Project Information Retrieval – Part 1

Adham Mohamed, Veronica Orsanigo

1 Introduction

Humans have invented categorization, classifications, hierarchical organizational schemes in general to make sense of the perceived complexity of the world. The explosion of the internet and digital repositories has brought a large amount of information and data so that we need more dynamic ways of finding information, also because the new uses of computers such as communication devices, multimedia players, media storage devices require ability to quickly find a specific piece of data, that can be of various type such as images, video, audio files in different formats. So we need a more flexible strategy and ad hoc queries that can quickly move across category boundaries and find exactly what we want. [1]

The code and the complete results on GitHub:

https://github.com/adham95/IR_Assignment

2 Lucene

Lucene is a Java-based Search library, it is used in any application to add search capability to it. It is used to index and search for any kind of text. This library provides the core operation required by a search application: indexing and searching. For this project we use PyLucene, a Python wrapper around Java Lucene, which embeds a Java VM with Lucene into a Python process. PyLucene is built with JCC, a C++ code generator that makes it possible to call into Java classes from Python. PyLucene is closely tracking Java Lucene releases, to support the entire Lucene API. PyLucene API exposes all Java Lucene classes in a flat namespace in the PyLucene module. PyLucene provides Java extension points as proxies for Java to call back into Python implementation. [2]

In this first part we present Lucene's features, which are also the principles on which PyLucene is founded.

Through Lucene it is possible to perform the following operations: [3]

- Acquire content
- Build the document from the raw content
- Analyze the document to decide which parts will be indexed
- Index the document so that it can be retrieved using certain keys
- Build query, prepare a Query object to inquire index database
- Search query, the index database is checked to get relevant details and content documents

- Render results so that they are available to the user in a consumable manner

In this section we describe how Lucene deals with some fundamental issues in information retrieval.

2.1 Indexing

In order to do indexing Lucene adopts these fundamental concepts: index, document, field, term. [4]

- Index -> it contains a sequence of documents
  Lucene uses inverted indexes, since it lists, for a term, the documents that contain it
- Field -> named sequence of terms
  They can be both stored and indexed.
  The text of a field can be tokenized into terms to be indexed or its text can be used as a term itself, one document can have more than one field with the same name, in this case the values of the fields are appended when searching, it's as if the text from all the fields were concatenated and treated as single text field
- Document -> sequence of fields
  They are referred by an integer document number starting from zero for the first document added to an index
- Segment
  A Lucene index can be composed of multiple sub-indexes (segments) that are fully independent indexes, segments are created new for new added documents and are also merged
  Each segment index maintains the following:
  - Fields names: set of field names used in the index
  - Stored field values: it contains, for each document, a list of attribute-value pairs, where the attributes are field names, they are used to store auxiliary information about the document
  - Term dictionary: dictionary containing all the terms used in all of the indexed fields of all of the documents
  - Term frequency data: for each term in the dictionary, the numbers of all the documents that contain it and the frequency of the term in that document
  - Term proximity data: for each term in the dictionary, the positions that it occurs in each document. Note that this will not exist if all fields in all documents omit position data
  - Normalization factors: for each field in each document, a value is stored that is multiplied into the score for hits on that field
  - Term vectors: for each field in each document, it consists of term text and term frequency
  - Delete documents: optional file that indicates which documents are deleted

In our project we created an object doc, instance of Document, for each file xml and we added three fields: "name" for the name of the file where the question and answers were contained, "path" for the path of the file, "contents" for the actual content of the file, so

the question (already without title) and the answers from Stackoverflow. The structure of the xml files before and after parsing is shown in part 3.

We can perform indexing using the following main classes:

- IndexWriter -> it creates a new index or opens an existing one, and adds, removes or updates documents in the index
- Directory -> the location of Lucene index
- Analyzer -> it is specified in the IndexWriter constructor, it extracts tokens out of text that should be indexed and eliminates the rest
- Document
- Field

The indexing is done through the Indexer.

2.2 Parser and Analyzer

Before text is indexed, it is passed through an analyzer, that extracts the tokens out of the text that has to be indexed and eliminates the rest. There are several implementations of the abstract class Analyzer, some of them deal with skipping stop words, some with splitting the text following white spaces, some with conversion of tokens to lowercase letters and so on (WhitespaceAnalyzer, SimpleAnalyzer, StopAnalyzer, StandardAnalyzer). If we are indexing rich documents, which contain metadata for example, this metadata should be separated in different fields, so we need a parser to do it, since the analyzer works on one specific field at time. [1]

In our project we used StandardAnalyzer, that is Lucene's more sophisticated core analyzer. It can identify certain kinds of tokens, it lowercases each token and removes stop words and punctuation.

