



Queensland University  
of Technology

IFN647 Advanced

Assignment Report – Sem2 2019

# Assignment 2

## EduQuiz Question Answering System

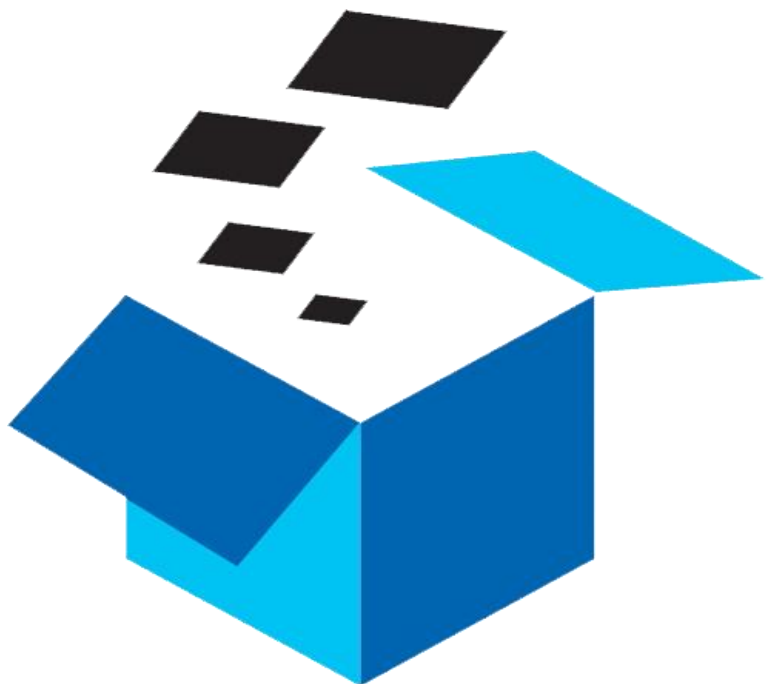
Jianwei Tang - N10057862

Yongrui Pan - N10296255

Shengguang Han - N10056084

(Jarvis) Zhe Han - N10153853

29<sup>th</sup> Oct 2019



# Contents

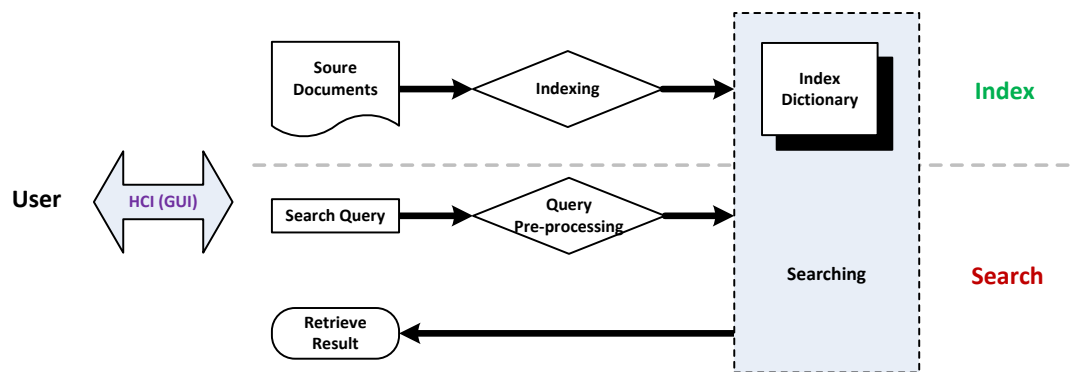
1 Statement of Completeness and Work Distribution .....	2
2 Design.....	2
2.1 Index.....	3
2.1.1 Indexing strategy .....	3
2.1.2 Collecting source documents.....	4
2.1.3 Handling source error .....	4
2.2 Search.....	5
2.2.1 Matching strategy .....	5
2.2.2 Query pre-processing.....	5
2.2.3 Query extension .....	6
2.3 GUI .....	6
3.Changes to Baseline .....	7
3.1 Using newer version of Lucene .....	7
3.2 Standard analyser.....	7
3.3 Okapi BM25.....	8
3.4 Passage_title field .....	8
3.5 Boosting .....	9
3.6 Query-processing .....	9
3.7 Query extension .....	9
4 System Evaluation & Comparison with Baseline.....	9
4.1 System efficiency.....	9
4.1.1 Index size.....	9
4.1.2 Time to produce index .....	10
4.1.3 Time to search.....	11
4.2 System effectiveness.....	12
5 User Guide.....	13
5.1 Index.....	13
5.2 Search and Retrieve Results .....	13
5.3 Save Results.....	14
5.4 Custom Similarity Modification.....	15
6 Advanced Features to Answer the Questions.....	15
6.1 Narrow the size of the search field .....	15
6.2 Reduce the time it takes for the system to search.....	16

