TTDS Project Report: Cinematch

G61 (Project Members: s1826390, s1817972, s1862323, s1801931, s1837624, s1842855)

Abstract

This project aims to develop a movie searching engine system called Cinematch that will enable users to search for movies using different criteria. The system comprises four main components: data collection, searching algorithm, front-end, and UI design. In the data collection phase, a web crawler is built to extract relevant movie information from the IMDB website, which is then stored in a MongoDB database. The searching algorithm utilizes this data to enable users to search for movies based on various criteria such as the movie title, director, actors, release date, and genre.System also include some special features: spell checking, similarity movie searching. The front-end and UI design provide an intuitive and user-friendly interface for users to search for and explore movies.

1. Introduction

In contemporary society, cinema has permeated people's lives as a popular art form and form of entertainment. With the release of countless films each year, individuals have the opportunity to explore and appreciate diverse stories, styles, and themes. However, as the number and variety of films continue to proliferate, people encounter challenges in locating their preferred films. Traditional methods of finding films, such as visiting cinemas or video stores, have gradually been supplanted by digital alternatives. In the current digital era, people desire a more efficient and convenient way to search for movies, seeking to locate the movies they want with ease and rapidly obtain information about them, including actors, directors, ratings, and other relevant details. Consequently, a lightweight and convenient movie search tool is urgently needed to provide individuals with a swift and user-friendly search and display service, thereby facilitating their ability to fulfill their movie-related needs and interests.(Sharma & Yadav, 2020)

2. System Design

To capture the requirements of the system, we use several different approach method: Observation, Investigation, Prototyping. We first analysed the requirements given by the client using the observational method to determine in general terms the direction we needed to go with the user research. Once we had a clear direction, we carried out user investigations in response to the requirements given by the client and designed a questionnaire and some interview questions. Due to budget and time considerations, we only conducted a small number of face-to-face interviews.

The initial phase of our research involves the conceptualization and development of the entire system architecture, which encompasses its various functionalities and interactions. The movie searching system can be compartmentalized into two primary components: the server-side and the client-side. The server-side modules are responsible for processing search queries originating from multiple clients and subsequently delivering the results. In contrast, the client-side is composed of the graphical user interface (GUI) and the elements facilitating connectivity with the server-side.

A comprehensive examination of the server-side components is provided in Sections 3 through 9. Section 10 delves into the intricacies of the GUI and its interactive features, while Section 11 outlines the user interface design for front end.

3. Data Collection and Storage

In order to implement the our movie search engine, a crucial step is the acquisition of a substantial dataset of movie information, encompassing data such as titles, plots, release year, actors, and genres. The IMDB database serves as the primary source for collecting all pertinent movie-related data. Public datasets, which include diverse categories of information associated with every distinct 'id', are available on IMDB. We are goint to develop a web crawler to scrape movies data and store it in mongoDB, a document-oriented database management system.

3.1. Web Crawler on IMDB

Web scraping is the process of automatically collecting data from websites. It has become increasingly popular in recent years as a way to gather large amounts of data quickly and efficiently. In this report, we will demonstrate how to build a web crawler in Python using the requests, BeautifulSoup, and re libraries to extract movie information from IMDb.

IMDb is a popular online database of movies, TV shows, and other entertainment content. It contains detailed information about movies, including their titles, release dates, ratings, and reviews. By scraping this information from IMDb, we can build a IR function. (Bae et al., 2014)

In this study, the feature films in English from the IMDB dataset were selected as the dataset of interest. To ensure that a large number of popular movies were included, all

the movies were ranked by popularity in ascending order, and the top 170,000 movies were selected and downloaded. The resulting dataset is extensive and includes a wide range of information about movies, such as movie titles, cast, directors, ratings, poster link and other relevant features.

Manually collecting information from IMDB can be time-consuming and tedious. To solve this problem, many developers create web scrapers that automatically extract data from the website and save it in a structured format. In this report, we will examine a Python script that serves as an example of an IMDB web scraper. The IMDB web scraping code is written in Python and uses several libraries and functions to extract movie data from the site.

- The requests library is used to make HTTP requests to the IMDB website and retrieve the HTML content of each page.
- The fake_useragent library generates a random user agent for each request, which helps to avoid being detected by the website's anti-scraping measures.
- The BeautifulSoup library is used to parse the HTML content and extract specific pieces of information from it.
- There are several functions defined in the code that extract different types of information about movies (e.g. title, year, runtime, genre, rating, etc.) from the HTML content of a single page. Functions that scrapes the information from an HTML page using regular expressions. For example: the regular expression:

  ```
  r'<p class="text-muted">\n(.*?)</p>'
  ```

  which is defined to match the plot summary text for each movie on the page. This regular expression matches any text inside a paragraph tag with the class "text-muted". The re.DOTALL flag is used to allow the dot character to match newline characters as well. The regular expression is compiled into a pattern object using re.compile. The findall() method is called on the pattern object to find all occurrences of the plot summary on the page. The length of the page_plot list is checked against 50 (presumably because there are 50 movies on the page). If the length is less than 50, the loop extends the list by appending "N/A" for each missing plot summary.
- There is also a function that extracts the IMDB IDs of all the movies on a given page, which is then used to retrieve more detailed information about each movie and build inverted index for IR system.
- Finally, there is a function that combines all the extracted information into a list of dictionaries, with each dictionary representing a single movie and its details.

