ITEC 4020

Assignment 3

Group 3 submission

Group Members:

James Chui

Zixiao Yang

Yat Long Lam

Taige Guo

**Contents:**

1. [INTRODUCTION](#h.lbnlrne5g7y)
2. [FILE PRE-PROCESS](#h.lintn2c5mr2t)
3. [INDEXING](#h.9iaiu8velz75)
4. [SIMILARITY SCORING & RANKING](#h.1xh1f7a1oxl8)

**REPORT**

# INTRODUCTION

We are given a single dataset file in text format which contains 11303 records. Out of these records, each believed to be a real web page crawled and saved with a unique document number. By given another set of queries in the similar format, we are able to calculate the similarity between each query and target document, and then rank the top 1000 most related documents by the similarity scores.

# FILE PRE-PROCESS

Initially, we need to index all the documents into a “bag of words”, and some “house clean” work must be done because the substantial file space is taken up by html tags, meaningless words, punctuations, URLs, number and scramble codes, etc. So we list up some items need to be processed:

* HTML tag striping
* Link removal
* Stop words removal
* Number removal
* Stemming
* special characters removal, mind the hyphens “-” (recommended) (i.e. @ $ ^)

However We found it is quite a challenge to process the file due to the file size, because any single step of file process would take up to hours and yet to consume a lot of system resources. As result, when the file processing is running, we have to stop and wait till it finished. To avoid wasting time on program coding and running, we decided to separate some processing work from Java programming as “pre pre-processing”. Simply with Notepad++, by implementing appropriate regular expression formulas we need to work on the html tags, numbers and special characters. Below are some technique we use for the work:

* For html tags, we want to keep <doc>, <docno> and <html> as they can be used for parsing in Java program. This means we need to be cautioned that we are eliminating most but not all of the html tags, for example, if we want to remove <br><p></p><img><b></b><div></div>, then we wrote **<\s\*/?\s\*(br|p|img|b|div)\b[^>]\*?>** this expression will find all the tags mentioned above and replace them with nothing (delete).
* We realized all the contents within <dochdr></dochdr>are useless, so we want to find the tags and delete everything within them. The regular expression for finding is **<dochdr[\s\S]+?<\/dochdr>**
* To find and remove all non ASCII code (special characters), the RegEX is **[^\x00-\x7F]**
* To final all URL, we use **(http.\*[a-zA-Z].\*\.[a-zA-Z]{2,4})t**

The main problem of applying regular expression is that there are always something out of your expression patterns, meaning with regular expression it is very difficult to find everything we want to remove. Another problem is time consuming, each finding also can take long time to run due to the file size. We understand webpage can contain anything in the real world, and it is almost impossible to filter out all useless information. Nevertheless, we do not have to remove all the useless information because with an extremely large volume of document, minor error won’t affect the final search result.

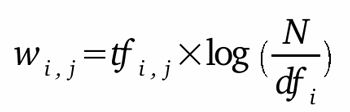
Moving on, we have our input for the real pre-processing in our search engine. Up to now, we have been taken care of html tags, links, numbers and special characters. The dataset file size shrinks from 416MB down to 235MB which is almost half the size of original file. Because of limited heap space when running such huge dataset on java, we have decided to use readline to chunk the dataset into smaller pieces. That way it is more efficient and you could notice errors (if any) at a much earlier stage rather than having to wait for a while and find out the program was at fault and incomplete. In our project, the file size for query topics in our case is smaller, we could parse it all at once, In contrast, it is best to have the same consistency on how we process the documents for maximum efficiency. *Jsoup* was used for data processing because documents can be parsed even if they include informal or improper tag. Moving forward, we loop through our parsed documents and load the dataset and query file, into our Java program for stop words removal and stemming. To do that, we get the text of each document from the chunk files, then we split the texts in *“string array”* with the removal of special characters and meaningless numbers. Next, we used *Weka isStopword()* to look for words that are irrelevant. And lastly, *Terrier Porter Stemmer* performs stemming process on the remaining text as the final step before indexing. The purpose of stop words removal is to eliminate all high frequency words with no significant meaning and have no effect in information retrieval, such as “the”, “a”, “is” and so on. The purpose of stemming is to improve the recall and to reduce the size of the indexing structure by suffix removal or stripping of the word in English to set the words back to the stem. For example, “teacher”, “teaching”, “teaches” and “teach” will all be reduced to “teach”.