# 1 Statement of Completeness and Work Distribution

Statement	
<p>The existing system has completed all the functions in section 3 and has successfully demonstrated the functions to the professor. The parts that were not explained in time during the demonstration were also explained in the advanced features.</p> <p>The four team members of this group are fully involved in this project. Although there are different focuses on the project tasks for each of us, no one is only focusing on their own parts, and we work together to complete the entire task.</p> <p>Overall:  Jianwei Tang &amp; Pan are mainly responsible for the programming implementation of the system functions.  Zhe Han &amp; Shengguang Han are mainly responsible for testing and writing project reports.</p>	
Signatures	
Team member name	Signature
Jianwei Tang	
Yongrui Pan	
Zhe Han	
Shengguang Han	

## 2 Design

This project is the information retrieve system implemented by utilising Lucene search engine library. The system mainly implements two main functions. One is to generate an index dictionary based on the existing data collection, and the other one is to retrieve relevant data in the original data collection through the generated index dictionary. In addition, we use the Microsoft Macro Test Collection as the original document collection.



**Figure 1 The schematic diagram of the functional structure of our system**

The system mainly consists of three main components: index, search and GUI. In order to achieve the design goal, we define a class “index” and several methods for the first two components, as shown in the following table.

**Table 1** The main method in the code of our system

Functional Module	Custom Method			The key system method used
Index	CreateIndex			StandardAnalyzer BM25Similarity IndexWriter
	CreateCollection	TokeniseString		Document UpdateDocument
		GetWebTitleFromUrl		
		GetWebPageTitle		
	CleanUpIndexer			Flush Dispose
Search	CreateSearcher			IndexSearcher BM25Similarity
	SearchAndDisplayResults	GetWeightedExpandedQuery	GettingSynSets	MultiFieldQueryParser
		SaveFile		

## 2.1 Index

### 2.1.1 Indexing strategy

Firstly, it is the strategy for generate terms for original documents. We do the punctuation and stop words removal, and then lowercase all of the tokens. Here, we have not chosen to make a stem on the tokens, although stemming can greatly reduce the size of the index dictionary and the search time, it is also helpful for finding more “relevant” documents. This is because we consider that stemming will lead to losing information about the original form of the query, and sometimes this information is

useful. For instance, the word “organization” will be stemmed into “organ” by Snowball, so a search for “organization” will return results with “organ”, but “organ” has more meaning than “organization”. Furthermore, we plan to add synonyms of query terms for extending the query, then synonyms of some other meanings of “organ” will also be extended to the query, so that there will be many words in the query that are completely different from the meaning of “organization”, and finally lead to the search for a lot of irrelevant documents. In programming, we use Lucence's Standard Analyzer to implement this part of the function. Compared to Snowball, the Standard Analyzer only performs punctuation removal, stop word removal and letter lowercase. Next, in forming the index dictionary, Okapi BM 25 algorithm is adopt for calculating the weight of a term and a document. The BM 25 is based on probability theory and performs well and can provide accurate weight.

### **2.1.2 Collecting source documents**

In order to improve the quality of the index dictionary and the accuracy of the search, we did not directly index the Microsoft Macro Test Collection. We pre-processed the content that needs collect. We only collect some of the fields in the original source, including the URL of the passage, passage\_text and passage\_id.