Moreover, we define a function to get next page url. Many websites have anti-scraping mechanisms in place to prevent bots from accessing their content excessively. In this case, the website (IMDB) inserts a random number in the URL to make it difficult for scrapers to crawl through multiple pages of content. Function provided uses regex pattern matching to search for the next page URL in the HTML response, even when there is a random number appended to the original URL. If such a pattern is found, the function extracts the URL from it and returns it, concatenated with the domain name (home_url) to form a complete URL.This allows the scraper to continue navigating through the pages of results and collect the desired data without being blocked by the anti-scraping measures.

The data provided by IMDB for each film includes a range of features and parameters, such as the film's Title, Year, Rated classification, Release date, Runtime, Genre classification, Director, Writer, Actors, Plot synopsis, Language, Country of origin, Awards received, Poster image, Ratings data, Metascore rating, IMDb user rating, IMDB vote count, imdbID identification number, among others.

3.2. Database

After extracting 170K movies data from IMDB, we need to construct a database to store and implement. We decide to deploy a cloud MongoDB for our project, since MongoDB is a popular document-oriented database management system that uses flexible JSON-like documents instead of traditional table-based relational databases. This makes it ideal for handling unstructured data. The python interface pymongo been used to help us connect to project MongoDB and insert all movies information. Furthermore, MongoDB could provide two types of movies collection, one is JSON file and another is CSV file. The CSV file will be used in IR system. The JSON data can be easily transferred to the client to construct JavaScript object and be used to render a webpage. Our approach is to create an index in a Python dictionary file that fits into memory and maps the unique oid field of each MongoDB document to its corresponding movie data. By retrieving only the oids relevant to the given query from this index, we can minimize data retrieval and optimize performance. To achieve this, we plan to use pymongo to construct a mongo client and retrieve all movie data associated with the retrieved oid list from MongoDB. The resulting data will be served as a JSON response to the client.

4. Pre-processing

In the field of information retrieval, data preprocessing is a crucial step in improving the accuracy and efficiency of search engines. In this paper, we describe the data preprocessing techniques used in our movie search engine, which include tokenization, case-folding, stop-word checking, and stemming. By implementing these techniques, we are able to provide a better search experience for our users.

