In this session:

- We will learn how to tell ElasticSearch to apply different tokenizers and filters to the documents, like removing stopwords or stemming the words.
- We will study how these changes affect the terms that ElasticSearch puts in the index, and how this in turn affects searches.
- We will complete a program to display documents in the tf-idf vector model.
- We will compute document similarities with the cosine measure.
- We will program a simple User Relevance Feedback cycle on top of ElasticSearch.
- We will evaluate this strategy over a collection of documents.

1 Modifying ElasticSearch index behavior

One of the tasks of the previous session was to remove from the documents vocabulary all strings that were not proper words. Obviously this is a frequent task and all these kinds of DB have standard processes that help filter and reduce the terms that are not useful for searching.

Text before being indexed can be subjected to a pipeline of different processes that strip it from anything that will not be useful for a specific application.

The first step of the pipeline is usually a process that converts raw text into tokens. We can for example tokenize a text using blanks and punctuation signs or use a language specific analyzer that detects words in an specific language or parse HTML/XML, ...

This section of the ElasticSearch manual explains the different text tokenizers available.

After we have tokens, we can also normalize the strings and filter valid tokens that are not useful. For example, usually strings are transformed to lowercase so all occurrences of the same word have the same token whether it is capitalized or not. Also there are words that are not semantically useful when searching such as adverbs, articles or prepositions. In this case each language will have its own standard list of words, these are usually called stop words. Another language-specific token normalization is stemming. The stem of a word corresponds to the common part of a word from which all variants are formed by inflection or addition of suffixes or prefixes. For instance, the words *unstoppable*, *stops* and *stopping* all derive from the stem *stop*. The idea is that all variations of a word will be represented by the same token.

This section of ElasticSearch manual will give you an idea of the possibilities.

2 The index reloaded

The first task of this session is to study how this pipeline changes the tokens and its total number. You have a new version of the last session indexer script named `IndexFilesPreprocess.py`. This has two additional flags `--token` and `--filter`.

The flag `--token` changes the text tokenizer, you have four options `whitespace`, `classic`, `standard` and `letter`. Use each one of them with the novels documents and compare the results. Do not change the filter that is used by default (in this case only lowercasing the string). Have a look into the documentation to understand what these tokenizers do. Use the `CountWords.py` script from the last session to see how many tokens are obtained.

After this, use the more aggressive tokenizer and use the filters available in the script: `lowercase` (obvious), `asciifolding` (gets rid of strange non ASCII characters that some languages love to use), `stop` (remove standard english stopwords) and the different stemming algorithms for the english language (`snowball`, `porter_stem` and `kstem`). You have to use the `--filter` flag, that must be the last one and you can put the filters to use separated by blank spaces, for instance:

```
$ python IndexFilesPreprocess.py --index news  --path /tmp/20_newsgroups \
 --token letter --filter lowercase asciifolding
```

Now you can answer the question, what word is the most frequent one in the English language?

As a bonus, you can learn how to configure the text analyzer of an index and you can change the script so more options can be used.

As a side project, you can check also if all this preprocessing changes or improves the fitting of Zipf's law.

3 Computing tf-idf's and cosine similarity

This part of the session is to make sure we understand the tf-idf weight scheme for representing documents as vectors and the cosine similarity measure. We will complete a script that receives the paths of two files, obtains its ids from the index, computes the tf-idf vectors for the corresponding documents, optionally prints the vectors and finally computes their cosine similarity

The script `TFIDFViewer.py` has a set of incomplete functions to do all this:

- The main programs follows the schema just explained
- The `search_file_by_path` function returns the id of a document in the index (the path has to be the exact full path where the documents were when indexed, not just a filename)
- The `document_term_vector` function returns two lists of pairs, the first one is (term, term frequency in the document), the second one is (term, term frequency in the index). Both list are alphabetically ordered by term.
- The incomplete `toTFIDF` function that returns a list of pairs (term, weight) representing the document with the given docid. It:
  1. First gets two lists with term document frequency and term index frequency
  2. Gets the number of documents in the index.
  3. Then finally creates every pair (term, TFIDF) entry of the vector to be returned.

  Your task is to complete the computations of the tf-idf value to fill this vector. You have all the ingredients ready, and you only have to apply the formulas seen in class.
- The incomplete `normalize` function should compute the norm of the vector (square root of the sums of components squared) and divide the whole vector by it, so that the resulting vector has norm (length) 1. Complete this function.
- The incomplete `print_term_weigth_vector` prints one line for each entry in the given vector of the form (term, weight). Complete this function.

- The incomplete `cosine_similarity` function can be implemented by first normalizing both arguments (if they are not already), then computing their inner product. Complete this function. **IMPORTANT:** It must be an efficient implementation, with at most one scan of each vector. Exploit the fact that the vectors are sorted by term alphabetically.

For computing the square root and log10 you can use the numpy library functions `log10` and `sqrt`. This library is already imported in the script as `np`.

3.1 Experimenting

Once you are done with your program, try it out with the test collections from the previous sessions. First, test your implementation by computing the similarity of a file with itself (what should it give?).

You may even want to create a very simple collection with two documents and three or four terms so that you can check by hand your implementation.

You can do all sorts of experiments, for example, are the documents of a specific subset of the corpus `20_newgroups` more similar among them that to other unrelated subset (e.g `alt.atheism` vs `sci-space`)?, ... (you know *l'imagination au pouvoir*).

