Spotlight, Search the Stars

Movie Actor Search Engine

Michael Schniepp
University of Amsterdam
12160067
michael.schniepp@student.uva.nl

Tom Verburg
University of Amsterdam
10769633
tom.verburg@student.uva.nl

Luca Verhees
University of Amsterdam
10837833
luca.verhees@student.uva.nl

1 INTRODUCTION

Due to the increasing amount of information on the web, the number of search engines has drastically increased. This trend has made it possible to find information one is looking for more easily [1]. Furthermore, the offer of topic-specific search engines is very extensive. For the entertainment industry, a number of websites provide databases with movie information, reviews, and actor profiles. Examples of such search engines are IMDb[1], RottenTomatoes[2] and MetaCritic[3]. However, the majority of these systems focus on a movie-centred approach, whereas searching from an actor perspective is less common.

Therefore, this paper presents *Spotlight*, an information retrieval system designed to provide unconventional actor searching. With this system, a user can search for quotes, names (the actual name of an actor or role), a movie title or genres through a user-friendly interface. The system will then provide relevant actor profiles offering details, filmography and related news. Furthermore, the system will show other actors and actresses which are similar to the actor searched.

2 ARCHITECTURE

The application consists of two primary components: the back-end, where the pre-processing, indexing and ranking happens, and the front-end in which the user interacts with the system. A brief description of both components, as well as their relation will be described in this section.

The back-end of the system is built using Flask [2]. Flask is a Python framework, which is very suitable for a simple application where the front-end and backend are tightly coupled, which is the case in the current application. Flask allows one to construct web applications and run custom back-end code on just Python alone. Using this framework, relatively simple code was written that provided a linkage between an interface, our data, the semantic model and our index.

Both the scraping and indexing of the data happens independently of the user interactions with the system. The scraping of the data is performed using the Scrapy framework, see section 3. This framework was chosen due to its ease of use, high performance and built-in functionality. The indexing of the data is performed using NLTK and several standard Python functions, see section 4. These packages were chosen based on their ease-of-use, efficiency and support within the information retrieval field. After indexing, relevant news articles and movies were matched to each actor, and both the indexes as well as the data itself were stored in JSON and Pickle files for easy access during user interaction. The data was split up into three components: actor data, news data, and movie data. By leveraging the indexes generated, documents related to a given actor can be selected and returned in the results page or an actor profile.

The front-end of the system is built using HTML, CSS and JavaScript on top of the Flask framework. The BM25, as well as the cosine similarity scores, are computed on the spot based on the processed query input of the user or the selected actor, see section 5. Flask templates were utilised to interact with the ranking functions and acquire dynamic information per page for the results. The web interface allows users to enter a query into a search bar and subsequently search and return results, see section 6 for more details of the interface.

3 DATA ACQUISITION

To supply our system with data the web was crawled for a variety of information related to popular actors. The Scrapy[4] framework was used to crawl and scrape the data needed. The gathered data ranges from movie databases to movie reviews and celebrity news sites.

3.1 Data Sources

The main goal of the presented system is to provide profiles for actors and actresses. The basic details about actors and actresses, such as biography and filmography, are retrieved from IMDb and RottenTomatoes. On the one hand, IMDb is a movie and actor online database, storing information about thousands of movies along with actor profiles. IMDb has a structured celebrity browser for all 5+ million actors in their database. The actors are listed in sequential order based on IMDb's "Star Rating", which orders stars based on their current and general popularity [3]. The first 10,000 actors were crawled for the designed application. The name, biography, filmography, quotes and profile photo of each actor were stored. Furthermore, this list of actors was used as input for the other sources to make sure that the retrieved information would overlap. On the other hand, RottenTomatoes is a website which is focused on movie ratings and also supplies other information concerning celebrity profiles. These celebrity profiles were scraped to retrieve additional information regarding actors and actresses, including quotes, biographies, birthday and birthplace.

An actor's filmography retrieved by IMDb was extended with additional information about the movies from IMDb, RottenTomatoes and MetaCritic. The year, genres and movie poster were retrieved from IMDb. Furthermore, the genre of the movies, as well as the reviews were retrieved from MetaCritic. Moreover, additional movie reviews were retrieved from RottenTomatoes.

[1] www.imdb.com
[2] www.rottentomatoes.com
[3] www.metacritic.com
[4] www.scrapy.org

Furthermore, to supply the system with related news, three news websites were crawled: HollywoodLife [5], TMZ [6] and MovieWeb [7]. For each website, the title, content and publication date of over 10,000 most recent news articles were stored.

