School of Informatics

Text Technologies for Data Science

Coursework3 - Group45
March 2022

Abstract

Financial News Search Engine is a website on which you can browse financial related news. Enter a query on it and you can return to the preview list of related news, and continue clicking on it to read more information. The site contains 447,307 news records and articles of about 458 words in length. We ended up storing the data in a matrix, and it took up 400GB of space, which was compressed for performance. We rank adopted the TFIDF algorithm to search for full text, headlines, and full text plus headlines. Also, we added advanced search features for the richness of the project, such as filtering search by year. The site can be easily accessed through the following link: http://34.105.226.169:8080(If the link doesn't work, please contact our team leader to set the system online: s2163972@ed.ac.uk).

Keywords: Financial News Search Engine, Advanced Time Search, Big Data-set, Google Cloud Platform, MongoDB

Date: Thursday 17$^{\text{th}}$ March, 2022

Supervisor: Dr Walid Magdy

1 Introduction

Financial news is one of the most important sources of information for many financial practitioners, while the rise of the Internet has allowed news of the listing of many companies' shares to spread even more rapidly. We were inspired to design a **financial news search engine**. We searched a large amount of data and tried to use crawlers to collect it, but we found that using web crawlers required a huge amount of work, even if it was divided up, so we turned to browsing public data-sets. We then set about evaluating the types of news users were using in order to filter out a large number of high quality articles. On this site, we target users with a need for financial news who enter keywords or sentences to search, and our search engine back-end processes the returned news articles in their entirety with their titles and sorts the retrieved articles according to relevance. For advanced search engine functionality, we use a variety of search modes to meet the needs of our users. We can search for the title of an article alone, or the content of an article alone. In addition, our search engine has a clean and simple style, which makes it easy for users to interact with the front-end. Also, for the choice of database, we chose **MongoDB**, a non-relational database, to store our data.

Our search engine uses **two schemes (Dictionary&Matrix)** and an asynchronous model for version control. The Development-Matrix branch, which was merged into the main branch after being integrated into the test branch, demonstrates the iterative process using **TFIDF**. A reverse index was created and stored in MongoDB. After version 1 had successfully ensured that the main functionality was working, we worked on the **advanced search** functionality. We decided to add a **time selection** feature to the federated search. In total we made **20 versions** of changes.

This report is organized as follows. In section 2, we describe the system architecture and techniques used in the project. **In section 3**, we explain the methods used to collect and use the data. **In section 4**, we explain the front-end overview and the techniques used. **In section 5**, we give the contribution of the back-end to the search engine, including preprocessing and indexing algorithms etc. **In section 6**, describes our database and servers. **In section 7**, the implementation of the project's API is explained. **In section 8**, we evaluate the search capabilities of the project and give directions for future improvements. **Individual contributions** to the project can be found in chapter 9.

2 System Design

We started working on our project by designing the functions and the whole system architecture. Then we used Google Cloud Platform to improve our system efficiency and helped the system run faster to get the results. The final system architecture is in Figure 1. There are two parts of the system: the client part and the server part. The client part is some graphical interfaces used by users and presents the results to users. And the server part processes clients' search requests and returns the results.

Firstly, we have to add title, full text and title plus full-text TFIDF matrix files to the memory, which are calculated from the inverted index. Then users can type the queries in the input box on the website. After that, the GUI takes the users' input to the back-end program and the back-end program searches the articles. Finally, the GUI presents the results to users. In addition, users can select a year to see all the financial articles which were published in that year. On the result website, there are 10 articles shown on the same page. When users want to go to the previous or the next page, it requests the results for the previous or next page.

We used **python** language for the back-end application and **MongoDB** to store data. The API is developed in **flask** framework and the fronted is developed in CSS, HTML and Javascript. For the development environment, we use **Github** for the version control. We used the **google cloud platform server** for the deployment environment.

