DS 3003 Data Processing Workshop II

Semester 1 of 2019-2020 Project Report

Group 9:

XvXiangyang m730026126

Zeng Xuan m730026143

Zhu Xiaolin m730026165

# Background:

From last semester, we have already learned about how to crawl a simple webpage by python. Also, in this semester, we have learned the knowledge of Hadoop and know how to use mapreduce to do indexing and ranking. Combing other techniques, we need to get into practicing to create a retrieve text webpage.

# Objectives:

Create a retrieve text webpage. Recommend the files given by the word, based on the sorting of argument to show relevance, for example, TF-IDF.

# Task contribution:

|  |  |
| --- | --- |
| member | Task contribution |
| Xv xiangyang | 1. Download data 2. HashMap 3. Mapreduce 4. Rank. |
| Zeng Xuan | 1. Use Hash map of Java to implement the index part(I failed and Xu Xiangyang solved it finally) 2. Use Python to implement the secondary sorting according to the TF-IDF value 3. Help ZhuXiaolin to do Django |
| Zhu Xiaolin | 1. Crawl static data from “ebook” website and write in the file by using python. (Data hasn’t been used in our project.) 2. Develop the front end webpages. 3. Use python Django to catch the post data, connect html files and views files, and to read the static files to render the webpages. |

# Implementation:

1. Text files collection.

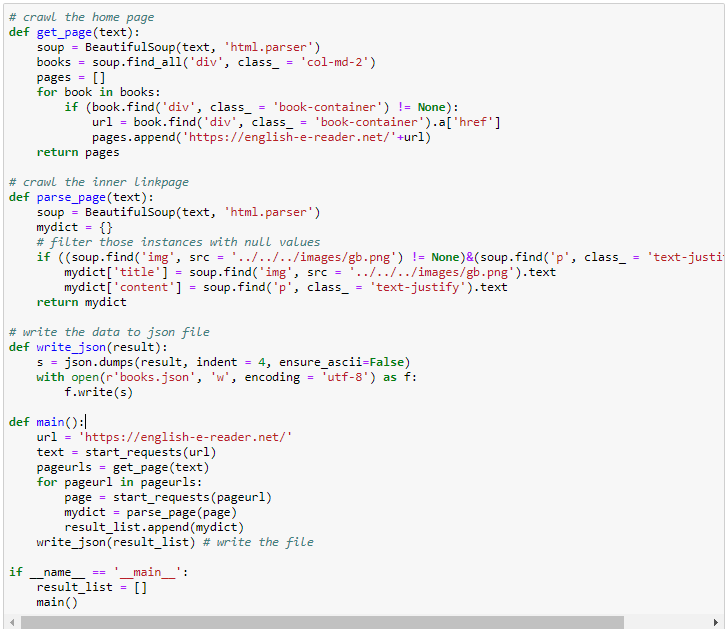
Two methods we used to get the data in our project.

1. Download txt from <http://novel.tingroom.com/>

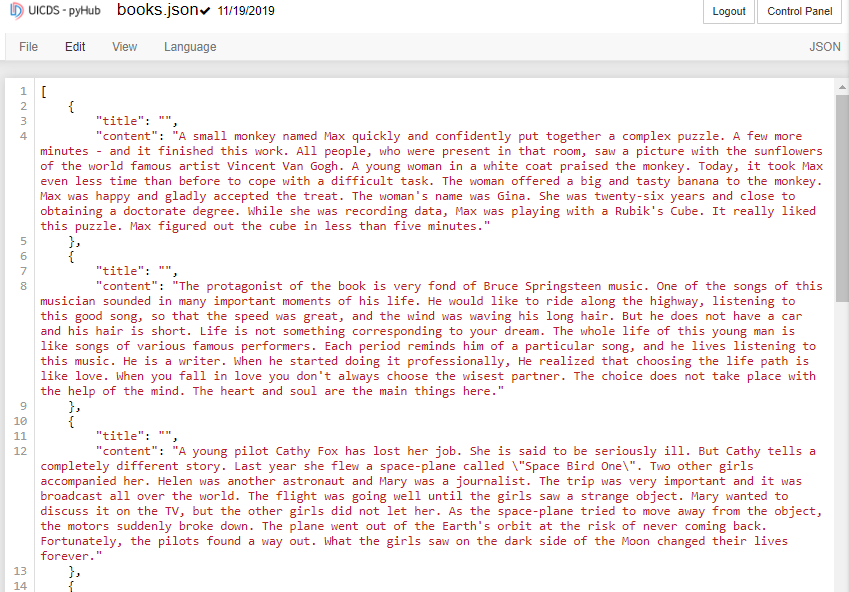
https://archive.ics.uci.edu/ml/datasets.php?format=&task=&att=&area=&numAtt=&numIns=&type=text&sort=typeUp&view=table.

