Movie Index - Search Engine for Internet Movie Database

TTDS Group Project

1st s2322928	2nd s2449890	3rd s1837774	4th s2402728	5th s2443454
Leo Li	*Shuyi Liu*	*Yvonne Ding*	*Baoyan Deng*	*Zhijun Zeng*

***Abstract*—The aim of this project is to provide users with a more advanced solution by embedding a series of carefully selected features in a movie search engine. In addition, the system offers advanced search functions such as searching by keyword and genre, filtering results by colour information and year of release. These features meet the needs of realistic scenarios and provide users with a more satisfying and intuitive search process. To ensure scalability, a fully functional web crawl class is also integrated into the system, allowing the latest data to be added to the existing dataset from time to time. The project was built by a team of five students from different disciplines and strictly follows the concepts of Scrum in the Agile software development methodology. The deployed version can be accessed via https://movieindex.me**

1. Introduction

Movies, TV shows, and anime have become the most popular means for people to unwind and seek pleasure during their leisure time. According to recent statistics, people worldwide spend an average of 2.5 hours per day watching movies or TV shows. However, with the abundance of options available, it can be challenging to find a movie or show that meets one's preferences. The search process can be frustrating, with users having to sift through countless pages of search results to find what they want.

This project aims to address this problem by providing a more progressive solution, embedding a carefully selected set of features into a movie search engine. Unlike the search function offered by IMDB.com, which only allows users to query the title of movies, this search engine goes deeper by including additional information such as storyline, genre, and keywords, providing users with more possibilities when searching for a film to watch.

The concept behind this search engine is to explore every single attribute of a movie and present the results obtained by ranking algorithms, which will be described in later sections. Additionally, the system also offers advanced search features such as searching by keywords and genre, filtering results by color information and year of release, etc. These features fulfill the needs of real-life scenarios, providing users with a more streamlined and efficient search process.

The system includes nearly 115,000 documents in XML format, being a subset of the Internet Movie Database (IMDB), which was available to be downloaded from Mendely . To ensure scalability, a fully-functional web scraping class was integrated into the system, enabling the effortless addition of more up-to-date data to the existing dataset from time to time.

This project was built by a team of 5 students from different disciplines, strictly following the concepts of Scrum - a subset of the Agile software development life cycle. Although the UML design of the project was lost, the mock-up of the original front-end design was provided alongside the code in the Design folder. The team faced challenges during the development process, such as refining the ranking algorithm and optimising the search performance. However, these challenges were overcome through collaboration and continuous improvement.

Upon completing the most basic function of the search engine, the team added advanced features such as auto-translation detection, query expansion, searching suggestions, and relevant movie redirection. These features are discussed in more detail in the section on Additional Features.

Overall, this search engine aims to improve the user experience of searching for movies and TV shows, providing a more efficient and streamlined process for users to find the perfect movie or show for their preferences.

2. Scope

The scope of this project is to develop an efficient and user-friendly information retrieval system focused on movie, TV show, and anime information. The system will allow users to search and browse the data based on various criteria such as title, genre, keyword, year of release, color information, etc. Advanced search modes such as phrase/proximity search and fundamental boolean logic operations (AND, OR, NOT) will be available. Customised BM25 algorithm will be implemented to sort the search results by relevance, providing users with the most relevant information first alongside the options of sorting result by other factors.

The dataset used for this search engine is a large collection of movie, TV show, and anime information in XML format. It consists of nearly 115,000 documents and serves as the basis for the search engine's database. The data was obtained from a subset of the Internet Movie Database (IMDB) and will be used to develop and test the search engine's functionality. Additionally, the search engine will be scalable to accommodate new data in the future.

The search engine will also include a user interface developed using Vue.js 3, providing a seamless and interactive user experience. The backend will be developed in Python 3, and Flask will serve as the middleware connecting the frontend and backend. Gunicorn will be used as the HTTP server to run the

Flask application, ensuring that the application can handle high volumes of traffic efficiently. Nginx will be used as the reverse proxy server to distribute traffic to the appropriate backend server, improving the overall performance and scalability of the search engine.

3. System Design

3.1 Designing Process

User story write-up: The first stage of the design process was writing up user story — the functional requirement. This involved collecting and analysing the needs of the users and stakeholders of the system. A couple of team meetings were held covering areas such as the desired features of the system, the types of searches users would perform, and the expected performance of the system. Addtionaly, the MoSCoW method was applied by the team at this stage to help identify the expectation.
Analysis: After gathering the requirements and expectation, the next stage was analysis. This involved analysing the information gathered in the requirement gathering stage to identify patterns and common themes. This allowed for the identification of the key requirements and features of the system.
Design: Based on the requirements and analysis, the next stage was the design of the system. This involved creating a detailed design document that described the system architecture, components, and data flows. A UML graph was drawn and a mock-up of the frontend was made illustrate the design. (However, it worth mentioning that the UML graph made at this stage was unfortunately lost so the UML provided in the appendix was made while writing this report)
Implementation: After the design stage, the implementation phase began. This involved translating the design document into a working system. The system was developed using Python, Vue.js, Flask, and other relevant technologies.
Testing: Once the system was developed, the final stage was testing. The system was tested thoroughly to ensure that it met the requirements of the project scope. Testing involved both manual and automated tests to check for functionality, usability, and performance.

3.2 Architecture

The system was divided into two main parts: the frontend and the backend. The frontend was responsible for displaying the GUI to the user, allowing them to interact with the system by means such as define search criteria, select search modes, sorting methods, and filtering options, as well as displaying movie details.

