Advanced Database System

Project 1 Query expansion

Chaozhong Lian (cl3190) & Jinxi Zhao (jz2540)

Catalog

1. Team members:

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

2. 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)
- ./bin (the folder containing the generated classes by our Makefile)
- 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)
- Other_test_cases_transcript.pdf (the full transcript of other interesting test cases using our algorithm)

3. 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"

   One additional note to make here is that, ./bin contains all the classes generated by this Makefile, by typing "make clean", all the generated classes will be deleted.

2) 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"

4. Bing Search Account Key

The Bing search account key is:

cnLsEsvYTSd+XBWkE4lO7z02Wgh3W14UTAwgJ/JURdc=

5. 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 ③ its length(count of words, used in BM25 scoring algorithm) ④normalized Sum(i.e. $w_1^2 + w_2^2 + \cdots + w_n^2$, will be used to calculate square root), used in cosine(NormalTf_Idf scoring method, MaximumTf scoring method) ⑤ maxTf, 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.

6. 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:

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

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 $\alpha\overrightarrow{q_m}$ 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:

$$\text{Score}(t) = \beta \frac{1}{|D_r|} \sum_{d_j \in D_r} \text{weight}(d_j, t) - \gamma \frac{1}{|D_{nr}|} \sum_{d_j \in D_{nr}} \text{weight}(d_j, t) \quad ②$$

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 d_j (i.e. how do we calculate $\text{weight}(d_j, t)$)? 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:

$$\text{Score}(t) = \beta \frac{1}{|D_r|} \sum_{d_j \in D_r} \frac{\text{tf}\times\text{idf}}{\sqrt{w_1{}^2+w_2{}^2+\cdots+w_n{}^2}} - \gamma \frac{1}{|D_{nr}|} \sum_{d_j \in D_{nr}} \frac{\text{tf}\times\text{idf}}{\sqrt{w_1{}^2+w_2{}^2+\cdots+w_n{}^2}} \quad ③$$

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:
For the full transcript, please refer to " Other_test_cases_transcript.pdf".

[1] Look for Mac OS X Snow Leopard, using[snow leopard]
Augmented by os apple
[2] Look for Bill Gates, using [gates]

Augmenting by original microsoft
[3] Look for columbia university, using [columbia]
Augmenting by university sipa
[4] Look for Brad Pitt, using [brad]
Augmenting by tv pitt
[5] Look for Penny in "The Big Bang Theory", using [penny]
Augmenting by sheldon leonard
[6] Look for Indiana Jones, using [professor jones]
Augmenting by character indiana
[7] Look for "Sleepless in Seattle", using [seattle annie]
Augmenting by jonah sleepless
[8] Look for Miami Heat(Basketball team), using [heat]
Augmenting by ticket pm
[9] Look for Resident Evil(game), using [resident]
Augmenting by reviews evil
[10] Look for Frozen (2013 film), using [frozen]
Augmenting by anna animated
[11] Look for Harry Potter, using [harry]
Augmenting by rowling voldemort
[12] Look for Eclipse(the software) related information, using [eclipse]
Augmenting by foundation software

- **Reorder List based on Term Proximity**

Below we descript the algorithm originated in the paper [Rasolofo, Yves, and Jacques Savoy. "Term Proximity Scoring for Keyword-Based Retrieve Systems", *Lecture Notes in Computer Science 2633, 1611-3349,2003*][1].

This algorithm takes the assumption that if two words A and B have more than 5 words in between them(i.e. A C D E F G B), then these two words will not have a very strong context meaning, we can then see these two words separately; the closer these two words are, the more meaningfully connected these two words will be. Therefore, for a given word pair t_i and t_j, we compute a term pair instance (tpi) weight as follows:

$$\mathrm{tpi}(t_i, t_j) = \frac{1.0}{d(t_i, t_j)^2} \quad ①$$

$d(t_i, t_j)$ is the distance between the two terms, notice the order is important, we will only consider the situation where t_j is behind t_i, and so when defining $d(t_i,t_j)=pos(t_j)-pos(t_i)$, we will only consider when $0<d(t_i,t_j)\leqslant 5$. The greatest possible value is 1 when the two terms are next to each other, and the smallest possible value is 0.04 when the two terms have 4 words in between.