3.2 Web Crawling

As mentioned before, the web scraping framework Scrapy was used to crawl information from the web. Scrapy allows for the construction of a crawler (spider) that will create HTTP requests to specified websites [4]. The data from the returned response is extracted and stored. Scrapy also makes it possible to extract and follow links to scrape numerous pages of our choice.

3.2.1 General Structure. Five custom web spiders were operated: one for the retrieval of actor profiles, one for the retrieval of movie information and one for each news source. Custom Scrapy items were used to directly parse the information retrieved and store it in a structured manner. For each item, the information was stored as a record in a JSON file. For each spider, a JSON file was stored, containing the retrieved items as records.

All spiders have a general structure, starting with declaring their name, the URLs to initialise the search, as well as the allowed domains. The spider starts from the declared starting URL's and immediately parses the retrieved pages to extract the relevant information. In order to retrieve relevant information as well as pagination links from the retrieved page, X-path selectors were identified using Chrome Developer Tools[8]. Once the page was parsed, and the relevant information was retrieved, the spider follows the link to the next page. This can either be the next page in a list of links with pagination, such as the news articles, or a link retrieved or generated from the current page, such as the corresponding reviews of a movie. This process continues until the spider has visited every page and retrieved all relevant information.

3.2.2 Politeness. To make sure that the various crawlers implemented were not harming the web-page, a variety of politeness settings were implemented using built-in Scrapy functionality [5]. For each crawler, the same politeness settings were used. First, the crawler identifies itself on the user-agent, by defining the *USER_AGENT* setting. Second, the crawler obeys the robots exclusion protocol provided by the website by setting the *ROBOTSTXT_OBEY* to True. Third, the crawler keeps a low bandwidth usage by maximising the number of concurrent (i.e. simultaneous) requests and setting a download delay. The maximum number of concurrent requests was set to sixteen. The amount of time that the downloader should wait before downloading consecutive pages from the same website to three seconds. Scrapy automatically introduces a random delay ranging from `0.5 * DOWNLOAD_DELAY` to `1.5 * DOWNLOAD_DELAY` seconds between consecutive requests to the same domain.

3.2.3 Distributed Crawling. Rather than performing the crawling on a single machine, a distributed setting was employed. The six data sources were split among three computers to improve the effectiveness of our crawling. This approach ensured that the task of making thousands of requests, downloading, and storing the individual websites was not restricted by a single machine's capabilities. In this way, distributed crawling was implemented. Each machine employed custom crawlers adapted to their respective websites. In order to make sure no duplicate information was stored, the visited URLs were clearly separated between the spiders and pagination and "load more" requests were utilised. Additionally, the built-in Scrapy setting that prevents the crawler from visiting the same URL twice in a single crawling session was applied.

3.2.4 Refreshing the Repository. A script was put together to automate the execution of all spiders in order to refresh the repository once desired. Moreover, in order to prevent duplicate pages from being scraped, additional information of each crawled web page was stored, such as the time stamp of crawling, the operating spider name and the original URL. This information could then be used to check if pages had to be crawled again. Furthermore, in order to most efficiently update the news articles, the URL and date of the most recent article were stored for each source. Then when crawling again, only the articles more recent than the previous most recent article had to be stored.

3.3 Crawling Results

Table 1 shows the total number of pages retrieved during the crawling process. The counts are split up per domain. As can be seen in the table, the total number of pages retrieved is 115,339.

Table 1: Number of pages downloaded for each domain.

Source	Page Count
www.imdb.com (actors)	10,000
www.imdb.com (movies)	44,239
www.rottentomatoes.com	14,002
www.metacritic.com	10,269
www.tmz.com	9,100
www.movieweb.com	11,980
www.hollywoodlife.com	10,484
www.brainyquote.com	5,265
Total	**115,339**

4 INDEXING

After the relevant data was crawled and stored in JSON files, the data was processed and indexed. An inverted index was constructed to reduce the data to the informative terms contained in them and provide a mapping from the terms to the respective documents containing them.

