Introduction to Web Search and Mining Option A - Project Report

Ruoyu Deng, Xutong Lu, Fuming Zhang, and Wencai Zhong
118033910030, 118033910116, 118033910025, and 118033910101

1 Introduction

Information retrieval (IR) is the activity that people search information they needed from data collection. A good information retrieval system will store and organize data in a proper way and return relevant information to users. In our project, we build an information retrieval system based on the data collected from Stack Overflow(https://stackoverflow.com/).

Stack Overflow is a community where everyone who codes can learn and share their knowledge, and build their careers. More than 50 million unique visitors come to Stack Overflow each month to help solve coding problems, develop new skills, and find job opportunities. Users can ask and answer questions about coding problems on the website.

In our project, we crawl questions and answers from Stack Overflow and build a Stack Overflow search engine. Four main functions of our search engine are listed below:

1) User can ask a question about a technique or model like those in the Stack Overflow and the system will return a ranked list of all linked questions with their answers and comments, and a ranked list of related questions with their answers.
2) User can ask a keyword which may be a machine learning framework, programming language or any key words that might appear in the page and the system returns a ranked list of answers and questions.
3) Advanced search: users can enter search keyword for a particular region in the page, e.g., search in the tag, the question or the answer. The advance search will give the interface to allow the user to choose what region to search from.
4) The system will return answers in a particular order.

The following of our reports are organized as follows, in Section 2, we introduce the design and implementation of our crawler. Section 3 introduces the database we store data and the web page of our search engine. The details of our search engine is introduced in Section 4. It contains the indexing part and the searching part. We compare the performance of our search engine with Whoosh in Section 5. Finally, we show conclusions in Section 6.

- *Project Link: https://github.com/AlexChang/WSM-SOSearch*
- *Raw Data Link: https://pan.baidu.com/s/16_C4dFrXuUfPwuurDWZFIA Extraction Code: ne6l*
- *Database Link: https://pan.baidu.com/s/1lCxzOB9GWIRqG1L4Zf8eYg Extraction Code: qgao*
- *Website Demo Link: http://202.120.38.54:8000, http://202.120.38.54:8001*
- *E-mails: 767423930@qq.com, zhangfuming-alex@sjtu.edu.cn*

2 Crawler

In this section we first analyze the StackOverflow website structure and design our crawl strategy in 2.1. Then we discuss our Scrapy-based python implementation in 2.2.

2.1 Analysis and Design

The goal of the crawler is to crawl at least 2-year (2016-2019) worth of questions and their corresponding answers from Stack Overflow. Before we started designing crawlers, we carefully examined the site structure of StackOverflow and learn some crawler techniques [1]. Soon we found out that StackOverflow's question list page (https://stackoverflow.com/questions) lists all the questions which the site contains, and it is sorted by time (by default). This page can display up to 50 items (i.e. questions) per page. Each item contains the title, link, summary, questioner and statistics such as votes, answers, views, etc. Follow the question link, in the question details page (https://stackoverflow.com/questions/<question_id>), we can see the details of the question, answer, comment and so on. Move back to the question list page, in the case of displaying 50 questions per page, the 100,000th (more accurately, when the crawler actually crawled the data, it was about the 101,500th page, and when the report was written, it was about the 104,000th page) page shows the questions raised by users at the end of 2016. Therefore, from mid-May 2019 to the end of 2016, there are about 5 million questions in total.

After performing a certain analysis of the structure of the website, we intend to crawl the questions and corresponding answers in the past two years in the following two steps. First, we follow the question list page to crawl 5 million question links, and then we will follow these 5 million question links to crawl specific questions and answers. We put the details of crawling and parsing in the next subsection.

2.2 Implementation

Considering that the amount of data we need to crawl is relatively large, we plan to accomplish this based on the python crawler framework, Scrapy. Scrapy is an open source and collaborative framework for extracting the data from websites in a fast, simple, and extensible way. With the help of Scrapy, we mainly implemented two crawlers, question list spider and question answer spider. Next we will introduce these two crawlers one by one in 2.2.1 and 2.2.2.

