Assignment 2: Retrieval Algorithm and Evaluation

Z534: Information Retrieval: Theory and Practice Fall 2014

Task 1: Implement your first search algorithm

Based on the Lucene index, we can start to design and implement efficient retrieval algorithms. Let's start from the easy ones. Please implement the following ranking function using the Lucene index we provided through Oncourse (*index.zip*):

$$F(q, doc) = \sum_{q_i \in q} \frac{c(q_i, doc)}{length(doc)} \cdot \log\left(1 + \frac{N}{k(q_i)}\right)$$

where q is the user query, doc is the target (candidate document in AP89), q_i is the query terms, $c(q_i, doc)$ is the count of term q_i in document doc, N is total number of documents in AP89, and $k(q_i)$ is the total number of documents have the term q_i. Please use Lucene API to get the information. From retrieval viewpoint, $\frac{c(q_i, doc)}{length(doc)}$ is called normalized TF, while $\log(1 + \frac{N}{k(q_i)})$ is IDF (inverse document frequency).

The following code (using Lucene API) can be useful to help you implement the ranking function:

```
        // Get the preprocessed query terms
        Analyzer analyzer = new StandardAnalyzer();
        QueryParser parser = new QueryParser("TEXT", analyzer);
        Query query = parser.parse(queryString);
        Set<Term> queryTerms = new LinkedHashSet<Term>();
        query.extractTerms(queryTerms);
        for (Term t : queryTerms) {
                System.out.println(t.text());
        }

        IndexReader reader = DirectoryReader
                              .open(FSDirectory
                                             .open(new File(pathToIndex)));

        //Use DefaultSimilarity.decodeNormValue(…) to decode normalized document length
        DefaultSimilarity dSimi=new DefaultSimilarity();

        //Get the segments of the index
        List<AtomicReaderContext> leafContexts = reader.getContext().reader()
                              .leaves();
        for (int i = 0; i < leafContexts.size(); i++) {
                AtomicReaderContext leafContext=leafContexts.get(i);
                int startDocNo=leafContext.docBase;
                int numberOfDoc=leafContext.reader().maxDoc();
                for (int docId = startDocNo; docId < startDocNo+numberOfDoc; docId++) {
                        //Get normalized length for each document
                        float normDocLeng=dSimi.decodeNormValue(leafContext.reader()
                                             .getNormValues("TEXT").get(docId-
startDocNo));
                        System.out.println("Normalized length for doc("+docId+") is
"+normDocLeng);
```

```
                }

                //Get the term frequency of "new" within each document containing it for
<field>TEXT</field>
                DocsEnum de = MultiFields.getTermDocsEnum(leafContext.reader(),
                                        MultiFields.getLiveDocs(leafContext.reader()),
"TEXT", new BytesRef("new"));
                int doc;
                while ((doc = de.nextDoc()) != DocsEnum.NO_MORE_DOCS) {
                        System.out.println("\"new\" occurs "+de.freq() + " times in doc(" +
(de.docID()+startDocNo)+") for the field TEXT");
                }
        }
`
```

For each given query, your code should be able to 1. Parse the query using Standard Analyzer (Important: we need to use the SAME Analyzer that we used for indexing to parse the query), 2. Calculate the relevance score for each query term, and 3. Calculate the relevance score $F(q, doc)$.

<u>The code for this task should be saved in a java class: easySearch.java</u>

Task 2: Test your search function with TREC topics

Next, we will need to test the search performance with the TREC standardized topic collections. You can download the query test topics from Oncourse (*topics.51-100*).

In this collection, TREC provides a number of topics (total 50 topics), which can be employed as the candidate queries for search tasks. For example, one TREC topic is:

<top>
<head> Tipster Topic Description
<num> Number: 054
<dom> Domain: International Economics
<title> Topic: Satellite Launch Contracts
<desc> Description:
Document will cite the signing of a contract or preliminary agreement, or the making of a tentative reservation, to launch a commercial satellite.
<smry> Summary:
Document will cite the signing of a contract or preliminary agreement, or the making of a tentative reservation, to launch a commercial satellite.
<narr> Narrative:
A relevant document will mention the signing of a contract or preliminary agreement , or the making of a tentative reservation, to launch a commercial satellite.
<con> Concept(s):
1. contract, agreement
2. launch vehicle, rocket, payload, satellite
3. launch services, commercial space industry, commercial launch industry
4. Arianespace, Martin Marietta, General Dynamics, McDonnell Douglas
5. Titan, Delta II, Atlas, Ariane, Proton
<fac> Factor(s):
<def> Definition(s):
</top>

In this task, you will need to use two different fields as queries: <title> and <desc>. The former query is very short, while the latter one is much longer.

Your software must output up to top 1000 search results to a result file in a format that enables the trec_eval program to produce evaluation reports. trec_eval expects its input to be in the format described below.

```
QueryID    Q0    DocID Rank   Score   RunID
For example:
10     Q0     DOC-NO1    1     0.23    run-1
10     Q0     DOC-NO2    2     0.53    run-1
10     Q0     DOC-NO3    3     0.15    run-1
:      :       :     :       :       :
11     Q0     DOC-NOk    1     0.042   run-1
```

The code for this task should be saved in a java class: searchTRECtopics.java

Task 3: Test Other Search Algorithms

Next, we will test a number of other retrieval and ranking algorithms by using Lucnen API and the index provided through Oncourse (*index.zip*).

