Implementing a Search Engine – Part 2

Objective

We are going to develop a complete search engine for HTML pages. We will create an index from a document corpus, then we will run queries in batch and interactive mode.

The recommended programming language is Java, as we provide templates for the code. If you prefer to do it in another language, feel free to do it.

Search Engine Structure

In Aula Global you have the template for the code. As it is, you can run 3 different functionalities:

```
> java ti.SearchEngine
Usage: ti.SearchEngine <command> <options>

where <command> and <options> are one of:
  - index <path-to-index> <path-to-collection> [<path-to-stopwords>]
  - batch <path-to-index> <path-to-queries>
  - interactive <path-to-index>
```

The class `SearchEngine` is the entrance point to the program. It just checks the input parameters and delegate the execution to the classes `Indexer`, `Batch` e `Interactive`, the ones in charge of running the search engine. The class `Index` contains the data structures of the index. An index is created by `Indexer`, and it will be used for retrieval from `Batch` and `Interactive`.

Also, the interface `DocumentProcessor` define the basic operations of a document processor for extracting the terms that will be included in the index. For this purpose, you can first use the class `SimpleProcessor` but it will be later replaced by `HtmlProcessor` to process HTML pages. The interface `RetrievalModel` defines basic operations for a retrieval model.

All the code goes with comments and we also provide the javadoc. Please read carefully the documentation to understand how everything should work.

Tuples

Most of the data stored in the index and used by the search engine are defined using tuples, i.e., ordered pairs of values. For example, there are tuples (document, norm), (term, IDF), (term, weight), (document, similarity), etc. The class `Tuple<T1,T2>` implements this structure for two values of any type.

```
int docId = 34;
double sim = 0.843;
// Create a tuple (int, double)
Tuple<Integer, Double> result = new Tuple<>(docId, sim);
System.out.println(result.item1); // 34
result.item2 = 0.23;
System.out.println(result.item2); // 0.23
```

HashMap<K,V>

Another used structure is `HashMap<K,V>` from Java, which maps keys of type K to values of type V (they can be seen as tuples `Tuple<K,V>`). It is a very efficient data structure with access cost O(1), so they are used to know which value corresponds to a specific key.

```
// Map Strings to int
HashMap<String, Integer> hm = new HashMap<>();
hm.put("car", 4);
hm.put("red", 32);
Integer a = hm.get("car"); // 4
Integer b = hm.get("red"); // 32
Integer c = hm.get("house"); // null
boolean d = hm.containsKey("car"); // true
// Go through all pairs key-value
for (Map.Entry<String, Integer> e : hm.entrySet()) {
    String clave = e.getKey();
    int valor = e.getValue();
    e.setValue(90);
}
```

Index

The index contains several data structures in the class `Index`:

- `HashMap<String, Tuple<Integer, Double>> vocabulary`: maps one term to one tuple (termID, IDF).
- `ArrayList<Tuple<String, Double>> documents`: the i-th element corresponds with the document docID=i. The value is a tuple (docName, norm).
- `ArrayList<ArrayList<Tuple<Integer, Double>>> invertedIndex`: the element i-th corresponds to the term termID=i. The value is the posting list of that term: each element is a tuple (docID, weight), identifying the weight of this term in this document. This structure will be used for retrieval.
- `ArrayList<ArrayList<Tuple<Integer, Double>>> directIndex`: the element i-th corresponds to the document docID=i. The value is the posting list of that document: each element is a tuple (termID, weight), identifying the weight of that term in that document.

So, both documents and terms will be identified with a numeric value `int`.

Also, the index offers a cache to set and get the clean version of a document already processed. It will be used for the interactive retrieval in order to show the snippets. The cache version of a document is an object `Tuple<String, String>` that contains the title and the main content of the document.

Weights

The weights equation follows TFxIDF:

- $tf_{td} = 1 + \log(f_{td})$
- $idf_t = \log\left(1 + \frac{n_d}{c_t}\right)$

  f_{td} is the frequence of the term in the document. n_d is the number of documents; c_t is the number of documents containing the term. Therefore, the weights are formulated as $w_{td} = tf_{td} \cdot idf_t$.

