Assignment 2: Retrieval Algorithm and Evaluation

Z534: Search, Fall 2016

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 Canvas (*index.zip*):

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

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

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

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 Canvas (*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 Lucene API and the index provided through Canvas (*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 -m all_trec 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 Canvas (*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
P@5	0.1560	0.2920	0.3000	0.3440	0.2800
P@10	0.1580	0.2980	0.2960	0.3240	0.2780
P@20	0.1500	0.2580	0.2670	0.2860	0.2420
P@100	0.1022	0.1636	0.1668	0.1682	0.1594
Recall@5	0.0308	0.0529	0.0468	0.0610	0.0514
Recall@10	0.0578	0.0940	0.0859	0.0985	0.0901

Recall@20	0.0852	0.1396	0.1338	0.1437	0.1292
Recall@100	0.2408	0.3518	0.3522	0.3428	0.3280
MAP	0.0849	0.1943	0.1971	0.2039	0.1908
MRR	0.2944	0.4596	0.4672	0.4785	0.4637
NDCG@5	0.1674	0.3043	0.3149	0.3486	0.2994
NDCG@10	0.1708	0.3121	0.3136	0.3400	0.2980
NDCG@20	0.1741	0.3003	0.3068	0.3299	0.2867
NDCG@100	0.2009	0.3153	0.3199	0.3280	0.3085

Long query

Evaluation metric	Your algorithm	Vector Space Model	BM25	Language Model with Dirichlet Smoothing	Language Model with Jelinek Mercer Smoothing
P@5	0.1240	0.2240	0.2440	0.2480	0.2000
P@10	0.1140	0.2160	0.2200	0.2240	0.1920
P@20	0.1100	0.1890	0.2147	0.2050	0.1800
P@100	0.0786	0.1278	0.1374	0.1384	0.1226
Recall@5	0.0147	0.0204	0.0298	0.0367	0.0294
Recall@10	0.0284	0.0447	0.0539	0.0563	0.0528
Recall@20	0.0499	0.0769	0.0908	0.0991	0.0858
Recall@100	0.1723	0.2554	0.2848	0.3038	0.2434
MAP	0.0662	0.1245	0.1387	0.1341	0.1215
MRR	0.2218	0.3579	0.3651	0.2896	0.3124
NDCG@5	0.1294	0.2388	0.2514	0.0367	0.2009
NDCG@10	0.1203	0.2288	0.2336	0.0563	0.1996
NDCG@20	0.1205	0.2112	0.2271	0.0991	0.1992
NDCG@100	0.1420	0.2269	0.2450	0.3038	0.2166

Please summarize your findings of this task:

It is clear to see that for precision the Vector Space Model performs better than the TF-IDF algorithm. BM25 algorithm performs better than the Vector Space Model. The Language Model with Dirichlet Smoothing performs better than the VSM Model. The Language Model with Jelinek Mercer Smoothing only performs better than the TF-IDF algorithm. For recall, on average VSM algorithm performs better than the TF-IDF, BM25 and Language Model with Jelinek Mercer Smoothing. The Language Model with Dirichlet Smoothing performs better than the VSM Model. On average it is clear to see that the Language Model with Dirichlet Smoothing and BM25 algorithms perform better than all other algorithms. On average the worst- prforming algorithm is the TF-IDF algorithm.

Submission: Please submit the java codes and results via Canvas.