Ricard Meyerhofer Parra
Alex Cuello Ortiz
2016-2017 Q1

CAIM - Second Deliverable: Programming on Lucene

First part of the session

We had to put the Stemmer as the last filter because as is stated in the session, lowercase terms are required (to make searches case independent) but also we have to do the stemming after removing stopwords because otherwise index size can increase up to a 40% (and words like prepositions, adverbs…are not going to be useful for indexing and will make us slower, due to the increase of Stemming operations).

Once we modified the classes (IndexFiles.java and StandardAnalyzer.java). We kept the index that we had for novels until now and run again IndexFiles to index the novels into another index and we compared both with Lukeall.
The results we appreciate between both, are the following:

- Just looking at the rankings between both indexes, we can see that the second index contains the words stemmed and also that the rankings between both indexes are different. Some words just increase their rank because of the stemming process for example “appear” (second index) is in a higher position than “appears” (first index), this can be explained because “appear” in the second index, includes appears, appeared etc so we have more representations).
- Also if we execute the command from last session that counts the words and different words in a text, we can see that in the first index we have more different words than in the second index. This is again because of the Stemming process again.

We would also like to highlight that with the Stemming process, we can have words that end having the same stem when originally, were different words. This can be solved with enriching (each term is associated to additional information that can be helpful to retrieve the “right” documents). At this way we’ll avoid confusing terms like “Windows” (OS) with “windows” (object).

<u>**Second part of the session**</u>

<u>**Implementation and explanation**</u>

Tf–idf is a numerical statistic that is used to reflect how important a word is to a document in a corpus or collection. This value increases proportionally to the number of times a word appears in the document, but is offset by the frequency of the word in the corpus, which helps to adjust for the fact that some words appear more frequently in general.

Once modified IndexFiles.java (to build and index with al information required for the tf-idf computation) as said in session, we indexed and cleaned the files and proceed to complete a code (TdIdfViewer.java) that behaves like this:

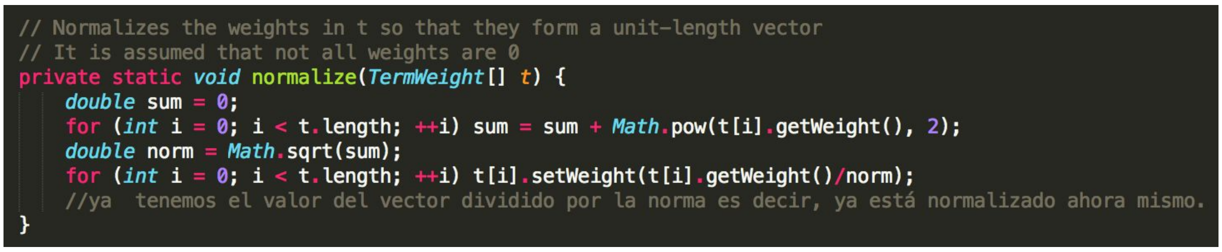
```
while (true) {
1. read two filenames f1, f2;
2. get the document identifiers id1, id2
   of the files f1 and f2 in the index;
3. compute the tf-idf representation of id1, id2
   as vectors v1, v2 (and print them);
4. compute the cosine similarity of v1, v2
   (and print it);
}
```

In the code that was given to us, an IndexSearcher object is used to search the index for the two files in step 2, and an IndexReader object to obtain the relevant information from the index in step 3; Steps 1 and 4 do not require accessing the index.

The functions that were incomplete in the code and we had to implement, are the following:

Normalize

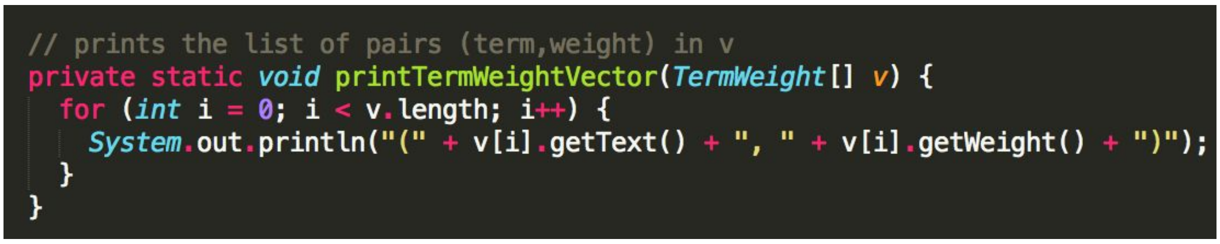
We compute the norm of the vector and divide the whole vector by it, so that the resulting vector has norm 1. Finally, we save in t[i] of the vector, its normalized value.



```java
// Normalizes the weights in t so that they form a unit-length vector
// It is assumed that not all weights are 0
private static void normalize(TermWeight[] t) {
    double sum = 0;
    for (int i = 0; i < t.length; ++i) sum = sum + Math.pow(t[i].getWeight(), 2);
    double norm = Math.sqrt(sum);
    for (int i = 0; i < t.length; ++i) t[i].setWeight(t[i].getWeight()/norm);
    //ya  tenemos el valor del vector dividido por la norma es decir, ya está normalizado ahora mismo.
}
```