Moreover, we extract the title of the passport from its URL to form another field. This strategy is based on the fact that the title is often the best summary of a document. If the title is associated with a query, then this document is very likely to be relevant to the query. We are currently using method "GetWebTitleFromUrl" which created by us to extract the title from the URL. The strategy is to extract the last valid field in the extracted URL.

For example, [https://en.wikipedia.org/wiki/Queensland\\_University\\_of\\_Technology](https://en.wikipedia.org/wiki/Queensland_University_of_Technology), this URL's last valid field (not ".html") is “Queensland\_University\_of\_Technology”, according to the naming convention of URL, this field is generally the title or theme of the article.

### **2.1.3 Handling source error**

In the experiment, we found that there are a lot of duplicate passages in the Microsoft

Macro Test Collection. Their passage\_ID are different, but their passage\_text and url are same. This will waste indexing time and memory. Therefore, I added a duplicate checking when collecting for prevent collecting a passage\_text which has been in the collection. When collecting a passage with a duplicate URL, we will use the passage\_text of this passage to override the passage\_text of the passage with same URL in the collection. The code is implemented as follows:

```
writer.UpdateDocument(new Term("url", passage.url.ToString()), document);
```

## 2.2 Search

### 2.2.1 Matching strategy

In order to be able to find relevant information, the similarity score is very important. Here we first use the calculated weights of the Okapi BM 25 algorithm to obtain a more accurate Similarity score. The method "BM25Similarity" is added into the main method "CreateSearcher". At the same time, we boost the high weight source field. As mentioned above, the title has high weight to indicate the passage relevant to a query. Therefore, we boost the title field. This is used as a strategy to improve retrieval accuracy. The code is implemented as follows:

```
Dictionary<String, float> boosts = new Dictionary<string, float>();  
boosts["passage_text"] = 2;  
boosts["title"] = 10;  
  
parser = new MultiFieldQueryParser(AppLuceneVersion, new String[] { "passage_text", "title" }, analyzer, boosts);
```

### 2.2.2 Query pre-processing

Before comparing the query with the index dictionary, we will pre-process the query. The strategy is to first remove the spaces and punctuation, lowercase them, and then remove the stop words. To be honest, in the presentation, we did not do stop words removal to the query. This is a serious mistake, although we cannot recover the previous points, but we think it is necessary, although adding this function is no longer able to recover the penalty points in our presentation, we think that stop words removal is necessary here, and the stop words in the source have been removed in indexing. The stop words in the query are completely incapable of successful matching in the search, and only wasting the resources and time.