2.2.1 Question List Spider

As for question list spider, whose logic is relatively simple, it starts from the base url (https://stackoverflow.com/questions?sort=newest&pagesize=50&page=), increments the request page number each time, sends the request and parses the response sequentially. In order to counter StackOverflow's anti-crawler mechanism, we added two extended downloader middlewares for it. One is a random user agent middleware, each request sent contains a fake user agent randomly generated by the third party library fake_useragent in Python. The other downloader middleware is a custom retry middleware, which additionally handles the HTTP 429 error code. Here HTTP 429 error code stands for too many requests. In the custom middleware, we specifically judge that when the HTTP error code is 429, we pause our crawler engine for 300 seconds. Back to the crawler, when we get the request response, we parse the question link and the question creation time, and *yield* them to Scrapy default item pipeline. Finally, the parsed data is placed in a list and stored as a json format file. In the actual crawling process, we constantly improves the function of the crawler and fixes bugs. Thus the results are divided into several segments. We merge these results together, remove duplicated items and finally get **5000798** unique questions (links).

2.2.2 Question Answer Spider

When it comes to the question answer spider, we take advantage of the concurrent requests provided by Scrapy to speed up crawling, since we need to crawl more than 50 times the data compared to the former. Thus we override the start_requests function to *yield* all the valid requests at the very beginning in current process. As for when these requests will be sent specifically, it is controlled by the core engine and scheduler of Scrapy. In order to achieve better concurrent access, we add parameters such as start and stop for this crawler, so that each independent process is only responsible for crawling a part of the question links. At the same time, in order to facilitate error handling and repeated crawling (for example, when an unexpected error happens and the program aborts, but the question links that the process is responsible for has not been fully crawled, we need to restart the process and let it finish crawling the remaining part), we turn off the function of filtering duplicate requests provided by Scrapy. We implement our own duplicate filter using set and request url, and store the content locally.

One troublesome aspect of crawling the content of the question details page is that some of the comment lists of questions or answers are not complete and we need to send an additional request to fetch the complete one. Initially we tried to use the inline-request third-party library to crawl the complete comments, but it would cause other problems with the request processing. Later, we learned from the Scrapy documentation that to handle this situation in a more 'scrapy' way is to use the meta field of the request object provided by Scrapy. Thus we use this meta field to pass information about the content that has been crawled and parsed and the list of comments that need to be crawled. After we complete all those incomplete comment lists in the question detail page, we *yield* the entire data.

In order to standardize the data format, we use the Item and Field provided by Scrapy to predefine the following six data types: Question, Answer, User, Comment, LinkedQuestion and RelatedQuestion. Question contains id, title, link, asked, viewed, active, vote, star, content, status, tags, users and comments these 13 fields. Answer contains id, vote, accepted, content, users and comments these 6 fields. User contains action, time, name, link, is_owner, revision, reputation, gold, silver and bronze these 10 fields. Comment contains score, content, user, user_href and date these 5 fields. LinkedQuestion contains id, title, link, vote and accepted these 5 fields. RelatedQuestion contains id, title, link, vote and accepted these 5 fields. Considering that the website framework Django we used only supports the database model, we convert the json format data into the sqlite3 database and the details will be introduced in Section 3.1.

What's more, we have also used two downloader middlewares in this crawler. The first one is a random agent middleware, which is exactly the same as the previous crawler. The other one is change proxy middleware, which is also a custom retry middleware. As the name of this middleware shows, in order to avoid receiving the HTTP 429 error code, we add HTTP(s) proxies in this crawler. We use the paid HTTP(s) proxies provided by Qingting IP (https://www.qingtingip.com/). When the crawler generates a request based on the question link, we assign a random HTTP proxy to each request. If a TimeoutError, TCPTimedOutError or ConnectionRefusedError occurs during the request, we will replace the proxy through the above middleware. In the crawler's own logic, we have also added a function to process exceptions, when the HTTP 429 status code is encountered, the proxy will also be replaced and the request will be resent. We maintain a list of available proxies in the crawler and its size can be specified by the parameter when the crawler starts, the default value is 5. However, since StackOverflow uses the https protocol, sometimes we will encounter a TunnelError when sending a request (even with https proxy).

