Title: Group assignment: Shakespeare portfolio search engine

G017 (s2020314,s2042691,s1985393,s2054175)

Abstract

The Shakespeare quotes search engine is a website that can retrieve complete content of quotes and its related detailed information — the work information where the quote is from, the line information e.g. the character who says the quote and the position where the quote emerges. There are three main different search types, which are (1) Exact Ranking Search using BM25 in support of both AND and OR searching, (2) Regular Expression (RE) Search for matching words and an OR search for them using TFIDF, (3) Phonetics Search for words that share similar phonetic patterns. We collected almost all Shakespeare's works known, including 73,410 sentences and detailed information related. The inverted index was established and the BM25 and TFIDF retrieval models are used for ranking. Additionally, queries the user entered is expanded using a LSTM model, which learned how Shakespeare writes and provides predictions to offer user hints. The website is accessed through this link http://165.22.177.130:3000/.

1. Introduction

Almost all of us have heard of "To be or not to be — that is the question", one of the most famous Shakespeare quotes known all over the world. William Shakespeare was an English Renaissance playwright and poet. The works of Shakespeare have been popular over the past 400 years. Many people are still fascinated by the philosophical and beautiful sentences written in Shakespeare's. People want to quote some sentences they like and appreciate in Shakespeare's works on some occasions. However, it's not uncommon for them to know only the keywords or general content of these quotes and cannot remember the exact sentence. In this case, people have a need to search for the complete Shakespeare quote and learn more information related to this quote. This inspired us to create a Shakespeare quotes search engine.

Users can retrieve relevant quotes and detailed information by typing keywords or general content in the search box of our website. The search engine will return the complete quote, the work of this quote with an overview and the cover image, the character who spoke of this quote and the position of it in the work. By clicking on the detail button on the left side of a quote that is retrieved, the detailed information will be displayed. There are different types of searches for users to choose by their own choice: Exact Ranking Search using BM25 (AND or OR), Regular Expression Search and Phonetics Search. In particular, A LSTM model offers prediction that expands the query entered by the user to offer hints for what content could be following.

To build a search engine for Shakespeare quotes, we collected 42 works of Shakespeare in the Folger Shakespeare Library. After parsing these XML files, we extract and process the data. We consider each sentence as a document to be retrieved (73410 quotation marks in total). In order to efficiently handle the retrieval and ranking purpose, we created inverted index and other useful statistics for calculating the ranking for speeding up the actual query process. We save all data into MongoDB and benefit from the inherent feature of MongoDB storage indexes. Data are split and

saved in different collections of one database.

For exact-search on all or some keywords, the BM25 ranking algorithm is used. As for Regular Expression Search, we apply the TFIDF sorting algorithm plus weight to rank the search results. In addition, we also designed a phonetic search to return a list of words that have appeared in the Shakespeare's with similar pronunciations to the word the user types. The user can choose from the returned word list and then search through the above-mentioned searching methods.

After finishing all the basic functions of the search engine, we try to explore the advanced feature. We implement the keyword prompt feature in the search engine based on a language model. Given the sequence of characters in our dataset, we train a LSTM model to predict the next sequence of characters. And ended with a longer predictive text sequence generated.

This report is organized as follows: In section 2, we outline the system architecture of the project and the technologies we used. In section 3, we introduce in detail the acquisition, processing and storage of the data set. The section 4, we introduce the method of preprocessing quotes with some special operations. In section 5, we introduce the inverted index structure. In Section 6, we offer detailed explanations on the algorithm and implementation of three different search types and the query expansion. The design of API and Graphical User Interface (GUI) is described in the 7 and 8 sections. In the last two sections, we evaluate our search engine project, mention the possibility for improvement, and attach each member's contribution to the project.

2. System Outline

We design the overall system architecture to determine the module development we need to complete. After initially determining the system design, we also improved the design to achieve a more efficient quote search engine. The system is mainly divided into server-side and client-side. The server processes the received query and returns the result. The client includes a graphical user interface (GUI) and its connection with the server.