4.1 Document pre-processing

Prior to indexing the documents, the data was pre-processed using the Python packages Pandas[9] and Natural Language Toolkit (NLTK)[10], a Python package for processing natural language. The

[5] www.hollywoodlife.com
[6] www.tmz.com
[7] www.movieweb.com
[8] https://developers.google.com/web/tools/chrome-devtools/
[9] pandas.pydata.org/
[10] www.nltk.org

pre-processing consisted of three steps: tokenization, stop-word removal and stemming.

Firstly, the JSON file was converted to a Pandas DataFrame for easy manipulation. Secondly, all textual data elements were tokenised using the word_tokenize() function for splitting strings into tokens based on white space and punctuation. Furthermore, all the remaining white space characters were removed. This step resulted in one list of tokens for each document. Thirdly, the stop words were removed from the list of tokens using the predefined list of English stopwords provided by NLTK. Lastly, to normalise the tokens, stemming was applied using the Porter stemmer. The list of tokens was stored as a new column in the DataFrame for easy access during the indexing process.

4.2 Indexing Architecture

Documents are generally stored as lists of words, but inverted indexes invert this by storing for each word the list of documents that the word appears in. This index can be stored in one inverted index using Python's nested *defaultdict*. Given these insights, the following procedure was executed to index the data.

First, the actor and movie data was matched, based on the filmography of each actor. By doing this, each actor contained relevant movie information. Second, separate frequency indexes were created for each of the actors, as well as for the news sources. A one-pass index with merging approach was applied. For the actors, the names were used as unique IDs, and for the news articles, the URLs were used as unique IDs. As a next step, the news indexes were merged, which resulted in the two final indexes: one for actors and one for news articles. Finally, the two indexes were stored as separate pickle files for later use.

4.3 Distributed Indexing

The distributed indexing was performed in a similar manner as the distributed crawling. A separate inverted index was created for each source to improve the effectiveness of indexing the data. The creation of these indexes was split among three computers to share the computational load, and the indexes were then stored as pickle files for easy accessibility. These indexes were then merged by a single machine using a mapping function to produce the final inverted indexes.

4.4 Updating the Index

As described in section 2.2.4, when refreshing the repository new data is processed, which means that the inverted indexes need to be updated. When the ID of a document is not found in the inverted index, it is added to the initial index using the append function of Python's defaultdict. However, if the ID is found in the index, the complete document is first deleted and then added again. This is done to ensure that no outdated trails of information are stored in the index.

5 SEARCHING AND RANKING

For a successful information retrieval system, a mechanism that could process queries and return relevant results was required next. This section described the approaches to implement searching and ranking in the presented system. Next to basic term-based search, a method for semantic-based search is described.

5.1 Processing query

After having indexed the documents related to specific actors, it is possible to set up a basic search. However, before any ranking can be performed, it is necessary to process the query first. The query was processed the same way as the documents for the creation of the index. This means that the same methods were applied as described in the indexing section 4: the pre-processing is done through the Python packages Pandas [6] and NLTK [7] where the word_tokenize() function is used to split the query into separate strings, and the white space characters removed. Furthermore, the same stop-words are removed, and stemming was implemented based on NLTK and porter stemmer.

5.2 Spell Checker

In order to help improve the overall functionality of the retrieval system, a basic spellcheck was implemented. The spellcheck employed is based on the implementation created by Peter Norvig [8]. This is a very rudimentary spellcheck that can help in simple typos, but the technology behind high-performance spelling correction is quite advanced. There are four main sections to the spellcheck as outlined below.

The spellcheck takes a single word and generates possible variants of the word by performing four actions: *deletions*, deletes one letter in the string for each position; *transposes*, switches the position of adjacent letters for each position; *replacements*, replaces each position with each letter of the alphabet; *inserts*, each letter of the alphabet is inserted in each position of the string. Naturally, there could be more candidates generated by taking all words that are two edits away from the original. To capture this we take the set of words that are one edit away and perform the algorithm again to generate an even larger set of variations.

Of course, this results in a very large set of nonsensical words. To capture only known real words the intersection of this set and a dictionary set is taken. The result is a set of words that are one or two edits away from the original word and also exist in the dictionary.

To determine the probability of a word occurring the number of times the word occurs in a large text corpus is counted and divided by the total number of words. The corpus that is used, consists of a concatenation of public domain book excerpts from Project Gutenberg and lists of most frequent words from Wiktionary and the British National Corpus. Our IMDb biographies and actor names were also concatenated to the corpus.

