CS631 Final Project - IMDb Datasets Analysis

Mingrui Zhang
20985422

Yanhao Lin
20481552

1 INTRODUCTION

In this project, we built two applications: the *Movie Production Decision Maker* which was what we originally proposed, and an additional program the *Movie Finder*.

The *Movie Production Decision Maker* is a piece of software that takes a set of very basic inputs from the user (the target user group is the movie production company) to find the top cast and crew candidates who are most likely to achieve a high-rating movie. Originally, we would like to mainly use the budget and the box office of the movies to implement the movie production decision-maker. However, the IMDb box office dojo does not have a downloadable database file, and data obtained by using the network request does not have a direct connection with the IMDb Datasets that we use. Therefore, our production decision-maker focuses on the average rating of the movie and the number of votes to achieve the rating. In general, by using the inputs from the user, the program should generate possible actors and crews to make the film and their average ratings which are calculated from their principal titles' ratings would be displayed in a horizontal bar chart.

The *Movie Finder* is a movie recommendation system that suggests movies to a user based on certain criteria according to the user's preference. The reason/purpose for including the second program in the project is to:

(1) Solving a similar but logically opposite problem: Compared with the *Movie Production Decision Maker* which targets the movie producers and takes a minimum number of inputs to generate many possible outputs, this program targets the audience and tries to collect as much information as possible to generate one concrete output.
(2) Unlike the *DataFrame* approach used in the *Movie Production Decision Maker*, this program takes the *RDD* as we want to provide examples of both techniques for other people and explore how these two approaches compare.

This report talks about the retrieval and processing of the raw data, the implementation, and results of both programs as well as the lessons learned by doing this project.

2 DATA

2.1 Data Source

The Internet Movie Database (IMDb) is an online database of information related to movies, television shows, and other forms of media. It includes data on actors, actresses, directors, producers, writers, and other cast and crew members, as well as information on the movies and television shows themselves, such as plot summaries, ratings, and release dates.

The IMDb datasets consist of a large collection of data taken from the IMDb database. It includes data on movies, television shows, and other forms of media, as well as data on the people who have worked on them.

Table 1: Data Source Metadata

File	Size	Row Number
name.basics.tsv	694 MB	53,590,746
title.akas.tsv	1.58 GB	12,150,957
title.basics.tsv	771 MB	34,248,687
title.principles.tsv	2.19 GB	53,590,746
title.crew.tsv	295 MB	9,450,421
title.episodes.tsv	177 MB	7,143,453
title.ratings.tsv	20.7 MB	1,257,371

The IMDb datasets are often used for data analysis and machine learning tasks, such as recommending movies to users or predicting the success of a movie based on its cast and crew. It is also used for tasks such as analyzing trends in the entertainment industry or studying the careers of individual actors and actresses.

There are seven tables in total which IMDb provides for free (download here directly from IMDb) [1]. The metadata of each table is shown in Table 1, and the relation between the tables is shown in Figure 1.

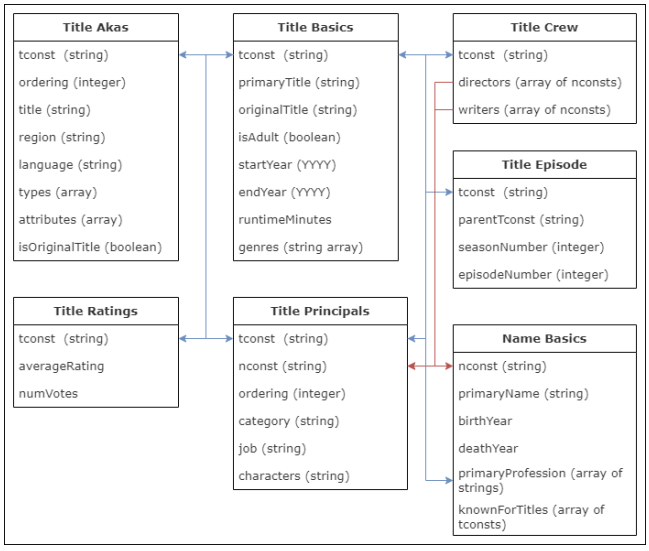

Figure 1: IMDb dataset table relationships

2.2 Data Retrieval

The connections between the tables are made via two unique alphanumeric identifiers:

- *nconst* - alphanumeric unique identifier of the name/person

- *tconst* - alphanumeric unique identifier of the title (movies/TV Shows)

A column could store a single identifier or an array of identifiers depending on the purpose. We use the table *Title Principles* as the bridge to link movies and people together as this table's primary key is the pair of <><*tcont, nconst*>. The tables are inner joined together following the arrows shown in Figure 1 and irrelevant columns are dropped to preserve memory resources.

