CA4022 Assignment 1

Adam Power - 17329051
All code used available at https://github.com/adampower48/ca4022/assignment1 after the submission deadline.

Dataset

The dataset used is MovieLens Small, consisting of 100,000 user ratings across 9000 movies and 600 users. The data is split into 4 files: links, movies, ratings, and tags. I will be focusing on the movies and ratings, which are described below.
https://grouplens.org/datasets/movielens/

Movies

This file consists of 3 fields:

- movieId: unique identifier of the movie
- Title: title and year the movie was released
- Genres: List of genres separated by a “|” character

Ratings

This file consists of 4 fields:

- userId: unique identifier of the user
- movieId: unique identifier of the movie
- Rating: user rating for the movie on 0-5 scale
- Timestamp: unix timestamp for the rating

Pig Data Cleaning

There were two things I needed to clean for the movies file: one was separating the title and year from the title field, and the second was splitting the genres into a list. For the title and year, I used the REGEX_EXTRACT function to extract the pieces of information. The patterns used are as follows:

- Year: "\(\(\d+\)\)" - This pattern matches any number of digits surrounded by round braces, and returns the digits. https://regex101.com/r/zJO7Gq/1
- Title: "\([\S]+) \(\d+\)" - This matches any number of non-space characters followed by a space, ending in the pattern for year. It returns just the first part which is the title. https://regex101.com/r/yMenYm/1

For the genres, I simply split them by the "|" character using the STRSPLIT function. This left me with four fields: movieId, title, year, genres which I could then save to a file.

```
-- Split out year, title. Split genres (| must be escaped with \\)
movies = foreach movies generate
    movieId,
    REGEX_EXTRACT(title, '\\((\\d+)\\)', 1) as year,
    REGEX_EXTRACT(title, '([\\S ]+) \\(\\d+\\)', 1) as title,
    STRSPLIT(genres, '\\|') as genres;
```

movieId	year	title	genres
1	1995	Toy Story	(Adventure,Animation,Children,Comedy,Fantasy)
2	1995	Jumanji	(Adventure,Children,Fantasy)
3	1995	Grumpier Old Men	(Comedy,Romance)
4	1995	Waiting to Exhale	(Comedy,Drama,Romance)
5	1995	Father of the Bride Part II	(Comedy)
6	1995	Heat	(Action,Crime,Thriller)
7	1995	Sabrina	(Comedy,Romance)
8	1995	Tom and Huck	(Adventure,Children)
9	1995	Sudden Death	(Action)
10	1995	GoldenEye	(Action,Adventure,Thriller)
11	1995	American President, The	(Comedy,Drama,Romance)
12	1995	Dracula: Dead and Loving It	(Comedy,Horror)
13	1995	Balto	(Adventure,Animation,Children)

Challenges

The biggest challenge I had with pig was figuring out how to load and save the data. Importing the data from a CSV file with headers is impossible as far as I could tell. I tried three different functions for this: the native PigLoader, and CSVLoader and CSVExcelStorage (both from the piggybank package). In the end these all worked the same in that I could load a text file separated by a given character, but none would handle column headers. A workaround was to explicitly name the columns and data types, and filter out rows that failed to convert properly to the datatype.

```
-- Load data
ratings = load 'input/ml-latest-small/ratings.csv' using CSVLoader(',') as (userId:int, movieId:int, rating:double, timestamp:int);
ratings = filter ratings by userId is not null; -- remove first line (headers)
```

Likewise, exporting the data to a single file was also difficult. There were no external packages I could find to help me with this, so I had to manually create the csv file. I used PigLoader to store the data into several tab-separated files, another file for the headers, along with some other automatically generated files. I used hdfs commands within the pig shell to remove the junk files, and merge the remaining files into a single tab-separated csv file.

```
-- Save csv
fs -rm -r -f output/movies -- remove old dir
store movies into 'output/movies' using PigStorage('\t', '-schema'); -- Save parts, headers & other gunk
fs -rm -f output/movies/.pig_schema -- Remove schema file
fs -rm -f output/movies/_SUCCESS -- Remove success file
fs -getmerge output/movies output/movies.csv; -- Merge into single file
fs -rm -r -f output/movies -- remove gunk
fs -rm -f output/.movies.csv.crc; -- Remove gunk
```

In addition to the data loading problems, debugging with the pig shell was also challenging. The error messages were more often than not unclear and did not point to what was their cause. It took me quite a while to build up any intuition around this.