2.3 Searcher

The second important operation is searching, the main classes needed to perform it are:

- IndexSearcher -> to search what IndexWriter indexed, it accesses the directory where indexes are stored using IndexReader, it accepts Query objects and returns a TopDocs object. There are different searching methods, the simplest takes a Query object and an int topN count as parameter and returns a TopDocs object
- Term -> basic unit for searching, it consists of a pair of string elements: name of the field and word (text value) of that field
- Query -> there are several different Query subclasses, that we will present later
- TopDocs -> simple container of pointers to the top N ranked search results (documents that match a given query). [1] [5]

2.4 Scoring

Lucene first uses the Boolean model to narrow down the documents that need to be scored based on the use of boolean logic in the Query specification and then ranks this subset of matching documents using different information retrieval models: [6]

- Vector Space Model (VSM)

- Probabilistic Models (Okapi BM25 and DFR)
- Language models

Scoring is dependent on the way documents are indexed. Lucene scoring works on fields and then combines results to return documents. Lucene, by default, returns documents reverse-sorted by this score, meaning the top documents are the best matching ones. Lucene also allows influencing search results by "boosting" at different times:

- Index-time boost before a document is added to the index
- Query-time boost for a query clause

Lucene can influence scoring and so determine weights terms at index-time and at query-time using similarity. It uses the class Similarity to do this, that has some subclasses such as:

- TFIDFSimilarity and its subclass DefaultSimilarity for the VSM
- BM25Similarity for the Probabilistic Model

2.5 Query

Query objects can be constructed from free text (the query from the user) thanks to the use of the front end QueryParser. [1]

There are different Query types in Lucene, whose instances are used to call one of IndexSearcher's methods, used for querying. Here some of them:

- TermQuery -> search an index for a specific term, that is the smallest indexed piece, consisting of a field name and a text-value pair.
- BooleanQuery -> the other query types can be combined using BooleanQuery, which is a container of Boolean clauses (subqueries).
- PhraseQuery -> index contains positional information of a term, PhraseQuery uses this information to locate documents where terms are within a certain distance of one another.
- WildcardQuery -> for terms with missing pieces but still find matches, it works with single terms, not with phrase queries.

The complete list can be found following point 1 of bibliography.

2.6 Spell correction

Lucene can compute the spell correction of queries thanks to the class SpellChecker, which is based on the string distance, for which exists the class StringDistance. [7]

The main idea to compute spell checking is to generate a suggestion list, Lucene's spellchecker module derives a dictionary from the search index to gather all unique terms seen during indexing from a particular field. Later it is necessary to enumerate the suggestions, the approach of the spellchecker module is to use letter n-grams to identify similar words, the last step is to select the best suggestion for each term in the user's query. [1]

2.7 Query expansion and relevance feedback

Lucene offers different possibilities for query expansion and relevance feedback:

- From package org.apache.lucene.wordnet that uses synonyms defined by WordNet, a large lexical database in English [8]:
  - Class SynExpand to expand a query looking up synonyms for each term, it is used together with Syns2Index because it is necessary to convert the prolog file wn_s.pl from the WordNet prolog download into a Lucene index [9]
- Framework LucQE Lucene Query Expansion Module to add search terms to a user's search to improve precision or recall. The additional terms may be taken from a thesaurus.
  These two modules have been implemented: Rocchio Query Expansion (QE) method and gQE [geek] (provides implementation of pseudo feedback QE utilizing Google's web API to query the world wide web to acquire terms for QE). [10]

2.8 Sorting

Lucene sorts the matching documents in descending relevance score order, such that the most relevant documents appear first, but it offers also other ways to do it. [1]

- by field value
- by relevance
- by index order
- by a field
- reversing sort order
- by multiple fields

3 Data and results

In this section we present how Lucene helps us to index a collection of documents and to search for them using queries.

3.1 Collection of documents

The collection of documents we work on is composed of 1 859 859 xml files which contain questions and answers from Stackoverflow about Python and C++. Each file presents one question and zero or more answers.