Similarity

The similarity is calculated using the cosine similarity:

$$sim(\vec{d},\vec{q}) = \frac{\sum_t w_{td} \cdot w_{tq}}{\sqrt{\sum_t w_{td}^2}\sqrt{\sum_t w_{tq}^2}}$$

TO DO

The objective of this lab is to implement the text processor of the search engine. This processor will extract the terms of given text (HTML document or query), as follows:

1. Parse to extract HTML labels. You should split the title text from the body text.
2. For each text:
   a. Tokenize them (divide it into words)
   b. Normalize the terms (convert to non-capital letters)

   c. Filter stopwords
   d. Reduce to the stem

Implementation

You should code the class `HtmlProcessor`. By default `SimpleProcessor` is selected for processing the text, so you must change it now to `HtmlProcessor` in the class `SearchEngine` (for retrieval you do not need stopwords, so you can just instantiate with null).

`HtmlProcessor` implements two public methods to process text:

1. The first one is `Tuple<String, String>` **`parse`**`(String html)`, that receives the complete text of the document and extracts the real text from the HTML structure. It returns the title of the page and the body.
2. The second one is `ArrayList<String>` **`processText`**`(String text)`, that receives the text already parsed and returns the list of terms already processed and ready to be indexed. The whole process is implemented using the followind methods:
   - `ArrayList<String> tokenize(String text)`: receives the text already parsed and returns the list of tokens.
   - `String normalize(String text)`: receives one token and returns the normalized term.
   - `boolean isStopword(String term)`: returns `true` if the term is stopword, and `false` if not.
   - `String stem(String term)`: receives one term and returns its stem.

Use

The `HtmlProcessor` will be used as follows. The classes `Batch` and `Interactive` run the method `processText` with the query, so you do not need to parse. However, in `Indexer` you should `parse` first to remove the HTML tags and split the title from the body. The terms to be included in the index come from the title and the body, so you should run `processText` over both of them.

You can find a file with stopwords in Aula Global (`stop-words.txt`), that you can use for indexing. You can also find a collection of documents “**2011-documents**” that you can use for testing.

```
> java ti.SearchEngine index 2011-myIndex 2011-documents stop-words.txt
Running first pass...
  Indexing file 2011-00-002.html...done.
  Indexing file 2011-00-003.html...done.
  Indexing file 2011-00-025.html...done.
[...]
  Indexing file 2011-99-111.html...done.
  Indexing file 2011-99-120.html...done.
...done:
  - Documents: 2088 (96,11 MB).
  - Time: 101,6 seconds.
  - Throughput: 0,95 MB/s.
Running second pass...
  Updating term weights and direct index...done.
  Updating document norms...done.
...done
  - Time: 0,3 seconds.
Saving index...done.
Index statistics:
  - Vocabulary: 92319 terms (1,84 MB).
  - Documents: 2088 documents (43,04 KB).
  - Inverted: 13,89 MB.
  - Direct: 0 MB.
  - Cache: 0 MB.
```

Details

Parsing

- As we have HTML documents, we need to parse them to remove HTML labels and comments and return a tuple containing the title and the body.
- We recommend the use of the Jsoup library (`http://jsoup.org`).

Tokenizing

- We should split the sentences in its corresponding words.
- Be careful with symbols and numbers.

Normalization

- Remove Capital letters
- You could also normalize numbers or symbols, but it’s not the aim of the lab.

Stopwords

- Meaningless words.
- It’s important to remove them to improve the efficiency.
- You can use the list `stop-words.txt` (Aula Global)
- Use an efficient structure for the stopwords: `HashSet<String> stopwords;`

Stemming

- The most usual stemmers are Porter and Krovetz.
- You can use the provided Porter stemmer Stem.java (Aula Global) or choose another, but consider that it should work for English.

Suggestions and Requirements

- For parsing and stemming use external libraries.
- Start with just few documents to test the code.
- You should be able to process all 2011 documents in a couple of minutes.
- You should just fill the code where the comment `// P2` is placed.

Using a new dataset

The objective is to test your implementation of the search engine with the data set resulting from the crawler you implemented in the first part of the lab.

Just collect a small set of webpages that allows you to know which ones are relevant for a small subset of queries that you are going to define.

Basically, you should follow the next steps:

- Generate a small subset of web pages with your own crawler (10-20).
- Design a short list of queries (start by one, and then go for 5) and identify the relevant documents for those queries.
- Run your search engine in interactive mode and check if the relevant documents get the best similarities.

