Assignment 1: Indexation

Z534: Search, Fall 2016 (deadline: September 28th)

Indexation is important for information retrieval and search engine. Based on Wikipedia definition, search engine indexing collects, parses, and stores (text) data to facilitate fast and accurate information retrieval. Index design incorporates interdisciplinary concepts from linguistics, cognitive psychology, mathematics, informatics, physics, and computer science. In this assignment, you will need to create index by using Lucene API, while using the index to solve a number of problems.

Before we start, please download and compile Lucene API (in Java). http://lucene.apache.org/ Note that, you also need to download both lucene-6.2.0-src.tgz (for source code) and lucene-6.2.0.tgz (compiled API). **Then, you need to read the tutorial carefully before starting this assignment. http://alias-i.com/lingpipe-book/lucene-3-tutorial-0.5.pdf** (This is for the old version, but it is a good one) A more comprehensive tutorial is available at: http://wiki.apache.org/lucene-java/FrontPage?action=show&redirect=FrontPageEN

Task 1: Generating Lucene Index for Experiment Corpus (AP89)

You can download the AP89 experiment corpus via Canvas: The document in the corpus is stored in the following format:

```
<DOC>
<DOCNO> AP890101-0001 </DOCNO>
<FILEID>AP-NR-01-01-89 2358EST</FILEID>
<FIRST>r a PM-APArts:60sMovies     01-01 1073</FIRST>
<SECOND>PM-AP Arts: 60s Movies,1100</SECOND>
<HEAD>You Don't Need a Weatherman To Know '60s Films Are Here</HEAD>
<HEAD>Eds: Also in Monday AMs report.</HEAD>
<BYLINE>By HILLEL ITALIE</BYLINE>
<BYLINE>Associated Press Writer</BYLINE>
<DATELINE>NEW YORK (AP) </DATELINE>
<TEXT>
   The celluloid torch has been passed to a new
generation: filmmakers who grew up in the 1960s.
   ``Platoon,'' ``Running on Empty,'' ``1969'' and ``Mississippi
Burning'' are among the movies released in the past two years from
writers and directors who brought their own experiences of that
turbulent decade to the screen.......
</TEXT>
</DOC>
```

As the first task, you will need to create index by using Lucene. From Lucene viewpoint, each document is a collection of pre-defined fields, where a field supplies a field name and value. By using Lucene API (in Java), we can easily generate corpus index (inverted index).

For this task, you will need to generate a Lucene index with the following fields: 1. DOCNO, 2. HEAD, 3. BYLINE, 4. DATELINE, and 5. TEXT. (Note: Merge content from multiple elements if they all bear the same tag. e.g. for the document above, you'll index “You Don't Need a Weatherman To Know '60s Films Are Here Eds: Also in Monday AMs report” for the <HEAD> field).

The following code can be useful to help you generate the index:

```
        Directory dir = FSDirectory.open(Paths.get(indexPath));
        Analyzer analyzer = new StandardAnalyzer();

        IndexWriterConfig iwc = new IndexWriterConfig(analyzer);

        iwc.setOpenMode(OpenMode.CREATE);

        IndexWriter writer = new IndexWriter(dir, iwc);

        for each trec doc, do {
                Document luceneDoc = new Document();
                luceneDoc.add(new StringField("DOCNO", DOCNO,
                                                  Field.Store.YES);
                luceneDoc.add(new TextField("TEXT", TEXT,
                                                  Field.Store.YES));
                writer.addDocument(luceneDoc);
        }

        writer.close();
```

