Search engine for generating a movie cast

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

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

ABSTRACT

A system to recommend a movie cast would require reviewing millions of movie titles and thousands of actors along with many other features making it a computationally challenging task. In this project, we have developed a large scale search engine to generate a movie cast using Spark and MRjob. In this report, we have shown how we used the data from IMDB[10] to generate a movie cast based on user input. To determine the recommended cast, we calculate a score based on genres, actor to actor relationships, and a brief movie summary. The user is presented with a ranked list of actors and the predicted IMDB user rating scores.

KEYWORDS

Hadoop, Spark, Film, Casting, Search engine

1 INTRODUCTION

Context and Motivation. Films or movies are not just an important art form but also a booming industry in various regions, all over the world. The economic impact of the film industry has been exponentially growing over time. The global film industry was worth an estimated $136 billion in 2018. Of all the various factors contributing to the success of a movie, the popularity/traction of the cast tops the list. The industry has seen many movies with amazing plot, but poor cast perform bad at the box office, while those with a popular cast outperform at the box office. This poses a serious threat to any film maker/ director. Choosing the dream cast for their plot has now become increasingly difficult. 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[2]. 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[10] 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. Hence, we developed a search engine which will address the above issues efficiently.

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 their lead actors to possess (like age, gender). The system uses data from IMDb to rank possible actors and returns a list of ranked actor(s) suggestions to the user. The rank is based on score which is a combination of past performance on related genre, actor to actor relationships, plot similarity and a subset of the actor characteristics.

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

Contribution Summary. There have been attempts to generate machine-learning based systems to predict movie success and box office collections. Use of recommendation systems for inspiration to choose actors to cast is also a possible option. But we have directly addressed the issue of generating a movie cast, given the particular user inputs by calculating a genre score for the subset of actors based on the input characteristics and plot summary. We believe this to be an efficient search engine since it can take any number of actors' descriptions as input and return a ranked list of the same number of actors or group of actors. We also developed an actor-to-actor graph and an efficient way to traverse through it to obtain relevant data. We have shown how using Spark/MRjob has significantly improved the overall efficiency of this project.

- 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.

Structure of report. This report is organised as follows. Section 2 provides an overview of related works and softwares that already exist. Section 3 describes the background and gives a clear understanding of terminologies used in the report. Section 4 provides the setup of the Hadoop cluster used in this project. Section 5 explains in detail the methodologies we used to address the problem statement, which includes all the steps from data procurement to implementation of algorithm. Section 6 contains the experimental evaluations and their respective outcomes. Finally, the concluding comments and possible future research directions are presented.

2 RELATED WORK

The IMDB datasets have been used to predict the gross movie revenue[7]. 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[13].

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.[1] is able to analyse and produce forecasts of how well a movie will perform based on country, cast and distributor. It also provides useful insights to optimize their content development, financing, production, and distribution decisions. The system ranks actors based on the predicted economic increase they have for a given project. They have partnered with multiple data sources like Netflix, google, the movieDB, Youtube, etc to gather various features relating to a movie.

Our system is similar to cinelytic, but we are basing the score only on IMDB data and use a small 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, almost like a movie encyclopaedia. It contains almost every movie that has ever been made and lets you browse through details like movie release year, plot summary, actors and actor details, genres, box office collections and much more. Users are prompted 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[8] to the votes to prevent manipulation, which means the rating is more like a weighted average. The box office collection value of a movie is the number of theatre tickets sold times the cost of a ticket. Some websites 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. Genre is the theme/type of the movie or the category into which the film is classified into, like drama or comedy. Plot summary is a brief storyline of 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[3]. 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 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[4] which means that the value we choose and the overhead had to be lower than the 3GB limit. To be sure that there was enough memory we decided on a spark.executor.memory value of 1512MB.

5 METHODOLOGY

5.1 Data collection

Acquiring the required data for this project in itself was quite challenging. A part the data can be downloaded from the IMDb website[9]. 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 an actor, they are assigned the label "0", representing male. Actresses are assigned the label "1", representing female. 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 to wescrape all the movie plots, because they stop responding to requests after a particular limit 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 was not 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[5] to retrieve the movie plots. Using this API, we were able to get all 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, 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 the movies in our dataset.

