Homework 4: Information Retrieval [starter code]

In this assignment you will be improving upon a rather poorly-made information retrieval system. You will build an inverted index to quickly retrieve documents that match queries and then make it even better by using term-frequency inverse-document-frequency weighting and cosine similarity to compare queries to your data set. Your IR system will be evaluated for accuracy on the correct documents retrieved for different queries and the correctly computed tf-idf values and cosine similarities.

Data

You will be using your IR system to find relevant documents among a collection of sixty short stories by Rider Haggard. The training data is located in the data/ directory under the subdirectory RiderHaggard/. Within this directory you will see yet another directory raw/. This contains the raw text files of the sixty short stories. The data/ directory also contains the files queries.txt and solutions.txt (and solutions_java.txt for a more Java-friendly version). We have provided these to you as a set of development queries and their expected answers to use as you begin implementing your IR system.

Your Assignment

All changes for this assignment should be made in the designated sections in **IRSystem.py**.

Improve upon the given IR system by implementing the following features:

- **Inverted Positional Index:** Implement an inverted index - a mapping from words to the documents in which they occur, as well as the positions in the documents for which they occur.
- **Boolean Retrieval:** Implement a Boolean retrieval system, in which you return the list of documents that contain all words in a query. (Yes, you only need to support conjunctions for this assignment.)
- **Phrase Query Retrieval:** Implement a system that returns the list of documents in which the **full phrase** appears, (ie. the words of the query appear next to each other, in the specified order). Note that at the time of retrieval, you will not have access to the original documents anymore (the documents would be turned into bag of words), so you'll have to utilize your inverted positional index to complete this part.
- **TF-IDF:** Compute and store the term-frequency inverse-document-frequency value for every word-document co-occurrence:
- **Cosine Similarity:** Implement cosine similarity in order to improve upon the ranked retrieval system, which currently retrieves documents based upon the Jaccard coefficient between the query and each document. Also note that

when computing $w_{t,q}$wt,q (i.e. the weight for the word w in the query) **do *not* include the idf term**. That is, $w_{t,q}=1+\log_{10}\text{tf}_{t,q}$wt,q=1+log10tft,q.

The reference solution uses *ltc.lnn* weighting for computing cosine scores.

To implement these features, you must implement and/or improve upon the following functions:

- **index():** This is where you will build the **inverted positional index** (data structure that keeps track of the documents in which a particular word is contained, and the positions of that word in the document). The documents will have already been read in at this point. The following instance variables in the class are included in the starter code for you to use to build your inverted positional index:
    - titles (a list of strings)
    - docs (a list of lists of strings)
    - vocab (a list of strings)
- **get_word_positions(word, doc):** This function returns a list of integers that identifies the positions in the document **doc** in which the word is found. This is basically just an API into your inverted index, but you must implement it in order for the index to be evaluated fully.
- **get_posting(word):** This function returns a list of integers (document IDs) that identifies the documents in which the word is found. This is basically another API into your inverted index, but you must implement it in order to be evaluated fully.
    - Keep in mind that the document IDs in each postings list to be sorted in order to perform the linear merge for boolean retrieval.
- **boolean_retrieve(query):** This function performs Boolean retrieval, returning a list of document IDs corresponding to the documents in which all the words in **query** occur.
    - Please implement the linear merge algorithm outlined in the videos/book (do not use built-in set intersection functions).
- **phrase_retrieve(query):** This function performs phrase query retrieval, returning a list of document IDs corresponding to the documents in which all the actual **query phrase** occurs.
- **compute_tfidf():** This function computes and stores the tf-idf values for words and documents. For this you will probably want to be aware of the class variable **vocab**, which holds the list of all unique words, as well as the inverted index you created earlier.

- **get_tfidf(word, doc):** You must implement this function to return the tf-idf weight for a particular word and document ID.
- **rank_retrieve(query):** This function returns a priority queue of the top ranked documents for a given query. Right now it ranks documents according to their Jaccard similarity with the query, but you will replace this method of ranking with a ranking using the **cosine similarity** between the documents and query.
    - Remember to use *ltc.lnn* weighting! This means that the query vector weights will be $1+\log_{10}\text{tf}_{t,q}$1+log10tft,q with no IDF term or normalization, and we only normalize by the length of the document vector (square root of the sum of squares of the tf-idf weights).
    - When we say normalize by "document length" or "length of document", we mean the length of the document vector, NOT the number of words in the actual text document.

We suggest you work on them in that order, because some of them build on each other. It also gets a bit more complex as you work down this list.

Evaluation

Your IR system will be evaluated on a development set of queries as well as a held-out set of queries. The queries are encoded in the file **queries.txt** and are:

- separation of church and state
- white-robed priests
- ancient underground city
- native african queen
- zulu king

We test your system based on the five parts mentioned above: the inverted index (used both to get word positions and to get postings), boolean retrieval, phrase query retrieval, computing the correct tf-idf values, and implementing cosine similarity using the tf-idf values.

Running the code

Execute

```
$ cd python

$ python IRSystem.py
```

This will run your IR system and test it against the development set of queries. If you want to run your IR system on other queries, you can do so by replacing the last line above with

```
$ python IRSystem.py "My very own query"
```

where **My very own query** is your query.

Note that the first time to run this, it will create a directory named stemmed/ in ../data/RiderHaggard/. This is meant to be a simple cache for the raw text documents. Later runs will be much faster after the first run. *However*, this means that if something happens during this first run and it does not get through processing all the documents, you may be left with an incomplete set of documents in ../data/RiderHaggard/stemmed/. If this happens, simply remove the stemmed/ directory and re-run!

Submitting Your Solution

Submit your assignment via **Gradescope** (www.gradescope.com). We expect the following files in your final submission:

- IRSystem.py

You will only be able to see your score on the dev set. The test set scores will be released after the deadline.