Big Data and Cloud Computing : Project 1

António Almeida, UP201505836
Miguel Sozinho Ramalho, UP201403027

April 7th, 2019

Contents

1 Introduction

In the context of the Big Data and Cloud Computing (BDCC) course, a study of techniques used for handling large amounts of data with typical languages and frameworks was developed. These tools include the Python[1] programming language, Jupyter Notebooks[2], Pyspark[3], Google Cloud Platform[4] and some of its functionality such as virtual machines and cloud buckets. The ultimate goal being the understanding of what works for machine learning applications, which approaches scale well enough as the amount of data increases and how all of this can be deployed through cloud computing tools.

Particularly, this project consists on a study of a benchmark dataset, MovieLens[5], for recommendation tasks using metrics such as *Term Frequency–Inverse Document Frequency*[6] (tf-idf) and the *Jaccard Index*[7] for similarity calculations. Section 3 includes details about the implementation of mandatory functions and Section 4, on the other hand, includes details on the implementation of optional functions that represent an extent to the previous functions and have a higher degree of freedom on how they should be implemented.

2 Dataset

From the Movielens Dataset, the following tables were used:

- **Movies**, with the columns **movieId, title**
- **Tags**, with the columns: **movieId, userId, tag**
- **Ratings**, with the columns: **movieId, userId, rating**

3 Functions to implement

The final notebook contains a top-down structure of dependencies, as is recommended for such notebooks, and is organised in a similar fashion to this report. In the subsections below, each of the mandatory functions is described in terms of implementation decisions and not functionality, as that information is already present in the project requirements document[8].

[1]https://www.python.org/
[2]https://jupyter.org/
[3]https://spark.apache.org/docs/2.2.1/api/python/pyspark.html
[4]https://cloud.google.com/gcp
[5]https://grouplens.org/datasets/movielens/
[6]https://en.wikipedia.org/wiki/Tf%E2%80%93idf
[7]https://en.wikipedia.org/wiki/Jaccard$_{index}$
[8]http://www.dcc.fc.up.pt/ edrdo/aulas/bdcc/projects/01/

3.1 tfidfTags

1. Group the tags dataframe by (tag, movieId) and count the number of users of the aggregation, naming it as f, in a new dataframe - *df_f*
2. Group the result from the previous step by movieId so as to obtain the maximum value of f for each movie - *f_max* - in a new dataframe - *df_f_max*
3. Call an auxiliary function - *get_idf* - that calculates the inverse document frequency values for a given dataset
4. Join the *df_f* dataframe with the *df_f_max* dataframe on movieId, so as to know the *f_max* value for each tag. At this stage the document frequency is also calculated and added to the result. Finally, a join with the *df_idf* dataframe is performed so as to get all the information together.
5. Lastly, the previous dataframe is returned with a new column which results from applying the tf-idf formula and multiplying the TF value with the IDF value in other columns.

3.2 recommendByTag

1. Filter the TFIDF_tags dataframe by the target tag, i.e. $singleTag$
2. Filter the resulting dataframe by removing entries with *f_max* inferior to min_fmax, to exclude movies tagged sparsely, in a new dataframe - *df_tag*
3. Join *df_tag* with the *movies* dataframe on movieId, so as to obtain the title of the movie each tag is associated with
4. Order the results by (TF_IDF, *title*), by descending and asscending order, respectively
5. The previous dataframe contains unnecessary columns, so only the relevant ones are selected, i.e., *movieId*, *title*, and TF_IDF
6. Lastly, the results are limited by $numberOfResults$ and then returned

3.3 recommendByTags

1. Call the provided function *createTagListDF* on *searchTags*, which creates a dataframe with one entry per target tag, in a new variable - *df_search_tag*
2. Filter $TFIDF\_tags$ by removing entries with *f_max* inferior to min_fmax, to exclude movies tagged sparsely
3. Join the resulting dataframe with *df_search_tags* on tag, so as to obtain entries that match the target tags, in a new dataframe - *df_tags*

4. Group *df_tags* by $movieId$ and calculate the sum of TF_IDF of the aggregation, naming it as SUM_TF_IDF
5. Join the result from the previous step with movies on movieId to get the movie titles
6. Order the resulting dataframe by (SUM_TF_IDF, *title*), by descending and ascending order, respectively
7. Reorder the columns of the resulting dataframe to match the examples, by selecting them in the correct order, i.e. (*moviedId*, *title*, SUM_TF_IDF)
8. Finally, the results are limited by $numberOfResults$ and then returned

3.4 jiMovieSimilarity

1. Filter the ratings of movies for greater than or equal to 4.0, the minimum value considered for a rating to be a *like*, *df_likes*
2. Group the liked ratings by $movieId$ and aggregate a set of the userIds of the users who rated the movie; these sets counted per movie and used to filter those movies that have less than the *minRatings* parameter and the set and movieId columns are renamed to $u1$ and $m1$
3. Duplicate the previous dataframe by changing the column names of the copy to $u2$ and $m2$
4. Lastly, a cross product is generated between both dataframes guaranteeing that $m1 < m2$ to avoid duplicates; the intersection and union of both user sets are taken and their size used to calculate the Jaccard Index. (This functionality has been isolated in the $jaccard\_index$ function)
5. Lastly, the results proper columns are returned

3.5 recommendBySimilarity

1. The JI dataframe is used twice to search the target movie on both the $m1$ and $m2$ columns
2. These two datasets are appended
3. Simply use the new dataset to join it with the movies dataframe, to obtain the movie titles, and then sort it by decreasing value of JI
4. Finally, the results are limited by $numberOfResults$ and then returned

4 Extended Functionality

Following the same concept of Section 3, this section describes the implementation decisions made for the extended functionality which, as said above, have a higher degree of freedom where function naming and parameter choosing is concerned.

