Assignment 1: Indexing

Nicolai Dahl Blicher-Petersen and Daniel Jonathan Smith
{s3441163, s3361789}@student.rmit.edu.au

RMIT

1 Ranked Retrieval

Building on the inverted index created in assignment one, we have implemented the BM25 similarity function to efficiently rank documents with respect to an input query. The SimpleQueryEngine implemented in assignment one has been extended in a way that allows us to reuse loading of the inverted index and the document map. Furthermore, looking up the list of postings for a term is available through the getResult method [2]. The class that carries out the ranked retrieval is called BM25RankedQueryEngine.

1.1 Processing a Query

We also reuse the parser and stopper modules designed in the first assignment when first processing an input query. This way we ensure that the query terms are treated the same way as when building the index [2].

1.2 Data Structures

For the ranking algorithm two essential data structures are used. First of all we use a java HashMap for our accumulators with the document ID as key and the accumulated sum as value. This allows for constant-time read and put operations when updating the accumulator values [3].

The second data structure we take advantage of is the Min-heap [5]. It is used in the final stage of the ranking procedure to retrieve the n accumulators with the highest similarity scores. The Min-heap is specified to only allow n elements in it, and every time a new accumulator is added it checks if the limit has been reached. If not, the accumulator's value is checked against that of the heap's root element and the accumulator is only added to the heap if it is of greater value than the root element. This is a constant-time operation, as the heapify routine, performed on insertion, is bound by the hight of the heap, which is constant. In total, the asymptotic complexity of finding the n highest-valued elements is $O(n)$.

explain the getResults method with reference to the algorithm/ranking function. Mention accumulators but don't mention MinHeap directly. It is explained below

2 Advanced IR Feature

As our advanced information retrieval feature we selected automatic query expansion, also known as pseudo-relevance feedback. Queries might be hard to specify for users and there is a danger that the user searches for a synonym to a word that is more significant in the document collection [1]. The idea is to expand the initial query with E additional terms that are statistically related to the query to mitigate this vocabulary mismatch [1]. The steps to automatic query expansion are [4]:

1. Perform ranked retrieval on the initial query with a good similarity measure and assume that the top R ranked documents are relevant.
2. Parse through these R documents and mark all terms in these candidate terms for query expansion.
3. Select the best E of these candidate terms for the query expansion by evaluating them with some statistical method.
4. Append the E terms to the initial query and run the ranked retrieval procedure again. This is the final result.

As the statistical method in step three, we use the Okapi Term Selection Value (TSV) approach to select a set of E terms to extend the initial query with [1].

2.1 Implementation

We have extended the BM25RankedQueryEngine presented in section 1 to handle the query expansion. The class is called QueryExpansionBM25QueryEngine and is automatically used as query engine if the -QEBM25 input flag is specified. The getResults method of the class follows the approach described above with the most difficult step being step 2, as the current invertedIndex does not allow us to retrieve all terms for a particular document.

To get this part working we have to do some extra work at indexing time. We save an uninverted index of documents, where document IDs in the term lexicon (termLex) point to a term list in the termIndex. We also save a term map that maps a term ID to a specific term, called termMap. Keeping track of this extra termMap allows us to reuse our indexing and compression code from assignment one, as we are once again only saving numerical data [2].

So when the list of candidate terms is requested, each document's term list is found by first looking up the byte offset (into the termIndex) of the list of terms, which is listed in the termLex (lexicon). The findCandidateTerms method performs this procedure for each of the documents and uses a HashMap [3] from term IDs to frequencies, to make sure that terms occurring in multiple documents are only saved as candidate terms once. The frequencies mentioned are not the within-document frequencies but the "number of documents in the initially retrieved pool that contain the term" [4].

Bibliography

[1] Bodo Billerbeck and Justin Zobel. Questioning query expansion: an examination of behaviour and parameters. In *ADC '04: Proceedings of the 15th Australasian database conference*, pages 69–76, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[2] Nicolai Dahl and Daniel Smith. Assignment 1: Indexing. Technical report, RMIT, September 2013.

[3] Oracle. Class hashmap<e>. `http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html`, 2011.

[4] Falk Scholer. Query expansion and relevance feedback, slideshow, 2013.

[5] Eric Weisstein. Heap. `http://mathworld.wolfram.com/Heap.html`, 2013.