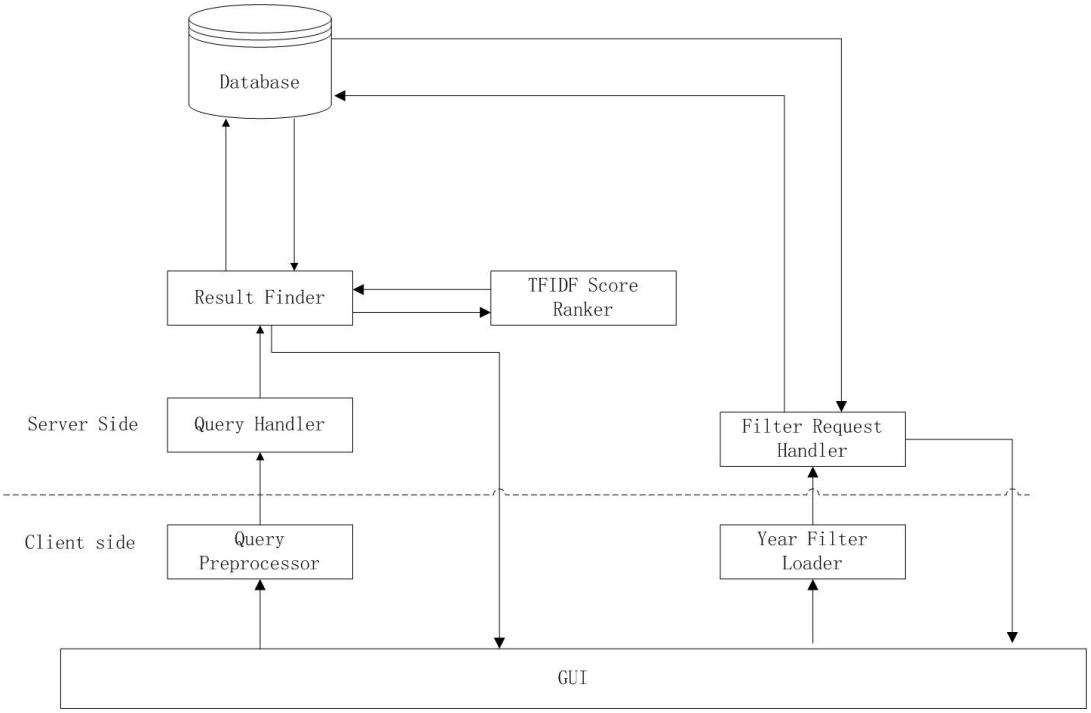

Figure 1: system structure

3 Data

We are indebted to Philippe Remy for contributing the raw data for the financial news search [1], including the title, author, date, url, and content. The data is stored in the txt format and is organised in folders. Each natural day is a folder that holds the day's Bloomberg news, with numerous pieces of information in each storey appearing in txt format.

3.1 Bloomberg Dataset

After deleting articles with missing content, our dataset has a total of 447307 articles, which are uneven distribution on the timeline. Prior to 2010, there was a dearth of data.

Table 1: article number distribution of each year

year	**number**
2012	140563
2011	139710
2013	106521
2010	59424
2007	504
2009	257
2008	230
2006	93

There are a total of 1084 records from 2006 to 2009, representing only 0.24 percent of the whole data set. In 2012, the data volume reached the highest.

In terms of months, the distribution is fairly even, with little variation in the amount of data collected between months. Between November and January, the news volume was lower; year-end and more intensive holidays could be contributing factors.

Table 2: article number distribution of each month

month	**number**
July	48108
October	47125
June	44240
September	41919
March	40724
May	39553
April	35917
February	34460
August	32322
December	28877
November	28755
January	25302

