Assignment #1

Junhan Liu 7228243

YingGin Guan 7285769

CSI4107

***Contribution***

Junhan Liu: Adapting ir.vsr package, creating IR system program, debugging, safety testing, running time testing, optimizing, script evaluation, report.

YingGin Guan: Creating IR system, safety testing, debugging, optimizing, safety testing, report.

***Before you run***

The IR program takes four inputs from users.

The inputs are:

>the given microblog tweet file path;

>an empty folder path to store all the preprocessed tweet text files, in this case, there will be 45899 independent tweet files written to the folder;

>the given 49 tweet query file path;

>a text file path to store query result;

Be careful when passing the second input, **REMEMBER** to have an slash “/“ at the end of your folder’spath. That is to indicate the program to write into the folder instead of writing into your root folder or desktop.

The program has a safety guard method to prevent one from forgetting to have a “/“ at the end of the second input path.

If one get a file path error indicating the system cannot not locate the stop words file, go to ir/vsr/Document.java, change the path to your local stopwords.txt path, recompile Document.java.

***How to run***

The name of the root directory is “a1\_CSI4107”. In the folder, there is a folder named “4107\_IRSystem”. This is our IR program. This file is already an eclipse project, and already successfully compiled using both eclipse and command line. That means you won’t need to recompile it if running in command line.

**The main method** is in *a1\_CSI4107/4107\_IRSystem/src/IRsystemStart/SystemStart.java.*

The program has been tested in command line in both mac OS and windows, there should not be any errors. If there is any unexpected errors, please contact me at jliu187@uottawa.ca.

**In Eclipse:**

The file named “4107\_IRSystem” is already an eclipse project. One can simply import it and run configuration to pass the necessary inputs.

**In Command line:**

As above mentioned, the source code have been compiled in command line. One won’t need to recompile it. Simply locate the 4107\_IRSystem/src folder and run IRsystemStart.SystemStart.java.

This is how to run it in mac:

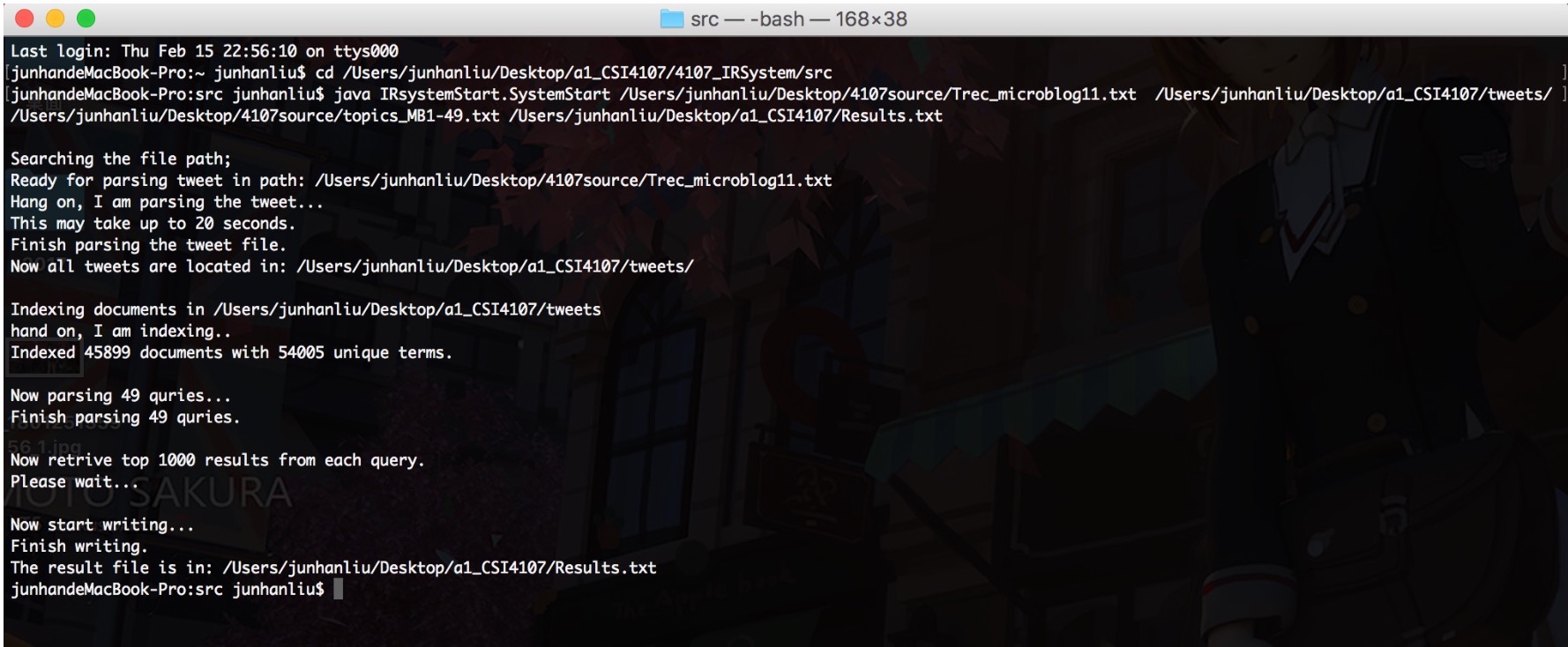
>cd /Users/junhanliu/Desktop/a1\_CSI4107/4107\_IRSystem/src