1. Crawl static data from “ebook” website:

Code:

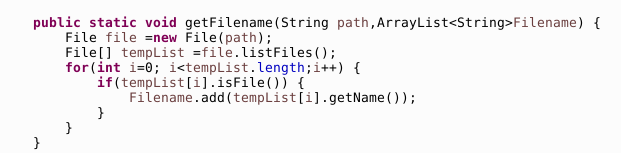


Result:

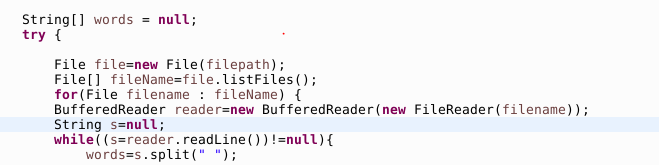


1. Indexing and ranking.

For first part—HashMap, is was really same with mapreduce.



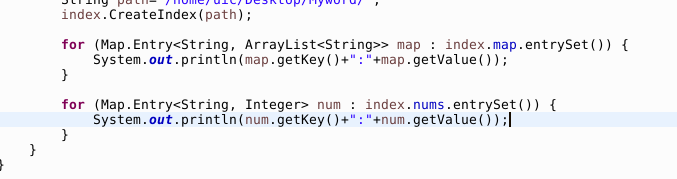
In this code, we can get the filename



in this code, we use bufferedreader to store and read all the file. And store it in the word

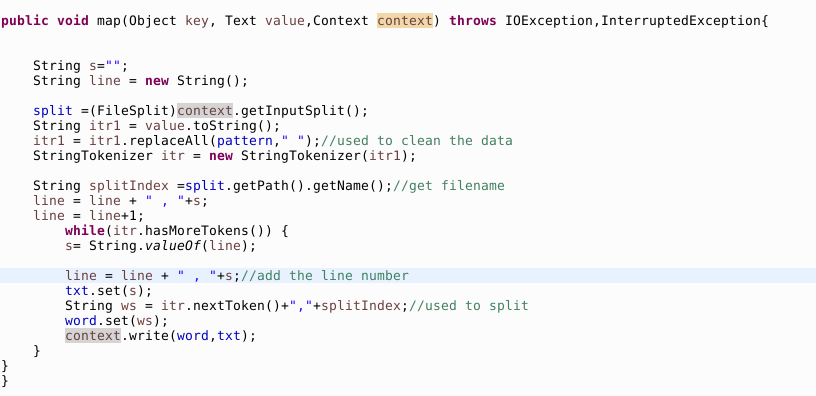


Use this code, for each word in string, we will check whether it was in map, if not, input it.

