TTDS Coursework 3

s2053694, s2120856, s2041707, s2184385, s2176866, s2238615

Abstract

The news search engine is a website that provide users a way of searching relevant news from incomplete news title. Given a query, it returns the most relevant news. The collection of documents contains 8.8GB news from *kaggle*. It uses a multi-level inverted index occupying 5.92GB on disk. The ranking algorithm uses BM25 for phrase search. Instead of searching certain type of news, the website can also perform a search for all most relevant news to the query using weighted BM25 algorithm. It also contains advanced search features, such as search based on date. The website can be reached from this link: 35.184.241.193:5000/search

1. Introduction

News is information about current events. This may be provided through many different media: word of mouth, printing, postal systems, broadcasting, electronic communication, or through the testimony of observers and witnesses to events. Common topics for news reports include war, government, politics, education, health, the environment, economy, business, fashion, entertainment, and sport, as well as quirky or unusual events. Government proclamations, concerning royal ceremonies, laws, taxes, public health, and criminals, have been dubbed news since ancient times.

News is an important way for people to find out what is going on, but as there is so many medias and types of news, it is hard to retrieve news from Internet. The situation inspires us to create a search engine for news.

Our website is targeted on users, who remember parts of the title of news or those who want to search relevant news with a query and want to find out the details about the whole news. The search engine returns some news article with title and content that relevant to the query, the media and author, and other details.

To be able to retrieve the most relevant results, a large number of news is necessary. However, it is not trivial to collect this number of quotes using web crawling, it can only be a means of upgrading our database. We used a dataset of existing news information, which included millions of news items from the past few years, provided by the *kaggle*. In our design, we followed the way we generated the inverted index in CW1, by iterating through the title and article of the news information in the database to generate an inverted index for each token. For the news search support, we created a positional inverted index. Since we do not have enough memory in our own computer. So we use google cloud to get the whole inverted index. We decided to use MongoDB which is based on B-Tree data structure to store the inverted index.

For the news search, we use the BM25 ranking algorithm coupled with popularity weighing. For the news search, the results are sorted by the popularity.

After we made sure that all the basic features of the search engine are functions very well, we started researching and implementing some advanced features which will be discussed in a later section. We decided to implement the Query Completion feature, which uses a language model to predict the next word given the previous one typed by the user. The model was trained the model using a Transformer (GPT2(Radford et al., 2019)).

This report is organised as follows. In Section 2 we describe the system architecture and the technologies used in the project. In Section 3 we explain the methodology used for collecting and storing the news dataset. In Sections 4 and 5, we explain the preprocessing methods and the indexing algorithm of the quotes that we used in detail. We describe the BM25 algorithm in Section 6 and explain the advanced search functionalities spelling correction and query completion implemented in the project in Section 7. The additional features is explained in Section 8. The design of the API is described in Sections 9. The project is evaluated and future work is discussed in section 10. The individual contribution of each team member is shown in Section 11.

Figure 1 shows architecture of the project.

2. System Design

2.1. Framework and modules

The goal of our project is to develop a website for users to do some news search on it, so the first question of our development work is choosing a suitable web framework. Our website is built by Flask framework, one of the most popular python website framework, and compared to Django framework, Flask is a microframework that can be easily extended and has no pre-existing components or libraries. It is very suitable for our project because the most important part of the project is algorithm and data collection.

First of all, we need a data collector module providing all of the data use in the project such as subtitle files and news data. This module collects all of the data and stores them in our system storage space in a customized directory format specified in section 3. After collecting the files we need, they should be processed and the inverted index should be generated. Hierarchical Indexer module processes the collected data and produces the hierarchical positional inverted index as described in sections 4 and 5. The GUI takes the user inputs and runs the search queries. Query completion module gives some suggestions to users while they are writing a search input. When the users finish writing and run the search algorithm, the GUI parses the query and sends it to the server. The documents retrieved from the inverted index are sorted by ranking results in the ranker module and stored in a session cache. If the user wants to see the details about any of the result quote, GUI requests and shows the details to the user.

To develop the system described above, we chose MongoDB as the database and Python language for the backend application and scripts. Our API is developed in Flask framework and the frontend is developed in jquery. For the deployment environment, we decided to use a cloud server (Google cloud) to host the DB, backend/frontend applications and the API. We used github for the version control.

2.2. Version control and teamwork

We used github for the version control.

We do not follow strict requirements of agile development or waterfall development, we reimplement our CW1 code and made it to a basic project. Then develop the website for query submission and result display, meanwhile do the preprocess of the data collection. After all of these, we also develop some additional features like query completion and database update.

3. Data Collection and Storage

We have collected many news datasets, but those that meet the required size are rare. We also tried to obtain website data by crawling, but due to the time-sensitive character of news, it is difficult to collect a sufficient amount of data in a short period of time by crawling on well-known news websites. Finally, we used a dataset provided by *David McKinley* on the *kaggle* website[1]. This dataset contains news information from the last few years and is a good fit for the search engine we plan to build.

3.1. Dataset

The basic data was provided by *David McKinley* and contains many attributes related to news articles such as: date of publication, authors, title, article, URL, etc. The dataset is 8.8 GB in total, about more than 220 million pieces of data, and is stored as a CSV file, with each attribute corresponding to a column, which also provides us with great convenience in pre-processing the data.

At the same time, considering the timeliness of the news information and the fast update speed, we have also implemented the function of regular database update. The data for this section is sourced from the *BBC News*[2], a leading news website that divides its news data into various sections. We chose to collect news from eight categories over time, including World, Business, Politics, Tech, Science, Health, Family&Education and Education&Art. This data is used for the regular expansion of the database. We used an existing crawler tool to obtain the news information and filtered it by the time of publication for de-duplication purposes. We removed redundant attributes such as download time and image information.