One notable challenge was that the data from IMDb is updated every day with new information. This means that you might not get the same result from the search engine as we did when you perform a search. We downloaded the data several times but at one point we noticed that the data can be corrupted, i.e. a lot of information was missing. There seems to be some "luck" involved in the download process. Some days none of the actors have birth years which means that the search engine will not work, other days the ratings were missing. To get the results in this report we used the version of data from the 6th of March 2020. The data from OMDb was only downloaded once on the 12th of March 2020.

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 a unique 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 a mandatorily 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 separate 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.

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 combination of three scores. We call the first one the genre score. It is calculated based on the actors previous performance in that particular genre. The second score is the similarity score. It is calculated from the plot summary 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.

The MRJob consists of one mapper and two reducers. The mapper reads each line and yields the data such that when reducing all of the information for a movie is available on the same line. An example of what is returned if the input line belongs to ratings.tsv is showed in code listing 1. In the first reducer the format is changed from being all of the information about a movie to yielding the information relative to the actor. If for example there are three actors in a movie, then there will be three yield calls with the genres and the rating of the movie. This is shown in code listing 2. The final reducer operates on the actor information and calculates the weighted average of the scores.

```
def mapper_collect_title(self, _, line):
    values = line.split("\t")
    if len(values) == 3:
        yield values[0], ["rating", (values[1], values[2])]
```

Listing 1: Genre score mapper

```
def reducer_title(self, key, values):
    # Generation of the genres object omitted.
    genre_ratings = {genre: rating for genre in genres}
    for value in values:
        if value[0] == "actor":
            yield value[1], genre_ratings
```

Listing 2: Genre score first reducer

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 is 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 [6] 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, shown in code listing 3. This means that all of the data had to be transferred 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 former method. 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 be 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.

```
scores = [None] * movies.count()
sim_arr = [search_plot, None]

# Iterate over every movie summary and calculate the
    similarity score
for i, row in enumerate(movies.rdd.toLocalIterator()):
    sim_arr[1] = row.summary
    sim = sim_model(sim_arr)
    scores[i] = (row.tconst, 10 * float(np.dot(sim[0],
        sim[1])))
```

Listing 3: Plot similarity

5.3.3 Relation score. The two previous scores are independent of 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 but also how well their on-screen combination worked. 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 were 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 ourselves. 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 you 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, see code listing 4. 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 exist 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 us 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 calculated in just one second the total time would be 45 minutes. When we implemented Dijkstra's algorithm on Spark, the runtime was even

```
queue = [start_node]
while len(queue) > 0:
    current = queue.get_lowest_cost_node()
    if current == goal:
        return score, path
    for edge in current.edges:
        edge.set_score()
        if edge in queue:
            queue.try_update_score(edge)
        else:
            queue.add(edge)
```

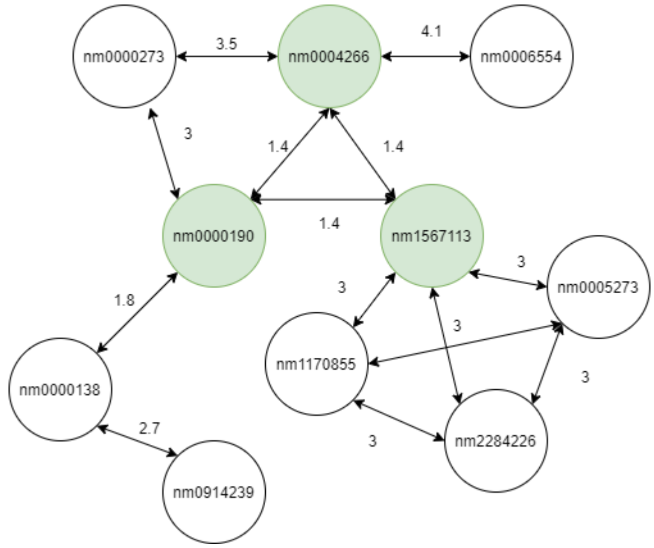
Listing 4: Dijkstra pseudocode

worse because each step requires that some data is transferred from a slave to the master and then back again. Because of the 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.