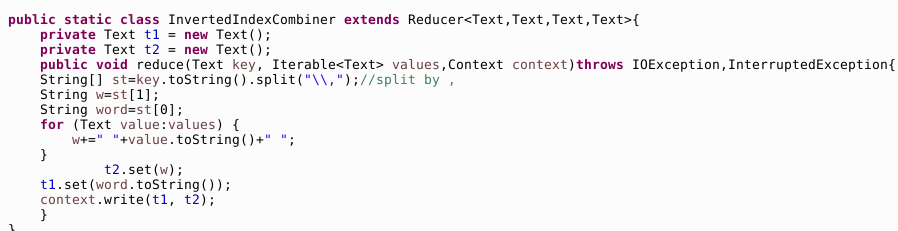


Use this code, we can get result:

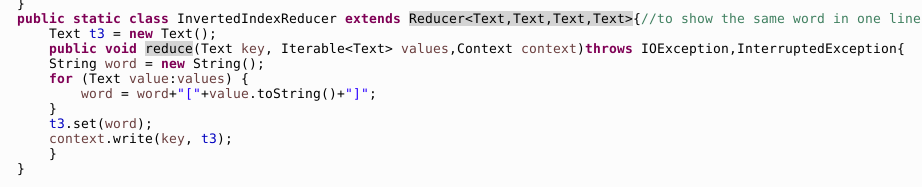
**In mapreduce part.**



In this code we can get filename and words.



In this code we select the filename and put it with line number together



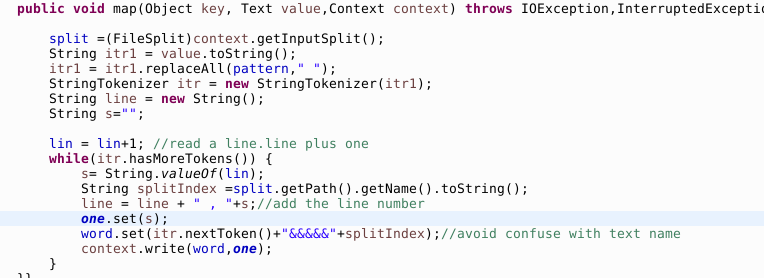
In this code, we let word be key and filename and line number be the value.

**In rank part:**

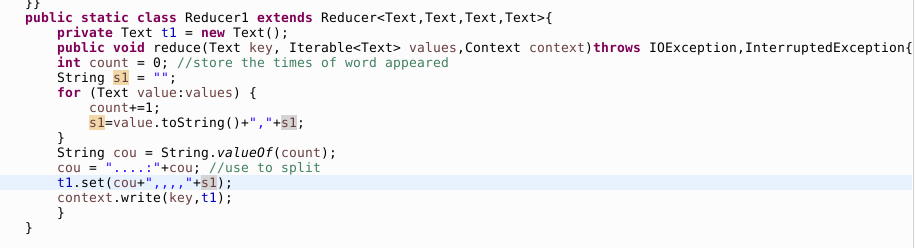
I create two class to get the rank, mapreduce class and TFIDF class.

For mapreduce class:

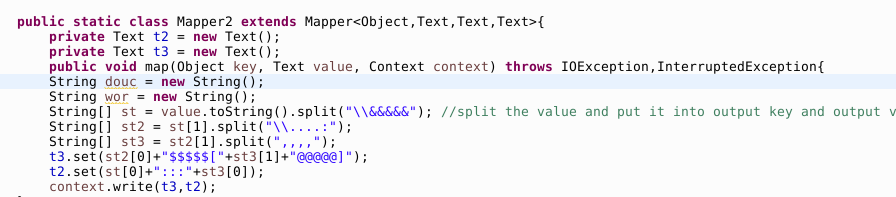
At the beginning, we create method “map”. In map we deal the data, replace the word we don’t need with blank. Read all the text name



We create reducer1 to get the times a word appeared.



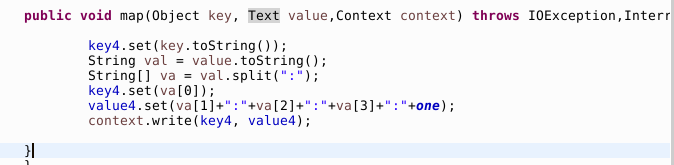
In the mapper2, we can get the line of word



Now we get all the information form mapreduce class

For TFIDF class,

In map method ,we can split word from result of mapreduce class to get information.