When the backend service is being started, the class named RetrieveData.py will read in the XML files and a preprocessing class named DataPreprocessing.py will then step into extracting, tokenising, and using other preprocessing technology, such as stemming, to utilise the text data retrieved from the XML documents. Finally, inverted positional indices will be created from the raw data, which will provides a fast and efficient way to perform queries.

Once the user completed the query request and sent it to the server, Flask handled the request and sent it to the backend. After that, the backend had finished the information retrieval process, Flask passed the results back to the frontend, which displayed a list of results with snippets, similar to other search engines. Moreover, a customised BM25 ranking algorithm was also embedded in Query.py to sort the search results by relevance. Users could then click on one of the results to view more detailed information about the movie. Overall, the system architecture was designed in the way to be efficient, scalable, and user-friendly.

3.3 Data Structure and Storage

A original dataset was provided and embedded into the system which was obtained from Mendely and consisted of nearly 115,000 documents in XML format. To make the system more scalable, a web crawler named webScrapin.py was implemented to extract additional movie information and store it in the same XML format. The XML files in the dataset are organised based on the **docid**, which allows for easy identification and retrieval of specific resources and a sample structure illustration of the XML files is demonstrated in "Fig. 1". The original dataset was stored in the folder **"/Dataset/IMDB Movie Info"** and samples of newly retrieved data by the web crawler were stored in **"/Dataset/IMDB TestData"**.

It is essential to note that the documents in the dataset share the same structure but vary significantly in their sizes, ranging from a few KBs to more than 1MB. This variation in size is attributed to the fact that each XML file contains different amounts of information for a specific resource. Some resources may have more metadata, such as multiple genres, languages, and keywords, resulting in a larger XML file. Additionally, the length of the plot summary or the number of cast and crew members could also contribute to the size of the XML file. The original dataset deemed sufficiently large to use as the resource to evaluate the system because some files, such as 544028.xml, embedded in to the system can contain up to more than 54,000 lines and exceeding 1.3 million of characters.

Due to the nature of this project is to build a reliable and high-performance information retrieval system, the decision was made that the processed data will be stored in memory rather than Database (DB) as popular database management libraries such as MangoDB include several built-in algorithms/functions which will potentially unfairly affects the speed of retrieval while evaluating the system's performance.

4. Technology Details

4.1 Preprocessing

The preprocessing step is a vital part of the system pipeline. Its primary goal is to convert the raw input data into a structured and cleaned format, which can be easily processed by the subsequent stages. In this system, the preprocessing step involves several subtasks, including XML parsing, data normalisation and indexing creation. This section will describe how was the data being handled and normalised. Firstly, the XML files are parsed using an XML parser library, which extracts the relevant information from the raw data. The parser library helped to extract specific elements and attributes from the XML files, and the data parsed are stored in a Python dictionary format.

Fig. 1. Structure illustration of XML files

Next, the extracted data is cleaned to remove any irrelevant or redundant information that may negatively affect the performance of the system. For example, the dataset downloaded contains some URL records from other projects, which were removed at this stage. Also, during the development phase, the decision was made to ignore the running time, release dates, and colour information while preprocessing because this will significantly decrease the accuracy of retrieving relevant data. Nevertheless, a filter was built to offer users the option to perform query regarding the release date and colour information.

The next step is data normalization, where pure text is tokenized by splitting non-characters and cast into lowercase initially. After carefully analysing the dataset on hand, the rest part had been found more challenging. As most fields except "spots" contain crucial information and can not be modified in any way such as stemming and punctuation/stopwords removal, otherwise the accuracy can no longer be guaranteed. Examples of this is shown in Listing 1. After careful consideration, stemming and stop words removal were only be applied to the "spots" fields while all leading and trailing punctuations in the following fields were removed: writers, editors, producers, composers, actors, roles.

To clarify, the stopwords list used was imported from NLTK library and the SnowballStemmer used was also from the same library.

```
docid: 000009
title: #Bfl O (ggGX /STwWcfl xZs 4
...
-----------------------------------
docid: 375972
...
certificates:
    certificate: {Finland: K–12}
...
```

Listing 1. Example of complex texts

4.2 *Indexing*

After the data has been preprocessed, the next step in the system pipeline is to create an index of the data. The purpose of indexing is to enable efficient search and retrieval of information by creating a data structure that maps the terms in the dataset to the documents that contain them. An inverted positional index was generated and saved in a dictionary format of Python. The index was constructed by iterating over the preprocessed data and recording the term frequencies and their positions in each document.

The inverted index can be used to quickly locate documents that contain the user's query terms. When a query is submitted, the system analyses it and retrieves the documents that contain the query terms. The documents are then ranked based on their relevance to the query. An example of the index structure can be seen in Listing 2

```
## The term
"adam": [
    ## Number of times the term appears
    1,
    {
    ## The document containing the term
        "tt8111088": [
    ## The position where the term appears
            "307"
        ]
    }
]
```

Listing 2. Example of Inverted index

To further improve the efficiency of the search process, five fields-specified indexing dictionaries were also created for the general, title, keywords, genre, and language fields. When the "Field-specific search" option is triggered, the system chooses the most relevant index to search, thus the time of unnecessary iterations through irrelevant data can be saved.

4.3 *Ranking of Results*

Once the inverted positional index has been created, the next step is to rank the documents that contain the query terms

based on their relevance to the user's search. In this system, the BM25 algorithm was used for ranking the search results, which takes into account the frequency of the query terms and their distribution within the documents.