2.3 Data Preprocessing

Although there are no default functions to read the TSV files and convert them into Dataframes directly like the CSV files, we can still use the *spark.read.csv* function with the "sep" parameter set to "\t" to specify that the file is a TSV file. However, the datasets that we choose contain multi-line rows, trailing white space, etc. Eventually, we decide to use *spark.read.option* function to handle the job for us:

```
spark.read.option("header","true")\
      .option("sep", "\t") \
      .option("multiLine", "true") \
      .option("quote","\"") \
      .option("escape","\"") \
      .option("ignoreTrailingWhiteSpace", True) \
      .csv("example.tsv").cache()
```

The *cache* at the end of the above code is also important since we need the raw data every time we take new user inputs. The size of the seven tables is the multiple of GBs and we should avoid reloading to happen. One remaining flaw is that the array of strings becomes a single string when it is loaded. For example, the column "knownForTitles" should contain multiple "tconst" like "tt0050419". However, the actual Dataframe will treat the array as a single string whereas the identifiers within are split by commas. Additionally, the integer and boolean types are also changed to the string type.

The above issues can be solved by using the functions within the *pyspark.sql.functions*. By using the comma delimiter, an array of strings would be generated for each row by applying the *split* function to the original string. The *cast()* function could be applied to cast the string in the column to its desired type.

3 IMPLEMENTATION

3.1 Movie Production Decision Maker

3.1.1 User Inputs. The Movie Production Decision-maker takes four user inputs:

- Genres
- IsAdult
- Maximum Age
- Vote/Rate Ratio

The "Genres" indicates the movie genres including action, adventure, animation, biography, comedy, crime, etc. The full list could be found in the *IMDbBigData python notebook* [2]. The user could choose to input "*" to include all movie genres. The data type for genre input would be a string containing all movie types that the user would like to include separated by commas. The "IsAdult" option would help to filter out whether the movie belongs to the adult category. "Maximum Age" serves the purpose of limiting the crews' age.

The "Vote/Rate Ratio" defines how the user would like to balance out the popularity and the good rating of a movie. The Vote/Rate Ratio that the user provides should be a floating point number between 0 and 1. As we can see from Equation 1, we use the number of votes times the average rating of a movie to get a number for each movie. This value is called "Vote/Rate Value". We then use the maximum Vote/Rate Value as the base to multiply the Vote/Rate Ratio. This final value would be the filter on how much balance the user wants. The larger the value is, the more popular and better rating movies are sampled. One thing that we have to be careful of is the difference between the number of votes for the movies. Some movies could have millions of votes with a bad rating, but others could only have less than ten votes with a good rating.

$$Max\langle NumVotes \times AverageRating\rangle \times \langle Vote/RateRatio\rangle \quad (1)$$

3.1.2 Algorithmic Flow. As Figure 3 shows, the algorithmic flow of the movie production decision-maker could be divided into seven stages.

Titles in the title basics table can be categorized as movies, shorts, tv-series, etc. The first stage would be filtering out the non-movie-type titles. Then, we use the title rating table to calculate the Vote/Rate Value for each title in the table. As mentioned in the data prepossessing section, we need to cast the data type of "averageRating" and "numVotes" to float. Now that the maximum Vote/Rate Value is derived by sorting the table in descending order, we have the final filter value by multiplying it by the Vote/Rate Ratio. By inner-joining the title rating table and the title basics table, we now have a new table that is both valid for being filtered using the Vote/Rate filter value and having movie genres data for continuing the third stage.

"Genres" in the third stage is hard to filter since it is now still a string containing different movie genres separated by commas. Firstly, considering the user input for genres could be mixed-case words, both the user input genre string and the string in the "genres" column would be converted into lowercase. Then, we add a new column "genresCompareI" containing the same user input string for each row. This is used to compare with the original "genres" column, which is now being aliased to a new column "genresCompare0" to avoid muddling the data. Then, the string in both new columns is split by using the delimiter comma to generate an actual array of genres string. Finally, the *array intersect* function would determine whether the user input genres have anything in common with the actual movie genres. "isAdults" would also be filtered in this stage. After the third stage, everything related to movie filtering is done.

