INLS 490-154: Introduction to Information Retrieval System Design and Implementation. Fall 2008.

7. Structured query processing

Chirag Shah*
School of Information & Library Science (SILS)
UNC Chapel Hill NC 27514
chirag@unc.edu

1 Introduction

So far we have looked at how we can process and represent unstructured text collection (indexing), process information requests (queries) from a user, and finally, use a variety of models for retrieving information. We also studied ways to incorporate real or pseudo feedback in probabilistic and language (relevance) models. We will continue this thread to see how we can provide even more control to the user while providing the information request. To be specific, we will let the user express the "importance" of various concepts or keywords in his search request. To do so, we will first look at constructing structured queries, and then see how we can interface with the feedback loop in a retrieval system.

2 Structured queries with Lemur

In a Web search scenario, a user typically provides a bunch of keywords to the search service. It is then the job of the search engine to do its best to match that information request to the collection and come up with a rank list that is *most relevant* to that user for that condition. This is a hard problem and no system, without additional knowledge, can satisfy every user for every situation. What is this additional knowledge? It could be the analysis of query-logs (White, Bilenko, & Cucerzan, 2007), or the search history of a user indicating his activities and interests (Dumais et al., 2003; Teevan, Dumais, & Horvitz, 2005). Even if we do not have all these information, the retrieval can be much more effective if only the user can give us more than a bunch of keywords. For instance, if the user can provide us with the relative importance of the terms that he used in his query, we can use that information to improve our ranking.

Let us see how we can handle query-term weights with Lemur. Code 1 shows how to formulate structured queries, which can incorporate weights. The first query is a simple query without any weights on the terms - assembled using the operator #SUM. Lemur allows us to provide weights for every term in the query, the syntax for which is

`#q<query_id> = #WSUM(<weight1> <term1> <weight2> <term2> ...);`

Query 2 and 3 in Code 1 are weighted queries, but each term weighted equally. Finally, we have query 4, which has different weights for each of the terms.

Code 1: Structured queries

```
#q1 = #SUM(presidential election Bush Kerry);
#q2 = #WSUM(1 presidential 1 election 1 Bush 1 Kerry);
#q3 = #WSUM(0.5 presidential 0.5 election 0.5 Bush 0.5 Kerry);
#q4 = #WSUM(2 presidential 5 election 0.5 Bush 0.75 Kerry);
```

For parsing a structured query as shown above, we need to use *ParseInQueryOp* application in Lemur, which is similar to *ParseToFile*. Code 2 shows an example parameter file for this application.

Code 2: Parameter file for 'ParseInQueryOp'

```
<parameters>
    <outputFile>query_parsed.txt</outputFile>
    <docFormat>trec</docFormat>
    <stopwords>stopwords.list</stopwords>
    <stemmer>krovetz</stemmer>
</parameters>
```

Once we run *ParseInQueryOp* with the name of the parameter file as the first argument and the name of the query file (shown in Code 1) as the second argument, we get output in 'query_parsed.txt', which should have our structured query parsed. Let us use that output on our indexed NYT collection to perform retrieval. Since we had a structured query, we will use *StructQueryEval* for retrieval, a parameter file for which is given in Code 3.

Code 3: Parameter file for 'StructQueryEval'

```
<parameters>
    <index>myindex</index>
    <textQuery>query_parsed.txt</textQuery>
    <resultFile>results.txt</resultFile>
    <resultFormat>3col</resultFormat>
    <resultCount>10</resultCount>
    <retModel>kl</retModel>
</parameters>
```

The output of this retrieval run (in 'results.txt') should have up to 40 lines (up to 10 for each of the four queries). The rank lists for queries 1, 2, and 3 should be identical. This is to be expected as every term in those queries were weighted the same for a given query, and thus, we implied that each term in those queries was equally important. The output of the fourth query, though, should be slightly different. It may still have some of the documents that the other queries have, but they may be in different order, and there may be some new documents. This is because even though query 4 has the same terms, we have provided different weights for them. This changed the matching and ranking process.

We saw how to incorporate terms weights in retrieval process. Now the question is how to obtain such weights. We cannot directly ask the users to provide us real numbers as weights for each of their query terms. What may be a better way of implementing this is providing the users with certain UI elements (e.g., sliders) that can capture such "importance" or weights.

3 Interfacing with feedback in retrieval

Another popular way of incorporating user feedback is providing the user with terms suggestions (Anick, 2003). The idea is to display a set of terms that are highly relevant to user's original query and let the user specify which of those terms he would like to have added to his query for the refinement. We will now see how this can be done by modifying some code in Lemur.

