

CS276 Programming Exercise 1 (100 points)

Assigned: Tuesday, April 12, 2011

Due: Thursday, April 21, 2011 by 5p.m.

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

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

Late policy, Honor code: Refer to the course web-page.

In this assignment, you will:

1. Build a spelling correction engine.
2. Familiarize yourself with Jakarta Lucene, a Java-based text search engine library.
3. Integrate your spelling corrector with Lucene to enable more robust search on a document collection.

Your hand-in will consist of two parts both to be submitted electronically from a Leland Unix machine:

1. The code you've written.
2. A short document in succinct, readable English sentences with tables, charts or figures as needed. The length will depend on what you've done but, shoot for about 2-3 pages (not including program output, charts, figures, etc.). The document should describe for each part what you've done, Representative output from your program, any notable design decisions made or trade-offs considered, and an error analysis of when your system fails.

Competence in Java is important for this assignment. If you are not comfortable with the Java programming environment, we strongly recommend that you find a partner who is (see note at the end). We recommend that you complete the assignment using one of the Leland Unix machines such as the myths or brambles, although you are free to use any platform you choose. However, the final code submission script must be run from a Leland Unix machine, so all of your work must eventually be put into your Leland AFS space. The course staff can't promise any help with non-Unix platform-specific difficulties you may encounter.

Preliminaries

These instructions assume you will be working on one of the Leland Unix machines over ssh. If you have never logged in to the Leland Unix machines, you can find information here: <http://www.stanford.edu/services/unix/loggingin.html>. A basic set of Unix commands on Leland is here: <http://unixdocs.stanford.edu/unixcomm.html>. You'll need to edit code using a text editor and many such editors are available. We recommend using nano, emacs, or vi. Tutorials for all three are available online.

First, copy the project starter code into a new directory somewhere under your home directory (here named pe1):

```
cp -ar /afs/ir/class/cs276/pe1-2011/starter ~/pe1
cd ~/pe1
```

You'll edit the Java files contained in this new folder. Eventually, you will submit this whole folder as part of the hand-in (more below).

- The src/ subfolder contains starter source files and utility classes.
- The bin/ subfolder contains a compiled version of the src/ folder
- The lib/ subfolder contains the required Lucene jar for parts 2 and 3.
- The build.sh file is a script to compile the project and build.xml is an ant build script (more below)

The folder also contains .project and .classpath files and .settings folders --- if you use the Eclipse IDE <http://www.eclipse.org/>, you can import the whole folder as an existing Java project into your workspace.

Building and Running

If you are on a Leland machine, running the command **./build.sh** from the command line in your pe1 folder will compile all the source files under src/ into corresponding class files under bin/. If you are using ant (on a non-Leland machine), the ant build script in build.xml contains a target called "build" which will accomplish the same.

To run your program, invoke java from the command line as in the following example (substituting in the class you'd like to run):

```
java -cp bin/:lib/lucene-core-2.4.1.jar cs276.pe1.ClassToRun [arg1] [arg2 ...]
```

Including the Lucene jar in the classpath is optional when running code from part 1, but you won't be able to run part 2 without it.

Submitting your results

When you are finished with the project and the write-up, you will run a program to submit the full contents your pe1 folder electronically. You should include your writeup in your pe1 folder in either Word or PDF format (please call it results.doc or results.pdf), but you should remove the bin/ folder before submitting. When you are ready to submit, change to the directory where all your files are located and run the submission script:

```
/afs/ir/class/cs276/bin/submit-pe1
```

Part 1: Spelling Correction

In this part you will build a k-gram based spelling corrector. In lecture 3 (tolerant retrieval), we saw that one way to offer spelling suggestions was to count the proportion of character k-grams that overlap between the query word and known words from a corpus. Doing so will require an inverted index that maps k-grams to dictionary words and a scoring function that counts k-gram overlaps between candidate words. You are on your own for building the inverted index - you should not use any existing libraries or code snippets to do this for you, although we do provide some helper classes, described below. You should implement the Jaccard score of k-gram overlap as described in class as the scoring function for candidate replacements.

Take a look at the **cs276.pe1.spell.SpellingCorrector** interface:

```
package cs276.pe1.spell;

import java.util.List;

/**
 * Basic interface for the spelling corrector.
 *
 * @author dramage
 */
public interface SpellingCorrector {

    /**
     * Returns list of possible spelling corrections for a given
     * (possibly misspelled) word. The length of the returned list
     * is up to the implementer, but enough high-scoring candidates
     * should be included so that the best corrected spelling is
     * present (high) in the list.
     *
     * @param word A single word whose spelling is to be checked.
     * @return A list of possible corrections, with better corrections
     * ordered first.
     */
    public List<String> corrections(String word);
}
```

Your k-gram based spelling corrector will implement this interface to return a set of candidate spelling corrections for a given (possibly misspelled) word. It is up to you to decide how many candidates to return, but about 10 is a good starting point.