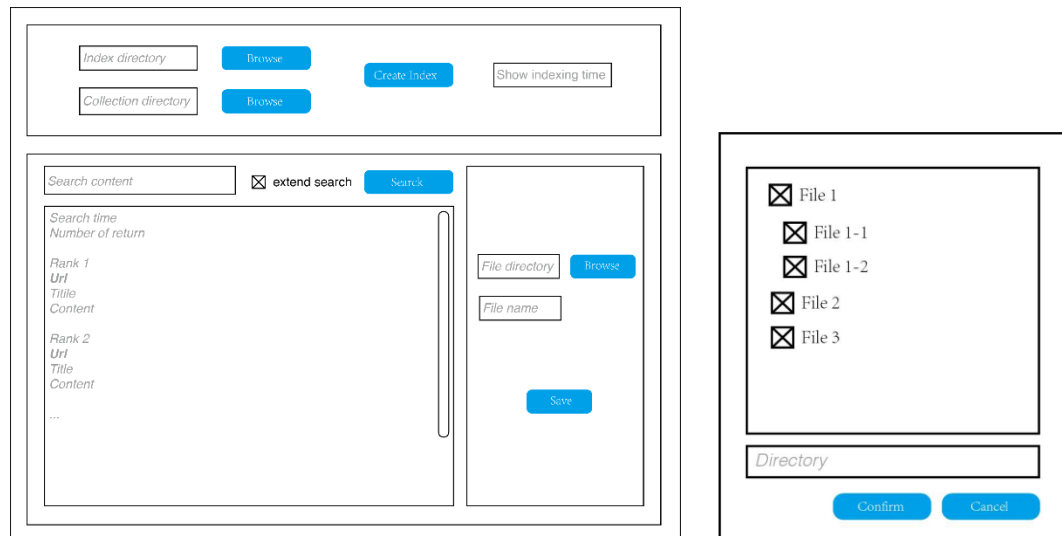
### 2.2.3 Query extension

In order to be able to retrieve more relevant documents, we do some query extension. The strategy is that using Wordnet to obtain the synonym of the terms in the query, and then add these synonyms into the query using for matching. In the program we have written two methods “GettingSynSets” and “GetWeightedExpandedQuery” to achieve this function.

Method “GettingSynSets” is used to get a synonym for a word. It will be embedding in the method “GetWeightedExpandedQuery”. The latter method can get the synonym of each term in the query and add them into the query.

## 2.3 GUI

Finally, as the GUI part of the human-computer interaction interface, we can complete all the functions and operations mentioned above in the GUI. In the index aspect, the user can choose the source collection package, custom index dictionary save address, and manually click to start indexing, the GUI can display the indexing time. In the search aspect, user can directly input the content that needs to be searched in the GUI, choose whether to extend the query, and then manually click to start the search. The GUI can display the relevant information of the search and the details of the top 10 related documents on the interface. In addition, User can save all the search results output as a txt file, and the user can customize the address and name saved by the file. Furthermore, if the user performs an error operation, such as the user click the button for starting indexing without choosing the source, the GUI will pop up an error prompt window. The mock-ups of this GUI are shown below.



**Figure2 Mock-ups of this GUI – Main interface & Browsing file address**

### 3.Changes to Baseline

Our system has made the following changes compared to baseline:

#### 3.1 Using newer version of Lucene

Lucene 3.0 is used in baseline, and here is Lucene e 4.8. This upgrade is to use “MultiFieldQueryParser” for create index. This is the basic of the that we can index in both field of “title” and “passage\_text”. In the baseline, it uses “QueryParser” for one field of “passage\_text”.

Next, it can get the term’s vectors when extracting the term. The 4.8 version of Lucene provides the method “TextField” to help me get the vectors of the term. Vectors can help to better calculate similarities, thus improving the accuracy of the search. In the method “Createcollection” we use “Textfield” instead of the “Field” method used in the baseline.

#### 3.2 Standard analyser

The baseline uses “SampleAnalyser”, and here is StandardAnalyzer. SampleAnalyer only provides the function of removing the punctuation space character and the lowercase of the letter. The StandardAnalyser adds the function of stop words removal. The stop words removal can greatly reduce the number of terms and reduce the size



of the index dictionary. And can eliminate the impact of stop words on Similarity.

### 3.3 Okapi BM25

Okapi BM25 is used for improving document, query weights. It also help calculating better relevant score of the document to a query. This has described in detail in the section 2.1.1 and 2.2.1. Presented in the code as follows.

```
public void CreateIndex(string indexPath)
{
    var dir = FSDirectory.Open(indexPath);

    luceneIndexDirectory = Lucene.Net.Store.FSDirectory.Open(indexPath);
    var indexConfig = new IndexWriterConfig(AppLuceneVersion, analyzer);
    indexConfig.Similarity = new Lucene.Net.Search.Similarities.BM25Similarity((float)1.2, (float)0.75);
    writer = new IndexWriter(dir, indexConfig);
}
```

```
public void CreateSearcher()
{
    searcher = new IndexSearcher(DirectoryReader.Open(luceneIndexDirectory));
    searcher.Similarity = new Lucene.Net.Search.Similarities.BM25Similarity((float)1.2, (float)0.75);
    //searcher.setSimilarity(new BM25Similarity(1.2, 0.75));
}
```

### 3.4 Passage\_title field

Extracting the title from passage URL and generating title field, this strategy has been described in detail in section 2.1.2. There are two points to add here.

One is why we extract the title from the URL instead of using the URL directly.