Algorithm	Time in seconds
Dijkstra	456
2-step Breadth first search (BFS)	13.54
2-step BFS Multi end	2.29
2-step BFS Multi start and end	0.0362

Table 1: Execution time of graph search algorithms

Our solution was to create a greedy fixed step algorithm inspired by the breadth first algorithm. The execution times of the different algorithm iterations we went trough are shown in table 1. 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 the challenge which is 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 were directly connected to Leonardo. For most relationships there is not a direct connection, 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 outward 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 therefore 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



For the Use case example (Interstellar): Matthew McConaughey: nm0000190, Anne Hathaway: nm0004266, Jessica Chastain: nm1567113 are the main actors. The edges represent the average of the inverse distance and the inverse rating.

Figure 1: A sample snapshot of the actor-actor relationship graph

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 or 3 step algorithm and the new cost function means that only the 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 or 3 steps, will not change the score.

Since 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 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 calculating the relation scores between the candidate lists A and B of size 30 each, the number of function calls is now reduced from 90 to just 30. A simplified version of this function is shown in code listing 5.

```
# Get initial nodes and their edges
current = graph.filter(F.col("node").isin(start_nodes))
current = current.join(graph, current.node == graph.node)
# Do n steps
for i in range(1, n):
    current = current.select("*", F.explode(current.edges))
    current = current.join(graph, current.last_node ==
            graph.node)
# Split by starting node and find best score
current = current.withColumn("score",
        calculate_score(current.columns))
current = current.select("start",
        F.explode(F.col("score")).alias("goal", "score"))
best_scores = current.groupby(["start",
        "goal"]).max("score").collect()
return best_scores
```

Listing 5: Simplified multi-start to multi-goal search algorithm

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 execution time is 4402 seconds, giving a time per relation of 1.63 seconds which is a 280 times improvement.

5.3.4 Overall score. The ranking algorithm is used to determine whether one actor could 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. 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. To make the average of the genre score and the similarity equally important for the actor score the similarity bound is changed to (0, 10).Then the actor score is changed to be in the interval (0, 1). Thus the expression for the score is: $actor(id) = \frac{1}{20}(avg(genrescore) + 5.(max(similarity) + 1))$.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, we use the relation score calculation above stated to calculate the actor relation score. 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 cast score is used to rank the groups. A higher score means a better group. The groups are sorted in descending order and returned to the user.

5.4 Workflow

The summary of the workflow for our project is pictorially represented in the two figures. The Python process is described in 2, and the Spark process is described in figure 3.

5.5 Algorithm results

The runtime of our algorithm is not the only evaluation we need to base our efficiency on. The result of a search is equally or more important. We first need to define what a good result is, and then how we decide whether or not a result is better than another. The obvious answer to what a good result is, is one that maximises the scores discussed in the previous sections. Following from that answer we also need to justify why those are good methods of measuring an actors' ability to perform well. We think it is intuitively true that if an actor has performed well earlier, they are more likely to perform well in their next movie of the same genre than someone who has a comparatively bad acting history. The same can be said about the similarity score, if someone has acted in a similar movie before then they are likely familiar with the setting and mindset they need to be in while acting. This does not however hold when arguing why the relationship score is relevant. Our reasonable justification for this is that we believe the assumption: "Any movie

