Information Retrieval
Programming with ElasticSearch

Laura Cebollero Ruiz, Alexandre Rodríguez Garau

17th October, 2018

1 Introduction

For this second lab session we were asked to perform three main tasks:

- Comment on the effects of using indexes.
- Understand and implement TF-IDF and cosine similarity.
- Understand and implement Rocchio.

2 Delivery files

- `TFIDFViewer.py`: Filled code with the TFIDF computations in order to compute the cosine similarity.
- `Rocchio.py`: Computes the most similar documents using `Rocchio` algorithm.

3 Indexes usage effects

The first part of the session has been used to experiment with indexes using different tokenizers.

The first obvious step on doing so was to execute the provided command:

```
$ python2.7 IndexFilesPreprocess.py --index news --path /tmp/20_newsgroups \
--token letter --filter lowercase asciifolding
$ python2.7 CountWords.py --index news  > index_reloaded_out.csv
```

and then take a look at the tail of the outputted file:

```
$ tail index_reloaded_out.csv

72018, that
75338, is
86895, in
89701, i
101602, and
109645, a
116338, of
129544, to
257648, the
94048 Words
```

We can see that the **most used word** in the **English language is the word** *`the`*.

Now, in order to see how the number of different words varies according to the different tokenizers available. Also we will apply a stemming method via the `-filter` option. The stemming method we selected is `snowball`. We have to highlight that every index is also created using the `lowercase` and `asciifolding` options, so that the words that we get are not repeated and have standard characters.

To see the impact of each tokenizer we will create an index with and without the snowball stemmer for each tokenizer.

For example, if we want to create an index using the `standard` tokenizer with snowball stemming we would use the following options:

```
$ python2.7 IndexFilesPreprocess.py --index news --path /tmp/20_newsgroups \
--token standard --filter lowercase asciifolding snowball
```

We have created a total of 8 indices combining all the tokenizers we found relevant and turning stemming on and off. All indices have been created using the same set of 33 files. The tokenizers we used are: `letter`, `whitespace`, `classic` and `standard`. Let's see how the tokenizer affects the number of different words per index.

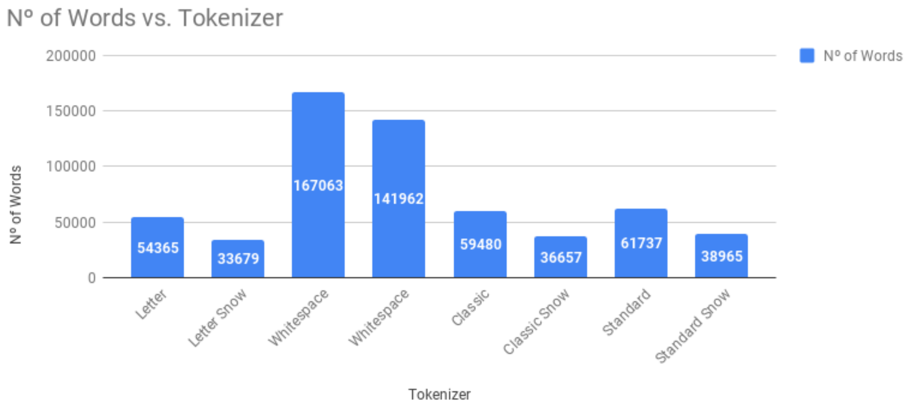

Figure 1: Number of total words with different settings.

We can see in the barplot that the tokenizer that identifies the most different words is `whitespace`. This was to be expected because the `tokenizer` breaks text into terms whenever it encounters a whitespace character which makes anything between spaces a word. This leads to having a larger index because strings like `xxxix.-forg` or `rikki-tikki!"` are also considered valid words that will likely have a frequency of 1.

The difference between the other tokenizers is not very big. This is surprising specially for the `letter` because it works in a reall similar way than `whitespace`: every time it encounters a character that is not a letter it breaks. Since the text is full of characters that are not letters it works fairly well.

It is also worth commenting about the impact of the stemming filter. In all cases the number of words decreases —naturally, since words with the same root will only be counted as one— by about 35% with the exception of the `whitespace` tokenizer, which only decreases by 15%.

4 Computing tf-idf 's and cosine similarity

We have successfully filled the gaps in the incomplete code.

4.1 Experiments

First of all, to check that the implementation is correct we compute the similarity of one file with itself. We know that it should be 1.

```
$ python2.7 TFIDFViewer.py --index news --files
↪  ../files/20_newsgroups/soc.religion.christian/0015362
↪  ../files/20_newsgroups/soc.religion.christian/0015362

Similarity = 1
```

Similarity is OK!

Let's do another experiment. We expect some files from the topic *atheism* to be related to the religious christian ones.

We have selected randomly some of them and have checked their similarity.

```
$ python2.7 TFIDFViewer.py --index news --files ../files/20_newsgroups/alt.atheism/0000181
↪  ../files/20_newsgroups/soc.religion.christian/0015362

Similarity = 0.46078
```

As we can see, since they touch the topic of religion, albeit from 2 very different point of views, we can see that their similarity is close to 0.5, which makes sense.