Consequently, of all possible correct spelling alternatives, we take the word with the highest probability of occurring. In making the final selection, a few assumptions are made. Firstly, if the word checked exists as is in the dictionary the original word is returned. Next, the assumption is that words that are a single edit away are more likely than ones that are two edits away and thus are prioritized in selection. Finally, words that are unknown to the dictionary are simply returned as is with no correction made.

The UI provides a suggestion for the alternate corrected query string to the user, see Appendix A figure 2. This behaviour resembles

Google's *"Did you mean...?"* feature. The user could then opt for the corrected variant if desired. An alternative implementation could be to correct each token before proceeding with the query processing as described earlier.

5.3 Term-based search

For the basic searching functionality of the system, the BM25 ranking function was implemented. BM25 is a state-of-the-art TF-IDF-like retrieval function used in document retrieval [9]. The main advantage of this model is its efficiency. Although BM25 implements inverse document frequency (IDF), it mainly revolves around adding weights for the term frequency (TF) and the length of the document. BM25 does not take into consideration the inter-relationship of the terms in the query in regards to the document, just like the regular TF-IDF score. The complete formula can be observed in equation 1.

$$\text{score}(D, Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{\text{avgdl}})} \quad (1)$$

Two tuning parameters need to be set. The other variables are the inverse document frequency ($IDF(q_i)$), TF ($f(q_i, D)$), document length ($|D|$) and average document length ($avgd$) and are calculated based on the corpus, query term i and selected document D.

Firstly, $k1$ is a parameter for calibrating the document term frequency scaling. This variable limits the amount of influence a single term in the query can have on the score. For the current implementation of the BM25 we upheld the same value as set by Elastic Search, which is 1.2 [10].

Secondly, there is the tuning parameter b. This parameter makes sure that the length of the document is also taken into account when computing the score. For a larger b, the effects of the length of the document is more amplified compared to a smaller b. For the current ranking, the score is the same as is implemented as the standard score for Elastic search, which is 0.75 [10].

Upon search, the BM25 score is calculated for every word in the query against all the documents. Following this, the scores are summed for all the documents in the corpus for the terms contained within the query. Based on these BM25 scores, the top 10 scoring documents are returned to the user. The user can search for a variety of different subjects, like the name of an actor, role, genre, movie name or director, and the system will retrieve actors related to the query based on all the textual data which was crawled as described in section 3.2.

5.4 Additional functionality

5.4.1 Ranking prioritization. The term-based search function as described in 5.3 treats all the information in a document equally. However, it could be argued that some parts of documents are more important than others. For example, in the present search system, it is likely that a user searches for (part of) an actor name and expects this actor to be returned. Therefore, it was decided to give exact matches of actor names priority over the regular term-based ranking method.

When a user searches for *Tom Cruise*, the BM25 score of all actors matching this name is increased by 50. By doing this, the actors with a name match will be on top of the ranking list. However, by also taking into account the BM25 score (on top of the initial score of 50), the actors are still ranked by their relevance to the query. To illustrate this, when a user searches for *Tom action*, all actors named *Tom* will appear on top of the ranking in the correct ranking regarding their relevancy to the term *action*.

5.4.2 Search filters. In addition to the basic term-based search method, an exclusion filter has been implemented. This filter gives the user more control over the search query. The exclusion filter is an additional input field, next to the regular search field, in which the user can add query terms which he would like to exclude from the results explicitly. Once the system identifies an input in the exclusion field, both the regular query as well as the filter are pre-processed and taken into account when ranking documents using BM25. If a document contains one of the terms in the exclusion filter, the score of the document is set to 0, regardless of the score based on the regular query.

For example, when a user wants to search for the query *Harry Potter*, but they do not want to see any results which include *Daniel*, the user can simply add this as a constraint to the search action. Consequently, the results will not contain *Daniel Radcliffe*, the main character of Harry Potter, as well as other actors referencing his name. Appendix A figure 4 shows an overview of user interaction concerning this exclusion filter.

5.4.3 News matching. As an additional feature, the application also matches the crawled news articles to individual actors. Initially, various methods such as TF-IDF were implemented to see whether actors could be linked to individual news articles. However, initial results indicated that this was not a very accurate method and mostly resulted in poor results not relating to the actor in question at all. For this reason, it was opted to do a literal search of the actor's full name over the news articles and return the most recent number of links to these articles. In the case there were no matches, no links to external news articles were provided.