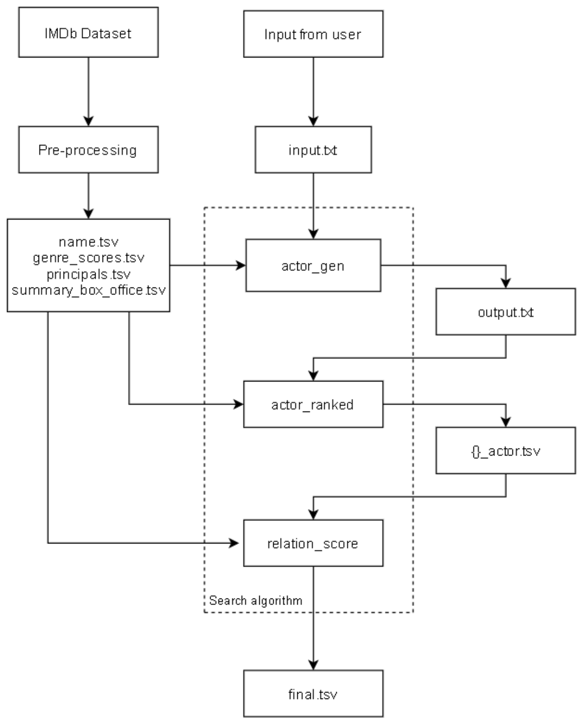


Figure 2: Python process workflow

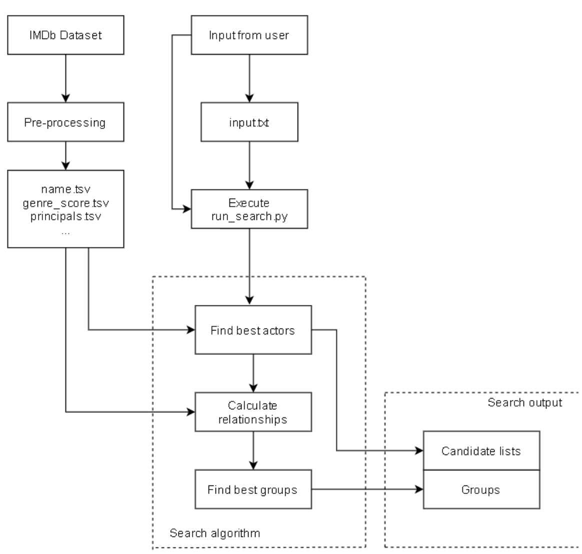


Figure 3: Spark process workflow

with a high user rating on IMDb has a good cast" to be true. From this we can say that if two actors have acted together in a movie with a high rating then they are good at acting together, and the opposite if the movie has a low rating then they should probably not act together. This assumption also means that we should have the following goal for our search engine: if you do a search with the attributes from a movie on IMDb, the original cast should be early in the result if the movie has a high rating. There is a fallacy to be aware of here. If you only try to achieve this goal, for example by weighing the similarity score and the relationship score by a large constant, then the individual is lost and the engine will most likely only return previous movie casts for any new query.

We believe we found a good balance between the importance of the different scores. A problem with the result we discovered while working on the engine is that when combining individual actors into groups then you need to have some measure of their relationship to avoid having the groups be a combination of just the individually best actors. What we mean by this is that if A and B actors have a sufficiently large score then they will be the suggested for several of the top groups, for example ABC, ABD, ABE, ABF and so on. This is of course the correct result based on the score, but it does not give a lot of information to the user because they are suggested a low number of actors relative to the number of groups. E.g. five groups make it possible to have 15 distinct actors, but the actual result might only contain 7 actors if the two first candidates are constant. This problem was our main motivation behind adding the relationship score. While it did help alleviate the problem, we still feel like this is a major problem with our search engine. To improve the result we decided to include the top ranked candidates from each list together with the top groups such that if the user feels like some actors are repeated too often then they can look at the individual lists for other alternatives.

6 EXPERIMENTAL EVALUATION

In this section, we are going to look at the difference in runtime between the implementation in Python and the Spark implementation. We are also going to show the results of some search queries.

6.1 Experimental setup

The MRJob/Spark implementation code ran on the Hadoop cluster described in the Hadoop cluster section. The Python implementation ran on a machine running Windows 10 Pro with the following hardware: Intel i5-8600K 3.6GHz CPU, 16 GB DDR4 2667MHz memory, and a SSD disk with a read speed of 500MB/s and a write speed of 320MB/s. The Python version used is 3.7.4 64-bit. To measure the execution time of the pre-processing scripts we used the time(1)[11] linux command. To be able to use linux commands on a Windows machine, we used the Git Bash terminal[12]. In the pre-processing steps each task is implemented in MRJob except the graph task which was implemented in Spark. The algorithm steps are implemented using Spark. The time unit is seconds unless otherwise specified. To measure the execution time, we used the time function from the time library in Python. We were not able to use the time(1) command because we wanted to measure several functions defined in the same script

6.2 Results

6.2.1 Pre-processing. As shown in the table 2 there is a significant difference between the implementations. The Python implementation is about 3.5 times faster than the implementation running on Hadoop. There are three main reasons for why the difference is so large.

The first and most important one is that there is an overhead when you use Hadoop. The time to connect and actually start the job is typically between 45 and 60 seconds. This means that since six tasks are executed the overhead could be as much as 360 seconds which is almost as much as the total time of the Python implementation.