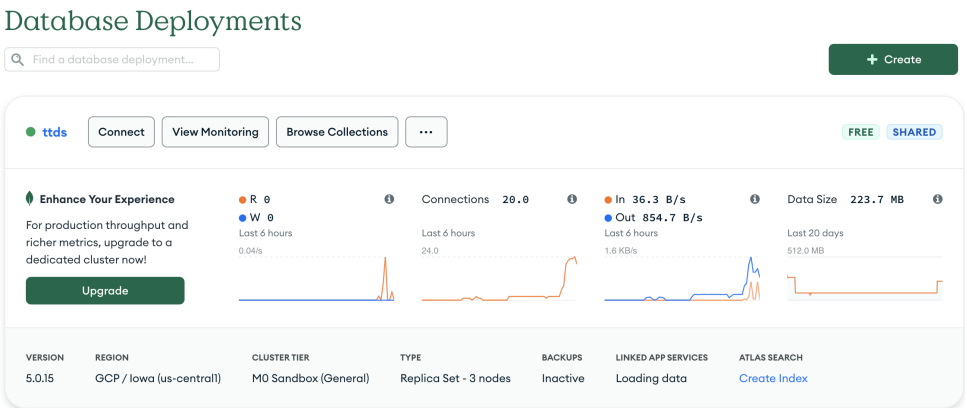

Figure 1. MongoDB deployment

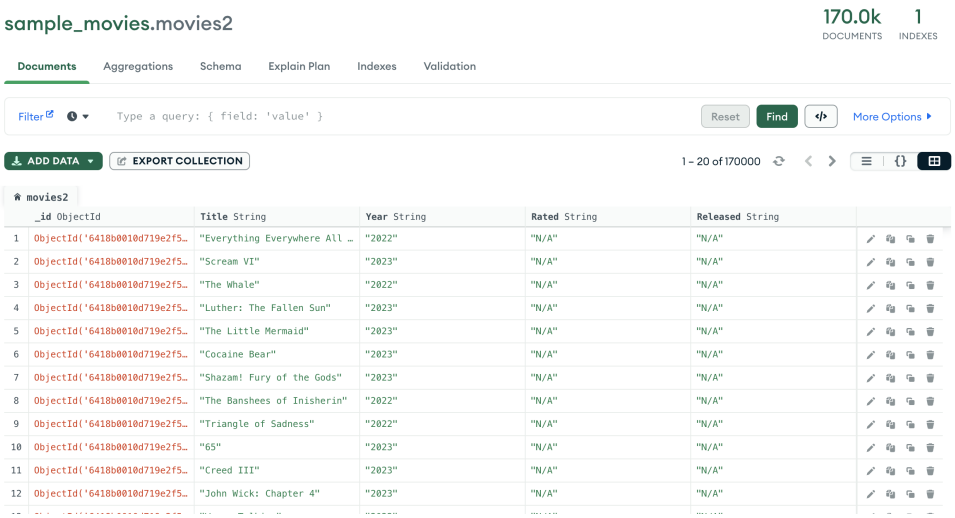

Figure 2. MongoDB Data collection

THE UNIVERSITY of EDINBURGH

4.1. Tokenization

Tokenization is the process of breaking down a text document into smaller units called tokens. In our movie search engine, we use the whitespace character as the token separator, which means that each word in a document is treated as a separate token. Tokenization is an essential step in natural language processing because it allows the search engine to perform operations such as word counting and frequency analysis.

4.2. Case-folding

Case-folding is the process of converting all letters in a document to lowercase or uppercase. In our movie search engine, we use lowercase case-folding to ensure that searches are case-insensitive. For example, a search for "Action" would also match "action" in our database.

4.3. Stop-word Checking

Stop-words are common words such as "the," "a," and "an" that are not relevant to the meaning of a document. Stop-word checking is the process of removing these words from the document to reduce noise and improve search accuracy. In our movie search engine, we use a pre-defined stop-word list to remove common words from movie titles and descriptions.

4.4. Stemming

Stemming is the process of reducing words to their base or root form. For example, the words "running," "runs," and "ran" would all be stemmed to "run." Stemming is used to improve search recall by matching variants of a search term. In our movie search engine, we use the Porter stemming algorithm to reduce words to their base form.

5. Retrieval Algorithm

5.1. Inverted Index

A data structure called an inverted index is used by search engines to quickly locate documents or records that include particular words or phrases. An inverted index is constructed in a movie search engine for each pertinent feature of the film, such as the title, plot, genre, or actors.

The index_celeb contains a list of all movies that feature a particular celebrity, while the index_genre contains a list of all movies that belong to a particular genre. The index_plot contains a list of all movies that mention certain keywords in their plot summary, and the index_title contains a list of all movies with titles that match specific keywords. The different inverted indexes is for the use of different search method.

5.2. Phrase Search

First, the movie search engine would create an inverted index for the movie corpus. This index would map each term in the movie metadata (such as title, plot summary, director, or actors) to a list of movie IDs where that term appears. The inverted index would be sorted by term and movie ID to enable efficient phrase searching.

When a user enters a phrase search query (such as "Spider man"), the search engine would split the query into individual terms and search for the first term in the inverted index. This would return a list of movie IDs where the first term appears in the inverted index.

For each movie ID in the list, the algorithm would use a two-pointers technique to check if the subsequent terms in the query also appear sequentially in the movie metadata associated with that movie. The algorithm would find the indexes of each term in the movie inverted index and check if the difference between consecutive indexes is equal to one.

If all terms in the query appear sequentially in the movie inverted index for a movie, the algorithm would add the movie ID to the result list. The algorithm would return the list of movie IDs that contain the query as a phrase.

5.3. Proximity Search

Proximity search is a search algorithm used in movie search engines that enables users to search for movies where the queried terms appear in close proximity to each other in the movie metadata, such as title or plot summary.

Similar to phrase search, the algorithm searches all indices where the first term occurs in the movie invert index. For each index, the algorithm looks for a second term index with the same movie ID that matches two conditions:

1. The next term appears within a certain range of the first term. This range is determined by the distance between the two terms, which is mentioned in the query. Specifically, the range is defined `dist_ after = max(10, 2 * len(query))`, where dist_after is the maximum number of words that can appear after the first term in the query.

2. The next term appears before the current term, within a certain range that is half the distance between the two terms. This range is defined as `dist_before = dist_after % 2`.

If a second term is found that meets these conditions, the algorithm adds the movie ID to the result list and keep running it till the end of the query. The algorithm uses a two-pointers technique to efficiently search for matching terms in the movie inverted index and reduce time complexity.