5.4.4 Semantic-based search. In addition to a term-based method for searching, as described for 5.3, a semantic-based method was implemented. In the application, cosine similarity is implemented to provide the user with similar actors on a selected actor page. The reason this was done with the use of the semantic search was that it seemed interesting to suggest relevant based on more than just term-based ranking methods such as BM25.

For the implementation of cosine similarity, a vector was constructed based on all the tokenised content of all the pages for all the separate documents. This is done through the use of the doc2vec package from Gensim [11]. This package converts the documents into a model consisting of separate vectors of a set size through the use of a shallow neural network and implementing both a continuous bag of words model (CBOW), as well as a skip-gram model. By training this model, a vector of a set length per document is returned. This enables the option of performing a cosine similarity on the different documents from a semantic perspective, seeing as the word order is kept when constructing the vectors.

The vector of the selected actor (the query vector) is created and the cosine similarity is computed between the query vector and all the vectors in the model. Based on the cosine similarity

[11] https://pypi.org/project/gensim/

scores, the first four most similar actors are returned. The formula for computing the cosine similarity can be observed in equation 2.

$$cos(x, y) = \frac{x \cdot y}{||x|| \cdot ||y||} \tag{2}$$

6 INTERFACE

The user interface of the application has a Google-like interface, which presents the user a search box to enter their query. The interface is built on top of the Flask framework using HTML, CSS and JavaScript. Furthermore, common design standards have been applied to ensure the usability of the application. The Bootstrap design framework has been used in the design of the interface to ensure the usability of the system. Moreover, common best practices for data visualisations and user interactions have been implemented, as described by Nielsen **??**. Screenshots of all important screens can be found in Appendix A.

The main screen of the application allows the user to enter a query and an optional exclusion filter, see Appendix A figure 1. When the user submits a query, a set of actors is retrieved based on the BM25 ranking algorithm (section 5.3). The results show a brief overview of each actor with their corresponding picture, a snippet of their bio and a link to their aggregated document. Furthermore, if the spellchecker found an alternative query, this spelling suggestion is presented right beneath the search bar, see Appendix A figure 2.

After clicking on a result, the user goes to the document of that individual actor. This page shows an overview of the aggregated information about the actor, including a photo, biography, filmography, related news and related actors, see Appendix A figure 3. From this detailed page, the user can easily navigate to the external links such as the profiles on IMDb or RottenTomatoes, the news articles or the movie pages on IMDb. Moreover, the user can easily navigate back to the search results using the back button to search again or select another result.

7 EVALUATION

In order to evaluate our retrieval system performance, several offline metrics were used as described below. Three people were asked to label ten documents for twenty different queries as either relevant or non-relevant. Relevancy was left to the user to decide, although they were encouraged to examine the actor profile for information such as biography and filmography to gauge relevance to the query. From this data, several offline metrics were calculated. These metrics demonstrate the ability of our system to retrieve relevant documents for a given query.

7.1 Precision

A typical measure of effective information retrieval is precision which can be defined as the number of relevant documents retrieved divided by the total number of retrieved documents. Spotlight retrieves a maximum of ten documents and thus we define *precision at k* documents in Equation 3. Additionally, *average precision* was also computed, which offers a measure of accuracy while also taking the result's ranking into consideration. Average precision is defined in Equation 4.

$$Precision@k = \frac{\#(relevant\ items\ at\ k)}{k} \tag{3}$$

$$AveragePrecision = \frac{\sum_{k=1}^{n} P(k) \times rel(k)}{numberOfRelevantDocuments} \tag{4}$$

Both metrics were computed for each of the twenty tested queries and for each of the relevancy judgments given by the raters. For each query, the average was taken across each of the three rater's relevancy judgments. The full results on a query level are shown in Table 4 in the appendix. To generalise the results, we can see that a score of 0.81 and 0.65 were achieved for the system Mean Average Precision and Precision@10, respectively. These results would indicate that for the given set of queries, on average six of ten results were relevant to the query, while many of the relevant results were towards the top, as indicated by the reasonably high mean average precision.

7.2 Mean Reciprocal Rank

Mean Reciprocal Rank (MRR) is used to assess how high a system puts the first relevant document upon being queried. The query-specific MRR is listed in Table 4. For each of the three relevancy judgments, reciprocal rank was computed then the average was taken of the three to yield MRR. Fourteen out of twenty queries gave an MRR of 1, indicating that most queries gave a relevant result in the first position. Moreover, the Average MRR across all queries was 0.87, indicating that our system reliably returned a relevant result in the top or near-top positions. This is a critical characteristic of a good retrieval system as users tend to look directly at the first few results when searching for information.