Let us first go ahead and see how we can use *RetEval* for performing a retrieval that incorporates pseudo relevance feedback. In the past, we had used *RetEval* to do retrieval without feedback, and used its output to feel in *RelFBEval* for doing feedback. Such a process involving two separate loops is useful if we intend to manipulate or look at the intermediate output, however, if we do not want to do this, we can simply use *RetEval* with feedback parameters. See Code 4 for an example parameter file that includes parameters for specifying pseudo relevance feedback.

Code 4: Parameter file for 'RetEval'

```
<parameters>
    <index>myindex</index>
    <textQuery>query_parsed.txt</textQuery>
    <resultFile>results.txt</resultFile>
    <resultFormat>3col</resultFormat>
    <resultCount>100</resultCount>
    <retModel>kl</retModel>
    <feedbackDocCount>5</feedbackDocCount>
    <feedbackTermCount>10</feedbackTermCount>
</parameters>
```

The output of running *RetEval* with the parameter file given above is the results of one initial retrieval followed by processing the modified query (pseudo relevance feedback). Now, let us look inside *RetEval* and extract the term information before and after the feedback,

thus enabling us to manipulate them if we wanted. In order to do so, we will use the code snippet given in Code 5.

Code 5: Snippet for printing terms with weights in Lemur

```
lemur::api::TextQueryRep * myRep = dynamic_cast<lemur::api::TextQueryRep *>(qr);
myRep->startIteration();
while (myRep->hasMore())
{
    lemur::api::QueryTerm *qt = myRep->nextTerm();
    cout<<qt->id()<<" "<<qt->weight()<<endl;
}
```

The first line in this snippet declares a pointer of type `TextQueryRep`, called `myRep`. This pointer, typecasted from Lemur's internal query representation (`qr`), contains the list of terms and their attributes. Line 2 in the above code asks to start iterating through those terms. Finally, in the `while` loop, we get one term at a time (`qt`) and print it's ID as well as weight.

Open 'RetEval.cpp' (typically found in app/src/ directory in your Lemur source distribution) and add Code 5 at the place(s) where you want to know the query terms and their weights. For our purpose, we will place this code before the feedback and after it. Find a line in 'RetEval.cpp' which says `model->updateQuery(*qr, *topDoc);` and paste the above code before and after this line. For the second time, you need to remove the first line from this code since it is a declaration of `myRep` and C++ does not allow declaring a variable with the same name in the same namespace.

Recompile your Lemur distribution by running `make`. This time it should compile 'RetEval.cpp' and its dependent files only. Once you have successfully compiled *RetEval* application, run it with the parameter file given in Code 4. This time you should see a different output on the screen (your result file generated should still be the same). It should print query term IDs followed by their weights before and after the feedback. Since your parameter file asked for 10 feedback terms, you may have up to that many new terms after the feedback. You will also notice the weights changed for the terms that belong to the original query.

Let us now find out the actual terms associated with these IDs. To do so, we will use a utility in Lemur called *dumpTerm*. Run it as

`dumpTerm <index_name> <term_id>`

It should print the term on the first line. The rest of the lines in the output are formatted as

`<doc_id>(<term_count>): <term_positions>`

Now you can obtain all the real terms and you already have their weights. You can use this information to prepare a structured query as demonstrated before and use that to do retrieval. In fact, this is exactly what *RetEval* is doing in the feedback loop of retrieval. The advantage of us getting hold of those terms is that we could manipulate them as well as their weights before the feedback loop is run. For instance, we can show these terms to the user and ask him to choose the relevant terms, and use that feedback to reformulate the query and re-run the retrieval. This is "real" relevance feedback.

4 Summary

1. If the user can provide us feedback about the importance of various concepts or terms in his information request (query), we can do more effective retrieval for that user in that situation.

2. Structured queries allow us to specify relative importance of query terms.

3. Query reformulation is a popular method in IR to improve the one-shot retrieval performance and match the retrieved information as much to the information need of the user in the given context as possible.

4. Lemur has all the support needed to refine or reformulate queries with or without user feedback.

References

Anick, P. (2003). Using terminological feedback for web search refinement - a log-based study. In *Proceedings of ACM SIGIR* (p. 88-95).

Dumais, S. T., Cutrell, E., Cadiz, J., Jancke, G., Sarin, R., & Robbins, D. C. (2003, August). Stuff I've Seen: A System for Personal Information Retrieval and Re-Use. In *Proceedings of ACM SIGIR.* ACM Press.

Teevan, J., Dumais, S. T., & Horvitz, E. (2005). Personalizing search via automated analysis of interests and activities. In *Proceedings of ACM SIGIR* (p. 449-456).

White, R. W., Bilenko, M., & Cucerzan, S. (2007, July 23-27). Studying the use of popular destinations to enhance web search interaction. In *Proceedings of ACM SIGIR.* Amsterdam, The Netherlands.