Here one example of how one document appears:

<question>

<Title>render a jinja2 template object in pylons with filters</Title>

*<Body><p>I'm working on a pylons project that uses jinja2 as its template engine. The project has lots of custom filters added into the template engine.</p>

<p>I have a template*

*object:</p>

<pre><code>>>> [..]</Body>*

<Tags>python,pylons,jinja2</Tags>

</question>

<answer>

<Body><p>You can load the filters into the application environment. Check out this post for Pylons-specific instructions on how to do so […]</Body>

</answer>

Every file starts with one question (included within <question> and </question>) and zero or more answers, each included in <answer> and </answer>. Questions have sub-elements title, body, and tags. In the body text there are special symbols. Furthermore, there is one special tag <code> that the users of StackOverflow can use to typeset code fragments.

3.2 Code and procedure

- File parser.py

  Before creating the indexer and the searcher, we parsed the content of these files in order to remove special symbols and XML tags, to obtain plain text and simpler to index.
  We also separated titles from the rest, in order to use them in different ways for indexing and searching.
  Here an example of one document after parsing:

  An MFC alication that I'm trying to migrate uses codeafxext.h, which causes code_AFXDLL to get set, which causes this error if I set code/MT:blockquote Please use the /MD switch for _AFXDLL builds/blockquoteMy research to date indicates that it is imssible to build an alication for execution on Windows NT 4.0 using Visual Studio (C++, in this case) 2005.Is this really true? Are there any workaround available?

  c++,visual-studio,comtibility,windows-nt

  No, there are many alications built with VS2005 that have to suort Windows XP, 2000, NT, the whole stack. The issue is that (by default) VS2005 wants to use libraries/exrts not esent on NT.See a href=htt//www.mombu.com/microsoft/windows-ogrammer-win32/t-vs2005-and-nt4-392831.html rel=noreferrerthis thread/a for some background.Then start limiting your dendencies via eocessor macros, and avoiding APIs which aren't suorted on NT. […]

There are two main parts that we created: files index.py to index documents and search.py to make queries and find documents that contain what we are looking for.

- File index.py

For this file we refer to the sample provided by the official Apache web site [11].
It is based on the Lucene (java implementation) class org.apache.lucene.demo.IndexFiles. It takes a directory as an argument and indexes all the files in it, recursively. It indexes on the file path, the file name and the file contents (text without titles), that are the fields of the Lucene document. The resulting Lucene indexes are placed in the current directory 'index'.
Fundamental elements here are the *analyzer* and the *indexWriter*, the first one is used to tokenize the text and the second one to create and maintain the indexes. We used *StandardAnalyzer*, that is Lucene's most sophisticated core analyzer, it has quite a bit of logic to identify certain kinds of tokens and it lowercases each token and removes stop words and punctuation.
Code to create the analyzer (that limits the maximum number of tokens per field) and the writer:

analyzer = LimitTokenCountAnalyzer(analyzer, 1048576)

writer = IndexWriter(store, config)

We defined two FieldType (t1, t2), the first one for the name and path of the files to index, the second one for their contents. So we created Lucene documents with their fields and so we did the indexing. To do it on our collection of 1 859 859 documents it took us 7 hours 29 minutes and 36 seconds.

- File search.py

For this file we refer to the sample provided by the official Apache web site [12].
It prompts for a search query, then it searches for the Lucene index in the current directory 'index' for the search query related to the 'contents' field, it also displays the 'path' and 'name' fields for each of the hits it finds in the index.
Fundamental elements here are the *analyzer* for the queries, the *directory* where there are the indexes, the *searcher* to look at the indexes in the directory, the *QueryParser* to interpret any sort of valid QueryString into a Lucence query (its first parameter says in which field look for the query).

analyzer = StandardAnalyzer()

searcher = IndexSearcher(DirectoryReader.open(directory))

query = QueryParser("contents", analyzer).parse(command)

We are able to search for single words or sequences and for wildcards and to decide how many documents, at most, we want to retrieve for each query.

To evaluate the performance of the search engine we are interested in precision and recall, the main issue correlated to this is that we need to know which are the relevant documents for each query, but we don't have this information, since we don't have a feedback from users and so we can't know, for example, which are the documents more visited, considered more useful or important.

To provide for this lack of *ground truth*, that means a set of correct answers for the queries, we decided, as suggested, to create manually the set, asserting that the correct document for each query is the one that has that query as title. So we searched for a title (query) using as field only the contents, removing the titles from the XML files in order to make it more realistic.

We proceeded with 3 experiments to examine our system:

1. We calculated precision and recall using the first 6 documents retrieved by the system, looking at them and reading their content to decide if they were correct answers for the query or not (manual labelling). Another way to evaluate if they are correct answers could be looking just at the title of the documents, but we thought that checking directly the answers would have been more precise and effective.

