DD2476 Search Engines and Information Retrieval Systems

Assignment 3: Relevance Feedback and Language Models

Hedvig Kjellström and Johan Boye

The purpose of Laboration 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 that of Lectures 2, 4, 6, and 7.

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 10 minutes or less, so have all papers in order.*

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

These grades are valid for review March 19, 2013. See the web pages www.csc.kth.se/DD2476/ir13/laborationer 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 either Task 2.2 or 2.5. 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 `Query`, 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 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 you have to normalize both the query weights and the document weights, by dividing the document/query vector with the length of the same vector. This has to be done prior to the Roccio computation.

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 represented in the same way, as #terms, or as the Euclidean length of the document/query vector. The results below were generated with tf weights (i.e., no idf weighting of query terms) and length(doc) = #terms in doc.

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

Play 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, load the `svwiki/files/1000` data set, select the "Ranked retrieval" option in the "Search Options" menu, and try the query:

tillvarons yttersta grunder

Found 33 matching document(s)

0. svwiki/files/1000/23.txt 0,03249
1. svwiki/files/1000/600.txt 0,01779
2. svwiki/files/1000/478.txt 0,00800
3. svwiki/files/1000/976.txt 0,00561
4. svwiki/files/1000/47.txt 0,00291
5. svwiki/files/1000/19.txt 0,00220
6. svwiki/files/1000/529.txt 0,00219
7. svwiki/files/1000/933.txt 0,00202
8. svwiki/files/1000/424.txt 0,00201
9. svwiki/files/1000/641.txt 0,00197
10. svwiki/files/1000/126.txt 0,00195
etc.

You will get a ranking similar to the one above, depending on what tf-idf function and document length measure you use. Select document 600 (Flygbussarna = The Airport Buses) as relevant using the buttons at the bottom of the GUI, and press ”New search”.

NOTE: The feedback buttons are numbered from 1, while the returned documents are numbered from 0 – a mistake from our side. For example, to mark document with rank 1 (in the list above, 600) as relevant, select feedback button #2.

The new search result should look **somewhat** like this (can vary depending on your length and term weight representation). This list was generated with $\alpha = 0.1$ and $\beta = 0.9$, a very heavy focus on the relevant documents:

tillvarons yttersta grunder

Found 797 matching document(s)

0. svwiki/files/1000/600.txt 0,00891
1. svwiki/files/1000/23.txt 0,00122
2. svwiki/files/1000/523.txt 0,00095
3. svwiki/files/1000/620.txt 0,00092
4. svwiki/files/1000/8.txt 0,00087
5. svwiki/files/1000/117.txt 0,00083
6. svwiki/files/1000/164.txt 0,00082
7. svwiki/files/1000/161.txt 0,00066
8. svwiki/files/1000/613.txt 0,00066
9. svwiki/files/1000/854.txt 0,00064
10. svwiki/files/1000/814.txt 0,00063
etc.

600 becomes the new top ranked document. The third most relevant document, 523, has the title Flygplats (= Airport). Thus, the list is now less about religion and philosophy, and more about air travel. By marking 600 as relevant, you indicated to the engine that this is what you want.

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? Why are there more highly ranked short documents?*

At the review

To pass Task 3.1, you should show that your search engine behaves in the manner above during relevance feedback search. You should also be able to discuss the questions in italics above, and to explain all parts of the code that you edited.

Task 3.2: Speeding Up the Search Engine (C or higher)

Implement **at least one of the following speedups** of the search engine:

1) Index Elimination – use only high-idf terms at query-time (remember to recompute word positions in documents after removing low-idf words)
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. 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 speedup, and how is it affected by parameters in the approximation method?*

At the review

To pass Task 3.2, you should be able to describe the speedup 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 3.3: Ranked Bi-Gram Retrieval (B or higher)

The task is now to **implement ranked retrieval using a bi-gram word model**. You will need to construct a new type of index called `BiwordIndex`, implementing the `Index` interface, as an alternative to the `HashedIndex/MegaIndex` class. The terms in this index should be bi-grams. (For example, the phrase **november eller december** consists of two bi-grams, **november eller** and **eller december**.) Implement a switch, so that you easily can change between `BiwordIndex` and the one term indexes `HashedIndex/MegaIndex`. You do not have to implement storage on disc for `BiwordIndex`.