In Reducer3, we will get the number of word appear, and the number of all words to calculate rank.



The rank part is really slow.

1. Web server and interface.

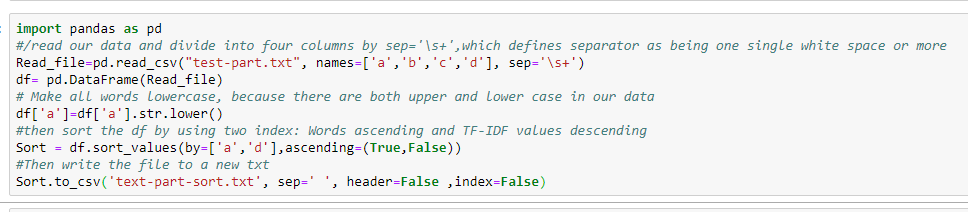
Steps:

1. Store the related files

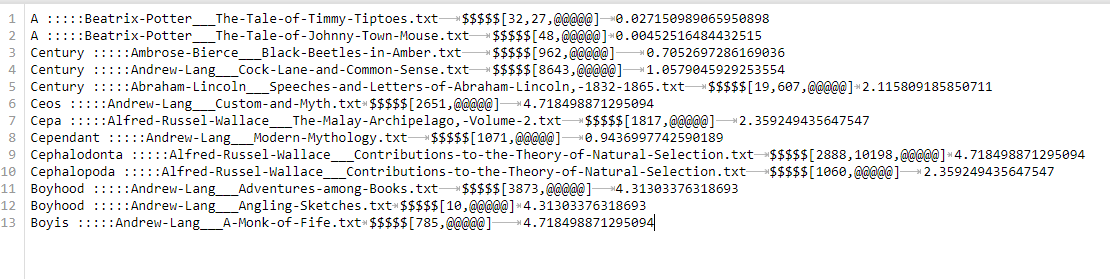
We stored text files downloaded from the webpages or crawled by using python, which we used in step 2. Also, we need to save the results obtained from step 2 (data.txt), including the information of the file name, indexing, TF-IDF values for each word.

Here we used a tricky way to sort our data by descending TF-IDF values. That is to use python for a secondary sort.

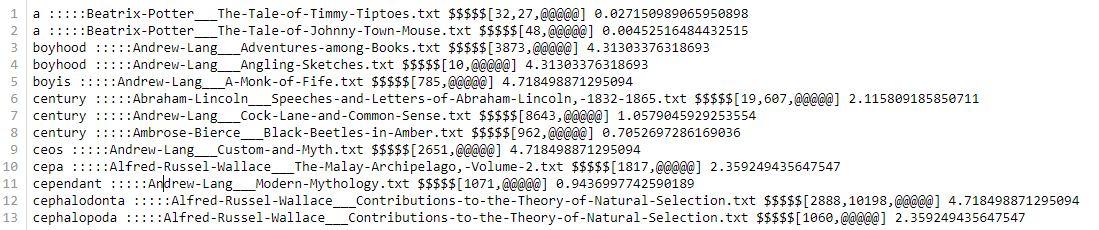
Here is the code:



Here is the training data“test-part.txt”,which is a small part of our data I selected.



After running the code above, I got my sorted data. The result is shown below:



