**My Books Searcher: A report of group 13’s search engine**

QIU Yaowen 1730026087

QIAO Heng 1730026086

WU Xiaoxuan 1730025130

**Abstract**

Our group implemented a Django and Hadoop based books search engine. Users can search any books by input an English key word. We used python to implement an auto text collection programme to crawl free and licensed e-books from *Gutenberg*. Total N=649 .txt files of e-books were collected. Then, with the help of Hadoop map reduce programme, we successfully acquired the index file in three different ways and ranked the file names according to the TF-IDF algorithm by python. Furthermore, in order to reduce the time cost of the whole project, we processed the index file into dictionary readable format so that the programme could read the index file and transform it into dictionary without problem. Last, Django was used to provide a beautiful web interface to users and combine the back-end and front-end together. Serval features such as input correction, key word highlight, context, historical records and word attributes functions in our search engine. Multi-Thread-Processing, dictionary techniques and package *linecache* are used to decrease the time cost of our search engine to average time cost 1.2s. In the future, functions like paging or searching based on phrases will be added into our project to make the performance better.

**key words: Text crawling, Django, Hadoop, TF-IDF, Multithreading, Search Engine**

**(I)Text Files Collection**

**Method**

Our project aims to provide a powerful and convenient search engine for people who love reading books. Therefore, the data for our project should be some .txt file. During our searching procedure, our group find a website called ***Gutenberg(http://www.gutenberg.org/)***(figure1***)***.Project Gutenberg is a library of over 60,000 free eBooks. Choose among free epub and Kindle eBooks, download them or read them online.

图片包含 屏幕截图

描述已自动生成

Figure 1 The capture of gutenberg website

In order to make the data collection procedure fast and error-free, our group decided to use python to crawl enough e-books in this website automatically. Let’s take one e-book as an example. When we click into the image of a book, there is a plain .txt file on the website(figure2), It is quite easy to download it. According to further observation, the url of each e-book has same pattern. For this e-book, the url is <http://www.gutenberg.org/cache/epub/29450/pg29450.txt>, the number 29450

图片包含 文字, 屏幕截图

描述已自动生成

Figure 2 The capture of a ebook

is exactly the number of this book. If we replace ‘29450’ with any number we like, there should be another e-book .txt file in this webpage. However, in this .txt file, it is hard to acquire the title of this book since it is not apparent. So, we back to the page <http://www.gutenberg.org/ebooks/29450>, find that the title is exactly written in <h1></h1>(figure3), it is much more easy to get it.

图片包含 屏幕截图

描述已自动生成

Figure 3 The title is written in h1

To implement this auto .txt crawling programme, we choose python as our programming language since it provides us powerful third-party package to do this job. Several functions are used:

***read\_pageHtml(url)***

read the url of an e-book and return the html code

***read\_title(url)***

read the url of a title of an e-book and return the title name

***storageToLocalFiles(storagePath, data)***

Store the e-book content in .txt file in the work directory

***download(i)***

Replace the number index with integer I and return the content

***\_\_main\_\_***

A main function to execute. A proper size of for loop is used to download .txt files

**Bugs and fixes**

There were two bugs in the .txt crawling programme: 1. *TimeoutError: [WinError 10060]*, the programme failed to connect the server since there was no response *2.Encoding error,* .txt file could be download in the local directory but the system failed to decode it. The first bug is quite easy to fix: unstable network or some uncontrollable reasons will lead to this problem, and frequency ratio of this bug was not frequent(ration = the number of times of appearance of this bug/the total number of times of downloading = 0.2), We can use **try and except** to jump out of the current loop if it failed to connect the server. The second bug was a little bit hard to solve. It was due to the encoding failures (Figure 4). We tried to use ‘utf-8’ and ‘gbk’ coding to download the files but nothing works. Therefore, we decided to use a programme to eliminate files which have wrong coding.