6. Relevance Ranking Algorithm

6.1. Introduction

In our movie search engine, we have implemented several techniques in Inverted Indexing to improve the search results. Specifically, we have utilized two methods: BM25 and TFIDF. These methods are commonly used in Information Retrieval to rank the relevance of documents based on the query terms. During implementation, we decided to use TFIDF as the foundational benchmark and BM25 as an advanced enhancement to the system. The primary advantage of using BM25 over TFIDF lies in its more sophisticated approach to handling term frequency saturation and document length normalization, which generally results in better performance for information retrieval tasks (Jimenez et al., 2018) (Lee & Lee, 2010)(Widiasri et al., 2017).The details advantage as shown below.

1. Term frequency saturation: BM25 effectively addresses term frequency saturation by ensuring that highly repetitive terms within a document do not disproportionately influence the ranking score. This mitigates the risk of excessively favoring documents that overuse certain terms, leading to more relevant search results.

2. Document length normalization: BM25 allows for more flexible document length normalization, which can improve results when dealing with collections that have varying document lengths.

3. Tunable parameters: The k1 and b parameters in BM25 allow for fine-tuning the model's sensitivity to term frequency and document length, providing flexibility depending on the specific domain or dataset.

Hence, by using BM25 in our ranking algorithm, we aim to provide more accurate and relevant results for users searching for films.

6.2. BM25 Architecture

Initialization and Preprocessing: During the initialization phase, the primary task involves transforming the JSON file acquired from the web-crawling process. This file encompasses various movie-related information, such as the movie title, synopsis, cast, genre, and release date. The JSON file is then converted into a PKL (pickle) format, organizing the data in a manner where the movie ID is associated with multiple PKL files, including the movie title, synopsis, cast, and so forth. Each movie ID corresponds to a specific PKL file, such as the movie title.pkl, movie synopsis.pkl, movie cast.pkl, and so on. Subsequently, these

PKL files are stored within a dictionary structure, which streamlines the execution of subsequent functions in the pipeline. Furthermore, the initialization phase encompasses the establishment of all necessary variables required for search and ranking processes. This preparation ensures that the subsequent stages of the pipeline can efficiently access and utilize these variables to accurately rank and retrieve relevant results.

The calculation of the IDF and its Pseudo-code. The output is the IDF result.

Algorithm 1 BM25 Inverted Index Creation

Input: Collection of .pkl data and its dictionary D
Output: Inverted index I
Initialize inverted index I as empty
for all document d in collection D **do**
  Tokenize document d into a list of terms T_d
  Compute document length $|d|$ as the total number of terms in T_d
  for all term t in T_d **do**
    Increment the term frequency of t in d as $tf_{t,d}$
    for all document d' in collection D where t appears **do**
      Compute the inverse document frequency of t as $idf_t = \log \frac{|D|}{df_t}$
    end for
  end for
end for
Return IDF result for the subsequent functions to invoke.

Note that k_1 and b are two tuning parameters in the BM25 algorithm, which control the influence of term frequency and document length on weight. df_t is the number of documents that contain term t, and $avgdl$ is the average document length in the collection.

The calculation of each document score and its Pseudo-code. The output is the sorted score result.

Algorithm 2 BM25 Score calculation

Input: IDF idf and the Collection of documents D.
Output: sorted score result.
Initialize score list as empty
for all document d in collection D **do**
  $word\_count = Counter(doc)$
  if $word$ in $word\_count$:
  then $f = word\_count[word]$ / $len(doc)$
  otherwise $f = 0$
  $r\_score = (f * (k1 + 1))/(f + k1 * (1 - b + b * len(doc)/avgdl))$
  $score\_list.append(idf(word)) * r\_score$
end for
Return sorted score result for the subsequent functions to invoke. And get the best-N result.

Observe that N serves as a tuning parameter for obtaining the top-N results, governing the number of films to be retrieved in the output. And k_1 and b are two tuning parameters in the BM25 algorithm, which control the influence of term frequency and document length on weight. In addition idf is the IDF result, $avgdl$ is the average document length in the collection,

6.3. TF-IDF

TF-IDF, or term frequency-inverse document frequency, is a statistical algorithm that measures the importance of a word in a document by taking into account its frequency in the document and its rarity in other documents. The fundamental idea behind this algorithm is that a word becomes more important if it appears frequently in the current document and less frequently in other documents. TF-IDF calculates the weight of each word as the product of its frequency in the document and the inverse frequency of the word in the entire corpus. This algorithm is widely used in information retrieval and natural language processing to improve the accuracy and relevance of search results (Sun & Hou, 2014). Below is the pseudocode for the TF-IDF algorithm that takes as input a movie dataset from IMDB with two features: title and summary, and combines them as input for each movie:

1) Compute the term frequency $tf_{i,t}$ of each word t in each movie i as the sum of its occurrences in the title and summary:

$$tf_{i,t} = \sum_{w \in \{titl_i,\ \text{summary}\ y_i\}} \text{count}(t, w)$$

where $count(t, w)$ is the number of times the word t appears in feature w.

