CS 276A Practical Exercise #1

(designed by Louis Eisenberg)

Assigned: Tuesday, October 12, 2004

Due: Thursday, October 21, 2004 by 11:59 p.m.

Review session: Friday, October 15, 2004, 3:15-4:05 p.m. in Gates B01

Delivery: All students should submit their work electronically. See below for details.

Late policy: Refer to the course webpage.

Honor code: You are allowed (but not required) to work in pairs for this assignment. Teams of two should only submit one copy of their work.

Overview

In this assignment, you will work with Jakarta Lucene, a Java-based text search engine library. Your main reference for familiarizing yourself with Lucene will be the Lucene API, located at http://jakarta.apache.org/lucene/docs/api/index.html. We provide you with a data set and starter code that handles most of the less interesting aspects of the assignment and implements a basic search engine; your task is to improve the performance of the search engine by exploring various features of Lucene.

How Lucene works:

When you feed it the corpus, it uses an IndexWriter to create for each document a Document object that consists of some number of Field objects. Each Field object has a name and a value, e.g. “title” and “experimental investigation of the aerodynamics of a wing in a slipstream.” When you close the IndexWriter, it saves the index to a collection of binary files in a directory. You can then run queries directly on the saved index – you no longer need the original corpus. You use a QueryParser to parse a query and pass it to an IndexSearcher, which returns a Hits object that is a sorted collection of Document objects representing the results of your query.

Before you start

Competence in Java is essential for this assignment. If you’re not comfortable with the Java programming environment, we strongly recommend that you find a partner who is.

You should probably complete the assignment using one of the Leland Unix machines such as the elaines or trees. Since the entire assignment is in Java, you’re free to use any platform you choose – but the course staff can’t promise any help with non-Unix platform-specific difficulties you may encounter.

To use the Lucene library, you must include the Lucene JAR file in your Java classpath. The file is located at /afs/ir.stanford.edu/class/cs276a/software/lucene-1.4-final/lucene-1.4-final.jar. If you are unfamiliar with modifying classpaths, try the following: Log in to your shell account and edit your .cshrc file (using emacs or your favorite text editor). Add the following lines to the very bottom of the file to add the Lucene JAR file to your classpath. After you have saved the modified .cshrc, you will need to log out and log back in or type “source ~/..cshrc” for the change to take effect.

```
setenv CS276A_CLASSPATH .:/afs/ir.stanford.edu/class/cs276a/software/lucene-1.4-final/lucene-1.4-final.jar

if ($?CLASSPATH) then
  setenv CLASSPATH ${CS276A_CLASSPATH}:$CLASSPATH
else
  setenv CLASSPATH $CS276A_CLASSPATH
endif
```

Supplied files

All of the files you will need are located in /usr/class/cs276a/hw1. Make a new directory and copy everything into it.

Data set: We’re using the Cranfield collection, a set of 1398 abstracts of aerodynamics journal articles. (If you think this is boring, you should see some of the other collections we considered…) This data set has special significance in the brief history of the IR field: it was the first corpus collection allowing accurate measurement of retrieval performance. Read more about it at http://portal.acm.org/citation.cfm?id=122861.

- cran.all: The corpus itself. For our purposes, we’re just interested in the T (title) and W (contents) fields of each entry.
- query.text: A set of 225 queries that you can run on this corpus. Note that the index numbers for the queries skip around some and are not exactly 1 through 225 – but to be consistent with the relevance judgments, just ignore the index numbers in this file and consider them to be numbered 1 through 225.
- qrels.text: Query relevance judgments: for each query, this file lists, one per line, the documents that human judges found to be relevant.

Starter code: The code does compile and run, but I can’t guarantee that it’s entirely free of bugs.

- indexer.java: This program goes through the cran.all file and creates a Lucene index that stores a tokenized representation of the title and contents of each document, plus an ID number that we’ll need when we’re evaluating the search engine’s performance.
- runqueries.java: This program loads the queries and relevance judgments, runs the queries on the index built by the indexer program, and evaluates its performance by comparing its results to the relevance judgments.

Other:

- english.stop (a list of very common words – a.k.a. stopwords – that you might find useful)

<u>What you have to do</u>

In this assignment, we measure the performance of a search engine by its top 10 precision: for a given query, how many of the top 10 documents returned by the search engine are actually relevant (according to the query relevance judgments)? A successful submission will achieve improvement over the baseline search engine by this metric. (More details on grading criteria are below.)

To accomplish this task, you need to:

1. Extend indexer.java to take advantage of some of Lucene's more advanced features. Specifically, you should implement all of the required extensions and some of the optional extensions.

Required:

a) Use IndexWriter.setSimilarity to replace the default scoring function (i.e. the function that determines how similar each document is to a query) with your own superior function. You will need to create a new subclass that extends the base Similarity or DefaultSimilarity class. Specifically, your subclass should override some or all of the following methods: coord, idf, lengthNorm, and tf.

b) Use Field.setBoost to try giving the title field increased (or decreased?) weight relative to the contents field – i.e. changing the relative importance of matching a query term to a term in the title field. Note that you should change the field boost values uniformly for *every* document; that is, each of the 1398 documents in your index should get the same field boost values.

Optional:

c) Replace the StandardAnalyzer currently supplied to the IndexWriter with a superior analyzer. The analyzer defines Lucene's policies for extracting index terms from the raw text of the corpus. The StandardAnalyzer tokenizes the text, converts it to lower case, and eliminates some stopwords. Potential modifications:

- Add Porter stemming. (This is actually extremely simple, since the Lucene library already includes a Porter stemmer. Look at the PorterStemFilter class.)
- Disable stopword filtering or try a different set of stopwords, e.g. the english.stop list.

Important: If you replace the StandardAnalyzer used by the IndexWriter, you almost definitely will want to make the same replacement in the query parsing code so that your queries are being processed in the same manner as your documents.

d) Anything else you can think of that might help (there's probably not much left beyond the items listed above): creating new filters to use in your analyzer, searching the title or contents exclusively, returning random documents…

2. Make a few small modifications to runqueries.java so it does what it's supposed to do. Right now it just runs a single sample query. What it should do is run all 225 queries and find the average precision. This part is trivial – we include it just to make sure you spend a few minutes working with the query generation and searching aspects of the code.

<u>Write-up and deliverables</u>

In addition to your code, you should write a report that summarizes your work. Think of this as an experiment: you're varying a number of factors and reporting the effects on performance. Outline form is perfectly acceptable (or maybe even preferable) as long as it's easy for us to follow. Your report should contain two main sections:

1. What you did: a brief list of the modifications you made in an attempt to improve the search engine
2. What happened: results and conclusions, what worked and what didn't, the optimal settings, etc. A chart might be helpful here.

How long it should be depends on how concisely you write, but aim for about 2 pages (not including any charts, figures, etc.).

So your submission should include:

- indexer.java, runqueries.java, and any other .java files you create (include brief comments within the code whenever what you're doing isn't obvious from the names of the variables and methods involved)
- your write-up in Word or PDF format (please call it results.pdf or results.doc)
- if necessary, a README file that lists any special information about your code that we should know

Make sure your code compiles cleanly on Leland computers! When you're ready to submit, change to the directory where all of your files are located and run the submission script: /usr/class/cs276a/bin/submit-pe1.

<u>Grading criteria</u>

quantity and quality of the extensions you implement: 50%
write-up: 25%
objective performance of your search engine (precision): 15%
coding style (brief comments as necessary, general readability): 10%