In addition to the above proxy mode, we also designed a direct connection mode. In this mode, when the crawler encounters the 429 error code, we randomly pause our crawler engine for 180 to 600 seconds. Besides, we have enabled the auto throttle strategy provided by Scrapy in both proxy mode and direct mode.

Through our test, when assigning 5 available proxies to a independent crawler process, we can crawl the question detail pages at a speed of 100 pages per minute (including the requests to crawl all comments), with a combined

success rate of 90%. The main reasons for request failure can be divided into the following three categories. First, the question pointed to by the question link does not exist (HTTP 404). Second, although the question exists, but it is automatically redirected by StackOverflow to refer to another question, which leads to duplication and will be automatically removed by our deduplication mechanism. Finally, some requests encounter TunnelError. It is worth mentioning that when we use the direct connection mode, the crawl speed is only 2 pages per minute on average due to StackOverflow's 429 anti-crawler mechanism.

As mentioned above, we have developed a proxy mode to verify that this method is feasible and efficient in dealing with the StackOverflow anti-crawler mechanism. But since the proxy we use is charged, we didn't spend a large amount of money to buy enough proxies to quickly crawl all the data. As of the time of this writing, we have crawled about 750,000 unique question details. Such a large amount of data is already a big challenge for our index and search engine. We will discuss the implementation of our index and search engine in Section 4.

At the end of this subsection, we would like to share a very unique error we encountered during crawling the data. When our crawler tries to crawl the following question detail page (https://stackoverflow.com/questions/44408405/sending-get-variable-in-echo-meta-refresh), Scrapy redirects the page to the following link (https://en.wikipedia.org/wiki/.$_GET['w']). After our careful inspection, we found that the reason why this error may occur is as follows, the default opened meta refresh middleware of Scrapy will check the first 4096 characters of the reponse, and try to match the "http-equiv refrsh" parameter. Very coincidentally, one line of the description of the question happens to be "echo '<meta http-equiv="refresh" content="0;URL=https://en.wikipedia.org/wiki/.$_GET['w']" />';", so the middleware incorrectly matches this url and redirects the page automatically. In order to solve this problem, we closed this middleware.

3 Database and Web

In this section, we introduce the database in Section 3.1 and web in our search engine in Section 3.2.

3.1 Database

We crawl data and store them as json files while crawling. In this section, we build a sqlite3 database to store the data.

SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine. SQLite is the most used database engine in the world. SQLite is built into all mobile phones and most computers and comes bundled inside countless other applications that people use every day.

We build 6 tables to store data, and they are listed below:

Table: answer

key	class
id	INT
rid	INT
vote	INT
accepted	INT
content	TEXT

Table: comment

key	class
id	INT
rid	INT
score	INT
content	TEXT
user	TEXT
user_href	TEXT
time	TEXT

Table: linked_question

key	class
id	INT
rid	INT
qid	INT
title	TEXT
link	TEXT
vote	INT
accepted	INT

Table: question

key	class
id	INT
title	TEXT
link	TEXT
asked	TEXT
viewed	INT
active	TEXT
vote	INT
star	INT
content	TEXT
status	TEXT
tags	TEXT

Table: related_question

key	class
id	INT
rid	INT
qid	INT
title	TEXT
link	TEXT
vote	INT
accepted	INT

Table: user	
key	class
id	INT
rid	INT
name	TEXT
action	TEXT
time	TEXT
link	TEXT
is_owner	INT
revision	TEXT
reputation	INT
gold	INT
silver	INT
bronze	INT

We read data from the json files and then write data into the database. There are some problems in building the database. The first is data missing. Some keys are empty while inserting data into the database. We use "None" to fill the empty data. Another problem is repeated data. Repeated answers and questions will cause errors when we search information from the database later. We need to filter out repeated data in the database.

