Search engine for generating a movie cast

My project subtitle

Christoffer Holmesland
University of Stavanger, Norway
cr.holmesland@stud.uis.no

Malavika Ramakrishnan
University of Stavanger, Norway
m.ramakrishnan@stud.uis.no

ABSTRACT

In this report we show how we used data from IMDB[**?**] to generate a movie cast based on user input. To determine the recommended cast we calculate a score based on genres, actor to actor relationship, and a brief movie summary. The user is presented with a list of actors and the predicted IMDB user rating score.

KEYWORDS

Hadoop, Spark, Film, Casting, Search engine

1 INTRODUCTION

Context and Motivation. For a low budget movie it is important to have a well-known actor. It can increase the box office sales or help attract investors[**?**]. If you are new to the film industry it can be hard to know which actors or actresses will be a good fit for your movie. It is possible to use previous movies where you liked the performance of the actor as inspiration, but there are too many movies being made that it is not possible to get a proper overview of all the possibilities. Another possibility is to look at websites like IMDb[**?**] to find inspiration. You can use features like the user score of a movie to select actors. We do not consider this as a good option because there are too many movies to compare, and in practice you will end up looking at only the top rated movies.

Research Problem. In this report we present a solution to the problem of selecting a movie cast. Our system lets the user input a brief summary of their movie and a list of actor characteristics they want the lead actors to have. The system uses data from IMDb to rank possible actors and returns a list of suggestions to the user. The rank is based on a combination of past performance, acting relationships and a subset of the actor characteristics.

Contribution Summary.

- We created an efficient method of calculating actor relationships on a large scale.
- We implemented a search engine returning single actors.
- We implemented a search engine returning groups of actors.

2 RELATED WORK

The IMDB datasets have been used to predict the gross movie revenue[**?**]. The accuracy of several machine learning models were compared to find the best way of predicting the gross revenue. The results show that the random forest model had the best performance and that the most important feature is the number of user votes. The datasets have also been used to compare the user rating with other features[**?**]. In the same analysis they found that lead actors are typically ten years older than lead actresses.

The movie project system cinelytic developed by Cinelytic, Inc.[**?**] is able to analyze and produce forecasts of how well a movie will perform based on country, cast and distributor. The system ranks actors based on the predicted economic increase they have for a given project.

Our system is similar to cinelytic, but we are basing the score only on IMDB data and use a subset of the features considered by cinelytic.

3 BACKGROUND

The Internet Movie Database, commonly known as IMDb is a website with a lot of information about movies. It contains almost every movie that has ever been made and lets you look at details like actors, genres, box office and many more. Users are able to create an account and rate movies between one and ten stars. This is what we consider to be the user rating. It is important to note that the user rating of a title is not just the average of the ratings. IMDb applies filters[**?**] to the votes to prevent manipulation which means the rating is more like a weighted average. The box office value of a movie is the number of theater tickets sold times the cost of a ticket. Some people use it for all sales, for example by including digital sales, but this is not done by IMDb. A movie cast is the actors who act in a given movie. The word comes from the casting processes which is when the director picks actors for the movie.

4 HADOOP CLUSTER

Our Hadoop cluster consists of one name node and three slave nodes. The name node has 8 GB RAM and a 2.4 GHz Intel Skylake processor with 4 cores. The three slave nodes have 4 GB RAM each and a 2.4 GHz Intel Skylake processor with 2 cores. We used Hadoop version 3.2.1, Spark version 3.0.0-preview2 and Python version 3.5.2.

The Spark documentation recommends using at least 8 GiB memory and between 8-16 cores on each node[**?**]. Because our cluster does not meet those requirements, we had to make some changes to the configuration files. We are running the project as a YARN job and we had to keep in mind that the maximum amount of memory configured in YARN had to be greater than what we configure in Spark. It is not recommended to use more than 75% of the available memory so we decided on 3 GB. The initial amount of memory Spark uses on startup is defined in the spark.driver.memory. The default value is 1GB which we noticed was a bit too much for our cluster. Instead we used 512MB. The Spark driver only runs if we are running Spark in cluster mode. If Spark runs in client mode, the value specified in spark.yarn.am.memory is used instead so this was set to 512MB as well. The other Spark configuration value we

Project in Data-intensive Systems (DAT500), IDE, UiS
2020.

had to set is spark.executor.memory. This is the memory available to each executor on the worker nodes. When deciding on the value it is important to keep in mind that there is a default overhead of 7%, and that the Java virtual machines also require some memory to run. The minimum value of the 7% overhead is 384MB[**?**] which means that the value we decide on had to be lower than what 7% would allow. To be sure that there was enough memory we decided on a spark.executor.memory value of 1512MB.