The reason is that if directly index the URL into the collection, it will get a lot of terms from the website's name. These terms are often not related to the content of the page. These terms will not only enlarge the volume of the index dictionary, but also disturb for terms and documents weight. Moreover, The impact relevance score of the document to the query.

In addition, in the code we also retain another method to extract the passage title, it is "GetWebpageTitle". This method's strategy is to enter the web page, obtain the title from the html information of the page, so that to get a more accurate passage title. However, because you need to open the URL to download the webpage of all of the passage, which leads to the indexing time is too long, so we did not use this method in the presentation.

### 3.5 Boosting

Comparing to the baseline, we use field-level-boosting here. Considering that the title related often means that the content is also relevant, so we have boosting the title field. When the title is associated with the query, the prediction of the relevance of the article to the query should be raised. This has been detailed in section 2.2.1.

### 3.6 Query-processing

The baseline only does lowercasing to the query. We add the punctuation and stop words removal to the improved system. The details of this has been described in detail in section 2.2.2.

### 3.7 Query extension


In addition to enhancing the query pre-processing, we have extended the query. The baseline does not do any extensions to the query, just search using the contents of the query itself. We use the wordnet to get the synonym of each term in the query, and add them all to the query to search, as described in the previous section 2.2.3.

## 4 System Evaluation & Comparison with Baseline

### 4.1 System efficiency

#### 4.1.1 Index size

The source we are using is the file “collection.json”, its size is 387,711 KB

 collection.json	2019/10/17 13:39	JSON 文件	387,711 KB
---	------------------	---------	------------

When we try to use Luke.Net 0.5 to open the indexing files, Luck.Net prompts a warning – “array dimension exceeds the supported range” and cannot work. We have to observe the Properties of the folder where the index file is located.

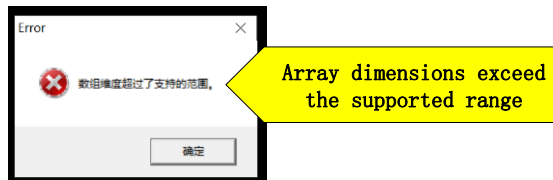


Figure 3 Error of Luke.Net

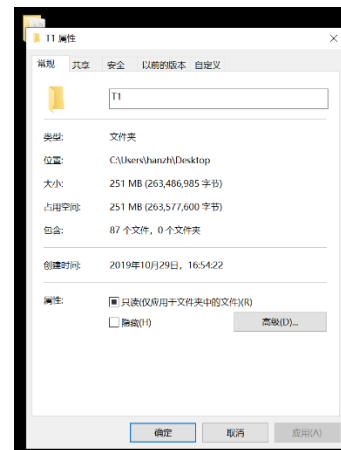


Figure 4 Index folder properties

As shown in the figure above, the size of the Index files is 263,489 KB, which is less than half the size of the Index file generated by Baseline (550,234KB). But it contains 87 files, and there are only 8 files in the Index file generated by Baseline. This may be the reason for Luke.Net to report an error.

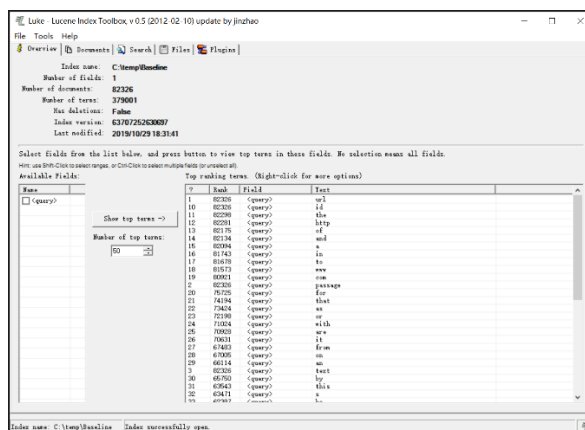


Figure 5 Baseline index shown in Luke.Net



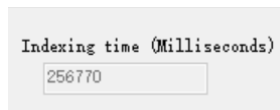
Figure 6 Baseline index folder properties

Firstly, the reason for the decrease in the size of the Index file is that we added the stop word removal strategy during the indexing process. But the reason why the file size is still large is mainly because we gave up stemming the tokens. Secondly, we have an additional field to collect passage title. In short, from the size of the Index file, the improved system greatly saves storage space.

