**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*):

��(��! , 𝑑𝑜��) 𝑁

!

𝐹 ��, 𝑑𝑜𝑐 =

!!∈!

𝑙𝑒𝑛𝑔𝑡ℎ(𝑑𝑜��) ∙ log  (1 + 𝑘 𝑞 )

, where q is the user query, 𝑑𝑜𝑐 is the target (candidate document in AP89), ��! is the query term, ��(��! , 𝑑𝑜��) is the count of term ��! in document 𝑑𝑜��, N is total number of documents in AP89, and 𝑘 ��! is the total number of documents have the term ��! . Please

!(!! ,!"#)

use Lucene API to get the information. From retrieval viewpoint,

!"#$%!(!"#)

is called

normalized TF, while log  (1 + ! ) 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)); "+normDocLeng);

System.out.println("Normalized length for doc("+docId+") is

}

//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 ��, 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 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 | 0.1226 | 0.1969 | 0.2004 | 0.2059 | 0.1935 |
| R-prec | 0.1519 | 0.2186 | 0.2129 | 0.2198 | 0.2057 |
| bpref | 0.2532 | 0.2913 | 0.2950 | 0.3003 | 0.2915 |
| recip\_rank | 0.3803 | 0.4696 | 0.4772 | 0.4785 | 0.4637 |
| P5 | 0.2520 | 0.2960 | 0.3080 | 0.3480 | 0.2880 |
| P10 | 0.2100 | 0.3020 | 0.3000 | 0.3260 | 0.2820 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| P15 | 0.2093 | 0.2773 | 0.2787 | 0.3053 | 0.2547 |
| P20 | 0.1990 | 0.2600 | 0.2700 | 0.2890 | 0.2440 |
| P30 | 0.1913 | 0.2493 | 0.2487 | 0.2667 | 0.2373 |
| P100 | 0.1272 | 0.1638 | 0.1678 | 0.1692 | 0.1606 |
| P200 | 0.0907 | 0.1143 | 0.1163 | 0.1159 | 0.1707 |
| P500 | 0.0546 | 0.0626 | 0.0642 | 0.0640 | 0.0622 |
| P1000 | 0.0330 | 0.0367 | 0.0371 | 0.0381 | 0.0364 |

Long query

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Evaluation  metric | Your  algorithm | Vector  Space Model | BM25 | Language  Model with  Dirichlet  Smoothing | Language  Model with  Jelinek Mercer Smoothing |
| map | 0.0684 | 0.1538 | 0.1676 | 0.1588 | 0.1518 |
| R-prec | 0.1066 | 0.1726 | 0.1898 | 0.1897 | 0.1813 |
| bpref | 0.2253 | 0.2712 | 0.2853 | 0.2791 | 0.2746 |
| recip\_rank | 0.2999 | 0.4490 | 0.4497 | 0.3483 | 0.3666 |
| P5 | 0.1560 | 0.2640 | 0.2840 | 0.2560 | 0.2360 |
| P10 | 0.1480 | 0.2480 | 0.2460 | 0.2420 | 0.2140 |
| P15 | 0.1253 | 0.2280 | 0.2413 | 0.2440 | 0.2173 |
| P20 | 0.1200 | 0.2210 | 0.2340 | 0.2340 | 0.2130 |
| P30 | 0.1220 | 0.2040 | 0.2233 | 0.2173 | 0.2060 |
| P100 | 0.0906 | 0.1410 | 0.1488 | 0.1460 | 0.1382 |
| P200 | 0.0648 | 0.0960 | 0.1032 | 0.1022 | 0.0933 |
| P500 | 0.0402 | 0.0534 | 0.0560 | 0.0577 | 0.0526 |
| P1000 | 0.0266 | 0.0315 | 0.0331 | 0.0330 | 0.0316 |

Please summarize your findings of this task:

Precision value decreases for long queries and small number of retrieved documents compared to short

Queries and small number of retrieved documents for my search algorithm. Also, the precision value is almost same for short and long queries for large number of retrieved documents. The precision value for

my search Algorithm is low compared to all other search algorithms..

For short queries Language model with Dirichlet smoothing is the best and my search algorithm is the

Worst. For long queries BM25 is the best and my search algorithm is the worst.

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