# 

# INDEXING

After stop words removal and stemming, we output the dataset via hashmap to count frequency for each term. The following is the vector space model with calculated TF-IDF measures:



To have TF-IDF, we have the equation that

* tf is the still term i frequency
* IDF is Inverse Document Frequency as where N is the total number of documents in the dataset (11303) and is the number of document containing term i in dataset

With the above calculation, we then have document represented as follow example:

WTX013-B14-470 ( 1, 1, 2, 1, 1 ...)

# SIMILARITY SCORING & RANKING

In order to calculate the cosine similarity, we focus on the TF-IDF weight in the document. As previously described, we have a number after every term representing the TF-IDF measure of the term and we call that weight. An example of how the program calculated the cosine similarity score is as follows:

For example, we are comparing QueryX and DocY where

QueryX = {tokyo 1, japan 2, earthquake 2, people 1, police 1, tsunami 2, rescue 1}

DocY = {tokyo 1, japan 2, earthquake 2, people 1, police 1, tsunami 1}

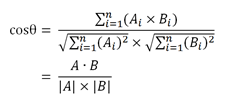
Please note that the DocY comes with TF-IDF measure, and for simplicity purpose we use integers in our demonstration.

We can easily get vector files by basically eliminating all words and characters and have all weight numbers instead as follow:

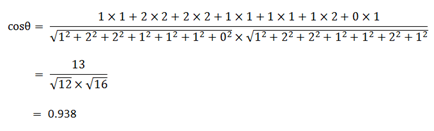
QueryX = {1, 2, 2, 1, 1, 2, 1}

DocY = {1, 2, 2, 1, 1, 1, 0}

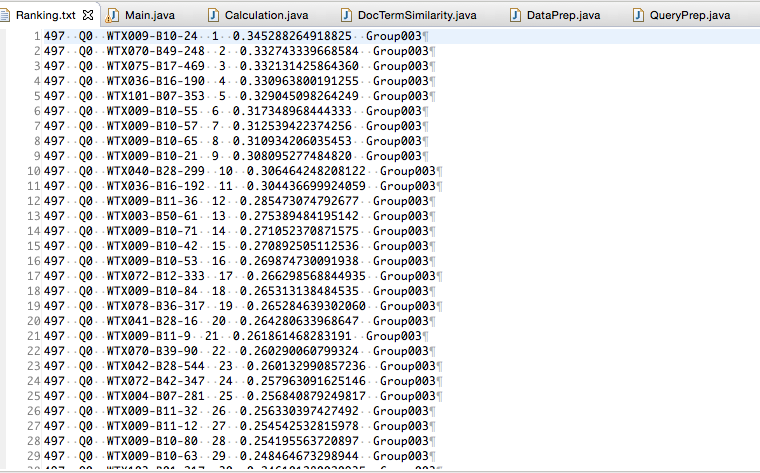
The formula below is to calculate the cosine similarity between two objects:



After formular transition, we can calculate the cosine similarity between QueryX and Doc1 as follows:



The output of Calculation.java is saved on a text document named Ranking.txt which is our final output with predefined format:



README HERE

Technology Requirements:

1. Eclipse Luna EE IDE
2. Dataset
3. QueryTopics
4. Java APIs and Libraries: Jsoup, Weka, Terrier Porter Stemmer, Commons-IO

Please follow the following steps to run the program:

1. Create the following folders in the package directory (4020G3A3): chunk, data\_index, query\_index
2. Name dataset “Dataset.txt”, and queries “QueryTopics.txt”
3. Copy and Paste them in the home directory of your downloaded package folder (4020G3A3)
4. Import your package into Eclipse by clicking File at the left top corner, and select import from the menu, click import, select existing projects into workspace, and next, then select 4020G3A3.zip at select archive file and select the project contain within. Click finish to complete the import process.
5. Right click on your project, and select Properties, click on Java Build Path, go to the libraries tab, add jars by referencing to our libraries included in the lib folder in our package, click apply and ok.
6. If your computer has low java heap memory space, add “-Xmx8G” to extend maximum memory by selecting the little arrow next to the run button -> click on run configuration -> click on the argument tab -> copy and paste at the argument at the VM argument session -> click apply -> if you are already at the main class click run, otherwise click ok
7. Run “main” class without any server as it is a static java application
8. Result file will be outputted in the home directory named “Ranking.txt”
9. Have a happy summer vacation! And give us a good grade!