> java IRsystemStart.SystemStart

/Users/junhanliu/Desktop/4107source/Trec\_microblog11.txt

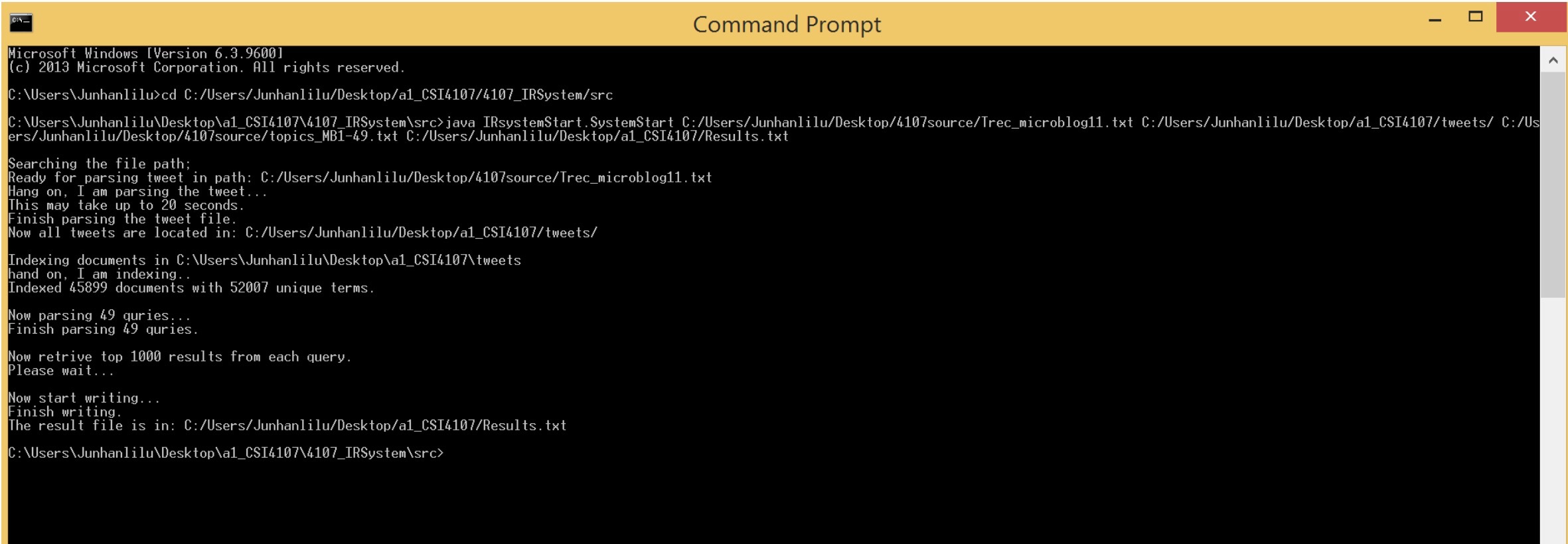
/Users/junhanliu/Desktop/a1\_CSI4107/tweets/

/Users/junhanliu/Desktop/4107source/topics\_MB1-49.txt

/Users/junhanliu/Desktop/a1\_CSI4107/Results.txt

**In windows cmd:**

If running in windows, please passing the input paths using slash “/“ instead of back slash.



***Functionality & Algorithm & data structures***

Our IR system makes use of the existing ir.vsr package the professor provided.

Specifically, we make use of the java class ir.vsr.InvertedIndex, and we adapt it to our IR system design.

The overall logic is as following:

Step 1 preprocessing:

The main program is src.*IRsystemStart* package, this package make use of *ir.vsr.InvertedIndex* class.