   Formulas of precision and recall:

   $P = | REL \cap RETR | / | RETR |$

   $R = | REL \cap RETR | / | REL |$

   Since we don't know how many the relevant documents for a query are, we can just put an "X" as denominator in the recall formula, since the numerator is sufficient to compare recall for two different systems.

   In particular we observed how these parameters changed using different Similarity in searching.

   Lucene, as default, uses BM25Similarity (roots in probabilistic information retrieval), we compared the results from it to the ones using ClassicSimilarity (modified tf_idf).

2. Another check we did was to see if the document with the title corresponding to the query was among the first results, so its position. We took the top-50 documents retrieved for each of the 100 queries we considered and we made an average of the position of the documents that have the query as title. We used also this method to compare BM25Similarity and ClassicSimilarity, but on larger-scale.

3. Finally we tried to understand how much deleting the title makes us lose information for finding matches with the query. We took the top-10 documents retrieved using a title as query, but keeping their titles, in this way we were supposed to find documents that matched better with the query, since we did not lose the information from the title. Then we retrieved documents as usual, so using documents without titles. Finally we observed how many documents were in both the sets of the top-10 retrieved documents.

3.3 Examples and performance

1. We decided, as start, to manually label some documents, we chose some titles as queries, we retrieved the top-10 documents and we examined the top-6 results, deciding which ones were relevant or not relevant.

   Y = correct answer for the query

   N = not correct answer for the query

   Red = document with title equal to the query

Similarity	Document of title	Top-10	Query
	Q1171		What is the most efficient graph data structure in Python?
BM25		Q1171.xml Y Q852334.xml N Q262232.xml N Q10025584.xml N Q1005494.xml Y Q902910.xml N Q463240.xml Q1094215.xml Q238008.xml Q872290.xml	
Classic		Q983855.xml N Q1397572.xml N Q1398445.xml N Q10005851.xml N Q10100399.xml N Q1119497.xml N Q1325673.xml Q4942.xml Q339217.xml Q35988.xml	
	Q10880		Any good advice on using emacs for C++ project?
BM25		Q671412.xml Y Q557555.xml N Q79210.xml Y Q10880.xml Y Q1078069.xml N Q1416882.xml N Q445595.xml Q623040.xml Q1231397.xml Q495579.xml	
Classic		Q79210.xml N Q1190595.xml N Q10880.xml Y Q24109.xml N Q944091.xml N Q649789.xml N Q671412.xml Q396074.xml	

		Q557555.xml Q1210700.xml	
	Q2933		How can I create a directly-executable cross-platform GUI app using Python?
BM25		Q818736.xml N Q1147199.xml N Q2933.xml Y Q1459087.xml N Q205062.xml N Q1060679.xml N Q450136.xml Q5313.xml Q716524.xml Q796364.xml	
Classic		Q1147199.xml N Q702395.xml N Q824458.xml N Q1140995.xml N Q101754.xml N Q32404.xml N Q818736.xml Q1060679.xml Q434583.xml Q789856.xml	

We showed the top-10 documents retrieved to see if the document corresponding to the title used as query appears in that set, but we calculated precision and recall considering the top-6 documents.

We can observe that generally BM25Similarity works better than ClassicSimilarity, as shown by precision and recall related to each query.

As denominator for the recall we use "X" since we don't know the number of the relevant documents.

For precision we can not do it, since the denominator changes retrieving a different number of documents, so we can say that we actually calculate precision@6.

Queries:

- "What is the most efficient graph data structure in Python?"

  BM25:
  Precision = 2/6
  Recall = 2/X
  Classic:
  Precision = 0/6
  Recall = 0/X

- "Any good advice on using emacs for C++ project?"

  BM25:

Precision = 3/6
Recall = 3/X
Classic:
Precision = 1/6
Recall = 1/X

- "How can I create a directly-executable cross-platform GUI app using Python?"

  BM25:
  Precision = 1/6
  Recall = 1/X
  Classic:
  Precision = 0/6
  Recall = 0/X

2. We examine here how the position of the relevant document (the one that has the query as title) changes using different kinds of similarities.
   We used 100 queries and, within a loop, we obtained that position (starting from 0) for each of them, so we made an average. We considered the top-50 documents retrieved. We observed that only for few queries the relevant document did not appear in the top-50, in that case we decided to consider 50 as position, since it did not hurt too much our comparison between different similarities.
   Our results said that most of the times the relevant document is in first position or, however, among the first positions.