The second reason for why the Python implementation is faster is because all of the data is able to fit in memory at once, while the Hadoop implementation requires data to be stored on multiple nodes and transferred between them.

The third reason is because the machine running the Python code is faster, it has more memory and a faster processor. Since the pre-processing is only supposed to happen once, we decided to not spend a lot of time optimizing the code so that we could focus on the algorithm instead.

Task	Python	MRJob / Spark
name_basics	25.5	102.3
title_basics	17.5	89.6
title_principals	32.9	664.3
title_ratings	2.0	67.0
graph	106.6	205.3
genre_score	195.2	232.3
Total	**379.7**	**1360.8**

Table 2: Pre-processing runtime of Python and Spark in seconds

6.2.2 Algorithm. The search query execution times were measured while using the attributes from the Interstellar movie. The total Python time is an estimate based on the relation score measurement. The relation score times are reported as a per relation score to better reflect the difference. The reason for why the difference is so great is because the Spark implementation is able to compute multiple relation scores at the same time, while the Python implementation can only compute one at a time. A more detailed explanation of the difference can be found in the Relation score section in part 5. In the execution time of the algorithm we can again see that when transferring data between nodes is required, the time increases a lot. This is why the Python implementation is faster in the candidate actors' task and the similarity score task. As we discussed in the similarity score section, we were not able to figure out a good method of sharing the tensorflow model between the worker nodes on Spark which is why the time is significantly greater in this task. The relation score time was calculated while running a 2-step search. When the number of steps is increased to three the run time per relation is increased to 0.4 seconds.

Task	Python	Spark
candidate actors	60.3	331.5
similarity score	89.6	448.8
relation score	438.0	0.0362
Total	**13.7 days**	**1182.1**

Table 3: Algorithm runtime of Python and Spark in seconds

The search query execution times were measured while using the attributes from the Interstellar movie. The total Python time is an estimate based on the relation score measurement. The relation score times are reported as a per relation score to better reflect the difference. The reason for why the difference is so great is because the Spark implementation is able to compute multiple relation scores at the same time, while the Python implementation can only compute one. A more detailed explanation of the difference can be found in the Relation score section in part 5. In the execution time of the algorithm we can again see that when transferring data between nodes is required the time increases a lot. This is why the Python implementation is faster in the candidate actors task and the similarity score task. As we discussed in the similarity score section we were not able to figure out a good method of sharing the tensorflow model between the worker nodes on Spark which is why the time is significantly greater in this task. The relation score time was calculated while running a 2-step search. When the number of steps is increased to three the run time per relation is increased to 0.4 seconds.

6.2.3 Search result. The search engine is not limited to five groups for a search query. It will return groups using the top 25 actors from each candidate list, and also return the complete candidate list for each query. A more complete search result is shown in Section 8, Appendix A - Search result. The following tables are the top five groups from two search queries. It is also possible to have queries where the number of actors is not three.

Actor 1	Actor 2	Actor 3	Score
nm0000190	nm0004266	nm1567113	9.145267
nm0000190	nm0544718	nm1567113	8.54046
nm0000354	nm0004266	nm1567113	8.534471
nm0000190	nm0004266	nm1325419	8.496985
nm0000190	nm0004266	nm0000234	8.451193

Table 4: Interstellar attributes with a 3-step relation score search

The original actors from the Interstellar movie are Matthew McConaughey: nm0000190, Anne Hathaway: nm0004266, Jessica Chastain: nm1567113. As the result show we achieved our goal of getting the original cast highly ranked by the engine.

However, it also highlights the largest problem we have been struggling to solve, which is that groups often contain the same actors. In this result the original actor from the second candidate list is the recommendation for eight of the top ten groups. The first candidate is repeated less, only four times in the top ten groups.

Actor 1	Actor 2	Actor 3	Score
nm2633535	nm1126657	nm1107001	6.8459167
nm2633535	nm1126657	nm0941777	6.8427186
nm3836977	nm2056274	nm2356421	6.8229513
nm2633535	nm1126657	nm1212722	6.8178062
nm2633535	nm1126657	nm1310016	6.790106

Table 5: 1917 attributes with a 2-step relation score search

The only original actor present in the result is George MacKay: nm1126657. Our initial thought was that the other actors must not be very good which turned out not to be true because both of them have acted in titles like Game of Thrones and Star Wars. The reason is that their birth year is missing from the data. This means that the engine is not able to consider them as candidates because their age cannot be determined.

