a)

Team Member: Bo Wang (bw2450)

Team Member: Hua Tong (ht2334)

b)

A list of all files we are submitting:

Main.java

BingSearch.java

FormatString.java

Order.java

RemoveDuplicates.java

Rhashtable.java

SortMap.java

Stopwords.java

TermNode.java

UserInterface.java

XmlAnalyser.java

columbia_search_result.txt

gates_search_result.txt

musk_search_result.txt

commons-codec-1.10.jar

README.pdf

c) How to run:

compile:

javac -cp commons-codec-1.10.jar *.java

run:

java -cp commons-codec-1.10.jar:. Main bingKey precision query

sample run:

java -cp commons-codec-1.10.jar:. Main NzspjS8F9PrxrRVM8NnCbNtVPKuSH9WGA6RC6jIKmAQ 0.9 columbia

My bingKey is: NzspjS8F9PrxrRVM8NnCbNtVPKuSH9WGA6RC6jIKmAQ

d)

We define a node structure called TermNode as follows

String name	Term name
int termFreq_r	Term frequency in all relevant documents
int termFreq_n	Term frequency in all non-relevant documents
int docFreq_r	Number of relevant documents containing this term
int docFreq_n	Number of non-relevant documents containing this term
double score	beta*termFerq_r*log(#results/docFreq_r)/#r_documents -gamma*termFreq*log(#results/docFreq_n)/#n_documents
List<String> prev	Adjacent terms before this term in all relevant docs
List<String> next	Adjacent terms behind this term in all relevant docs

(1) Use a hashmap<String, TermNode> as the data structure to present vectors. We choose the term as the key and the corresponding TermNode as the value stored in the hashmap.

(2) Use Bing API to get the url, title and summary of ten results returned from Bing and combine them as a List<String>. Then according to uses' feedback, these ten results are divided into rel_title_summary and nrel_title-summary.

(3) For terms in relevant documents, we update its TermNode variable: termFreq_n to record its term frequency in the relevant documents and then store the TermNode in the hashmap with the term string as the key. Thus, we get a hashtable with all terms in relevant documents.

(4) Remove the duplicate terms in each relevant document. And then calculate the number of relevant documents containing each term in the hashmap and update its TermNode variable: docFreq_r.

(5) For each term in relevant documents, we record its two adjacent words as previous and next in two lists. To accurately record adjacent words, we used punctuations and newline sysmbols to separate sentences, so that our program will only see two words as adjacent if they are in one sentence.

(6)For terms in the hashmap, we traverse all non-relevant documents and update its TermNode variable: termFreq_n to record its term frequency in the non-relevant documents.

(7) Remove the duplicate terms in each non-relevant document. And then calculate the number of non-relevant documents containing each term in the hashmap and update its TermNode variable: docFreq_n.
(8) Calculate the score of each term in the hashmap by the formula derived from Rocchio algorithm as follows:

score = beta*termFerq_r*log(#results/docFreq_r)/#r_documents

-gamma*termFreq*log(#results/docFreq_n)/#n_documents

(9) Except the terms already in the query, select the top two terms with highest score. Then for each new terms T to be added to the query, we check in sequence from the first word to the last word in the old query. For each word W in the old query, we pull out its two adjacent lists to see the number of times T appears in these two lists. If the TW appears more frequently than WT, we will arrange them as TW in the expanded query, vice versa. If T is not in neither of the two adjacent lists of W, we move on to the next word of W. If T is not in all adjacent lists of all words in the old query, we add T at the last of the query. We repeat the above process for the two words to be added to the query at each iteration.

e)

Algorithm for selecting expanded words

Our algorithm is based on The Rocchio algorithm:

$$\overrightarrow{q_m} = \alpha\overrightarrow{q_0} + \beta\frac{1}{|D_r|}\sum_{\overrightarrow{d_j}\in D_r}\overrightarrow{d_j} - \gamma\frac{1}{|D_{nr}|}\sum_{\overrightarrow{d_j}\in D_{nr}}\overrightarrow{d_j}$$

where q_0 is the original query vector, q_m is the modified query vector, D_r and D_{nr} are the set of known relevant and non-relevant documents got from users' feedback. This algorithm is based on vector space model. We denote each document by the vector with one component in the vector for each term in that document. The value of each component is the tf-idf weight of that word t in the document d.

The basic idea is to move the initial query vector q_0 toward the centroid of the relevant documents and some distance away from the centroid of the non-relevant documents to get our modified query q_m. Then we will choose two components with the top two highest tf-idf except the existing components in q_0. The corresponding words for that two components will be our expanded words for a single iteration.

Algorithm for ordering words in the expanded query

As we know, the order of the words in the expanded query is also important for the information retrieval results. We use the following algorithm to find the best order automatically.

For each term, we record its two adjacent words as previous and next in two lists. Order the words in the query by placing the new query word before/after an old query word if it is placed before/after that word in relevant documents.

(f)

My bingKey is: NzspjS8F9PrxrRVM8NnCbNtVPKuSH9WGA6RC6jIKmAQ