For instance, you can use the following code to search the target corpus via BM25 algorithm.

```
IndexReader reader = DirectoryReader
                              .open(FSDirectory
                                             .open(new File(pathToIndex)));
IndexSearcher searcher = new IndexSearcher(reader);
searcher.setSimilarity(new BM25Similarity()); //You need to explicitly specify the
ranking algorithm using the respective Similarity class
Analyzer analyzer = new StandardAnalyzer(); QueryParser
parser = new QueryParser("TEXT", analyzer);

Query query = parser.parse(queryString);
TopScoreDocCollector collector = TopScoreDocCollector.create(1000, true);
searcher.search(query, collector);

ScoreDoc[] docs = collector.topDocs().scoreDocs;
for (int i = 0; i < docs.length; i++) {
        Document doc = searcher.doc(docs[i].doc);
        System.out.println(doc.get("DOCNO")+" "+docs[i].score);
}

reader.close();
```

In this task, you will test the following algorithms

1. Vector Space Model (org.apache.lucene.search.similarities.DefaultSimilarity)
2. BM25 (org.apache.lucene.search.similarities.BM25Similarity)

3. Language Model with Dirichlet Smoothing (org.apache.lucene.search.similarities.LMDirichletSimilarity)
4. Language Model with Jelinek Mercer Smoothing (org.apache.lucene.search.similarities.LMJelinekMercerSimilarity, set λ to 0.7)

You will need to compare the performance of those algorithms (and your algorithm implemented in Task 1) with the TREC topics. For each topic, you will try two types of queries: short query (<title> field), and long query (<desc> field). So, for each search method, you will need to generate two separate result files, i.e., for **BM25**, you will need to generate **BM25longQuery.txt** and **BM25shortQuery.txt**

The code for this task should be saved in a java class: compareAlgorithms.java

Task 4: Algorithm Evaluation

In this task, you will need to compare different retrieval algorithms via various evaluation metrics, i.e., precision, recall, and MAP.

Please read this document about trec_eval:
http://faculty.washington.edu/levow/courses/ling573_SPR2011/hw/trec_eval_desc.htm
And, you can download the trec_eval program from
http://trec.nist.gov/trec_eval/trec_eval_latest.tar.gz
We will use this code to evaluate the search result performance.

TrecEval can be used via the command line in the following way:
trec_eval groundtruth.qrel results (the first parameter is the ground truth file or to say judgment file, and the second parameter is the result file you just generated from the last task.
trec_eval --help should give you some ideas to choose the right parameters

You can download the ground truth file from Oncourse (*qrels.51-100*).

Please compare the different search algorithms (files generated in task2 and task 3) and finish the following table:

Short query

Evaluation metric	Your algorithm	Vector Space Model	BM25	Language Model with Dirichlet Smoothing	Language Model with Jelinek Mercer Smoothing
map	0.1273	0.2028	0.2063	0.2120	0.1994
R-prec	0.1559	0.2241	0.2167	0.2242	0.2100
bpref	0.2594	0.2945	0.2971	0.3036	0.2944
recip_rank	0.3960	0.4649	0.4848	0.4903	0.4708
P5	0.2667	0.2958	0.3083	0.3500	0.2875
P10	0.2187	0.2958	0.2979	0.3250	0.2771

P15	0.2139	0.2750	0.2778	0.3056	0.2514
P20	0.2083	0.2604	0.2698	0.2906	0.2417
P30	0.1972	0.2521	0.2507	0.2694	0.2389
P100	0.1315	0.1675	0.1706	0.1725	0.1644
P200	0.0936	0.1162	0.1177	0.1173	0.1094
P500	0.0554	0.0630	0.0645	0.0643	0.0625
P1000	0.0330	0.0367	0.0370	0.0379	0.0364

Long query

Evaluation metric	Your algorithm	Vector Space Model	BM25	Language Model with Dirichlet Smoothing	Language Model with Jelinek Mercer Smoothing
map	0.0719	0.1595	0.1738	0.1640	0.1573
R-prec	0.1113	0.1769	0.1948	0.1944	0.1860
bpref	0.2323	0.2787	0.2934	0.2858	0.2812
recip_rank	0.3070	0.4552	0.4462	0.3407	0.3595
P5	0.1583	0.2667	0.2875	0.2542	0.2375
P10	0.1521	0.2521	0.2521	0.2417	0.2167
P15	0.1306	0.2306	0.2458	0.2458	0.2194
P20	0.1229	0.2240	0.2385	0.2344	0.2167
P30	0.1250	0.2083	0.2285	0.2194	0.2104
P100	0.0948	0.1454	0.1533	0.1492	0.1419
P200	0.0682	0.0991	0.1062	0.1044	0.0956
P500	0.0420	0.0545	0.0573	0.0566	0.0536
P1000	0.0273	0.0319	0.0336	0.0332	0.0320

Please summarize your findings of this task:

As seen above, for short queries the best performance is given by Language Model with Dirichlet Smoothing, followed by BM25, Vector Space Model, Language Model with Jelinek Mercer Smoothing and then MyAlgo. For Long queries, the best performance is given by BM25, followed by Language Model with Dirichlet Smoothing, Vector Space Model, Language Model with Jelinek Mercer Smoothing and then MyAlgo.

For Long queries, the performance drops for all algorithms as compared to short queries.

The way MyAlgo behaves is that it just does the summation of the Normalized Term Frequency * Inverse Document Frequency. So, for a query string if the NTF can be very high and then the document in which that query string appears is also ranked high even though the other query terms do not have the same NTF as that particular query term. Hence the performance of MyAlgo is not as good as other algorithms.

Submission: Please submit the java codes and results via Oncourse system.