However, in order to account for the varying importance of different fields, the adjustments had been made to the term frequency. Specifically, when a query term appears in the field of cast, it is weighted by a factor of 0.5; in the title, it is weighted by 3.5; in the spot, it is weighted by 1.8; and in the keywords field, it is weighted by 2.3. These adjustments were determined through experimentation to produce the best results of ranking.

To exemplify, without the adjusted term frequency, a search for "Monsieur Vincent" would return irrelevant results, such as Deux femmes en or, which includes the term "monsieur" six times in the cast field and has nothing to do with "Monsieur Vincent". However, with the adjusted weights, the search returns more relevant results that accurately reflect the user's intended query.

The equation used for calculating the BM25 score and detailed descriptions are as follow:

$$score(q,d) = \sum_{i=1}^{|q|} \frac{tf_{t,d}}{k \cdot \frac{L_d}{\overline{L}} + tf_{t,d} + 0.5} \cdot \log_{10}\left(\frac{N - df_t + 0.5}{df_t + 0.5}\right)$$

where:

$score(q,d)$ is the score of document d with respect to query q
$|q|$ is the length of the query q in terms.
$tf_{t,d}$ is the term frequency of term t in document d.
k is a constant which was set to 1.5.
L_d is be the number of terms in document d.
$\overline{L}$ is the average number of terms in a document.
N is number of documents in the dataset.
df_t is number of documents contains the term t.

4.4 Advanced Search

In addition to the basic search, this system also provide proximity search, long phrase search, boolean search in different fields (all fields, title, genres, keywords and language), and filtering.

- **Field-specific Search:** The users can choose where the queried content appears (in any places, titles, genres, keywords or languages), and searching in a specific field will be optimised as the indexing were created and stored separately like mentioned in Section 4.2. For example, users can choose to search for movies whose title contains the phrase "One Day" or is the genres of "Comedy".
- **Proximity Search:** The application provides the function of proximity search. The user can input the two words and the max distance between the 2 words. After parsing the input, the query function in the sever side will firstly query for documents containing the two words and filtering the results by distance between the them.
- **(Long) Phrase Search:** When the query is wrapped in quotes (e.g."a query sentence") and the query includes at least two words, the current query will be recognised as phrase search automatically and handled by the backend. The logic of the phrase search is to call the proximity search function recursively with each adjacent pair and a distance of 1. After each recursion, the results will be merged and deducted until the last two words in the query are reached.
- **Multiple Additional Boolean Queries:** Users can merge several query conditions into a single search (*the **max** number of the additional queries is **5***). The field of searching, query type (i.e. whether it's a proximity search) and boolean search operators (AND /OR /NOT) can be set separately for each additional query line. For example, the following screenshot shows how a user can effortlessly perform a customised query with high complexity (see Figure2).
  (Any documents includeing "Basic Query" (and) **NOT** including "comedy, xxx" in the field of genres)
  OR (Title contains "Another Query" **AND** "(word1, word2)" appears in the document with a maximum distance of 3).
  The order of these queries matters, as the way the system handling these boolean queries is to split them by OR operators. An OR query always means a new query and the AND/NOT query belongs to the nearest previous basic query or OR query. In this example, the "NOT" query is mixed with the basic query and the "AND" query is mixed with the "OR" query. Furthermore, an additional query can also be a proximity query if the user activates the switch.
- **Filtering**: Users can use the released year and colour information to filter the search results when perform an advanced search by inputting a year range (from YYYY to YYYY) or by ticking the colour information select boxes, users can choose whether to search for black-and-white or coloured movies only.

4.5 GUI

For the development of GUI, Vue.js and Vite were chosen to implement the front end design. Vue.js is a progressive JavaScript-based frontend framework, which also supports TypeScript. The front end contains 3 pages, a landing page, a page listing searching results with snippets, and a page displaying details of a specific movie. Each page has a search bar for users to input their queries and perform search. If users want to add some filters or apply binary searches or proximity search, they can click on the "ADVANCED SEARCH " button to customise the query according to their needs (e.g. whether to use proximity search or not, field-specific search, boolean search, year range, colour information)(see Figure 2). Once a user submits the query, the query will be sent to the backend and the URL path will be redirected to the results page path.

In order to deliver a more enjoyable and intuitive experience, only the brief information of the top 200 results in relevance will be fetched and rendered in the front end if there are more than 200. After fetching the results data, the front end will render them with pagination, 10 results in one page. Besides, the number of all relevant results and a list of these movie ids will be returned to the front end. When users want to look up more results, they can click on the "Show more?" button to get more results. The default sorting method is by relevance score calculated by customised BM25 algorithm mengioned in section 4.3, and users have the option to switch the sorting method to either by time or by alphabet via ticking