The following class skeleton (included) might help to get you started. The default constructor will read all the words from a large file of English text. The text file comes from Peter Norvig's page on [How to Write a Spelling Corrector](#), which takes an alternative approach to building a spell checker. [From Norvig's site: "The file is a concatenation of several public domain books from [Project Gutenberg](#) and lists of most frequent words from [Wiktionary](#) and the [British National Corpus](#)."] The skeleton uses some utility classes provided in the package **cs276.util** that you may find useful. It's worth taking a minute to skim through those classes to see what else they provide.

```

package cs276.pe1.spell;

import java.io.File;
import java.util.List;

import cs276.util.IOUtils;
import cs276.util.StringUtils;

public class KGramSpellingCorrector implements SpellingCorrector {
    /** Initializes spelling corrector by indexing kgrams in words from a file */
    public KGramSpellingCorrector() {
        File path = new File("/afs/ir/class/cs276/pe1-2011/big.txt.gz");
        for (String line : IOUtils.readLines(IOUtils.openFile(path))) {
            for (String word : StringUtils.tokenize(line)) {
                // TODO do kgram indexing
            }
        }
    }

    public List<String> corrections(String word) {
        // TODO provide spelling corrections here
        return null;
    }
}

```

Also included in the utility classes is **cs276.util.Counter** (with associated convenience methods in **cs276.util.Counters**), a simple way to associate objects with counts or numeric scores. For instance, if you place possible spelling corrections in an instance of **Counter<String>**, then calling its method **topk(10)** would return an ordered list of the 10 highest scoring spelling corrections.

You must complete the following sub-problems:

1. Implement a k-gram spelling corrector based on the words in the file big.txt.gz. Briefly describe your implementation and provide some example mis-spelled words along with the possible corrections your algorithm provides. What is the output of the **cs276.pe1.spell.SpellingScorer** class when run on this corrector (see below)? What are the sources of error?
2. In class, professor(s) mentioned that a k-gram based corrector could be used in concert with an edit distance function. We've provided an implementation of Levenshtein edit distance in the **cs276.util.StringUtils.levenshtein** function. Write a new implementation of SpellingCorrector called **KGramWithEditDistanceSpellingCorrector** that wraps the underlying spelling corrector and re-scores its output using the edit-distance function. In particular, use your k-gram spelling correct to get some number of closest candidates in the dictionary, score each with the Edit distance function and then suggest the corrections with lowest edit distance. Provide output, examples, and error analysis as you did for sub-problem 1.
3. Integrate word frequency information (how often each dictionary word appears in the input file) into one or both of the spelling correctors' ranking function. Which corrector did you modify? Why? How did you modify it? What was the impact of the change? Why?

Using SpellingScorer and Automatic Quantitative Grading

The included class `cs276.pe1.spell.SpellingScorer` provides a way to test the results of your `SpellingCorrector` implementations across a collection of spelling errors from Roger Mitton's [Birkbeck spelling error corpus](#) from the Oxford Text Archive. (Thanks to Peter Norvig for extracting a clean subset, which we use for scoring.) To run the SpellingScorer on the KGramSpellingCorrector that you modified in the first sub-problem, use the following command line:

```
java -cp bin/ cs276.pe1.spell.SpellingScorer cs276.pe1.spell.KGramSpellingCorrector
```

To run the SpellingScorer on the KGramWithEditDistanceSpellingCorrector from the second sub-problem, use the following alternative:

```
java -cp bin/ cs276.pe1.spell.SpellingScorer  
cs276.pe1.spell.KGramWithEditDistanceSpellingCorrector
```

The SpellingScorer loads a list of words with various mis-spellings from a file stored on AFS (`/afs/ir/class/cs276/pe1-2011/spelltest.txt`). It also creates an instance of the class named on the command line (which must have a public constructor that takes no arguments). The SpellingScorer then checks, for each mis-spelling, if the **first result** returned by the given SpellingCorrector is the expected corrected spelling. SpellingScorer will report the proportion of mis-spelled words correctly spelled.

You must include the output of this program in your writeup for 1, 2, and 3 above. We also may use the SpellingScorer on a different set of words to evaluate the quality of your implementation. Consequently, **all of the code you submit (both parts of the assignment) must compile and your SpellingCorrector must be usable through an unmodified SpellingScorer**. You will automatically lose points if your code fails to compile or if either of the two commands above crashes the SpellingScorer. It is important that your scores are reasonable and that you provide an analysis of the results, but you don't need the best scorer around to receive full credit from the grading script.

Part 2: Lucene

This section of the assignment involves setting up an index and performing queries with Lucene. Your main reference for familiarizing yourself with Lucene will be the Lucene API, located at <http://jakarta.apache.org/lucene/docs/api/index.html> or http://lucene.apache.org/java/2_4_0/api/index.html.

For this year we're going to be using a collection of IMDB movie plot synopses from <ftp://ftp.fu-berlin.de/misc/movies/database/> (not the "exciting" collection of aerospace abstracts used in past years). The basic task of this section is to get yourself familiar with Lucene and be able to do basic operations using it.

