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 strings that were not proper words. Obviously this is a frequent task and many document-oriented DB such as ElasticSearch have standard processes that help filter and reduce terms that are not useful for searching.

Text before being indexed can be subjected to a pipeline of different processes, stripping it from anything that will not be useful for a specific application. In ElasticSearch these preprocessing pipelines are called *Analyzers*; ElasticSearch includes many choices for each preprocessing step.

Figure 1: Analyzer Pipeline, from `https://www.elastic.co/blog/found-text-analysis-part-1`

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.

Once we have obtained tokens, we can normalize and/or filter them. For instance, strings can be transformed to lowercase so all occurrences of the same word have the same token regardless of whether they are capitalized or not. Also, there are words that are not semantically useful when searching such as adverbs, articles or prepositions (i.e. *stopwords*), which are typically filtered out. Each language will have its own standard list of 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 the preprocessing pipeline changes how tokens are produced. You have a new version of last session's 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.

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 (t, frequency of t in the document), the second one is (t, number of docs in the index that contain t). 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`)? Explore and use your imagination.

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? So, what did we do differently when indexing the documents so that we can look for an exact match in the path field? Check the script.

5 Rules of delivery

1. No plagiarism; don't discuss your work with other teams. You can ask for help to others for simple things, such as recalling a python instruction or module, but nothing too specific to the session.
2. If you feel you are spending much more time than the rest of the classmates, 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.
3. Write a short report with your results and thoughts. Make it at most 2 pages. Strive to summarize what new things you learned in this session. You are welcome to add conclusions and findings that depart from what we asked you to do.
4. Turn the report to PDF. Make sure it has your names, date, and title. Create a single .zip file all the python scripts that you created or modified; in the modified scripts, make sure you mark visibly with comments the parts that you modified.
5. Submit your work through the Racó.

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