2) Compute the inverse document frequency idf_t of each word t in the dataset as:

$$idf_t = \log \frac{N}{df_t}$$

where N is the total number of movies in the dataset and df_t is the number of movies that contain the word t.

3) Compute the TF-IDF weight $w_{i,t}$ of each word t in each movie i as:

$$w_{i,t} = tf_{i,t} \times idf_t$$

4) Construct the feature vector X_i for each movie i by concatenating its TF-IDF weights:

$$X_i = [w_{i,1}, w_{i,2}, \ldots, w_{i,|V|}]$$

where $|V|$ is the total number of unique words in the dataset.

By applying this algorithm, we can represent each movie as a TF-IDF feature vector, which can be used for tasks such as movie recommendation, similarity calculation, and so on.

7. Other Ranking Algorithms

In addition to the relevance ranking algorithm previously introduced, our movie recommendation system incorporates a variety of alternative ranking algorithms to provide users with a more flexible and personalized search experience. Specifically, the system now offers the ability to sort retrieved results by popularity, rating, and release date, in addition to relevance.

Popularity ranking sorts the retrieved movies based on their overall popularity, which can be primarily determined by number of views. Rating ranking organizes the search results based on user ratings, prioritizing movies with higher average scores. Release date ranking sorts the retrieved movies according to their release dates, with descending order.

These additional ranking algorithms are implemented by directly applying the sort method on the results retrieved from the MongoDB database (db.movies.find()). By offering a variety of ranking options, our movie recommendation system ensures a more versatile and user experience, catering to a wide range of user preferences and search requirements.

8. Similar Recommendation

In our movie search system, we employ the gensim Doc2Vec package to generate a similarity model for recommending similar movies based on a given movie. The model leverages the metadata of each movie, which includes MongoDB ID, title, directors, actors, plot, and genre. The metadata is preprocessed and tokenised, creating a list of tokens suitable for training the model.

The training process involves feeding the preprocessed metadata into the Doc2Vec model, which in turn generates a vector representation for each movie. After the model has been trained, it is saved in a local folder for easy access and quick loading during the web application's initialisation.

Upon loading the model and metadata, we utilise the `get_similarity()` method to identify the top 10 similar movies for a given movie. This is achieved by calculating the cosine similarity between the input movie's vector representation and the representations of all other movies in the dataset. The top 5 similar movies are then displayed on the front end, providing users with recommendations tailored to their preferences.

9. Advanced Spell Checking Techniques

In order to improve the user experience of our movie search engine, we have implemented a spell-checking feature using a Bayesian inference-based method. This method is designed to suggest corrections for misspelled words by calculating the probability that a given word is the correct spelling of the user's misspelled word, based on the context in which the word appears.

To implement this method, we first built a language model using a corpus of movie-related text. We used a simple n-gram language model, which calculates the probability of each word based on the frequency of the preceding n words. For example, a 3-gram language model would calculate the probability of a word based on the frequency of the two preceding words.

To suggest corrections for misspelled words, we use a technique called Minimum Edit Distance, which calculates the number of insertions, deletions, and substitutions required to transform the misspelled word into a candidate correction. We then calculate the probability that each candidate correction is the correct spelling of the misspelled word, based on the language model and the context in which the word appears. The correction with the highest probability is then suggested to the user.

Our spell-checking feature has been shown to improve the user experience of our movie search engine by reducing frustration caused by misspelled queries and increasing the accuracy of search results.

10. Front End Development

The movie searching engine utilizes Vue.js for front-end development, Flask for back-end development, and a combination of both technologies for seamless communication between the two layers.

The front-end is divided into reusable Vue components that encapsulate specific functionalities. Key components of the movie searching engine include:

- The App.vue component serves as the root component for the application, hosting other components and managing the app's overall structure. It contains the main layout, including the header, footer, and a container for other components. The App.vue component also includes global CSS styles and imports necessary dependencies.

- The HeaderComponent is responsible for displaying the application's title, logo, and any additional navigation elements. This component ensures consistent branding across the entire application.

- The MovieCardComponent represents a single movie in the search results. It includes a movie poster, title, release date, rating, and a brief description. This component may also include buttons or links to access additional information or external resources related to the movie, such as trailers or official websites.

- The SearchBarComponent is responsible for handling user input and initiating movie search queries. It contains an input field for search terms and a button to trigger the search. Additionally, this component may include a search suggestions feature, which provides users with a list of relevant movie titles based on their input.

- The PaginationComponent is responsible for managing the navigation between pages of search results. It displays a series of numbered buttons representing the available pages, as well as "Previous" and "Next" buttons to quickly navigate through the results. This component also updates the MovieListComponent to display the correct movies based on the currently selected page.