   BM23Similarity: average postion = 15.2
   ClassicSimilarity: average position = 16.9

   We can observe how these results confirm the conclusion of the previous experiment: BM25Similarity seems to work better than ClassicSimilarity.

   We show here the results for two of the 100 queries we used, both for BM25Similarity and ClassicSimilarity, to clarify our approach.

- BM25 SIMILARITY

  FILE NAME: Q171765.xml
  QUERY: "What is the best way to get all the divisors of a number"
  LIST OF DOCUMENTS RETRIEVED
  ['Q171765.xml', 'Q942198.xml', 'Q1280229.xml', 'Q1010381.xml',
  'Q571488.xml', 'Q134834.xml', 'Q1342975.xml', 'Q567222.xml',
  'Q248400.xml', 'Q875027.xml', 'Q157039.xml', 'Q1179461.xml',
  'Q175544.xml', 'Q618169.xml', 'Q884650.xml', 'Q566846.xml',
  'Q684684.xml', 'Q376296.xml', 'Q526070.xml', 'Q1233381.xml',
  'Q601430.xml', 'Q407587.xml', 'Q729905.xml', 'Q799434.xml',
  'Q701402.xml', 'Q556730.xml', 'Q734754.xml', 'Q117429.xml',
  'Q283297.xml', 'Q1436351.xml', 'Q8948.xml', 'Q1089936.xml',

'Q1083105.xml', 'Q785202.xml', 'Q1155617.xml', 'Q150355.xml',
'Q1003965.xml', 'Q986847.xml', 'Q1428786.xml', 'Q480178.xml',
'Q860602.xml', 'Q359903.xml', 'Q360338.xml', 'Q1154494.xml',
'Q1096003.xml', 'Q625314.xml', 'Q772817.xml', 'Q1218922.xml',
'Q332852.xml', 'Q554204.xml']
POSITION OF DOCUMENT: 0

FILE NAME: Q678236.xml
QUERY: "How to get the filename without the extension from a path in Python"
LIST OF DOCUMENTS RETRIEVED
['Q678236.xml', 'Q959837.xml', 'Q377017.xml', 'Q541390.xml',
'Q120656.xml', 'Q1027714.xml', 'Q599953.xml', 'Q51520.xml',
'Q646955.xml', 'Q51949.xml', 'Q225735.xml', 'Q837606.xml',
'Q542596.xml', 'Q918154.xml', 'Q50499.xml', 'Q221185.xml',
'Q67631.xml', 'Q1176624.xml', 'Q43580.xml', 'Q904170.xml',
'Q775351.xml', 'Q660160.xml', 'Q200737.xml', 'Q82831.xml',
'Q1122924.xml', 'Q757933.xml', 'Q881639.xml', 'Q889333.xml',
'Q185936.xml', 'Q140758.xml', 'Q993265.xml', 'Q524137.xml',
'Q595305.xml', 'Q959168.xml', 'Q879530.xml', 'Q352837.xml',
'Q714063.xml', 'Q1001538.xml', 'Q389398.xml', 'Q973473.xml',
'Q894802.xml', 'Q40705.xml', 'Q974821.xml', 'Q973231.xml',
'Q955504.xml', 'Q168409.xml', 'Q437589.xml', 'Q740836.xml',
'Q375154.xml', 'Q242065.xml']
POSITION OF DOCUMENT: 0

- CLASSIC SIMILARITY

FILE NAME: Q171765.xml
QUERY: "What is the best way to get all the divisors of a number"
LIST OF DOCUMENTS RETRIEVED
['Q942198.xml', 'Q171765.xml', 'Q1280229.xml', 'Q571488.xml',
'Q1010381.xml', 'Q1210099.xml', 'Q566846.xml', 'Q1195656.xml',
'Q700905.xml', 'Q248400.xml', 'Q1179461.xml', 'Q134834.xml',
'Q1342975.xml', 'Q1082078.xml', 'Q1435442.xml', 'Q571538.xml',
'Q1087694.xml', 'Q466692.xml', 'Q567222.xml', 'Q487283.xml',
'Q867602.xml', 'Q538300.xml', 'Q480178.xml', 'Q1316682.xml',
'Q553372.xml', 'Q780057.xml', 'Q603595.xml', 'Q684684.xml',
'Q1469421.xml', 'Q360338.xml', 'Q756630.xml', 'Q1380985.xml',
'Q1233381.xml', 'Q558539.xml', 'Q477183.xml', 'Q618169.xml',
'Q1154494.xml', 'Q1096003.xml', 'Q1446137.xml', 'Q1384235.xml',
'Q1075712.xml', 'Q799434.xml', 'Q1140194.xml', 'Q807979.xml',
'Q979397.xml', 'Q884650.xml', 'Q788699.xml', 'Q1138024.xml',
'Q722992.xml', 'Q656946.xml']
POSITION OF DOCUMENT: 1

