Méthodes d'accès aux données 2021–2022
Professor : Nastaran Fatemi
Assistant : Christopher Meier

Labo 1 : Indexing and Search with Apache Lucene

1 Introduction

1.1 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.

1.2 Organization

The lab should be realized in a group of 2 students (with one group of 3 students).

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

Deadline: See deadline on *Moodle*.

1.3 Setup

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

2 Familiarizing with Lucene

2.1 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 Demo API described at `http://lucene.apache.org/core/8_10_1/demo/overview-summary.html`. The code is available in the `package ch.heigvd.iict.mac.demo` in the downloaded project.

Create an index of the directory `docs` (available as a zip file on *Moodle*) using the `class IndexFiles`. Answer the following questions by inspecting the demo code:

1. What are the types of the fields in the index?
2. What are the characteristics of each type of field in terms of indexing, storage and tokenization?

Use the `class SearchFiles` to do a few simple searches and answer the following questions:

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

REPORT Attach the response to the above questions to your report.

2.2 Using Luke to explore the index

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. Check your answers to questions 3-5 of subsection 2.1.

Note: *Luke* is distributed with *Lucene*. You can download both from https://lucene.apache.org/core/downloads.html (binary release 8.10.1). After extracting the archive, the launch script is available inside the `luke` folder.

3 Indexing and Searching the CACM collection

We are now going to use *Lucene* to index and analyze a list of scientific publications. A text file containing the list 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, separated by ';')
- the title
- the summary (if any)

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 answer the questions in the following subsections by completing the code in the `package ch.heigvd.iict.mac.labo1`.

3.1 Indexing

The goal of this part is to index the CACM publication collection, by completing the `class CACMIndexer`.

You can refer to the slides of the *Lucene* course and the demo example seen in subsection 2.1 for examples of indexing.

Lucene provides some simple field types for common use case. Take a look a the documentation for the `class Field` at https://lucene.apache.org/core/8_10_1/core/org/apache/lucene/document/Field.html. However if finer control over the index is needed, it is required to use the `class FieldType`.

- In the `class Main` use the `StandardAnalyzer` for this part.
- Use the appropriate field types to enable queries on author, title and summary attributes.
- Note that if needed, it is possible to add multiple fields with the same name to a document.
- Keep the publication id in the index and show it in the results set of queries, but do not enable queries on this field.
- For the summary field, store the offsets in the index.

Answer the following questions:

1. Find out what is a "term vector" in *Lucene* vocabulary?
2. What should be added to the code to have access to the "term vector" in the index? Have a look at the different methods of the `class FieldType`. Use *Luke* to check that the "term vector" is included in the

index.
3. Compare the size of the index before and after enabling "term vector", discuss the results.

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

3.2 Using different Analyzers

Lucene provides different analyzers to process a document. Below, we will test some of these analyzers. Note that here indexing and searching will use the same analyzer.

Attention: Remember to rename the index directory to keep the result of each analyzer.

1. Index the publication list using each of the following analyzers:
   - `WhitespaceAnalyzer`
   - `EnglishAnalyzer`
   - `ShingleAnalyzerWrapper`[1] (using shingle size 1 and 2)
   - `ShingleAnalyzerWrapper` (using shingle size 1 and 3, but not 2)
   - `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. Explain the difference of these five analyzers.
3. 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 (e.g. using `System.currentTimeMillis()` before and after the indexing).
4. Make 3 concluding statements bases on the above observations.

REPORT Attach your answers into the report.
Remember to keep the creation of the different analyzer in the submitted code.

3.3 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.

REPORT Attach your code and the answers to the questions into the report.

[1] Shingles are n-grams of words

3.4 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 + ")"`.

REPORT Attach your code in the report.

Write the necessary code to perform the following queries on the `summary` field:

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).

REPORT For each query: provide the text of the query used by `QueryParser`, the total number of results and the top 10 results.

The *Lucene* querying syntax can be found at: http://lucene.apache.org/core/8_10_1/queryparser/org/apache/lucene/queryparser/classic/package-summary.html#package_description.

3.5 Tuning the Lucene Score

The goal of this part 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*[2]. You can force[3] *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/8_10_1/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html.

The scoring function used in *Lucene* is the following:

[2] BM25: https://en.wikipedia.org/wiki/Okapi_BM25
[3] See https://lucene.apache.org/core/8_10_1/core/org/apache/lucene/index/IndexWriterConfig.html#setSimilarity-org.apache.lucene.search.similarities.Similarity- and https://lucene.apache.org/core/8_10_1/core/org/apache/lucene/search/IndexSearcher.html#setSimilarity-org.apache.lucene.search.similarities.Similarity-

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

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: $1 + \log(\frac{docCount+1}{docFreq+1})$);
- t.getBoost: the boosting factor, if required in the query with the " " operator on a given field (if not specified, set to the default field).
- norm: encapsulates indexing time boost and length factors:
  - *Fieldboost* - 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.

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

1. Create a custom similarity class that inherits the class:
   - `import 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 lengthNorm(int numTerms)`

   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` : $\log(\frac{numDocs}{docFreq+1}) + 1$
   - `lengthNorm` : 1
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. Show the top 10 results (with scores) of both similiraty functions. Describe the effect of using the new parameters.

REPORT Attach your code and the answer to the question in part 5 in the report.