The following examples will help you get some useful statistics about the index:

```
        IndexReader reader = DirectoryReader.open(FSDirectory.open(Paths.get(
        (pathToIndex)));

        //Print the total number of documents in the corpus
        System.out.println("Total number of documents in the corpus:
        "+reader.maxDoc());

        //Print the number of documents containing the term "new" in
<field>TEXT</field>.
        System.out.println("Number of documents containing the term \"new\" for
field \"TEXT\": "+reader.docFreq(new Term("TEXT", "new")));

        //Print the total number of occurrences of the term "new" across all
documents for <field>TEXT</field>.
        System.out.println("Number of occurrences of \"new\" in the field
\"TEXT\": "+reader.totalTermFreq(new Term("TEXT","new")));

        Terms vocabulary = MultiFields.getTerms(reader, "TEXT");

        //Print the size of the vocabulary for <field>TEXT</field>, applicable
when the index has only one segment.
        System.out.println("Size of the vocabulary for this field:
"+vocabulary.size());

        //Print the total number of documents that have at least one term for
```

```
<field>TEXT</field>
        System.out.println("Number of documents that have at least one term for
this field: "+vocabulary.getDocCount());

        //Print the total number of tokens for <field>TEXT</field>
        System.out.println("Number of tokens for this field:
"+vocabulary.getSumTotalTermFreq());

        //Print the total number of postings for <field>TEXT</field>
        System.out.println("Number of postings for this field:
"+vocabulary.getSumDocFreq());

        //Print the vocabulary for <field>TEXT</field>
        TermsEnum iterator = vocabulary.iterator();
        BytesRef byteRef = null;
        System.out.println("\n*******Vocabulary-Start**********");
        while((byteRef = iterator.next()) != null) {
            String term = byteRef.utf8ToString();
            System.out.print(term+"\t");
        }
        System.out.println("\n*******Vocabulary-End**********");
        reader.close();
```

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

Please answer the following questions:
1. How many documents are there in this corpus?
2. Why different fields are treated with different kinds of java class? i.e., StringField and TextField are used for different fields in this example, why?

Task 2: Test different analyzers

In general, any analyzer in Lucene is tokenizer + stemmer + stop-words filter. By using Lucene API, you can choose different kinds of analyzers and even tailor the analyzer.

Tokenizer splits your text into chunks, and since different analyzers may use different tokenizers, you can get different output *token streams*, i.e. sequences of chunks of text. For example, KeywordAnalyzer *doesn't split the text at all* and takes all the field as a single token. At the same time, StandardAnalyzer (and most other analyzers) use spaces and punctuation as a split points. For example, for phrase "I am very happy" it will produce list ["i", "am", "very", "happy"] (or something like that). For more information on specific analyzers/tokenizers see its Java Docs.

Stemmers are used to get the base of a word in question. It heavily depends on the language used. For example, for previous phrase in English there will be something like ["i", "be", "veri", "happi"] produced, and for French "Je suis très heureux" some kind of French analyzer (like SnowballAnalyzer, initialized with "French") will produce ["je", "être", "tre", "heur"]. Of course, if you will use analyzer of one language to stem text in another, rules from the other language will be used and stemmer may produce incorrect results. It isn't fail of all the system, but search results then may be less accurate. KeywordAnalyzer do not use any stemmers, it passes all the field unmodified. So, if you are going to search some words in English text, it isn't a good idea to use this analyzer.

Stop words are the most frequent and almost useless words. Again, it heavily depends on language. For English these words are "a", "the", "I", "be", "have", etc. Stop-words filters remove them from the token stream to lower noise in search results, so finally our phrase "I'm very happy" with StandardAnalyzer will be transformed to list ["veri", "happi"]. And KeywordAnalyzer again do nothing. So, KeywordAnalyzer is used for things like ID or phone numbers, but not for usual text.

In this task, please generate Lucene index for AP89 with the four analyzers listed in the table below. Let's only work with the <TEXT> field for this question. Fill in the empty cells with your observation.

Analyzer	**Tokenization applied?**	**How many tokens are there for this field?**	**Stemming applied?**	**Stop words removed?**	**How many terms are there in the dictionary?**
KeywordAnalyzer					
SimpleAnalyzer					
StopAnalyzer					
StandardAnalyzer					

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

Extra question (you don't have to answer in this assignment): *can you index the documents via n-gram (tip: org.apache.lucene.analysis.commongrams and org.apache.lucene.analysis.shingle)?*

Submission: Please submit java source code and your answers to the questions via Canvas system.