Lab work, Session 2 | **Fall 2016**
Information Retrieval | **Ricard Gavaldà, Marta Arias**

In this session:

- We will complete a program to display documents in the tf-idf vector model.
- We will compute document similarities with the cosine measure.

For this session, we strongly recommend using Eclipse or NetBeans to manage files, compiling, etc. Please see Appendix A.

1 Viewing documents in tf-idf format

This part of the session is to make sure we understand the tf-idf weight scheme for representing documents as vectors and the cosine similarity measure. We will complete a program that basically does the following:

```
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);
}
```

We will use an *IndexSearcher* object to search the index for the two files in step 2, and an *IndexReader* object to obtain the relevant information (tf's and idf's) from the index in step 3; Steps 1 and 4 do not require accessing the index.

2 Adding term frequencies to the index

First of all, we need to make sure that the index we build contains all the information required to compute tf-idf's, which is mainly frequencies of terms in documents. At least in some versions of *Lucene*, the `IndexFiles.java` class does not really store them by default (to save space). With the program we are going to complete, if we use such an index we will get a "null pointer error" because we will ask *Lucene* to give us a certain *TermFreqVector* that is not there.

To solve this, we need to change `IndexFiles.java` as follows. Locate the line where the "contents" field is added to the document that is being indexed. Will look roughly like this:

```
doc.add(new Field("contents",something-about-a-FileReader-or-BufferedReader));
```

and add the flag saying that the new *Field* should store *TermVector* information:

```
doc.add(new Field("contents",something-about-a-FileReader-or-BufferedReader,
Field.TermVector.YES));
```

Compile `IndexFiles.java`, then index the novels directory.

3 Computing tf-idf's and cosine measure

Get the classes `TfIdfViewer.java` and `TermWeight.java`. `TermWeight.java` simply represents a pair (string, weight). Have a look at it and compile it now. `TfIdfViewer.java` is the (incomplete) program that implements the pseudocode above. When it is complete we should call it as:

```
$ java TfIdfViewer -index mydirectory
```

and then enter pairs of filenames to display these files in tf-idf format and get their similarity. The names should include the directory paths as they appear in the index (you can use *Lucene* to view them).

Study the existing code:

- The main method parses the -index argument, creates *IndexReader* and *IndexSearcher* objects, then implements the code above by calling appropriate functions
- The *findDocId* function creates a query to ask the index: "give me the docid's of documents in the index having this string in the 'path' field"
- The *docFreq* function returns the number of documents in the index containing a given term in the 'contents' field. Note that a *Lucene Term* is a pair (fieldname, string)
- The (incomplete) *toTfIdf* function returns a vector of *TermWeights* representing the document with the given docid. It:
  - First gets a *TermFreqVector* with (essentially) pairs (Term, frequency-of-the-Term-in-the-doc).
  - Splits the *TermFreqVector* into two vectors, one of strings with the terms, the other with the corresponding frequencies. IMPORTANT: Lucene guarantees that the vector of strings is sorted according to String.compareTo, the funny Java way for comparing strings. Recall that if x and y are strings, x.compareTo(y) returns a negative int if x goes before y (in dictionary order), a positive int if x goes after y, and 0 if x == y.
  - Gets the number of documents in the index.

    – Then finally creates every *TermWeight* entry of the vector to be returned.

  Your task is to complete the computations of the tf-idf value to fill this vector. You have all the ingredients ready, and only have to apply the formulas seen in class.

- Function *normalize* should compute the norm of the vector (square root of the sums of components squared) and divide the whole vector by it, so that the resulting vector has norm (length) 1. Complete this function. You may want to use function *Math.sqrt.*
- Function *printTermWeightVector* just prints one line for each entry in the given vector of the form (term, weight). Complete this function.
- Function *cosineSimilarity* can be implemented by first normalizing both arguments, then computing their inner product. Complete this function. IMPORTANT: It must be an efficient implementation, with at most one scan of each vector. Use strongly that the vectors are sorted by term, according to *String.compareTo.*

4 Experimenting

Once you are done with your program, try it out with the test collections from the previous session. First, test your implementation by computing the similarity of a file with itself (what should it give?).