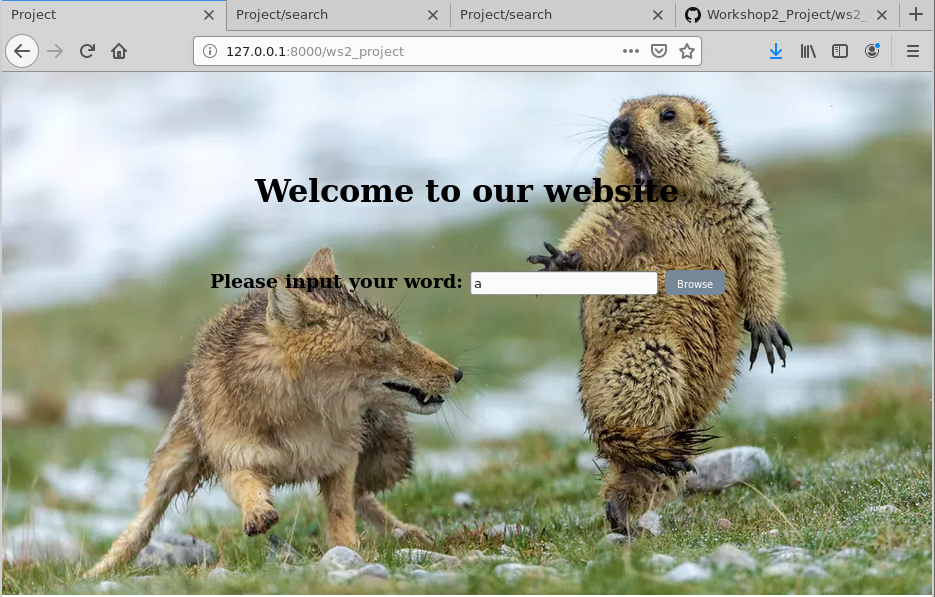
Then we can get our final data set (>200M) by the way.

1. Create the html and CSS

As requirement for this task, we need three html files, first one for the home page (index.html), second one for searching the articles (search.html) and the third one for showing the content of the article (details.html).

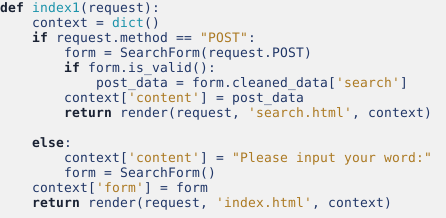
We also connect a CSS file to add the background and set the style of our webpage.

Result:

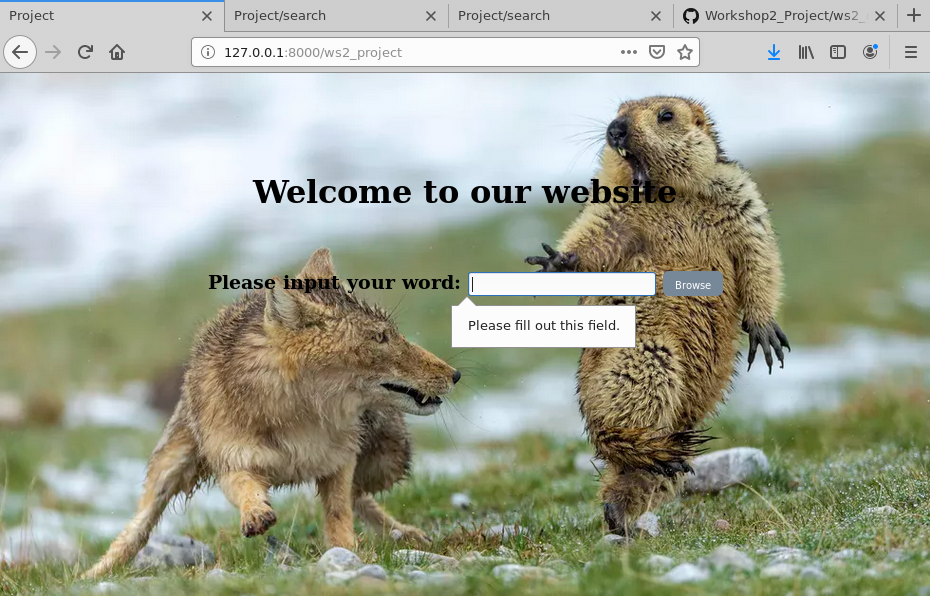


1. Create a new function in views.py (views.index1) to get the post data and render web page. Create a new path for the home page, related to views.index1.

Since we modify the form.py (required=true), it will return a reminder if we input an empty word.