3.2. Database

There are various database management systems for us to choose in this assignment such as Mysql, PostgreSQL and Mongodb. Most of them are relaional database, After taking into consideration, we finally chose to use Mongodb to store our data. The reason is that Mongodb is convenient to process the structure in python especially dictionary and list. It can store every relation according to index from nested structure of the dictionary. The store structure of Mongodb is B-tree. The characteristics of B-trees are better suited for single data queries, which also used in search engine. So mongodb is our final choice.

In our database, we have two tables. One is named *index*, the other is called *news*. In the index table, we use database interact code to upload the Inverted Index. And in the news table, the content is our original data with simple pre-processing that those with empty title and article are deleted. We also built some indexes inside these two table to enhance the speed of searching. In this project, the back-end will give database a word to get document numbers. So the speed of searching word is important. To satisfy this requirement, we build word index inside index table. The condition in news is same. We get document number from back-end and return details of each article to the back-end. Therefore, we built id index inside news table. There is a limitation in Mongodb. The biggest size of each item is 16M. Under this circumstance, some word may have more than 1 item, as shown in Fig 2.

3.3. Database API

We have created a class called *MongoDB* to implement the interaction with the back-end. The main functions provided by the *MongoDB* class are inserting new data (news and inverted indexes), querying news, querying inverted indexes and some filtering of query conditions.

The functions of inserting new news information and inverted index are mainly used for regular updating of data, where the id of the index entry is obtained by returning the largest id in the database and then adding 1. After the

[1]https://www.kaggle.com/datasets/davidmckinley/all-the-news-dataset

[2]https://www.bbc.co.uk/news

successful insertion of news information, the corresponding file number is generated for the generation of the inverted index according to the id number they are finally stored in the database. The Query News function returns the contents of a document based on the document number provided. The query inverted index function returns the corresponding inverted index based on the token provided. It is worth noting that when a token has multiple index entries in the database, this function can combine them into a complete inverted index and return it.

In addition to this, a simple date filtering function is provided in the *MongoDB* class. This function compares the time values stored in the date attribute of the news information to see if they are within the time range provided and ultimately returns only the news information that meets the criteria.

4. Preprocessing

4.1. Dataset Processing

We have tried a number of methods for pre-processing the data. After an initial look we found that there were very many empty rows in the dataset file. To deal with these empty rows, we filtered the first element of each row in the file and only used data beginning with numeric elements. However, using this pre-processing method we found that the original data would be fragmented, i.e. each news message would lose many attributes.

We then tried to leave the empty lines unprocessed and store the whole as text in the article attribute as part of the article attribute in the news message.Using the *csv.reader* function to extract the elements from the dataset, it was easy to obtain well-organised news information attributes. After filtering, we only kept five attributes, namely: *date* (time of publication), *authors*, *title*, *article* and *url*, as shown in Listing1.

```
{
    _id: 0
    date: "2016-12-09 18:31:00"
    authors:
        0: "Lee Drutman"
    title: "We should take concerns about ..."
    article: "This post is part of Polyarchy..."
    url: "https://www.vox.com/polyarchy/..."

}
```

Listing 1. Example of news information stored in the database

Since the target of our search was the title and article of the news information, information with these two attributes empty was of no use to us.After filtering, we removed such news items.

4.2. Tokenization, Text Normalization, Stopwords Removal and Stemming

We first remove all punctuation in our dataset to generate inverted index. Then we turn all texts in lower case. Considering our dataset category, we will apply stopping words removal to our dataset, because people will be more likely to search one piece of news based on some keywords. They will use a place name to find some regional news or a specialized vocabulary to search for news in a field. Stemming is also necessary because most of words from the same stem have similar meanings and we should not make our search too strict, which may lead to missing some important information.

4.3. Query Processing

To match our dataset and make sure we can find words in our database, we should do the same operations on the query as the original data prepossessing, which are stopping words removal, punctuation removal and stemming. Addition to preprocessing, our search distinguish Proximity search from other search queries by "#", so we need to check whether "#" exists and then preprocessing the search query.

The preprocessing of query must be unified with the preprocessing of the dataset, because only same preprocessing procedures could guarantee that the search will match the index. Therefore, tokenization, text normalization, stopwords removal and stemming were same as dataset preprocessing.

We split the preprocessed query into three list based on "AND", "OR" and "NOT". These three lists of words will be used for searching respectively. If there are not any of these three words, all words will be bound as a list, which means they are considered as a phrase, and then it will be put in "OR" list.

5. Inverted Index

We assume that the users' input is one of the query of keywords to search an article, the name of an article, and a time range. We will explain search features in the next section. In this section, we complete inverted index to support the search features. The index structure we decide to use is same as what we have implemented in CW1. We also need the length of each document to complete BM25 ranking algorithm, but we choose to save this information in another pkl file to keep our index simple which could save searching time. We have tried several ways to run the index generating code which was implemented based on the CW1 code. Since the dataset was so large that our existing hosting resources could not support the amount of computing required to generate their inverted indexes, we finally use Google Cloud to help us with the generation of the inverted indexes.

As in MongoDB, the BSON size of a single document cannot exceed 16M, but the inverted index of some common tokens in the generated index is relatively large. Obviously 16M is not enough for the storage of these token inverted indexes. Our solution is to split them into multiple documents and store them in the database. This means that a token may correspond to more than one document in the database.