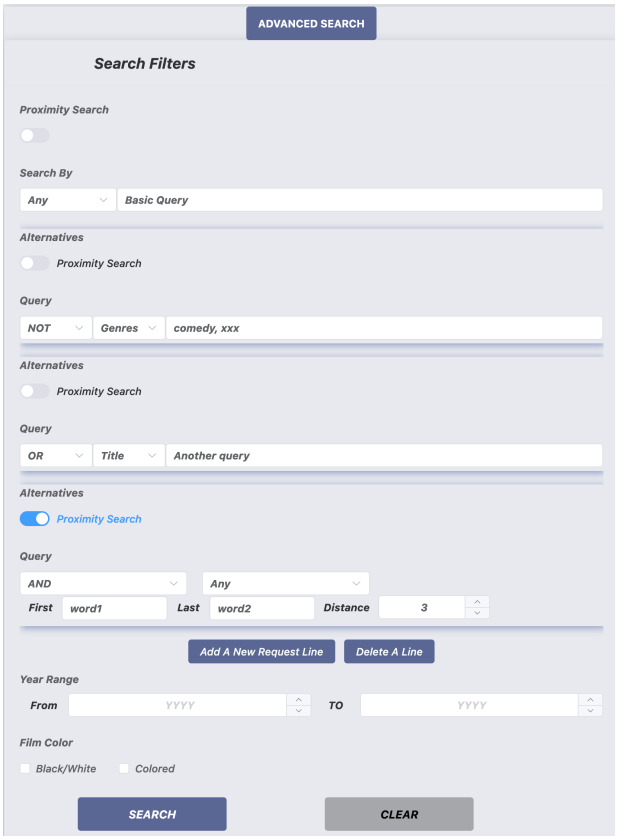


Fig. 2. Multiple additional boolean queries example

the corresponding box. The running time for handling the query in the backend will also be recorded and shown If the query has some possible typos or needs translation, a message in the format "Do you mean: $\langle suggestion \rangle$?" will appear. By clicking the suggestion, the page will be redirected to the corresponding results page.

By clicking on a specific movie on the search results page, the user will be redirected to the movie detail page. The movie poster, title, detailed information, and staff list will be shown on this page. If the movie has no poster or the poster is loading, a default image will be rendered. The genres and the keywords are linked to the results page of these words. We use horizontal scroll bars in Cast, Directors, Writers, Editors and Composers parts to provide a clean display if there are too many people here. When there are too many keywords, the page will show the first 5 keywords and collapse the rest, which can be shown by clicking the button "+ $<$*num of the rest*$>$" in the keywords area.

To further improve the user experience, Vue.js $\langle KeepAlive \rangle$ component was used to cache the recent pages. Thus, when the route goes back to the previous search results page from the movie detail page, the data will not be fetched again. Moreover, this websites is carefully followed the **mobile-first responsive design concept**, which means it'll also be platform-independent and mobile-friendly.

5. Additional Features

After the essential search functions were implemented, the system was further developed to offer additional features that enhance the search experience for users and make the system more robust. These features include:

- **Advanced Search Options:** The system offers advanced search options to enhance the search experience for users, details can be reference back to Section 4.4.
- **Spell Check Functionality:** The system also includes a spell check functionality that is ideally powered by SERP API. However, in cases where SERP API is not available, the system utilises PySpellChecker library as a backup option. The spell check feature enables the system to suggest corrected spellings for the user's search query, improving the user experience and increasing the likelihood of relevant results being retrieved.
- **Related Search Suggestions:** The system provides related search suggestions based on the user's query, allowing users to refine their search by exploring related search terms. This is also powered by SERP API.
- **Result Ranking:** The system supports result sorting, enabling users to sort the retrieved results by relevance, release date, or alphabetic order by ticking the corresponding box.
- **Cross-lingual Search Functionality:** This feature is achieved through the integration of DeepL API, which detects and translates all other languages to English where possible. If the connection to DeepL fails, then the translation uses the translation python package instead. This enables users to search for information across different languages, improving the system's usability and accessibility.
- **Mobile-first responsive design:** By strictly following the concept of mobile first responsive design, the system provides a seamless user experience across different devices.
- **Multiuser Support:** Multiuser support is also integrated into the system using Gunicorn + Nginx, allowing multiple users to access the system concurrently without compromising performance or functionality. Deteils about this will be described in Section 6.3.
- **HTML Documentation:** An accessible and easy-to-follow documentation was created and can be accessed via https://doc.movieindex.me. This provides users with comprehensive information on how to use backend apis and makes future development easier.
- **Movie Poster Retrieval:** Another notable feature of the system is the ability to retrieve movie posters. As the original dataset did not include movie posters, the getImage method from webScrapin.py was implemented. This method retrieves the image associated with a movie or TV show title from IMDb, using web scraping techniques with Python libraries such as BeautifulSoup and urllib. After searching and accessing the website for the desired image, the method returns the image's URL, which can be processed or displayed as necessary. This feature enhances the user's experience by providing visual representation of the movies or TV shows in addition to the textual information.
- **Result Caching:** In order to enhance the user experience,

the system implements result caching using the Vue.js ⟨ KeepAlive ⟩ component. This component caches recently viewed pages, which means that when the user navigates back to a previously viewed search results page from the movie detail page, the data will not need to be fetched again. This improves the system's speed and responsiveness, and allows for a smoother user experience overall.

- **Indexing Pickling:** The system also offers optional index pickling, which can help to save time and resources when processing data. Before starting the backend service, users can choose from three different modes: normal, export, or import. Details In normal mode, the system creates the indexing without performing any other action. In export mode, the read-in data and inverted positional index are pickled and outputted as files, which can then be used to avoid the need for reprocessing data in the future. Finally, in import mode, the system imports the pickle files, which can help to save time when working with large datasets. The index pickling feature provides greater flexibility and control over the indexing process, allowing optimising the workflow.
- **Naive query expansion:** In order to retrieve more relevant results, there were two measures of simple query expansion in place. First one is to query for two set of words - one for original query while the other one will be stemmed. Second measure is to split keywords where there's a dash in middle while creating index - For example, "door-knocker" will be stored as three different terms in index which are: "door", "knocker", and "door-knocker".
- **Live Indexing and Scalability:** the system's web crawler functionality enables scalability and expansion of the system. The crawler can automatically gather new data from a wide range of sources and add it to the existing dataset, allowing the system to continuously grow and adapt to changing user needs. Details described in Section 5.1.