Pig Analysis

For the analysis of the data, I found that pig was not suitable for more complex queries. I stuck with computing simple aggregations which I could then use for more detailed analysis with hive.

Ratings by Movie

First I aggregated the ratings by movie and rating value. This would give me more compact data to work with. Here I simply counted the number of ratings for each unique (movieId, rating) pair.

```
-- Aggregate ratings
agg_ratings = group ratings by (movieId, rating);
ratings_counts = foreach agg_ratings generate group.movieId, group.rating, COUNT(ratings) as num_ratings;
```

Next I calculated the average rating using this aggregate table. This would have been easier to do with the raw ratings data, but I wanted to test nested queries for pig. Here I calculate the total rating for each rating value, then calculate the average rating per movie with this.

```
-- Calculate the average rating for each movie
groups = group ratings_counts by movieId;
movie_avg_ratings = foreach groups {
    mul = foreach ratings_counts generate rating * num_ratings;
    generate group as movieId, SUM(mul) / SUM(ratings_counts.num_ratings) as avg_rating;
};
```

Bringing this all together, I joined the movie data with both rating aggregations and exported the final table as a csv.

```
-- Join tables together and clean up
movie_ratings = join movies by movieId, movie_avg_ratings by movieId, movie_total_ratings by movieId; -- Join with title data
movie_ratings = order movie_ratings by movie_avg_ratings::avg_rating desc; -- Sort by avg rating
movie_ratings = foreach movie_ratings generate -- Fix headers
    movies::movieId as movieId,
    movies::year as year,
    movies::title as title,
    movies::genres as genres,
    movie_avg_ratings::avg_rating as avg_rating,
    movie_total_ratings::num_ratings as num_ratings;
```

	movieId	year	title	genres	avg_rating	num_ratings
1	5416	2002	Cherish	(Comedy,Drama,Thriller)	5.0	1
2	5468	1957	20 Million Miles to Earth	(Sci-Fi)	5.0	1
3	71268	2009	Tyler Perry's I Can Do Bad…	(Comedy,Drama)	5.0	1
4	5490	1976	The Big Bus	(Action,Comedy)	5.0	1
5	112512	2010	Colourful (Karafuru)	(Animation,Drama,Fantasy,My…	5.0	1
6	5513	2002	Martin Lawrence Live: Runt…	(Comedy,Documentary)	5.0	1
7	5537	2002	Satin Rouge	(Drama,Musical)	5.0	1
8	5607	2001	Son of the Bride (Hijo de …	(Comedy,Drama)	5.0	1
9	5723	1981	Continental Divide	(Comedy,Romance)	5.0	1
10	5745	1981	Four Seasons, The	(Comedy,Drama)	5.0	1

Ratings by User

The next thing I wanted to do was calculate summary statistics for the users. This would allow me to build on them and do more complex analysis in hive. First I calculated the same thing as I did for the movie ratings: the count and average.

```
-- Calculate summary statistics for user ratings
g = group ratings by userId;
user_avg = foreach g generate -- Get avg user rating and number of ratings
    group as userId,
    AVG(ratings.rating) as avg_rating,
    COUNT(ratings) as num_ratings;
```

I also wanted to calculate the standard deviation of the scores. This would allow me to compare how consistent the users are in their ratings. Unfortunately, pig does not have a builtin function to do this, so I had to do it manually. It is defined as:

$$\sigma = \sqrt{\frac{\sum(x-\mu)^2}{N-1}}$$

There are 3 steps to this calculation which need to be handled separately using the map-reduce paradigm:

1. Map: Subtract the mean rating (which we already have) from each rating, then square it.
2. Reduce: Sum up the squared differences
3. Map: Divide by the number of ratings minus one, and take the square root.