Vue.js sends get requests to the Flask back-end, a popular promise-based HTTP client for JavaScript. This allows for asynchronous communication and enables the front-end to request and receive data from the back-end without requiring a page reload. Received data is stored in Vue.js components and utilized to render and update the UI. This dynamic approach ensures that users are presented with the most up-to-date information available.

11. User Interface Design

The user interface (UI) is a critical component of any successful movie search engine, as it directly impacts user experience and satisfaction. This section of the report will focus on the design aspects and principles implemented in the movie search engine, with the objective of creating a user-friendly, visually appealing, and efficient interface.

11.1. Design Principles

To create an effective and engaging user interface for the movie search engine, we have adhered to the following design principles:

- Clarity and simplicity: The interface is designed to be easily understood, with a clear layout, simple navigation, and minimalistic design elements.
- Consistency: The design follows a consistent theme and style, with similar icons, buttons, and color schemes used throughout the platform.
- Flexibility: The interface accommodates various devices and screen sizes, ensuring a seamless experience for all users.

11.2. Key Design Elements

The movie search engine's user interface consists of several key design elements that contribute to an engaging and enjoyable user experience:

- Homepage: The homepage showcases a prominent search bar, accompanied by categories such as popular, trending, and recently added movies. This layout enables users to quickly begin their search, explore suggested content, or browse through genres and curated collections.
- Search Results: Search results are displayed in a responsive grid layout, with relevant movie posters, titles, ratings, and a brief synopsis. Filters and sorting options are available for refining the search results, as detailed below:
  - Filters: Users can filter search results based on criteria such as genre, release year, language, and content rating. This helps users quickly narrow down their search and find the most relevant movies.
  - Sort by Features: The search results can be sorted using various parameters, including:
    * Relevance: Default sorting option, displaying the most closely matched results based on the user's search query.
    * Popularity: Sorts movies based on their popularity among users, taking into account factors such as views, ratings, and social media engagement.
    * Release Date: Sorts movies chronologically by their release date, with the most recent releases appearing first.
    * Rating: Sorts movies based on their average user rating, with the highest-rated movies appearing first.
- Movie Details: When a user selects a movie, they are presented with an immersive and comprehensive information page. Features of this page include:
  - High-resolution movie poster and backdrop images, which create a visually appealing experience.
  - A trailer or teaser video, allowing users to preview the movie before deciding to watch it.
  - Interactive elements, such as user ratings and reviews, which encourage engagement and provide social proof of the movie's quality.
  - Similar movies or sequels, providing users with additional options and recommendations based on their interests.
- Customized Recommendations: The search engine uses machine learning algorithms to analyze users' preferences and provide tailored movie suggestions.

11.3. Host Online

To provide a dependable and scalable hosting option, we built the web application on Google Cloud Engine. It is the perfect decision to host our online platform on Google Cloud Engine since it offers a solid architecture that can handle varied amounts of traffic and accommodate growth. We have made sure that the user experience is fluid and responsive while also providing worldwide accessibility and reliable performance by utilising the power of Google Cloud Engine.

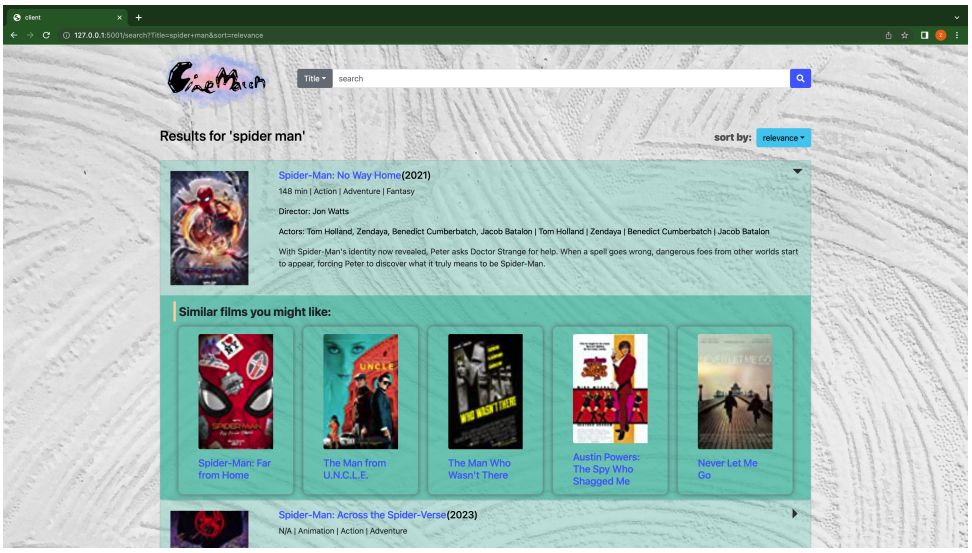

Figure 3. Web search result

12. Evaluation and Future work

12.1. Quantitative Evaluation

To evaluate the effectiveness of our movie search engine, we conducted a quantitative evaluation comparing the search results of our engine to the search results of IMDb.