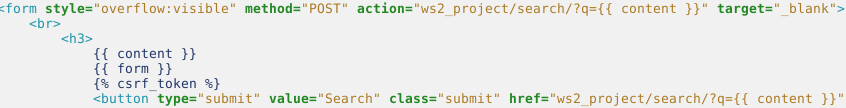
Result:



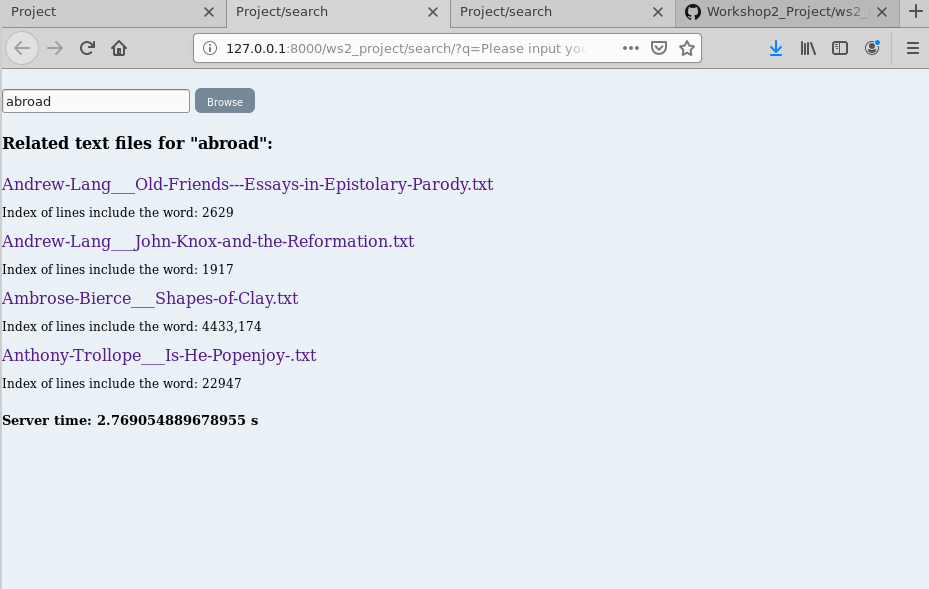
1. Create a second function in views.py (views.search) to restore the post data. From reading the data.txt file, we used a for loop to find out the post data and return the file names and index of lines in each file. Meanwhile, we recorded the start time and end time to calculate the total sever time for every running.

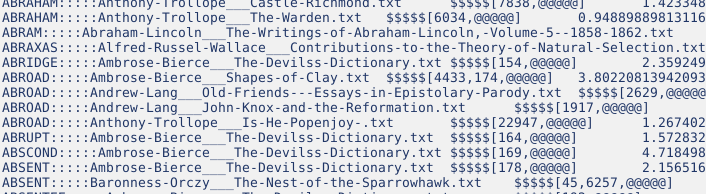


Finally, update the hyperlinks in related html file to jump to this page whenever clicking on the button. Create a new path for the searching page, related to views.search.



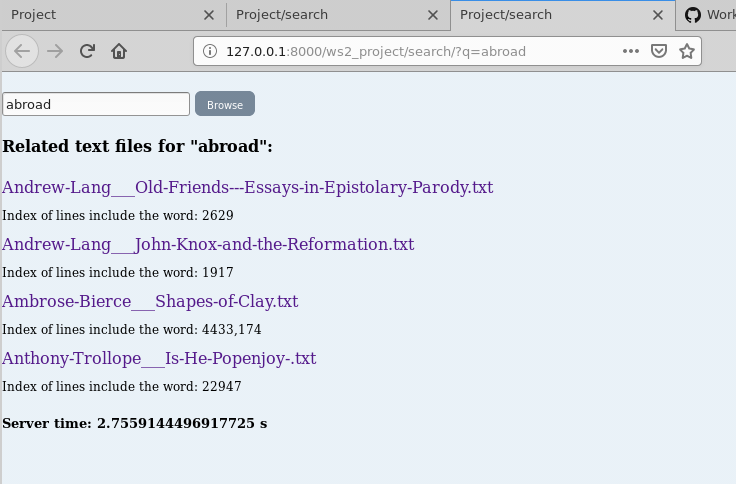
Result:





1. In the next step, we want to keep the word the user input as a variable store in the path. So, we modify the urls and hyperlinks to execute storing the variable.