The first two steps are straightforward, I used an intermediate “diff” variable before squaring it, as I was not able to do this in one line.

```
-- compute standard deviation (step 1)
user_ratings = join ratings by userId, user_avg by userId;
usr_ratings = foreach user_ratings {
    diff = ratings::rating - user_avg::avg_rating; -- Difference
    generate ratings::userId as userId,
             diff * diff as diff_sq; -- Squared
};
```

```
-- compute standard deviation (step 2)
g = group usr_ratings by userId;
usr_ratings2 = foreach g generate
    group as userId,
    SUM(usr_ratings.diff_sq) as sum_diff_sq;
```

The final step is more complex. When there is only a single rating for a user, there is no standard deviation. Here I use the CASE statement to set the standard deviation to 0 when there is only one rating, and use the formula above otherwise.

```
-- compute standard deviation (step 3)
usr_ratings3 = join usr_ratings2 by userId, user_avg by userId;
user_rating_stats = foreach usr_ratings3 generate
    usr_ratings2::userId as userId,
    user_avg::avg_rating as avg_rating,
    (
        case user_avg::num_ratings
        when 1 then 0
        else SQRT(usr_ratings2::sum_diff_sq / (user_avg::num_ratings - 1)) end
    ) as std,
    user_avg::num_ratings as num_ratings;
```

This leaves me with a new table with the summary statistics for each user. Again I saved this as a csv using the workaround mentioned earlier.

userId	avg_rating	std	num_ratings
1	4.366379310344827	0.8000480467733448	232
2	3.9482758620689653	0.8056145345791144	29
3	2.4358974358974357	2.090641701977143	39
4	3.5555555555555554	1.314203858975363	216
5	3.6363636363636362	0.9904405665441197	44
6	3.4936305732484074	0.850647678719864	314
7	3.2302631578947367	1.3295938469468664	152

Hive Analysis

With hive, there is a further level of abstraction, and I can use HiveSQL to work in a real relational database setting.

Loading data

The first thing I did was create the database, create the tables and load the data into the tables. This was much easier than with pig, as there are additional table properties you can set such as skipping the first line as a header.

```
-- Create database
drop database if exists movielens cascade;
create database if not exists movielens;
use movielens;
```

```
-- Create Tables
--      User ratings
create table user_rating_stats
(
    userId int,
    avg    double,
    std    double,
    n      int
)
    row format delimited fields terminated by "\t"
    tblproperties ("skip.header.line.count" = "1");

--      Movies
create table movies...;

--      User Ratings
create table ratings...;
```

```
-- Load data into tables
load data inpath "/user/adam/input/pig_output_data/users.csv" overwrite into table user_rating_stats;
load data inpath "/user/adam/input/clean_data/movies.csv" overwrite into table movies;
load data inpath "/user/adam/input/ml-latest-small/ratings.csv" overwrite into table ratings;
```

Normalise user ratings

One thing I wanted to look at was normalising the ratings users give to the movies. This would help account for different users, such as “optimistic” users who rate all movies good, or “passionate” users who rate movies either extremely good or bad. Normalising the scores would mitigate these differences between users somewhat. I used the summary statistics that I calculated previously with pig to do this. Here I calculate the normalised z-score for each user, again accounting for when the standard deviation is 0. The resulting scores will have a mean of 0 and a standard deviation of 1.

$$z = \frac{x - \mu}{\sigma}$$

```
-- Normalise user ratings
create table norm_user_ratings as
select ratings.userid,
       movieid,
       if(std == 0, 0, (rating - avg) / std) as norm_rating
from ratings
         join user_rating_stats urs
              on ratings.userId = urs.userId;
```