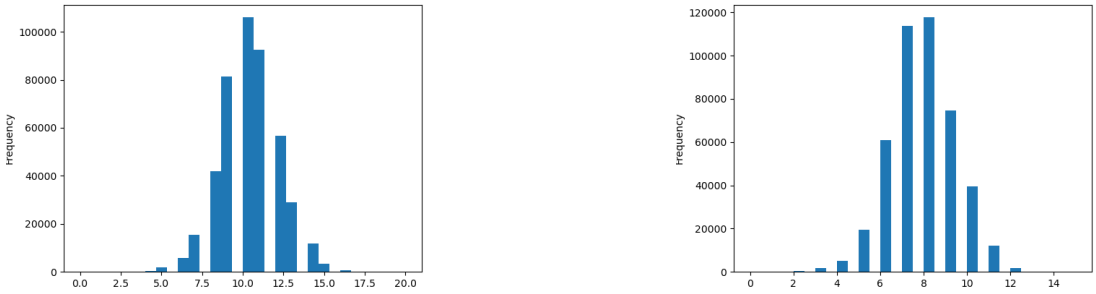

Figure 2: distribution of headline length

Before stop words are removed from the title text length distribution (the left of figure 2), the average text length is 10.26 words. The Kolmogorov-Smirnov test is used to verify normality; since the p-value is near to 0, the distribution can be considered normal. After deleting stop words (the right of figure 2), the average text length is 7.71 words, which also passes the Kolmogorov-Smirnov test.

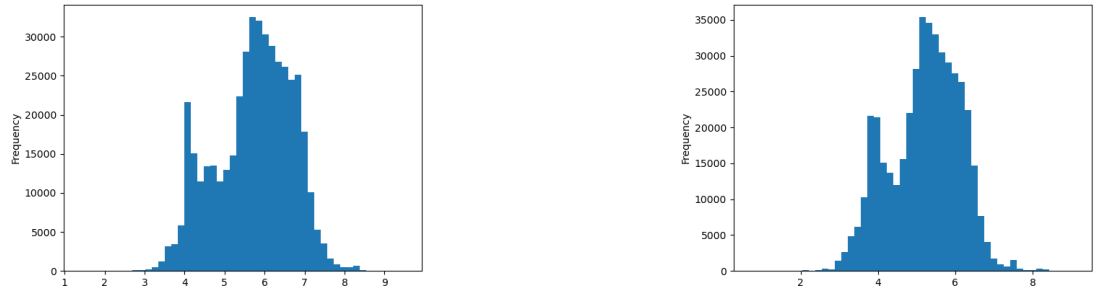

Figure 3: distribution of content length

According to the main text's text length distribution, the average text length is 464.56 words before stop words are removed (the left of figure 3). The Kolmogorov-Smirnov test is used to verify normality; since the p-value is near to 0, the distribution can be considered normal. However, the distribution map (after logarithmization) reveals an additional spike around the 50-word line. After stop words are omitted (the right of figure 3), the average text length is 268.58. It also passed the Kolmogorov-Smirnov test, and 50-word spikes remain.

4 Front-end

Front-end web pages are written in HTML, CSS, and Javascript. The website can be divided into four parts: home page, advance search page, search results page, and other pages.

4.1 Home page

The Home page provides a search box to allow users to perform basic searches, as seen in Figure 4. This interface is mainly designed for users who do not have special search needs and users can input free-text query of any length. Users can jump to the advanced search page by clicking the link below the search box. In addition, several of the news are displayed below the home page.

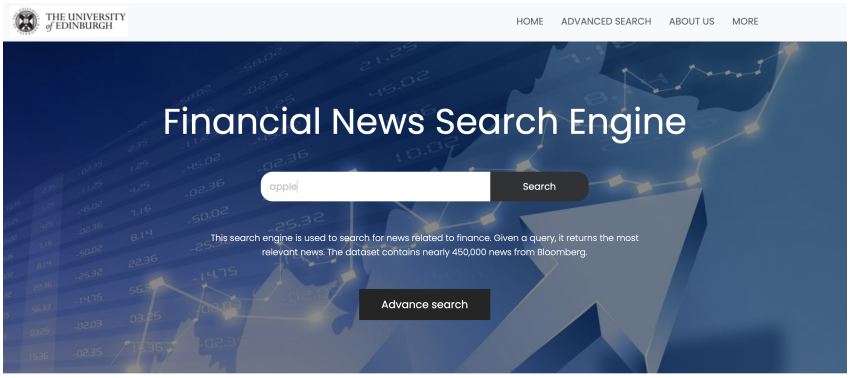

Figure 4: Home page view

4.2 Advance search page

Advanced search allows users to search for keywords by only title, only article, and full text (Figure 5). In this page, I created radio buttons for the user to select, and the default is to retrieve by title.