6. Ranked Retrieval

6.1. Searching Strategy

6.1.1. Boolean search

Boolean search is to use some special conjunctions to split a query into several sub-groups and process these search results of sub-group based on the conjunctions. As introduced in Section 4.3, a query will be split into three lists represent for "AND", "OR" and "NOT". These three words lists will be used for searching and search results of each phrase from "AND" list will be intersected, search results from "OR" list will be unioned with the results of "AND" list, while search results from "NOT" list will be subtracted with "OR" list or all documents in the database if there is only "NOT" existing in the query.

6.1.2. Phrase search

Phrase search is to find a list of words which appear adjacent. In our search system, we consider all consecutive words as phrase search in the query, in other words, phrase search is our default search method and users no longer need to use quotation mark to tell whether our search system the query is a phase. If users do not hope a phrase search. “AND” “OR” “NOT” could be used to separate it into several phrases. For example, if one query is “A B AND C”, we will regard A and B as one phrase, while C is another phrase.

Phrase search in our search system supports phrases of any length. To meet this requirement, we firs find all documents that the keywords in the query are in common, then compare the locations of each words. Only when a document has at least a string of location lists such that each keyword in the query can correspond to the location list, this document can be confirmed as one search result. In our code, we used a double cycle to traverse all key words in the phrase, compare the location distance between these two words and judge whether the distance of locations in the documents equal to the location distance of these two words in the query. If all words meet this condition, we will put the document in the search result list and rank them based on BM25 algorithm.

6.1.3. Proximity search

Proximity search is a kind of less strict phrase search. In phrase search, the relative position of the word in the document must match that in the query. However, in proximity search, the distance of key words in the document will be determined by users. As introduced before, query preprocessing will check whether a query is a proximity search query by judging whether the first character in the query is "#". If so, the proximity distance could be parsed from the query and this variable will be used to distinguish phrase search and proximity search because the default value will be -1 if it is not proximity search. The searching process is similar as phrase search. The differences between them are different requirements for the distance between words.

6.2. BM25 Rank

All search results will be ranked based on BM25 algorithm in our search system. BM25 algorithm is an improved rank algorithm based on TFIDF. The formula of BM25 algorithm includes three main parts.

$$IDF(q_i) = log\frac{N - df_i + 0.5}{df_i + 0.5} \tag{1}$$

The first part is the weight of words $IDF(q_i)$. In formula (1), N is the number of all documents. df_i is the number of documnets including word q_i. This part is similar as that in TFIDF algorithm.

$$S(q_i, d) = \frac{(k_1 + 1)tf_{td}}{K + tf_{td}} \tag{2}$$

$$K = k_1(1 - b + b * \frac{L_d}{L_{ave}}) \tag{3}$$

The second part is the relevance between words and documents. In formula (2) and (3), tf_{td} is the word frequency of word t in document d, L_d is the length of document d, L_{ave} is the average length of all documents, k_1 could be adjusted to normalize the range of term frequency in an article, while b could also be used to adjust the effect of text length on relevance.

In the TFIDF algorithm, the larger the value of the relevance between words and documents tF_{td}, the larger the value returned by the entire formula. However, in BM25 algorithm, the relationship between word frequency and relevance is non-linear, which means the relevance score of each word to the document will not exceed a certain threshold. When the number of occurrences of the word reaches a threshold, its influence will not increase linearly, and the threshold is related to the document itself.

$$S(q_i, Q) = \frac{(k_2 + 1)tf_{tq}}{k_2 + tf_{tq}} \tag{4}$$

The last part is used to calculate the relevance between words and queries, which is important for long queries. In equation (4), k_2 is used to correct word frequency range in query.

$$BM25(q, d) = \sum_{t \in q} IDF(q_i) \cdot S(q_i, d) \cdot S(q_i, Q)$$

The final BM25 score is the product of the three parts. Compared with TFIDF, the first main advantage is as introduced that the relationship between word frequency and relevance is non-linear. The second advantage is that it has three adjustable parameters and they could be modified based on the performance of ranking.

6.3. Result Process

```
{
    query: Polyarchy
    date: "2016-12;2018-6"
    allnews:[
    {
        title: "We should take concerns about ..."
        article: "This post is part of Polyarchy..."
        author: ["Lee Drutman"]
        url: "https://www.vox.com/polyarchy/..."
        date: "2016-12-09 18:31:00"
        highligh: {
            name: []
            content: [(21,30)]
        }
    }
    ]
}
```

Listing 2. Example of news information returned to the web

After search and rank, twelve search results with the top BM25 score will be transferred to the front end first. The main task of processing data from the database is to extract a snippet of news based on the location of key words, find all locations of key words in this snippet and keep locations in a list, which is used to tell the front end which words need to be highlight.

To find highlighted keywords of the query, there will be incomplete matches regardless of whether the keywords are stemmed or not. For example, if we implement stemming, "city" will be converted to "citi" and all "city" in the news will not be matched. "lover", "loving" will not be matched if we do not implement stemming on "love". Therefore, keywords with stemming and without stemming should all be used to match the words in news one by one.

7. Spelling Correction and Query Completion

After finished the implementation and the testing tasks of our basic features, we are now focusing on using natural languages processing methods and recommendation algorithms to implement some advanced features. We wanted to implement query expansion and similar search recommendation at first, but these features were kind of useless for our search engine and might cause a time delay in our website. We decided to drop these features and we will discuss them more clearly in our Future Work part. In this case, this part will focus on the introduction of spelling correction and query completion which are two features we implemented successfully.