I then joined both the raw ratings and normalised ratings with their movies and calculated both averages for each movie.

```
-- Add ratings & normalised ratings to movies
create table all_ratings as
select movies.*, r.userId, r.rating, nur.norm_rating
from movies
         join norm_user_ratings nur on movies.movieId = nur.movieid
         join ratings r on nur.userid = r.userId and
                           nur.movieid = r.movieId;
```

movieId ▲ 1	title	year	userId	rating	norm_rating
1	Toy Story	1995	1	4.0	-0.4579466343583529
1	Toy Story	1995	5	4.0	0.36714607208101
1	Toy Story	1995	7	4.5	0.9549809853745547
1	Toy Story	1995	15	2.5	-0.8365491046057494
1	Toy Story	1995	17	4.5	0.571252302326406
1	Toy Story	1995	18	3.5	-0.3671083261391037
1	Toy Story	1995	19	4.0	1.501780168363096

```
-- Calculate average of ratings
create table movie_averages as
select movieId,
       title,
       avg(rating)      as avg_rating,
       avg(norm_rating) as avg_norm_rating
from all_ratings
group by movieId, title;
```

movieId	title	avg_rating	avg_norm_rating
1	Toy Story	3.9209302325581397	0.34237854367196446
2	Jumanji	3.4318181818181817	-0.05879114574333638
3	Grumpier Old Men	3.2596153846153846	-0.23861515979569675
4	Waiting to Exhale	2.357142857142857	-1.1880643032348404
5	Father of the Bride Part II	3.0714285714285716	-0.626767782737354
6	Heat	3.946078431372549	0.42291859846498064
7	Sabrina	3.185185185185185	-0.43897223320340656
8	Tom and Huck	2.875	-0.5793577549178015
9	Sudden Death	3.125	-0.5008660041476464
10	GoldenEye	3.496212121212121	-0.06403586613318636
11	American President, The	3.6714285714285713	0.13775812493131623
12	Dracula: Dead and Loving It	2.4210526315789473	-0.8942497604305828
13	Balto	3.125	-0.30197050535645725

Ratings for genres

To analyse the data for genres, I had to first do some data processing. While I had split up the genres using pig, in saving the data it introduced more formatting I had to deal with.

genres
(Comedy,Drama,Thriller)
(Sci-Fi)
(Comedy,Drama)
(Action,Comedy)
(Animation,Drama,Fantasy,My…
(Comedy,Documentary)
(Drama,Musical)
(Comedy,Drama)
(Comedy,Romance)
(Comedy,Drama)

To split these up, I used the regexp_replace function to remove the brackets, the split function to split by the “,” character, and the explode function to un-pivot the table, putting one genre on each line.

```
-- Genres: split into rows
create table genres as
select movieId, genre
from movies lateral view explode(split(regexp_replace(genres, "[\(\)]", ""), ",")) genres as genre;
```

When I had the genres processed, I joined them with the movie ratings, and grouped the ratings by genre. I then calculated the average and standard deviation of the movie ratings for each genre. Hive has a builtin std function, so this simplified the calculations a lot.

```
-- Get average & std rating per genre
create table genre_ratings as
select genre,
       count(*)             as num_movies,
       avg(avg_rating)      as avg_rating,
       std(avg_rating)      as std_rating,
       avg(avg_norm_rating) as avg_norm_rating,
       std(avg_norm_rating) as std_norm_rating
from genres
         join movie_averages on genres.movieId = movie_averages.movieId
group by genres.genre;
```

genre	num_movies	avg_rating	std_rating	avg_norm_rating	std_norm_rating
Action	1828	3.0944984491955134	0.8380389688082553	-0.33354401355853297	0.7957473735648707
Adventure	1262	3.215229808197848	0.7919604505537718	-0.22647991586746272	0.7442443737237839
Animation	610	3.497119150128774	0.9016368122425888	0.08300471693452327	0.8606101300319705
Children	664	3.1076903605293134	0.9070572057402113	-0.29933089723048645	0.8164674997716566
Comedy	3753	3.181716291867792	0.8835220757273569	-0.20988979624795837	0.8053833494229503
Crime	1196	3.301843831031174	0.8187938043299933	-0.10471062320067333	0.7359196265601671
Documentary	438	3.7816816901269963	0.710285000093363	0.3294455240764658	0.7355783195356165

Challenges

Similarly to pig, I had difficulties exporting the data to single csv files. It is not possible to use hdfs commands inside of hive, so I had to export the data parts into a hdfs folder, bring it back to my local folders, and then combine the files and manually add column headers using bash scripting.