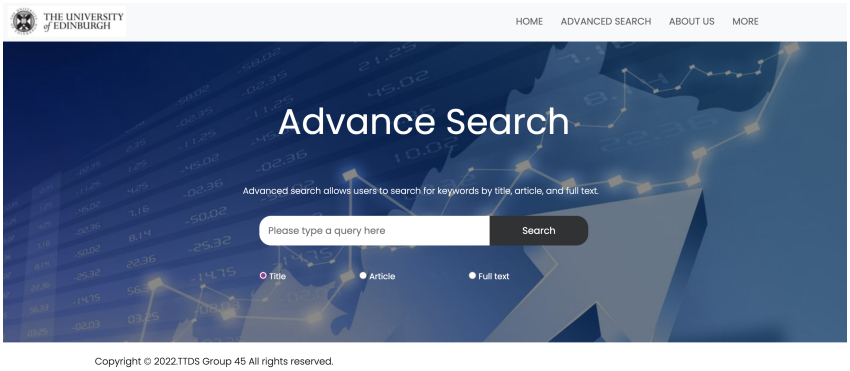

Figure 5: Advance search page view

4.3 Search results page

Figure 6 shows an example of a user-interfaced results page. The title and year of publication are displayed for each result. By clicking read more, the user can read the full text. The results page also

shows the total number of articles retrieved and the time spent on retrieval. Users can click the Anytime button to see a drop-down list, and click the drop-down list to filter the results according to time. The results interface also provides a search box, allowing users to conduct new searches more easily. The bottom of the interface is designed with page turning buttons.

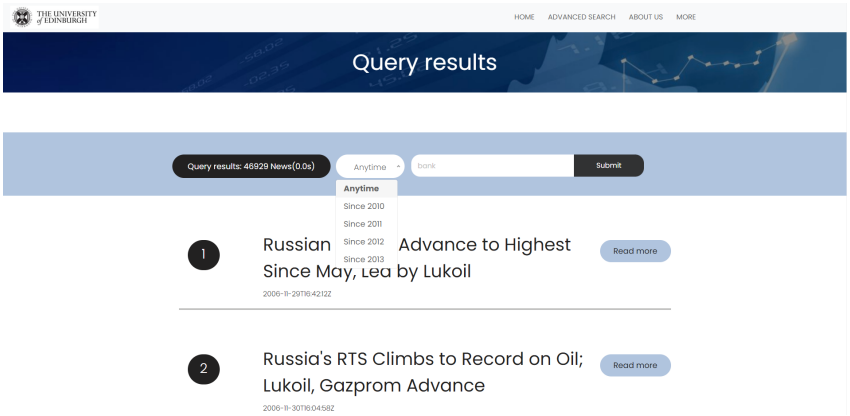

Figure 6: Result page view

4.4 Other pages

In addition to searching related web pages, the about us page and the more page are also designed to enrich the system. The about us page has the introduction and the jobs of each team members. The more page shows the introduction, system design, and data of our system. The upper sidebar of the web page can jump to various pages of the system. The badge in the upper left corner leads to the Edinburgh school website.

5 Back-end

In this section, we will introduce the back-end part of our system where search process happens.

5.1 Preprocessing

The task is to find a term that can be included in an article. We begin by parsing the txt files and extracting all structured data from them in order to create an xml file. The following 447307 articles have been extracted. After tokenizing the phrases, punctuation symbols and stop words were deleted, and the tokens were stemmed prior to indexing. Additionally, we saved the author, URL, and time stamp for each title for future reference.

