Programming for Data Science – Assignment 1/3
A Boolean Information Retrieval System

MSc in Data Science

Handed out: 20 October 2017
Due by: 10 November 2017

The Exercise

In this assignment you will implement a basic boolean text retrieval system. This is the simplest of search engines. The document collection you will use is Cran, located under `~/Datasets/cran/cran.all.1400`, in your VM. The boolean model allows users to search for text documents based on boolean queries, e.g. `(velocity AND vorticity) OR (velocity AND boundary-layer AND approximation)`.

Given a boolean query provided in the standard input, your system should return the IDs of all documents that satisfy the query in the standard output as a space-separated list of document IDs.

In order to be able to respond to boolean queries, your system must:

1. Parse and clean-up the document collection, similar to the warm-up exercise.
2. Build an (inverted) index to allow for query evaluation, **taking into account both the terms appearing in the body as well as in the titles of documents**. Your inverted index should also include term and document frequencies, in the form {`term: [df, {doc_id: tf}]`}. `df` is the number of documents `term` appears in, while `tf` is the number of times `term` appears in document with `doc_id`.
3. Parse, pre-process (exactly as the documents) and evaluate queries.

Assumptions

1. All indexes and supplementary structures are in text form, i.e. for this assignment there is no need to use DBs, external technologies, binary files, etc.
2. Feel free to use loops, comprehensions, lambdas, etc., as you see fit.
3. Make sure you use the functions as provided and that your solutions respect the data types specified

More implementation hints are provided in the source code template.

Deliverable

A single Python program, named `boolret.py`, with a main function, so as to be executable from a terminal, with the following I/O specification.

Input: A boolean query given on the standard input. Each line should represent logical conjunction (`AND`) between terms, while multiple lines should represent logical disjunction (`OR`) between line expressions. E.g., the input:

```
alpha beta
gamma delta epsilon
```

should represent the query `(alpha AND beta) OR (gamma AND delta AND epsilon)`.

Output: A list of space-separated document IDs (integer numbers) satisfying the query, ordered by document ID. If you implement ranked retrieval (see below) rank the document ID in inverse order of score.

Sample run:

```
$ echo -e "regional rotational flow \n plane centrifugal force" | ./boolret.py
 2 110 317 660 984 1165 1248
```

Background

Query Evaluation

Suppose you have built the following index (leaving out document and term frequencies for the sake of the argument – you need to include them in your inverted index implementation).

```
term1 -> docId1, docId2, docId3
term2 -> docId1, docId3, docId4, docId5
term3 -> docId3, docId5, docId6
```

The query `term1 AND term2` should return the IDs of all documents containing **both** `term1` and `term2`, i.e. it should return: `docId1 docId3`.

The query `(term1 AND term2) OR term3` should return **the union of** the IDs of all documents containing both `term1` and `term2` **and** the IDs of the documents containing `term3`, i.e. it should return `docId1 docId3 docId5 docId6`.

The lists, in the wider sense, of document IDs are also referred to as **postings lists** (they are typically sets). If the postings list of term t_i is P_i and $\vee$ and $\wedge$ represent the `OR` and `AND` operators respectively, then a query of the form $\bigvee(\bigwedge_i t_i)$ is evaluated as: $\bigcup(\bigcap_i P_i)$.

The Cranfield Collection

The Cranfield experimental collection is already downloaded in your VM at `$HOME/Datasets/cran/`. If you need to, you can download the collection at `http://ir.dcs.gla.ac.uk/resources/test_collections/cran/` and extract it somewhere in your home folder. For this exercise you will only need to use the text collection file `cran.all.1400` – ignore the rest. This file contains 1400 documents and it is structured as follows:

```
.I 21
.T
on heat transfer in slip flow .
.A
stephen h. maslen
.B
lewis flight propulsion laboratory, naca, cleveland, ohio
.W
on heat transfer in slip flow .
a number of authors have considered the effect of slip on the heat
transfer and skin friction in a laminar boundary layer over a flat plate .
reference 1 considers this by a perturbation on the usual laminar
boundary-layer analysis while some other studies.dash e.g., reference
the impulsive motion of an infinite plate .
```

Lines beginning with a `.` are used to delimit sections of interest. For each document in the file, `.I` is followed by the document ID, `.T` is followed by the title, while `.W` is followed by the text body. The body of each document is delimited by the next line starting with `.I`. You do not need to be concerned with the other delimiters.

In this exercise you first need to extract document IDs, titles and text bodies and keep associations between them.

The Porter Stemmer

The Porter stemmer is one of the most widely used stemmers for the English language. An efficient stemmer is included in the `nltk` module and you can use it as follows:

```
>>> from nltk.stem.porter import PorterStemmer
>>> stemmer = PorterStemmer()
>>> stemmer.stem('initialize')
u'initi'
```

Stop-words

You can obtain a set of stop-words in Python also via the `nltk` module as follows:

```
>>> from nltk.corpus import stopwords
>>> stop = set(stopwords.words('english'))
```

Extension: Ranking based on the tf-idf weighting scheme

One characteristic of boolean retrieval is that documents either satisfy a query or they don't. This leads to an inherent lack of ranking of documents (which is why you are asked to rank the documents by ID). Ideally, we would like to have a measure of how relevant a document is to a query in order to list more relevant documents higher in the results list.

One term-weighting scheme which can help us rank documents (including in boolean retrieval, even though this is not standard practice) is *tf-idf*, which stands for "term frequency.inverse document frequency".

This scheme assumes that the more times a term appears in a document, the more relevant that document is to the term. Term frequency $f_{t,d}$ reflects the number of times a term t occurs in a document d. Because we want to avoid introducing bias towards larger documents we define $\text{tf}_{t,d} = 1 + \log(f_{t,d})$.

Document frequency indicates the resolving power of a term in a document collection. The more documents a term appears in, the less important it is in distinguishing relevant from non-relevant documents. We typically scale document frequency by the total number of documents in the document collection. The inverse document frequency of term t can be defined as $\text{idf}_t = \log \frac{N}{d_t}$, where N is the total number of documents in the collection, and d_t is the number of documents t appears in.

Based on the above, tf-idf is defined as $\text{tf-idf}_{t,d} = \text{tf}_{t,d} \times \text{idf}_t$.

To implement tf-idf-based ranking you first have to calculate it for all documents satisfying a conjunction (AND) by summing the td-idf scores for each term of a query:

$$\text{tf-idf-conj}_{q,d} = \sum_{t \in q} \text{tf-idf}_{t,d}$$

where q is a conjunctive query consisting of terms t. This assigns a tf-idf score to each resulting document.

After evaluating the AND parts of a query, and when different disjunction factors return the same document, keep the maximum tf-idf score obtained.

$$\text{tf-idf}_{q,d} = \max_d(\text{tf-idf-conj}_{q,d})$$

More information on tf-idf can be found at `https://en.wikipedia.org/wiki/Tfidf`.

As an extension, implement the above weighting approach. You will need to make use of the document and term frequencies of your inverted index, and adapt your retrieval implementation. Since now you will have a score associated with each resulting document ID, rank your results by inverse score – i.e. higher scoring documents should come first.