Export parts:

```
-- Save genre ratings
insert overwrite directory "/user/adam/output/genre_averages"
    row format delimited fields terminated by '\t'
select *
from genre_ratings;
```

Bash helper functions to combine into tab-separated csv:

```
join() {
  # Joins strings with a tab
  local headers=$1
  for ((i=2;i<=$#;i++))
    do
      printf -v headers '%s\t%s' "$headers" "${!i}"
    done
  echo -e "$headers"
}

create_tsv() {
  # Concat files in folder and add given headers
  local filename=$1
  echo -e "$(join ${@:2})" > $filename.csv
  for part in $(ls $filename)
  do
    cat $filename/$part >> $filename.csv
  done
}
```

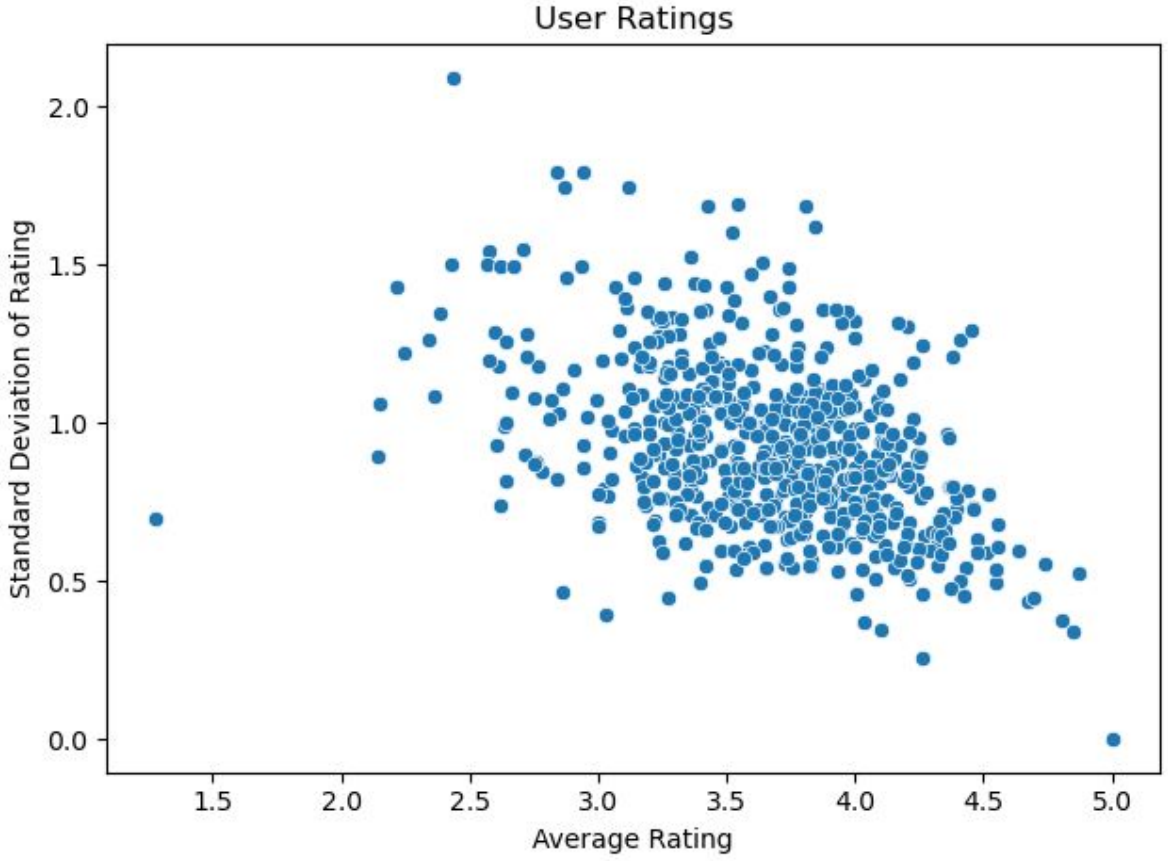
Use of the helper functions:

```
# Combine outputs into tsv files
create_tsv output/movie_averages movieId title avg_rating avg_norm_rating
create_tsv output/genre_averages genre num_movies avg_rating std_rating avg_norm_rating std_norm_rating
create_tsv output/genres_split movieId genre
create_tsv output/movie_ratings_all movieId title year userId rating norm_rating
# Remove old folders
rm -r output/movie_averages
rm -r output/genre_averages
rm -r output/genres_split
rm -r output/movie_ratings_all
```