FILE NAME: Q678236.xml
QUERY: "How to get the filename without the extension from a path in Python"
LIST OF DOCUMENTS

['Q1176624.xml', 'Q541390.xml', 'Q678236.xml', 'Q959837.xml',
'Q660160.xml', 'Q991813.xml', 'Q51520.xml', 'Q1027714.xml',
'Q51949.xml', 'Q599953.xml', 'Q757933.xml', 'Q837606.xml',
'Q542596.xml', 'Q1299855.xml', 'Q1157134.xml', 'Q1122924.xml',
'Q225735.xml', 'Q775351.xml', 'Q959168.xml', 'Q470567.xml',
'Q904170.xml', 'Q974821.xml', 'Q120656.xml', 'Q375154.xml',
'Q595305.xml', 'Q242065.xml', 'Q973231.xml', 'Q647500.xml',
'Q918154.xml', 'Q969553.xml', 'Q50499.xml', 'Q377017.xml',
'Q881639.xml', 'Q955504.xml', 'Q562519.xml', 'Q67631.xml',
'Q210247.xml', 'Q40705.xml', 'Q1383863.xml', 'Q524137.xml',
'Q1135499.xml', 'Q1457308.xml', 'Q1430446.xml', 'Q1038070.xml',
'Q1359090.xml', 'Q1299018.xml', 'Q1130402.xml', 'Q616480.xml',
'Q1271337.xml', 'Q659847.xml']
POSITION OF DOCUMENT: 2

3. We check here how many documents are in both the results obtained keeping the titles of the documents and deleting them. We decided to compare the top-10 documents obtained.

   We used, as always, the titles as queries.

   Queries:

- "Alpha blending sprites in Nintendo DS Homebrew"
  Document: Q7209

  Searching in documents with titles

```
name: Q7209.xml score:  60.4775276184082 Doc ID : 19334
name: Q214687.xml score:  23.312545776367188 Doc ID : 15253
name: Q1284205.xml score:  22.395992279052734 Doc ID : 22582
name: Q187057.xml score:  21.814754486083984 Doc ID : 3418
name: Q759778.xml score:  21.173290252685547 Doc ID : 3378
name: Q993922.xml score:  21.13459014892578 Doc ID : 19057
name: Q1185702.xml score:  20.179750442504883 Doc ID : 23312
name: Q1247921.xml score:  19.980066299438477 Doc ID : 5990
name: Q1192032.xml score:  19.766178131103516 Doc ID : 669
name: Q1100917.xml score:  19.707029342651367 Doc ID : 8913
```

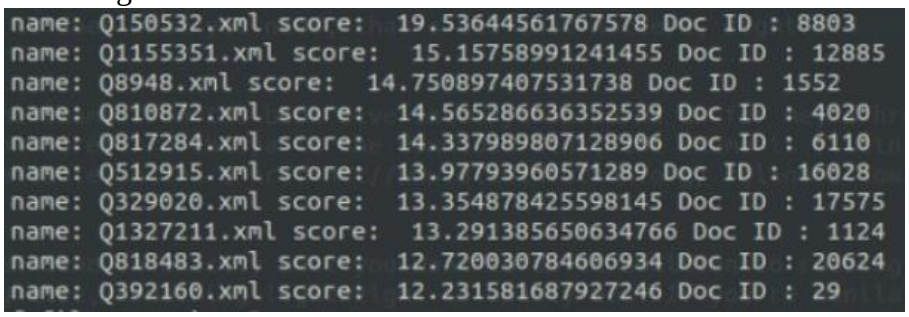
  Searching in documents without titles

```
name: Q7209.xml score:  49.46549606323242 Doc ID : 19334
name: Q214687.xml score:  23.05401039123535 Doc ID : 15253
name: Q1284205.xml score:  22.056758880615234 Doc ID : 22582
name: Q187057.xml score:  21.453771591186523 Doc ID : 3418
name: Q759778.xml score:  21.13924789428711 Doc ID : 3378
name: Q1185702.xml score:  20.26767349243164 Doc ID : 23312
name: Q1192032.xml score:  19.941814422607422 Doc ID : 669
name: Q993922.xml score:  19.112146377563477 Doc ID : 19057
name: Q960176.xml score:  18.45117950439453 Doc ID : 12508
name: Q1003497.xml score:  18.034679412841797 Doc ID : 187
```