7.3 Discounted Cumulative Gain

Discounted Cumulative Gain (DCG) is another metric to measure the effectiveness of search results. This metric behaves very similarly to Average Precision as it takes the position of results into consideration. The formula is shown in Equation 5. DCG was computed for each of the twenty queries assessed and averaged across each assessor. Table 4 shows that perfect results yield DCG of 4.54. The system average comes to 3.13, which is approximately 70% of the maximum, signalling acceptable performance similar to what Mean Average Precision reports. Although, MAP may be a better metric to evaluate as only a binary relevance system was used where DCG was designed to use a more variable scale.

$$DCG = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{log(i + 1)} \tag{5}$$

7.4 Inter-Rater Agreement

Inter-rater agreement was also evaluated using Cohen's Kappa Coefficient. This metric (Equation 6) is computed over the total results of all queries to determine overall agreeableness of the raters. As can be seen in Table 2, the achieved Kappa Coefficient came out to be 0.45, which would indicate moderate agreement across raters. An explanation for this score could be that Cohen's

Kappa takes into consideration the probability that the raters agree by chance. This assumption may not be entirely relevant within the context of this experiment as our computed average agreement across each search result indicates otherwise. It is also reasonable to note that the nature of relatedness of actors to a given query is somewhat subjective and thus some disagreement is expected. General agreement across all items returned was 75% meaning the raters agreed on document relevancy on an average of 75% of items. This result would indicate to us that the raters overall mostly agreed and the results obtained are reliable judgments.

Table 2: Queries and Evaluation Results

Kappa Coefficient	Avg Agreement
0.45	0.75

8 CONCLUSION

The final result of the Spotlight system consists of an end-to-end pipeline starting with data collection, cleaning, indexing and storage. Finally, a clean, user-friendly interface was implemented to permit users to search through our collection of data. Some additional features were implemented to enhance the overall quality of the system, such as search-term exclusion, spell checking, supplying relevant news, and suggesting related actors to explore. Live evaluation of the system by three users was performed and indicates that the system is functional and indeed succeeds in its task to retrieve information relating to actors upon relevant queries.

9 DISCUSSION

In this study, an information retrieval system for unconventional actor search has been proposed. The system has proven to work as expected. However, several limitations and directions for future work can be identified, which will be discussed in this section.

9.1 Limitations

The user has the option to search for a name, movie, quote and much more and in all cases, the system retrieves a set of 10 actors. This static number of results makes it more difficult to measure the accuracy and overall quality of the system since only the first result could be relevant. For the offline evaluation, it has proven to be complicated to label a result as relevant or non-relevant. For example, when searching for a specific actor name, nine other actors were also returned, who were in some way related to the query. This relevance of specific actors is very subjective to the intent of the user and the content of the query. The judgement of the assessor is dependent on whether the user is familiar with the respective actors. For someone who searches for *Frodo* and has not seen the *the Lord of the Rings*, the actor *Ian McKellen* (an actor who has also starred in *the Lord of the Rings*) may seem a random non-relevant document. This makes it difficult in measuring metrics such as precision and seems to have influenced the average agreement more for the more complicated queries.

Another issue which came forward during the development of the system is that there were a variety of individual IMDb actor pages which were not retrieved using polite crawling. Therefore, there is a set of famous actors which are not available in our dataset and therefore cannot be retrieved. These include actors such as *Johnny Depp* and *Leonardo DiCaprio* who star in a variety of popular movies. This limitation resulted in particular searches which were not able to retrieve these specific actors and would have influenced our evaluation (seeing as if the documents corresponding to these actors were not in our index). Therefore, the queries used for evaluation were determined beforehand.

Furthermore, it was decided not to use a search engine, such as Elastic Search, to keep full control over the search system and to increase our learning curve regarding information retrieval systems. However, building a system from scratch means that not everything is implemented as smoothly and efficiently as a refined system such as Elastic Search has. Features such as pointers for the index and tuning of the hyper-parameters of the neural network used to create the doc2vec model are all elements of the system which could be tweaked and possibly result in improvements of the system.

9.2 Future work

There are several features which could be implemented to improve the overall use and experience of this system. One of these features would be the ability of the user to set their own filters when searching. This addition would make the system more accurate and retrieve more relevant results, seeing as the retrieval system would then only query over the relevant data. An example of this would be when searching for a quote. The query would only look at the quotes crawled instead of looking at the document as a whole.

