Information Retrieval and Analysis
OCTOBER 2020

Lab Session 3: Programming on ElasticSearch

PROJECT REPORT

Elías Abad Rocamora
Victor Novelle Moriano
Barcelona, UPC - FIB & FME

Analysis of index generation

In this section we will analyze the performance of the program *IndexFilesPreprocess.py* by generating indexes over the same text repository using different tokenizers and filters.

First of all we are going to analyze the performance of the different tokenizers without using any stemming method or asciifolding.

Without modifying the filter applied by default (*lowercase)*, we have computed, using *CountWords.py,* the number of different words present on our index. Moreover, we have applied a small modification in the code in order to see the number of total words in our corpus, to later see how this value changes when an application of filter is performed.

Token	Different words	Total words
whitespace	300604	5219884
letter	94003	5291784
classic	132956	5162904
standard	130290	5241852

As we can extract from the table, the whitespace tokenizer is the one that identifies the most number of different words, as it for example can identify "hello" and "hello," as different words while they are the same.

We can also see that the *letter* tokenizer identifies the lowest number of different words while getting the biggest number of total words. This is because it separates words whenever it encounters a non-letter character and therefore separates into smaller words and more quantity of them.

If we compare these results with the ones using *classic*, we can see that there are quite more different words (132956 v.s. 94003) and less total words (5162904 v.s. 5291784). This performance makes sense as *classic* is the one that recognizes better the words in the english language and doesn't separate words with a " . " or a " ' " inside a word, and therefore gets more variety of different words and less total words.

Now, we will proceed to analyze how the performance of the most aggressive tokenizer (*letter)* varies depending on the selected filters. In this phase we will take profit from our code modification, as we will be able to see how many words each filter removes from the total number.

Filter: lowercase + asciifolding +	Different words	Total words
NA	93990	5291784
stop	93957	3753092
stop + snowball	70754	3753092
stop + porter_stem	70716	3753092
stop + kstem	75198	3753092
stop + all	68515	3753092

As it can be extracted from the table, the addition of the *stop* flag, although only removing 43 different *stopwords*, which is a very small number in comparison to the number of different words, removes 1538692 terms that do not provide any useful information, improving our index humongously.

It can also be seen that, with the addition of the stemming filters the number of different words is reduced but the total words stays invariant. This is due to the fact that the stemming mechanisms, unlike stopwords removals, do not delete terms but collapse them into their root.

When creating an index without doing stopword removal, we obtained that "the" was the most popular word in both *arxiv* and *news.* But, when applying stopword removal, the results were quite different.

Here we can see the top 10 common words in both repositories:

> **ARXIV:** we, model, from, use, which, can, our, time, result, system
> **NEWS:** i, you, s, have, t, edu, can, from, what, one

Only the word "can" appears in both lists, the rest of the vocabulary is quite different. This might be due to the fact that both repositories use the same language, english, but in a different context and ambit. *Arxiv* repository contains scientific articles in which the vocabulary is more technical and the first person in singular is never used, that is why "we" and "our" are used, and not "i". *News,* as its name says, is a news repository, where a more day to day language is used and therefore more common words are in the top 10.

Tf-idf's and cosine similarity

In this section, we were asked to complete the program *TFIDFViewer.py.* This process was pretty straight-forward as the structure of the code was provided and we just needed to fill the incomplete functions using formulas and concepts previously seen in class.

After checking that our code was behaving correctly (using the docs files*)*, we proceed to perform similarity analysis between files of the 20*_newgroups* folder. The first cosine similarity was comparing a file with itself, and obviously obtaining a similarity of 1. Afterwards, we proceed to check if the similarities between documents of the same subject, *atheism*, were higher than comparing them with files from other subjects (*sci-space).*

In order to do so, we developed a Python script that allowed us to perform this task automatically. This program computes the mean inner-group similarity for both folders selecting *n* files (defined by the user), obtaining as a result 0.0382 for *atheism* and 0.0285 for *sci-space.* In general, despite some files of the same group having higher similarities [0.2-0.4], the overall group similarity is quite small. After this step, the code computes the mean similarity between all the possible pairs of documents from different folders. The similarity obtained was 0.0170, being smaller than the previous in-group averages.

Regarding the last question presented, if the path field was tokenized we could not perform its search in the index successfully, as key characters of this field ("/",".", e.g.) would derive in words separation that we do not want to. To avoid this problem, the script explicitly indicates that this string should not be tokenized (*Line 118 in IndexFilesPreprocess.py)*.