7.1. Spelling Correction

Spelling correction is the task of predicting whether the word is misspelled and returning the correct word automatically (Whitelaw et al., 2009). This additional function can automatically detect the misspellings which are made by the users and recommend the correction to the users immediately. Usually end-to-end spellchecking system uses an error model and an n-gram language model for the correction task (Whitelaw et al., 2009), we implement our spelling correction program based on the work of Peter Norvig. The program consists of four parts, selection mechanism, candidate model, language model, and error model. As for selection mechanism, we use max with a key argument to achieve 'argmax' in Python. The candidate model returns a sets of all the edited string which can be made with one simple edit (deletion, insertion transposition, and replacement) from the input word. When it comes to the language model, we estimate the probability of each word by counting the number of times a word appears in a million words level text file. Lastly, we implement our error model based on a tricky assumption which claims that all the candidate words of edit distance one are more probable than the candidate words of edit distance two. Some simple testing examples shows that our spelling correction program achieves 90 % accuracy at a processing speed of at least 10 words per second which could perform well and show real time feedback in our website.

7.2. Query Completion

Query Completion, which is also known as Query Auto-Completion, is widely used in today's search engine (Kim, 2019). We find that this function could save users much time (Kim, 2019). The goal of our query completion is to train a model which takes users current input (usually two or three words) and returns a longer sentence based on the input. The long sentences might match users' search expect. We use several natural language generating models to do this task. We first implement three Query Completion models using a tri-gram language model, a Long Short-Term Memory (LSTM) model (Gers et al., 2000), and a fine-tuning model based on pre-trained model GPT2 (Radford et al., 2019) separately. The first two models are trained on a subset of our dataset because of the limitation of time and computing power. The pre-trained GPT2 is trained on a much larger corpus, but we also use the same subset of our dataset to fine-tune the model. Then we compare the quality of output sequences and the feedback time of each model. The result shows that fine-tuned GPT-2 achieves the best performance. After discussion, we decide to use the fine-tuned GPT-2 model in our website. Although we could use beam search or other methods to generate more recommendations, we only present one recommendation for query completion for users now to reduce the time delay in the website. What is more, to make the completed query easier to search, the recommendation query is preprocessed before showing in the website. The example is shown in Fig 3

8. Additional Features

8.1. Database Update

The value of information is partly rejected with its timeliness, the updating of data is very important for search engines, and this is even more so for news data. We used an existing crawler tool, *news-please*(Hamborg et al., 2017), to extract the news information from the site. This tool offers very powerful news data collection capabilities for websites, but for reasons of flexibility we have only used it here to extract information from a specific page containing news information. Using the *BeautifulSoup* library in python, we constructed methods to obtain the URL contained in a specific section page of *BBC News*. The URL containing specific news information were identified by filtering the keywords in all the URLs obtained, and a list of URLs was generated. Then using news-please, we extracted the news information from the URLs we had fetched. For the information collected, we extracted only five attributes that were consistent with the information in the database.

It was observed that for each category of news information in BBC News, a page of the website could be refreshed in about three or four hours, i.e. each one contains roughly three or four hours of news information in the past. Based on this, we set the data collection and database update to be done every three hours. By comparing the time of publication of the news contained in the data with the time three hours before the current one, we only use news published during this period. This also works well to de-duplicate news information, preventing the same news item from appearing twice or more in the database.

Once we have all the news information for the three hour period, we build the inverted index from this data in the same way as the base data. As mentioned above, we can update the newly generated index directly into the general index table of the database without having to locate the corresponding token, as we use multiple entries to store the same token's inverted index. We also built a way to interact with MongoDB to insert updated news information and the corresponding index in the corresponding table of the database.

The regular database updates are automated by calling the system's time, which is executed every three hours. During each execution, the system automatically collects the news information from the BBC News website, filters (de-duplicates) the time of publication, extracts the attributes, generates the inverted index and updates the data in the database sequentially.

8.2. Time Limit

Compared with inaccurate sorting, long waiting time of a long query will be more irritating. Therefore, we will consider the time an entire query process has taken and skip some steps of calculating TF-IDF and show a less than perfect search results on the website. For example, if we only complete the step of calculating TF but the user's waiting time exceeds our preset time limit. We will sort the search results based TF, which means the News that contain more keywords will have a higher score and will be ranked higher on the page.

8.3. LRU Cache

For news search, it has timeliness, which means users are more likely to search for the latest hot news. The latest hot news usually has the characteristics of small batch and high repetition. Therefore, our search system may process same queries during a short period and these queries could update as time goes by.

Based on the feature of news, we will save results in cache after searching. By saving the cache in memory or on disk, we can make search engines display results more quickly when searching for cached results. However, cache is not infinite and we need to update cache constantly. LRU is a cache replacement algorithm. When cache is full, a new search result will replace a least recently used one in cache. Using LRU algorithm, our system can continuously update the cache according to the user's high-frequency search query, so as to realize the rapid display of high-frequency hot news.

8.4. Multi-threading

In fact, python interpreter only works on one thread, so using multi-threading in Python could not get as much benefit as in other programming language. However, it was still worth to use multi-threading in our project to solve some problems and optimize searching.

As introduced before, LRU cache will save all search results while Section 8.2 introduced our search results might not be ranked based on BM25 if search time exceeded the limit, which means some results ranked by TF or even without ranking will be saved in cache. To solve this problem, we applied multi-threading in the search module. A child thread will be created for calculating BM25 score, if the time was out of range, an intermediate value will be put in the queue. This thread will go on to calculate the final BM25 score and save it in cache. By doing this, in the first search, maybe the search results were not perfect ranked by BM25, but well ranked results could be found in cache in the second search. In Addition to using multi-threading to solve the problem, it can also find results in cache and find results by searching at the same time. Once found from the cache, our website could show the results immediately, while the searching process was still running in the background, which could save a lot of searching time.