In addition to the aforementioned additional features, the system is robust and fast in retrieving result. For short queries, results will be returned in ms. It also works well with long query - For instance, "The Lord of the Rings: The Return of the King" contains 10 terms and the time consumption will be under 5 seconds evern for phrase search. (This may vary due to different CPU and hardware where the backend service is deployed)

5.1 Web Crawler

Before beginning the web scraping process, it is crucial to examine the structure of the target webpage to identify the HTML elements that contain the desired data. The IMDb's Advanced Title Search was used to retrieve a list of videos based on specific search parameters.

The Requests library was used to send an HTTP GET request to the IMDb search results page URL based on the page's structure. The response object contains the page's HTML content, which was passed to BeautifulSoup to be parsed. However, the search results page lacks some essential information. To retrieve the desired content, the system needs to scrape the url for each individual video page from the search results page, as well as the Top Cast and Technical Specs sections of the page. The web crawler will then navigated the DOM with the BeautifulSoup object (soup) to locate the HTML elements containing the desired data. Each movie entry is enclosed in a *'div'* tag with the class *'lister-item'*. Because IMDb displays search results across multiple pages, in order to extract data from each page, the pagination has to be handled by appending the appropriate page number parameter to the URL and repeating the data extraction process for each page. Additionally, IMDb employs a number of anti-scraping techniques to prevent the harvesting of its content. To overcome these obstacles, the web crawler has to keep changing the user agent string in the requests to avoided being detected as a bot. To adhere to IMDb's terms of service, only publicly available information will be retrieved and no copyrighted images or content will be invovled.

The *getResponse* method is the main method of the **WebScraping** class, which performs the web scraping of the IMDb website by making HTTP requests to the given URL and retrieving the HTML documents. It then extracts the desired information from the HTML documents using methods, such as *getCrew*, *getCast*, *getImage*, *certainResponse*. and stores the information in the allInfo defaultdict. The information is then converted to XML format using the dict2XML method, which creates an XML tree for each movie document and writes it to a file with the ID of the movie document as its filename.

Using the web scraping strategy, more than 10,000 videos from the IMDb database were retrieved in addtion to the original dataset. These information was sanitised, organised, and incorporated into the movie search engine for processing, indexing, and proved to be well-fit.

However, there were some challenges to be faced. For instance, the sheer volume of data available on IMDb's website presented a challenge in terms of processing time and storage requirements. To overcome this, parallel processing technique was employed and cloud storage solution was used to to optimise our web scraping process. Another significant challenge is that the web pages are dynamic - Some IMDb web pages use JavaScript to load content dynamically, making it difficult to extract data using traditional web scraping methods.

6. Implementation and Deployment

This section will cover other sides of the implementation and the instruction of deployment.

6.1 API and Documentation

The backend of the system is implemented in Python and is well-written, with clear and concise code. The codebase adheres to the Google-style comment standards, making it easy for developers to understand and contribute to the project. Additionally, a easy-to-read Api documentation was made and can be accessed via doc.movieindex.me, which provides comprehensive information about the backend codebase, APIs, and data structures. This documentation facilitates the development process and helps new developers onboard quickly. This can reflect the development team's commitment to best practices and software quality.

6.2 Middleware Communication

Flask was used to handle the request from the client side, pass the request to the information retrieval functions and send

the response back. It has the following functions and their corresponding URLs. Since it was rather stressful to organise a the documentation for this part in a HTML format, the thorough explanation is included in this report as following:

- **Function *searchQuery() ('/search')*:** It's the core function to handle the user's query. It will first extract the GET parameters from the query URL. After parsing, a dict containing query, search category, additional AND/OR/NOT query and filter parameters (year range and colour) will be constructed. Then it will check whether it's a proximity search or not to decide which method in the *Query* module will be called. After getting the list of movie ids from the *Query* module, it will check whether there are any additional boolean searches and call the *Query* module to handle these queries one by one. The results will be combined according to its type (AND, OR or NOT) . After getting the final result list, the *bm25_ranking* method in the *Query* module will be called to calculate the relevance score for each id and sort them. If there are more than 200 relevant movies, only the brief description of the top 200 movies in relevance will be in the response body and sent back to the front end to reduce the responding time and improve user experience. Besides, a list of all relevant movie ids will be returned as well to ensure the user can ask for more results of the same query without calculating again.
- **Function *fetchmore() ('/fetchmore')*:** This function is called when the number of total relevant results is more than 200 and the user wants to see more in the client side. It will receive a list of movie ids (no more than 200) and return a response containing the corresponding brief descriptions of these movies.
- **Function *getMovie(id) ('/movie/⟨id⟩')*:** It receives a string standing for a movie id and returns the movie details as a JSON to the client side.
- **Function *getImg(id)('/img/⟨id⟩')* :** It also receives the movie id and then looks the title up. After getting the title, it calls the *return_img* function in the *Util* module to get the poster URL and returns it to the front end.
- **Function *spell() ('/spellcheck')*:** This function is for getting the query suggestion, spell check and translation. It uses the query message as input and returns a list of possible suggestions, spell checked query and translations.

