Market-basket analysis: finding frequent itemsets on IMDb data

Andrea Marconi

[1]Università degli Studi di Milano, Diparimento di Informatica, via Celoria, 18, Milano, 20135, Milan, Italy.

Contributing authors: andrea.marconi3@studenti.unimi.it;

Abstract

After introducing the framework of the market-basket analysis I propose implementations of simple algorithms for finding frequent itemsets. These sets can be exploited to build association rules and calculate confidence and interest metrics for their evaluation.

Keywords: market-basket, association rules, frequent itemsets, A-Priori

1 Introduction

The following project was implemented in Python, using the COLAB environment provided by Google. I didn't setup a virtual machine, I relied on the free service provided by Google.

I decided to do 'Project 2: Market-basket analysis'[1] on IMDB dataset. The aim of the project is to find frequent itemsets of dimension k, where k is a parameter that can be chosen by the user. The analysis is known as market-basket analysis. We have on one side baskets, which are sets of items and on the other items. We assume that the number of baskets is large, so it usually cannot sit in main memory. Drawing a parallel with brick-and-mortar stores, we want to find which items are bought together so we can profit from these patterns. We are interested in finding sets of items of dimension k that are frequently bought together. By finding these frequent sets we can build association rules and evaluate them with confidence and interest metrics.

Frequent itemsets are useful to build meaningful association rules, from which one can infer that the presence of some items imply the presence of others.

2 Data understanding

The dataset used is the IMDb database[2] which stores information about Titles, Persons and the relation between the two entities, with additional informations regarding Titles and Persons. IMDb[6] is a famous website for ranking movies, tv-series and other entertainment related concepts. The dataset is provided on the Kaggle website[4] and it's accessible via the Kaggle API having uploaded the Kaggle token, personal to the user and after having configured the environment with the right directories. The dataset is structured and is composed of five files:

Definition 1 Each dataset is contained in a gzipped, tab-separated-values (TSV) formatted file in the UTF-8 character set. The first line in each file lists headers that describe the content of each column. A \N is used to denote that a particular field is missing or null for that title/name.

We are interested in movies and actors / actresses. In particular, we consider movies as baskets and actors as items. We want to find actors who acted together in many movies.

We consider the following files:

- title.basics.tsv.gz - Contains information for titles;
- title.principals.tsv.gz – Contains the principal cast/crew for titles;
- name.basics.tsv.gz – Contains the information for names.

  These files will be stored in the following variables:

  – titles = title.basics.tsv.gz;
  – association = title.principals.tsv.gz;
  – people = name.basics.tsv.gz.

These files, once imported with the right encoding and data types, contain structured data. We are interested in movies and in actors / actresses in these movies. The files were imported with the PySpark library, in particular using the SQLContext of the SparkContext. I relied on an implementation available here on Kaggle[5] for importing and pre-processing the data. I also did the same operations with the Pandas library which I know better.

Considering the E-R model, there are two entites: titles and people. The two entities are related with a many-to-many association in which for every title there are one or more persons and for every person there could be zero or more titles. The following picture contains the UML class diagram for the considered domain.

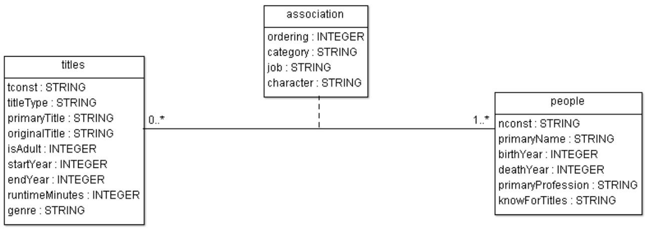

Fig. 1 UML class diagram for the the considered domain.

The following tables list the attributes available in each of the three files.

Table 1 Titles: information for titles

tconst	titleType	primaryTitle	originalTitle	isAdult	startYear	endYear..
title ID string	type of title string	known as string	original title string	for adults ? boolean	release year YYYY	end year YYYY
tt0000009	movie	Miss Jerry	Miss Jerry	0	1894	

Table containing information about titles, which can be TV series, movies, short movies and more. We are interested only in movie titleType.