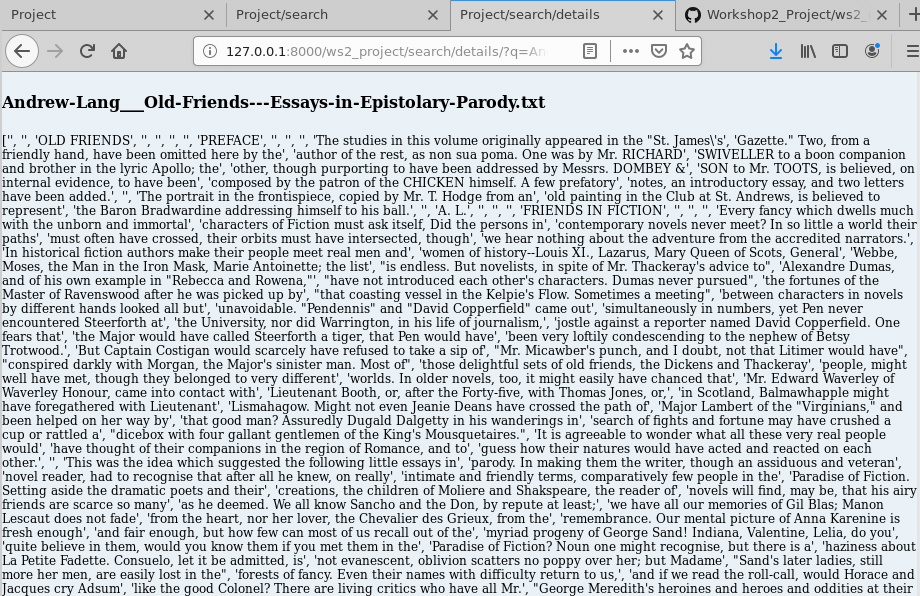
Result:



1. Create a third function in views.py (views.details) to render the webpage for details. From opening the file using its path, we can print the text content in our detail webpage. Create a new path for the details page, related to views.details.



Results:



# Advantages:

1. Both upper case and lower case word can be read to search for the results.
2. For searching page and detail page, we can update each url with storing the word or file name as a variable, for example: …/search/?q=abroad.
3. Whenever we want to search a new word in current page, click the browse button we can jump to a new webpage rather than cover the current page.
4. Reminder will be thrown when click the browse button with an empty input.
5. The results also return index of all the lines in each file include the required word.
6. Total time can be seen easily, labeled at the end of the results.

# Problems and difficulties:

1. Real server time and the computed time have a gap from returning all the results, since we record the end time before returning and render the webpage.
2. As we assume that every time user might search a single correct word, we haven’t addressed the cases with multiple words for one search, or with a word not exists in the file.
3. Since we open the txt file with its local path, once we upload our file to GitHub, it will not be successful to open the file “/home/uic/ws2\_project…”.
4. For some other improvements in Django: to output the context around the text pattern, and highlight the target word in the file.

# Conclusion:

Overall, in this project, we learned how to crawl the raw data, then use hashmap to replace mapreduce. And also can use TFIDF to get the rank of word ，and finally to render the webpages to give the relevant results. Meanwhile, we usually will cost a lot of time in all these three procedures, if we run the server dynamically. However, since we only use static data to store in our file, we consume time only on running Django. Still at the same time, we think to verify a sever system is good or not, time consuming is the principal judgement. In addition, tolerance for wrong typing or diverse input is the second judgement for a good project. So, in our project, we think we still need more contributions on these kinds of improvements.