This step is done by *src.IRsystemStart.Tokenize.java* and *src.ir.vsr.InvertedIndex.java.*

*src.IRsystemStart.Tokenize.java*

>Input:original tweet document path, an empty folder path to store all the tweet files;

OR

>Input:original tweet document path, an empty folder path to store all the tweet files, boolean remove\_hyper\_link;

*src.ir.vsr.InvertedIndex.java*

*>*Input: tweets document file, which stores all tweets each as a independent text file

We start with the preprocessing part by parsing the tweet document file. In the document provided, there are roughly 45,900 tweets lines. We scan the document line by line and filter out special characters, punctuations and hyper links. Then we create a new text file for each tweet, and store them in an empty folder. When done, all the tweet files will be stored in the second input path folder that a use indicates. The user can pass input boolean remove\_hyper\_link as a third input to indicate they want to remove hyper links or not. The default action is to remove them. Then we pass the folder’s path as input to InvertedIndex.java to generate inverted index and to stem tokens. Tokenizing each tweet file in the folder. Creating tokens for further indexing. The default action is to stem words, but one can indicate otherwise. Tokenization is done by *HashMapVector vector = doc.hashMapVector(),* which the main computation is done by Document.java class. In Document.java, it will check a boolean variable “stem” to make sure whether to stem a token or not, and remove stop words from stop words list using hash table. HashMapVector is a data structure for a term vector for a document stored as a HashMap that maps tokens to weight's that store the weight of that token in the document. A sample of 100 tokens is listed below in the Summary.

Step2 Indexing:

Indexing all the tweets. Generating an index object for further query processing. Retrieving queries and output the tweet id and it’s score. Indexing is done by *HashMapVector vector = doc.hashMapVector(), indexDocument(doc, vector), and computeIDFandDocumentLengths().* Indexing each token and link token’s information with the token itself; storing indexed tokens in a hash map, each token is linked with token information.

Token information contains: idf value and a list of TokenOccurences.

—The idf value indicates how much to weight an occurrence. Tokens that appear in many documents are not very discriminative and therefore weighted less. The idf value is computed by double N = docRefs.size(), which is the total size of the collection; double numDocRefs = tokenInfo.occList.size(), which is the df value of t; and double idf = Math.log(N / numDocRefs) indicates the idf value of token t.

—The list of TokenOccurences contains document id and term frequency.

In InvertedIndex(), we use a document iterator to iterate each document. For each document, we invoke HashMapVector vector = doc.hashMapVector() to create a hash map for each token in the document. The hash map maps each token as the key to its weights(number of occurrence of the token in the document).

Then for each document we index tokens, we put its tokens to the existing index(a hash map<token, tokenInfo>). If the token is already in the index, add its token information to the existing posting list. If the token is new in the index, put the token in the map, and link token information. This is done by calling indexDocument(doc, vector); —> for each <token, tf> pair{ indexToken(token, count, docRef); }. When done, the index(hash map) has been completed.

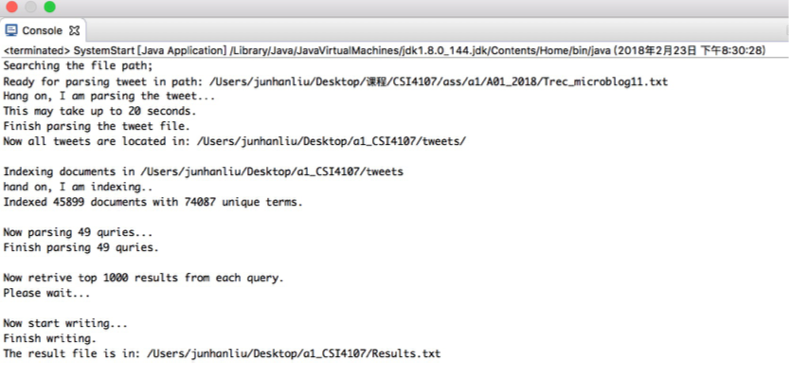
After done indexing, the last step is to compute IDF for each token in the index and document length. This is done by void computeIDFandDocumentLengths(). We loop over the index hash map, for each <token, tokenInfo> entry, we compute its idf using idf formula: idf = log(collection size/df). We put token’s idf into tokenInfo.

When done, the indexing is completed. The index contains <token, tokenInfo> entries.

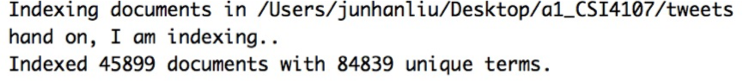
TokenInfo contains <idf, list of document> entries. List of document contains <documentID, tf> entries.

Overall, index is stored in a hash map, which contains tokens and token information(including idf values, document id, and term frequency).