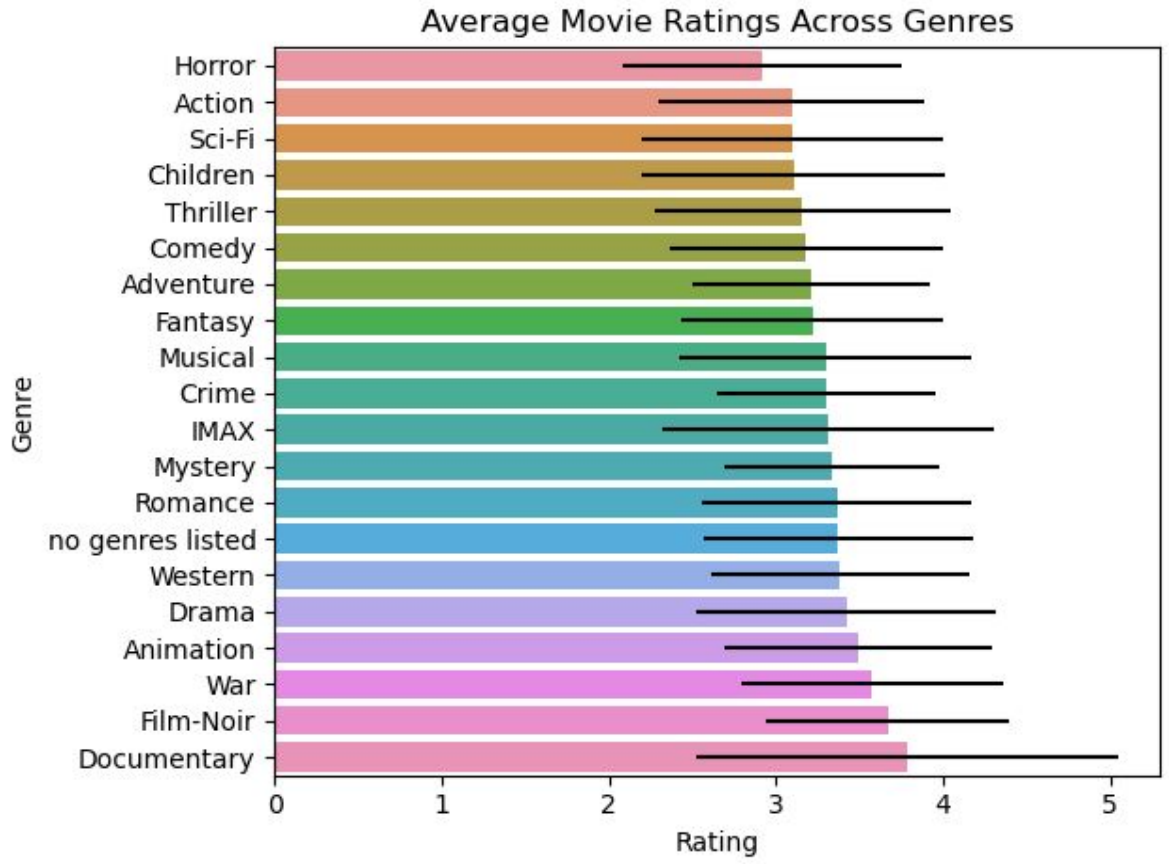
Visualising the Data

While hive is good for doing calculations on the data, it doesn't have any visualisation capabilities. Instead I used Python alone with the seaborn package to create plots of the data.
https://seaborn.pydata.org/

