**Advanced Database System**

**Project 1 Query expansion**

Chaozhong Lian (cl3190) & Jinxi Zhao (jz2540)

Catalog

**1. Team member1**

**2. File List1**

**3. How to run2**

**4. Bing Account Key2**

**5. Internal design of our project3**

**6. Detail of our query expansion algorithm6**

**7. Other heuristic attempts And why we don't use them4**

**8. Other interesting test cases using our algorithm4**

1. **Team members:**

* Chaozhong Lian (cl3190)
* Jinxi Zhao (jz2540)

1. **File List:**

* run.sh (the shell script to run our program)
* Makefile (the makefile to run our program)
* ./src (the folder containing the source code)
* commons-codec-1.9.jar, org-json.jar (the libraries needed to run Bing API)
* htmlparser.jar (the library to analyze a webpage, extracting the real content in "body" tag, ignoring any words inside tag declaration. Google open source library )
* all other files are generated by Eclipse, and are not useful to run our program, but for maintenance purpose, we include these files

1. **How to run:**

There are **two ways to run our program**, one is using run.sh, one is using "make" command

1. Using Makefile

Type in the following command in shell:

"make ACCOUNT=<value> PRECISION=<value> QUERY=<value>"

* The <value> following ACCOUNT= should be the Bing search account key
* The <value> following PRECISION= should be a real between 0 and 1
* The <value> following QUERY= should be a list of words in double quotes

Examples:

* make ACCOUNT= cnLsEsvYTSd+XBWkE4lO7z02Wgh3W14UTAwgJ/JURdc= PRECISION=0.9 QUERY="snow leopard"
* make ACCOUNT= cnLsEsvYTSd+XBWkE4lO7z02Wgh3W14UTAwgJ/JURdc= PRECISION=0.9 QUERY="gates"
* make ACCOUNT= cnLsEsvYTSd+XBWkE4lO7z02Wgh3W14UTAwgJ/JURdc= PRECISION=0.9 QUERY="columbia"

1. Using run.sh

Type in the following command in the shell:

"./run.sh <account key> <precision> <query>"

* <account key> is the Bing search account key
* <precision> is a real between 0 and 1
* <query> is a list of words in a double quotes

Examples:

* ./run.sh cnLsEsvYTSd+XBWkE4lO7z02Wgh3W14UTAwgJ/JURdc= 0.9 "snow leopard"
* ./run.sh cnLsEsvYTSd+XBWkE4lO7z02Wgh3W14UTAwgJ/JURdc= 0.9 "gates"
* ./run.sh cnLsEsvYTSd+XBWkE4lO7z02Wgh3W14UTAwgJ/JURdc= 0.9 "columbia"

1. **Bing Search Account Key**

The Bing search account key is:

**cnLsEsvYTSd+XBWkE4lO7z02Wgh3W14UTAwgJ/JURdc=**

1. **Internal Design of our project**

All the packages and .java files are listed follows:

Please see detail description for every class and package below:

│

├─bing

│ BingSearch.java

│ Result.java

│

├─expansionAlgorithm

│ │ Algorithm.java

│ │

│ ├─rocchio

│ │ │ Rocchio.java

│ │ │

│ │ └─sortAlgorithm

│ │ BM25.java

│ │ MaximumTf.java

│ │ NormalTf\_Idf.java

│ │ SortAlgorithm.java

│ │

│ └─termProximity

│ ProximityEval.java

│

├─indexer

│ IndexCreator.java

│ Posting.java

│ PostingNode.java

│

└─praser

UrlPraser.java

**As a general description, the workflow of our program is:**

**BingSearch** class captures the query user wants to make, and query to Bing API, and then storing each piece of result into a Result class. **BingSearch** class also interacts with the user, confirming which of the results are relevant, if the precision is reached or precision is 0, then we start the following process: **BingSearch** call on **IndexCreator** to create a inverted index file for these 10 webpages, the **IndexCreator** will for each result, call on **UrlPraser** to analyze and extract the real content in the body tag of each webpage and also filtering out stop words, and then building a inverted index file.