The current table in stage four would join with the principal's crews table and name basics table to get the movies' crews and their information. Then, the user inputted "Maximum Age" would be used to filter the crews that are older than this input. The current table is being cached here since we need to join this table later. One thing to note is that the people that have death year on the table would also be filtered out silently.

In stage five, an optimized way of rating the crews is introduced. As mentioned in the data source section [1], each crew has its

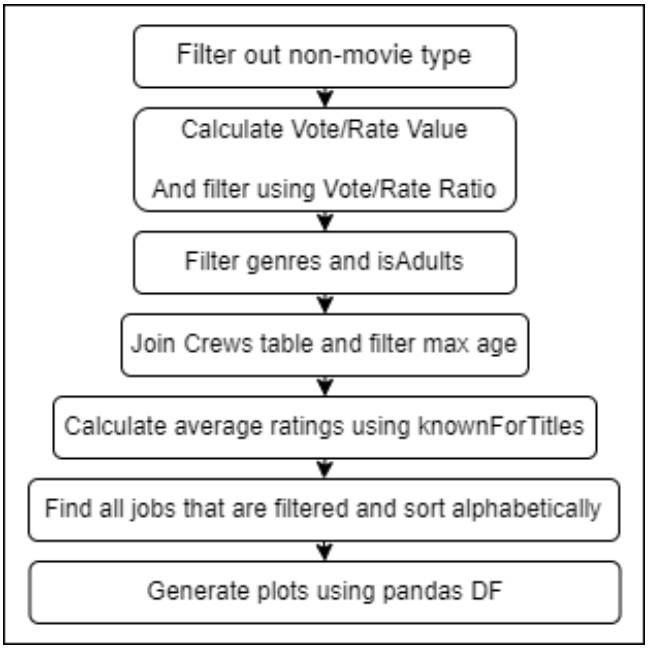

Figure 2: Movie production decision-maker algorithmic flow

"knownForTitles" data. The optimized rating of a crew would be the average of all of its known titles' ratings instead of one title. In order to achieve this, a *split()* is also required to convert the "knownForTitles" string to an array of "tconst" strings. An additional function *explode()* would also be introduced so that a single row of crew information would be exploded into several rows. Each newly generated row would contain a unique title coming from the original "knownForTitles" array. Followed by joining the title rating table with the current table, the new table would be ready for calculating the average rating of the titles. This is achieved by firstly using *groupBy()* to group the rows that have the same primary name and then using *avg()* to derive the average rating. Now, we used the previously cached table to join the current table so that we have a crew information table with its optimized average rating.

Since every time the principal crews in a movie are different, there would be certain types of jobs not presented in the table. To get the unique movie jobs from all crews, the "category" column is selected uniquely and collected. By using the list comprehension on the collected data, we now have a list of unique jobs. This list is also sorted alphabetically to be better presented afterward.

In the final stage, we would like to generate horizontal bar charts that display the best-qualified crews for every unique job. Initially, by using *where()* on the Dataframe, we could have a new Dataframe that is the subset of the original Dataframe. The condition for *where()* is set to be every job in the unique job list. By using the list comprehension again, a list of new Dataframes is generated and each of the DataFrame within the list contains the data for the crews that have one unique job with different ratings. If there are no possible crews that satisfy the conditions, the list shall be empty. Eventually, the *pyspark* DataFrames are required to be converted to *pandas* DataFrames to draw the diagram, and only the best five rated crews are selected in each Dataframe to display.

3.1.3 Performance. The program takes around 2 mins to run on average. However, the very first run would take much longer since no actions are invoked initially. After the first run, all of the raw Dataframes are cached which leads to a performance boost.

3.2 Movie Finder

3.2.1 User Inputs. The Movie Finder takes the following inputs:

- Preferred Actor ('*' means any actor) & Actor Factor
- Preferred Actress ('*' means any actress) & Actress Factor
- Preferred Director ('*' means any director) & Director Factor
- Preferred Genre ('*' means any genre) & Genre
- Preferred Year ('*' means any year) & Year Factor
- Rating Factor
- XXX Factor means how much matching XXX matters (scale 0-10), XXX Factor is 0 if '*' is entered for the corresponding field

3.2.2 Decision Making Heuristics. Each movie has a score calculated based on the following set of formulas:

$$MovieScore = rF + aF + acF + dF + gF + yF$$

$$rF = MovieRating * RatingFactor$$

$$aF = ActorFactor/10 \; if \; matches$$

$$acF = actressFactor/10 \; if \; matches$$

$$dF = directorFactor/10 \; if \; matches$$

$$gF = genreFactor/10 \; if \; contains \; genre$$

$$yF = yearFactor/10 * (100 - abs(PreferredYear - year))/100$$

In such case, if any field is indicated as "doesn't matter" by the input of '*', that field would become irrelevant to the final score as its factor is set to 0. For the year category, the closer it is to the target, the higher score it will generate. Overall, the movie with the highest score is the most recommended movie.