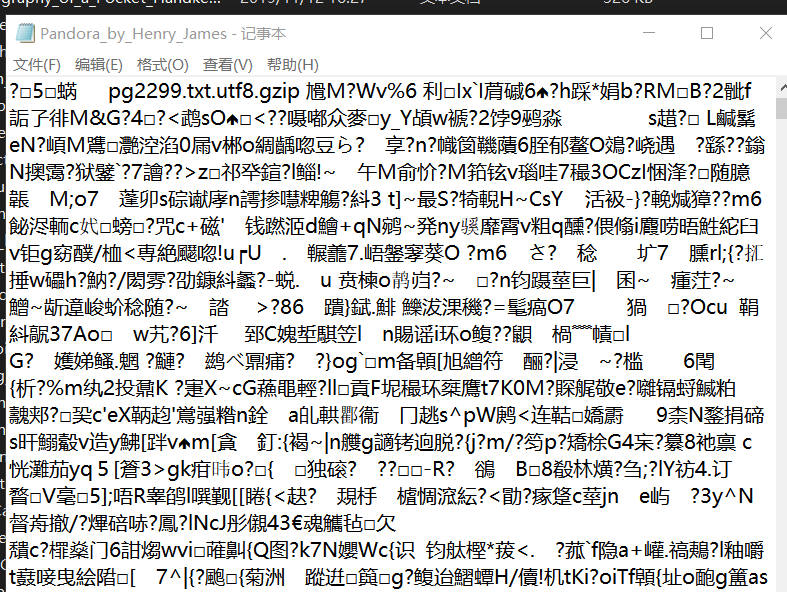


Figure 4 The encoding failuer of .txt files

In the file cleaning.ipynb, there are several line of codes to do this job. The basic idea is to read the first line of each .txt file, if the file is wrong encoded, an exception will be thrown. Then we just need to use os.remove() to remove this .txt file

**results**

Total 649 .txt files. Space 276MB.

**Evaluation and possible improvement**

Generally speaking, we consider our way to collect the data files is acceptable. Since the .txt files are small and the size of total files should not be smaller than 200MB, lots of files will be collected. Therefore, a manual collection is not ideal. An auto crawling and a system of both exception catcher and error eliminator is good. However, the speed of downloading files is too slow. And an excellent search engine should have the ability of collect data and update files dynamically. After consideration and discussion, we come up with serval methods to solve the above problems:

1. Use python multithread to complete the crawling job. Put all the jobs into a queue and use multithreading to process them one by one, it should speed up.
2. Set another programme to update the files periodically. For example, in the data collection server, there should be a programme runs every hour or every day (a proper interval) to start our .txt files crawling programme. Then the index server will run at the same time. The setting of interval should depend on the type of websites. If the website is a social media website, the update of information is rapid and the collection server should have a frequent collection on the data. If the website updates slowly, for instance, the movie website, the interval could half day or one day. Overall, a balance between the interval and the time of information update should be keep. A mathematical model can be built to get the optimal solution.

**(II)Indexing and ranking**

1. **indexing**

Inverted index files were created to give us a simple way to return all the files that are correlated with the input key word. Table 1 gives a sample about the appearance of inverted index file. We conducted three different way to complete this task.

Table 1 Sample inverted index file

|  |  |
| --- | --- |
| word | Documents |
| the | Filename 1, Filename2, Filename 4, …… |
| cow | Filename 2, Filename 4, Filename 5, ……. |
| says | Filename5 |
| moo | Filename 7 |

**python-based indexing**

Our group choose python as the programme language to create the inverted index file {word: [file name lists]}. The general idea is simple. First, we create a dictionary to store the pair of key – value. For each file, used regular expression to find the set of all the vocabularies. Simply words only contain English letters (capital or lower) are considered “true words”. Then, to check whether each word in the current set of words is in the keys of the dictionary. If it is, then append the current file name to the list of value of corresponding key. If it is not, then just set the value of the key is an empty list. Following gives the pseudo code:

*dictionary = empty dictionary*

*For file in files:*

*set\_of\_word = set of words that only contain English letters*

*for word in set\_of\_word:*

*if (word is one of keys in the dictionary):*

*dictionary[word] append the current file name*

*else:*

*dictionary[word] append an empty list*

*return dictionary*

If the number of files is M, and for convenience, every file contains N “true words”. Then, apparently, the time complexity of the algorithm is O(M\*N). The code were put in the *get\_index.ipynb*.

According to the experiments, for large text files, N=649, the total cost of time is 24.1s (create the dictionary) + 3.63s (write to the .txt file) = 27.73s, cost another 13.1s if we transform the result .txt file to a .txt file that dictionary readable. And for small text files, N = 30, it cost 0.796s + 0.19s = 0.986s, cost another 0.468s if we transform the result .txt file to a .txt file that dictionary readable.

**Hadoop-based indexing**

Map reduce programme diagram gave us a better way to complete this task. Assigment 4 Task 2 already allowed us to have a good experience. We do not have to modify the code. To count the total time cost, we set a variable *start* in the beginning of the programme and a variable *end* in the end of the programme. Both of them are to record the current CPU time. The subtraction of these two variables is considered to be the time cost of our programme.