5 YOUR METHOD

5.1 Getting the data

First we had to get all of the required data. Most of it can be downloaded from the IMDb website[**?**]. This data does not include actor gender, movie plot or box office values. To determine the gender we used the list of known professions for every person. If they are known for being a actor they are assigned the label "0". Actresses are assigned the label "1". Some people in the dataset have never acted, for example directors or producers. They are removed from the data. There is no way to find the movie plot in the datasets, so we had to find an external source. One alternative is to use webscraping on the IMDB website. We tried doing this but quickly realized that it would not be possible for us because they stop responding to requests after too many in a given time period. We were able to get about 9 000 requests per hour. Since the dataset includes about 500 000 movies this would not be a good solution. It was also not possible for us to know if there were other rate limits e.g. 50 000 requests per day. Instead we used the OMDb API[**?**] to retrieve the movie plots. Using this API we were able to get all of the plots in 1 hour and 43 minutes. Our original idea was to also make a prediction on the box office value. After using the OMDb API to also find the box office values we saw that only 6381 movies have this value. We do not believe this is enough data to make proper predictions on (why?) so we did not use this data. Instead we decided to generate a prediction for the IMDb user score because this value is available for all of the movies in our dataset.

5.2 Preprocessing

The data from IMDb includes a lot of information we don't need and it also has missing values. It is also split over several files e.g. title.basics.tsv and title.ratings.tsv. Every person in the dataset has an id, name, birthyear, deathyear, a list of primary professions and a list of titles they are known for working on. We replace the primary profession list with a number indicating the gender. We remove people who are dead or who has never been an actor or actress before. There are also some people who are actors but not known for any titles. They are also removed because we need that information to calculate a score. These steps reduce the number of people from 9.9 million to 200 thousand.

The data in the title.basics.tsv file is an id, type, well known title, original title, adult, start year, end year, runtime, and a list of genres. The only relevant information for our project is the id, type and list of genres. The type is used to decide if we should keep it. We are only interested in looking at movies, represented by "movie" or "tvMovie" in the data. Shorts are typically news segments which would not be a good indicator of movie performance. If we included tv series then the actors in them would have inflated scores because there could be over 100 episodes for a given series. That would show that the person is good at that specific role, but not represent how well he or she can adapt to a new role. The runtime is not used in our system because it is missing for most of the titles. Some movies are missing the genre list and are thus removed because it is required for the system. This reduces the number of titles from 6.5 million to 580 thousand. This might seem like a lot, but most of it is because the data includes seperate entries for each season and episode for every tv series. In this step we also use the plot data from the OMDb API. Movies without the plot information are removed. This reduces the number further to 212 thousand (check this number again).

The movie ratings are in the title.ratings.tsv file from IMDB. The format is id, average rating and number of votes. Any title removed from title.basics.tsv is also removed from this data. The title.principals.tsv file contains a list of the most important people for each title. The fields are title id, ordering, actor id, category, job, characters. We use this information to determine what actors have worked together. We only look at the titles with id which was not removed in the previous step, and the ordering, category, job and characters fields are dropped because they provide no relevant information to us.

5.3 Ranking algorithm

The ranking algorithm is used to give a score to the actor groups. The score is given as a number between 0 and 10, where a higher score indicates a higher rank. The final score is a combiantion of three scores. We call the first one the genre score. It is calculated based on the actors previous performance. The second score is the similarity score. It is calculated from the content of the movies a person has acted in. The third and final score is the relation score between the actor and the other actors in the group. The final score is the average of these scores.