9. API

9.1. Front-back end data transfer

We use JQuery to make query and fetch data from the server, the primary data format is JSON, as we applied in search result, quick spell-check and sentence filling. Such information always incorporates a large amount of text and a hierarchical architecture such as different categories of news and news of different date, JSON can support such data very well. The query itself is a plane text, it is transferred in URL format. We use POST method to pose query and get search result. we use GET method to fetch new HTML pages and local files.

9.2. GUI

Our GUI is implemented on a web interface. We apply Semantic UI API which integrates clear and beautiful GUI modules that is adaptive for both desktop and mobile platforms. It support modules such like real-time search response fields, different layout container and variety views to display text information. To keep our UI concise and clear, we set one big search box at the main search page, the search box also support real-time search recommending (implemented by Semantic search API, JQuery and JinJa language). This feature requires to detect user's input to the search box constantly. In order to not overwhelm the server by every letter user type in, the data fetch delay is set to 700ms after user press the last key. Such time delay will make sure the query only sent when user type complete words.

The result page apply Card view of Semantic UI, which organize four search result boxes in one row. There will be total 12 results (3 rows) displayed at one time in the sequence of relevance. If the user requires more result, there is also a show more button at the bottom which loads twelve more result to the page. On the top of the result page, there will display the total amount of found records. The Result page also has a search box for performing next search. The example is shown in Fig 4

10. Evaluation and Future Work

10.1. Evaluation

Flask is a good and helpful microframework, very suitable for those who need to develop a website with few pages. The spelling correction costs less than 0.1s per token, while the average operating time of query completion is around 1s.

The size of our dataset is about 8.8G, and it contains many attributions such as author, title, publishing date, which are rich and accurate. If users want to search something in certain areas. They will be satisfactory about the results. Besides, the speed of interacting with the back-end is fast. After testing, this process will cost about 0.5s or less in both of the news information and inverted index.

After many optimizations of searching algorithm and database. The average searching time can be limited around one second. Three main searching methods could meet a variety of search needs. BM25 algorithm made it easier for users to find the results they want. Some addition features, such as time limit, cache, threading made our system more flexible in the face of long queries.

At last, the top 12 related news will be shown on the web and all keywords in the query will be highlight. Our website only show a snippet of news, but users can get access to the original website by clicking the title of the news. Highlight words can help users to find what they want easily. What is more, although our search is based on queries after stemming, all words with the same stem as the keyword will be highlighted, which could provide a better user experience.

10.2. Future work

We can try to apply RESTFul style on our interface design, in our current project, the way browser access server's resources is in a chaos, very unfavorable for subsequent development. As for the query completion, both beam search and topK sample could provide more recommendations, we remain this feature for the future development. Although query completion have already done the initial work about recommendation, we will apply search recommendations to make our system more interactive,

Our data still has its limitations. Now we only have one dataset and update our data via BBC website. So we only have two news sources. Considering the diversity of news information, expanding the sources of news information is an effective way to greatly improve the usability of this search engine in the future. Also, the *kaggle* dataset is missing some news information for the last two years, and the regular update feature we have implemented is only available for current and future data. Due to timing issues, we have not yet found suitable data to fill this missing section. It is hoped that a more complete news dataset can be collected in the future to supplement the data. Besides, if word appears in many documents, the speed of searching is little slow but it does not affect the overall performance.

Our query parser can be more humanize. It is possible to determine which search method will be used based on the search results or based on NLP. For example, if phrase search can not get many search results, we will try to use Proximity search or even insert "OR" in the query to find more results in our dataset when the user do not input any flags of proximity search or Boolean search.

11. Individual Contributions

We six are mainly divided into three sub-groups, which are responsible for the front-end, back-end and database respectively. Code of our project does include much about interaction with database, but database sub-group is still very important because they developed the preprocessing, inverted index, and building database. The back-end sub-group took responsibility for text analysis and connecting database and front-end. The division of work is not absolute, and there was also assistance between sub-groups.

s2053694 - Zhiyuan He

I was in the back-end sub-group and developed query preprocessing, searching and ranking. The main task of back-end was to connect front-end and database. Therefore, my task is to parse the queries input from users, preproprocess it to find keywords for searching, interact with the database module (implemented by database sub-group) to get some related data in the inverted index, implement TFIDF and BM25 to determine which rank algorithm was better in our system to find the most related results and transfer results to the front end. In these process, I have implemented query analysis, Boolean search, Phrase search, Proximity search, BM25 and TFIDF rank, LRU cache, multi-thread and search result process(find which snippet of news to extract and the locations of all keywords which should be highlight in the snippet).

At the beginning, I need to implement a basic project, which includes query preprocess, three main search methods and rank. Since my CW1 code is already able to implement long phrase search, this part was not much work for me, and the only thing I need to be careful is using API from MongoDB module as less as possible because of the time limitation. To show our search results in the website, I need to extract a short piece of news from the full text and find all words that need to be highlight. This was not a simple work because I need to match all words that were in the same stem as the keywords and consider many special situations.

Considering the features of news, I think cache was worth to be applied in our search system, although our system was fast enough. I applied LRU cache algorithm to keep all search result in cache (In RAM or disk) and the least recently used cache will be deleted if the cache was full. In case of long query cost a long time to search, if the time cost was out of range, Ranking will be based on some intermediate value such as TF instead of BM25 score. To solve the problem caused by cache and incomplete BM25 rank process, I implemented multi-threading to make sure users could get results as soon as possible and cache could be updated in every query just like introduced in Section 8.4.