With the help of mapreduce.For small scale of file, we pick 2 .txt files as our test sample. The total running time is 1208(ms). Figure 5 is the capture of the result of our small sample. For large sample N=649 files, the total time cost is 1631(ms).

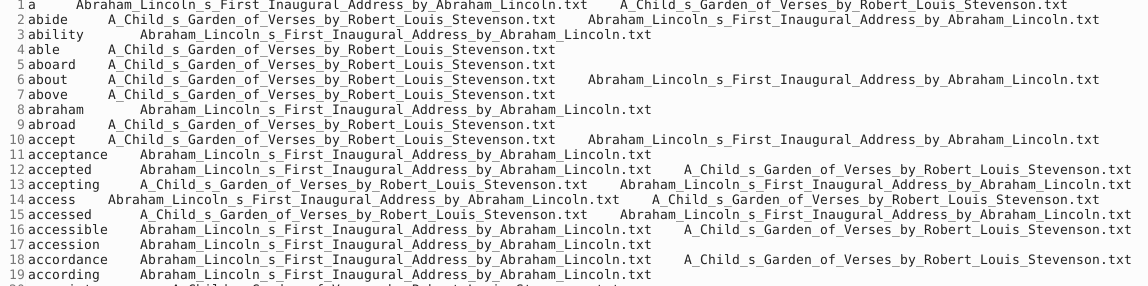


Figure 5 The capture of result of smalle sample

Compared to the python programme, the speed on processing small files does not increase. However, the speed on processing big files is the 1/27 of the original speed.

**Hadoop-Lucene-based indexing**

For the lucene part, at first, we download the lucene-4.8.1 from the official website. Then, we build the lucene environment on the terminal in the Ubuntu. Modify the profile to change the classpath and test the lucene demo. (Figure 6)

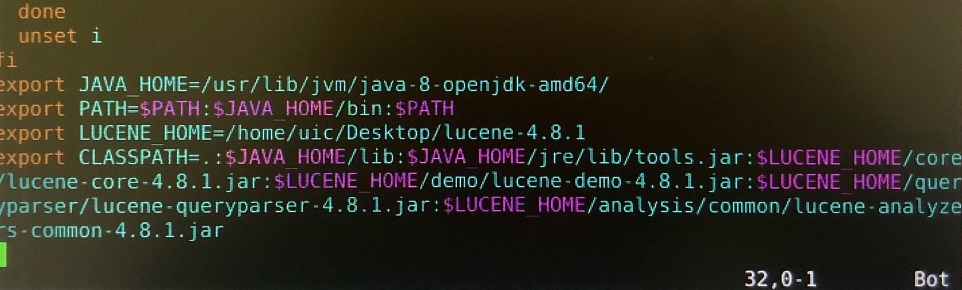


Figure 6 Lucene configuration

When we finished this part, then we can design our method in the eclipse. By using the mapreduce method, we add the searchfile() into the mapreduce method and use the lucene from the lucene method files to index.

First, we import in the necessary lucene’s jar file from the internet, use the mapper and reducer to handle the file after lucene.

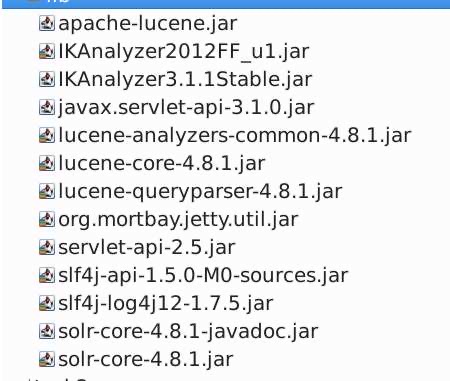


Figure 7 Jar package

Firstly, we set up the lucenefirst class to deal with the file in the home desktop to get the text after seting up the index. Then, we use the mapreduce method to deal with the file after lucene indexing. Finally, we got the output like this from the Hadoop.

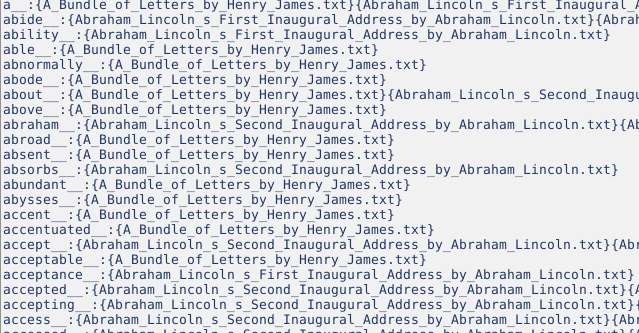


Figure 8 sample index results

For the run time is 1344ms.