First, we set up a data collection module for all the data used in the project. Build the inverted index after preprocessing the data. Store the inverted index and other required data in the database. When the GUI receives the user's input query, the GUI parses the query and sends it to the server. The query handler module API processes the user's request and triggers the result search. Obtain data information from the database and use the ranking scores in the ranking retrieval module to sort the documents retrieved in the inverted index.

In the process of developing the above system, we used MongoDB as the database and applied the Python language to the back-end programs and scripts. We integrated the Flask application with the front-end framework, developed the API in the Flask framework, and the front-end in React.

We use git to do version control. We apply all ends in a Docker and manage packages using yarn. We use Digital Ocean cloud servers for hosting.

3. Data Collection and Storage

For building the search engine for Shakespeare quotes, we collected all known works of Shakespeare. The Shakespeare's work collection is downloaded from the Folger Shakespeare Library. From Shakespeare's works, we can collect every sentence in the work and all the information about this sentence, such as which work the quote is from, the quote position (line number) of the work and which character said the quote said (only in Shakespeare's plays) etc. The Shakespeare works we obtained are in XML format. After parsing the XML files, we extract and deal with the data. Finally, quotes, inverted index, and works are saved in the MongoDB database.

3.1. Quote Information Collection

From the Folger Shakespeare Library, we download files of the complete Folger Shakespeare

collection in XML format. We firstly use Beautiful Soup, a Python library, to help parser the documents. A few methods for navigation, search and modifying a parse tree provided by Beautiful Soup make it more flexible and faster to extract the data. After dealing with the original data, we regard each quote as a document. It's important to mention that in the original file, one quote could be divided into different lines. It is due the unique organization of lines in poem and plays etc. In order to not let the search result to be an incomplete line. We concatenate those lines by identifying uppercase characters and punctuation used at the end of a line (for example, a period, an exclamation mark, a question mark or no punctuation).

We get the collection of documents containing 73410 quotes from 42 Shakespeare works in TXT format this way. The details of the data contains are shown below:

lineinfo.txt — contains line information of quotes such as the complete sentence, the work where the quote is from and the character who says this quote. Specifically, characters are obtainable in Shakespeare's plays and we set the 'NOBODY' as the character in other Shakespeare's works where there is no mentioning of who speaks the quote. The "lineinfo.txt" file also includes the order of the scene, the act and the line position (a range of line numbers) of each quote.

workinfo.txt — contains information of Shakespeare's works: the title, the genre and the synopsis of the work. Particularly, Shakespeare's poems do not provide the overview, so we choose to ignore the synopsis of the poem works and assign the value to be empty. Later, to save and deal with the mentioning of work in each line more efficiently, we apply a mapping for each work from code to the title.

Since the aim is to search Shakespeare quotes, our data collection only contains one language. However, the English used in Shakespeare's time is not exactly the same as the current standard English. For example, there are some special characters in Shakespeare's works such as 'é', 'æ' and 'œ' etc. We will introduce our preprocessing methods in detail in Section 4. The spelling of a word can also be different, which is why we introduce Phonetics Search in section 6 in the first place. Right now, the raw data collection is mostly text data but we also have pictures. When we click a certain quote searched for more information, the cover of the corresponding Shakespeare's work will appear.

3.2. Database

We choose MongoDB to store our quotes, works and inverted index etc. (discussed in section 5). The B-tree ("balanced tree") index structure of MongoDB ensures effective inverted index storage and search. To benefit from the merits of MongoDB, we will have to structure the data carefully and apply index on search keys.

We use pymongo (MongoClient) to connect and insert data into MongoDB. There are seven collections created in one database. The details of each collection is available in its corresponding feature descriptions. Within this MongoDB module, it can retrieve the inverted index of a term from the quote, the complete quote by the quote ID and Shakespeare's work details based on the work ID. The quote sentences and works are respectively indexed by their IDs. Additional index are created when multiple searching requisites are present.

```
lineinfo
metadata
phobox
stem2idc
steminfo
token2info
workinfo
```

```
>db.lineinfo.findOne()
{"_id": 0,
  "display": "And put the same
     into young Arthur's hand,",
  "detail": {
     "code":"Jn",
     "actnum":"1",
```

```
      "scnum":"1",
       "speaker" : "CHAYTLLION",
       "lineRange" : "14",
         }
}
>
```

```
>db.metadata.find()
{"_id":0,"avgLen":10.94304596552241}
>
```

```
>db.workinfo.findOne()
{"_id": "R2",
  "title": "Richard II",
  "genre": "History",
 "synopsis":"In Richard II,.....
    "
}
>
```

