Prof. Nastaran Fatemi
Gary Marigliano

Indexing and Search with *Apache Lucene*

Lab Nº 1

A. Objectives

The goal of this lab is to discover the *Lucene* platform and to learn its functionalities by using its Java API. *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 on the Internet. It attempts to balance efficiency, flexibility, and conceptual simplicity at the API level.

B. Organization

The lab should be realized in a group of maximum 2 students.

Report: It is required to submit a report containing both the answers to the questions asked in different questions, and the source code of your implementation.

Deadline: See deadline on *Moodle*.

C. Import the project

Download the lab 1 Maven project on Moodle and import it in your favorite IDE. The required libraries will be downloaded automatically.

D. Understanding the *Lucene* API

Implementing a full-text search application using *Lucene* requires two steps: (1) creating a lucene index on the documents and/or database objects and (2) parsing the user query and looking up the pre-built index to answer the query.
Follow the *Lucene* Demo API at: http://lucene.apache.org/core/6_6_1/demo/overview-summary.html to build an index of the directory `lucene-6.6.1/docs`. The `SearchFiles` class is already in the package `ch.heigvd.iict.dmg.demo` in the downloaded project. Do a few simple searches to make sure it works. Answer the following questions:

1. Does the command line demo use *stopword* removal? Explain how you find out the answer.
2. Does the command line demo use stemming? Explain how you find out the answer.
3. Is the search of the command line demo case insensitive? How did you find out the answer?
4. Does it matter whether stemming occurs before or after *stopword* removal? Consider this as a general question.

E. Using Luke

Luke is a GUI tool written in Java, which lets you browse the contents of a *Lucene* index, examine individual documents, and run queries over the index. Use *Luke* to examine the content of the index created in the previous section. This index is located in the folder called `indexDemo` in the downloaded project.

Note: You can download it from https://github.com/DmitryKey/luke/ (release 6.6.0) the file "Lucene-Installation-Instructions" contains also some information on how to use it.

F. Indexing and Searching the CACM collection

We are now going to employ the full text search provided by *Lucene* to analyze a publication list. A text file containing a list of publications is provided to you in the file cacm.txt. Each line in the text file contains the following information, separated by tabulations: the **publication id**, the **authors** (if any), the **title** and the **summary** (if any). The authors of a publication are separated by ";". There might be publications without any author or without the summary field.

Your task is to index the publication list, to perform a few queries, and to report the results in the following subsections.

Indexing

Refer to the slides of the *Lucene* course for an example of indexing.

1. Use *StandardAnalyzer* for this part.
2. Import the maven project and implement a program that creates an index of the publication list and allows searching queries on *author*, *title* and *summary* attributes.
3. You need to define one field for each attribute. Note that a document in *Lucene* can have multiple values for a given field. For example the author field accepts multiple author values.
4. Keep the *publication id* in the index and show it in the results set of your queries.
5. *Lucene* provides different types of fields. Take a look at *Lucene* Field class: http://lucene.apache.org/core/6_6_1/core/org/apache/lucene/document/Field.html and explain which field type can be used for *id*, *title*, *summary* and *author*.
6. What should be added to the code to have access to the term vector in the index? Have a look at the *Lucene* slides presented in your course (look at different methods of *FieldType*). Use *Luke* to check that the term vector is included in the index.

Attach the code of indexing to your report. Tip: look at the `TODO student` comments to know where to implement the requested features.

Using different Analyzers

Lucene provides different analyzers to process a document. In this lab we will use the same analyzer for indexing and for searching.

1. Index the publication list using each of the following analyzers:
   - *WhitespaceAnalyzer*
   - *EnglishAnalyzer*
   - *ShingleAnalyzerWrapper* (using shingle size 2)
   - *ShingleAnalyzerWrapper* (using shingle size 3)
   - *StopAnalyzer* with a custom stop list. A list of common words is provided in the file `common_words.txt` of the publication dataset. Use this list as stopwords.
2. Look at the index using *Luke* and for each created index find out the following information:
   a. The number of indexed documents and indexed terms.
   b. The number of indexed terms in the summary field.
   c. The top 10 frequent terms of the summary field in the index.
   d. The size of the index on disk.
   e. The required time for indexing.

Reading Index

Luke is a practical way of getting info on your index that you can use for verification and debugging. You can also read the index via the *Lucene* API to get basic statistics. For this, use the class *HighFreqTerms* and write the code to answer the following questions:

1. What is the author with the highest number of publications? How many publications does he/she have?
2. List the top 10 terms in the title field with their frequency.

Attach your code into the report.

Searching

Rebuild your index using *EnglishAnalyzer*. Complete your program to perform a search based on a given query and show the results in the specific format as given in the following example. Use *QueryParser* for analyzing the query as shown in the *Lucene* slides.

