Johan Boye, Dmytro Kalpakchi, KTH, 2019-02-28

DD2476 Search Engines and Information Retrieval Systems Assignment 3: Relevance Feedback and Tolerant Retrieval[1]

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) why query expansion is an alternative to relevance feedback; 3) how to build k-gram index; 4) how to perform tolerant retrieval with wildcard queries and 5) how to implement isolated spelling correction for intersection and phrase queries.

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

Assignment 3 is graded, with the requirements for different grades listed below. In the beginning of the oral review session, the teacher will ask you what grade you aim for, and ask questions related to that grade. The assignment can only be presented once (unless you get an F) – you cannot raise your grade by doing additional tasks after the assignment 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, 3.3 with some mistakes that could be corrected at the review session.*
D: *Completed Tasks 3.1, 3.2, 3.3 without mistakes.*
C: *E + Completed Task 3.4 without mistakes.*
B: *C + Completed Task 3.5 without mistakes.*
A: *B + Completed Task 3.6 without mistakes.*

These grades are valid for review March 26, 2019. See the Canvas pages 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. 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.

Please download additional files needed for this assignment on Canvas and put all the classes to the `ir` folder of your code repository.

[1] With contributions by Carl Eriksson, Jussi Karlgren, and Hedvig Kjellström.

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.

Try different values for 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. Check these using the checkboxes next to the document title in the results list in the GUI, and press "Enter". 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?

Try different values for 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 is relevance feedback called a local query refining method? What are other alternatives to relevance feedback for query refining?*

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 lecture slides. 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: Evaluation using non-binary judgments

In Tasks 1.5 and 2.3, you have learnt how to evaluate a search engine by measuring precision and recall for a representative query and a representative data set (the Davis wiki). In those tasks, you estimated the relevance of each result on a scale from 0 to 3, but we actually never used this refined scale in the evaluation. Let's do that now!

The file **`average_relevance.txt`** contains your relevance estimations for the query "**`graduate program mathematics`**" (averaged over all students, and rounded off to the nearest integer). Use this file and your search engine, and:

1. Compute the *normalized discounted cumulative gain* (nDCG) at 50 for the query **`graduate program mathematics`**.
2. Mark "Mathematics.f" (the most relevant document according to everyone) in the search interface, and re-search (thereby applying relevance feedback).
3. Now compute the nDCG at 50 for the new results list. Do **not** include the "Mathematics.f" document in the computation, even if it is among the top 50. (QUESTION: Why do we want to omit that document?).
4. Compare your result in 1 and 3 above. What do you see?

At the review

To pass Task 3.2, show your calculations to the TA, and explain your results. Also prepare an answer for the question above.

Task 3.3: K-gram index

In this task you will lay the foundation for implementing **tolerant retrieval methods**, which will make your search engine more robust to misspellings, alternative spellings, foreign phrase spellings etc.

There exist several data structures that enable tolerant retrieval (see more in the section 3 of the textbook). The data structure we will be exploiting for the purpose of this assignment is called **k-gram index** (see section 3.2.2 of the textbook). In short, a k-gram index stores a mapping from a character-level k-grams to the words containing those k-grams. Boolean retrieval operations are identical to those of a standard inverted index.

Task 3.3 will consist of the following two parts:

1. complete the **KGramIndex** class so that your search engine can build a k-gram index based on the specified file and perform an intersection search on this index;
2. modify the method **insertIntoIndex** of the **Indexer** class, so that your search engine can construct a k-gram index at the same time as a standard inverted index is being built.

For (1) in the list above the **KGramIndex** class should be compiled separately:

```
javac -cp . -d classes ir/KGramIndex.java
```

The program is then executed as follows:

```
java -cp classes ir.KGramIndex -f kgram_test.txt -p patterns.txt
-k 3 -kg "ove mea"
```

The above command means that you want to build a 3-gram index based on the file **kgram_test.txt** using **patterns.txt** for tokenization and then search for all words containing 3-grams **ove** and **mea**.

When you finished implementing task the first subtask, construct a 2-gram index from the provided file called **kgram_test.txt**. Try the search queries

ve

which should result in 27 postings, and

th he

which should result in 21 postings.

At the review

To pass Task 3.3, your **KGramIndex** implementation should return correct results on the queries above (based on file **kgram_test.txt**) and on any other queries specified by the teacher. The indexing of one file should be immediate and the indexing of davisWiki should not increase overall indexing time by more than 1 second. After the indexing of davisWiki corpus is done you should print the search results over the constructed k-gram index to the console **only** for two example queries specified above. You should also be able to explain all parts of code that you added.

Task 3.4: Wildcard Queries (C)

In this task you will implement a wildcard queries functionality, which will allow users to issue queries like "historic* humo*r". For the purpose of this task we will consider only queries containing maximum one asterisk (*) per word, either in the beginning or in the middle or at the end.

How would you interpret the meaning of the query "historic humo*r"?*

The way to handle wildcard queries in this task is to use a **k-gram index** that you implemented in task 3.2. A simple Boolean retrieval on this index will then allow you to get

all words matching the wildcard query terms. The obtained words are then used to query the standard inverted index as in previous assignments.