In addition to my personal work, I was also involved in some work that intersects with other group members such as finding a suitable dataset, trying to solve memory full because of the large size of the dataset. As I have to test various query to analyze my search logic and output results, I often found some errors in other modules. For example, I found the preprocessing of our dataset did not remove all stopping words because of removing stopping wording without lowercase in advance.

s2176866 - Runze Xia

I'm responsible for constructing a front-end GUI for users to perform the search task and check the result. Traditionly, I tend to apply the spring framework and use Java as the back-end language. This time, as we discussed, we applied a light weighted FLASK framework based on python. I learned it from scratch and quickly managed to find its subtle mechanism. The back-end coding is more compact and concise, and the front-end also involve a new language called JINJA 2. I struggled with grammar at first, but through some document reading and example walk-through, I find it pretty helpful and powerful for logic expressions, as I use it to parse the result heretical JSON data.

The GUI layout should be concise and clear; I tried to make the search area significant and big enough to accommodate big queries on the search page. The search page also enables a query auto-complete function, which detects the user's input in real-time and sends it back to the back-end for spell correction and sentence completion. The traditional way might be using input and form elements, but I applied semantic UI in this project, whose documents only mention a few lines about the usage. I spent much time on testing and parameter tuning. I used a Card-view to display the search result for the result page. Which makes the result organised into different boxes with the key-word highlighted in red.

Many difficulties are encountered in constructing the two pages, such as figuring out the usage of response fields API of semantic UI dealing with the different encoding formats with HTML and UTF-8, which makes matching key-words in the result page difficult. However, with the help of active discussions with group members and constant meetings, I manage to overcome all these difficulties and continue to implement new features.

s2041707 - Yu Jiang

I am a member of database sub-group. At first, I collected datasets about various topics because we're not quite sure what we're going to do, and I also need to determine what database management system we will use to store our data. My major job is to study Mysql which is familiar to most of us. But after taking into consideration, Yujia Zhang and I thought that Mongodb is an optimal choice. Because the most important part in our design, index, could be easily stored in the software. And then we cooperate with the back-end sub-group teammates. They produce the index and our data sub-group wrote codes to upload them into Mongodb. Mongodb has cloud edition. So we first choose to upload data in the cloud edition. In that way, the back-end sub-group could debug to optimise their programming. I also cooperate with Yujia Zhang to finish some API, and then the back-end sub-group could get data what the front-end sub-group want.

After the initial build-up of our search engine, Zhiyuan He and I tesetd the performance of our engine. We wrote logging to find the time every part consumed. After discussion, We thought we should use local database so the sever could access the database faster.

In the end, we find our dataset does not meet the assignment requirements, so I moved to find dataset again. Because the data is so large so we decide to process the original data in my computer. But there is a new problem when I run a python program to pre-process the data. My memory had no ability to write a so large index, which wasted me a lot of time. We had tried various method. In the end, we use google cloud to solve it.

s2120856 - Guipeng Hu

I am one of the back-end sub-group. I developed dataset preprocessing, inverted index, and all the features which could be treated as natural language processing tasks. My main job is to decide which natural language processing features should be included on our website and chose suitable models to initialize these features. With the help of the discussion with my teammates, I found that spelling

correction and query completion were two features which were most worth to be added on our website. Other features, such as query explanation, were able to applied, but were not as practical as the former two for our system. Besides, I also helped with other groups to write codes when they needed a hand. For example, I helped database group to preprocess the whole dataset.
At the beginning, I tried to train a neural network based on language model to correct each input word. It turned out that both the training time and running time were not ideal. In this case, I used a simple language model with some other components which were introduced clearly in Section 7 to complete real-time spelling correction. After the test, the model worked pretty well. As for the query completion, I trained three natural language generating models and tested them in the same test set. My teammates and some of my friends who had linguistic background helped me evaluate the output. The model which we decided to use on our website was a fine-tuned GPT2(Radford et al., 2019). I trained the model in a single GPU in Google Cloud. After loading the pre-trained GPT2 parameters, I used those parameters and a subset of our dataset to start my training. To save training time and avoid overfitting, I only trained the model with 5 epochs. Then I developed a function based on pytorch to save the model after each epoch. The whole process cost 5 hours. After I got the model, I wrote the code which used pytorch to load the saved model to complete the users' input query.

s2184385 - Yujia Zhang

I am a member of database sub-group. I was involved in the selection and deployment of the database, the collection and pre-processing of the dataset, the implementation of the database and back-end interaction methods, and the implementation of the regular data update function. I started by comparing the performance and usage scenarios of Mongo database and MySQL with Jiang Yu, and in this part I was mainly responsible for testing the effectiveness of Mongo database. For the final dataset, I built a pre-processing method to filter the complex and existing data. In terms of interaction with the backend, I wrote part of the interface and worked with the backend to debug and modify it. For the regular data update, I wrote code to obtain valuable URLs from specific news sites and worked with existing crawler tools to capture news information at specific times. At the same time, I wrote the corresponding database interface to update the data.

At first, I collected many types of datasets and discussed with the group which ones could be used for the construction of the search engine and eventually chose the dataset for the lyrics. However, because I had neglected the size of the dataset required for the coursework, it was necessary to change the dataset after we had already successfully built the overall framework. We therefore replaced the dataset with news information. It's worth noting that because our own computer had difficulty in generating the inverted index in the beginning, I sliced it into six separate files during the pre-processing phase. However, this resulted in an increase in the number of index entries corresponding to a token in the index, which did not make full use of the space. In the tests, we found that this resulted in a huge time cost in the database search phase. Together with my group member Yu Jiang, I reorganised the storage structure of the inverted index in the database to achieve a situation where we could avoid traversing the entire database when searching. This code also ended up being useful in the function that updates the inverted index when the data is updated periodically. Although the processing of the dataset was not complex, it was very tedious and the constant testing and improvement of the solution has improved all aspects of my skills immensely.

s2238615 - Yifan Ye

I initially set up the project structure on Github and installed Flask so that the connection of front-end and back- end is established before we started working on the project. I joined the discussion about our data source and I chose Flask for it is a convenient microframework that very suitable for out project, easy to extend, no pre-installed modules, satisfies all our need. And I involved to some frontend work and backend work for I am responsible for architecture work.