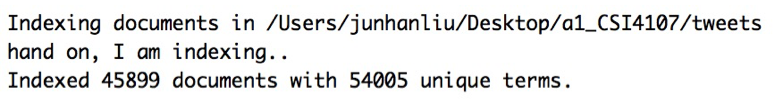
The vocabulary size depends on to stem or not and to remove hyper links or not.

—Without removing hyper links and to stem the vocabulary size is 74087 as it shows below:

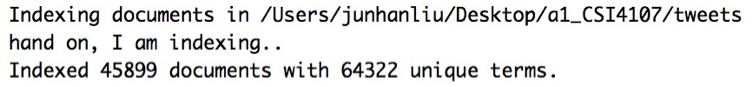
—without removing hyper link and not to stem, the size is 84839.



—removing hyperlinks and to stem, the size is 54005;



—removing hyperlinks and not to stem, the size is 64322.



In *ir.vsr.InvertedIndex.java,* we added several new methods in it in order to adapt our IR system.

What we have added are the followings:

> void printTokens(int number): takes a input counter to indicate how many tokens one want to print in the console;

>List<QueryResult> processQuery(String query): process one tweet query and store the top 1000 query results in a List. The result contains the tweet id and result score.

> List<QueryResult> getRetrievals(Retrieval[] retrievals, int start): private method for processQuery(String query);

We changed some minor details of several methods in the class. All the things we changed to we added have been commented of the format “added:2018-02-11: Junhan Liu” .

Step3 retrieve and ranking:

As above-mentioned, idf is computed in the indexing step. Tf and idf are stored in the tokenInfo object which is part of the index.

After all the documents have indexed, we parse the tweet query file to extract 49 queries. This is done by *TweetQueryParser.java* .

>Input: tweet query file path;

We read the query document line by line and extract each query and queryID. Store them in a List<Query>. Retrieving queries by public List<QueryResult> processQuery(String query), it calls public Retrieval[] retrieve(HashMapVector vector) to do the computations. Retrieval[] retrieve(HashMapVector vector) creates a hash table to store the retrieved documents. Keys are document references and values are DoubleValues which indicate the partial score accumulated for this document so far. As each token in the query is processed, each document it indexes is added to this hash table and its retrieval score (similarity to the query) is appropriately updated. The score is computed by incorporateToken(). The weight of a token in the query is is IDF factor times the number of times it occurs in the query : double weight = tokenInfo.idf \* count. Update the score for this document by adding the product of the weight of this token in the query and its weight in the retrieved document (IDF \* occurrence count) val.value = val.value + weight \* tokenInfo.idf \* occ.count.

Step4 results:

By processing each query, retrieving their results in the last step, now we write all the results in a txt file. This is done by *WriteQueryResult.java*.

>Input: List<Query> query, the index object, the output file path;

We process each query and call methods in InvertedIndex.java to process the query.

After 49 queries have been processed, we write them out to the text file. The file is included in the assignment folder.

***Data Structures***

Data structures in the program:

List<Query>, List<QueryResult> , List<String>, List<Documents> to store all the queries that have been parsed of the format “queryID, query”, maintain query results retrieved in the format “tweetID, score”, store intermediate strings, and store document references. Hash map stores documents to be indexed in the indexing step.

Using Hash table to search for stop words and filtering out the stop words.

HashMap to search for token information, store tokens and related token information, store index information, and to store and search documents’ information.

Using java util.MAP to store and retrieve document references and scores.

Arrays to store final retrieved documents’ ids and scores.

***Optimization***

In the program, we choose to remove hyper links in the preprocessing part. Since URLs added to the index decrease the precision of the results. Also the default action is to stem tokens in our program.

Overall, our program is doing five jobs: tokenizing, indexing, parsing query, retrieving, writing to file. Initially, when we testing the program, we found that the average time it takes to finish the whole job is 54 seconds. We searched reasons unit by unit. We found that the reason it took so long is due to two reasons. First, we are writing each tweet line into a new text file. Second, in the retrieving part (WriteQueryResults.java), we store results in a String, we had a O(n^2) running time, that is about O(49000). We refined the method; and we reached a O(n\* O(1)) running time, that is roughly O(49). By re-testing it, our running time have significantly reduced to an average 11 seconds on Mac OS. When testing in windows, my partner's lap top running time can be up to 50 seconds, whereas mine remains a running time of 10 seconds. It much depends on the speed of the machine.

***Summary***

To remove hyper links, and to stem, we have 54005 unique terms.

