DD2476 Search Engines and Information Retrieval Systems

Assignment 3: Relevance Feedback and Language Models

Johan Boye, Carl Eriksson, Jussi Karlgren, Hedvig Kjellström

The purpose of Assignment 3 is to learn about ways to get more powerful representations of query and documents. You will learn 1) how to use relevance feedback to improve the query representation; 2) how to use language models to improve the document and query representation; and 3) how to speed up the search in various ways.

The recommended reading for Assignment 3 is Chapter 2.4.1, 7, 8 and 9.

Assignment 3 is graded, with the requirements for different grades listed below. In the beginning of the oral review session, the assistant will ask you what grade you aim for, and ask questions related to that grade. All the tasks have to be presented at the same review session – you can not complete the assignment with additional tasks after it has been examined and given a grade. ***Come prepared to the review session!*** *The review will take 15 minutes or less, so have all papers in order.*

E: *Completed Tasks 3.1, 3.2 with some mistakes that could be corrected at the review session.*
D: *Completed Tasks 3.1, 3.2 without mistakes.*
C: *E + Completed Task 3.3 without mistakes.*
B: *C + Completed Task 3.4 without mistakes.*
A: *B + Completed Task 3.5 without mistakes.*

These grades are valid for review March 28, 2017. See the web pages www.kth.se/social/course/DD2476, VT 2017 ir17 - Computer assignments in the menu, for grading of delayed assignments.

Assignment 3 is intended to take around 50h to complete.

Computing Framework

For Assignment 3, you will be further developing your code from Task 2.2 or 2.7. Make sure that you **correct all errors in the code from Assignment 2**, that were pointed out at the examination, so that the ranked retrieval works without errors.

Task 3.1: Relevance Feedback

The first task is to **implement relevance feedback in your search engine**. You will need to add code to the method **`relevanceFeedback`** in the **`Query`** class, so that when this method is called with the **`queryType`** parameter set to **`Index.RANKED_QUERY`**, the system should expand the query using the Rocchio algorithm with parameter $\gamma = 0$, i.e., by multiplying the weights of the original query terms with α, and adding query terms from the documents marked as relevant, setting the weight of these terms to β*<weight of term in doc>.

Note that prior to the Rocchio computation, you have to length-normalize both the query vector and the document vectors (by dividing the document/query weights with the length of the vector, as usual).

The query and the document vectors should be represented in the same way, with either tf or tf-idf weights. The length of the query and the documents should also be computed in the same way, e.g. as the number of terms, or as the Euclidean length of the document/query vector.

Many terms will reoccur in different documents. Make sure not to add terms twice to the query, but instead add to the weight of the existing term instance.

Play around with different values of α and β. Make sure that your tf-idf score computation takes the query term weights into account.

When your implementation is ready, compile and run the search engine, indexing the data set **`davisWiki`**. Select the "Ranked retrieval" option in the "Search Options" menu, and try the search queries

zombie attack

money transfer

as well as **your own word query (at least 3 words long)**. For each of the three queries, select two documents in the top ten list of retrieved documents, that you think are the most relevant[1]. Mark these using the buttons at the bottom of the GUI, and press ”New search”. The Rocchio algorithm will now be applied using the original query and these two documents.

What happens to the two documents that you selected?

What are the characteristics of the other documents in the new top ten list - what are they about? Are there any new ones that were not among the top ten before?

Play with the weights α and β: *How is the relevance feedback process affected by α and β?*

Ponder these questions: *Why is the search after feedback slower? Why is the number of returned documents larger?*

---

[1] The easiest way to inspect the documents is in the actual Davis wiki. Most documents called **`Doc_Name.f`** have the web address **`https://daviswiki.org/Doc_Name`**. If that isn't the case, you will have to look in the text file in the corpus.

At the review

To pass Task 3.1, you should be able to start the search engine and perform a search in ranked retrieval mode with a query specified by the teacher, and then perform relevance feedback, marking a set of documents specified by the teacher.

You should be able to explain the Rocchio algorithm using pen and paper, using the concepts and illustrations in the book and in the slides of Lecture 4. You should also be able to discuss the questions in italics above, and to explain all parts of the code that you have modified.

Task 3.2: Designing an evaluation

In Tasks 1.5 and 2.3, you have learned how to evaluate a search engine by measuring precision and recall for a representative query and a representative data set (the Davis wiki).

Now you will use this knowledge to design an experiment to evaluate how the performance of the search engine improves with the relevance feedback you introduced in Task 3.1. The experiment should use the same test queries as in Tasks 1.5 and 2.3. The concepts of *precision-recall curve* and *precision at 10* should be used.

Important: The experiment should only use a dataset of the kind you constructed in Tasks 1.5 and 2.3, with queries and relevance scores to documents for these queries. It should not use test persons – the test persons have been used to create the dataset of queries and documents with relevance scores.

NOTE: You do not have to carry out the experiment, but you must describe it in text, in such a degree of detail that another student in the course could carry out the experiment with this text as guidance.

At the review

To pass Task 3.2, you should show your written account of the experiment design, and be able to explain all parts of it to the teacher.

Task 3.3: Speeding Up the Search Engine (C or higher)

Implement **at least one of the following speed-ups** of the search engine:

1) Index Elimination – use only high-idf terms at query-time (you do not have to make it work for phrase retrieval, i.e., no need to keep track of the relative word positions in the original query).
2) Index Elimination – use only multi-term occurrences at query time (≥*n* terms from the query must appear in the document in order for it to be returned).
3) Champion Lists – at indexing time, save a list of the *n* top ranked documents for each term, and use for ranked retrieval.