For example, the query `compiler program` could return the results as following:

```
Searching for: compiler program
3189: An Algebraic Compiler for the FORTRAN Assembly Program (1.2440429)
1459: Requirements for Real-Time Languages (1.1556565)
2652: Reduction of Compilation Costs Through Language Contraction (1.1202306)
1183: A Note on the Use of a Digital Computerfor Doing Tedious Algebra and Programming (1.0969465)
1465: Program Translation Viewed as a General Data Processing Problem (0.99523425)
1988: A Formalism for Translator Interactions (0.99523425)
1647: WATFOR-The University of Waterloo FORTRAN IV Compiler (0.9907691)
1237: Conversion of Decision Tables To Computer Programs (0.9245252)
2944: Shifting Garbage Collection Overhead to Compile Time (0.9245252)
2923: High-Level Data Flow Analysis (0.9200119)
```

Each result is formatted as: *publication id* + ":" + *title* +"(" + *Lucene score* + ")".

Write the necessary code to perform the following queries on the *summary* filed:

1. Publications containing the term "Information Retrieval".
2. Publications containing both "Information" and "Retrieval".
3. Publications containing at least the term "Retrieval" and, possibly "Information" but not "Database".
4. Publications containing a term starting with "Info".
5. Publications containing the term "Information" close to "Retrieval" (max distance 5).

For each query provide the text of the query used by *QueryParser*, the total number of results and the top 10 results. Attach your code to the report.

The *Lucene* querying syntax can be found at:
http://lucene.apache.org/core/6_6_1/queryparser/org/apache/lucene/queryparser/classic/package-summary.html#package_description.

Tuning the Lucene Score

The goal of this part of the lab is to use a customized formula for the calculation of the similarity score.

Lucene, starting from version 6, computes a similarity score based on *Okapi BM25*[1]. You can force[2] *Lucene* to compute a similarity score based on TF-IDF of document and query terms. See *Lucene*'s similarity formula is described here:
http://lucene.apache.org/core/6_6_1/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html.
The scoring function used in *Lucene* is the following:

$$\text{score}(q,d) = \text{coord}(q,d) \cdot \text{queryNorm}(q) \cdot \sum_{t \text{ in } q} \left(\text{tf}(t \text{ in } d) \cdot \text{idf}(t)^2 \cdot \text{t.getBoost}() \cdot \text{norm}(t,d) \right)$$

where q is the query, d a document, t a term, and:

- tf: a function of the term frequency within the document (default: $\sqrt{freq}$);
- idf**:** inverse document frequency of t within the whole collection (default: $\log\left(\frac{numDocs}{docFreq+1}\right)+1$);
- boost: the boosting factor, if required in the query with the " " operator on a given field (if not specified, set to the default field).
- coord: overlapping rate of terms of the query in the given document. Default: $\frac{overlap}{maxOverlap}$
- queryNorm**:** query normalization according to its length; it corresponds to the sum of square values of terms' weight, the global value is multiplied by each term's weight.
- norm: encapsulates indexing time boost and length factors:
  - Field boost - set by calling field.setBoost() before adding the field to a document.
  - lengthNorm - computed when the document is added to the index in accordance with the number of tokens of this field in the document, so that shorter fields contribute more to the score. lengthNorm is computed by the Similarity class in effect at indexing.

[1] BM25: https://en.wikipedia.org/wiki/Okapi_BM25
[2] See https://lucene.apache.org/core/6_6_1/core/org/apache/lucene/index/IndexWriterConfig.html#setSimilarity-org.apache.lucene.search.similarities.Similarity- and https://lucene.apache.org/core/6_6_1/core/org/apache/lucene/search/IndexSearcher.html#setSimilarity-org.apache.lucene.search.similarities.Similarity-

To define the new score based on TF-IDF proceed as following:

1. Create a custom similarity class that inherits the class:
   - *org.apache.lucene.search.ClassicSimilarity*

2. Override and implement default similarity functions:
   - public float tf(float freq)
   - public float idf(long docFreq, long numDocs)
   - public float coord(int overlap, int maxOverlap)

   Note that search time is too late to modify this norm part of scoring. You need to re-index the documents using your specialized similarity class that implements computeNorm().

3. Use the following functions to implement the similarity functions mentioned above:
   - tf : $1+\log freq$
   - idf : log ($\left(\frac{numDocs}{docFreq+1}\right)+1$
   - lengthNorm : 1
   - coord : $\sqrt{\frac{overlap}{maxOverlap}}$

4. Set the custom similarity in Indexer and Searcher using the method setSimilarity(mySimilarity) of *IndexWriter* and *IndexSearcher*. Use the *EnglishAnalyzer* as before.

5. Compute the query compiler program with the *ClassicSimilarity* and new similarity function using the above parameters and show the *ClassicSimilarity* and new rankings/scores for top 10 publications. Describe the effect of using the new parameters.

   Attach your code into the report.