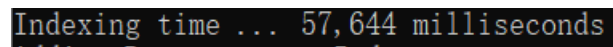
#### 4.1.2 Time to produce index

The display on the system GUI, the time that the system completes the generation of

the index dictionary is 256,770 milliseconds.



**Figure 7 System indexing time**



**Figure 8 Baseline indexing time**

The total time spent on Baseline indexing is 57,644 milliseconds, which is much shorter. It is because that during the indexing process, the improved system performs more operations, such as extracting title from the URL, adding the title to the new field, removing stop words, and introducing the Okapi BM25 for calculating the weight of term and document, etc., although the index dictionary generated by the improved system is half smaller than the baseline, but its indexing takes a lot longer.

### 4.1.3 Time to search

We set up four irrelevant and different types of queries for search testing. In the improved system, both of the extended query and the original query were searched. At the same time, the four queries were searched in the baseline. The test time is shown in the following table.

**Table 2 Searching Time**

Search Query	Searching Time ( millisecond)		
	Improved System		Baseline
	Extended Query	Original Query	
London Bridge	48	24	39
panda	175	75	96
what is conifer	218	54	117
ronald reagan	183	20	71

As can be seen from the above table, the search time used by the improved system to search without query extension is shorter than the search in baseline. This is because the stop words is removed both from the documents and query. This reduces the number of terms in the index dictionary and the query. The search for extended query is much slower than for the original query. The reason is that the query after the expansion adds a lot of terms. For example, the original query “London Bridge” only includes two words, which is 2 terms. After the extension, it includes 14 phrases. After tokenize, there are 15 terms, so the search time is greatly lengthened.

## 4.2 System effectiveness

We utilized trec\_eval to get these results, which the queries are “London Bridge”, “panda”, “what is conifer” and “ronald Reagan”. The first two questions are to make the search results more universal. The last two questions are that we deliberately added the misspelled words to see if the system will give The information that the user really wants to search. In the test we will use Luke.Net to collect and analyze the relevant parameters, and the results are shown in Figures 1, 2 and Table 3 show the relevant metrics to prove the effectiveness of the system.

num_q	all	4
num_ret	all	350
num_rel	all	35
num_rel_ret	all	33
map	all	0.4316
sm_ap	all	0.3591
r-prec	all	0.4410
bpref	all	0.9444
recip_rank	all	0.6875
ircl_prn.0.00	all	0.7500
ircl_prn.0.10	all	0.7500
ircl_prn.0.20	all	0.5833
ircl_prn.0.30	all	0.5417
ircl_prn.0.40	all	0.4396
ircl_prn.0.50	all	0.4229
ircl_prn.0.60	all	0.4229
ircl_prn.0.70	all	0.4169
ircl_prn.0.80	all	0.3583
ircl_prn.0.90	all	0.2572
ircl_prn.1.00	all	0.2572
P5	all	0.5000
P10	all	0.4000
P15	all	0.2667
P20	all	0.2250
P30	all	0.2000
P100	all	0.0825
P200	all	0.0413
P500	all	0.0165
P1000	all	0.0082