5.3.1 Genre score. To calculate the genre score we need to convert the movie data to a different format. The raw data we use are the ratings, principals and title files. They are read by a MRJob script which calculates the weighted average for each genre for each actor. The weights are the number of votes for that title. If an actor has acted in two movies, one with the genres "Adventure" and "Drama", the other with the genres "Adventure" and "Action" then the actor will have genre scores for the genres "Adventure", "Drama" and "Action". The "Drama" and "Action" scores will be the rating that those movies have on IMDb. The "Adventure" score will be a weighted average of both ratings. Assuming that the first movie had 10 votes and a rating of 9, and that the second movie had 90 votes and a rating of 3 then the "Adventure" score is $\frac{10}{10+90} \cdot 9 + \frac{90}{10+90} \cdot 3 = 3.6$. Our initial version of the genre score did not consider the number of votes. We quickly realised that a lot of movies have very few votes, and that those movies often have a high rating. It would be hard to set a threshold of minimum number of votes because it could exclude new actors who might not have a lot of experience.

5.3.2 Similarity score. The similarity score is calculated by comparing the summary plot given as input by the user and the summary of the movies acted by the actors with high genre scores. For this purpose, we tried the following methods:

- **Cosine similarity using Tf-idf:** The cosine similarity is the cosine of the angle between two vectors. In text analysis, each document can be represented as a vector. Mathematically, cosine is the dot/scalar product of two vectors divided by the product of their Euclidean norms. The lesser the value of cosine, the lesser the similarity between the two documents and vice versa. Tf-idf short for term frequency-inverse document frequency and is often used in information retrieval and text mining. The text in the documents are tokenized and lemmatised, then tf-idf measures the frequency of each word occurring in a document, and comes up with a tf-idf matrix whose similarity is then computed.
  This method depends very much on the number of words in a document. Which is why this was a bad choice for our task, since our plot summaries are small, with very few words. Nevertheless, we tried implementing this, but the results were highly dissatisfactory.
- **Word2Vec:** Word2Vec as the name sounds is a neural network that maps words to a vector and then analyses them mathematically. Its purpose is to cluster the vectors of similar words together in a vector space. That is, it detects similarities mathematically. Word2vec creates vectors that are distributed numerical representations of word features, features such as the context of individual words.
  This method was highly promising but yielded poor results since it was computing word-wise similarity while we needed sentence-synonymity.
- **Tensorflow model from Google:** The model[**?**] available in the TensorFlow Hub is also a form of word2vec model but is instead trained and optimized for greater-than-word length text, such as sentences, phrases or short paragraphs. This perfectly matched our criteria and gave excellent results. This model would return a value between 0 and 1 which we multiplied by 10 to normalise it like the genre score.

There is a large drawback to using a tensorflow model when the program runs on Spark. The model is quite large, almost one gigabyte. Usually a spark broadcast variable would be used to share an object between nodes but in this case it couldn't be done because the model cannot be pickled which is a requirement for broadcast variables. To be able to use it on the slaves each of them would have to load it into memory instead. To achieve this we defined closures for each partition of the data where the tensorflow library was imported and the model was loaded. Importing the tensorflow library is quite expensive. It usually halts for 30 seconds before the import is done. Loading the model is equally expensive, thus the total time on each partition is a minute before work can start. Because there is multiple partitions per node, the model is loaded several times on each of them. In our case this took so much time that the algorithm hadn't finished after several hours. We didn't consider this acceptable and looked for a different alterantive. The best solution we found was to do the calculations on the master node instead. Because it has more memory available we were able to load the model once and calculate the similarity score for each title using the rdd.toLocalIterator() generator of the DataFrame. This means that all of the data had to be transfered over the network to the master node, and then back again to the slave nodes when the calculations were done, but this was actually faster in our cluster than the alternative. The runtime of similarity score is typically about 8 minutes. The initial version was a bit slower, but when we decided to pre-allocate the lists where the similarity score is stored before being turned into a DataFrame the performance improved. We do however recognize that transferring the data over the network might is a bad option in general and think that if we had more time to work on this we would have looked for a better solution.

5.3.3 Relation score. The two previous scores are independent for each actor. We believe this is a good way of measuring the performance of each actor. However when combining the actors to groups we wanted to have some measure of how good two actors work together such that the groups were not just the top ranked candidates based on the independent scores. Our initial idea was to consider the first candidate list as the primary actor, and only include actors from the other candidate lists if they had acted with someone in the primary candidate list previously. This turned out to be a really bad way because there are not that many direct relationships in the data. When looking for alternatives we found the Spark GraphX framework and got inspired to make a relationship graph instead to achieve the same goal. GraphX cannot be used in PySpark so we had to implement the graph ourself. The main idea is that if two people have acted together then they have a relationship of strength equal to the IMDB rating of the title they worked on.