5.1.1 Parsing txt files

Our parser uses positioning to determine the location of various pieces of information in txt, extracts it, and stores it in xml. Each sort of information in a standard txt file is represented by a single line beginning with "—". The first line represents the article title, the second line represents the author, the third line represents the publication date, and the content follows the blank line.

```
—— LME Says 'Expressions of 'Interest Exceed 10 as Trading Expands to Record
—— By Agnieszka Troszkiewicz
—— 2011-09-29T13:31:32Z

—— The London Metal Exchange, the .....134
```

134,952 of these articles have dislocation issues. For instance, many blank lines appear between each line of information, while "—" and the associated data display on distinct lines. All of them can be extracted following a conditional judgment.

```
<document>
  <DOC>
    <DOCNO>0</DOCNO>
    <HEADLINE>Inco's Net Soars on Higher Metal Prices, Breakup Fee</HEADLINE>
    <BYLINE>Dale Crofts</BYLINE>
    <TIME>2006-10-20T20:16:16Z</TIME>
    <URL>http://www.bloomberg.com/news/2006-10-20/inco-s-net-soars-on-higher-metal-
    prices-breakup-fee-update4-.html</URL>
    <TEXT>
```

5.1.2 Tokenization, Stopwords Removal and Text Normalization

We preprocess text using industry-standard pipelines. It's worth noting that throughout the processing, we discovered that non-standard punctuation such as Chinese quote marks and Chinese semicolons appeared in approximately 5,000 articles, which could be the result of the original author's input technique being incorrect. This issue does not exist in cw1 or cw2. As a result, we added these characters to Tokenization and subsequently transformed them evenly to lowercase. The list of stopwords we implement is included in CW1. The stemming is accomplished by utilising the **stem** function provided by **stemming.porter2**.

5.2 Indexing

Our search engine will allow you to search by article title, article content, or full text. As a result, we performed indexing independently. We employed a hierarchical inverted positional index. The first level associates the inquiry with the article's number. The second level corresponds to the article's place in which the question appears. Additionally, we calculated the phrase frequency and document length.

```
 {
    "inco": {
        "0": [3,5,6,54],
        "4": [4,7,56],
    }
}
```

5.3 Back-end Structure

For query search, we use the TFIDF ranking algorithm. As our document set remains unchanged, each term's TFIDF in a specific document is also theoretically fixed. In this way, to boost the search process, we have adopted a strategy of trading space for time. Firstly, we generate a matrix whose cell is a term's TFIDF value in a specific document. According to **Zipf's Law**, the matrix could be sparse and wasted space. Thus, we compress and save it on the disk. Before compressing, the matrix file is over **400GB**, making a massive contrast to **160MB** after compressing. A mapping dictionary from term to its unique termID is also stored.

5.4 TFIDF Calculation

Our TFIDF scores consist of two parts: term frequency and inverted document frequency. We can calculate its TFIDF weight by equation (1) for a specific term in a specific document. In this equation, $tf(t,d)$ is the number of occurrences of term t in document d, N is the amount of documents and $df(t)$ is term t's document frequency. Then, for a query q and document d, the TFIDF score is the sum of every term's weight in d, shown in equation (2). For a query q, if a document contains any one of its

terms, we calculate its TFIDF score and store it. After all documents are checked, we sort the results using descending scores as the primary key and ascending Document ID as the second key.

$$w_{t,d} = (1 + log_{10} tf(t,d)) \times log_{10}\left(\frac{N}{df(t)}\right) \tag{1}$$

$$Score(q,d) = \sum_{t \in q \bigcap d} w_{t,d} \tag{2}$$

5.5 Process Description

When our system starts, it reads the compressed TFIDF matrix into the memory. Thanks to this matrix, when acquiring TFIDF values, the system only needs to get some values from memory instead of calculating the values and repeating the traversal operations across a tremendous amount of news documents. After loading, the system will listen to query requests from the website.

When a query comes, we preprocess it in the same way we do to the documents. After preprocessing the query, we split it into terms and get their termIDs. We go to the matrix and find every term's TFIDF values with the IDs. We also acquire the Document's ID that contains these terms. Lastly, we sort the search results in descending order by TFIDF value and return the list of Doc-ID to the front-end.