Interfaces design is also a part of my work, interfaces between frontend and backend are well designed, which is very helpful to our development work. I also tried to follow some software engineering development philosophy, like agile development, I combined our CW1 code and simple UI to a basic project and deployed it on server, and I keep it browsable during developing. We have meeting every week to response fast to others' work and thinking.

References

Gers, Felix A, Schmidhuber, Jürgen, and Cummins, Fred. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

Hamborg, Felix, Meuschke, Norman, Breitinger, Corinna, and Gipp, Bela. news-please: A generic news crawler and extractor. In *Proceedings of the 15th International Symposium of Information Science*, pp. 218–223, March 2017. doi: 10.5281/zenodo.4120316.

Kim, Gyuwan. Subword language model for query autocompletion. *arXiv preprint arXiv:1909.00599*, 2019.

Radford, Alec, Wu, Jeffrey, Child, Rewon, Luan, David, Amodei, Dario, Sutskever, Ilya, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8): 9, 2019.

Whitelaw, Casey, Hutchinson, Ben, Chung, Grace Y, and Ellis, Gerard. Using the web for language independent spellchecking and autocorrection. 2009.

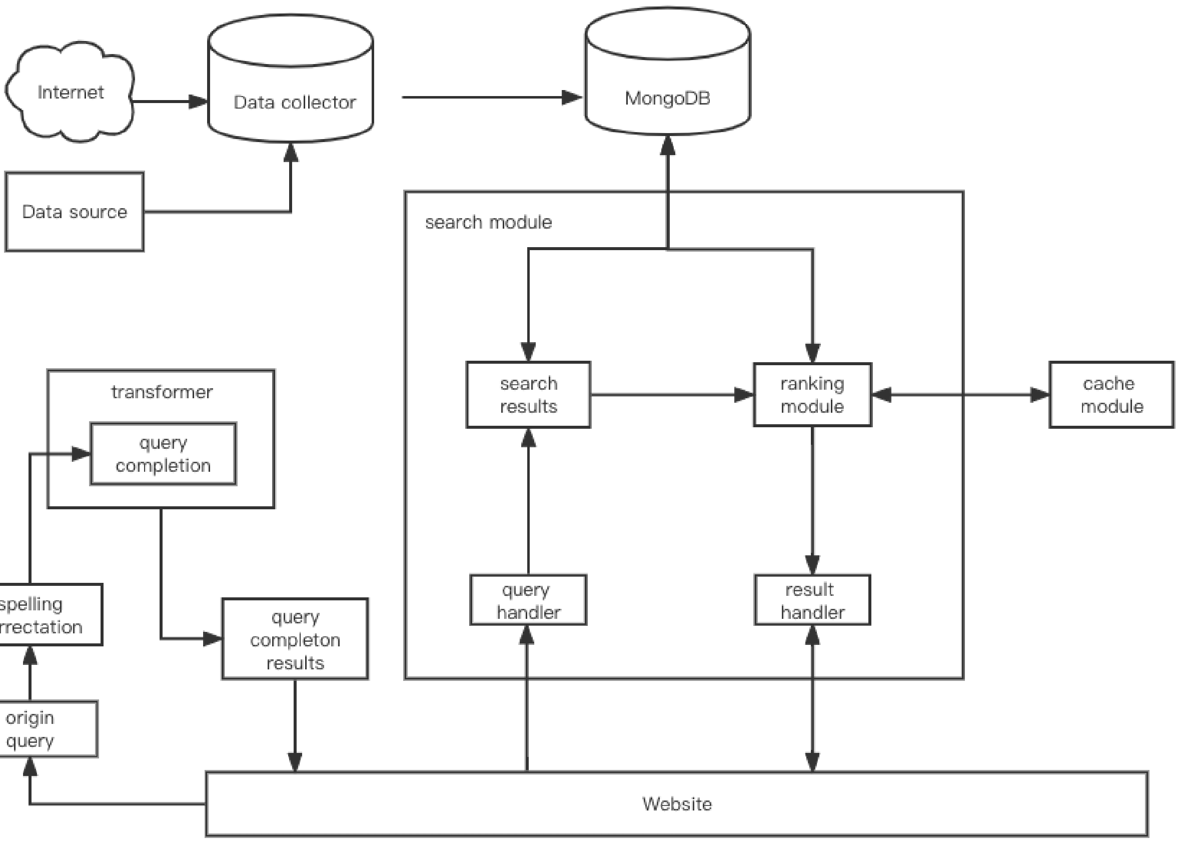
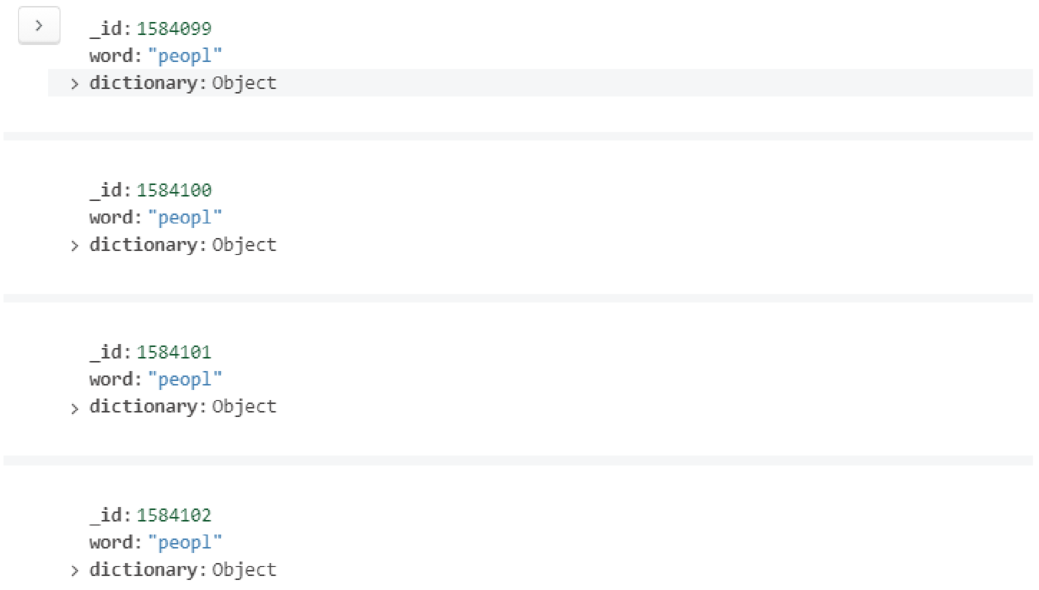


