UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Lab work, Session 1 **Fall 2016**
Information Retrieval **Ricard Gavaldà, Marta Arias**

In this session:

- We'll learn what *Lucene* is, what it can do, and what it does not do.
- We'll install *Lucene* and some related package.
- We'll use *Lucene* to index a set of text files, and to ask some simple queries about these documents.
- We'll use *Luke* to look into the indices created by *Lucene*.
- We'll study whether the given texts satisfy Zipf' and Heaps' laws.

You can use either *Linux* or *Windows* for this session. Some familiarity with Java is expected.

1 What is *Lucene*?

Lucene is a library for indexing and searching text files, written in Java and available as open source under the Apache License. It is not a standalone application; it is designed to be integrated easily into applications that have to search text in local files or from the Internet. It attempts to balance efficiency, flexibility, and conceptual simplicity at the API (Application Programming Interface) level.

Some things that *Lucene* does *not* do – so it would be the application's programmer task:

- Ask the user which files have to be indexed.
- Show the results of indexing.
- Ask the user which query s/he wants to ask.
- Show the user the results of the query.
- Display statistics.

Of course, lots of people have built code that does these things on top of *Lucene*. In particular, we'll use a separate program called *Luke* to examine *Lucene*'s indexes and to build and ask queries.

Lucene's homepage is `lucene.apache.org/java/docs/index.html`. Initially, *Lucene* was developed by Doug Cutting and was part of Jakarta, a subproject of Apache. On february 2005, it became a top-level project within Apache.

2 Installing *Lucene*

NOTE: In this course we will use version 3.6.2. There was a major restructuring of the lucene code from 3.6.2 to 4.0.0. Some of the explanations and code provided MAY NOT work with versions 4.x or higher, and it is generally a bit more tricky to setup and run. Indices created with lucene-4.x can't be read using lucene-3.x.

Step 1: Download from `https://archive.apache.org/dist/lucene/java/3.6.2/` the compressed versions of:

- The *Lucene* executable and demo files: `lucene-3.6.2.zip` (or `.tar.gz`);.
- The *Lucene* source code: `lucene-3.6.2-src.zip` (or `.tar.gz`). We won't use for this session, though.

and from the google code luke site

- The *Luke* executable file, `lukeall-3.5.0.jar`.

Step 2: Inside `lucene-3.6.2.zip`, locate the two following `.jar` files:

- `lucene-core-3.6.2.jar`, the *Lucene* executable (compiled)
- `lucene-demo-3.6.2.jar`, the demo executable (compiled)

and place them, for example, in your work directory.

Step 3: Make sure that the Java interpreter will know where to find these `.jar` files at runtime, adding them to the `CLASSPATH`.

- The best option is to define variables e.g. LUCENEDIR and LUKEDIR with the paths, and add the definitions to a command file.
- The command files `setclass` (for *Linux*) and `setclass.bat` (for *Windows*) provided in this session's pack should be good approximations. Check the directories in your install, change them as required, and execute.
- Alternatively, if you are familiar with platforms such as Eclipse or Netbeans, install them. Lucene can be installed as a library there, so you can modify, recompile, and execute lucene classes comfortably. We will need to do this in the next session; it's optional today (instructions provided in the text file `eclipse-netbeans.txt` provided).

3 Indexing and querying

Unzip the files `20_newsgroups.zip` and `novels.zip` provided (place them in directories, say *newsgroups* and *novels*). These directories should contain:

- `newsgroups`: text from 20 usenet groups on various topics, a classic corpus in IR evaluation

- `novels`: A number of random novels and other texts in English from the Project Gutenberg, with a tendency towards late 19th and early 20th centuries

The methods in the executable demo can be called from the command window line. For example, to add all the files in the `./novels` directory to an index, just type:

```
java org.apache.lucene.demo.IndexFiles -docs novels
```

Try it. An *index* will appear in your working directory with the index files – not meant to be read by humans. You may add another flag `-index indexname` besides `-docs` to specify a directory name.

To query this index, type for instance

```
java org.apache.lucene.demo.SearchFiles -index index
Query:  +word1 +word2 +word3
```

This will return all files that mandatorily contain all three words *word1*, *word2* and *word3*. The search options are described in *Lucene*'s documentation at `http://lucene.apache.org/java/2_4_1/queryparsersyntax.html`. Besides `+` and `-`, queries can include AND, OR and NOT (in uppercase) and other connectives. For example:

```
Query:  (word1 OR word2) AND word3
```

Two observations: 1) The difference between `"+word"` and simply "`word`" is that with `+`, `word` *must* appear in the document; without `+`, the word just increases the document's similarity to the query (hence its position in the ranking), though it is not strictly mandatory. 2) The `NOT` operator is binary and interpreted as a set difference. A query cannot start with `NOT`.

Task 1: Play a bit with the searcher. Look into the groups in the corpus and try to imagine queries that would select, for example, mostly files from one particular newsgroup, or Dickens' novels and no others, etc.

After creating the index, you can use *Luke* to examine its contents and make queries. Click on the `jar` file or type

```
java -jar $LUKEDIR/lukeall-3.5.0.jar -index ./index
```

In *Windows*, use `%LUKEDIR%` or simply click on the `.jar` file.

You have tabs to 1) see various statistics and word ranking 2) see info about the indexed files 3) ask queries 4) details about the index files (don't ask me...) and 5) try different analyzers. In the first tab, you can choose the number of *top words* you want to see typing the number in, then clicking on the *Show top terms* button.

Task 2: Now index the `novels` corpus. Use the same line as above, but note that you can add a "-index indexnovels" option if you want to create a different index directory. Can you find a query that returns the novels by Dickens and no other? (just an example...)

4 Heap's and Zipf's law

Now let us look at the word distribution in these texts, and in particular whether Zipf's and Heaps' laws hold. We recommend using the novels corpus because it is much cleaner.

The `CountWords3.java` program in the pack reads and index and writes on the standard output all the terms it contains, with their counts. Compile it, then execute it redirecting the output to a file.

Task 3: Inspect the file. Order by word frequency and remove stupid terms such as numbers, url's, binary or unreadable stuff, dates, etc. Leave only "proper" words. Now we are ready to analyze the data.

For Zipf's law, check if the rank-frequency distribution seems to follow a power law; what are the power law parameters ($f = c * (rank + b)^a$) that seem to describe best what you see? We suggest using a spreadsheet and trying different triples (a, b, c) to try to imitate the number of occurrences of the word as a function of the rank. Plotting a graph (either in linear scale or in log-log scale) is also a good option. Do not depend too much on the very first (most frequent) terms, which are noisy. What matters is the long tail. Do you approximately get to frequence 1, 2, 3, ... in the same places with your formula as the data?

For Heaps' law, check if the number of distinct terms in a piece of text with N words contains about $k * N^\beta$ different words for some k and β. So: 1) create indices containing different numbers of novels, or more precisely different quantities of text, as the sizes of the novels are very different. 2) Use the little program to count the total number of words in each index, and the number of different words in each. 3) See if you find k and β that explain your results well.

5 Deliverables

To deliver: **Write a short report with your results and thoughts on the Zipf's and Heaps' tests. Explain things like: What best values of (a, b, c) and (k, β) did you find? How did you find them - what method did you use? We are less interested in long lists of words of screen prints than in your conclusions, so please 2-3 pages maximum.**

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.