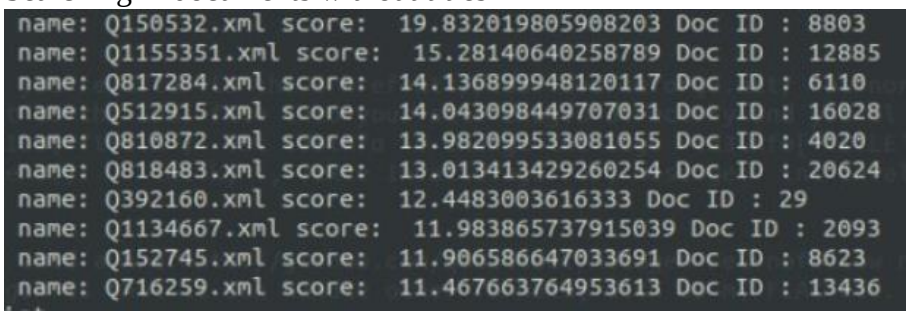
  Intersection = 8/10 documents

- "Accessing mp3 Meta-Data with Python"
  Document: Q8948

Searching in documents with titles



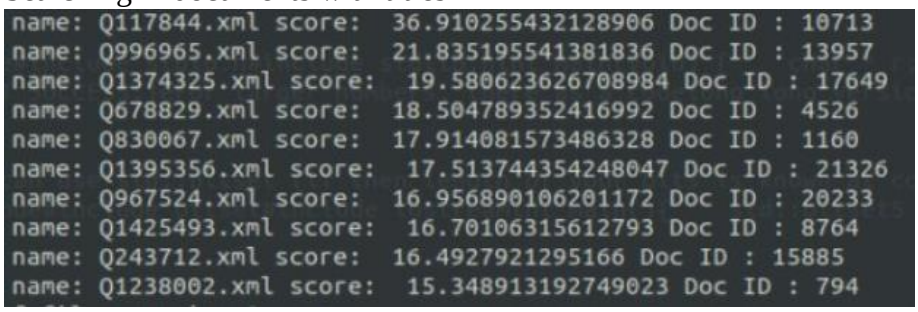
Searching in documents without titles



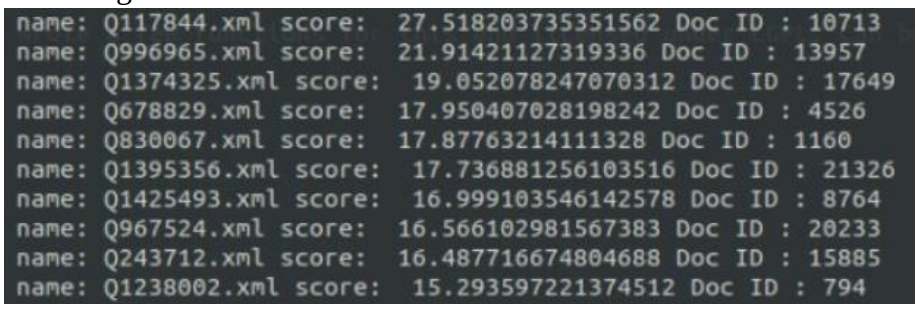
Intersection = 7/10 documents

- "Converting string of 1s and 0s into binary value"
  Document: Q117844

  Searching in documents with titles



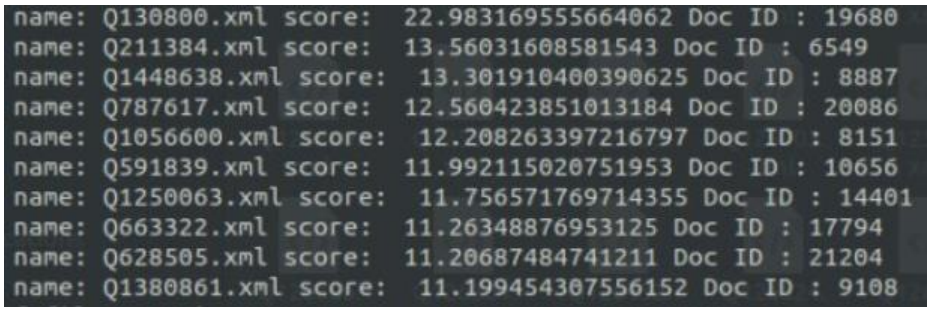
  Searching in documents without titles



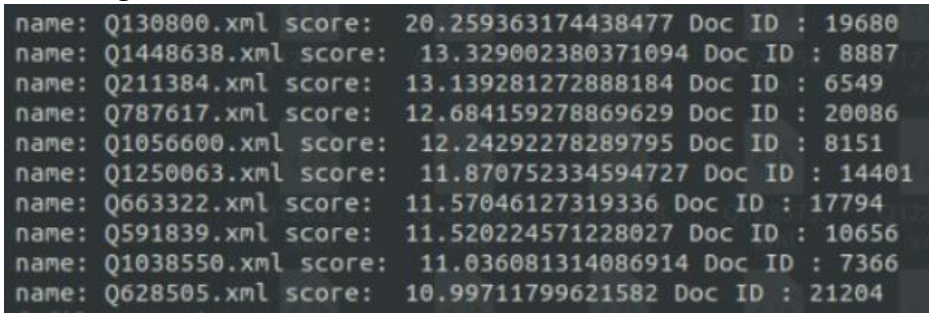
  Intersection = 10/10 documents

- "Data visualization in desktop applications"
  Document: Q130800

  Searching in documents with titles



Searching in documents without titles



Intersection = 9/10 documents

With this experiment we can observe how keeping or removing titles changes the results, we tried using 15 queries and we saw that the intersection was between 7 and 10 documents.
This means that the title contains some information, but, even if we remove it, we manage to keep a large part of information.
Moreover, looking at the contents of the documents, we saw that best matching results depend mostly on the query, not on keeping or removing titles, since some queries are more appropriate and effective to find correct results for them.

4 Conclusion

Lucene offers the possibility to create a system to search for information among a huge number of documents and we saw how the two main functions to do this (indexing and searching) work.

Indexing makes us able to look for a correspondence to a query in a faster way than simply scan the files as given.

Searching permits to find answers to a particular query, it's important to set it in order to have the best results.

Analyzing our system and our collection of data, we can discuss the limits that it presents.

- Our approach of removing the titles makes us lose part of the information, since titles are informative themselves, we can observe how the results change looking at the experiment 3.