You may even want to create a very simple collection with two documents and three or four terms so that you can check by hand your implementation.

5 Deliverables

***To deliver:* Write a brief report (less than 1 page) explaining a) if you read and understood the code that was provided, b) if you succeeded in implementing everything, c) any major difficulties you found, and d) any observations on your experiments or on what you learned this way. Pdf format is preferred. Make sure it has your name, date, and title. You must also deliver all the classes that you modified. Not the java classes you didn't modify. Please mark with visible comments the parts where you made changes. Pack everything in a .zip file.**

Procedure: Submit your work through the Rac at `https://raco.fib.upc.edu/`. Since these sessions can be done in pairs, it is enough if 1 person of the pair submits, however, state clearly both names so that I can account for it. The restrictions on what couples are allowed will be stated in class.

Deadline: Work must be delivered within 2 weeks from the lab session you attend. Late deliveries risk being penalized or not accepted at all. If you anticipate problems with the deadline, tell me as soon as possible.

A How to import Lucene source into `Eclipse` and `Netbeans`

```
How to import and run the Lucene source from Eclipse:

This has been tested on

Eclipse Standard/SDK
Version: Kepler Service Release 1
Build id: 20130919-0819

and

NetBeans IDE 7.3.1

and

lucene-3.6.2

on Windows 7

by Ricard Gavald‘a, september 29th, 2013

----------------- Instructions for Eclipse

0. Install Eclipse
1. Download lucene-3.6.2-src.tgz and uncompress it to a folder lucene-3.6.2-src
2. Open Eclipse. Create a project (File -> New -> Javaproject).
   Give a project name (e.g. practicaRI), Next, Finish
3. Import the lucene core source code:
   - In the package explorer on the left, open practicaRI folder
   - rightclick on src, then Import then General, then File System
   - "From directory": browse to the lucene-3.6.2-src -> lucene-3.6.2 -> core -> src.
     Select the java folder for import
   - in the next form, tick on the java folder (or "select all"), then Finish.
     You should see a lot of org.apache.lucene.* packages imported.
4. Import the lucene demo source code:
   - Same with the folder lucene-3.6.2-src -> lucene-3.6.2 -> contrib -> demo -> src
   (say whatever to an "overwrite" warning)
5. Run the IndexFile.java program:
   - look for IndexFile.Java under org.apache.lucene.demo
   - (optional) click on it to check that it opens in the editor.
     if the syntax-checker flags the "package org.apache.lucene.demo" line,
     you did something wrong with the import, the packages are not in the right place.
   - right click on IndexFiles.java. Run As Java Application.
   Hopefully it will run, and the console window will show
   "Usage: bla bla". This is IndexFiles Running!
6. Go to the "Run -> Run Configurations" option in the main menu,
   look for the IndexFiles Configuration, then the Arguments tab, and enter
   the command line you want to give indexfiles, something like:
   -docs directorydirectory\novels -index mynovelsindex
   Now run again, and it should index.
7. You can edit now IndexFiles.java (and/or other) files and run again
```

```
---------------- Instructions for Netbeans
0. Install Netbeans
1. Download lucene-3.6.2-src.tgz and uncompress it to a folder lucene-3.6.2-src
2. Open Netbeans. Create a project
   File -> New Project -> Java + Java project with existing sources
   Give a project name (e.g. practicaRI) then Next
   Add two folders:
    - .../lucene-3.6.2-src/lucene-3.6.2/core/src/java
    - .../lucene-3.6.2-src/lucene-3.6.2/contrib/demo/src/java
   Finish
3. You should see on the left a bunch of packages org.apache.lucene.*
   and, in another directory, org.apache.lucene.demo.
   Go to the IndexFiles.java file within the demo, right-click on it,
   choose Run.
   Hopefully it will run, and the console window will show
   "Usage: bla bla". This is IndexFiles Running!
4. Go to the Files option in the File menu, choose Project Properties (practicaRI)
   In the Run option, choose IndexFiles.java as main class (browse to it)
   and in Arguments, enter whatever command-line arguments you want, such as
   -docs directorydirectory\novels -index mynovelsindex
   Now run again, and it should index.
5. You can edit now IndexFiles.java (and/or other) files and run again
```