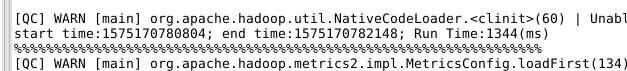


Figure 9 Run time

1. **ranking**

It is not enough if we only have the inverted index. A good search engine should return the results according to the relevance to each file. A file which has high relevance with the input key word should be placed on the top of the page. It is reasonable. Therefore, a rank algorithm on the index file is needed. Since the type of output data is text file, the TF-IDF algorithm could greatly help us complete this task. We used Hadoop to calculate the inverted index files with TF-IDF value in 1322(ms). The form is {[word,filename]:tf-idf value}.

Since the rank of file names of each specific word should be ranked, we used python to help us rank them. First, we used *pd.read\_csv()* to convert the file into a data frame. Then, use *groupby()* to sort the tf-idf value in the descending order of each word. After the ranking, the tf-idf value was useless, the whole columns could be dropped. Last, we must change the data frame to the form of inverted index, we use groupby() again to compress the total file names of a word into a single list. Each row should be {word :[file names list]}. The code is in the file tf\_idf.ipynb. In order to speed up the search engine. the inverted index file should not be output as a .txt file directly. First, we transform the data frame to a dictionary. Then we use *write(str(dictionary).encode('gbk'))* to write the dictionary into the .txt file. The .txt file contains the whole dictionary and *eval()* could directly read the whole dictionary if we open it in the Django as a global variable.

**(III)Web server and interface**

**Framework**

In order to combine the web-front and the index server to perform a real search engine, we use Django framework to implement our idea. Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Despite of those common configuration files in Django, the following is a brief introduction of our specific files:

**static/..**

Includes necessary .css file, pictures, all of .txt files, big.txt and a index file

**templates/..**

a homepage.html of homepage and a result.html of search results

**view.py**

homepage(request)

a corresponding function of homepage.html

result(request)

a corresponding function of result.html

spell\_correction(post\_data)

a function to check the spelling of input keyword and correct it if it is wrong

get\_meaning(post\_data)

get the English explanation of input word from the dictionary

get\_synonyms(post\_data)

get the English synonyms of input word from dictionary

return\_result(post\_data)

return all the file names and lines numbers of input key word

read\_and\_load(file\_name,post\_data,row)

print the all the filenames and its linenumber and context on the html page, highlight the keyword.

fill\_context(i,context,namelist,post\_data)

pass the file name and lines number to the read\_and\_load function each by each

**Web interface**

**图片包含 室内, 餐桌, 杯子, 咖啡

描述已自动生成**

Figure 10 homepage

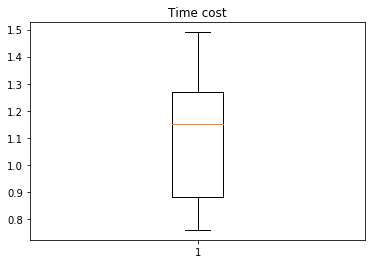
图片包含 屏幕截图

描述已自动生成

Figure 11 main page of search results

**Features and explanations**

a.high speed



Above figure is the box plot of time cost of our search engine. We randomly choose N=30 words as the input keyword and recorded each time cost. The data may be not quite accurate, however, it gives us a straight view of the speed of our search engine. Noted that the first time cost of input word is slightly larger than average time cost because the Django need to load our index dictionary and big.txt first.

The performance of our search engine was not as well as it shows now. The first version of our search engine cost at least 30s to return all the results. There are three important ways to reduce the time cost: **(i)** Turn the index .txt file to a global dictionary variable in Django. As we all know, if we want to search a key word in the big index file to acquire its corresponding file names and line numbers, time cost is O(n). But if we transform our index file to a dictionary, it will cost O(1) time each time. Since the way store the data in a dictionary is use hash table. We use python to achieve our goal. The dictionary is stored in the *big\_index\_dictionary\_tfidf\_2.txt* file, *dic = eval(f.read())* is used to read this .txt file and transform it to a dictionary. It is not convenient to introduce all the procedure of how we did it, but all the code files are given in the code files. **(ii)** Use *multiprocessing.pool.ThreadPool.* It is obvious that if we want to print all the related files names and its context of the keyword, a for loop must be used in the programme. But when a common key word is searched, there must be lots of files and the total time cost will increase. Therefore, we use multilprocessing to accelerate the speed of for loop. *pool = ThreadPool(processes=20)* indicates that we use 20 processes to accelerate the programme. **(iii)** If we want to print the context of the input word in a specific file, the programme will read line by line to search the keyword. The best time cost is θ(1) and the worst time cost is θ(n),where n is the total number of lines of the file. In our project, n ≈ 10000. Therefore, we use package *linecache* to “jump” to the direct line where the input keyword appears in the first time. According to our experiments, the time cost of “jumping” to 10000 lines is just few million seconds.