Figure 9 All documents results

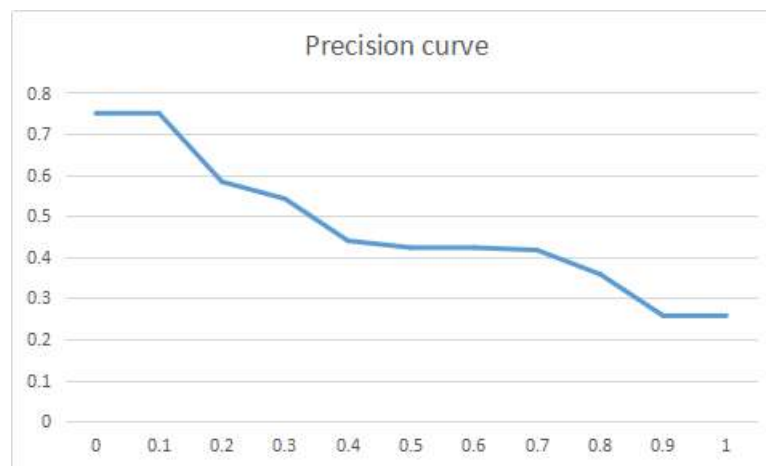


Figure 10 Precision curve

Table 3 Effectiveness metrics

Precision @ 10	Mean Reciprocal Rank	MAP
0.4	0.6875	0.4316

## 5 User Guide

### 5.1 Index

In current system we want to make the user better use this application, we divide the index into two different tasks. First, the user needs to create an index path through the interface. In this step, the user can arbitrarily select the path he wants to save the data. Then the user needs to confirm the collection path in the file. This path is used to import the previously present json file. After that user can click “Create Index” button to start creating index. In addition, user can clearly see how much time he uses to do the index.

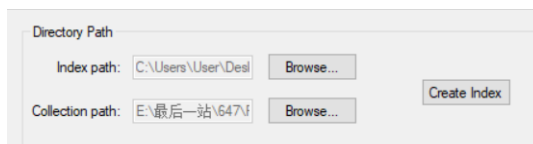


Figure 11 index

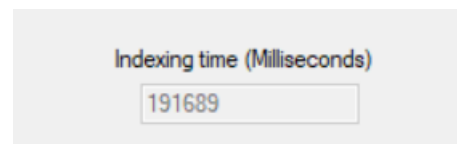


Figure 12 Indexing time

### 5.2 Search and Retrieve Results

After completing the index step, the user can start searching for the content that he wants to know. First, the user needs to enter the content that he wants to know in the box, and then click the “search” button to make the system search for the relevant content. After that, the related content will appear, and the content presented will have a rank. This is to determine the fit of the content and the keyword through the program. According to the requirements, a total of the top ten search results will be presented to the user.

It should be noted here that the system do searching by default using the **improved query**. In the result display, you can see the that the query displayed after “Search for” is far exceeds the inputting query. The term labelled by “^5” in the display is the term from the inputting query, and the other terms are extended from the original term.

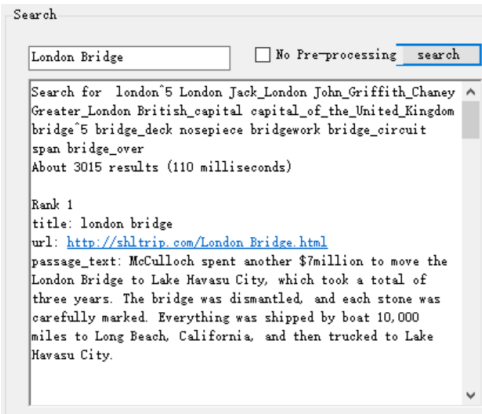


Figure 13 Search and the result for extended query

If the user does not want to do query extension, the user can check “No Pre-processing” and do searching. The result is shown below.

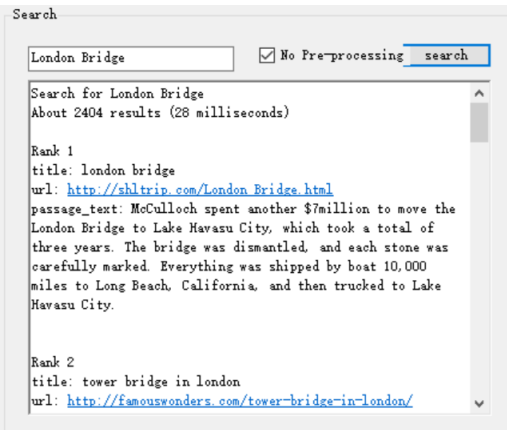


Figure 14 Result for original query

5.3 Save Results

If the user wants to save the searched content, the user first needs to create a save path and then name the file. The saved file will be in txt format.