The initial version of this was implemented in Python using Pandas dataframes, because we found that the overhead of working on Spark made it harder to explore ideas quickly. The graph was stored in a dataframe where index is the multi-index from_node, to_node. The only column is what we call the inverse of the rating. The inverse of the rating is simply ten minus the rating. This is done because it enables us to use an algorithm to find the shortest path between two nodes on the graph. If you are unfamiliar with graph algorithms we would like to note that the shortest path doesn't actually mean as few nodes as possible. The proper description is the path which minimises the cost function between two nodes. The shortest path would be along the lowest numbers, which in turn means along the highest ratings. A graph with the rating itself as the edge cost would require an algorithm which maximises the cost. The maximum cost would be to never arrive at the goal node because the cost could always be increased by simply visiting a different node before visiting the goal node. We decided to implement Dijkstra's algorithm to find the path between two nodes. It finds the path by looking at the edges from the starting node, picking the one with the lowest cost and then looking at that node's edges. This continues until the goal node is found. Because we always look at the node with the lowest cost we can be sure that the path to the goal node actually is the shortest path. The original version of the algorithm works on two nodes at a time, but there exists variants that can find the path between one node and every other node.

Dijkstra's algorithm works great when all of the data is in memory like it is when it is stored in a Pandas dataframe. It is quite slow to execute because of the size of our data. The graph consists of almost one million nodes, and between 10 and 350 edges from each node. The execution depends on the nodes we are finding the

path between but in general the time is about 20 minutes. When we moved over to the Spark version of the algorithm we kept this in mind. Let's first make some assumptions. If we take the top 30 candidates from each candidate list, and there are three candidate lists, then we need to calculate $3 \cdot (30 * 30) = 2700$ relationship scores. If each of them needs 20 minutes to finish then the execution time would not be acceptable. Even if each score was calcualted in just one second the total time would be 45 minutes. When we implemented Dijkstra's algorithm on Spark the time runtime was even worse because each step requires that some data is transfered from a slave to the master and then back again. Because of they way Dijkstra's algorithm works state has to be shared between each worker, and thus data has to travel over the network. To avoid this we decided to implement our own algorithm instead which doesn't have that requirement. We tried to find previous work on graph algorithms on Spark, but all of them were using Spark GraphX which we couldn't do.

Our solution was to create a greedy fixed step algorithm inspired by the breadth first algorithm. The breadth first algorithm simply looks at all of the edges for a node before continuing. The reason for making it a greedy fixed step algorithm is that because of the size of our graph. As an example you can start at Leonardo DiCaprio and look at the number of edges. In our graph he is connected to 350 other actors. It would be really quick to calculate the shortest path if the goal actor was directly connected to Leonardo. For most relationships there is not a direct conncetion, and so we have to also look at the edges of each edge. This increases the number of paths to 58 339 which isn't that many considering the size of the graph. If the goal is still not found the number of paths is increased again by looking at the edges of the edges to 8 901 575. If we again assume 2700 relationships then the number of paths to check is over 24 billion. This is doable but it does increase the execution time a lot. If the relationship isn't found in those 8.9 million paths then another step outwards can be taken to increase the number of paths to 1.3 billion. This would be a total of 3.5 trillion different paths which is so large that the execution time would not be acceptable. We therefor decided that the fixed step should be either 2 or 3. This also makes sense at a conceptual level. If you consider the relationship between two actors A and B. If A worked with C and C worked with B then it makes sense to say that their working relationship is important. The same can be said about the relationship A-C-D-B which is a 3 step relationship. If the relationship is more distant it makes sense to value it lower than a close relationship. This caused us to change the cost function between two nodes. Instead of being just the inverse of the rating, it was changed to be the average of the inverse distance and the inverse rating. The combination of the 2/3 step algorithm and the new cost function means that the only close relationships will affect the group score. A close and good relationship increases the score, while a close and bad relationship decreases the score. Any distant, meaning greater than 2/3 steps, will not change the score.