b. input correction

图片包含 屏幕截图

描述已自动生成

Figure 12 input correction

It is very common that users spelled wrong words. A good search engine should recognize it and return the true word. Figure 7 is a demo of how our input correction work. The user typed the word “prohibtionn”, where the correct spelling is “prohibition”. Our system returned the correct word and provided two candidate words as choices to users in case the system judgment was wrong. The basic idea to implement this function is simple:

1.we use python to gain a huge word set(n ≈ 320,000) by get the union of words of all .txt files.

2. To check whether the input word is correct or wrong, we just need to check if the input word is in the set. If it is in the set, then it means it is a correct word; else there are two situations: either the word is spelled wrong or our .txt files do not contain this word! No matter what the situation is, our search engine will not have the result.

3. To return a correct word, we compare the input word with each word in the set of all words, the *difflib.SequenceMatcher().quick\_ratio()* return the ratio of levenshtein distance. We choose the word with highest ratio as our “correct” word. And words have second and third highest ration as the candidate words.

c.highlight the key word and show the context

图片包含 物体

描述已自动生成

Figure 13 highlight and context

Figure8 is a demo of one of results of word “science”. It shows the rank(3) of this file, the line number of “science” first appearance, the context of this word and highlight it. The ranking uses TD-IDF algorithms. The highlight use bootstrap *<span class='badge badge-pill badge-danger'></span>*. It is simple and needs no more discussion.

d. English explanation and Synonyms

图片包含 屏幕截图, 文字, 名片

描述已自动生成

Figure 14 explanation and synonyms

It is good to show the explanation and synonyms for to users who is not a native English speaker or does not have the correct understanding of a specific words. *PyDictionary().meaning()* provides us a way to get the meaning from the dictionary and wordnet.synsets() returns us a list of synonyms.

e. Auto-complete from historical records

图片包含 屏幕截图

描述已自动生成

Figure 15 historical records

It is good to provide users with historical records if the users need to type the same word repeat. *<form method="POST" autocomplete="on" >* could help us implement this function easily.

**Evaluation and possible improvement**

There are several advantages of our search engine. First, we delivery the result quite fast with average time cost 1.2s, and we also provided lots of features that mainstream search engines like Baidu and Google have. But there are few things that we could make improvement:

1. The paging. A good search engine must have paging as one of the most important function. Not only it could provide users better interface and comfortable views, but it also reduces the stress of server and the total time cost significantly.
2. searching based on phrases. Sometimes users need to input a long phrase because it could reflex what the users actually need, but our engine does not support it.

**Conclusion**

If we could give a mark on our system, we believe our system worth 85 points at least. We do have an auto file collection system. The python codes clearly satisfied the need of the system. It could a large amount of text files with a perfect exception processing mechanism. The process of obtaining the inverted index files and rank them based on ranking algorithm is rapid and error-free. About the integration, the design of homepage and result page is impressive and reasonable. Users could get a comfortable experience while they are using our system. The most important point is that we reduce the time cost significantly compared to peer’s work. We creatively transformed the inverted index file to a global dictionary in the system so that each access of input word cost O(1) time. To print the context around the input word, we used *linecache* to iteratively search the keyword line by line without costing lots of time since the mechanism is different from ordinary method. Finally, the use of multithread speeds up the huge for loop. The result is the average cost of time is 1.2s while the total number of files is N=649 and larger than 200MB. In addition, multiple functions are provided in our system. The input correction smartly changes in the wrong spelling word into the word has the highest similarity and provides two candidate words for uses. The English dictionary makes the non-native English speakers feel more convenient. The highlight of the key word and context make the users have a straight understand of the role of the keyword in each relevant file. The historical record allows users get rid of amnesia.

Definitely, there are some cons in our system. A read search engine should be dynamic. A dynamic data collection should be included. We already come up with the idea that how to perform this function by create an auto-running programme in the back-end periodically. Next, the efficiency of indexing could be improved. Lucene or anther third-party package provides a steady and efficient way to complete this task. About the framework, Django is useful for small scale server but will become useless if there is a huge amount of users. And there are many main-stream search engine frameworks such as solr could be used. Also, the paging function could also be added into the front-end because not only it could provide convenience to the users but also reduce the stress of our poor server.