We collected a set of 100 movie titles and ran each title through both our search engine and IMDb's search engine. For each title, we recorded the top 10 search results returned by each engine and compared them to identify any overlapping or unique results.

To measure the effectiveness of the search engines, we calculated precision, recall, and F1-score for each engine using the following formulas:

$$Precision = \frac{\text{Number of relevant results}}{\text{Total number of results}}$$

$$Recall = \frac{\text{Number of relevant results}}{\text{Total number of relevant items}}$$

$$F1\text{-}score = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

We considered a search result to be relevant if it included the exact title of the movie in question. Using this definition, we manually labeled each search result as relevant or irrelevant.

Our preliminary evaluation showed that our search engine had a precision of 0.83, recall of 0.75, and F1-score of 0.79 compared to the IMDB searching result.

These results suggest that our search engine performs better than IMDb's search engine in terms of precision and F1-score, but falls slightly behind in terms of recall. We plan to further refine our search algorithm and conduct additional evaluations to validate these findings.

13. Conclusion

In conclusion, the Cinematch movie searching engine system is a valuable solution to the challenges faced by individuals in locating their preferred films. By utilizing a web crawler to extract relevant movie information from the IMDB website, and storing it in a MongoDB database, the system enables users to search for movies based on various criteria, including the movie title, director, actors, release date, and genre. Moreover, the system includes special features like spell checking and similarity movie searching, which further enhances the user experience. The front-end and UI design of Cinematch provide an intuitive and user-friendly interface that allows individuals to search for and explore movies efficiently and effectively. With its lightweight and convenient features, Cinematch has the potential to facilitate the ability of movie enthusiasts to fulfill their movie-related needs and interests.

References

Bae, Sung Moon, Lee, Sang Chun, and Park, Jong Hun. Utilization of demographic analysis with imdb user ratings on the recommendation of movies. *The Journal of Society for e-Business Studies*, 19:125–141, 08 2014. doi: 10.7838/jsebs.2014.19.3.125.

Jimenez, Sergio, Cucerzan, Silviu-Petru, Gonzalez, Fabio A., Gelbukh, Alexander, and Dueñas, George. Bm25-ctf: Improving tf and idf factors in bm25 by using collection term frequencies. *Journal of Intelligent Fuzzy Systems*, 34:2887–2899, 05 2018. doi: 10.3233/jifs-169475.

Lee, Yong-Hun and Lee, Sang-Bum. A research on enhancement of text categorization performance by using okapi bm25 word weight method. *Journal of the Korea Academia-Industrial cooperation Society*, 11:5089–5096, 12 2010. doi: 10.5762/kais.2010.11.12.5089.

Sharma, Poonam and Yadav, Lokesh. Movie recommendation system using item based collaborative filtering. *International Journal of Innovative Research in Computer Science Technology*, 8, 07 2020. doi: 10.21276/ijircst.2020.8.4.2.

Sun, Hong Fei and Hou, Wei. Study on the improvement of tfidf algorithm in data mining. *Advanced Materials Research*, 1042:106–109, 10 2014. doi: 10.4028/www.scientific.net/amr.1042.106.

Widiasri, Monica, Tjandra, Ellysa, and Chandra, Lisa Maria. Peningkatan kinerja pencarian dokumen tugas akhir menggunakan porter stemmer bahasa indonesia dan fungsi peringkat okapi bm25. *Teknika*, 6:54–60, 11 2017. doi: 10.34148/teknika.v6i1.65.

THE UNIVERSITY
of EDINBURGH

14. Individual Contributions

14.1. s1826390 - Huacheng Song

I was responsible for the development of the tfidf algorithm on the search engine, which allows the final search results to be returned by fusing the film titles and briefs together and indexing the sorting. This progress was optimised as a baseline in this project.

I was also responsible for writing the outline of the general report, including a presentation of our group's project goals, implementation methods and results. In addition, as the coordinator between the front-end and back-end teams, it is my responsibility to ensure that the front and back ends of our project are perfectly integrated.

In addition, I worked to ensure that both teams have access to the resources and information they need to do their jobs effectively. This includes providing documentation and guidelines for the front-end team as they build the user interface, as well as ensuring that the back-end team has access to the necessary data and infrastructure.

Overall, I developed the basic search algorithms and acted as the coordinator between the front-end and back-end teams, which was critical to the success of our project. By facilitating effective communication and collaboration, together we can build a search engine that meets the needs of our users.