3.2.3 Flow of The Program. The program provides a CLI interface for users to enter information on what kind of movie they are interested in.

After obtaining info from the user, the first thing the code does is filter the title basics Dataframe to include only movies by checking that the "titleType" column is equal to *movie*. This filtered Dataframe is then joined with the title basics Dataframe on the "tconst" column, which is present in both Dataframes. The resulting Dataframe is then filtered to include only certain columns and to lowercase the genres column.

Next, this filtered Dataframe is joined with the title principals Dataframe on the "tconst" column, which is present in both Dataframes. The resulting Dataframe is then joined with the name basic Dataframe on the "nconst" column, which is present in both Dataframes. Finally, the resulting Dataframe is filtered to include only certain columns.

The resulting Dataframe is then converted into an RDD (Resilient Distributed Dataset) and mapped to a new RDD that includes only certain columns and that converts some of the columns to different data types (rating to float and year to int).

Next, the code defines a variable called *ratingFactor* that is equal to the rating factor specified in the *userInfo* dictionary for the user. The code then creates a new RDD called *finalRdd* that is based on *rdd* and that includes only the movie title and the product of the movie's rating and the *ratingFactor*.

The code then checks the various fields in the userInfo dictionary to see if the user has specified any preferences for actors, actresses, directors, genres, or years. If the user has specified a preference for any of these fields, the code creates a new RDD based on *rdd* where each tuple inside the created RDD has the format of '(Movie Title, Weighted Matching Level for The Category)'. These RDDs are then unioned with the *finalRdd* RDD.

Finally, the code uses the *reduceByKey* function to sum the weighted matching level values for each movie title and then sorts the resulting RDD by the sum of the total score in descending order. The top movie is then returned as a recommendation.

4 RESULTS

4.1 Movie Production Decision Maker

As mentioned in the implementation section, the result of the program would be multiple horizontal bar charts representing recommendations for crews in different jobs. Figure 3 shows an example of the chart. There are five actors suggested for the given user input and Mr.Kevin Spacey has the best rating among all.

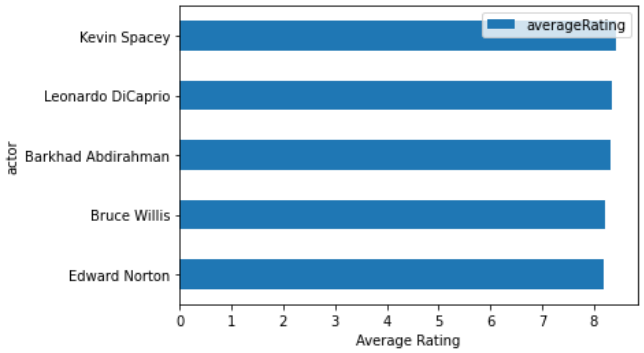

Figure 3: Example plot

4.2 Movie Finder

As mentioned in the implementation section, the result of the program is one concrete value: the name of the movie with the highest score:

- Input: userInfo = 'actor': ('Chris Evans', 9.8), 'actress': ('Elizabeth Olsen', 9.5), 'director': ('*', 0), 'rating': (", 10), 'genre': ('*', 0), 'year': (2019, 10)
- Output: (Avengers: Endgame, 11.33)

However, this does not always work despite the program being written correctly (for more details, please see the python notebook), and we believe this is because the extremely high memory consumption triggers resource constraints on Google Colab, as the number of tuples in the final RDD before reducing is approximately at the magnitude of tens of millions (about 1.2 million movies in total and each movie has multiple corresponding tuples). Such observation leads us to think why the *Movie Production Decision Maker* always gives a proper output in under 2 mins but the *Movie Finder* can often "explode" as they are analyzing the same raw data. Our conclusion is that this is caused by the performance difference between RDD and DataFrame in Spark. A further discussion is included in the performance section.

5 PERFORMANCE ANALYSIS

5.1 Movie Finder

In general, Spark DataFrames offer higher performance and more efficient operations compared to RDDs (Resilient Distributed Datasets). This is because DataFrames are built on top of the Catalyst optimization engine, which includes a number of performance enhancements such as query optimization, code generation, and data partitioning.

