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.

1 Modifying ElasticSearch index behavior

One of the tasks of the previous session was to remove from the documents vocabulary all those strings that were not proper words. Obviously this is a frequent task and all these kinds of DB have standard processes that help to 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 strips 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 instance, usually strings are transformed to lowercase so all occurrences of the same word have the same token no matter if 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 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 (included in this session scripts) 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? (you will be surprised, or not, if you do this with the arxiv corpus)

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 program 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 lists 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 here 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 explained 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. Use strongly 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`.

In order to test your implementation you have a set of documents inside the `doc` directory that correspond to the ones used in the theory slides examples.

4 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 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? By default, all text fields are tokenized. Yet, if the path field *had* been tokenized, these searches would not succeed, right? What did we do differently when indexing the documents so we can look for an exact match in the path field? Check the script.

5 Deliverables

To deliver: Write a short report (3-4 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

You must also deliver all the 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. You should solve the problem with one other person, we discourage solo projects, but if you are not able to find a partner it is ok. 2. No plagiarism; don't discuss your work with others, except your teammate if you are solving the problem in two; if in doubt about what is allowed, ask us. 3. 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.