A second feature which could improve the overall accuracy and experience of the system would be to prioritise certain features. As it stands now, only the actor names are given a higher priority in the ranking. This means that every query is checked against a set of names of all the actors in our dataset. If (part of) a name corresponds with the query, the document corresponding to this query is given priority over other documents. The reason for this is because users mainly use the system to retrieve specific actors, and will, therefore, be inclined to enter the specific name of an actor they are looking for.

To further refine the effectiveness of the system a more robust index could be created to handle bi-grams and tri-grams and even specific phrases, such as movie names or person's names. Moreover, an ensemble approach to computing a custom ranking based on system-specific features could enhance our results and ranking. Ideally, many more factors would be considered upon the query such as entity type (name, genre, etc.) and all factors such as an actors filmography, common genres, and other factors could be evaluated to create a more sophisticated ranking.

10 REFLECTION

As discussed in Section 9.1, the presented system was built from scratch as opposed to using a search engine. Building the system from scratch indeed highlighted the major challenges in implementing a working retrieval system. Firstly, much thought was put into how all the collected data could be organised and presented in a logical and useful way. Organisation and presentation of information showed to be an interesting challenge especially when a wide variety of data had to be collected and linked together. Next,

indexing also proved critical in developing an effective system as thought must be given in order to match a query to relevant documents effectively. It became clear that careful consideration must be made to ensure an effective and efficient index. Lastly, the project demonstrated the considerable complexity in which documents can be ranked and evaluated for relevancy when returning search results. It was made clear how many factors can be and should be considered when selecting relevant documents. The Spotlight project resulted in a functioning retrieval system with satisfactory results and definite room for improvement. More refinement for methods in which relevant documents are selected could be made to produce a more sophisticated system.

The division of work between team members can be found in Table.

Table 3: Division of tasks.

Task	Owner(s)
Crawl TMZ and Metacritic	Luca
Crawl IMDB	Michael
Crawl RottenTomatoes	Tom
Crawl Metacritic and Movieweb	Luca
Write Introduction	Luca & Michael
Write Data sources	Tom
Write Web crawling	Luca
Distributed recrawling	Luca
Create inverted index	Luca
Implement spell check for query	Michael
Build first TF-IDF search	Tom
Create BM25 search	Luca
Write BM25 text	Tom
Create intial TF-IDF cosine similarity	Tom
Build Interface	Luca & Michael
Write about Interface	Luca & Michael
Implement doc2vec for semantic search	Tom
Set up evaluation form	Michael
Evaluation metrics and summarisation	Michael
Discussion	Tom

REFERENCES

[1] Detlor B. Turnbull D. Choo, C. W. Information seeking on the web–an integrated model of browsing and searching. 1999.
[2] Flask. http://flask.pocoo.org/, 2019.
[3] Imdb: Star rating as determined by users. https://help.imdb.com/article/imdbpro/industry-research/faq-for-starmeter-moviemeter-and-companymeter/GSPB7HDNPKVT5VHC, 2019.
[4] Scrapy: Http request documentation. http://doc.scrapy.org/en/latest/topics/request-response.html, 2019.
[5] Scrapy: Politeness settings. https://blog.scrapinghub.com/2016/08/25/how-to-crawl-the-web-politely-with-scrapy, 2019.
[6] Pandas. https://pandas.pydata.org/, 2019.
[7] Natural language toolkit (nltk). https://www.nltk.org/, 2019.
[8] Peter norvig's spelling correcter. http://norvig.com/spell-correct.html, 2016.
[9] Zaragoza H. Robertson, S. The probabilistic relevance framework: Bm25 and beyond. *Foundations and TrendsÂő in Information Retrieval*, 3(4):333–389, 2009.
[10] Elastic. Similarity module. https://www.elastic.co/guide/en/elasticsearch/reference/current/index-modules-similarity.html, 2019.

APPENDICES

Appendix A. Screenshots Interface

Click here for a link to the screencast of the system.

