Lucene - Filters/TF-IDF

CAIM

CS - FIB

Curso 2016/2017

Apache Lucene

This session

- We will learn about the basic Lucene classes, and, in particular, about those for reading, preprocessing, and indexing files.
- We will reprogram Lucene so that it applies a Porter stemmer and removes stopwords.
- We'll study how these changes affect the terms that Lucene puts in the index, and how this in turn affects searches.
- We will complete a program to display documents in the tf-idf vector model.
- We will compute document similarities with the cosine similarity

Lucene Clases

Field class

Field: *It represents a feature of a document that the user wants to remember in the index (such as name, path, date, size, contents, topic, owner...)*

A field can be defined to be indexable or not, that is, whether it can be used in searches or not. The user can also specify whether a field should be tokenized; typically, the file contents is the only tokenized Field.

Lucene Clases

Document class

Document: *Once the file has been indexed, it becomes a Document object, which is conceptually no more than a series of pairs Field=value.*

Lucene Clases

Index class

Index: *A Lucene index is conceptually a set of Document, those that the user has indexed. A single Index object contains thus information about all the indexable fields of all the documents that it contains. Within an Index, each Document receives a unique identifier, which is an integer number.*

Lucene Clases

Query class

Query: *An object that describes a query that the user applies to an Index. The result will be a list of Document identifiers. After that, the user can retrieve additional information about these Documents (such as their names) querying the Index with their identifiers.*

Lucene Clases

IndexWriter/Analyzer classes

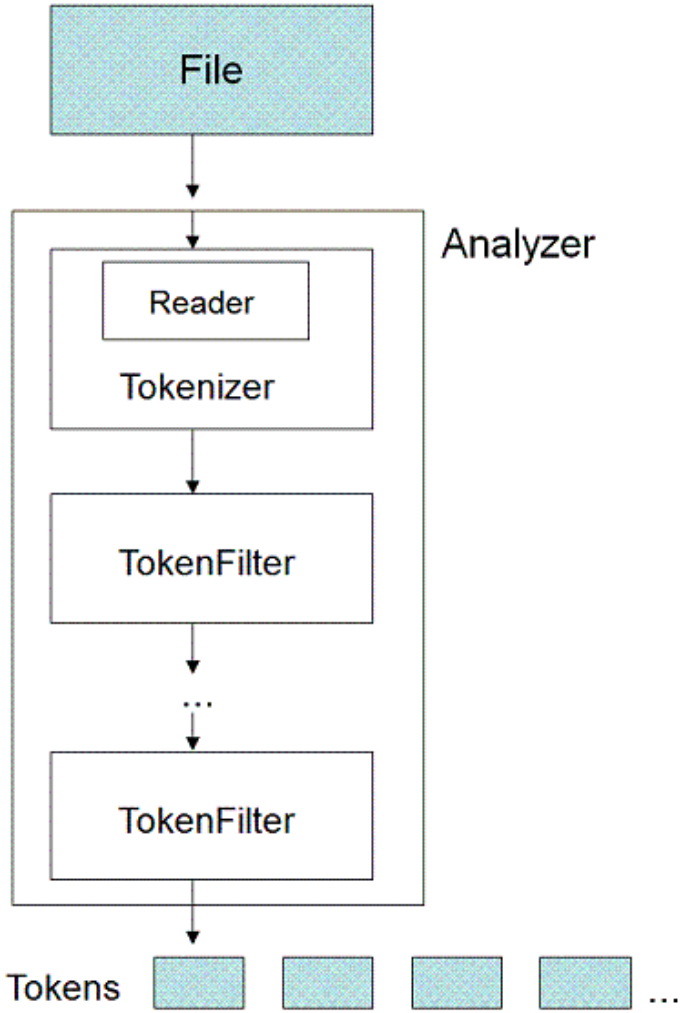
IndexWriter: *An IndexWriter object, after adequate parametrization, builds or modifies an existing Index.*

Analyzer: *An Analyzer object is one of the pieces that form an IndexWriter. It reads a text file and produces a TokenStream. A Token can be whatever the user chooses it to be, but is most normally a word that we want to store and possibly search later. An Analyzer applies to the input text a Tokenizer and later a variable number of TokenFilters. In pictures:*

Lucene Clases

Analysis

A Tokenizer reads the text file and chops it into Tokens, producing a TokenStream; a TokenFilter reads a TokenStream and emits another TokenStream. The Tokenizer uses a Reader, an object that one opens, asks for chars repeatedly, and closes.



Lucene Clases

Analysis

As the name indicates, a TokenFilter may simply eat up undesired tokens (words without meaning, punctuation signs, html tags, etc.).

It can also do other things, such as transforming the tokens (e.g., stemming) or even add new tokens to the stream.