When your implementation is ready, select the "Ranked query" option in the "Search Options" menu, and perform the same search with and without the speedup, for a number of queries. Measure the computation time with and without the speedup, using the function **`System.nanoTime()`** or one of the profilers available for Java.

What is the average speed-up, and how is it affected by parameters in the approximation method?

(There might be other factors affecting the computation time, to such an extent that you do not get a clear speed-up. This is ok, the important thing is that you show – and is able to explain – your method of evaluation.)

At the review

To pass Task 3.3, you should be able to describe the approximation you implemented, how much is gained in terms of computation time, and what approximations to the search are made in order to obtain the speedup. You should also be able to explain all parts of the code that you edited.

Task 3.4: Ranked Bi-Gram Retrieval (B or higher)

The task is now to **implement ranked retrieval using a bi-word model**. You will need to construct a new type of index called **`BiwordIndex`**, implementing the **`Index`** interface, as an alternative to the **`HashedIndex`** class. The terms in this index should be bi-grams. (For example, the phrase **sleeping on campus** consists of two bi-grams, **sleeping on** and **on campus**.)

You will need to add code to the **`search`** method, so that when this method is called with the **`structureType`** parameter set to **`Index.BIGRAM`**, the system should perform retrieval based on the bi-word index rather than the standard one-word index. You only have to make ranked retrieval possible with this setting.

Change the parsing of the query so that each bi-gram in the query is compared to the bi-word index in turn, and that a cosine score is computed as a combination of the tf-idf scores for each bi-gram in the query.

How can the tf-idf score of a bi-gram be defined?

When your implementation is ready, compile and run it, indexing the entire, or parts of, the data set **`davisWiki`**. (The bi-gram structure takes a lot more space than the standard uni-gram index.) Select the "Ranked retrieval" option in the "Search Options" menu, the "Bigram" option in the "Text Structure" menu, and try the search queries

zombie attack

money transfer

as well as **your own query**. The returned lists will vary depending on how large part of the dataset you have indexed. You can try to verify the correctness of your result by manually checking some files on the results list.

Compare to **the same three queries on the same dataset** using a standard ranked retrieval with a one-word index, as described in Assignment 2.

Run each of the 6 tests above using your implementation, look at the 10 highest ranked documents and label them as relevant (labels 1, 2 or 3 according to the description in Task 2.3) or non-relevant (label 0 according to the description in Task 2.3) to the question. Assume the total number of relevant documents in the dataset to be 20 for all queries – this only affects the recall measure up to a scale factor.

Produce 6 precision-recall graphs and note (if applicable) the precision at 3, recall at 3, precision at 10 and recall at 10.

Are the bi-gram search results generally more precise than the standard uni-gram results (higher precision at 3, 10)? Does the bi-gram ranking list miss important relevant documents, that were returned by the uni-gram search (lower recall at 3, 10)?

At the review

To pass Task 3.4, you should be able to start the search engine and perform a search in ranked retrieval and bi-gram mode with a query specified by the teacher, and be able to explain the results in the two modes.

You should also be able to discuss the questions in italics above, showing 6 precision-recall curves for the three queries above on the dataset **`davisWiki`** or on a subset of that, using bigram and unigram search, and to explain all parts of the code that you edited.

Task 3.5: Ranked Sub-Phrase Retrieval (A)

In Task 3.4, you saw that a standard ranked retrieval sometimes has a low precision, while a bi-gram ranked retrieval sometimes has a low recall. **A realistic search engine combines the advantages of both methods!** Use both the **`BiwordIndex`** and the **`HashedIndex`** at the same time, performing both a bi-gram search and a standard uni-gram search. (If you had more computational resources, it would be good to add a **`TriwordIndex`**, a **`TetrawordIndex`** and so on; Google has indexes up to 6-gram.)

You will need to add code to the **`search`** method, so that when this method is called with the **`structureType`** parameter set to **`Index.SUBPHRASE`**, the system should perform sub-phrase search. You only have to make ranked retrieval possible with this setting.

The sub-phrase retrieval commonly used in search engines works like this:

Let the maximum indexed phrase length be n words. Let the query length be m. In your case it is enough with $n = 2$.

First, an min(n,m)-gram ranked retrieval is performed. (As an example, a 3-gram retrieval in the **`davisWiki`** data set with the query **sleeping on campus** returns 19 matches, same as for phrase search with the same query.)

If less than *k* documents are returned, proceed to do an (*n*–1)-gram retrieval. (As an example, a 2-gram (bi-gram) retrieval in the **`davisWiki`** data set with the query **sleeping on campus** returns 1202 matches.)

If less than *k* documents are returned from the (n–1)-gram retrieval, and *n* > 1, proceed to do an (n–2)-gram retrieval. Repeat until *k* documents are found or until *n* = 1. (As an example, a uni-gram (single term) retrieval in the **`davisWiki`** data set with the query **sleeping on campus** returns 9885 matches, same as for standard ranked retrieval with the same query.)

Weigh together the document scores so that an *n*-gram match is always worth more than just an (*n*–1)-gram match, i.e., that documents are ranked according to how long substrings from the query are found in the document. *How should the engine weigh together the n-gram, (n–1)-gram, etc, score for a document, to achieve this?*

At the review

To pass Task 3.5, you should be able to start the search engine and perform a search in ranked retrieval and sub-phrase mode with a query specified by the teacher, that behaves in a manner that cohere with the search results in Task 3.4.

You should be able to reason about the search result in relation to the results in Tasks 3.1 and 3.4 and how cosine scores for different phrase lengths are weighed together, and to explain all parts of the code that you edited.