The size of our database is 5.52GB in total. It contains 791,812 answers and 940,219 questions.

3.2 Web

In this subsection, we first introduce our design overview in 3.2.1. Then in 3.2.2, we will discuss the implementation details.

3.2.1 Design Overview

Considering the flexibility and ease of use of Python, we decided to build our webside with Python from the very beginning and certainly we would not start from scratch. So before we started, we investigated some of the most popular Python web frameworks, including Django, web2py, falsk, Tornado, web.py, etc. And finally, we chose Django [2] since it's fully functional and well documented. Its detailed debugging features provide great convenience for our development. However, as a heavyweight web framework, we did spend a lot of time learning how to use it. After selecting the web framework, we chose the well-known front-end component, Bootstrap, to help us build web page styles. Bootstrap uses a responsive design philosophy that focuses on mobile prioritization, providing a rich set of predefined styles, components and js plugins.

In addition to the above framework and component, we have also used a Django search extension called Haystack. It pre-defines some search-related tools and even provides support for the four search backends of Solr, ElasticSearch, Whoosh, and Xapian, allowing us to implement a wide variety of search functions with just a few basic configurations. But of course, in this project, we were asked to implement our own search engine. And we achieved this by extending the standardized interface provided by Haystack. Details on index building, search engine, and the extension to Haystack search backend are discussed in Section 4. The search webside we implemented provides three types of pages, the question list page, the question details page, and the search results page. In the next subsection 3.2.2, we'll introduce the implementation details of these three types of pages.

3.2.2 Web Implementation

First we would like to introduce the overall structure of our project. We built a project called SOSearch and then built an application called websearch in the same directory. In the project layer configuration file, setting.py, in the final server deployment (http://202.120.38.54:8000), we turned off the debug option and assigned a list of trusted hosts to it. In addition, in order to provide static files (Bootstrap and custom style files) services in production(since we have closed debug mode) we used whitenoise and configured it in the middleware. In the installed_apps setting field, we added websearch and Haystack, and configured our own test backend, which is the extension of Haystack's simple backend, in the corresponding haystack_connections field. Next we configure the path matching rules in the urls.py of the project layer. Here, we pass the url that starts with 'admin' to the administration component that comes with Django, and the rest will be forwarded according to the settings in file url.py of our application layer.

Specific to our project layer directory 'websearch/', it mainly contains the generated 'migrations/' directory, which is used to store the database operation instructions generated by the *makemigrations* command, the 'static/' directory, used to place style files, the 'templates/' directory, used to store html templates, the 'templatetags/' directory, used to place custom template tags. In the 'templates' directory, we first define base.html, which contains the most basic elements of the html page, such as head, meta, and link tags which link to css files. Then we define the navigation bar with the search function fixed at the top in the body and reserve the expandable block for information display, and finally introduce the javascript scripts needed for the page effect. Since Django's html template does not support python callable, it only supports variable access, and simple if and for logic, we have also defined some auxiliary tags in the 'templatetags/' directory to implement link construction and pagination management.

In addition to these subdirectories, there are some other important files placed in the base application directory. The file models.py defines the model, which uses the model interface provided by Django to encapsulate the data. In this way, no matter what the database backend is actually useed, a unified interface is provided for Django development. Usually, it works well when we build a database from ground up by defining model classes at first and then let Django generates the database schema itself. But this time, we already have a database. So we use the *inspectdb* command to convert the existing database schema to the model interface required by Django. The model it generates automatically selects a field in the database as the primary key. The Django data model does not support multiple fields as primary keys.But the user, comment, linked question and related question tables we originally wanted to define do not have any field whose value is unique and can be used as the primary key. In order to solve this problem, we manually assign an incremental unique id to each data when creating the table.

