Lab Session 3: Programming on ElasticSearch

Alex Carrillo Alza, Oriol Domingo Roig
Cerca i Anàlisi de la Informació, Ciència i Enginyeria de Dades, UPC

30 September 2019

1 Modifying ElasticSearch index behavior

1.1 The index reloaded

As a case of study, we tried all tokenizers available in the `IndexFilesPreprocess.py` script with the `/novels` documents. It should be noted that `lowercase` and `asciifolding` filters have been used in all tests to ensure consistency and character compatibility.

The least aggressive tokenizer was the `whitespace` one (as we were expecting) because it breaks terms whenever it encounters that character, leaving a substancial number of 167063 indexed words. Next, both `standard` and `classic` tokenizers provided similar results: 61739 and 59480 words, respectively, even when the first one is more general (and works well for most languages) and the second one is good for English because it treats acronyms, company names, emails and internet hosts. However, both tokenizers, for instance, kept indexing numbers. Lastly, `letter` offered the best reduction of words, at first sight, with almost the same amount as our function in lab Session 1, 54365 words indexed, because it uses the same approach we implemented (essentially, getting rid of non-letter characters).

Then, we kept the `letter` tokenizer and the `lowercase`, `asciifolding` filters as before and tested some others:

	Indexed words	Most frequent word
`stop`	54332	ⓘ
`porter_stem`	34258	the
`kstem`	37921	the
`snowball`	33679	the

It was clear that there was a big difference between using or not some stemming algorithm as the number of indexed words was reduced considerably in all three tests (`porter_stem`, `kstem` and `snowball`) and not when using the `stop` filter. This particularity along with the fact that our result was a bit disappointing led us to look for which stopwords list was being used by Elasticsearch. We found that the list was only composed by *33 terms*. This was the reason why the `stop` filter returned '`i`' as the most frequent word, whereas all the others returned '`the`' (a stopword). In any case, it was curious to note '`i`' is a pronoun usually considered a stop term, so we concluded that the stopwords list provided was not enough for a proper indexing. We tried unsuccessfully to create our own list but many implementation errors were given and we desisted because it was not the focus of the report.

2 Computing tf-idf 's and cosine similarity

In this section we developed some code that helped us to better understand the **Term Frequency-Inverse Document Frequency** (tf-idf) concept and the **Cosine Similarity**. The last one appeared to be quite tricky, however, after considering alphabetic order the proper algorithm was developed and produced the desired results. In our case, we decided to normalize the weight vectors at the computation of the tf-idf phase instead of doing it in the cosine similarity computation.

3 Experimenting

Once `TFIDFViewer.py` script was working then some experiments were conducted. First, same documents were shown to the script to test whether it was working well or not. In this case, the desired result for the cosine similarity should be 1, since both documents are identical, and in fact, this is what we observed[1].

Next test was to show the small phrases (*doc*) that were used in Theory Classes. We already knew the tf-idf and the expect results of cosine similarity between doc 3 and doc 4, thus, we could compare our output with the truth one. After that, we concluded that our algorithms were working properly, hence, further experiments could be reliable.

If you wish to compute the similarity between documents about different fields, you should get a small value that makes you think that there is non-similarity. For example, if someone compares the *20_newsgroups/rec.sport.baseball /0009972* with *20_newsgroups/sci.med/0013000*, it appears to be that there is not an identical word in both documents. This is an example of two documents from different fields that seeks to have a connection or a similar meaning.

The approach we were following was a bit simple since in some cases documents from different fields seemed to be more similar rather than documents based on the same topic. For instance, two documents (0013001,0013000) from the same field: *sci.med*[2] showed a similarity of 0.005019, whereas the same document (0013000) compared with one from *alt.atheism/0000000* present a similarity of 0.016873 almost four times more. This shows that context-based approaches might be better, because taking into account context when analysing a word can completely change the meaning of it. Here, we show an example:

- I have a pain in the **back** : (n. Human body part)
- I'll be **back** in ten minutes : (adv. Backward)

With the present algorithm we might get that 'back' appears in both documents, hence, the similarity will not be null. Nevertheless, not only both words have different meaning, but they are different type of words as well. One common approach that is perform is computing an embedding such as **Word2Vec**, where words that share common contexts in the documents are located close to one another. Then, similarity can be computed regarding context.

Finally, we could compare documents using the path due to the properties that Elasticsearch and the `IndexFilePre Process.py` script presents. The script got the path from any document despite only providing the folder's path, which contains all the documents, thank to the generate_files_list. Then, Elasticsearch stores an attribute regarding the path, hence, you could query for the data in a specific document via path.

4 Conclusions

All in all, in this lab session two main conclusions are deduced. Firstly, the fact of having tested a considerable amount of tokenizers and filters has given us a glimpse of how delicate is to choose a proper tokenizer for a specific language, for example, considering that not all are good for the English files used in this report. Also, we have noticed that the choice of a filter plays a revelant role when discarting terms; even when the stemmers used here returned a similar result (but we have found that the implementation behind are far distinct among diferent languages), we concluded that the list of stopwords must be checked very delicately.

Secondly, we can also state that tf-idf approach is a good insight into a nice mathematical idea used in the computation of similarity between text. Nevertheless, we noticed that some improvements are necessary to achieve a more reliable information retrieval system because the current algorithm presents poor properties regarding context understanding.

[1]To be more precise the result that normally was found is 0.9999999 or 1.00000001, which is 1 regarding float limitations.
[2]They can be found in the *20_newsgroups* folder.