Information Retrieval

SS2014

Exercise 1 - Group 19

Our task was to build a basic ranking engine for a specific set of text documents that were given to us. More specifically, these documents are a collection of answers from 20 different newsgroups subsets.

We implemented our application as a command line interface (CLI) in the Java programming language. We used for it a piped parser with multiples pipe stages, where each stage is implemented as a thread and managed by a thread pool in order to take advantage of the architecture of contemporary multi-CPU computers. This enables us to read the documents and simultaneously create the inverted index. The basic steps of our application are described below.

Each document contains much more information (header, possible previous quoted question or answer, additional user information such as email) than it is actually needed for our search. Our first concern was to remove all these and hold only the actual reply to the newsletter topic. The next step was to use a regular expression to remove all non-word characters and all numerical values, and when the removal was over to turn the remaining words to lower case. We wanted then to remove the so called stop words from our text. The stop words are words that occur very often in the English language and wouldn’t help us in our search. We used a set of 185 stop words found at the following site <http://snowball.tartarus.org/algorithms/english/stop.txt>. The next step was to use a stemmer to further reduce our set of words. Stemming reduces similar words to their basic root form. Our stemmer was again taken from the same site above (<http://snowball.tartarus.org/dist/libstemmer_java.tgz>) and used as an external jar file in our code. All the above techniques (case folding, stemming, and removal of stop words) are implemented as pipes in our code and can be switched on/off through specific commands on the CLI. The resulting unique (meaning each word appears only once) set of words is our vocabulary.

For the inverted index we implemented two algorithms: the bag-of-words, which is also the default, and the bi-gram index. The first one simply counts the occurrence of each (unique) word in the complete vocabulary, and the second uses all the adjacent words as pairs and counts the occurrence of these pairs in the whole vocabulary. Both versions are implemented also as pipes and can be switched on/off again as a command on the CLI. In order to adequately identify each document, we used a unique ID, which comprises of the parent folder name of the document (each name has a corresponding number) and the name of the document itself. We used for the storage of the indices in memory Java’s TreeMap. This data structure is a Red-black Tree, which is a balanced binary tree and allows us to search pretty fast for existing terms. The inverted index is also stored in a single block on the hard disk, in order to be used at will and not having always to be created on the fly.

We were also given 20 test queries for testing purposes. These documents have the same form of our initial documents, and thus are read and processed the exact same way. Our scoring is based on the following formula:

Score =,

Where tf is the term frequency and idf is the inverse document frequency. The result for each search is given finally in the following form:

*topic Q0 document-id rank score run-name*

Here is a list with all available CLI commands (and their functionality):

!info - displays application information

!setPath - sets documents folder path

!bigram - uses bi-gram index

!stemmer - switches stemmer use on/off

!stopwords - switches stopwords removal on/off

!buildVoc - builds Vocabulary

!test - tests topics

!search <topic#> - searches for the specified topic and creates a scorelist

This exercise has shown us once more that the devil lies in the details, wheter in the proper selection and modification of probable terms during the index creation, or during querying and score computation. However dispite certain difficulties with runtime optimization and architectural issues, we are certain, that our implementation has offered some insights to us.