6.3 Deployment and multiuser support

The deployment of the system was carried out on Google Cloud Platform. The backend service was deployed using Gunicorn as Web Server Gateway Interface (WSGI) and Nginx as a reverse proxy server. Gunicorn was chosen for its ability to handle concurrent requests and its simple configuration, while Nginx was selected for its strong performance and security features.

To ensure that the system could handle multiple users concurrently without compromising performance or functionality, the deployment process utilised the Gunicorn + Nginx setup to handle load balancing and process management. The number of Gunicorn worker processes was set to $numberofCPUcores \cdot 2+1$ to balance performance and resource utilisation and gevent was employed for threading control. Furthermore, pre-loading mode of Gunicorn was turned on so that the system can be more memory-sufficient. In addition, SSL has been enabled to increase the security of the data transmitted between the client and the server.

For dependencies, a **requirements.txt** file was created and can be used for resolving dependency-related problems.

The deployment process was well-documented in README.md with all configurations saved in the folder of **Code/Configs**, which allowed for easy deployment and configuration of the system on new servers if necessary. By including the detailed documentation outlining the deployment process, it is ensured that the system can be easily replicated by other teams or individuals.

7. Testing and Evaluation

Since there is no public available testing set can be used, the unit test had been chosen to test whether the backend functionalities work as desired and how much time will it consume for each query. Every aspects of the query function had been tested, including general search, keywords search, title search, language search and genres search, as well as two advanced query methods, proximity search and phrase search. While the three Boolean queries (AND, OR, NOT) and color filters was implemented in the frontend, they were manually tested via web browsers.

Two test classes were created, one without the year filter and one with the year filter set up, the former used the partial dataset from the web crawl of imdb and the latter using the 115,000 documents provided in the original download link for the full test. The test data set without the year filter is smaller, and thus the query execution time is shorter. Two documents were randomly chose the test will pass if: the desired document id appears in the first fifteen results. All the seven sets of test were passed and returned the result with no more than 3 ms. The tests containing the year filter were performed on 7 randomly selected document ids, similar to what was described above. The results took a total of 2990 ms, except for the keywords search, which took bit longer. Nevertheless, all the other tests did not exceed 1s.

After manual testing on the advanced search, it's deemed that the frontend is also fully-functional without any error. However, the system might takes longer to return the result if highly complicated advanced search were triggered.

Generally, the system appears to be well-designed and implemented, with a robust and efficient backend that incorporates a number of useful features such as indexing, caching, and pickling. The front-end is also well-designed and provides a clean and intuitive user interface. Additionally, the use of Google Cloud Platform for deployment and SSL for increased data security are both positive factors that contribute to the overall quality of the system. Overall, the system appears to be a successful implementation of a movie search engine with good potential for scalability and further development.

8. Future Work

While the current version of the system is functional and meets the requirements, there are several areas where improvements could be made. Some of the potential areas for future work are:

1) Enhancing the recommendation engine: The current recommendation engine is based on simple collaborative filtering. In the future, more advanced techniques such as content-based filtering, hybrid filtering, and deep learning models could be explored to improve the accuracy of recommendations.
2) Adding user authentication and personalisation: Currently, the system does not have any user authentication or personalisation features. In the future, there's potential of adding these features to provide a more personalized experience for users.
3) Although the team had thought of sing techniques such as delta encoding and variable byte encoding to reduce the size of the index while maintaining retrieval speed. The time constraint made it rather impossible, currently there's only data pickling developed and can be further improved in the future.

9. Development Life Cycle and Individual Contribution

The development of the search engine system followed the Agile development methodology, which allowed the team to be flexible and adaptive throughout the project. This methodology helped the team to respond quickly to changes in requirements, make adjustments to the system design, and deliver the product incrementally. Before the actual development phase, the team had carefully analysed the requirements and made initial mock-ups for the UI design(which can be found on the **Design** folder).

9.1 Agile Development Methodology

The Agile development methodology emphasizes continuous iteration and improvement, with a focus on delivering working software at the end of each iteration. The development team followed the Agile approach by breaking down the project into small, manageable tasks and prioritizing them based on user feedback and requirements. The team held daily stand-up meetings to discuss progress, identify any roadblocks, and plan for the next iteration.

The Agile methodology also allowed the team to make changes to the system design based on feedback and testing results. The team used an iterative approach to software development, with each iteration building on the previous one. This approach helped to reduce the risk of project failure and ensured that the final product met the needs of the end-users.

9.2 Project Management

The project was managed using the Scrum framework, which is a popular Agile methodology for managing software development projects. The Scrum framework helped the team to prioritize tasks, assign responsibilities, and track progress.

Since development sprint was set as weekly, weekly meetings were held to review progress and plan for the next iteration. All meetings were provided with a pre-written agenda and with meeting minutes being created so that the progress could be tracked progressively(Records can be found in the **"Meeting Records"** folder).

9.3 Continuous Integration and Version Control

While the team did not use or implement any dedicated CI/CD pipelines, GitHub is chose as the version control system. Through the use of branches, pull requests, and code reviews, the team were able to maintain a high level of code quality and ensure that changes were thoroughly tested before being merging different functionalities. Overall, this allowed the team to adopt a continuous integration and deployment mindset.

9.4 Individual Contribution

s2322928 - Leo At beginning of this project, I organised the meetings, wrote agendas, and was the note taker for each meeting. After that I moved on to contributing on the system architecture design. When the actual development phase began, I started coding in the backend side, implemented the data parsing class which parses the data from XML files and store it as Python dictionary. After designed the backend architecture, I initialised the proprecessing class and the query class. For preprocessing, I implemented most of them while a couple of fields were left over by mistake. Upon the completion of preprocessing, I finished and implemented the logic of a single retrieval.