- Assuming that the document which has the title corresponding to the query is the correct answer, we may lose other documents that could be even more relevant for that query, in particular this observation is important for the experiment 2, where we are interested only in that singular document, ignoring other possible correct results.

- Our method of reading the contents of the top-k retrieved documents to decide if they are correct answers or not is manual, but necessary since we don't have a feedback by users. The situation would be different if users started to use the system, indeed the way to decide which the relevant documents for a query are could be to use directly the information from users, who could express their opinion about each document they look at.
- Another problem in our "manual" evaluation is that it is not possible to use so many documents to make an average, but we are able to work on a little number of them if compared with the total amount of almost 2 millions of elements. We deal with this problem through experiment 2, that allows us to do a larger-scale test, even if less accurate than the manual evaluation.

Bibliography

[1] E. Hatcher, O. Gospodnetic', M. McCandless, Lucene in action, Stamford: Sebastian Stirling, 2010.

[2] http://lucene.apache.org/core/8_3_0/demo/overview-summary.html#About_the_code.

[3] https://www.tutorialspoint.com/lucene/lucene_overview.htm.

[4] https://lucene.apache.org/core/8_3_0/core/org/apache/lucene/codecs/lucene80/package-summary.html#package.description.

[5] https://lucene.apache.org/core/8_3_0/core/org/apache/lucene/search/package-summary.html#package.description.

[6] https://lucene.apache.org/core/8_3_0/core/org/apache/lucene/search/similarities/Similarity.html.

[7] https://lucene.apache.org/core/8_3_0/suggest/org/apache/lucene/search/spell/SpellChecker.html.

[8] https://wordnet.princeton.edu/.

[9] https://lucene.apache.org/core/3_0_3/api/contrib-wordnet/org/apache/lucene/wordnet/package-summary.html.

[10] http://lucene-qe.sourceforge.net/.

[11] https://svn.apache.org/viewvc/lucene/pylucene/trunk/samples/IndexFiles.py?view=markup.

[12] https://svn.apache.org/viewvc/lucene/pylucene/trunk/samples/SearchFiles.py?view=log.