PrintTermWeightVector

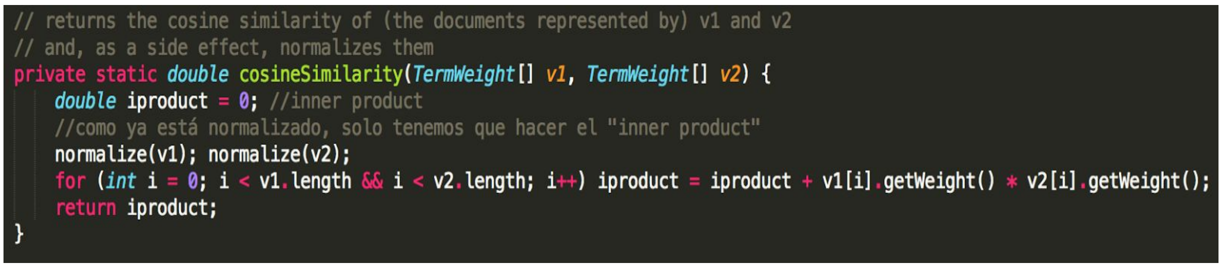
We print one line for each entry in the required format



```java
// prints the list of pairs (term,weight) in v
private static void printTermWeightVector(TermWeight[] v) {
  for (int i = 0; i < v.length; i++) {
    System.out.println("(" + v[i].getText() + ", " + v[i].getWeight() + ")");
  }
}
```

CosineSimilarity

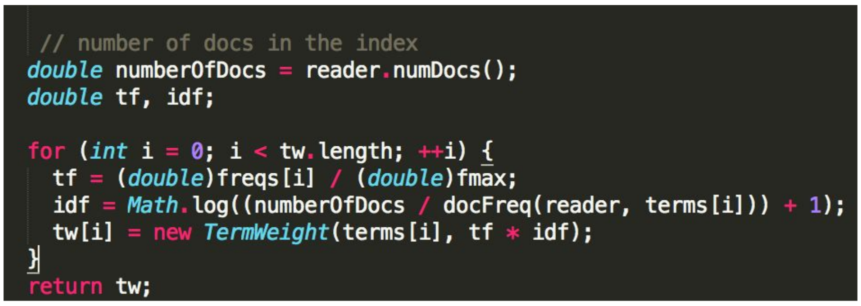
First of all we normalize both arguments and then, we compute the inner product.



```java
// returns the cosine similarity of (the documents represented by) v1 and v2
// and, as a side effect, normalizes them
private static double cosineSimilarity(TermWeight[] v1, TermWeight[] v2) {
    double iproduct = 0; //inner product
    //como ya está normalizado, solo tenemos que hacer el "inner product"
    normalize(v1); normalize(v2);
    for (int i = 0; i < v1.length && i < v2.length; i++) iproduct = iproduct + v1[i].getWeight() * v2[i].getWeight();
    return iproduct;
}
```

toTfIdf

In function toTfIdf we returns an array of TermWeights representing the document whose identifier in reader is docId in tf-idf format, with base 10 logs.



```
// number of docs in the index
double numberOfDocs = reader.numDocs();
double tf, idf;

for (int i = 0; i < tw.length; ++i) {
  tf = (double)freqs[i] / (double)fmax;
  idf = Math.log((numberOfDocs / docFreq(reader, terms[i])) + 1);
  tw[i] = new TermWeight(terms[i], tf * idf);
}
return tw;
```

The first issue we had, was that tf apparently was always 0 when we were trying to apply the formula:

$$tf = frequence/maxfrequence$$

Even frequence was different to 0 and maxfrequence wasn't infinite we had a 0 as result. The reason why it was always 0 is because freq[] was an array of Integers and the same happened with fmax, fmax was a integer variable so we just had to cast it as a double.
Also, we added a *+1* on the logarithm to avoid having a NaN as value because log(0) = NaN and we had problems with some results appeared to us as NaN instead of a value.

<u>Experiment and conclusions:</u>
We tried our code between two collections of different book authors of same genre (science fiction in our case, indexed with the novels collection).

- Novel1, novel2 and novel3 belong to the same author (H.P. Lovecraft).
- Novel4, novel5 and novel6 belong to the other author (Edgar Allan Poe).

So what we should expect is that novels from same author have a higher tf-idf between them and novels from different authors have a lower tf-idf (this is what should be expected). The results are the following:

	novel1	novel2	novel3	novel4	novel5	novel6
novel1	1					
novel2	0.2278612263	1				
novel3	0.1126663321	0.1028834111	1			
novel4	0.2354096752	0.1770621542	0.2751425674	1		
novel5	0.15126868	0.1156421484	0.2745448579	0.3907779044	1	
novel6	0.1468890746	0.1246828446	0.1943085434	0.4124057821	0.2701543904	1

As we can see the tf-idf of a novel with itself is one because it's exactly the same text so there's any difference between them. Also it can be seen that novels 4,5 and 6 are more similar than novels 1,2,3 so the second author that was Edgar Allan Poe, it looks more consistent than the first one that was H.P. Lovecraft.

Surprisingly, we can find that for example novel 1 and novel 4 are quite similar being from different authors even that in general, is not the case. We know that Lovecraft really liked Poe (is something that can be seen in some short stories that Lovecraft has published) and also both authors are not this far away in time so it can be that somehow both can have similar novels. Also Lovecraft has novels that are really close to each other in subject-matter but also there are others that are really different between them (we took non-related ones).