Because we have not been able to find a description of a multi start and end node breadth first algorithm we would like to include a description of it. The first iteration of our algorithm is the normal breadth first algorithm. The graph is stored in a DataFrame, and the current paths are stored in another DataFrame. Start by taking the starting node and adding it and every edge from it to the DataFrame. Continue adding edges to the DataFrame n times, where n is the number of steps you are using. The resulting DataFrame should have a several columns where the first one is the starting node, the following columns should be the edge and cost to that edge. By applying a function to each row of the DataFrame we can calculate the cost from the starting node to the last edge node following the path along every edge in that row. If the goal node is found along the path then we return the cost, otherwise a token value is returned. The token value should be outside of the cost function domain, or be on an extreme of the domain meaning that in any situation it will not be considered as a possible path to the goal node. In our case we use a token value of -1 because the domain of our cost function is [0, 10]. The inverse of the cost function is returned and stored in a score column. Finally the score column is aggregated to find the maximum and this is returned as the relation score between the two actor nodes. By observing that calculating the path from A-B-C is done by a 2 step search from A is done no matter what our goal node is the algorithm can be improved by allowing a list of goal nodes. When applying the cost function simply check if any of the goal nodes are found along the path. This is a simple change that improves the exeuction time a lot. If you look at the score between one actor and 8 other actors the exeuction time is about 20 seconds for each relation if you are using the breadth first algorithm. With the multi goal optimisation the time is reduced to about 2.5 seconds for each relation. When calcualting the relation scores between the candidate lists A and B of size 30 the number of function calls is now reduced from 90 to just 30. By observing that the same calculations are done when finding the path from A1 to B and A2 to B we can improve it further to a multi start node algorithm. The most expensive part of the algorithm is transferring the graph edges to the current path. The number of times we do this grows with the number of times we call the merge function on two dataframes. By looking at the path from multiple start nodes to multiple goal nodes the number of merges is reduced and the number of function calls can be reduced to 1.

Following the assumptions from before the total time for a 2 step search is 129 seconds. This gives a time per relation of 0.048 seconds which means that compared to the original algorithm the execution time has been improved by a factor of 9500. The number of paths for the 3 step search is quite a bit larger but the improvement is still significant. The total exeuction time is 4402, giving a time per relation of 1.63 seconds which is a 280 times improvement.

The ranking algorithm is used to determine whether one actor is going to perform better than another. This is done by calculating a score for the actors, before combining them to get a score for the whole group. Our algorithm assumes the first actor description to be the primary actor and tries to maximise the cast rank based on this. The following is a description of our algorithm. The first step is to find every actor that matches the primary actor description. For every matching actor a score is calculated. It is a combination of the genre score, past acting relationship(maybe) and how similar the plots of the movies the actor has acted in are to the user plot. The genre score is taken as the average of the actor genre scores, given that they match the user genres. This is done because one actor might have a high score in war and action movies, but low in romance and drama. If the user is making a romance and drama movie, then the genre score is taken as the average of just those

two scores. (write about past acting relationship if we decide to use it). To find how similar the plots are the python library gensim is used to compare. The plot of every movie the actor has been in is compared to the user plot. The maximum similarity score is used for the actor score. The similarity from gensim is the cosine similarity in the range (-1, 1). To make the average of the genre score and the similarity equally important for the actor score the cosine similarity bound is changed to (0, 10). Finally the actor score is changed to be in the interval (0, 1). Thus the expression for the score is: $actor(id) = \frac{1}{20}(avg(genre_score)+5\cdot(max(similarity)+1))$.

To find the other actors the procedure above is repeated on the actors matching the other descriptions. Because we consider it beneficial to cast actors who have worked together before, the actor scores of the secondary actors are increased by X if they have worked with the primary actor previously. To calculate the cast score we also make a prediction of the IMDB rating that the resulting movie will have if the selected actors are in it. The prediction is calculated from the average of the actors average genre scores. The expression for the cast score is: $cast(ids) = \frac{1}{\#ids}\sum_{i=1}^{\#ids} worked(ids_i, ids_1)\cdot actor(ids_i) + rating(ids)$. The *worked* function returns X if they have worked together before, 0 otherwise. The score of the primary actor is not increased by X.

The cast score is used to rank the groups. A higher score means a better groups. The groups are sorted in descending order and returned to the user.

6 EXPERIMENTAL EVALUATION

Here you evaluate your work using experiments. You start again with a very short summary of the section. The typical structure follows.

6.1 Experimental setup

Specify the context and setup of your experiments. This includes e.g. what hardware (VMs) you are running on, what operating system these machines are running, how they are connected, ...

