CAIM Lab, Session 2: Programming with ElasticSearch

Adrián Sánchez Albanell

Preprocessing

The first sections of this sessions are about trying ElasticSearch tokenization and filter features, which we will use to do the preprocessing of our data. The tokenization process extracts units or words from our documents, and then we use our filters to flatten out the words collected, transforming, normalizing or excluding words.
The tokenization process is well explained in the documentation, briefly: whitespace divides on whitespace characters, letter on nonletter characters, classic uses a grammar based tokenizer and standard divides into terms on word boundaries specified by the Unicode Text Segmentation algorithms. All of the proposed tokenizers in this session are word oriented tokenizers, and I haven't tried partial word or other types of tokenizers since they seemed not what we need for this session next sections.
On the other hand we have the filter, which have different categories. Lowercase and asciifolding filters normalize words. The stop filter excludes english stop words (such as "the", "is", "at"..). Finally snowball, porter_stem and kstem filters are the most common algorithmic stemmers used, and porter_stem and kstem stemmers needs the lowercase filter to work correctly. There are dictionary stemmers also (we need to download or install the dictionary to use it with ElasticSearch). Dictionary stemmers, theorically, should be better than algorithmic stemmers, since they can distinguish irregular words, present in all languages but Esperanto, but they depend on the dictionary quality and size (any word not present in the dictionary wouldn't be recognized) and are slower, so actually algorithmic stemmers show better performance.
There are other filters, from stemmers to synonym handlers, but we have enough options to experiment with the proposed ones.

Experimenting with Zipf's law

To experiment with the filter's, I did a script to index all filter combinations with every tokenizer option and generate it's *CountWords* with each using *IndexFilesPreprocess.py* and *CountWords.py*. Even if I said porter_stem and kstem filters need the lowercase filter, I tried them without it to compare latter the results obtained. All the files generated by my script with *CountWords.py* are in the *cwords* driectory.
To analyze the results i used a script based on the jupyter Zipf's script of the last session. It tries to fit zipf's power law, then it calculates the mean square error between the fitting and the real data. After it removes some of the most frequent words and calculates again the mean square error, and repeats it until the mean square error variation is minimum, and stores the results (mean square error of all the data, removing the tail, and how many words we removed). It gives another interesting information, the first words and each frequencies (whith or without removing the tail). It is stored in *results.txt* file included with the code.
Also (manually) I generated a file with the information of the mentioned stemmers with and without the lowercase filter. I expected to have better results doing as the documentation says (using lowercase always with these stemmers), it surprised me that it wasn't that way (or not always). It's not significant since *ElasticSearch* documentation is not oriented on the experiment we are doing here and have nothing to do with it, it just says to use lowercase filter before the stemmer to have the expected performance, but I encountered it curious and wanted to try it and see what happens. The results are in *lowercase_ copmpare_ stem.txt* file.
The best result was using *whitespace* token and *stop* and *snowball* filters (observing the attributes I mentioned).
We also observed, as proposed in the session, that the most frequent word in English is the word *the*. It changes if we sue the *stop* filter (which excludes words as *the*). Doing that I could see that the most frequent word in *novels* is *I*, on the other hand, if we do the same with the arxiv corpus, the most common word is *we*.

Computing tf-idf and cosine similarity

In this section of the session we have to complete a script to compute tf-idf and cosine similarity between documents, and then experiment and analyze the results obtained.

Completing the script

There were some functions to complete in the script. For the *toTFIDF* and *print_term_weight_vector* we had to do just as said and it was very straightforward.
The normalize and cosine_similarity functions need more calculations, I used *normalize* and *cosine_similarity* functions from the **sklearn** library which are already implemented and optimized, with the changes needed to work with our data structures. The cosine_similarity function could be done efficiently using that the words are ordered to scan both weight vectors and complete them without repeated calculations.

Example documents

After completing the script, the first try was with the test documents. The result of trying the same example we have seen in theory had a cosine similarity of 0.03505, the same result we have in the theory slides (almost, we are working with double precision and Python so it have a small marge of error). Then we also tried to compare a document with itself, and the result was 1.00005, which is also what we expected.

Experimenting

To do some experiments I compared different files from the news documents. Comparing different files of politics, for example, it showed a similarity of 0.01336, comparing one from politics with another from computer graphics 0.0028. Doing another comparisons between politics documents it shows similarity around 0.013 and 0.014 , comparing politics and computer graphics around 0.002 and 0.006. Doing some tries it seems, that documents about the same topic are more similar between them than with documents of other topics.

Final question

As comented in the script itself, the *path* has *type keyword* so it is not tokenized so we can do exact match search.

Code changes

At the end of *CountWords.py*, due to format errors (caused by working with Windows 10 i supose).

```
print('{}, {}'.format(cnt, pal.decode('iso8859-1').encode('utf-8')))
```