Building an Index

The IMDB movie plot database collection is a text file that, for many different *movie titles* contains *plot summaries* by one or more *authors*. These are the fields which we are interested in and what your Lucene index will comprise of. A typical record in the text file `/afs/ir/class/cs276/pe1-2011/imdb-plots-20081003.list.gz` looks like this:

MV: "\$1,000,000 Chance of a Lifetime" (1986)

PL: A short-lived quiz show hosted by TV veteran Jim Lange. Husband-and-wife

PL: teams would compete against each other in answering a tough series of
PL: trivia questions. The winning couple would then go on to the championship
PL: round, where they would have a chance to win one million dollars.

BY: Jean-Marc Rocher <rocher@fiberbit.net>

To help you get started, we've provided a parser to help read the collection into Java objects for you in **cs276.pe1.lucene.IMDBParser**. The parser can be used to iterate over a set of **MoviePlotRecords**. Your task is to fill out the skeleton code in **cs276.pe1.lucene.IMDBIndexer** to create a Lucene Index based on the three fields *title*, *plots*, *authors*. (**Note:** the full index will take up 127MB, so be sure to have enough space before running the indexer. By enrolling in cs276, your Leland Unix home directory will have been given an extra 300MB.)

After you have built your index, create a new class file to load the index and run some queries. Search for the following queries in your database. The output we want to see is a list of top 20 documents. Comment on each set of results if different from expected.

2.1

- Items that an author called 'Rob' has posted.
- Name of movie for which the plotline has the words 'murdered' and 'eighteen' within 5 words of each other.
- Documents for which the movie title is "10 items or less(2006)"

2.2

Consider the following scenario: You generally agree with reviews with the user whose last name is *X* (where *X* could be any plot summary author name from the raw IMDB file). You see that the last name is pretty common, and you think that the first name is *Y*. So you want to see results from everyone with last name *X* but you would like Lucene to return people with first name *Y* before everyone else. [eg: X='Hart' and Y='Rob'] .

Can Lucene handle such keyword-weighted queries? What is the syntax? Play around with what Lucene has to offer and comment on the quality of results produced due to your re-weighting on several examples. Under what conditions do the results tend to get better? Get worse?

Spelling correction for Lucene

Using Luke (see note below) or your own code, search for documents that contain 'Timmy' in their 'Title' field. What happens if you write 'Trmmy' (wrong spelling) instead? What spell check facilities does Lucene have built in? Document the different ways you found Lucene would handle spelling mistakes. Hint: the Spellcheck class would be one of them.

Edit your query code from "Building an Index" to automatically spell-correct queries before passing them to Lucene. You're expected to write a program in java that uses an object of your spellchecker class ,and does the following:

1. Take in a query from the user (as read from System.in). This query may be misspelled.
2. Run the spellchecker you wrote in part 1 on this input to suggest alternative spellings.
3. Using these suggested spellings, query Lucene, printing out the titles of top 20 documents.

Show output of your system on queries where the automatic spell correction does well and where it does poorly. Compare your results with Lucene's built-in spelling correction mechanism. Does your spellcheck work better than Lucene's? What advantages does Lucene's engine have over yours? How can your engine be improved?

Extra Credit (up to 20 additional points): Implement one of your suggestions for improvement to your spelling correction engine by making non-trivial use of the IMDB data or Lucene index. The quality of your idea, implementation, and writeup will all be considered.

An extra note about Luke:

Luke is a great GUI tool to look at the contents of your Lucene index. <http://www.getopt.org/luke/>. Using Luke for this assignment is **optional** but fun (may save some programming efforts in exploring your index). If you have been running your code on your own machine, you can download Luke and point it to the index you created above. If you have been running on a Leland Unix machine that you are currently logged into locally (not remotely), you can run Luke with the following command line:

```
java -jar /afs/ir/class/cs276/pe1-2011/lukeall-0.9.2.jar
```

If you are running remotely from a computer that supports ssh connections with X forwarding, you can also run the command above. If you are running remotely and don't know what X forwarding is (or you know you don't have it), don't worry, We recommend you play around with Luke on its homepage, but you don't need to use it to complete the assignment above.

A note on alternatives to Java:

For part 1 of the assignment, we strongly encourage you to use Java (don't worry, this should not require in-depth knowledge of the language). Java will allow you to make use of the provided stubs and help us use the exact commands listed earlier for semi-automatic grading. If you are not yet comfortable with Java, you are more than welcome to partner with someone who is and to learn from them. If it is absolutely necessary, for some reason, for you to use a language other than Java, please let the course staff know as soon as possible (by 5pm on Thursday, April 14th, at the latest), so that we could make proper arrangements for fairly grading your submission.

For part 2, you may use any programming language that you wish. For example, there is a C++ port of Lucene called Clucene (<http://clucene.sourceforge.net/>) and also a Python port, PyLucene (<http://lucene.apache.org/pylucene/>).