However, there are some problems with using only Boolean retrieval. For instance, for a query "re*ve", the Boolean retrieval on bigram index would also return the word "revenge", which is not following a general pattern provided by a wildcard query.

*Why could the word "revenge" be returned by a bigram index in return to a query "re*ve"?*

How could this problem of false positives be solved?

Your search engine should perform the in-memory k-gram indexing of a **davisWiki** corpus and allow issuing of **intersection, phrase and ranked** multiword wildcard queries.

How would you get the ranking for the ranked wildcard queries?

Here are some examples of wildcard queries on the davisWiki corpus obtained using a bigram index:

Query	Intersection search results	Phrase search results
mo*y transfer	142	3
b* colo*r	478	51
having *n	1742	310

The results for a ranked retrieval (using tf-idf only) are presented below:

mo*y transfer	b* colo*r	having *n
Found 2761 matching document(s) **0. Marguerite_Montgomery_School.f** **1. Monkey_Bar.f** **2. Rialto_Lane.f** **3. MattLM.f** **4. Angelique_Tarazi.f** **5. JordanJohnson.f** **6. Danbury_Street.f** **7. Meadowbrook_Drive.f** **8. Monterey_Avenue.f** **9. Susan_Montgomery_Ranch.f**	**Found 13923 matching document(s)** **0. jessicperson.f** **1. Beau_Bagels_%26_Bakery.f** **2. Blood_Source.f** **3. Butterfly_Home_Cooking.f** **4. Biofuel.f** **5. PJSmith.f** **6. AlexanderStewart.f** **7. Blue_Bird_Limo_Services.f** **8. Baskin_Robbins.f** **9. BBQ.f**	**Found 15249 matching document(s)** **0. LoganSchwab..f** **1. Alex_Kloehn.f** **2. RobinHolm.f** **3. Matt_Dudman.f** **4. DaveThomas.f** **5. MasonHarrison.f** **6. Liam_Creighton.f** **7. Evan_Rothstien.f** **8. The_Saddest_Fountain.f** **9. Steve_Robinson.f**

Which of the three queries was the fastest? Which was the slowest? Why?

At the review

To pass Task 3.4, you should be able to explain how you use a k-gram index to handle various types of wildcard queries. You should be able to demonstrate that your search engine returns correct results on the queries above and any other queries specified by the teacher. You should be able to explain all parts of your code and should be prepared to answer all questions in italic. Any wildcard query should take **at most 1s on average**.

Task 3.5: Isolated spelling correction of one-word queries (B)

It is advantageous if a search engine offers the wildcard queries functionality. However, one should be an advanced user in order to use them properly. In tasks 3.5 and 3.6 you will be working with a more frequent use case requiring tolerant retrieval - spelling errors.

It is quite frequent that users issue queries with typos to the search engines, like "cpital of Sewedn" and still expect to get the results for "capital of Sweden". In this case a search engine should perform tolerant retrieval and suggest the best spelling correction or in some cases even issue a corrected query instead of the one with a typo.

There exist two types of spelling correction methods: **context-dependent** and **isolated**. The former tries to suggest a spelling correction based on the surrounding words while the latter operates based on the word itself and the dictionary of the correctly spelled words. In this task we will concentrate on the **isolated spelling correction for one-word queries.** This means, for example, that for the query "mre" your search engine should be able to suggest a ranked list of spelling corrections, e.g. "more", "mere", "mare" etc (**without** taking the context into account).

For the purpose of this task we will employ the following spelling correction strategy:

- if the search engine returned some results for a query, a spelling correction should **not** be performed;
- if the search engine returned 0 results for a query, a spelling correction should take place as follows:
    - all possible k-grams should be generated from a word using the same way of generating k-grams that was used for k-gram indexing in task 3.3;
    - all words containing at least one of the k-grams should be retrieved from a k-gram index;
    - for each retrieved word, a Jaccard coefficient between the set of k-grams of this word and a set of k-grams of a query word should be calculated, and the words that do not pass the pre-set threshold should be discarded;
    - for each word that passed a Jaccard threshold, the edit distance should be calculated using a dynamic programming algorithm (allowed operations are insertion, deletion and substitution; the first two have a cost of 1 and last one has a cost of 2) and the words that are unable to pass the pre-set threshold should be discarded;
    - the survived words are spelling correction candidates and should be ranked according to a sensible ranking function designed by yourself (*Which properties should this function have? Can you re-use the calculations performed during the previous steps?*), the ranked list of spelling correction suggestions for a query word will be referred to as a **candidate list**.

The spelling correction suggestion **should be instant** and the strategy above requires quite a lot of calculations, so can easily become too slow (although it might still work decently for a davisWiki corpus). Therefore, one should think about implementing every part as efficiently as possible. To aid you, we have compiled the following list of hints and suggestions.