One simply way to calculating the term proximity score would then be, for each term pair (t_i,t_j) with $0<d(t_i,t_j)\leqslant 5$, summing up all the $tpi(t_i,t_j)$. However, this method

will bias long passage, because long passage will likely to contains more (t_i,t_j) pair, therefore, we should take some method to normalize the documents length.

In the paper, the author combining this algorithm with BM25 algorithm, provide us a way to calculate the term proximity score using normalized passage length:

$$\text{proxi\_score}(t_i, t_j) = (k_1 + 1)\frac{\sum_{\text{occ}(t_i,t_j)} \text{tpi}(t_i, t_j)}{K + \sum_{\text{occ}(t_i,t_j)} \text{tpi}(t_i, t_j)} \quad ②$$

where:

$$K = k_1 \cdot \left[(1 - b) + b \cdot \frac{l}{\text{avdl}}\right] \quad ③$$

As in BM25 algorithm,
k_1 is normally set to 1.2
b is normally set to 0.75
l is the length of this document
avdl is the average length of all relevant documents, we only calculate score for relevant documents.

Therefore, for a given term pair (t_i,t_j), we can now calculate the score $\text{proxi\_score}(t_i, t_j)$ using the algorithm described above.

Now, assume that the two terms to be extended is [e1, e2], and the query is [q1, q2 ... qn], our algorithm will first consider adding e1 into the query, so we first consider term pair (e1, q1) (q1, e1) (e1,q2) (q2,e1) ... (e1,qn) (qn, e1), by calculating the score of each term pair, we then know where we should add our e1 into. If all position to add are 0, then we just append e1 to the last of the query.

The tricky part in how we should add our e2 term, we might just add e1 into the query, and adding e2 into the query. But this approach has a potential problem. Consider an extreme case, where e1 is always 6 words ahead of q1, e2 is always 3 words ahead of q1, then the problem comes, the score of the term pair (e1,q1) will be 0, and then e1 will be appended to the end, then after e2 is added, the query becomes [e2 q1 e1], not optimal. The optimal obviously should be [e1 e2 q1], which can happen if we add e2 first, "connecting" e1 and q1.

The problem above is that we bias e1 and add it first. Therefore, in our algorithm, we first calculate where we should add e1 and the highest score of which e1 can get, then, we don't add e1 into the query, calculate where e2 should be added and the highest score of which e2 can get against the original query. After doing this, we add the expansion term with the highest score, and calculate where the another term should be added against the new query(the expanded one).

This algorithm is based on the observation that the expansion term with the highest score is obviously more related to the original query, and is likely to be a "bridge" to solve the problem we discussed above.

Here is a list of the expanded query of some of the test cases that finish in 1 round (for full transcript, please refers to Other_test_cases_transcript.pdf):

[1] Look for Mac OS X Snow Leopard, using[snow leopard]
Becomes: snow leopard apple os
[2] Look for Bill Gates, using [gates]
Becomes: microsoft gates original
[3] Look for columbia university, using [columbia]
Becomes: columbia sipa university (one document do contains many columbia sipa, and so the result becomes in this order)
[4] Look for Brad Pitt, using [brad]
Becomes: brad pitt tv
[5] Look for Penny in "The Big Bang Theory", using [penny]
Becomes: sheldon leonard penny
[6] Look for Indiana Jones, using [professor jones]
Becomes: professor indiana jones character
[7] Look for "Sleepless in Seattle", using [seattle annie]
Becomes: seattle sleepless annie jonah
[8] Look for Miami Heat(Basketball team), using [heat]
Becomes: heat ticket pm
[9] Look for Resident Evil(game), using [resident]
Becomes: resident evil reviews
[10] Look for Frozen (2013 film), using [frozen]
Becomes: frozen anna animated
[11] Look for Harry Potter, using [harry]
Becomes: voldemort rowling harry
[12] Look for Eclipse(the software) related information, using [eclipse]
Becomes: eclipse software foundation

- **Pre-processing stop words and wikipedia's spam words**

Simply using the algorithm we described above do not yields the optimal result(that we could get), we also need to filter out stop words and spam words from wikipedia.