4. Preprocessing

After parsing the data, we consider each sentence as a document.To preprocess each document, firstly, we convert all the letters to lower case and replace all punctuation with blank. Before we implement the tokenization, we need to pay attention that the English we used today is different from Shakespeare's time. That is, English pronunciation, spelling and grammar were less standard. It needs more steps to implement those early modern English works.

During the preprocessing, we notice that the text has some special characters that seem like accents, such as 'é'. Implement the unicodedata function to normalize the character. In this case, each sentence line contains only '1'-'9', 'a'-'z', blank space (separated by one space only), 'æ' and 'œ'. Then, we tokenize all the text.

As we aim to search quote in Shakespeare's work, we choose to be cautious and careful to remove stop words. To be more specific, considering almost the simplest quote "to be, or not to be", if we search for this query, we would end up with getting nothing if we are to remove stop words like 'to', 'be', etc. Thus, we choose very few words that are trivially mentioned. The stop words we remove are a, an, the, s, t, and, of, or. Especially, character s is what emerged when we remove punctuation for words like "it's".

We design three search type: the BM25 search, the regular expression search and the phonetics search. For the BM25 approach, we search all or part of the keyword user query. When applying the stemming, we use PorterStemmer from the NLTK in BM25 search. Stemming works for standardizing vocabulary and improving classification accuracy. However, in regular expression search, we aim to query several characters and find terms that contain those characters. For example, when searching "straw", the search engine will return quotes that respectively contain "strawberries", "straw" and "strawberry" etc. If we implement stemming, it will change the formation of characters we want to search for. Thus we do not apply stemming for regular expression search, but rather use tokens directly. As for the phonetics search, we choose not to do the stemming operation when preprocessing text for the same reason.

5. Indexing

Users can type query to search quotes and related more detailed information. We build an index for each line document, give each sentence an ID, and map it to the corresponding work information. In order to efficiently process our search, we use the inverted index. Build the inverted index structure as shown in the following figure. For each term, we map each term with an ID to sentences the term appears in. The frequency of the term appearing in each level is added for computing the ranking scores faster. For the same purpose, we also added the value of idf to the inverted index structure. We use the inherent characteristics of MongoDB to store inverted indexes in a B-Tree structure which helps to optimize our query and retrieval.

```
>db.steminfo.findOne()
{"_id": 0,
  "stemname": "mainten",
```

```
  "stemDocc": 3
  "stemIDF": 4.3216930956934405,
  "stemUIDC":{
       "28110":1,
       "18950":1,
       "46432::1,
  }
}
```


6. Retrieval

6.1. BM25 Search

The BM25 Search(Connelly)implement the BM25 to retrieve the query. BM25 is an approach of a bag-of-words search model. It does not consider the distance of the terms of query in the document. BM25 scores based on the terms that appear in the document and then rank these documents based on their scores. BM25 is a probabilistic model of retrieval with efficiency. The BM25 score formula is shown as bellow:

$$BM25(q_i D) = \sum IDF(q_i) \frac{f(q_i, D \cdot (k1+1)}{f(q_i, D) + k1 \cdot (1 - b + b \cdot \frac{fieldLen}{avgFieldLen})}$$

With qi being the ith query term. IDF(qi) is the inverse document frequency of the ith query term and measures the frequency of a term occurs in all of the documents. fieldLen / avgFieldLen represents the length of a document compared with average document length. Parameter b measures the importance of fieldLen / avgFieldLen and b has a value of 0.75 in our experiment. f(qi,D) represents the frequency of ith query term occur in document D. The parameter k1 limits the degree to which a single query term affects the score of a given document. In our experiment, the default k1 value is 1.2. In order to improve the efficiency of retrieval, we add the frequency and idf value of the term in the document to the term inverted index. At the same time, in the index structure of the quote, the length of the quote is added.