When it comes to ranking, I developed the utility methods to calculates the required value for computing BM25 score, such as term frequnecy, document frequency, and etc. At the same time until the end of the pure backend development, I had been doing code review and debugging. Also, I designed and developed the customised BM25 weighted ranking described in Section 4.3.

After the backend was almost complete, I was responsible for interfacing the frontend with the backend. At this point, I found the proximity search and phrase search previously developed by Zhijun was not possible to be connected to frontend. And thus, in order to provides useful Apis to frontend, I re-developed the whole query class and added filtering functionality in the backend side with few more additional search methods added.

When frontend and backend is successfully connected, I started to optimised the system by implementing data pickling, reconsidering the code logic, increasing retrieval speed.

At the end of this project, I developed related query suggestion, optimised/redesigned the UI. Before the deployment, I also cleaned the code, wrote comments for code, generated HTML documentation. Finally, I deployed the system on my own and wrote most of the report. During the same time, I developed multi-user support, enabled SSL for better data security, and wrote deployment instructions.

s2449890 - Shuyi
I have contributed to the Frontend and Flask middleware to set up the connection between the client side and the server side. For the Flask part, I wrote all the URLs and the JSON data parser so that the GET request from the client side can be read and passed to the retrieval function in correct format. For the frontend, I wrote the http request method by Axios module. Besides, I wrote the scripts of frontend, such as pagination, sorting, routes (URL in the frontend), links, and cache. I also

adjust part of the GUI to make it look normal on cell phones. In addition, I contributed to the multiple boolean search part and local spellcheck/translation part. For local spellcheck, our system will use the *Spellchecker* python package and check whether there is any possible typos in 6 languages. For translation, the backend will first try the *Deepl* API. If the connection fails, then the translation uses the *translation* python package instead.

s1837774 - Yvonne

s2402728 - Baoyan At beginning, I helped the development of the backend by adding code to read the cast information from the data, removing the 'default' keyword in 'runningtimes'. I also developed some query expansions - Split keywords by "-" and store it properly. I fix few minor bugs such as directors info missing in the processed data, adding the missing preprocess of cast information, separating composer names.

s2443454 - Zhijun Based on the previous work, I finished general search and keywords search. I also initialised BM25 calculation, proximity search, and phrase search. At the end of this project, I wrote test.

Appendix

Fig. 3. Screenshot of Movie Detail

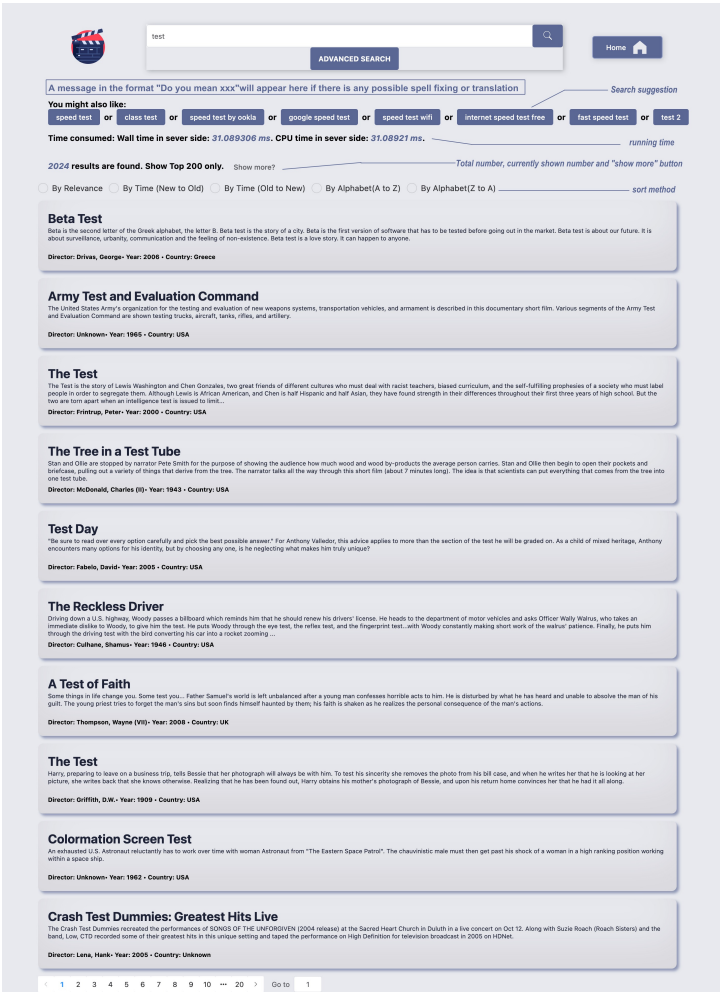
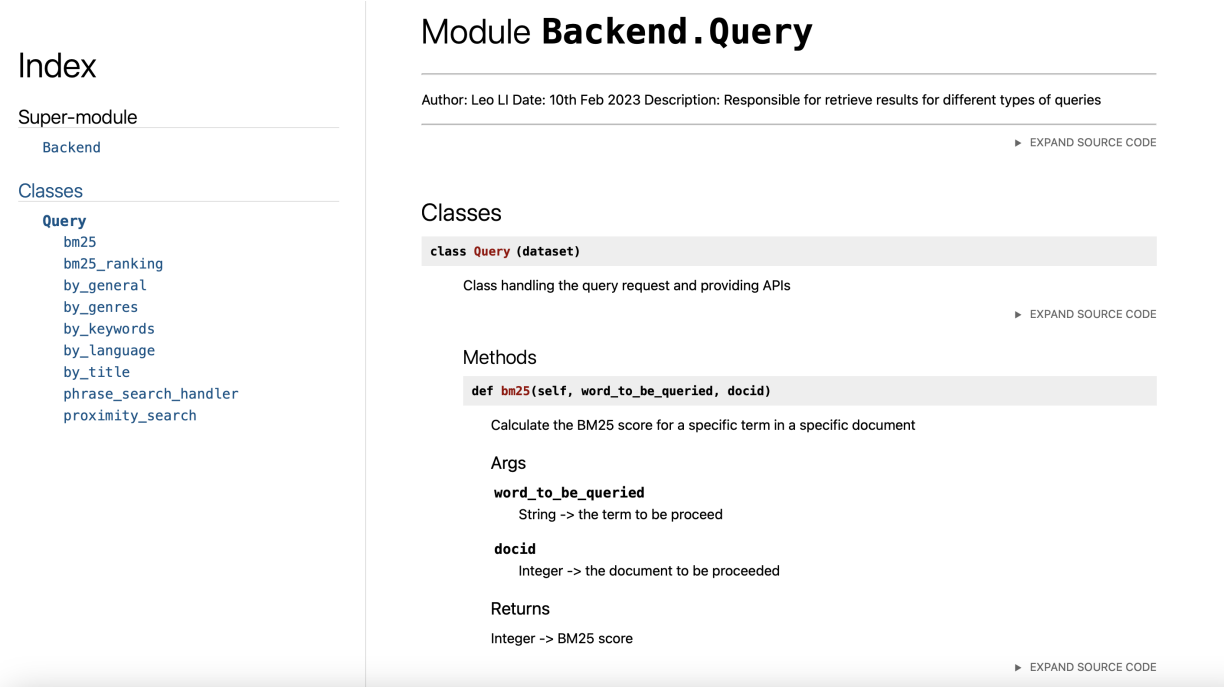