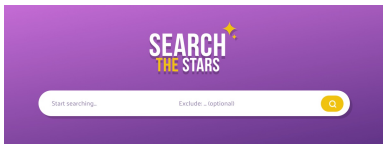

Figure 1: Start page containing search bar

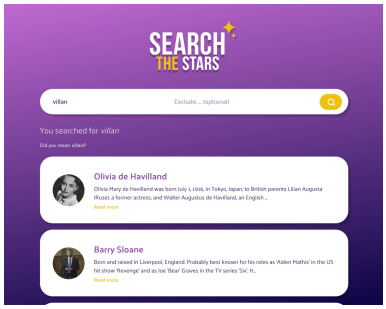

Figure 2: Search results with spell check suggestion

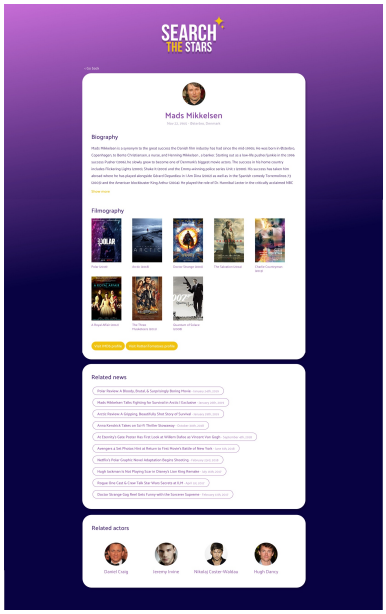

Figure 3: Difference between regular query and query including exclusion

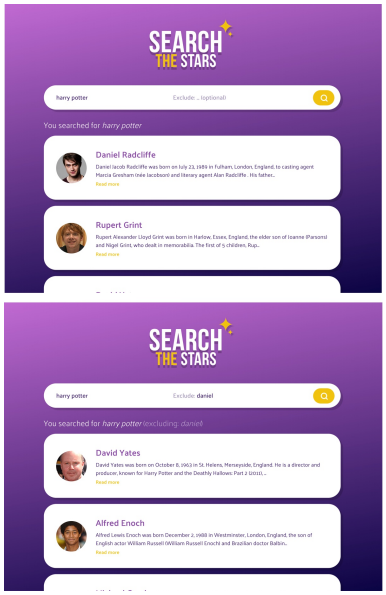

Figure 4: Exclusion filter on search query

Table 4: Queries and Evaluation Results

Query	Type	Avg Agreement	Avg Precision	Precision@10	MRR	DCG
Action	Genre	1.00	1.00	1.00	1.00	4.54
Comedy	Genre	1.00	1.00	1.00	1.00	4.54
Horror	Genre	0.87	0.96	0.87	1.00	4.11
Brad Pitt	Name	0.53	0.78	0.57	1.00	2.86
Bruce Lee	Name	0.67	0.71	0.37	1.00	2.06
Tom	Name	1.00	0.86	0.90	1.00	3.91
Netherlands	Location	0.93	0.79	0.77	1.00	3.49
China	Location	0.87	0.69	0.70	0.50	2.81
Italy	Location	1.00	1.00	1.00	1.00	4.54
Frodo	Role	0.67	0.91	0.63	1.00	3.33
Villain	Role	0.73	0.88	0.87	0.83	3.79
Voldemort	Role	0.60	0.91	0.67	1.00	3.41
Harry Potter	Movie	0.67	0.93	0.70	1.00	3.56
Star Wars	Movie	0.67	0.93	0.60	1.00	3.22
Pulp Fiction	Movie	0.60	0.65	0.43	0.67	2.12
Luke I'm your father	Quote	0.73	0.76	0.37	1.00	2.21
One ring to rule them all	Quote	0.60	0.44	0.27	0.50	1.4
Smell of napalm in the morning	Quote	0.60	0.58	0.30	0.66	1.79
Luke science fiction	Combined	0.67	0.53	0.33	0.55	1.62
English comedy	Combined	0.53	0.84	0.67	1.00	3.32
Column Average		**0.75**	**0.81**	**0.65**	**0.87**	**3.13**

Appendix B. Evaluation

$$\kappa = \frac{P(A) - P(E)}{1 - P(E)} \tag{6}$$

$$P(A) = \frac{\#agreed}{\#total} \tag{7}$$

$$P(E) = P(rel) + P(nonrel) \tag{8}$$

$$P(rel) = \frac{\#a1relevant}{\#total} \times \frac{\#a2relevant}{\#total} \tag{9}$$

$$P(nonrel) = \frac{\#a1non-relevant}{\#total} \times \frac{\#a2non-relevant}{\#total} \tag{10}$$