Table 2 People: information for names

nconst	primaryName	birthYear	deathYear	primaryProfession	knownForTitles
person ID string	name string	DoB YYYY	DoD YYYY	top-3 professions array of category	person is famous for ? array of tconst
nm0000001	Fred Astaire	1899	1987	soundtrack,act...	tt0050419,tt01...

Table containing information about people, DoB, DoD, their main professions and why they are known to the public.

3 Data pre-processing

After importing the data with the right encoding and delimiter, I performed selection and filtering on the imported files to extract:

- from the variable 'titles' only the rows having 'titleType == movie' and storing the result in variable 'movies';
- from the variable 'associaton' only the rows having 'category == actor or category == actress' and storing the result in variable 'relationActors'.

Table 3 Association: association between titles and person: who did what in what title

tconst	ordering	nconst	category	job	characters
title ID string	row ID for title ID integer	person ID string	role of the person string	more specific string	character name string
tt0000005	2	nm0653042	actor		Assistant

Table containing the association between the entity Titles and the entity People. We are interested only in actors and actresses categories.

I performed a 'Left-Semi Join' which is a particular type of join available in SQLContext of PySpark. The table on the left side is 'relationActors' which is the table 'association' filtered. The table on the right is 'movies' which is the table 'titles' filtered. The join is performed on the column 'tconst' which is the identifier of a title. The 'Left-Semi Join' performs a query like this:

```
SELECT *
FROM relationActors rA
WHERE EXISTS (SELECT * FROM movies M WHERE rA.tconst = M.tconst)
```

Finally, I need to have all the baskets on a single column like:

```
from this:
| movieKey1 | actorKey1 |
| movieKey1 | actorKey2 |
| movieKey1 | actorKey3 |
....

we need to group the result using the keys of movies,
the desiderata is something like this:
| movieKey1 | actorKey1, actorKey2, actorKey3, ... |
| movieKey2 | actorKey1, actorKey4, actorKey8, ... |
....
```

To do this last step, I used the groupBy clause on the movie key and collected in a list the result. This particular procedure for pre-processing the data was insipired by another implementation present on Kaggle[5].

4 Implemented algorithms

I implemented three solutions, in descending order of importance and efficiency:

- A-Priori algorithm from scratch: main function:

  `scratch_APriori`

- A-Priori algorithm from scratch, different implementation: main function:

`scratch_APriori_v3`

- Tentative of SON algorithm from scratch: main function:

  `distributedScratch_APriori`

The first one, is the main solution. It is an implementation of the A-Priori algorithm from scratch which exploits the monotonicity observation for finding frequent itemsets:

Definition 2 *If a set I of items is frequent, then so is every subset of I.*

This observation implies that we can avoid to create every single combination of k-size itemsets since we can generate only the combinations created starting from the frequent itemsets found in the previous pass. The first solution exploits this fact and differs from the second in the fact that I don't explicitly generate the combinations of (k-1)-size itemsets and check for every item generated but I simply scan the baskets and generate only those combinations which contain (k-1)-size frequent itemsets. This saves time and memory.

The third solution is a tentative implementation of the SON algorithm without the use of RDD of Spark. In an ideal implementation (I wasn't able to properly implement this solution, I wanted to execute the counting of sampled baskets with the proportional threshold in parallel) this algorithm would be very fast.

The three solutions are designed to keep looping through data until we can't build bigger itemsets. So, depending of the threshold set we can get frequent itemsets of small to big size. The main solution outputs two dictionaries which for every k-sized itemsets stores respectively its count and the frequent ones. Keeping the counts of itemsets is very demanding in terms of memory but permits to calculate association rules very easily. The implementations of the algorithms[7] are self-explanatory and the code is commented as neeeded.

5 Discussion on scaling

Since we exploit the monotonicty property, the solutions proposed tend to scale and perform on bigger datasets if the latter fit in main memory during execution. The dictionaries used to count items can be very memory expensive if the are a lot of singletons. The monotonicity implies that we should find fewer frequent itemsets as we go on with the size k so I think this shouldn't be a big issue for high thresholds.

My main concern is the parallel execution of the solutions. Since I couldn't find an efficient way to perform parallel executions in the experimental environment I couldn't really test this aspect, in particular with regard to massive datasets and main memory occupation. I think, though, that by splitting the data of baskets in chunks and performing the same algorithm with a proportional threshold calculated w.r.t. the number of total chunks and then composing the counts can be efficient for good thresholds.

