-Uem4GTPD5ULiZ0WUrbMDK6-CU3xIe1a

**ALGORITHM FOR MASSIVE DATASETS: MARKET BASKET ANALYSIS**

---



#####Laura Ciurca
#####Camilla Gotta
#####2020/2021

**SPARK SETUP**

---
"""

#install Java8
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
#download spark3.0.2
#!apt-get update
!wget -c http://apache.osuosl.org/spark/spark-3.0.2/spark-3.0.2-bin-hadoop2.7.tgz
#unzip it
!tar xf spark-3.0.2-bin-hadoop2.7.tgz
#install findspark
!pip install -q findspark

!cat /proc/cpuinfo

import os
```

```
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.0.2-bin-hadoop2.7"

import findspark
findspark.init("spark-3.0.2-bin-hadoop2.7")  #SPARK_HOME
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[*]").getOrCreate()

import pyspark
sc = spark.sparkContext

"""**KAGGLE SETUP**

---


"""

!pip install kaggle

"""Import the dataset through the Kaggle API"""

#upload kaggle.json, the file containing the API, to Colab runtime
from google.colab import files
files.upload()

#move kaggle.json into the folder where the API expects to find it
!mkdir -p ~/.kaggle
!mv kaggle.json ~/.kaggle/

!chmod 600 /root/.kaggle/kaggle.json

!kaggle datasets download -d ashirwadsangwan/imdb-dataset

"""Unzip the dataset"""

!unzip imdb-dataset.zip

"""Read the dataset on spark"""

df = spark.read.csv("/content/title.basics.tsv/title.basics.tsv", sep=r'\t', header=True)
df1 = spark.read.csv("/content/title.principals.tsv/title.principals.tsv", sep=r'\t', header=True)
df2 = spark.read.csv("/content/name.basics.tsv/name.basics.tsv", sep=r'\t', header=True)

"""**DATA PREPROCESSING**

---
```

```
Retrieve only actors and actress as primary profession
"""

#filtering actor dataset according just for Primary names that have actor or actress roles
df2 = df2.filter((df2.primaryProfession == 'actor')|(df2.primaryProfession == 'actress'))

"""Select film under category "movie""""

#dataset␣taking␣just␣movies
df␣=␣df.filter(df.titleType␣==␣'movie')

df.createOrReplaceTempView("df")
df1.createOrReplaceTempView("df1")
df2.createOrReplaceTempView("df2")

df␣=␣df.select(['tconst',␣'primaryTitle'])
df1␣=␣df1.select(['tconst',␣'nconst'])
df2␣=␣df2.select(['primaryName',␣'nconst'])

""" Dataset needed for analysis derived from an inner join on three main datasets"""

#inner join of the datasets
dataset = spark.sql("""SELECT DISTINCT df2.primaryName, df.primaryTitle, df1.tconst, df1.nconst
                      FROM df
                      INNER JOIN df1 ON df.tconst = df1.tconst
                      INNER JOIN df2 ON df1.nconst = df2.nconst
                      LIMIT 70000""")

#libraries needed
from pyspark.sql.functions import collect_set
from pyspark.sql.functions import size, col
from pyspark.sql import functions as F
from collections import defaultdict
import itertools
import pandas as pd
import time
import matplotlib.pyplot as plt

#create baskets
basketdata = dataset.groupBy('tconst').agg(collect_set('nconst').alias('nconst'))
basketdata.createOrReplaceTempView('basketdata')
#basketdata.toPandas().head(5)
#basketdata.count()

#check number of actors for each movie
basketdata=basketdata.select('*',size('nconst').alias('actors'))

"""Creation of the RDD of the transactions"""
```

```
#create rdd
transactions=basketdata.select('nconst').rdd.flatMap(lambda x: x)
lines = transactions.map(lambda line: ','.join(str(d) for d in line))
lines.saveAsTextFile('baskets.txt')
bask = sc.textFile('baskets.txt').map(lambda x: [str(y) for y in x.strip().split(',')])

"""**APRIORI ALGORITHM**

---


"""

#define support
count= basketdata.count()
supports= 0.0003*count
numPartitions = bask.getNumPartitions()