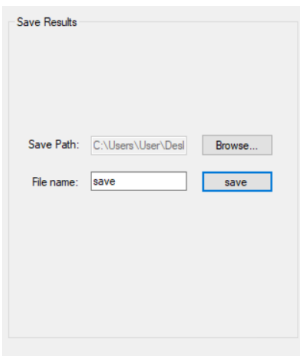


Figure 15 Save result

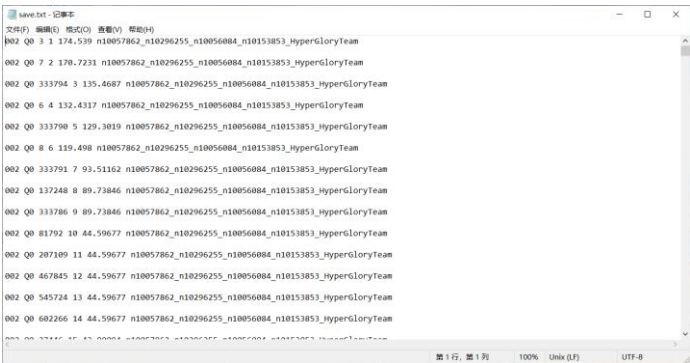


Figure 16 Format of results

## 5.4 Custom Similarity Modification

We use Okapi BM25 to improve custom similarity in the current system. This method adds two tuneable parameters,  $k_1$  and  $b$ , to the TF-IDF, which represent term frequency saturation and field length specification. After using this method, the search results will be more credible, and the higher the ranking results, the higher the fit with the keywords.

## 6 Advanced Features to Answer the Questions

### 6.1 Narrow the size of the search field

This method is to make the results obtained by the user after searching for keywords more credible. By narrowing the size of the search domain, you can rule out other factors and ensure that the searched content is in line with user expectations. And on this basis users will have a better search experience. In order to achieve this, we need to make changes here:

```
public void CreateCollection(string collectionPath)
{
    using (StreamReader r = new StreamReader(collectionPath))
    {
        string json = r.ReadToEnd();
        dynamic array = JsonConvert.DeserializeObject(json);
        foreach (var item in array)
        {
            foreach (var passage in item.passages)
            {
                Lucene.Net.Documents.Document document = new Document();
                string title = GetWebTitleFromUrl(passage.url.ToString());
                document.Add(new TextField("title", title, Field.Store.YES));
                document.Add(new StringField("url", passage.url.ToString(), Field.Store.YES));
                document.Add(new TextField("passage_text", passage.passage_text.ToString(), Field.Store.YES));
                document.Add(new StringField("passage_id", passage.passage_ID.ToString(), Field.Store.YES));
                writer.UpdateDocument(new Term("url", passage.url.ToString()), document);
                //writer.AddDocument(document);
            }
        }
        System.Console.WriteLine("All documents added.");
    }
}
```

Here the Document class is mainly used to access Field-related field information. At this time according to the code system is adding "title", "url", "passage\_text" and "passage\_id" four types of objects, here we add the "update document" command to verify that there is a duplicate ID and discard the duplicate ID if there is one. This way, the size of the search field is reduced.



## 6.2 Reduce the time it takes for the system to search

This method is implemented by a Word statement. Its main function is to minimize mismatch by adding special words to the query during the process of query expansion (by Wordnet). This not only reduces the search time but also the accuracy of the search results. This command using Word should be added to the query expansion process, as shown in the following figure:

```
public string GetWeightedExpandedQuery(string query)
{
    string expandedQuery = "";
    if (true)
    {
        string[] tokens = TokeniseString(query);
        foreach (string token in tokens)
        {
            expandedQuery += " " + token + "^5";

            var synSetList = GettingSynSets(token);
            string[] words = { };
            foreach (var synSet in synSetList)
            {
                words = words.Concat(synSet.Words).ToArray();
            }
            var unique_words = new HashSet<string>(words);
            foreach (string word in unique_words)
            {
                if (word != token)
                {
                    expandedQuery += " " + word;
                }
            }
        }
    }
    return expandedQuery;
}
```