File search_index.py is required by Haystack and is handed over to the Haystack backend to determine which fields need to be indexed and which fields need to be

reserved as extra fields. In our project, we create a full-text index of the title and content of questions and the content of answers, while retaining the number of votes of questions and answereds as additional fields for sorting. The file url.py defines the path matching rules of the application layer. The empty path and the path starting with 'questions/' are controlled by the index view. The path starting with 'questions/' and connected with a question id is controlled by the detail view. Finally, The path starting with 'search/' is handed over to the search view provided by Haystack.

The last file we want to mention is views.py, where we process the received request and get the required data from the model (database) or search index and finally pass it to the html template for rendering. In the index view, we first fetch all the questions from the model (database) and sort them according to the key selected by the user in the page, then paginate it and add some additional information like the number of answers, the questioner, the asked time, etc. All of this information is then sent as a context to the html template. Similarly, in the detail view, we fetch the corresponding question from the model (database), complete the information, and sort the answers, then pass all the information to the template to render according to predefined page styles. The search view is mainly provided by Haystack, but we override its get_context function to sort search results and complete information according to our needs.

4 Search Engine

4.1 Index

In the age of big data, information retrieval technique becomes more and more important. Take the website of Stack Overflow as example, there are millions of questions produced in the past two years. How to get wanted information quickly in such volume of data?

For structural data like user information, it's much easier to query with the help of database techniques. But for text data like websites, it takes too much time to check whether a document contains the key word by brute force search, which makes it unacceptable. That's why we need index technology.

4.1.1 Preprocessing

Before indexing, we need to extract terms from the texts of documents.

Tokenization. Given a sequence of characters (text in our task), tokenization is to split it into pieces. A token is an instance of a sequence of characters in some particular document that are grouped together as a useful semantic unit for processing. We use the Natural Language Toolkit NLTK to tokenize, it first split the text into sentences by punctuation, and then uses regular expressions to tokenize sentences as in Penn Treebank.

Token Normalization. A type is the class of all tokens containing the same character sequence. A term is a normalized type that is included in the IR system's dictionary.Token normalization is the process of transforming tokens into a canonical form for better searching. For example, the word 'LIKED' should match 'like'. To normalize tokens, we first turn tokens in lower case form and eliminating numbers. Then we use Porter Stemmer to extract stems of tokens, which is called called stemming.

Eliminating Stop Words. There are some common words which may occur in most documents, which should not consider when calculating relevance between query and documents. So before indexing, those stop words are filtered out.

4.1.2 Boolean Index

A intuitive way to make the index is called Boolean Index. It records the inclusion relation between documents and key words, which looks like:

Table: Boolean Index

	doc1	doc2	doc3	...
term1	0	1	0	
term2	1	0	0	
term3	0	1	1	
...				

With this Boolean Index [3], given a key word, the corresponding vector of 0 and 1s represents whether the key word appears in the documents. If we want to documents contain both term1 and term2, it can be easily done with the AND operation of the vectors of two terms. Although Boolean Index is easy to implement and fast to query, it has several disadvantages:

- It can't show the relevance between terms and documents. For example, the times the term appears in the document can reflect the relevance, the Boolean Index doesn't consider it. So it's more like data extraction other than information extraction.
- Apparently for some uncommon terms, the vectors should be sparse, which means only few 1s in the vector. In this case, storage the whole vector will waste much space. Therefore, the storage efficiency of Boolean Index is low.
- Complicated Boolean expressions when the query is complex.

4.1.3 Inverted Index

Considering the disadvantages of Boolean Index, we use Inverted Index in our project. Common index refers to query document content from document ids. Inverted Index do it inversely, query document id from text of documents. Compared to Boolean Index which stores relations of terms with all documents, Inverted Index [4] only records documents with terms appeared. Figure 1 shows the structure of the inverted index.