RDDs, on the other hand, are lower-level data structures that do not have the same optimization capabilities as DataFrames. RDD operations are typically slower and less efficient than DataFrame operations because they do not benefit from the Catalyst optimization engine.

Here are some specific ways in which DataFrames may offer better performance compared to RDDs [3]:

- Efficient execution: DataFrames use query optimization and code generation to execute queries more efficiently. This can result in faster execution times and lower resource usage.
- Lazy evaluation: DataFrames use lazy evaluation, which means that transformations are not actually executed until an action is performed. This allows the Catalyst optimization engine to analyze the entire query plan and apply optimizations before execution, which can result in more efficient execution.
- Data partitioning: DataFrames can automatically partition data based on the data distribution, which can improve the performance of distributed operations such as join and groupBy.
- Columnar storage: DataFrames store data in a columnar format, which can be more efficient for certain types of operations (e.g. aggregations, filters) because it allows the engine to skip over unneeded data.

It's worth noting that the performance difference between RDDs and DataFrames can vary depending on the specific operations being performed and the size and complexity of the data. In some cases, RDDs may offer better performance for certain types of operations. However, in general, DataFrames are a more performant choice for data processing in Spark.

6 POSSIBLE IMPROVEMENTS

6.1 Movie Production Decision Maker

One of the most important improvements of this program is the complexity of the algorithm that evaluates the crews. The basis of the algorithm is the Vote/Rate Ratio and the optimized average ratings from the crews' famous titles are used to better refine the result. One possible enhancement is to consider the number of votes

when calculating the optimized average ratings. This would further improve the balance between popularity and ratings.

Additionally, we could also consider the correlation between the crews. For example, if one actor once cooperated with a director in a low-rating movie and then participated in other high-rating movies respectively on their own, we should somehow lower the chance for them to cooperate again. Furthermore, we could also take advantage of the unused data. For example, one actor has participated in a high-rating TV series, but he has never acted in a high-rating movie. The weight of this actor should also increase considering his previous experience.

6.2 Movie Finder

The code could be made more efficient by reducing the number of unnecessary data transformations and data shuffles. For example, the code currently filters the title basics DataFrame multiple times and performs multiple joins and selects, which can be expensive operations. It might be more efficient to perform these operations in a single pass. Similarly, instead of using a union to glue multiple RDDs together with a lot of zero-value entries, it can achieve the same goal by using consecutive *map()* calls with significantly less memory consumption.

The code could be made more flexible or accurate by introducing more parameters. For example, the code currently only accepts several preference categories (e.g. actor, actress, director). It might be more flexible and accurate if the program allows users to identify all relevant categories that they want to match and have more customized heuristics for each category.

The code could be made more robust by adding error handling and input validation. For example, the code currently assumes that all of the input data (e.g. title basics, title ratings, etc.) are valid and well-formed, but this may not always be the case. It might be useful to add checks to ensure that the input data is valid before processing it.

7 CONCLUSION

In this project, we implemented a movie production decision-maker that helps to figure out the best crews to make a film. It helps movie producers make informed decisions about which crews to hire for their film projects, which can impact the quality and success of the final product. We also created a movie recommendation system that suggests movies to a user based on their preferences. It provides a convenient and personalized way for users to discover new movies that they may enjoy, which can enrich their viewing experience and expose them to a wider range of film genres and styles.

It is worth noting that the IMDb datasets are large. The system requires significant processing power to manipulate and analyze if we need to use RDDs and invoke actions like *count()* or *reduce()*.

REFERENCES

[1] IMDb Copyright Agent, "IMDb Datasets," IMDb. [Online]. Available: https://www.imdb.com/interfaces/. [Accessed: 14-Dec-2022].

[2] Zhang, M. and Lin, Y. (no date) IMDbBigData/IMDbBigData.ipynb at main · MINGRUIZHANGW/IMDbBigData, GitHub. Available at: https://github.com/MingruiZhangW/IMDbBigData/blob/main/IMDbBigData.ipynb (Accessed: December 14, 2022).

[3] L. Arora, "Differences between rdds, Dataframes and datasets in Spark," Analytics Vidhya, 25-Aug-2022. [Online]. Available: https://www.analyticsvidhya.com/blog/2020/11/what-is-the-difference-between-rdds-dataframes-and-datasets/. [Accessed: 14-Dec-2022].