7 CONCLUSION

We believe the search engine presented in this report can be a useful tool for movie directors who are looking for casting inspiration. Even though the top ranked groups contain a lot of actor repetition, we believe that the groups in addition to the ranked candidate lists can be significantly useful. The use of Spark and MRjob to handle such large amounts of data and processing has proven to be a useful application. We feel that we were able to show the strengths and weaknesses of using Spark by showing the significance of the overhead and how you can optimize an algorithm by utilizing the operations defined on a Spark DataFrame.

Future works could be to find a measure that does not repeat actors as much, or include additional actor features in the search query, like skin color or spoken languages. One could also develop region-based systems for say Hollywood, Bollywood, etc. which would help provide a better insight.

REFERENCES

[1] Inc Cinelytic. 2020. Cinelytic | Built for a Better Film Business. https://www.cinelytic.com/

[2] Stephen Follows. 2018. How important are quality and cast for dramas? https://stephenfollows.com/how-important-are-quality-and-cast-for-drama-movies/

[3] Apache Software Foundation. 2020. Hardware Provisioning. https://spark.apache.org/docs/3.0.0-preview/hardware-provisioning.html

[4] Apache Software Foundation. 2020. Running Spark on YARN. https://spark.apache.org/docs/3.0.0-preview/running-on-yarn.html

[5] Brian Fritz. 2020. OMDb API - The Open Movie Database. http://www.omdbapi.com/

[6] Google. 2020. Universal Sentence Encoder. https://tfhub.dev/google/universal-sentence-encoder/4

[7] Jae Huang. 2017. IMDB Data - Machine Learning (predicting movie gross). https://medium.com/@jae.huang111/imdb-data-machine-learning-predicting-movie-gross-2113513513bb

[8] 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

[9] Inc IMDb.com. 2020. IMDb Datasets. https://www.imdb.com/interfaces/

[10] Inc IMDb.com. 2020. IMDb: Ratings Reviews and Where to Watch the Best Movies and TV Shows. https://www.imdb.com/

[11] Michael Kerrisk. 2020. time(1) - Linux manual page. http://man7.org/linux/man-pages/man1/time.1.html

[12] Johannes Schindelin and contributors. 2020. Git for Windows. https://gitforwindows.org/

[13] Max Woolf. 2018. Analyzing IMDb Data The Intended Way, with R and ggplot2. https://minimaxir.com/2018/07/imdb-data-analysis/

8 APPENDIX A - SEARCH RESULTS

The top ranked entities from each file of the Interstellar search.

Actor 1	Actor 2	Actor 3	Score
nm0000190	nm0004266	nm1567113	9.145267
nm0000190	nm0544718	nm1567113	8.54046
nm0000354	nm0004266	nm1567113	8.534471
nm0000190	nm0004266	nm1325419	8.496985
nm0000190	nm0004266	nm0000234	8.451193
nm0000138	nm0004266	nm1567113	8.4200945
nm0436778	nm0004266	nm1567113	8.402761
nm0124930	nm0004266	nm1567113	8.354022
nm0749263	nm0004266	nm1567113	8.337727
nm0000190	nm0180411	nm1567113	8.323141
nm0001602	nm0004266	nm1567113	8.321685
nm0000630	nm0004266	nm1567113	8.321585
nm1663205	nm0004266	nm1567113	8.319469
nm0413168	nm0004266	nm1567113	8.319171
nm0000190	nm0680983	nm1567113	8.316085
nm0009716	nm0004266	nm1567113	8.308516
nm0001618	nm0004266	nm1567113	8.30351
nm0000190	nm0272581	nm1567113	8.302987
nm0670408	nm0004266	nm1567113	8.298609
nm0000190	nm0004266	nm0757855	8.295568
nm0005377	nm0004266	nm1567113	8.2876
nm0000255	nm0004266	nm1567113	8.279599
nm0000190	nm0103797	nm1567113	8.276432
nm0000190	nm2368789	nm1567113	8.272125
nm0790688	nm0004266	nm1567113	8.272066
nm0000190	nm0424060	nm1567113	8.264262
nm0001082	nm0004266	nm1567113	8.255302
nm0531602	nm0004266	nm1567113	8.255118
nm0000190	nm2225369	nm1567113	8.252507
nm0000190	nm0004266	nm0000701	8.248112
nm0000190	nm2057859	nm1567113	8.244077
nm0000190	nm0004266	nm0010736	8.242351
nm0000375	nm0004266	nm1567113	8.241773
nm0000190	nm1297015	nm1567113	8.240495
nm0719637	nm0004266	nm1567113	8.236309
nm0000190	nm0000204	nm1567113	8.235472
nm0000190	nm0004266	nm0487884	8.232474
nm0000190	nm0004266	nm0636426	8.229024
nm0000190	nm0004266	nm0000213	8.225317
nm0101710	nm0004266	nm1567113	8.224039
nm0000190	nm0004266	nm0000124	8.221664

