Implementing a Search Engine – Part 1

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, during the first sessions, you can use the class `SimpleProcessor` but it will be later replaced by `HtmlProcessor` to process HTML pages (Part 2 of the lab). The interface `RetrievalModel` defines basic operations for a retrieval model. For P1 you should implement the class `Cosine`.

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.

Part 1

For the P1 we provide the already created index (`2011-index`) for a corpus of documents (`2011-documentos`). You should implement the vector model and the cosine function following the next steps:

1. Load the index from disc.
2. Load the queries from disc.
3. For each query:
    a. Get its terms.
    b. Calculate the similarity with the documents.
    c. Generate the output with the first 500 documents.
4. Evaluate the effectivity of the search engine.

Implementation

You will basically need to code into the class `Cosine`, that uses the information from `Index`. For each query you will run the method:

```
ArrayList<Tuple<Integer, Double>> runQuery(String queryText,
                                           Index index,
                                           DocumentProcessor docProcessor)
```

This method returns tuples (docID, similarity).

Internally, first it extracts the terms of the query, and then call the method:

```
ArrayList<Tuple<Integer, Double>> computeVector(ArrayList<String> terms, Index index)
```

This method receives a list of terms and returns the query vector, represented as tuples (termID, weight).

Then, it calls the method:

```
ArrayList<Tuple<Integer, Double>> computeScores(ArrayList<Tuple<Integer, Double>> queryVector,
                                                Index index)
```

That returns the similarity between the indexed documents and the query vector.

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:

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

The terms weights in the documents are already calculated in the index (`invertedIndex`), and also their norms (in `documents`). The query weights and its norm should be calculated at query time.

Suggestions and Requirements

- The cosine divisor should only consider the terms that appear in the query. The contribution of the other terms is 0.
- The IDF of one term appears once in the document weight and once in the query weight.
- You cannot use the direct index for P1.
- With `invertedIndex` you can immediately know which documents contain the terms of the queries.
- The use of external libraries is forbidden.
- Check for comments `// P1` to see which parts of the template you should fill.

Evaluation

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:

```
> 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
```

We provide a file `2011-CorrectOutput.run` as example of the output of the system, to check if your code is correct.

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
```

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 with the next ones on March 15th, where the students should have 15 minutes to demonstrate the right behaviour of the labs.