#determine candidates
def get_candidates(frequent_items,k):
    elements=set()
    if(k>1):
        for itemsets in frequent_items:
            for item in itemsets:
                elements.add(item)

    candidate_sets=[set(itemsets) for itemsets in list(itertools.combinations(elements, k))]

    return candidate_sets

def candidates_basket(iterator,candidates):
  return iterator.flatMap(lambda x: [(tuple(c), 1) for c in candidates if c.issubset(set(x))]).reduceByKey(lambda a,b: a+b).filter(lambda

#determine frequent itemsets
def get_frequent_itemset(iterator):
    baskets = iterator.collect()
    support_part = supports
    k = 2
    d = {}
    frequent_items = []
    frequent_itemset = []

    for b in baskets:
        for i in b:
            if i not in d:
                d[i] = 1
            else:
```

```
                        d[i] = d[i] + 1

        for i in d:
            if d[i] >= support_part:
                frequent_items.append(i)

        frequent_itemset = [(i,1) for i in frequent_items]

        return(frequent_itemset)

#define apriori
def apriori(iterator):

    iterator.cache()
    freq_itemsets = get_frequent_itemset(iterator)
    freq_items = [{i[0]} for i in freq_itemsets]
    k=2
    candidates = get_candidates(freq_items,k)

    while len(candidates) != 0:
        freq_itemsets_2 = candidates_basket(iterator,candidates)
        freq_itemsets += freq_itemsets_2
        freq_items2 = list(map(lambda x: {x[0]}, freq_itemsets_2))

        # new candidates
        candidates = get_candidates(freq_items2, k)

        k += 1

    iterator.unpersist()

    return freq_itemsets

#run apriori
start_time = time.time()
apriori(bask)
print(time.time() - start_time)

"""**FP-GROWTH ALGORITHM**

---


"""

#implement FpGrowth
from pyspark.ml.fpm import FPGrowth
start_time = time.time()
```

```
fpGrowth = FPGrowth(itemsCol="nconst", minSupport=0.00003)
start_time = time.time()
model = fpGrowth.fit(basketdata)
print(time.time() - start_time)

#Display frequentItems
model.freqItemsets.show()

f= model.freqItemsets
f.createOrReplaceTempView("f")

query = """select items, freq
            from f
            where size(items) > 2
            order by freq desc
            """
spark.sql(query).show()

"""**DATA VISUALIZATION**

---

Plot the time needed for Apriori algorithm with different sample size of dataset
"""

# Commented out IPython magic to ensure Python compatibility.
#get graph for time (seconds) running apriori
w = {'size_dataset': [5000, 25000, 50000, 70000],
     'time_seconds': [63.071523904800415,272.9228575229645,695.7591323852539, 1719.518134355545]}
results = pd.DataFrame(w)
# %matplotlib inline
x = results['size_dataset']
y = results['time_seconds']
plt.plot(x, y, label="Apriori Algorithm")

plt.xlabel("Dataset size")
plt.ylabel("Time in seconds")
plt.title('Time needed for Apriori algorithm')

leg = plt.legend()

plt.savefig('image.pdf')

plt.show()

"""Plot the difference of the runtime for both algorithms with different support values

"""
```

```
# Commented out IPython magic to ensure Python compatibility.
#get graph for time (seconds) running both algorithm for different support values

# %matplotlib inline
x =[0.0002, 0.0003, 0.0004,0.0005, 0.0006, 0.0008, 0.001]
y = [5139.251349925995,2022.997619152069,733.544305562973,367.53301644325256,185.84784150123596,51.371659994125366,17.269320249557495]
plt.plot(x, y, label="Apriori Algorithm")

x1=[0.0002, 0.0003, 0.0004,0.0005, 0.0006, 0.0008, 0.001]
y1=[104.76757955551147,112.26622992477417,111.86981964111328,111.74520134925842,111.484503077698,110.27831411361694,111.74520134925842]
plt.plot(x1, y1, label="Fp-Growth Algorithm")

plt.xlabel("Support size")
plt.ylabel("Time in seconds")
plt.title('Apriori vs Fp-Growth algorithm')

leg = plt.legend()

plt.savefig('image.pdf')

plt.show()
```