Evaluation of the search engine

Once we have implemented the retrieval system, we should evaluate its effectiveness. For that purpose, we provide an evaluation tool, a file with queries and a file containing relevance judgements (qrels).

First, you should generate the run file using **SimpleProcesor**:

```
> java ti.SearchEngine batch 2011-index 2011-topics.xml > 2011.run
Loading index...done. Statistics:
  - Vocabulary: 209732 terms (5,05 MB).
```

```
     - Documents: 2088 documents (43,04 KB).
     - Inverted: 17,95 MB.
     - Direct: 0,01 MB.
   > cat 2011.run

   2011-001     Q0      2011-72-101     1       0.2202240167408362      sys
   2011-001     Q0      2011-13-109     2       0.1932897248909958      sys
   2011-001     Q0      2011-05-064     3       0.16377197192626178     sys
   2011-001     Q0      2011-36-150     4       0.15096442132660134     sys
   2011-001     Q0      2011-75-005     5       0.1499282777439169      sys
```

Then, we should run the evaluation tool:

```
   > java -jar ireval.jar
   usage: java -jar ireval.jar TREC-Ranking-File TREC-Judgments-File
```

that receives the run file from the system and the file containing the qrels. As output, it will show the effectivity for each query, and the the average for all the queries. For example:

```
   > java -jar ireval.jar 2011.run 2011.qrel

num_ret                    2011-001        126

num_rel                    2011-001         70

num_rel_ret                2011-001         70

P@5                        2011-001     0.8000

P@10                       2011-001     0.8000

P@15                       2011-001     0.7333

P@20                       2011-001     0.7500

P@30                       2011-001     0.6333

P@50                       2011-001     0.7200

P@100                      2011-001     0.6500

P@200                      2011-001     0.3500

P@500                      2011-001     0.1400

R@5                        2011-001     0.0571

R@10                       2011-001     0.1143

R@15                       2011-001     0.1571

R@20                       2011-001     0.2143

R@30                       2011-001     0.2714

R@50                       2011-001     0.5143

R@100                      2011-001     0.9286

R@200                      2011-001     1.0000

R@500                      2011-001     1.0000

iP@0.00                    2011-001     0.8889

iP@0.10                    2011-001     0.8889

iP@0.20                    2011-001     0.7568

iP@0.30                    2011-001     0.7568

iP@0.40                    2011-001     0.7568
```

```
iP@0.50                   2011-001      0.7568
iP@0.60                   2011-001      0.7568
iP@0.70                   2011-001      0.7568
iP@0.80                   2011-001      0.7568
iP@0.90                   2011-001      0.7412
iP@1.00                   2011-001      0.6087
RP                        2011-001      0.7429
RR                        2011-001      0.5000
AP                        2011-001      0.7236
nDCG@5                    2011-001      0.6608
nDCG@10                   2011-001      0.7163
nDCG@15                   2011-001      0.6888
nDCG@20                   2011-001      0.6740
nDCG@30                   2011-001      0.5977
nDCG@50                   2011-001      0.6649
nDCG@100                  2011-001      0.8371
nDCG@200                  2011-001      0.8800
nDCG@500                  2011-001      0.8800
```

Once implemented `HtmlProcessor`, you should compare the effectiveness of the search engine with the `HtmlProcessor` vs `SimpleProcessor` - compare the size of the vocabulary, indexes, and the effectiveness measures.

TREC format

The output file `.run` has the following format:

```
2011-001  Q0  2011-72-101  1    0.2041027796949601    sys
2011-001  Q0  2011-69-107  2    0.20122488707165842   sys
...
2011-023  Q0  2011-25-036  99   0.029122264017657634  sys
2011-023  Q0  2011-62-113  100  0.02901768957593081   sys
```

The columns are: ID of the topic, the identifier `Q0` (never changes), the ID of the document, the position of the document, the similarity with the query, the ID of the system (for us, it will be always **sys**).

The file .qrel containing relevance judgements has the following format:

```
2011-001        0        2011-51-039      2
2011-001        0        2011-64-042      1
...
2011-023        0        2011-88-050      0
2011-023        0        2011-82-066      2
```

The columns are: ID of the topic, the identifier `Q0` (never changes), the ID of the document, the relevance judgement. In our scenario, the relevance is gradual with levels 0, 1 y 2.

Delivery

This lab will be submitted on March 13th, where the students should have 15 minutes to demonstrate the correct behaviour lab 2.