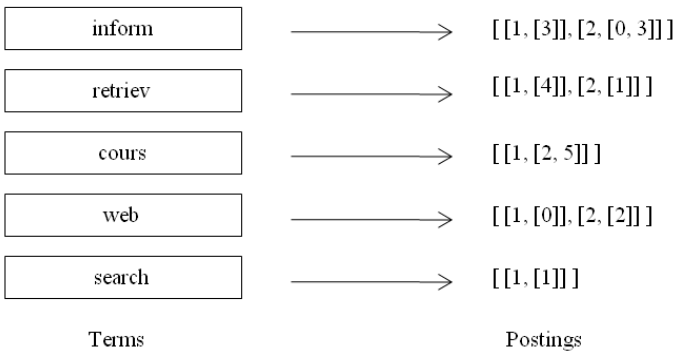

Fig. 1. Inverted Index

As we can see in the picture, there are mainly two parts of inverted index. The first part is a dictionary which contains the terms. Each key of the dictionary is a term. The dictionary can be implemented as hash or B-tree. The second part is the posting list of documents related to terms. The elements of the list are corresponding to the documents related to the terms, each of which contains the term frequency and the document id. Considering the size of index, we don't storage the contents of the documents.

4.1.4 Index Construction

When the volume of data are large, it's not realistic to construct index of all data at one time. Further more, in practical environment of search engine, the data are generated in real time. Therefore, we use the Single-pass in-memeory indexing (SPIMI) algorithm. The pseudocode of SPIMI is:

```
SPIMI-INVERT(token_stream)
 1  output_file = NEWFILE()
 2  dictionary = NEWHASH()
 3  while (free memory available)
 4  do token ← next(token_stream)
 5     if term(token) ∉ dictionary
 6        then postings_list = ADDTODICTIONARY(dictionary, term(token))
 7        else postings_list = GETPOSTINGSLIST(dictionary, term(token))
 8     if full(postings_list)
 9        then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
10     ADDTOPOSTINGSLIST(postings_list, docID(token))
11  sorted_terms ← SORTTERMS(dictionary)
12  WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```

Fig. 2. SPIMI Algorithm

4.1.5 Regional Index

To support advanced search of searching in questions or in answers or in specific tag, we support regional index. Specifically, we construct indexes for questions, answers and tags respectively. All indexes are stored in database to improve the reading and writing performance. The table of index is:

Table: index	
key	class
term (primary key)	TEXT
df	INT
docs	TEXT

4.2 Search Model

All web pages are indexed and stored in the database after index procedure. Now we can build a search model for the search engine. We have many optional methods like vector distance, probability model, BM25 model and so on. The BM25 [5] algorithm is very famous and has a good performance. We apply the BM25 model for searching.

4.2.1 BM25 Search Model

Okapi BM25(BM stands for Best Matching) algorithm is a ranking algorithm which can be used for search engines to estimate the relevance of documents to a given search query. The BM25 algorithm based on the probabilistic retrieval framework. It is a bag-of-words retrieval function that ranks a set of documents based on the query terms appearing in the documents, regardless of their proximity within the document. Our implementation of BM25 model based on the formula below:

$$BM25_{score}(Q,d) = \sum_{t \in Q} w(t,d)$$

$$w(t,d) = \frac{qtf}{k_3 + qtf} \times \frac{k_1 \times tf}{tf + k_1(1 - b + b \times l_d/avg_l)} \times log_2 \frac{N - df + 0.5}{df + 0.5}$$

In which, we have:

- qtf: the word frequency of query
- tf: the word frequency of document
- l_d: length of document
- avg_l: average length of document
- N: number of documents
- df: the frequency of documents
- b, k_1, k_3: hyper-parameters

It is complicated but easy to understand. The first formula is an external formula. A query Q may contain multiple terms. For example, "install python" contains two terms "install" and "python". We need to calculate the contribution of "install" and "python" separately. The contribution scores of document d are $w(t,d)$, and then adding them together is the score of the document d for the query Q.

The second formula is to calculate the score of a term t in document d, which consists of three parts.

The first part is the score of the term t in the query Q. For example, in the query "Chinese speaks Chinese", the term "China" appears twice, at this time qtf=2, indicating the document that the query hopes to find is more relevant with "China". In other words, the weight of "China" should be greater, but in general, the query Q is very short and is unlikely to contain the same term, so this factor is a constant that we can ignore when we implement our system.

The second part is similar to the TF term in the TF-IDF model. That is to say, the more times a certain term t appears in the document d, the more important t is. But if the document becomes very long, the tf tends to be larger too. So we normalize the effect by using the length of the document divided by the average length, i.e. l_d/avg_l . In this part, k1 and b are tun-able hyper parameters.

The third part is similar to the IDF item in the TF-IDF model. That is to say, although the stop words such as "a", "is", "are" appear many times in a document d, they have appeared in many documents, so the contribution of these words to d is not high. Instead, words that appears very rare such as "diabetes" can distinguish between different documents, and the contribution of these words to the document should be higher.

Therefore, according to the BM25 formula, we can quickly calculate the scores of different documents t on the query Q, and then give the results according to the ranking of the scores. The pesduo code can be seen as the fllowing. The calculation procedure has been described in this part.