Then, **BingSearch** call on **Algorithm** class to expand the query, **Algorithm** class first calls **Rocchio** class, **Rocchio** class will calculates the two terms with the highest score and then return them to the **Algorithm** class. Because **Rocchio** is just a frame, it need an corresponding algorithm to calculate the weight of each term, and so, here we examine three algorithm, the cosine algorithm(**NormalTf\_idf** class), the maximum tf algorithm(**MaximumTf** class), the BM25 algorithm(**BM25** class). These three scoring algorithm can be switched to use one another just by changing a line in the **Rocchio** class(line 53: SortAlgorithm sa = **new** NormalTf\_Idf();), after our experience, we discover that **NormalTf\_idf**(cosine) is the best among all three, so the submitted program is using this algorithm.

After the two terms with the highest score is returned to the **Algorithm** class, it calls the **ProximityEval** class to find the best two place to insert these two terms, **ProximityEval** will changes the query String directly, and return the modified query back to the **BingSearch** class, going to the next round.

**Below is the detail description of the functionality of each package and class:**

* bing package

This package contains the major functionality for interacting with user and getting result from the Bing API.

* BingSearch.java

Class, this class acts as the entry point of our program. Also it encapsulates the operations to retrieve result from Bing API, and storing the result in an ArrayList of Result object.

* Result.java

Class, this class encapsulates the attribute of a retrieved webpage from Bing, its member variables are: ①url, title, description ②whether this webpage is relevant ③its length(count of words, used in BM25 scoring algorithm) ④normalized Sum(i.e., will be used to calculate square root), used in cosine(NormalTf\_Idf scoring method, MaximumTf scoring method) ⑤maxTf, the frequency of the term that appears most frequently in this page (used in MaximumTf scoring method)

* praser package

This package contains classes that will analyze the content of a given url(i.e. a result returned by Bing).

* UrlPraser.java

Class, this class encapsulates the operations to analyze the content inside a url's "body" tag. It will extracts all the words that is real content and then return an ArrayList of String. The String ArrayList is then processed(eliminate duplicates but recording the total repeat count) and returned to the IndexCreator class.

* indexer package

This package contains the classes related to building the inverted index file.

* IndexCreator.java

Class, encapsulates the operation to build the inverted index file. The inverted index file is a HashMap<String, Posting>, String is the word, and the Posting is a class representing the list of documents that has this word.

* Posting.java

Class, represents the list of document that has a specific word.

* PostingNode.java

Class, represents the a document that belongs to a Posting, this will includes a pointer to the Result class(pointing to the document), an ArrayList<Integer> storing the positions which this words shows in this document(the position list will be useful in deciding the order of the words in the ProximityEval class).

* expansionAlgorithm package

This package contains the classes to calculate the two terms to be added and reorder the terms in the best order.

* Algorithm class

This class acts as a linkage between Rocchio and ProximityEval, it calls the Rocchio class first to get the two terms to be added, and then pass those two terms to ProximityEval to calculate the best order, and return to BingSearch an expanded and reordered query String.

* expansionAlgorithm. rocchio package

Rocchio related algorithm, to calculate the two terms to be added.

* Rocchio.java

This class implements the Rocchio algorithm, it is a frame of Rocchio algorithm, because Roccio algorithm does not define how we should calculate the weight of each score, we implements its frame, and examine 3 weight scoring algorithm, cosine(normal tf idf), maximum tf and BM25 to complete the Rocchio algorithm.

* expansionAlgorithm. rocchio.sortAlgorithm package

Implements the scoring method used in the Rocchio algorithm.

* SortAlgorithm.java

Interface, all three other classes in this package will implement this interface, it define how the Rocchio algorithm will call the weight scoring algorithm.

* BM25.java

Implement the BM25 algorithm to be used in Rocchio algorithm.

* MaximumTf.java

Implement the maximum tf algorithm to be used in Roccchio algorithm.