A final question, have you noticed that we are searching the documents using the path name? Was the path tokenized by the index? What did we do differently when indexing the documents so we can look for an exact match in the path field? Check the script.

4 Document relevance

To show more precisely how ElasticSearch works you have a script named `SearchIndexWeights.py` that it is similar to the script from the first session, but now only shows a specific number of hits (it shows the total number of documents that matches the query as the last line of the output) and also has a relevance score computed by ElasticSearch representing how well the document matches the query. The scripts has a `--nhits` flag that changes how many hits are shown and a `--query` flag that accepts a list of words (performs an AND query with all the words), this flag has to be the last one when invoking the script.

Play a little bit with this script, performing different queries and observing the scores. The syntax of the query allows using the fuzzy operator `~n`, but also the boost operator `^n` with n indicating how important is this term compared with the others, this changes the relevance score. For example with the `20_newsgroups` corpus try the following queries:

```
python SearchIndexWeights.py --index news --nhits 5 --query toronto nyc
python SearchIndexWeights.py --index news --nhits 5 --query toronto^2 nyc
python SearchIndexWeights.py --index news --nhits 5 --query toronto nyc^2
```

You will see that the scores and the positions of the documents change. Invent new queries and observe the results.

5 We will, we will Rocchio you

Finally, you should program a script `Rocchio.py` that implements a User Relevance Feedback system using Rocchio's rule. In fact, we will implement Pseudo-relevance Feedback since we will not ask the user which documents s/he finds relevant: simply we will assume that the first k documents are the relevant ones, for a k of our choice.

More precisely, we want our script to do the following:

1. Ask for a set of words to use as query
2. For a number of times (*nrounds*):
   (a) Obtain the k more relevant documents
   (b) Compute a new query applying Rocchio's rule to the current query and the Tf-Idf representation of the k documents
3. Return the k most relevant documents after the n iterations

You will have to implement the function that computes the new query applying the Rocchio's rule. Applying the Rocchio's rule to a query and a list of documents involves computing[1]:

$$Query' = \alpha \times Query + \beta \times \frac{d_1 + d_2 + \cdots + d_k}{k}$$

You will have to compute the tf-idf vector for each document. To average all the documents you will have to sum vectors with a large number of elements, consider using a dictionary to do it more efficiently instead of merging vectors.

Also the resulting list of terms will be large. Consider pruning the list to only the R more relevant terms (larger weights).

Most of the elements to solve this you already have from previous lab work:

1. From `SearchIndexWeights.py` you have the code for building a query for a list of words and then retrieving the k more relevant documents.
2. Adding the weights computed using Rocchio's rule to each word in the search needs only concatenating the word, the boost operator (`^`) and the weight.
3. Computing the tf-idf vector for a document.

Observe that there are several parameters you can play with, at least:

- *nrounds*, the number of applications of Rocchio's rule
- k, the number of top documents considered relevant and used for applying Rocchio at each round
- R, the maximum number of terms to be kept in the new query
- α and β, the weights in the Rocchio rule.

Please make them easy to change them in the code (e.g., their values defined only once!).

[1]Notice that we have set $\gamma = 0$ in the more general formulation of Rocchio's rule.

5.1 Experimenting

Once you are done with your programming, try it out with the test collections from the previous sessions. Do the queries that pseudorelevance feedback produce make sense? for example, do the new terms seem related to what the user is looking for?

In which sense do the results improve? Recall? Precision?

Investigate to some extent the effect of each parameter. Do you get very different results if you change the parameters *nrounds*, k, R, α, β? Do you find, for each one, some value or value range that seems to be optimal in some sense?

6 Deliverables

To deliver: Write a short report (4-5 pages max) with your results and thoughts. PDF format is preferred. Make sure it has your names, date, and title.

For the first part of the session, comment on the effects you observed on the index (size, number of terms etc). Avoid using only vague terms like "many", "a lot", etc. but also avoid giving too many numbers, tables, screenshots, etc. and no conclusion.

For the second part of the session, explain

1. if you read and understood the code that was provided
2. if you succeeded in implementing everything
3. any major difficulties you found
4. any observations on your experiments or on what you learned this way

Finally, for the last part of the session (Rocchio), explain

1. if you more or less followed the scheme suggested above to implement URF, or if you changed it in some significant way
2. any major difficulties you found
3. a few of the experiments you performed, and the results you observed
4. any thoughts or conclusions that depart from what we asked you to do – in fact, they'll be highly valued if they are intelligent and show that you can go beyond following instructions literally

Please do not explain (again) what Rocchio rule is or stuff that is explained in this document. We know it already!

You must deliver all new and modified python scripts. Please mark with visible comments the parts where you made changes. Pack everything in a .zip file.

You are welcome to add conclusions and thoughts that depart from what we asked you to do. In fact, they'll be highly valued if they are intelligent and show that you can go beyond following instructions literally.

Rules: 1. No plagiarism; don't discuss your work with others, except your teammate; if in doubt about what is allowed, ask us. 2. If you feel you are spending much more time than the rest of the group, ask us for help. Questions can be asked either in person or by email, and you'll never be penalized by asking questions, no matter how stupid they look in retrospect.

To deliver: You must deliver a brief report describing your results and the main difficulties/choices you had while implementing this lab session's work. You also have to hand in the source code of your implementations.

Procedure: Submit your work through the raco platform as a single zipped file.

Deadline: Work must be delivered within 2 weeks from the lab session you attend. Late submissions risk being penalized or not accepted at all. If you anticipate problems with the deadline, tell me as soon as possible.