Table 6: Groups - Total 2700 groups

Actor	Score
nm0000190	9.049950122833252
nm0000354	6.7763671875
nm0436778	6.718069911003113
nm0124930	6.564349174499512
nm0190613	6.301462888717651
nm1663205	6.283859431743622
nm0670408	6.2649906873703
nm0000630	6.255883157253265
nm0000138	6.236672580242157
nm0531602	6.223446309566498
nm0757327	6.208463907241821
nm0001082	6.2076098918914795
nm0101710	6.087757587432861
nm0000255	6.054446578025818
nm0749263	6.041021287441254
nm0009716	6.0345669984817505
nm0000226	5.939751863479614
nm0719637	5.922141969203949
nm0005377	5.915450751781464
nm0020354	5.897231578826904
nm0001602	5.893832206726074
nm0000881	5.863463044166565
nm0001618	5.862046122550964
nm0790688	5.84554660320282
nm0413168	5.8450204730033875
nm0881631	5.833773970603943
nm0000375	5.821726679801941
nm0000998	5.801511287689209
nm0005561	5.750837206840515
nm0293509	5.747084081172943
nm0586568	5.745926558971405
nm0151419	5.74288672208786
nm0252961	5.712387979030609
nm0000179	5.71123081445694
nm0000288	5.698324650526047
nm0922335	5.695802807807922

Table 7: Actors in candidate list 1 - Total 504 actors

Actor	Score
nm0004266	9.06173324584961
nm0544718	6.815100431442261
nm0180411	6.46216607093811
nm0272581	6.433978080749512
nm2057859	6.293877363204956
nm0103797	6.289476096630096
nm2225369	6.165146470069885
nm2573928	6.126913785934448
nm2368789	6.103429675102234
nm1297015	6.065259516239166
nm0452860	6.001994490623474
nm0000204	5.967656970024109
nm2239702	5.938843250274658
nm0461136	5.921574085950851
nm1429380	5.8644490242004395
nm1800338	5.828929603099823
nm0304801	5.819633185863495
nm0680983	5.7804147601127625
nm2095800	5.766095459461212
nm0471036	5.72269082069397
nm0004754	5.676045477390289
nm0755267	5.67033189535141
nm0488953	5.669280827045441
nm0945349	5.656118392944336
nm0424060	5.622214674949646
nm1066974	5.57268351316452

Table 8: Actors in candidate list 2 - Total 206 actors

Actor	Score
nm1567113	9.092016696929932
nm1325419	6.8247504234313965
nm0000234	6.506186246871948
nm0757855	6.293374180793762
nm0487884	6.217782735824585
nm1385871	6.092527389526367
nm0636426	5.981423169374466
nm0000239	5.963446497917175
nm2239702	5.938843250274658
nm0000612	5.909444570541382
nm0628601	5.859347224235535
nm0000701	5.854821145534515
nm0000213	5.85224175453186
nm0593664	5.753634691238403
nm0001303	5.742932319641113
nm0000124	5.698307454586029
nm0010736	5.678892076015472
nm0005392	5.672181874513626
nm0755267	5.67033189535141
nm0000106	5.664946645498276
nm0945349	5.656118392944336
nm0446935	5.6499714851379395
nm0006969	5.6108222007751465
nm1735332	5.56104975938797
nm0004851	5.537906765937805
nm1546686	5.535746097564697
nm0000586	5.534887909889221
nm0221046	5.506050765514374

Table 9: Actors in candidate list 3 - Total 255 actors