* NormalTf\_Idf.java

Implement the normal cosine algorithm(tf×idf) to e used in Rocchio algorithm.

* expansionAlgorithm. rocchio. termProximity package

This package contains classes that expand the query and then reorder the query in the best order.

* ProximityEval.java

As stated earlier, this class will takes the original query(query from the previous round) and the two terms needed to be inserted. Then it calculates the best position to insert this two terms, and return to the Algorithm class the expanded query string.

1. **Detail of our query expansion algorithm**

Strictly speaking, our expansion algorithm only consists of two parts. The Rocchio algorithm, combined with scoring method, is used to select the two best terms to add into our query, and also the Proximity Evaluation algorithm, which is used to calculate the best place to add in these two best terms.

But only using the above two algorithm to expand query is not enough, the terms that are chosen are very likely to be useless words. We need to do a lot of pre-processing, filtering out stop words and meaningless single letter words. Also, due to our observation, passage in wikipedia.org, because their "Reference" sections, always contains junk words, if we are not careful enough, we will add many words in the "references" section in wikipedia.org, for example, "retrieved" from "retrieved from" appears in basically every references.

In considering all the previously mentioned problems, we discuss our query expansion algorithm in three parts. Firstly we discuss the Rocchio algorithm and the scoring method we choose, and then we discuss the Proximity Evaluation algorithm used to decide the optimal order, and finally we will discuss what we should be careful in the pre-processing stage, how we avoid adding junk words.

* **Rocchio Algorithm and its corresponding scoring algorithm**

After we have build our inverted index file(notice that we omits the pre-processing stage here, we will discuss it shortly), using the set of results which are labeled relevant or not, we can then choose the two terms to add into our query.

Recalls that the rocchio algorithm is as follow:

①

The main characteristic of this algorithm is that in order to compute the query in the next round, we add all the vectors belongs to the documents that are relevant, and minus all the documents vector that are not relevant. Because we don't need to delete words from the original query, only adding words, therefore we can ignore in this equation.

And so the only thing we need to do is, for each word that appears in relevant documents, we add their weights in the relevant documents, and also subtracting out their weights in the non-relevant documents. So we will have the score of each term:

②

After calculate the score for each term, we select the two non-negative score terms which has the highest score, all the terms with negative or 0 score will be ignored. We do this by maintaining a two element array bestTerm[2] to only storing the two best terms. For the value of β and γ, we use the recommended value suggested in chapter 9 in "Modern Information Retrieval: A Brief Overview", so β=0.75, γ=0.15.

But there is an important part missing here, that is, how do we calculate the weight of term t in a document (i.e. how do we calculate )? In here, we consider three weight scoring algorithm, BM25(one of the most successful probabilistic model algorithm), Maximum tf and cosine(normal tf idf). After our comparison, we conclude that using the cosine method(normal tf×idf) yields the best answer. So in our program, we use cosine(normal tf×idf), but we have left the completed BM25 method and Maximum tf method in our program, anyone who want to try the result of these two method could easily do that just by changing line 53 in Rocchio algorithm, in default this line is (line 53: SortAlgorithm sa = **new** NormalTf\_Idf();).

**For the detail description of how we tried to use BM25 and Maximum tf in our heuristic attempts, and the detailed comparison of the result of the three algorithm, please refer to part 7 " Other heuristic attempts And why we don't use them ".**

Back to our discussion, when using the cosine method, the equation we calculate the score of each term, equation ② becomes:

③

One small problem is that when calculating the score of a term, we need the weight of all terms, which we still don't know. Our solution is, we first calculate the tf×idf weight of every term in every file, and storing the sum of the square of weight in Result class. Then we run through every terms in every file again, only this time we already knows the sum of the square of weight, so calculating score(t) is a trivial operation now.

**Below is some test cases using our algorithm and which two words the algorithm adds in the first round, to our surprise, 85% of query can finish just in the first round, the below is some of the test cases that finished in 1 round:**