**100 tokens:** it is stored in a txt file and included in the assignment folder;

undermining

spilling

abrupt

salary

childs

artieka

bbctbq

avant

blasts

liferichmond

분들께

remax

flfw

wingsuit

timah

chiles

cribsies

unitedkingdom

ballestas

nonviolent

recallcheck

pnsky

ifyouletme

coliseo

matttbastard

glade

sisil

rahmemanuel

stoning

circoncis

twodisc

timao

bschool

panamericana

bindelwijk

soekarnohatta

revdlesley

flip

mssql

taylorswift

mpkirk

posture

échenme

eating

iwebslogcom

oracaopefabio

lickmyjs

glady

songsthatdropthepanties

dropsby

automating

wwwgoogleartprojectcom

information

sinuses

adrian

ofar

prebailout

estudiando

pourrastu

tnfisherman

swapsies

jobshirt

wwwglobalcamnl

analytics

hotairblog

myriad

arrojado

glace

irrelevance

tambunan

windburn

shakespear

ª

informative

pagewrite

nagwoworry

remek

categorically

º

scholars

remet

chapelle

jozan

tcells

wowzer

remed

offs

benjetzrt

srry

salama

motive

shrinks

berlinspandau

swishing

schubart

bruceleroy

cabbies

colouring

à

offc

***First 10 results of query 1 and 25***

1 Q0 30051954050727936 1 0.83556 ohlala

1 Q0 33823403328671744 2 0.83556 ohlala

1 Q0 30198105513140224 3 0.81201 ohlala

1 Q0 30260724248870912 4 0.79202 ohlala

1 Q0 30162553262841857 5 0.76051 ohlala

1 Q0 29978962599870465 6 0.76051 ohlala

1 Q0 30496268895657984 7 0.72277 ohlala

1 Q0 29983478363717633 8 0.69587 ohlala

1 Q0 30236884051435520 9 0.6715 ohlala

1 Q0 30319208176820224 10 0.66902 ohlala

25 Q0 33207460051292160 1 0.64239 ohlala

25 Q0 32165530513178625 2 0.61584 ohlala

25 Q0 31849003209461760 3 0.61584 ohlala

25 Q0 31286354960715777 4 0.54998 ohlala

25 Q0 33609772196560896 5 0.5342 ohlala

25 Q0 31320463862931456 6 0.51106 ohlala

25 Q0 31738694356434944 7 0.50127 ohlala

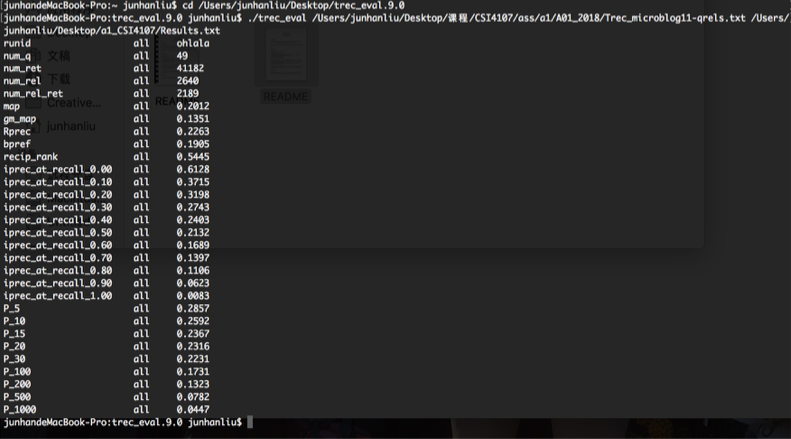
25 Q0 31773184512495616 8 0.49459 ohlala

25 Q0 31823291815567361 9 0.46976 ohlala

25 Q0 32563584391188480 10 0.46451 ohlala

***Evaluation***

Using trec\_eval\_script to evaluate out results.

Run the script in command line, locate the script file, type in make. Then type ./trec\_eval and inputs path.

Our MAP is 0.2012, p\_5 is 0.2857, p\_10 is 0.2592.

***Discussion***

As mentioned before in step2 indexing, we have tested several cases as:

1, to remove hyperlinks and to stem

2, to remove hyperlinks and not to stem

3, not to remove hyperlinks and to stem

4, not to remove hyperlinks and not to stem

Our program is running using option 1.

1,

map all 0.2012

P\_5 all 0.2857

P\_10 all 0.2592

2,

map all 0.1925

P\_5 all 0.2449

P\_10 all 0.2551

3,

map all 0.1756

P\_5 all 0.1878

P\_10 all 0.1898

4,

map all 0.1707

P\_5 all 0.1796

P\_10 all 0.2061

As the statistics indicate, removing hyperlinks increases our precision and stemming have a roughly same effect on the precision.