Figure 1. Architecture of the project



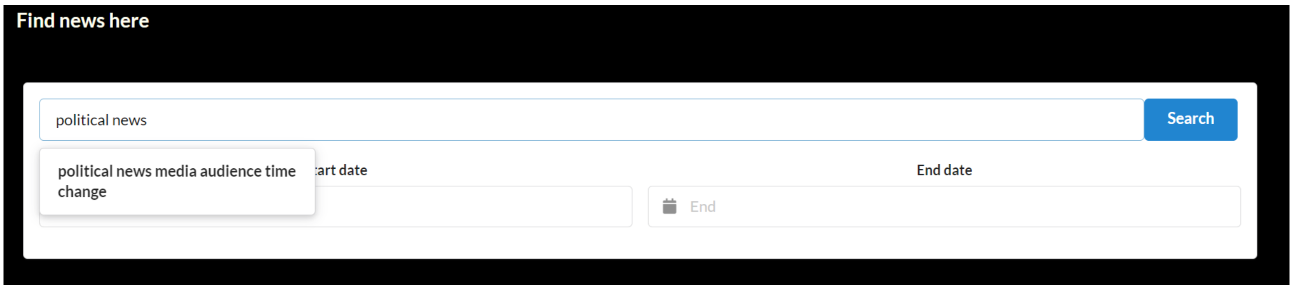
```
_id: 1584099
word: "peopl"
> dictionary: Object

_id: 1584100
word: "peopl"
> dictionary: Object

_id: 1584101
word: "peopl"
> dictionary: Object

_id: 1584102
word: "peopl"
> dictionary: Object
```

Figure 2. Example of index structure

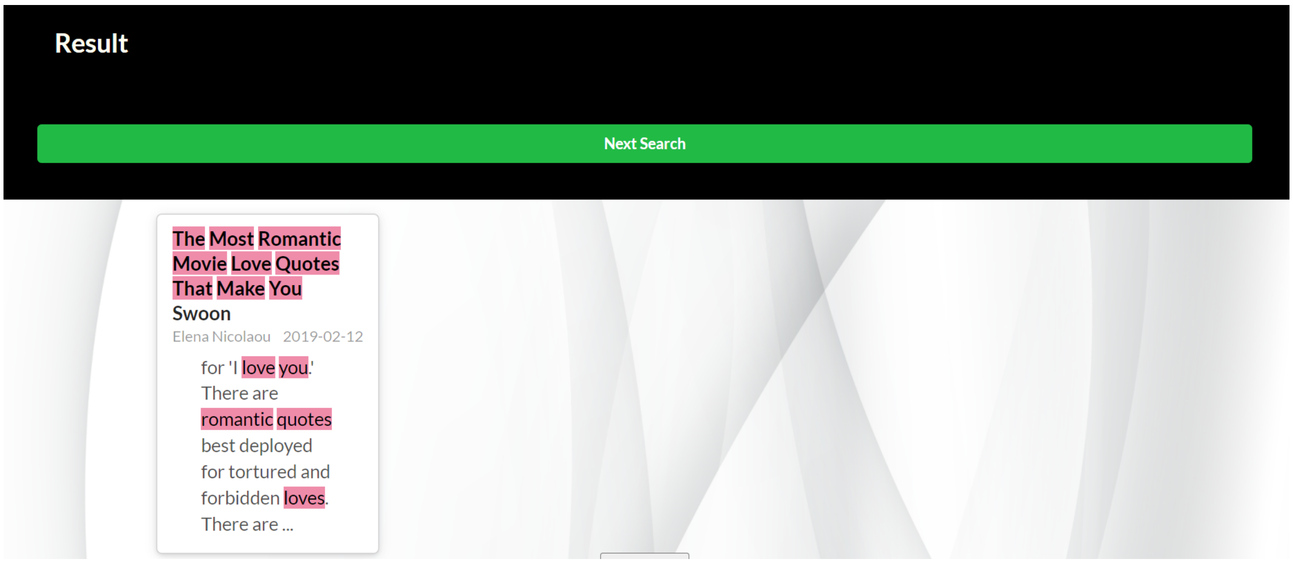


Find news here

political news

political news media audience time change

Figure 3. Example of query suggestion



Result

The Most Romantic Movie Love Quotes That Make You Swoon

Elena Nicolaou 2019-02-12

for 'I love you.' There are romantic quotes best deployed for tortured and forbidden loves. There are ...

Figure 4. Example of result in search engine