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?* Make sure that you do not destroy the query parsing for uni-grams – the switch between `BiwordIndex` and `HashedIndex/MegaIndex` should also control the switch between the two parsing methods.

When `BiwordIndex` is activated instead of `HashedIndex/MegaIndex`, both the ”Intersection query” and ”Ranked retrieval” options in the "Search Options" menu should work with bi-grams instead of standard uni-grams as in Assignments 1 and 2. The "Phrase query" option should be deactivated.

When your implementation is ready, load the `svwiki/files/1000` data set, select the "Ranked retrieval " option in the "Search Options" menu, and try the query:

tillvarons yttersta grunder

Found 3 matching document(s)

0. svwiki/files/1000/23.txt 0,05746
1. svwiki/files/1000/47.txt 0,00391
2. svwiki/files/1000/199.txt 0,00221

The similarity scores above might differ quite a lot from your solution, depending on how you compute the tf-idf score of a bi-gram. However, the ranking should be identical, as 23 and 47 contain both bi-grams, 47 is longer than 23, 199 contains only the first bigram, and 199 is longer than 47.

Compare to the same search using a standard ranked retrieval with a uni-gram index (`HashedIndex` or `MegaIndex`), which will return a ranking similar to:

tillvarons yttersta grunder

Found 33 matching document(s)

0. svwiki/files/1000/23.txt 0,03249
1. svwiki/files/1000/600.txt 0,01779
2. svwiki/files/1000/478.txt 0,00800
3. svwiki/files/1000/976.txt 0,00561
4. svwiki/files/1000/47.txt 0,00291
5. svwiki/files/1000/19.txt 0,00220
6. svwiki/files/1000/529.txt 0,00219
7. svwiki/files/1000/933.txt 0,00202
8. svwiki/files/1000/424.txt 0,00201
9. svwiki/files/1000/641.txt 0,00197
10. svwiki/files/1000/126.txt 0,00195
11. svwiki/files/1000/199.txt 0,00177
etc.

Redo this **with different queries**. *Compare the documents in the two lists – are the bi-gram search results generally more precise than the standard uni-gram results (higher precision)? Does the bi-gram ranking list miss important relevant documents, that were returned by the uni-gram search (lower recall)?*

At the review

To pass Task 3.3, you should show that the search engine indeed returns 3 documents in response to the query **tillvarons yttersta grunder** on the `svwiki/files/1000` data set, with the same ranking as the one above. (The scores themselves depend on many factors, so they are not relevant.) You should also be able to discuss the questions in italics above, showing printouts of examples with different queries and returned lists, and to explain all parts of the code that you edited.

Task 3.4: Ranked Sub-Phrase Retrieval (A)

In Task 3.3, you saw that a standard ranked retrieval sometimes has a too low precision, while a bi-gram ranked retrieval sometimes has a too 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.)

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

Let the maximum indexed phrase length be n words (n = 2 in your case). Let the query length be m.

First, an min(n,m)-gram ranked retrieval is performed. (As an example, a 3-gram retrieval in the `svwiki/files/1000` data set with the query **tillvarons yttersta grunder** returns two matches, documents 23 and 47.)

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 `svwiki/files/1000` data set with the query **tillvarons yttersta grunder** returns three matches, documents 23, 47, and 199.)

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 `svwiki/files/1000` data set with the query **tillvarons yttersta grunder** returns 33 matches, documents 19, 20, 23, 47, 61, 126, 149, 199, 334, 389, 424, 478, 525, 529, 542, 548, 576, 600, 641, 656, 664, 721, 741, 834, 837, 878, 902, 909, 933, 935, 975, 976, and 990.)

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 3.4, you should show that the search engine with *k* = 10 indeed returns 33 documents in response to the query **tillvarons yttersta grunder** on the `svwiki/files/1000` data set, with the documents 23, 47, and 199 at the top of the ranking. (The scores themselves depend on many factors, so they are not relevant.) You should also be able to explain how cosine scores for different phrase lengths are weighed together, and to explain all parts of the code that you edited.