5.6 Advanced Feature

Because of the diverse search needs and the timeliness of news, two new advanced search features have been added to our search engine: filtering the results by news time and searching for keywords in the news headlines only or content only. When searching, users can choose the published time of news they prefer, like **'since 2012'** by a dropdown box. Another feature is in the Advanced Search module, where users may search for news **headlines only** or **contents only**. We build two additional indexes for this as well as two extra matrices.

6 Database & Server

6.1 Database

We used **MongoDB** to store all of the financial news collections with JSON format, which can be accessible for any function. The database structure is shown in Figure 8. And the total data is 1.28GB. The main function of the database is to store all the collections, find records by document id and filter data according to user's requests. We used the **flask_pymongo** package to connect the back-end program to the database.

6.2 Server

We deploy our system on a virtual machine on **Google Cloud Platform**. As we need to load the whole matrix into memory, we use a Virtual Machine with 16 vCPU and 128GB memory. The operating system is **Ubuntu 18.04.6 LTS**. The VM is connected to our **MongoDB** database server, where news details are stored.

7 API

Application Programming Interface (API) is a software middleman that allows GUI, back-end program and the database to communicate with one another. We used **flask** to build the API, which is a very

Figure 7: data structure

famous and popular lightweight Python web framework. We also used the **flask_pymongo** to connect to the MongoDB database.

There are two search functions in the system: fundamental search function and advanced search function. The fundamental search function receives the users' input from the website. Then it calls functions for processing the input and getting the document id lists. After that, the program can use the id list to find the financial news collections' headlines, authors', published time, and the content. Then, GUI presents the news title, authors, published time and content on the website. Users can click the "Read more" button to see the full content . On the result website. There are some years that users can select to filter the results. If users select one year, the back-end program can receive the requests and use the MongoDB API to filter eligible results. Then, it will present the results to users.

The advanced search function works similarly. It has the same input and similar processing progress. There are three types of advanced search: title search, content search and title and content search(this is the fundamental search function). Title search means searching the user's input against words in the title. Content search means searching the user's input against words in the contents. Same as the fundamental search, users can also select one year to filter the results.

8 Evaluation and Future Work

8.1 Evaluation

We tested our search engine on a virtual machine on Google Cloud Platform. The environment and configuration we set up go as follows: 16 vCPU and 128GB memory, Python 3.9.7, g++ (GCC) 4.8.5. We conduct query searches with 1, 5, 20 words respectively on both standard search and advanced search to test the system's performance. To reduce testing error, we performed five tests for each scale with different keywords and calculated the average. The results are shown in table.

Table 3: Table of Time Consumption in Test(in seconds)

Query Scale(word)	Standard Search	Headline Search	Content Search	Time Filtered Search
1	3.804	3.774	3.994	4.198
5	4.612	3.628	5.864	5.052
20	12.14	3.954	12.936	13.82

From the result, we can conclude that our search engine needs more time to process when the query grows larger. However, the processing time of headline search shows no noticeable fluctuation. This might owe to the small scale of headline data compared to the news itself.

The average article length in our dataset is 458 words, and the total document amount is 447307. With this scale of data, the results achieved by our search engine are acceptable. Both our search engine's basic

and advanced features work fine and adequately. Our GUI is easy-understanding and straightforward. For users who are interested in financial news, they will find our search engine a valuable and multi-functional tool. However, indeed, there are many future developments that can be performed to improve its performance.