In BM25 search, we implement "and" and "or" operations respectively. BM25And search retrieves all terms entered in the query, while BM25Or searches and retrieves some terms in the query. The query typing of BM25 search is a free-text query and users can enter any content they want to search. There is no limit on the length of the query, but we recommend that users do not type a long query under the BM25Or search function. Because BM25Or implements "or" logic, the retrieval time will greatly increase when the length of the input query is too long.

6.2. Regular Expression Search

Regular Expression Search is used to check if a string contains the specified search pattern. A RE is a sequence of characters that forms a search pattern. In RE search, we set the limitation before entering the real search process. The user is suggested to type only one term which contains a sequence of characters. If the user searches for more than one term, a RE will regard all the content typed as a phrase and retrieve them as a whole. That is, a RE will end with no result found when searching for more than one term.

The idea of implementing the RE search is to expand the query entered first and find a list of words containing the typed string content in Shakespeare's works. We expand the query for search from a characters string to a list of words, and then retrieve the expanded query. We use the findall() function in the RE library to complete word recognition with the same string pattern and thereby completing the expansion of the typing content.

Our search task is converted into retrieving a series of words in the expanded query and retrieve quotes that include any word in the expanded query word list. When retrieving the extended query, we use the term frequency-inverse document frequency (TFIDF) ranking algorithm to score. TFIDF is suitable for evaluating the importance of keywords in the document and is easy to operate calculations. The TFIDF score formula is shown in equation:

$$Idf(t) = log_{10}\frac{N}{df(t)}$$

$$Wt.d = (1 + log_{10}tf(t,d)) * idf(t)$$

IDF(t) is the inverse document frequency of term t, which measures the importance of the term. Wt.d is the weight combining TF and IDF (Wt,d) of the term t. Tf(t,d) is the number of term t appeared in document d, df(t) represents the number of documents term t appeared in, N is the overall number of documents. To facilitate calculation, we add the frequency of the term in the document and the value of idf to the inverted index of the term. For the retrieval of the extended query, each term in the extended query obey "or" operation, so the score of document d on the query is the sum of all wt,d. Finally, it returns the search result according to the ranking of all the document scores.

In particular, when the number of characters entered by the user is very small, a large number of words will be expanded in the expanded query. In this case, a lot of quotes will be retrieved. To avoid that, we do not search and display the list of quotes results but remind the user to select words displayed on the left side of the web for entering a more accurate query.

```
>db.token2info.findOne()
{"_id": 0,
  "tokenname": "enter",
  "idf": 1.6981425513439212,
  "entriesType":[
             1
             ,2],
    "entries":[
    {"entrycount":1
    "entryset":[
    4576],[57959],...
    }
    ]
}
```

6.3. Phonetics Search

As we mentioned before, the English we used today is different from Shakespeare's time. However, even if the English spelling and pronunciation in Shakespeare's time were less standard, the spelling and pronunciation of the words do share some similarities compared with what we use today.

A phonetics algorithm creates a specific phonetic representation of a single word. Metaphone(Wikipedia) is a phonetic algorithm that uses information about changes and inconsistencies in English spelling and pronunciation to produce more accurate encodings. Although there are some small differences in spelling and pronunciation of some words in Shakespeare's works, users can search for the target sentence on our search engine through the realization of the Metaphone algorithm. In addition, the phonetic algorithm can also help to correct the incorrect spelling of the entry, in case that the user enters the wrong word but does not change the pronunciation greatly.

We use the Double Metaphone function in the Phonetic software package to encode the Shakespeare text lexicon and the content typed by the user. Then we find the text words output with the same encoding. In order to improve the search efficiency, the newly encoded Shakespeare text lexicon is stored in the MongoDB database. In addition to that, we make the limitation of the query entered in the Phonetic search. In the phonetic search, the user is asked to enter only one keyword, and then words with similar pronunciation in Shakespeare's work will be retrieved to the left side of the web page. Users are able to select words that match their search objectives among these retrieved words to retrieve famous quotes and sentences in the BM25 search.

```
>db.phobox.findOne()
{"_id":ObjectID("6061e1ecac957a2
   546882074"),
  "Sound": "KNSNT",
  "tokenbox": [
            "cosent",
```

```
            "cosigned"
    ]
}
>
```

6.4. Query Prompting

In our commonly-used search engines like Google, Bing and etc, when we type a query, it will fill in some popular keywords in the drop-down box. Displaying the search keyword prompt function can save the user's search time to a certain extent. At the same time, the expansion and prompting of keywords can help users search more efficiently and find more suitable search content.