Lucene Clases

IndexSearcher class

IndexSearcher: *Is used to query indices.*

The query becomes effective by passing a Query object to the IndexSearcher.search() method. This returns a Hits or Topdocs object (depending on the version), basically a Vector of Document identifiers, in order of computed relevance of each document for the query.

Each IndexSearcher object can have an associated function to evaluate the relevance of a Document for a Query.

Lucene Clases

QueryParser class

QueryParser: *This is a tool to generate Query objects from strings.*

For example, in the format defined by Lucene, QueryParser transforms the string "`+Spielberg -Steven`" into a query that, informally speaking, when applied to an Index, will return the documents that contain "Spielberg" but not "Steven".

The work

Now we are going to modify the sources of Lucene to perform some experiments using a Stemmer as a filter (Porter Stemmer).

You will need to download the sources from the Lucene Site (3.6.2) or copy the directory `/opt/lucene-3.6.2-src`

Follow the instructions of the documentation for this session.

Indexing and Tokenizing

- Edit the class `org.apache.lucene.demo.IndexFiles`.
- Look where `IndexFiles` create an `IndexWriter` object. Study the code fragment where it is created and used.
- See how when creating an `IndexWriter` one specifies the desired Analyzer. In this case, it is `StandardAnalyzer`.
- Look for the code of `StandardAnalyzer.java` (which will be in the Lucene core, not in the Lucene demo).
- Locate where it defines its `Tokenizer` and the `TokenFilters` it uses.

Removing Stopwords and Stemming

Follow the instructions from the documentation of the session for:

1. Changing in `IndexFiles.java` the analyzer to process the tokens removing the stopwords included in the `englishstop.txt` file.
2. Changing `StandardAnalyzer.java` so the Porter stemmer is included in the chain of filters.

To rebuild the lucene sources you can follow the instructions for using an IDE (netbeans or eclipse) from the session web page

Using only the command line

- Find the files using the shell command `find`
  - in a terminal type :

    ```
    find . -name IndexFiles.java
    find . -name StandardAnalyzer.java
    ```
- Now you have the path to the files
- Edit the files with your favorite editor following the session instructions

Using only the command line

- Rebuild lucene using `ant` (in the laboratory pc's there are some previous steps for `ant` working correctly)
  - Create a `.ant` directoty in your root directory
  - In this directory create a file `ant.conf` typing:

    ```
    cat > ant.conf
    rpm_mode=false
    ^D
    ```

  - Inside the root file of the lucene source type:

    ```
    ant ivy-bootstrap
    ant
    ```

  - Inside the lucene directory `contrib/demo` type:

    ```
    ant
    ```

  - now you have inside the lucene `build` directory the new binaries for the **core** and **demo** lucene packages

Using only the command line

- Modify the `setclass` shell file that you used the last week (or you can make a copy)
- Modify the `LUCENEDIR` variable so it is the path to the lucene source `CLASSPATH` variable so the `.jar` files are now the new recompiled binaries

  ```
  $LUCENEDIR/build/core/lucene-core-3.6.2-SNAPSHOT.jar
  $LUCENEDIR/build/contrib/demo/lucene-demo-3.6.2-SNAPSHOT.j
  ```
- Now you can do `source setclass`
- Create a new index for any of the documents set from the last sesion and use `luke` to see the results
- Compare the original words ranking with the new one

TF-IDF - Preliminaries

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. We will complete a program that does basically 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);
}
```

TF-IDF - Preliminaries

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.

TF-IDF - Indexing

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.*

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.*

TF-IDF - Indexing

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.TermVe}ctor.YES));
```

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

TF-IDF - Results

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).

Follow the instructions of the session (programming involved).

The work

Once you are done with your program, try it out with the test collections from the previous sessions. 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.

Deliverables

Write a short report (2-3 pages) with your results and thoughts.

For the first part of the session, comment on the effects you observed on the index (size, number of terms etc). Avoid using only vague terms like "many", "a lot", etc. but also avoid giving too many numbers, tables, screenshots, etc. and no conclusion.

For the second part of the session, explain 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.

You must also deliver all that classes that you modified (and only those). Please mark with visible comments the parts where you made changes. Pack everything in a `.zip` *file.*

Deliverables

- **Delivery:** Using Racó via the "Practiques via web" option
- **Deadline:** Work must be delivered within 2 weeks from today. Late deliveries risk being penalized or not accepted at all.
- **Rules:**
  1. You can solve the problem alone or with one other person, but at most one lab assignment with the same person in the whole course.
  2. No plagiarism; don't discuss your work with others, except your teammate if you are solving the problem in two; if in doubt about what is allowed, ask us.
  3. If you feel you are spending much more time than the rest of the group, ask us for help.