- The most time consuming part of the strategy above concerns calculating a Jaccard coefficient. Using its definition one can come up to the following formula that simplifies the calculations $J(A,B) = \frac{|A \cap B|}{|A|+|B|-|A \cap B|}$. *Why is this formula valid?*
- Computing the intersection of the two sets of k-grams might become very time consuming, therefore it might be a good idea to adapt a linear scan you used for handling intersection queries in the assignment 1. However, this time you are not interested in getting intersection, but rather a union of k-gram postings lists (see

more details in section 3.3.4 of the textbook). *Why? How will it help you to compute the size of the intersection between two sets of k-grams?*

- One needs to know the number of k-grams for every spelling correction candidate. Computing this on the fly will slow down computations dramatically (especially for larger corpora), therefore one should think about storing this information during k-gram indexing.

In order to help you in attacking this task, we provide a skeleton of the **`SpellChecker`** class. You should complete the methods **`jaccard`**, **`editDistance`** and **`check`** in order to pass task 3.5. We have also provided an auxiliary **`SpellingOptionsDialog`** class that enables using a dialogue window with a drop-down UI element for presenting a list of ranked candidates.

After completing the **`SpellChecker`** class, you'll need to initialize its instance in the **`Engine`** class (the instance variable called **`speller`**). If **`SpellChecker`** is implemented properly, the search engine should show a **`SpellingOptionsDialog`** window with the spelling alternatives for any misspelled query.

Here are the ranked candidate lists for some example queries produced by our implementation of a spell checker (please note that your ordering could be somewhat different, but most words should match).

thn	dcmber
than	**december**
then	**dumber**
thin	**camber**
th	
thun	
tn	
thnx	
thorn	
thon	
thahn	
thien	

We thresholded a Jaccard coefficient at the value of 0.4 and the edit distance at the value of 2. Note that these threshold values are not necessarily optimal.

NOTE. There is a small bug with timing of spelling suggestion process. To fix it, you'll need to locate the following two lines of code in the **`SearchGUI.java`** and place them directly after initializing the **`corrections`** array.

```
elapsedTime = System.currentTimeMillis() - startTime;

System.err.println("It took " + elapsedTime / 1000.0 + "s to check spelling");
```

At the review

To pass Task 3.5, you should be able to explain how you use a k-gram index for spelling correction. You should be able to demonstrate that your search engine returns reasonable spelling suggestions for the queries above and any other queries specified by the teacher. The suggestion of spelling alternatives **must be instant (less than 0.1s)**. You should be able to reason about the ranking function you chose and the thresholds you selected for both Jaccard coefficient and edit distance. You should be able to explain all parts of your code that added to and should be prepared to answer all questions in italic.

Task 3.6: Isolated spelling correction of multiword queries (A)

In this task you will extend your spelling corrector, so it can handle multi-word queries. You should build upon the **`SpellChecker`** class that you implemented in the task 3.5 and complete the last untouched method **`mergeCorrections`**, that takes a list of candidate lists for each query term and a limit on the total number of suggested spelling corrections and produces the candidate list for the whole query phrase.

When talking about multiword queries, one can think about spelling correction of either intersection or phrase queries, which are handled in different ways. In this task you are expected to implement a spelling correction **only for intersection queries**. However, you should also be able to **explain how you would handle phrase queries**. We encourage you to implement spelling correction for phrase queries as well, but this is **NOT** a requirement and nobody will be failed based on this.

The handling of intersection queries has some pitfalls both design-wise and implementation-wise. One problem is a ranking function for spelling suggestions. In the previous task it could have been any function that ordered the spelling corrections by similarity with the misspelled query. In this task, we are also interested in a query that results in a (hopefully) non-zero number of search results. One has to deal with a trade-off between these two desired properties. A "good" suggestion is one which is both similar to the misspelled query, and has a high probability of returning a non-empty list of results.

The score of a spelling candidate for a multi-word query phrase should be some function of the scores of the individual spelling candidates for each word. A merging procedure then should return a candidate list sorted by the score defined above (we will refer to the algorithm solving this problem simply as **merging algorithm**). The problem itself is not easy, since one can't simply calculate all possible combinations between candidate lists - that would be computationally infeasible. Imagine that you have a query that has 5 words with 5 candidate corrections for each of them. You would have to examine $5^5 = 3125$ candidate query phrases, which is too much, given that spelling correction should work instantly.

There are multiple ways of tackling the problem, one of which is using a Divide & Conquer style algorithm. First, one could try merging candidate lists for the first two query terms, then for the merged list and the third query term etc. Nonetheless, this algorithm alone would not solve the posed problem. One interesting observation is that usually we don't need 3125 spelling correction suggestions anyway - we might need approx. 10-15 of them.

How would you use this observation to improve your algorithm?

At the review

To pass Task 3.6, you should be able to demonstrate how your phrase spelling correction works for intersection queries (should work instantly for davisWiki corpus) and reason why your ranking function is a feasible one for this problem. You should be able to explain how your merging algorithm works and **why it is scalable** with respect to query size. You should be able to explain all parts of code that you added and explain how would you tackle a spelling correction of phrase queries.