```
import numpy as np
def BM25_alg(sentence, docs):
    seg_list = segmentation of sentence
    cleaned_dict = preprocess of seg_list
    BM25_scores = {}
    for term in cleaned_dict.keys():
        for doc in docs:
            calculate the BM25 score
    results = sort by BM25 score
    return results
```

4.2.2 Query Preprocess

We need to carry out preprocess of queries mainly because of two reasons. The first reason is that the user may enter a sentence with some meaningless terms or symbols. The second reason is that we must process the queries to find the search pattern of the query. For example, we use [] to stands for field search. We must recognize it by preprocess of queries.

A little different with the preprocess of documents, we first recognize the patterns of the queries. If we find that the query is the pattern for field search, we can choose the corresponding search tool. And after the pattern recognization, we carry out a preprocess similar with the document part. We do tokenization, normalization and elimination.

Tokenization is to split the queries into pieces. We use the Natural Language Toolkit NLTK to tokenize, it first split the text into sentences by punctuation, and then uses regular expressions to tokenize sentences. Then we make normalization. A term is a normalized type that is included in the IR system's dictionary. Token normalization is the process of transforming tokens into a canonical form for better searching. For example, capital words are normalized. To normalize tokens, we first turn tokens in lower case form and eliminating numbers. Then we use Porter Stemmer to extract stems of tokens, which is called called stemming. There are some common words which may occur in most documents, which should not consider when calculating relevance between query and documents. So before indexing, those stop words are filtered out. Then we get the filter query and we can get it for BM25 ranking.

5 Results

In this section, we compare the performance of our search engine with Whoosh.

Whoosh is a library of classes and functions for indexing text and then searching the index. It allows users to develop custom search engines for their content. For example, if users were creating blogging software, users could use Whoosh to add a search function to allow users to search blog entries.