Long short term memory (LSTM)(TensorFlow) is a recurrent neural network. Our idea is to train a character-based LSTM model to generate text. Given a sequence of characters in our data set, the model is trained to predict the next character in the sequence. By repeatedly calling the model, a longer sequence of predictive text can be generated. We use TensorFlow and Keras to implement the LSTM model in Python. In this case, as the user types characters in the search engine, the drop-down box of our search box will perform real-time text prediction based on Shakespeare's works and the content entered. This can give our users a hint and help them search for suitable content more efficiently.

BM25 And search using the query "death and love". Return a list of quotes with details.Shown in figure 1.

BM25 OR search by typing "death love dream". Shown in figure2 that the difference between And and Or search is obvious. That is, the BM25 And search all the keywords and when enter "death love dream" no results return BM25 And returns. Moreover, detailed information shown after clicking the detail on the left side of a quote.

Type very few number of characters in RE search will retrieve many quotes. Then the user should choose words returns on the left side of the web and perform a more precise search. Shown in figure 3, typing the word "straw" in RegularEx search would return a list of quotes with details and the related word list shown in figure 4.

In Phonetics search, we discover words that have similar pronunciation with "throw". It returns a list of words. Query completion works at the same time. Shown in figure 5.

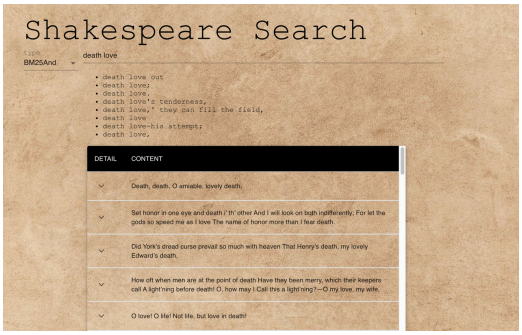

Figure 1. Result of query "death and love"(BM25 And)

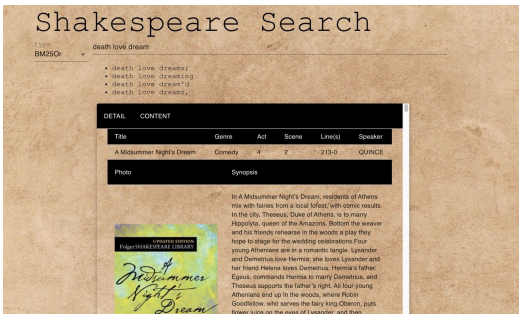

Figure 2. Result of query "death and love"(BM25 Or)