4.1 recommend_by_keywords

The goal here was to extend the TF-IDF recommendation of movies to one that used, besides the user tags, the individual words inside the movie titles. The parameters are: $(searchTags, movies, min\_fmax = 10, numberOfResults = 10, debug = False)$

1. Split each movie title into tokens and remove punctuation, saving one row per token for each movie (stop words could easily have been removed, but we had no cue on the impact of those on the recommendation results, or on different languages besides english in the dataset)
2. For each movie and title word, a negative and unique id was created so that we eneded up with a dataframe of $movieId$, tag (token) a, $userId$ (negative unique id) this was so that previously defined functions could be invoked
3. Given the previous point, the remaining actions were simply to join the previous dataset with that of tags, as they have the same columns, and call the $tfidfTags$ (see Section 3.1) function followed by returning the result of *recommendByTags* (see Section 3.3).

4.2 jiMovieTags

To achieve the goal functionality, two functions were developed: $ji\_similarity\_tags\_movies$ - calculates the Jaccard similarity between tags based on the films they are applied to, and $recommend_tags$ - for a given movie m, returns n tags based on their similarity to the tags associated to m

4.2.1 ji_similarity_tags_movies

1. On the tags dataframe, for each tag - as column $t1$, collect the set of movies it has been associated with as a column $m1$, in a new dataframe - *df_t*1
2. Duplicate *df_t*1 with $t1$, $m1$ renamed as $t2$, $m2$, in a new dataframe - *df_t*2
3. Call external function for JI calculation ($jaccard\_index$)

4.2.2 recommend_tags

1. Filter the tags dataframe to get the entries for the target movie, i.e. m, in a new dataframe - *df_tags*

2. Match *df_tags* with the ones on the Jaccard similarity dataframe, i.e. *ji*, by creating two dataframes with matches on the $t1$ and $t2$ columns, and later uniting them
3. Order the resulting dataframe by descesding (JI,i,u), limit the results by n, i.e. the number of target tags, and return the result

4.3 jiUsersBehaviour

To achieve the goal functionality, two functions were developed: $ji_similarity_user_ratings$ - calculates thee Jaccard simmilarity between users based on what films they rate, and $recommend_by_ratings$ - for a given user u, returns the top-rated film per each of the most n similar users, as long as u has net yet rated or tagged the movies at stake.

4.3.1 ji_similarity_user_ratings

1. On the ratings dataframe, for each user - as column $u1$, get the set of its ratings as a column $r1$, in a new dataframe - *df_u*1
2. Duplicate df_u1 with $u1$, $r1$ renamed as $u1$, $r2$, in a new dataframe - *df_u*2
3. Call external function for JI calculation (*jaccard_index*)

4.3.2 recommend_by_ratings

1. Get the n most similar users to the target user, i.e. u, by filtering the Jaccard similarity dataframe, i.e. ji, by userId, in a new dataframe - *df_u*
2. Get the ratings made by the 'top n' users by joining *df_u* with the ratings dataframe on userId, in a new dataframe - *df_top_n_ratings*
3. Get the movies user u has rated or tagged, by filtering the ratings and tags dataframes on userId and unifying the results, in a new dataframe - *df_u_rt*
4. Get movies from 'top n' users that user u has rated or tagged by joining *df_top_n_ratings* with *df_u_rt*, in a new dataframe - *df_common*
5. Get the eligible movies from the 'top n' users by calculating the difference between *df_top_ratings* and *df_common*, in a new dataframe - *df_top_movies*
6. Filter *df_top_movies* by getting the top rated movie for each user, by calculating a dataframe with the max_rating for each user, joining with with *df_top_movies*, and removing "duplicate" entries, as we want a movie from each user
7. Finally, join the resulting dataframe with the movies dataframe to get the titles, select relevant columns, order by desceding JI, and return the result

5 Conclusions

This project has proved invaluable for the group members to understand the theory learned in classes, its challenges and how it can be leveraged to reach real-world applications with big data and cloud computing. In the future, these skills will help the students be more proficient in applying the proper techniques and also fairly more adaptable to the presented challenges. It also proved that the tools used are more than capable of performing the required tasks efficiently and with few development costs.

On a note for the teachers, the group also felt some inconsistencies associated with the naming of mandatory functions using *camelCase* and the lack of a normative approach for parameter names (which have both *camelCase* and *snake_case*, these do not follow most Python naming conventions, such as Google Python Style Guide [9].

All in all, this was a more work than initially expected, since the teachers already provided us with a structured notebook and very clear instructions but we learned that with time event what seems simple, can be simplified further until a point of perfection, which we have fought for in this project.

[9]https://google.github.io/styleguide/pyguide.html