CSE454 Project Part 3: Web Ranker
Assigned: Friday, November 11, 2005
Due: 4pm, Wednesday, November 23, 2005

1 Project Description

For your previous assignment you ranked different documents by their relevance to a given query. While the text of these documents came from the web, your code ignored the most interesting thing about web pages: the links.

Your job, again, is to rank a set of web pages in response to a user query. However, this time you will incorporate knowledge about the web graph. You will have access to the link structure for every page in the collection, in addition to the textual data from the previous assignment.

For this assignment you will work in groups of two; **however, you must choose different partners from those used in part 2 of the project.** As soon as you determine your new groups, email Alan at cse454-ta@cs.washington.edu.

2 Project Objectives

By the end of this project, you should see how the web's graph structure can improve information retrieval tasks. You should know how the PageRank algorithm works, and how hyperlink anchor text can be used. Finally, you should understand how to incorporate these factors with the text-only scores you computed previously.

As in the previous assignment, you will turn in your code as well as a paper of no more than 4 pages. This paper should explain how your code works, describe the experiments you ran, why you chose the algorithm(s) you did, and any lessons learned (e.g., were there any decisions you now regret? Are you convinced that some of your decisions were good ones?). As always, if you incorpoiated any code, which you did not write yourself, then list this in an appendix. An appendix should also explain the roles of each project team member.

We recommend you take this part of the project in the following chunks:

- Implement the PageRank algorithm as described in class. Once you are sure your PageRank code is correct, you can move on.

  Note that PageRank may take a lot of time to compute. Once your implementation is correct, you will probably want to save your scores to disk. Reload them from then on.

- Select a text-only ranking function from Assignment 1. You may use code from either member of your group, or use new code (e.g., written by a different group), but your paper must document the origin of the code.

- Modify your text-only scoring function to also process hyperlink text. You now have two different bodies of text for each document: the textual content of the page itself, and a synthetic document that consists of all the text from incoming hyperlinks. These two text regions will have different TF-IDF scores, different lengths, etc.

- Create a method to incorporate all the information you now have: the PageRank score, the content text score, the incoming hyperlink text score, and anything else you might come up with. Combine all of this information to compute a single score. This score will be used to rank the results of each query.

You should be sure your code is clear and well-documented. Each method and class should have a header comment describing what's going on. All of the ranking code, especially, should have *very* detailed comments. An independent reader should be able to understand what your code is trying to do and why.

Be sure your paper explains:

- Your PageRank code
- Your body text ranker. Also include a brief explanation of why you chose this ranker, possibly with experimental results. This should be treated as a miniature version of the report from part 2.
- Your code that processes incoming anchor text. This could be similar to the body-text code.
- Any other useful features you devise.
- Your method for combining scores

Also, your paper should answer the following questions:

- What is the difference between PageRank and inlink counting? Can you give an example where PageRank is a clear improvement?
- Why did you combine scores as you did? Include experimental results to back up your decision. What pieces of information seem most valuable? What weighting works best?
- What other useful pieces of information can you think of? You now have access to a fairly rich set of information, and you might be able to come up with something quite clever. In what scenarios would your factor help? In what scenarios might it hurt? As usual, provide the best experimental evidence you can.

3 Getting Started

The cse454 support program *operates* just as it did previously. You can find it at: `/projects/instr/cse454-05au/assignment3/bin/cse454`

Note that the path is different! This new program will run slightly different support code from Assignment 2, so it is important that you switch to it.

Its standard output is almost exactly the same.

```
Usage:  cse454 query [-rankerclass your.ranker.classname] [-crawlsize med,large] [-maxHits
n] [-postProcessHits m] -query queryTerm0 (queryTerm1 ....)  [-rankerArgs rankerArg0
(rankerArg1 ....)]
```

- The `-maxHits n` optional argument lets you tell the search engine how many hits to emit for a single query. Maximum 100 hits.
- The `-query queryTerm0 [queryTerm1 ...]` arguments indicate the terms to be sent to the search engine. Nothing happens without these.
- The `-rankerArgs rankerArg0 [rankerArg1 ...]` optional arguments let you send command-line args to your ranking code. This might be helpful if you want to experiment with different score weightings, but you don't want to recompile every time you make a change. These command-line args are exposed to your code via the `DocumentSetInfo2` interface.

You must implement the class `edu.washington.cse454.DocumentRanker2` . This can be compiled against course code just as in project part 2. You should refer to the part 2 handout for details on setting environment variables, compiling, etc. (Of course, you should point to the Assignment 3 libraries.)

You can find skeleton code for this class at
`/projects/instr/cse454-05au/assignment3/samples/DocumentRanker2.java`
You should copy this file to your work area and use it as a starting point.

4 Interfaces

The tools are largely the same, but many of your coding interfaces have changed. You can find code for the following interfaces here:
`/projects/instr/cse454-05au/assignment3/reference`

4.1 IRanker2

The `IRanker2` interface is somewhat different in the current assignment. Your code must now implement several different functions:

```
public interface IRanker2 {
    /**
     * Called when the object is instantiated. Passes in a standard object that
     * gives information about the collection.
     */
    public void init(DocumentSetInfo2 info);

    /**
     * Called when there is a hit on the given document, on either the body or
     * the anchor text. Returns the relevance score for the given document.
     */
    public double getRelevance(String queryTerms[], Document2 doc,
            boolean bodyHit, boolean anchorHit);

    /**
```

```
     * Called for the top N hits on a given query. Allows Ranker to re-score or
     * annotate with text.
     */
    public double postProcess(String queryTerms[], Document2 doc,
            boolean bodyHit, boolean anchorHit, StringBuffer annotation);
}
```