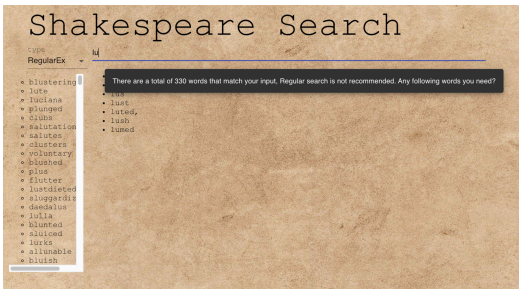

Figure 3. retrieve quotes

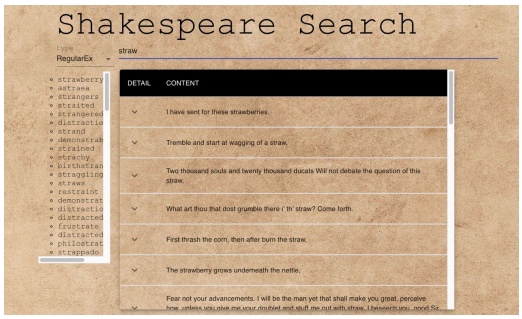

Figure 4. "straw" RegularEx search

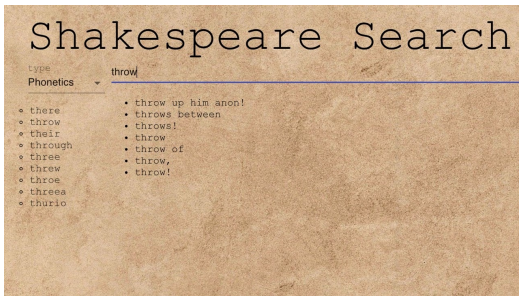

Figure 5. Phonetics search

7. API

The API is the middleware between the front end and the back end, built using a popular Python Web framework, Flask, which allows you to create endpoints for a GUI to retrieve data. The main roles of the API in this project are as follows. First, select the search type from the selection box and perform different search functions according to the search type. BM25 AND, BM25 OR and so on. Second, accept the query in the query input box, preprocess the query according to the corresponding way according to the instructions of the search type, and then search according to the corresponding search function. Third, after completing the above two steps, the search results will be extracted from the database to get the complete results. Fourth, return the generated result as JSON. For processing and display on the front end. We use axios to handle requests. Although docker is used thus all members can run application on their laptop, we still use CORS to allow API accessing from the server.

These are the API exposed and accessible from the server. All parameters can be changed to support different needs. Words like 'love' appears in a large amount of documents, which takes longer time. (1) BM25 AND 'power love' (2) BM25 OR 'power love' (3)RE 'love' return word list (4)RE 'dkill' return query result (5)query prompt 'po' (6)phonetics search 'power'.

In summary, the function of the API in this project is to read the search type and query from the front end, pass parameters back to the back end, and retrieve results accordingly according to the functions on the back end. The result is eventually transmitted back to the front end. The JSON data sent back to the front end is processed by the front end and displayed in the form of a table.

8. GUI

In the front-end design of our team, we concentrated the whole search system on one interface. This page mainly contains three parts: type selection box, input box of Query, and display box of search results as is shown in former pictures. We also added some additional features, such as when the user is typing the query in the input box, the searching results of RE etc. are listed in a similar way to the query prompt. The search results interface is in a scrollable manner, which keeps the whole interface clear and clean.

8.1. The main tools and libraries used

In this project, we mainly used React to construct the front-end interface. At the same time, we use Docker to create a virtual environment, so that we can display the effect in real time when constructing the front-end interface. On the component side of the front end interface, we mainly used the Material-UI. Because the UI library is relatively concise and easy to use. The input fields, selectors, and so on of our interface will be based on this UI library.

8.2. Search type selection box

For the design of the Query selection box, we use the Select component in the Material UI. When a user searches for Shakespeare, he should first select the Query type. Such as BM25AND, BM25OR, etc. So we put the marquee before

the Query input. When the user completes the selection, we pass the selection result to the back end for processing.

8.3. The query input box

For the design of Query input box, we use the TextField component from Material UI. At the same time, we set a function in the search box to match the input content with the query according to the input content and the selected retrieval type. When the user enters some Queries, the system can match them according to the contents, and display the similar Queries directly below, and the user can directly select the corresponding Queries. When the user enters the complete Query and presses Enter, the data in the Query is passed to the back end for processing and retrieval

8.4. Result output interface

For this interface, we set it up as a table, using the Table component in the Material UI. We also set it up to be a paging table. Each page displays 10 results, which can be paged through the buttons in the next section to display more search results. The search results displayed are retrieved by the search system at the back end according to Query, transmitted to the front end and displayed on the table.

9. Evaluation and Future Work

We will make a systematic evaluation from the following aspects. First of all, at the user level, our search interface is a clean interface. When users use our search engine, our design allows them to quickly learn the basics. For example, when the user does not enter query, the search box will display a prompt for use. In addition, we have a Phonetics option in the search box, which can help users who are not familiar with Middle English to search. However, there is still room for improvement at the user level. For example, search instructions can only be implemented by the Enter key. We can add a search button to the right of the search box, which can give the user more choices of operations. Second, in the response level of search, the data returned by search is complete and accurate at present. However, under some queries, when we return too much data, the response time will be too long. Therefore, in the future, we will optimize the search engine of this project and adopt a more advanced and fast search algorithm. In other respects, the project has added some practical small functions. For example, for words or phrases entered, the system will automatically think of a close query for the user to select. This is user-friendly. Of course, the best way to do evaluation is to test with a large number of users, but due to limited conditions, we can only do this through the team members' own experience.