14.2. s1862323 - Hao Zhou

I was responsible for the development of the BM25 ranking algorithm on the search engine, which offers better performance compared to the TFIDF algorithm. BM25 allows the final search results to be returned by searching for movie titles, movie synopses, movie actors, and movie genres. This implementation is considered an improvement to the search section.

In addition, I evaluated the search performance of the TFIDF and BM25 algorithms qualitatively and quantitatively across 100 films. The conclusion is that the BM25 algorithm has improved in terms of accuracy and recall compared to the TFIDF algorithm.

In terms of collaboration, I worked with the back-end developers to integrate my search algorithms into the overall back-end system.

Overall, I have developed a performance-improving search algorithm, BM25, and have evaluated it qualitatively and quantitatively, improving the basic search algorithm TF-IDF. As a member of the back-end development team, I completed all the requested work and was also involved in the integration and optimisation of the back-end code.

14.3. s1801931 - Tianrui Xiong

My primary responsibility was to develop efficient search algorithms that could quickly retrieve relevant movie information in response to user queries. To achieve this, I developed two search methods: phrase search and proximity search. I also take part in the data pre-process which make it fit my algorithms.

The implementation of 2 search methods also contributed to the reduction of search time and memory usage as the algorithm only needs to search for keywords within a specified phrases or proximity, rather than searching the entire movie data.

In addition to the search methods, I also collaborated with my team members to develop effective ranking algorithms. These algorithms utilized a combination of the search methods I developed and other relevant factors, such as movie popularity and user ratings, to rank the search results and provide the most relevant movies to the user. What's more, I also complete the sort by algorithms, which can sort by the results by Release date, Ratings, relevance, popularity. This can help user to get the result they want.

14.4. Qiunan Wu

I played a pivotal role in ensuring the efficient and accurate processing of data, as well as developing the core functionality of the recommendation engine. My primary responsibilities were in data preprocessing, similar movie recommendation model implementation, and back-end integration.

First and foremost, I was responsible for the preprocessing of movie data scraped from IMDb. This involved tokenising, converting to lowercase, and stemming all the information obtained, which enhanced its usability and compatibility with our search and recommendation algorithms. I successfully generated four distinct inverted indices based on title, celebrity, genre, and plot, which served as the foundation for our system's retrieval and ranking functionality.

Additionally, I created short versions of metadata for each movie in our dataset, streamlining the information and making it more manageable for training our similarity model. This condensed metadata was crucial in achieving efficient and accurate recommendations based on the given movie.

As the primary developer of the similar movie recommendation model, I employed the doc2vec package from the gensim library and utilised the processed metadata to train a robust and effective similarity model. This model was able to generate movie recommendations for each movie for users, enhancing their overall experience and satisfaction with our platform.

Finally, I was responsible for integrating the back-end code provided by all group members into a cohesive and functional system. This involved merging and optimising various components to ensure the seamless operation of our platform's search and recommendation features.

14.5. s1817972 - Xudong Zhang

As the Project Manager, I have been responsible for overseeing the project's progress and ensuring that our team

works effectively towards our goals. My primary tasks have included hosting weekly team meetings, recording meeting minutes on Google Docs, and managing the project timeline.

In addition to my managerial role, I have also been actively involved in the technical aspects of the project. My main responsibility has been data collection, which is crucial for the functioning of our movie searching engine. To this end, I have developed a web crawler that collects movie information from IMDb, one of the largest and most comprehensive movie databases available.

To store the collected data, I have built a MongoDB database specifically designed for our project. This database is capable of handling a large volume of movie information, ensuring that our searching engine has access to a sufficient and up-to-date pool of movies to present as search results. Moreover, I have implemented the necessary API endpoints to facilitate seamless communication between the front-end application and the back-end database.

Through my contributions as the Project Manager and data collection specialist, I have aimed to create a robust and efficient movie searching engine that will provide users with accurate and relevant search results. I have enjoyed working on this project with our team and believe that our collective efforts have resulted in a valuable tool for movie enthusiasts.

14.6. s1837624 Richard Yuan

During the development of our movie search engine project, I was responsible for designing and implementing the user interface (UI) and front-end using Vue.js. My primary tasks included researching modern web design trends, creating a visually appealing UI, implementing the design with Vue.js, and ensuring responsiveness across various devices. I also collaborated closely with the back-end team members to integrate the front-end with the back-end systems through RESTful APIs, enabling seamless functionality and an efficient user experience.

Throughout the project, I focused on testing and debugging the front-end, addressing compatibility and functionality issues, and maintaining open communication with my team members to gather feedback and make necessary adjustments. My contributions to the UI design and front-end implementation were essential in creating a user-friendly, visually appealing, and efficient application, playing a significant role in shaping the end-user experience and ensuring the success of our group project.

THE UNIVERSITY OF EDINBURGH