4.1.1 DocumentSetInfo2

Here is the interface for the above-mentioned `DocumentSetInfo2`, passed in at initialization. Note new methods.

```
public interface DocumentSetInfo2 {
    ////////////////////////////////////////////
    // Useful for doc set overall
    ////////////////////////////////////////////

    /**
     * Returns an array of Strings passed in on the command-line.
     * This lets you pass arguments to your DocumentRanker, which might
     * be useful for running tests.
     *
     * Of course, your final code should work correctly *without* any
     * args being passed-in.
     *
     * If no args were given on the command line, this function returns
     * a zero-length array.
     */
    public String[] getRankerArgs();

    /**
     * Return how many hits will be printed to the screen
     */
    public int getMaxHits();

    /**
     * Return how many hits will be post-processed
     */
    public int getPostProcessHits();

    /**
     * Returns how many docs there are in the overall index.
     * All docs are numbered (0 .. numDocs()].
     */
    public int numDocs();
```

```
    /**
     * Get all the documents that the doc numbered 'docid' links to.
     */
    public int[] getOutlinks(int docid);


    //////////////////////////////////////////////////////////
    // Useful for computing doc scores re: contained terms
    //////////////////////////////////////////////////////////

    /**
     * Find how many indexed docs there are containing the given term.
     */
    public int numDocsContainingTermInBody(String term);

    /**
     * Find how many indexed docs there are containing the given term in incoming anchortext.
     */
    public int numDocsContainingTermInAnchortext(String term);

    /**
     * Find how many times the term appears in the document set (possibly
     * multiple times per page).
     */
    public int numTermOccurrencesInBody(String term);

    /**
     * Find how many times the term appears in the document set's incoming
     * anchor text (possibly multiple times per doc).
     */
    public int numTermOccurrencesInAnchorText(String term);
}
```

4.1.2 Document2

There's also the `Document2` class, which is passed in whenever the engine wants you to rate a doc's relevance:

```
public interface Document2 {
    // ////////////////////////////////////
    // Generic methods for the doc
    // ////////////////////////////////////
    /**
     * Returns the document id. All documents are ranked from 0 to
```

```
 * (DocumentSetInfo2.numDocs() - 1)
 */
public int getDocId();

/**
 * Allows caller to see the document's last-set relevance score. It's useful
 * to call this from inside postProcess(), to see what was set there last
 * time. If called from inside getRelevance(), this method will return 0.0.
 */
public double getRelevanceScore();

// ////////////////////////////////////
// For the document body
// ////////////////////////////////////
/**
 * Simply return the number of terms that appear in the Document. Note that
 * for efficiency's sake this number might not be completely accurate, but
 * it will be very close.
 */
public int bodyLength();

/**
 * Returns how many times the given term occurs in the Document. Must be one
 * of the original query terms.
 */
public int bodyFreq(String term);

/**
 * This is the normalizer of the tf-idf equation, computed with a base-10
 * log.
 */
public float bodyNormalizer();

/**
 * Returns an array of every position the given term occurs in the document.
 * This information can be used to compute proximity between query terms.
 *
 * If the term was not one of the original query terms, returns null.
 */
public int[] positionsInBodyText(String term);

// ////////////////////////////////////////
// For the document's incoming hypertext.
// ////////////////////////////////////////

/**
```

```
     * Returns the number of terms in the incoming anchortext.
     */
    public int incomingAnchortextLength();

    /**
     * Returns how many times the given term occurs in the incoming anchor text
     * for the Document. Must be one of the original query terms.
     */
    public int incomingAnchortextFreq(String term);

    /**
     * tf-idf neormalizer for the incoming-anchor document
     */
    public float incomingAnchortextNormalizer();

    // ////////////////////////////////////////////////////////////////////
    // The following methods are very useful, but are extremely
    // time-consuming to call. You can recreate most, but not all, of their
    // functionality using the methods above. There is little reason
    // now to call getText().
    //
    // If you really want to call one of the below methods, you should
    // avoid doing so inside every getRelevance() call. For example,
    // you might only examine the URL if the doc's score is otherwise
    // unusually low or high.
    //
    // ////////////////////////////////////////////////////////////////////

    /**
     * Get the URL of the current Document
     */
    public String getURL();

    /**
     * Get the extracted title text of the current Document.
     */
    public String getTitle();

    /**
     * Get the full un-tokenized content of the Document.
     */
    public String getText();
}
```

5 What to Hand In

- Your implementation of DocumentRanker2, with extensive comments. Be sure your code is clear and easy to understand.
- A description of your ranking code, no more than 4 pages in length. Be sure to cover the topics and question listed above.

  Also, make sure your description of your ranker and your choices behind it match your code and your code comments.

We will supply details on the hand-in process via the class mailing list.

6 Grading

The grade for this assignment will be computed as follows:

- Clear and sensible algorithm, cleanly implemented and described, 40 %
- Clear and convincing writeup, 40 %
- Ranking performance on human-judged test set, 20 %

The last element on the list consists of a set of ranking tests, compared against a set of human-judged rankings. There is one set of test queries that will be applied to every project, but this list will not be revealed until after the deadline (so that rankers do not overfit the test set).

As always, you can earn extra credit for features above and beyond the call of duty.

Late assignments will be penalized as described in the class policies at http://www.cs.washington.edu/education/courses/cse454/05au/policies.html.

7 Groups and Collaboration

You will work with one other person on this first project. You are expected to contribute equally. Also, each person should be knowledgeable about all aspects of the project. Some division of labor is fine when working, but by the end you must both know the entire project. Note that the report should detail the division of labor as described above.

Discussions between different groups are allowed, subject to the Giligan's Island rule and other important directives listed in the class policies.