For the future optimization direction, we mainly put forward some ideas according to the content of the evaluation. Some necessary features will be added. For example, we can add more options in filtering the wanted results such as which play is the quote in. The connection between different lines in the same work is not clear. One user may be interested in a consecutive lines of work but we can not offer that yet. The content interface of the search results will also be further optimized. At present, the detailed content interface of the works only has the cover and synopsis. We will try to design a new interface to place the specific content and chapters of Shakespeare's works. This will greatly improve the usability of the search engine. At the same time, we will also explore the search algorithm to improve the speed of response.

10. Contribution

- QiZhou (s2020314)

  I identified project targets and jobs for team members and started off the project. I created a fully-connected skeleton project on Docker (docker-compose), which consists of the front-end (React) and back-end (Flask, mongoDB) and is easily reproducible. I found collections of Shakespeare's (in xml). I extracted lines/information (mainly use beautifulsoup). I preprocessed raw data (con-

catenating complete lines that are seperated etc.) in preparation for further processing. I made BM25 query search and used LSTM to train a naive prediction model for query suggestion. After receiving teammate's work on regular expression and phonetics search, I further processed all data and transformed them into reasonable structures. I managed mongodb and used pymongo to save all data. I managed the back-end using Flask, transferring all query logic, connecting with mongo, front-end and prediction model. I cooperated with my teammates on front-end designing and coding. Lastly, I deployed the whole project onto the server.

- ZhenhangWang(s2042691)

  My main work focuses on the front end. Since I had taken courses related to human-computer interaction, I was involved in the design of the front-end interface. It took a lot of discussion to go from the initial three interfaces to one. After the design of the interface was decided and the team members completed the layout of the interface, I began to transfer the back-end data to the front-end. I pass the front-end query into the back end, extract the retrieval results from the database, send them back to the front end, and display them on a table. After finishing the above work, I continued to carry out some extension work on the basis of the original GUI. First of all, I completed the function of automatic association of query in the search box according to the input content. Secondly, I have completed the system reminder information, such as the system will prompt the user to wait for more while searching, and the feedback information when the search results are not found.

- WanchenChen(s1985393)

  Before the project officially started, I discussed the goals and design of the project with my teammates. I also took part in the discussion about how to deal with Shakespeare's works data. I mainly completed the design and implementation of regular expression search and phonetic search. I chose not to use stop words for these two search functions, so I re-preprocessed the data and built an inverted index. In the regular expression search, use regular expressions to find words with the same characters and expand the query. Retrieve the expanded query and use the TFIDF score for ranking. For phonetic retrieval, I used the Double Metaphone function in the Phonetic software package to complete the recognition and search of words with similar pronunciation. Finally, based on the back-end work of the project, I compiled the first draft version of the first to sixth sections of the report.

- ChengkangLou(s2054175)

  My main work focused on the front end, the GUI part. At the beginning of the project, I did some data set searching .In the front end work, I completed the design of the front end interface. At the beginning of the project, I designed three interfaces for the whole project, including the search page, the results page and the work content page. But after some discussion, we finally decided to concentrate the entire GUI into one interface. In addition, I participated in the production of GUI interface, including the layout of front-end components, formatting of some components and so on. At the same time, after the completion of the front end, I tested the functions of the front end, such as the function of nearby association according to the input query and the page-turning . Finally, I also wrote the report of the frontend(Section 789),and I integrated the entire report, rendered it into LaTeX, and finished editing the report.

References

Connelly, Shane. Practical bm25 - part 2: The bm25 algorithm and its vari-

ables. https://www.elastic.co/cn/blog/practical-bm25-part-2-the-bm25-algorithm-and-its-variables.

TensorFlow. Rnn. https://www.tensorflow.org/tutorials/text/text_generation.

Wikipedia. Metaphone. https://en.wikipedia.org/wiki/Metaphone.