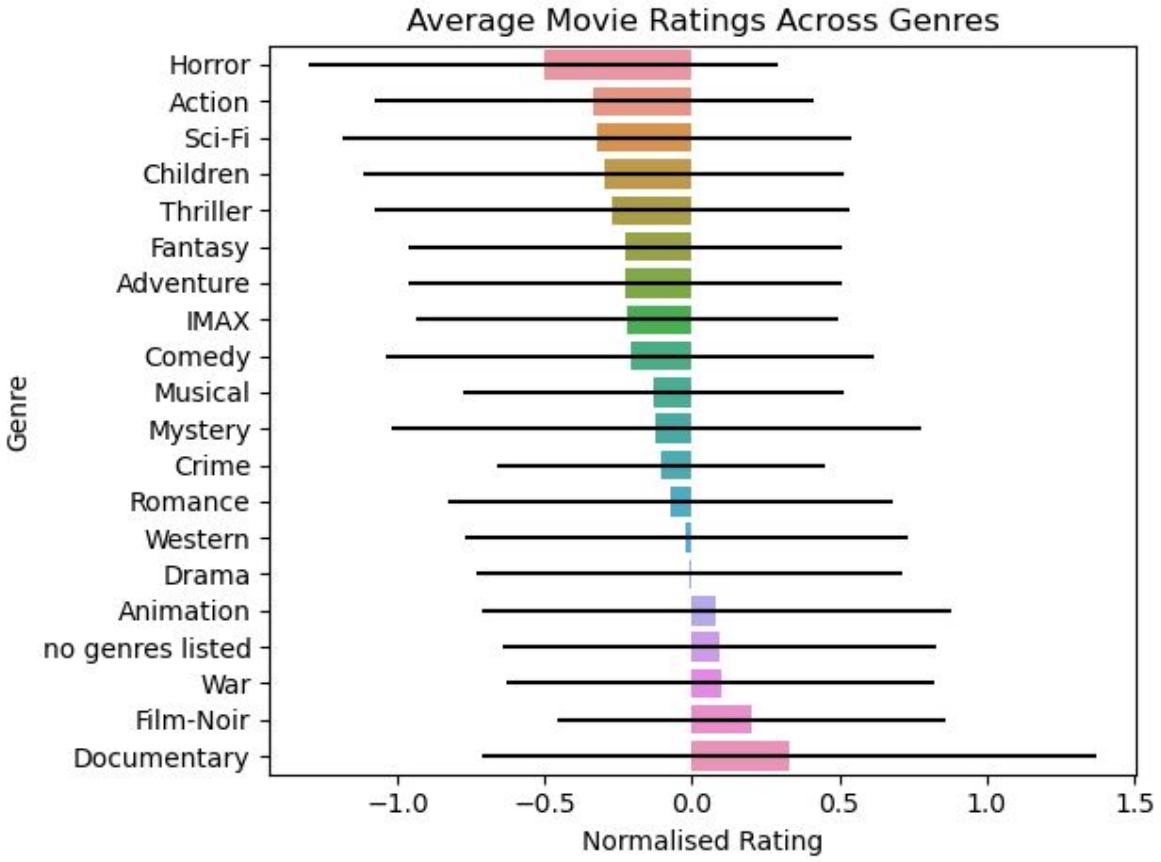
Looking at the distribution of ratings for each user, we can see that the majority of users have average ratings around 3.5-4 and standard deviation of around 0.75. In the bottom right, there is one example of an extremely "optimistic" user, who only rated movies a 5. Towards the top there are some "passionate" users who have a very high variation in their ratings.



On average, we see that all of the genres perform within 1 point of each other. The black bars show one standard deviation of the scores. The documentary genre is rated the highest by a decent margin, however there is a much larger variation in the scores. This shows that while there are a lot of very good documentaries, there are also quite a few very bad ones. Horror movies are by far the least liked, with a similar variation to the other genres.



With the normalised ratings, the story is pretty much the same. The ranks of some genres have moved slightly but they are generally unchanged. What we do see however, is that the majority of genres are rated below average. The variation for the mystery genre has increased substantially compared to the raw scores.



Looking at the raw and normalised ratings, there is definitely a correlation between them. However there is also a large variation: A raw rating of 1.5 for one person was “above average”, and a rating of 4.5 for another was “below average”.