6 Results

Using the main solution, on all the baskets with a threshold of 20 this is the output:

```
Threshold: 20
Pass on baskets data number 1
Creating singletons...
Filtering item sets of dimension 1...
Frequent item sets of dimension 1: 13127
['nm0509573', 'nm0101071',...
Pass on baskets data number 2
Counting 2-sized item sets...
Item sets of size 2: 382224
Filtering item sets of dimension 2...
Frequent item sets of dimension 2: 1029
[('nm0063138', 'nm0124279'), ('nm0286594', 'nm0591704'),....
Pass on baskets data number 3
Counting 3-sized item sets...
Item sets of size 3: 232
Filtering item sets of dimension 3...
Frequent item sets of dimension 3: 10
[('nm1698868', 'nm2366585', 'nm2384746'), ...
Pass on baskets data number 4
Counting 4-sized item sets...
Item sets of size 4: 0
Filtering item sets of dimension 4...
Frequent item sets of dimension 4: 0
[]
--- 1559.9288759231567 seconds ---
```

The association rules which can be derived from the k-sized itemsets are evaluated with the two metrics proposed in the textbook 'RU', in particular confidence and interest:

$$Confidence(I \to j) = \frac{Support(I \cup \{j\})}{Support(I)} \tag{1}$$

$$Interest(I \to j) = Confidence(I \to j) - \frac{Support(j)}{\|Baskets\|} \tag{2}$$

For instance, the 10 frequent itemsets of dimension 3 depicted in the output above can be used to build association rules and display their confidence and intereset:

```
Association rule
===========================
If ('nm1698868', 'nm2366585') then nm2384746
Confidence: 0.26666666666666666
|---> 24/90
Interest: 0.2662754621632423
|---> 0.26666666666666666 - (154 / 393656)
===========================

Association rule
===========================
If ('nm1698868', 'nm2384746') then nm2366585
Confidence: 0.2891566265060241
|---> 24/83
Interest: 0.28863078668648623
|---> 0.2891566265060241 - (207 / 393656)
===========================

Association rule
===========================
If ('nm2366585', 'nm2384746') then nm1698868
Confidence: 0.25
|---> 24/96
Interest: 0.24955290913894365
|---> 0.25 - (176 / 393656)
===========================

Association rule
===========================
If ('nm1698868', 'nm2366585') then nm2367854
Confidence: 0.24444444444444444
|---> 22/90
Interest: 0.24424884219273227
|---> 0.24444444444444444 - (77 / 393656)
===========================

Association rule
===========================
If ('nm1698868', 'nm2367854') then nm2366585
Confidence: 0.3728813559322034
|---> 22/59
Interest: 0.37235551611266554
|---> 0.3728813559322034 - (207 / 393656)
===========================
...
```

As shown above, the interest has tiny differences w.r.t. confidence because the fraction of sets in which the typical consequent of the rule is very small. So typically, the rules displayed are meaningful of true relations.

7 Conclusion

The main solution described in this report and available in the COLAB notebook works well on the considered dataset and I think would perform pretty well on bigger instances compared with similar implementations. As summarized in comments in the implemented solutions I didn't implement solutions with the use of RDD of PySpark because the operations required the use of for loops inside .map() calls on baskets and the use of .collect() which is typically and operation to avoid since can be very time and space consuming.

Declarations

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.
When I took inspiration of other solutions on the Web I clearly reported both in my code (in comments) and in this report the source.

References

[1] List of project available, https://docs.google.com/document/d/1aB-gpEdge7jscDwasyJhVwARdV_L0Ine_Jso6oviY7o/edit

[2] IMDb Dataset available on Kaggle, https://www.kaggle.com/ashirwadsangwan/imdb-dataset

[3] Implementation frequent itemsets from scratch, ethen8181.github.io/machine-learning/association_rule/apriori.html

[4] Kaggle Website, https://www.kaggle.com/

[5] Importing and organizing data, https://www.kaggle.com/surtrsurtr/imdb-dataset-2

[6] The IMDb website, https://www.imdb.com/

[7] The implementation of my solution https://colab.research.google.com/drive/1CkFNCIyK15chAsuczQMhnZXzZz-WLOfT