When choosing the best two terms to add into our query, one important thing we need to be care about is we should ignore all the stop words, otherwise the expanded query would become useless. Therefore, in our program, we maintains an hashmap of stop words, and we will not build the inverted index posting for any stop words, in this way, stop words' score are always 0, and are ignored from consideration. When user query using stop words, our program terminates, because we don't have stop words in the inverted index file.

Because Bing search engine generally rank wikipedia page in a very high ranking, therefore we are frequently meeting with wikipedia webpage, yet wikipedia webpage contains a major problem: the "References" section always contains many spam words that are not relevant to the query terms.

After our observation, the "References" section will contains "retrieved" or "archived" in generally every item of reference, also the months "January" to

"December" appear very frequently.

And so, in our algorithm, when building the inverted index file, for every page, we first changes whether the url is starting by "http://en.wikipedia.org", if so, we ignore words including "retrieved", "archived" and from "January" to "December", we also ignore the year "19xx" to "20xx", we think these years are generally used in the reference section, the years other than this year are in many cases some historic moment and are important.

This algorithm will only ignore these spam words from wikipedia, will not affect other webpage, also, because page in wikipedia contains so many these spam words, therefore ignoring them from wikipedia will only increase our accuracy.

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

➢ Maximum tf, BM25 vs Normal tf×idf

As discussed in part 6, when using Rocchio algorithm to select the best two terms to add into the query, because Rocchio algorithm is just a framework, it needs a underlining weight scoring algorithm to support. In our program, we use the normal tf idf algorithm, in this section, we will briefly discussed what is BM25 and Maximum tf algorithm, and then compare the three algorithm using our test cases.

- Maximum tf

  Maximum tf is a common improvement on normal tf idf, its main purpose is to normalize the tf weights to the maximum tf in that document.

$$w(t,d) = ntf(t,d) \times idf$$

  where

$$ntf(t,d) = \alpha + (1-\alpha)\frac{tf(t,d)}{tf_{max}(d)}$$

  According to the textbook, the suggested value of α is 0.4.

- BM25

BM25 is one of the most successful search engine weighting scheme, it is based on the probability model. The following weighting algorithm is taken from ["这就是搜索引擎 核心技术详解" Chapter 5 "检索模型与搜索排序", *Publishing House of Electronics Inducstry, ISBN 978-7-121-14865-1*][3].

$$w(t,d) = \log\frac{(N-n_i+0.5)}{(n_i+0.5)} \cdot \frac{(k_1+1)tf}{K+tf} \cdot \frac{(k_2+1)qf}{k_2+qf}$$

where

$$K = k_1\left[(1-b) + b\frac{dl}{avdl}\right]$$

N is total number of documents,
n_i is the number of document containing this term,
dl is the length of this document

avdl is the average length of all documents,
k_1 is an experience value, suggested to be 1.2
b is an experience value, suggested to be 0.75
k_2 is an experience value, suggested to be 200
tf is the term frequency in the document
qf is the term frequency in the query

Because we are adding term, not matching query term, so the third part becomes useless, and so our BM25 algorithm becomes:

$$w(t,d) = \log\frac{(N - n_i + 0.5)}{(n_i + 0.5)} \cdot \frac{(k_1 + 1)tf}{K + tf}$$

We implemented all three, and can be easily change to use one another by changing line 53 in Rocchio class line 53: `SortAlgorithm sa = new NormalTf_Idf(); to new BM25() or MaximamTf().`

Below is the result lists are these three algorithms:

[1] Look for Mac OS X Snow Leopard, using[snow leopard]
- Normal tf idf : snow leopard apple os
- BM25: snow leopard feb shipping
- Maximum tf: snow leopard tech quicktime

[2] Look for Bill Gates, using [gates]
- Normal tf idf: microsoft gates original
- BM25: melinda gates technology
- Maximum tf: gates inequities advances