To finish with another self-explanatory experiment, we can now check the similarity with two files that are from topics that are **very** different from each other, for example between a computer's topic: **Windows X** and a politics topic: **guns**.

```
$ python2.7 TFIDFViewer.py --index news --files ../files/20_newsgroups/comp.windows.x/0005001
↪  ../files/20_newsgroups/talk.politics.guns/0011926

Similarity = 0.06711
```

As we can see, since they are not related, their similarity is very close to 0, meaning they are clearly and indubitably unrelated.

So to summarize:

Files relationship	**Itself**	**Very related**	**Unrelated**
Similarity	1	0.46	0.067

5 Rocchio's rule

You can find the implementation of Rocchio's rule in `Rocchio.py`. We have played with the parameters α and β to see how it affected the resulted queries.

Unsurprisingly, these three cases when constructing a new query were found as expected:

- $\alpha >> \beta$, more priority was given to the original query (in each round) rather than the related words.
- $\alpha << \beta$, more priority was given to the related words rather than the original query (in each round).
- $\alpha = \beta$, the same priority is given on the new words and the original query for each iteration, albeit the weights on the new words have already been normalized.

Default parameters on the delivery.

- $\alpha = \beta = 1$ to give them the same priority.
- $R = 3$ to avoid not getting results (reasoning behind explained on the next section)
- $k = 5$ to retrieve only the first 5 documents.
- $nrounds = 5$, to only perform the operations 5 times.

5.1 Experimenting

We have found that when using a somewhat big R, such as $R > 3$ Rocchio's rule does not work that well and the original words used in the first query may be lost, giving more priority to the related words and ending up producing 0 results.

So in order to not use too many unrelated words which may produce a large new query and thus close to or directly 0 results, we have decided to set $R = 3$.

Related to this, we have seen that when using a big $k > 5$, say for example $k = 10$, the retrieved documents are many and the related words with their weights may tamper the new query, thus resulting in ending up with no results, for it would be using words that may not be that well related to our initial query.

Because of this we have settled with $k = 5$.

Additionally, we have also seen that setting a greater number of rounds leads to a reinforcement of the words found on the first iterations, thus not differing that much from the first words found.

Havng said that, we can experiment with the provided query in document relevance, without giving *toronto* any special weight:

```
$ python2.7 Rocchio.py --index news --query toronto
```

The related R words after 5 rounds are toronto, detroit and vancouver, and all the related documents are about hockey. A quick search on Google tells as that Detroit has Hockey team, called the *Detroit Red Wings*. Toronto, our original query word, also has a team called *Toronto Maple Leafs*, and so does Vancouver with *Vancouver Canucks*.

So basically the retrieved documents are related to Toronto because Toronto has a Hockey team, as do Vancouver and Detroit, thus relating them on the query for they appear on the hockey articles.

Now let's perform another query:

```
$ python2.7 Rocchio.py --index news --query jesus
```

This case is more interesting. In the first iterations the retrieved documents are about Christianity, but after the $nrounds = 5$ *jesus* is lost and the final words on the query are *mercedes*, *eternal* and *girl*.

The related documents (only 3) are news under the category of atheism.

So Rocchio's rule has transformed our original query from jesus (and thus Christianity) to atheism!

If we tune α and β a little, setting them such that $\alpha = 5$ and $\beta = 1$ and also change the query a little setting a weight to the word *jesus* of 5, making it very important, the retrieved results are the same even tho we are prioritizing jesus over the new words.

So now let's try to force the christianity topic by including it onto the query, and rolling back α and β to $\alpha = 1, \beta = 1$:

```
$ python2.7 Rocchio.py --index news --query jesus^1 christianity^5
```

We end up with documents from miscelaneous religion and atheism and a final query of jesus, christianity and convenient, being the later word the one with most weight.

If we try giving more priority to new words *jesus* and *christianity* still persist on the words obtained in the new query, so they stay until the last new query formed with Rocchio's rule.

5.2 Conclusions

We can derive from our experiments many conclusions:

1. R should not be very big for Rocchio's rule to work well. Since our query is boolean and the retrieved documents have to include all the words, even if the weight's are very small, we may end up with no results at all.
2. k should also be somewhat small. We found that a value of k around 5 is good for us. A $k < 5$ returns a small batch of documents and thus the new words for the query are VERY related to the original words in the query. So locality is strong. A large value of k implies losing locality and probably not having results at all.
3. α and β work well for prioritizing what should weight more for queries, but when used in conjunction of a large number of *nrounds* and a small R ends up in most cases with the original words being reinforced so $\alpha << \beta$ in many rounds do not assure having a very different final query from the first one (in terms of words, the weights will change, but the words will probably be the ones from the original query plus i words until reaching R.

So, all in all, Rocchio's rule is useful to try and search documents related to the terms of the original query, but does not work that well with a big number of rounds or of R or k.