Also explain how you generate load for your system and what parameters you used here. The general idea is to include enough information for others to reproduce your experiments. To that end, you should provide a detailed set of instructions for repeating your experiments. These instructions should not be included in the report, but should be provided as part of the source code repository on GitHub, typically as the `README.md` file, or as Shell scripts or Ansible scripts.

If your experiments give strange or unexpected results, analyze, profile and debug your code. **Do not simply re-run experiments until they give the expected results.**

Finally, running experiments is very time consuming and you may need to go through multiple rounds of experiment, debugging and optimization. **Do not delay running experiments until the end of your project period.**

6.2 Results

The results of your experiments. Compare different variants of your design (e.g. with and without optimizations) or compare performance to other designs or systems. Plots should show the average over multiple runs (at least 10 as a rule of thumb), including error bars, percentiles or min/max values.

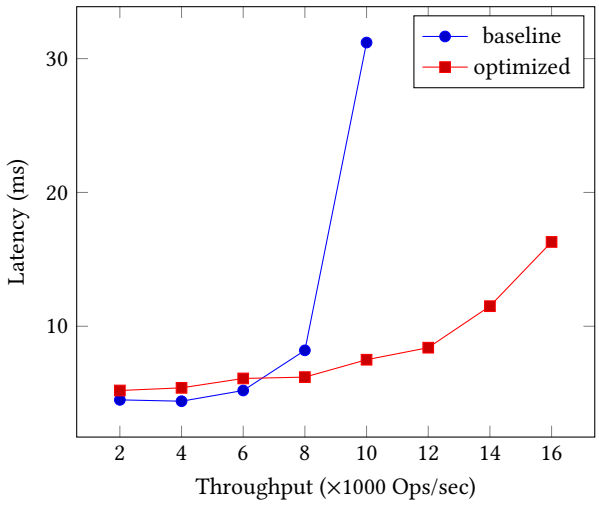

Figure 1: A graph showing latency and throughput of a baseline and optimized implementation. The axes show latency in milliseconds, and throughput in thousand operations per second. Data is made up.

Discuss the plot and extract the overall performance. Do not repeat all numbers in the text, but mention relevant differences in numbers, e.g. our optimization improves throughput by 26%. Discuss how the results validate or contradict your assumptions.

Perform experiments to evaluate your system under operating normal conditions, when experiencing failures or attacks, or with different workloads.

Figure 1 shows a graph generated with `pgfplots` from experiment data.

7 CONCLUSION

Here you need to summarize what you did and why this is important. Do not take the abstract and put it in the past tense. Remember, now the reader has (hopefully) read the paper, so it is a very different situation from the abstract. Try to highlight important results and say the things you really want to get across such as high-level statements (e.g., we believe that .... is the right approach to .... Even though we only considered LAN, the .... technique should be applicable ....) You can also formulate next steps if you want. Be brief.

REFERENCES

[] Inc Cinelytic. 2020. Cinelytic | Built for a Better Film Business. https://www.cinelytic.com/
[] Stephen Follows. 2018. How important are quality and cast for dramas? https://stephenfollows.com/how-important-are-quality-and-cast-for-drama-movies/
[] Apache Software Foundation. 2020. Hardware Provisioning. https://spark.apache.org/docs/3.0.0-preview/hardware-provisioning.html
[] Apache Software Foundation. 2020. Running Spark on YARN. https://spark.apache.org/docs/3.0.0-preview/running-on-yarn.html
[] Brian Fritz. 2020. OMDb API - The Open Movie Database. http://www.omdbapi.com/
[] Google. 2020. Universal Sentence Encoder. https://tfhub.dev/google/universal-sentence-encoder/4

[] Jae Huang. 2017. IMDB Data - Machine Learning (predicting movie gross). https://medium.com/@jae.huang111/imdb-data-machine-learning-predicting-movie-gross-2113513513bb
[] Inc IMDb.com. 2020. How do you calculate the IMDb rating displayed on a title page? https://help.imdb.com/article/imdb/track-movies-tv/ratings-faq/G67Y87TFYYP6TWAV#calculate
[] Inc IMDb.com. 2020. IMDb Datasets. https://www.imdb.com/interfaces/
[] Inc IMDb.com. 2020. IMDb: Ratings Reviews and Where to Watch the Best Movies TV Shows. https://www.imdb.com/
[] Max Woolf. 2018. Analyzing IMDb Data The Intended Way, with R and ggplot2. https://minimaxir.com/2018/07/imdb-data-analysis/