We choose a data set which size is 35.5M. It contains 4,999 questions and 6,055 answers. We build our search engine and Whoosh engine on the data set at the same time, and we compare the performance of the two search engines.

First, we compare the two search engine's speed in building index. It takes 87 seconds to build index for our search engine, and for Whoosh, it takes 6 minutes. Our search engine is 4.13 times faster than Whoosh.

Then we test different queries to compare the results of the two search engine. We use five different queries: python, sklearn, print, index, hello world. And the number of searched results are listed below:

TABLE 1
Number of searched results of our search engine and Whoosh

	python	sklearn	print	index	hello world
our	360	6	745	469	177
Whoosh	425	7	771	575	70

As we can see from the result, when we search only one word, our search engine returns a bit fewer results than Whoosh. When the query contains more than two words, our search engine returns more results than Whoosh. It is because our search engine also returns the search result of each word.

Here are the result of query "sklearn" of our search engine and Whoosh:

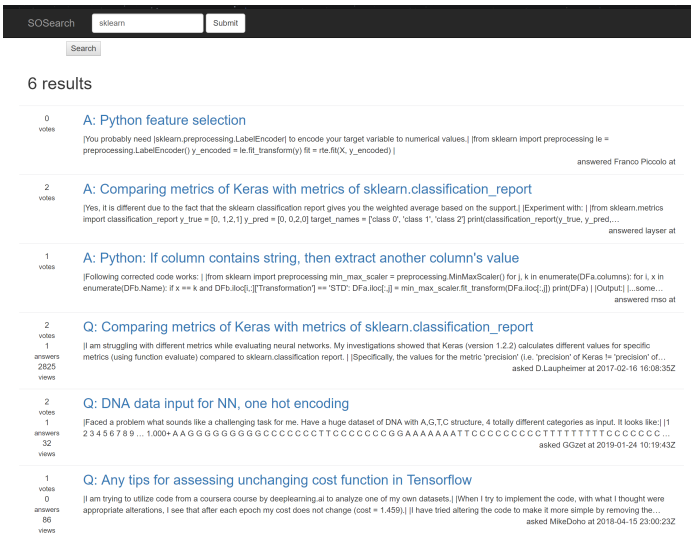

Fig. 3. Our model's searching results for "sklearn"

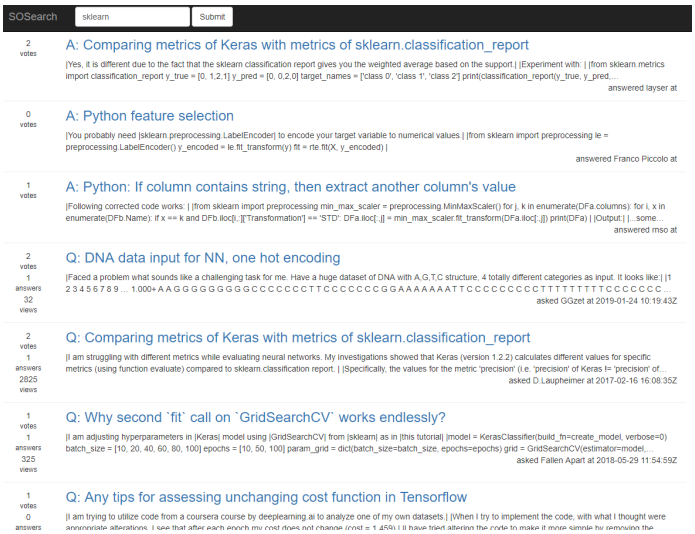

Fig. 4. Whoosh's searching results for "sklearn"

Comparing the two results, we can find that the two search engines return similar searching results.

6 Conclusion

In our report, we firstly introduce the design and implementation of our crawler. Then we introduce the database we store data and the web page of our search engine. The details of our search engine is introduced in Section 4. It contains the indexing part and the searching part. Finally, we compare the performance of our search engine with Whoosh.

Acknowledgments

The authors would like to thank Prof. Kenny Q. Zhu and TA Flora Huang and Kelsey Huang for their instructions on this work.

References

[1] M. A. Kausar, V. Dhaka, and S. K. Singh, "Web crawler: a review," *International Journal of Computer Applications*, vol. 63, no. 2, 2013.
[2] A. Holovaty and J. Kaplan-Moss, *The definitive guide to Django: Web development done right*. Apress, 2009.
[3] E. Greengrass, "Information retrieval: A survey," 2000.
[4] D. Harman, E. A. Fox, R. A. Baeza-Yates, and W. C. Lee, "Inverted files." 1992.
[5] S. Robertson, H. Zaragoza *et al.*, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.