8.2 Future Work

Our search engine is already great, but there are still a few points that can be improved:

1. **Dynamically generate indexes.** Our index is static and we think that if there is an updated news release, we cannot automatically update our search engine content.So that's where we can improve in the future. Server memory is limited, so which data source to take and which rebuild strategy to use (Complete Re-Build, Re-Merge, In-Place, Hybrid) is worth thinking about.
2. **Increase the amount of data for your news.** It can be seen that although our data has 400G of storage matrix before initial compression, it is still very small compared to Google.
3. **More advanced compression processing algorithms.** We've adopted .pkl files, but there are still a lot of possibilities worth exploring.
4. **Add more advanced search function.** For example, query search, expansion search, etc.
5. **Recommendation algorithm.** In order to achieve personality recommendations, the user's personal information may need to be considered, and it is also an extension that can be considered for model training and user feature labeling.
6. **Resolution adaptation issues for mobile devices.** At present, the web page only supports the access of the web end, and the UI of the mobile end does not have an adaptive resolution, making the user experience of mobile phone access not good enough.
7. **CI/DI control**. Use git-lab to help design the pipeline to achieve automatic test and publish.

9 Individual Contribution

9.1 s2163972-Peiyao Guo

I worked on the API and database functions. Specifically, I created all the functions to connect GUI(front-end), back-end program and database to one another by using flask and flask_pymongo packages. I implemented all the search functions in the API by using flask package, which receives the users' input and returns the search results to the GUI. I also wrote functions which are associated with the MongoDB database and established the database to store data. I converted the financial news collection txt file to csv file and stored all the data in MongoDB database. Then, I used the flask_pymongo and wrote functions to connect our project database on the MongoDB Cloud to our program and find financial news records by document id according to the search results and filter the database results by the published years.

After finishing my work, I also tested the whole system to know whether the connections between front-end websites, back-end functions and database work well. For the report, I wrote the **System Design**, **Database** and **API** part.

I am also the group leader and mange the whole group work, including task assignments, coursework progress and holding a meeting every week to discuss the finished work, some problems and weekly future work.

9.2 s2259358-Gang Zheng

I worked on the back-end with Zhe Wang as well as server deployment section and evaluation part. My contribution is mainly focused on the search process after the data is preprocessed for the back-end part. I designed the back-end structure and calculated the TFIDF value, which is stored in matrices. Then I formalize the results and send them to the front-end.

I created an instance on Google Cloud Platform and set up the requirements and environments. Then, I deployed the alpha version of our system on the VM. I then continue to test and iterate on our releases while fixing various bugs and adding extra features to our system. We have made more than 10 prototypes of our system. For the report, I finished the **Back-end**, **Server** and **Evaluation** sections.

9.3 s2151508-Lanshan Xu

I am responsible for all design and implementation parts of the system's front-end. I finished practically all of the GUI's features, from the architecture to the rendering of the results into each component. More specifically, I firstly designed the structure of the web page using html, including lists, radio buttons, etc., and designed how users should interact. I looked up a lot of elements and added them to the webpage. Then I used CSS to make the user interface more user-friendly. At last I used Javascript to complete the user and web pages interactive behaviour. In addition, I found some students to use our webpage and put forward suggestions, so as to optimize our webpage and make it more user-friendly. Also, I came up with the idea about how to retrieve the query. For the report, I finished the **Front-end** section.

9.4 s2186708-Chenyan Jiang

Responsible for front and back-end connection and managing API design documentation using swagger-hub (appendix). Responsible for version control, using github (appendix). Building 20 versions of the prototype. Completed Post function for the API. Build the project structure and establish the overall architecture in the initial git version using the flask package to successfully connect the front and back ends. Worked with teammates to design database and use CSV format for database storage. Tested each version and corrected them, such as not being accessed by external IP, and dead loops in page-flipping repeat searches of the full index. For this report, I have worked on **abstraction**, **introduction**, **future work**.

9.5 s2187031-Zhe Wang

I work with Gang Zheng on the back-end. My responsibilities include the following: I came up with ideas about data sources based on the results of our first meeting discussions, and fetching and processing raw data.I designed parser using positioning to extracts information from original dataset to convert the information in txt into xml structure; I inspected data and perform data cleaning and structuring so that we can find issues such as Chinese punctuation that affects indexing and ranked retrieval; I preprocessed text using standard pipelines, including tokenization, stopwords removal and text normalization; I make descriptive statistics for the dataset; I employed hierarchical inverted positional indexes. For the report, I completed **Data** and part of **Back-end**.