[3] Look for columbia university, using [columbia]
- Normal tf idf: columbia sipa university
- BM25: columbia internship prof
- Maximum tf: columbia sipa marija

[4] Look for Brad Pitt, using [brad]
- Normal tf idf : brad pitt tv
- BM25: actor brad movie
- Maximum tf: brad tuxes jan

[5] Look for Penny in "The Big Bang Theory", using [penny]
- Normal tf idf : sheldon leonard penny
- BM25: sheldon leonard penny
- Maximum tf: penny holding 30th

[6] Look for Indiana Jones, using [professor jones]
- Normal tf: professor indiana jones character
- BM25: professor professor jones jones
- Maximum tf: professor jones religion profession

[7] Look for "Sleepless in Seattle", using [seattle annie]
- Normal tf idf : seattle sleepless annie jonah

- BM25: seattle sleepless seattle annie
- Maximum tf: seattle annie 66th 51st

[8] Look for Miami Heat(Basketball team), using [heat]
- Normal tf idf : heat ticket pm
- BM25: heat dancer ticket
- Maximum tf: heat intro websites

[9] Look for Resident Evil(game), using [resident]
- Normal tf idf :resident evil reviews
- BM25: series resident gamerankings
- Maximum tf: resident apr jul

[10] Look for Frozen (2013 film), using [frozen]
- Normal tf idf : frozen anna animated
- BM25: frozen grossing original
- Maximum tf: frozen partners sorry

[11] Look for Harry Potter, using [harry]
- Normal tf idf : voldemort rowling harry
- BM25: voldemort lily harry
- Maximum tf: harry trademarks universal

[12] Look for Eclipse(the software) related information, using [eclipse]
- Normal tf idf: eclipse software foundation
- BM25: eclipse ide software
- Maximum tf: eclipse rap resources

From the list above, we can clearly see the different between the three algorithms, we can see that in general, normal df idf yields a better result of all three of them, BM25 ranks second, Maximum tf is the worst. In fact, in our experience, 85% query can be finished in 1 round using normal tf idf algorithm, 60% can be finished in 1 round using BM25, 20% can be finished in 1 round using Maximum tf.

Therefore, in our program, we use normal tf idf algorithm as the support for Rocchio algorithm.

- **Considering term proximity when computing weight in Rocchio algorithm**

One idea that will easily pop up during the completion of our project is that "since we are using term proximity evaluation, why don't we incorporate the proximity evaluation when calculating weight score in Rocchio algorithm"?

This thought can be summarized as this: because we use rocchio algorithm to select two best terms to add to our algorithm, maybe we can choose the terms which are close to the searched term. We can then consider both the tf×idf value and the proximity factor.

Yet this approach will have to solve two difficulties, one is that because we are adding new words, therefore we may need to check every word to see their proximity to every query terms to determine their proximity score, this can take exponential time, which is highly impractical.

The previous difficulty can be solved by just calculating the proximity score of the

10 terms that with the highest tf×idf value, in this way, we can avoid trapping into an exponential time problem.

Yet another difficulty remains, that is how do we balance tf×idf score and proximity score? If we are not careful enough, one of the value may becomes the dominant factor, that is, we may bias one factor to another. The solution to this difficulty is to use machine learning to calculate the way to normalize these two factors. But because of the limited time and limited test cases, we don't want to take the risk of biasing one of another factor, and so we didn't use this scheme.

Because of the two difficulties we discussed above, we did not use this method in Rocchio algorithm, since our algorithm now can already generate a acceptable result.

8. References

[1] Rasolofo, Yves, and Jacques Savoy. "Term Proximity Scoring for Keyword-Based Retrieve Systems", *Lecture Notes in Computer Science 2633, 1611-3349,2003*

[2] Lee, Dong-Hyun, "Multi-Stage Rocchio Classification for large-scale Multi-labeled Text data ", *Web.14,Feb,2014,<http://lshtc.iit.demokritos.gr/system/files/lshc3_lee.pdf>*

[3] "这就是搜索引擎 核心技术详解" Chapter 5 "检索模型与搜索排序", *Publishing House of Electronics Inducstry, ISBN 978-7-121-14865-1*