Fig. 4. Screenshot of Result Page



Index

Super-module

Backend

Classes

Query
bm25
bm25_ranking
by_general
by_genres
by_keywords
by_language
by_title
phrase_search_handler
proximity_search

Module **Backend.Query**

Author: Leo LI Date: 10th Feb 2023 Description: Responsible for retrieve results for different types of queries

▶ EXPAND SOURCE CODE

Classes

```
class Query (dataset)
```

Class handling the query request and providing APIs

▶ EXPAND SOURCE CODE

Methods

```
def bm25(self, word_to_be_queried, docid)
```

Calculate the BM25 score for a specific term in a specific document

Args

word_to_be_queried
String -> the term to be proceed

docid
Integer -> the document to be proceeded

Returns

Integer -> BM25 score

▶ EXPAND SOURCE CODE

Fig. 5. Screenshot of Documentation Page

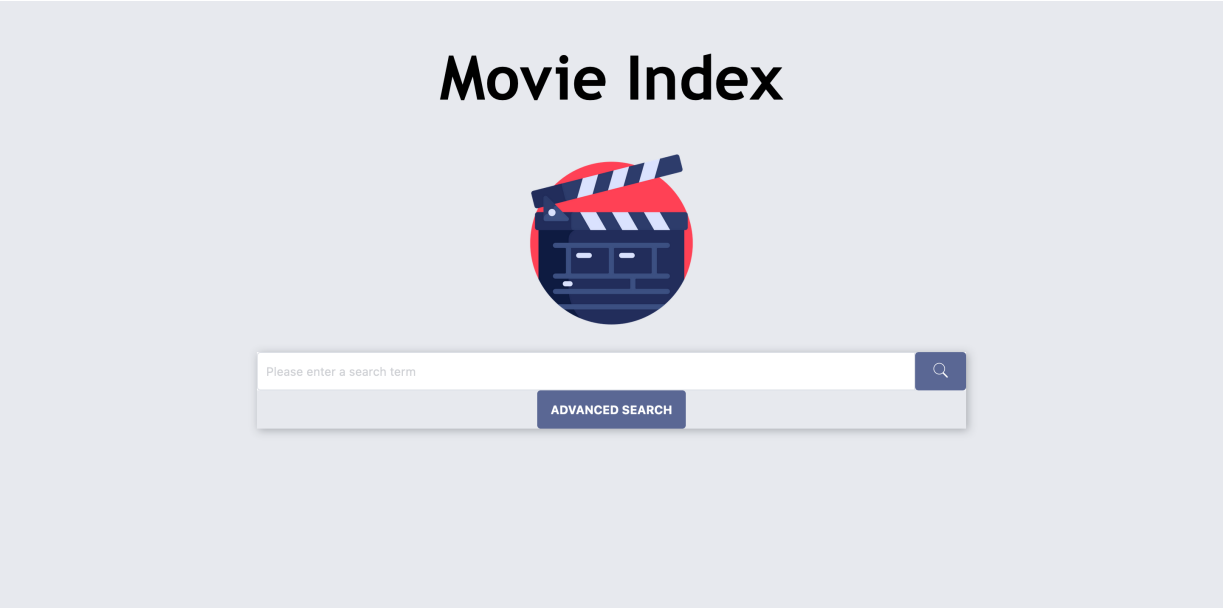


Fig. 6. Screenshot of Landing Page