References

[1] Xiao Ding Philippe Remy. Financial news dataset from bloomberg and reuters. `https://github.com/philipperemy/financial-news-dataset`, 2015.

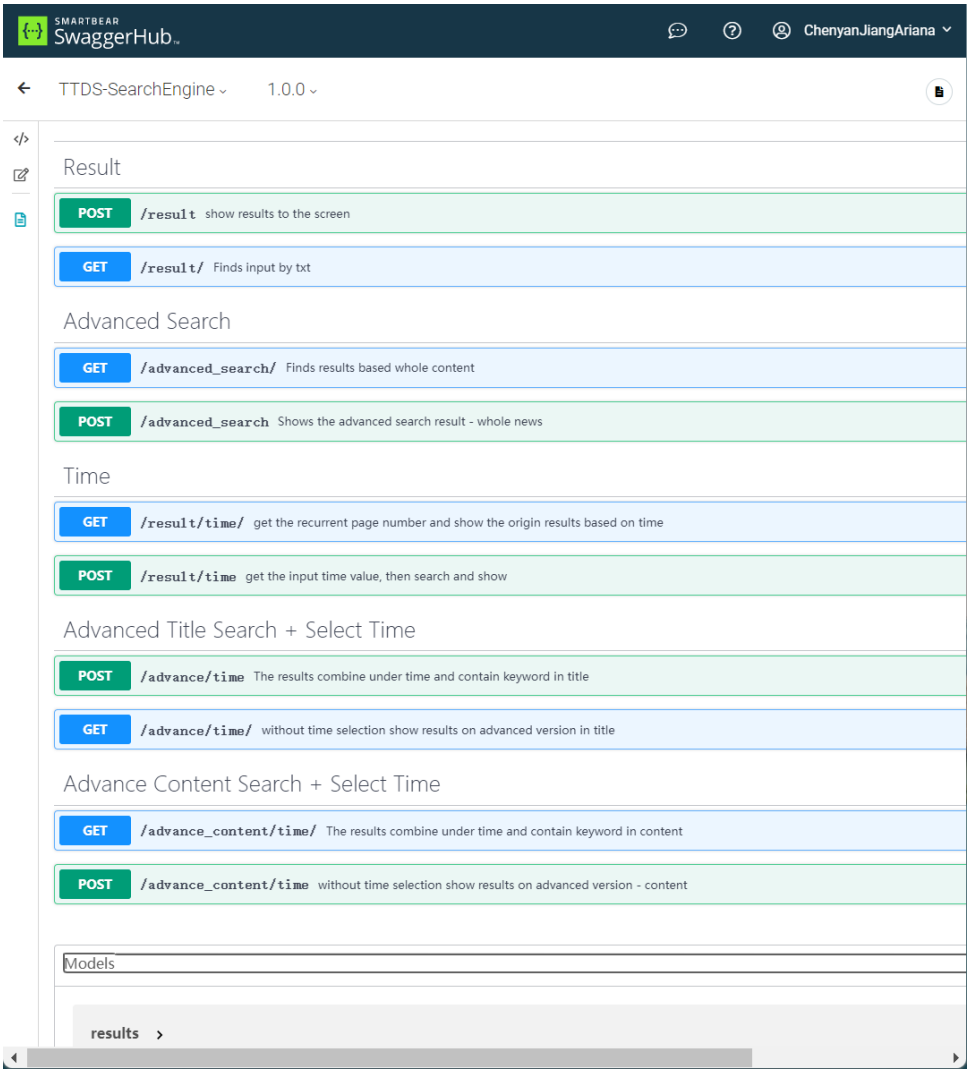

Figure 8: API overview design

10 appendix

10.1 Version control - git

`https://github.com/ChenyanJiangAriana/search_engine_TTDS_group45`

10.2 API visual - Swaggerhub

`https://app.swaggerhub.com/apis/ChenyanJiangAriana/TTDS-SearchEngine/1.0.0`