CS458 - Assignment 3
Due: Sunday Oct. 21 at midnight

For the next step in our IR system we're going to be adding functionality to do ranked retieval using vector space models (i.e. TF-IDF). Given a query, the system will return a ranked set of documents using the cosine similarity between the query term vector and the document term vectors.

Before starting this assignment read through the **entire** document. Although I've given you more freedom this time, certain interfaces are made explicit to both assist you in developing your code and to make my life easier for testing. At the end I've included some helpful hints and tools that may be useful. If there is any ambiguity or question about what you are being asked to do, come talk to me.

Basics

For the query vector representation, we will always just use term frequency counts. For the document representation, we will support a number of normalization/weighting techniques that we have discussed, specifically:

- frequency modifications
  - no modifier (just the normal TF)
  - log frequency (log of TF)
  - boolean frequency (term either occurs in the document or it doesn't)
- weightings
  - no weighting
  - inverse document frequency weighting
- normalization
  - no normalization
  - cosine length normalization

Your index should be able to be built with a combination of these, depending on user specifications.

Although we will be modifying the data stored in the index, our new system will still be able to support BOTH boolean and ranked retrieval at the same time. Any query that contains "AND", "OR" or "!", will be treated like a boolean query and return the boolean query results. Maintaining this is not challenging, but I want to make it explicit so that you keep it in the back of your mind as you are modifying the code. For this assignment I am giving you a fair amount of freedom with how you implement this functionality. The main requirements are that you follow the minimal specification below, that your code work :) and that it works efficiently.

We will be building on top of the code that we used last time. I have provided a working version of the code from last time at:

`/home/dkauchak/PUBLIC/cs458/assign3/`

You are welcome to extend your solution to assignment 2, instead of using this starter code. If you decide to use your own code, you will be held accountable for any issues that carryover from assignment 2, though.

What to implement/changes

Below are descriptions of the classes you must implement. In each case, I have included a skeleton of the class in the code I provided you and have defined which public methods you must define. Since you may be reusing your own code, for pre-existing classes (e.g. PostingsList) I've explicitly listed the additional functions that you must add.

- Handling different query results: `QueryResult.java`

  To be able to support both boolean search and ranked retrieval, the `QueryResult` interface provides a shared interface. You don't have to do anything here, but take a look at this new interface. Two of our classes will implement this interface. The `PostingsList` class will implement it to support boolean queries. We'll be writing a new class, `VectorResult`, to support ranked queries.

- Posting list representation: `PostingsList.java`

  I have provided a postings list implementation for a boolean index. The postings list is a singly linked list created with a `Node` class. Each entry in the postings list is of type `Posting` which right now just stores the docID.

  To make life easier for you, I've made the `PostingsList` class `Iterable`, which means that you can use `PostingsList` objects in a "for each" loop, or you can just get an iterator over the postings in the list by calling the `iterator` function. This should make your life easier.

  You will need to change the `PostingsList` class to support functionality for doing ranked retrieval. The following MUST be changed, but you may also need to add other functions not listed here:

  - Notice that the `PostingsList` now implements the `QueryResult` interface. To support this, I've added the `getScores` function which just returns 1.0 for each docID returned.

- Rather than just occurring or not occurring, you must modify the postings list representation to support weights associated with each posting entry. For example with no normalization, this would represent the term frequency of a term in a document.
- Make sure that the other functions (`not`, `andMerge` and `orMerge`) still work appropriately for boolean queries.

- Ranking query results: `VectorResult.java`

  For a boolean query, the result is a `PostingsList`. For a ranked query, the result is a `VectorResult`. This class stores the results of our ranked query, which is a ranked list of the document ids and the score associated with each document. As with `PostingsList`, `VectorResult` implements the `QueryResult` interface.

  To start with, you can leave this as is, but as you fill in the details below, you'll need to fill in the implementation for this class.

- Index generation and query processing: `Index.java`

  As before, our `Index` class will generate and store the index as well as handle queries to the index.

  - Modify the index construction to keep track of term frequencies.
  - I've added enums in the index class that define our different normalization techniques. These are used to specify how we should normalize our document counts in our index when it is being constructed. To support these normalization techniques, the constructor has been modified.
  - Modify the index construction to support the different normalization techniques. This will require some cooperation with the `PostingsList` class so you may need to add some additional functionality there as well.
  - Implement the `rankedQuery` method, which issues a ranked query to our index:

    ```
    public VectorResult rankedQuery(String textQuery)
    ```

    This is should be roughly equivalent to the functionality in Figure 6.14 from the book, except we will return a full sorted list of results. Note that the resulting `VectorResult` must be sorted in decreasing order by score (ties should be broken by docID, with lower docIDs occurring before/earlier than higher docIDs). You can enforce this sorted order either in this method or in the `VectorResult` class, but either way, it must be efficient, that is only sort once.

    In implementing `rankedQuery` you'll need to fill in the details for the `VectorResult` class.

- Querying the index: `Search.java`

  The class `BooleanSearch` has been changed to `Search`. The class now supports ranked queries (which are the default), but also detects boolean queries (as described above). You'll also notice the index construction now passes in the normalization parameters to `Index`. You don't need to make any changes to this class, but it is provided as an interface into the index and for you to see how the index is created and used.

Hints/Comments

- If you're unsure about the normalization techniques, look at figure 6.15 in the book.
- Your code should be efficient! It will take a little longer to create the index and to answer ranked queries than before, but it should all still happen relatively quickly.
- When applying the normalization techniques, the easiest way to do it is to first create the index with just term frequencies. Then, for each of the normalization steps, make a pass over the index and modify the weights.
- The length normalization does NOT normalize the postings list length; it normalizes the document lengths. This is a little tricky to do, but look at the book/notes and think about how you might do this.
- We are NOT doing any normalization on the query. For the query, we're just using term frequencies.
- Note that there is a natural ordering to apply the normalization techniques (and make sure that the last thing you do is the length normalization!).
- Use natural logs for all of your logs (this is the default for java)
- Make sure you don't change any methods of the existing code. We want to be able to issue both boolean and ranked queries using our predefined interfaces.
- It can be useful to make a sample test set to check your results. Try as best as possible to do incremental changes and then test to make sure that functionality is working before adding additional functionality.
- The amount of code actually written for this assignment won't be a lot. The hardest part will be figuring out what to change. I would encourage you to spend an hour mapping out how you plan to modify the existing implementation before actually starting coding. This will make your life much simpler in the long run and save you time.

What to turn in and how to turn it in

What to turn in:

- A "jar" file of your code, which should contain all classes required to get your code working, including the original files I provided, the two new classes noted above as well as any supporting classes you need. See the assignment 1 writeup for details on creating a jar file.
- A text file with the following information:
  1. Name(s)
  2. What was the most challenging part of this assignment?

How to turn it in:

See the course web side for details (it's the same procedure as last time). Make sure to make a separate folder that contains the jar file and your text file and then copy this whole folder over (some of you didn't do this or accidentally copied the contents instead of the folder).