Homework #3 (due in 3 weeks)

Computing Document Similarity

The principal function of inverted files is to support ad hoc querying where a document collection is relatively static, and a sequence of *a priori* unknown user queries is evaluated. In this assignment you will put together the pieces from the first two programs (*i.e.,* dictionaries and inverted files) and create a simple but true to practice retrieval engine based on cosine similarity using algebraic vector space models. This time you will need to process a substantially larger document collection than what we used on the first two assignments. You'll then demonstrate the ability to score and rank documents in order of their presumed relevance to queries. For full credit, you need to first build an index on disk (*i.e.,* as you did in Program #2), and then have your query processing program load the dictionary and retrieve postings (as needed) from the inverted file in order to rank documents for a provided set of queries.

Note: you are expressly forbidden from using existing text retrieval APIs/engines to build an index and compute document similarity (e.g., Apache Lucene or similar). The goal of this assignment is to learn how to implement this functionality yourself. If you have doubts about the permissibility of using open source code on this assignment, please post a clarification question to the discussion forum or contact me by email.

Dataset
For this assignment you will work with a modest sized test collection based on the FIRE 2010 data, which includes a collection of news stories that are about eight years old. The corpus is encrypted[1], and to complete the assignment, you will have to decrypt the corpus using the "gpg" tool[2] using a passcode that I will provide you with. Though similarly formatted, note that this collection has different characteristics from the *Bible and Les Miserables* collections: it is larger; the content is based around the Indian subcontinent; the text is encoded as UTF-8, and the historical period is recent. There are also some encoding oddities - for example, I believe that apostrophe and quote characters were replaced with spaces or deleted. There are approximately 120,000 documents that take up around 300 MB of text when uncompressed. I suggest running your Program #1 code on the collection, or performing a similar analysis, to see if you are satisfied with your program's tokenization. You might also consider testing/debugging your software on a smaller subset of the corpus (1% or 10%), especially if your processing times are non-negligible.

Query Parsing (10 points)
Along with the single SGML file containing documents, a similarly formatted set of queries for the FIRE 2010 English collection has also been provided on the course web page. Your retrieval program must automatically process this file and create a representation for each query (*i.e.,* a list of strings (terms) and weights (counts)). Print out the query vector representation your system uses for just the first query (*i.e.,* the processed query terms and their weights for topic #76).

Cosine Scoring (up to 55 points) **or Dot-Product Scoring** (up to 40 points)
Cosine Option (55 points): For full credit on this project, implement cosine scoring. Use TF/IDF term weighting for both documents and the query, and compute the cosine similarity measure for documents in the collection containing at least one of the query terms. Produce a single output file that contains ranked documents for all topics, according to the formatting instructions below. If you do not want to attempt cosine scoring, you can opt instead for dot-product scoring, with only a 15% deduction. Important details about implementing cosine scoring correctly include: (a) computing IDF values appropriately - the IDF weights are learned from the corpus, not the queries; and (b) properly computing document vector length, and query vector length.

Dot-Product Option (40 points): A simpler way to score documents against a query that still uses the inverted file information for each term, is to use the dot product measure. Use term frequency information from the query and documents to compute a score for each document that contains one or more of the query terms[3], and do not use IDF-weights. Here is a dot product example:

[1] The data are copyrighted and encryption protects against unauthorized uses; you should not redistribute the data or use it for commercial purposes.
[2] Available from: http://www.gnupg.org/
[3] You may remove stop words from your documents and/or query vectors. This is quite reasonable, and even a good thing to do. If you do this, just indicate to me that you are doing this, and rank documents that contain at least one of the remaining terms.

Query (Q): “cool sledding locations”
Document (D): “the best local sledding locations are near schools that have big hills, suitable for sledding”.

The term ‘cool’ doesn’t appear in D. The query term ‘sledding’ occurs 2 times and ‘locations’ appears once. Thus the dot product for this query and document would be (1*0 + 1*2 + 1*1) = (0 + 2 + 1) = 3.

Rank documents according to either metric and produce an output file carefully following the description below. Dot-product scoring does not generally produce as accurate of a ranked list as cosine scoring. In ranking documents, you may split ties in any fashion. **Note**: If you implement cosine scoring, you should not bother with producing a dot product ranking; only supply the cosine-based ranked list.

Batch Processing and Ranked Lists (20 points)
To put all these pieces together, your program should process an input query file and produce an output file that indicates the ranking of scored documents. A web search engine like Bing, Yahoo!, or Google usually offers the top 10 documents. For this assignment, I want you to provide the top 50-ranked documents for each query. Your single output file must follow ***precise*** formatting instructions. For each query (list them in the same order as the input file) include documents in ranked order (ascending, with 1 being the first rank). Each line should have six columns that are separated by a single space. The first column should be the query id (76, 77, ..., 125); the second should be the string “Q0” (read as 'capital Q zero'); the third should be the integer document id, the fourth should be the rank (1, 2, ... 50); the fifth should be your numerical score (real or integral, but no scientific notation like 1.2e-5); and, the last column should be your last name (no spaces). Your ranked list should contain the top ranked 50 documents for each query (unless you cannot find 50 documents with the query terms, which is unlikely), but do not include more than 50. This file should be named *yourlastname*-a.txt and be emailed to me at paulmac@jhu.edu.

Desired output format (example):
```
 76 Q0 396 1 0.287899 yourname
 76 Q0 498 2 0.260185 yourname
 76 Q0 100876 3 0.228558 yourname
 ...
 125 Q0 11223 50 0.38762 yourname
```

Stemming Experiment (15 points)
To explore whether stemming improves query performance, I want you to create a second, separate index of the collection that uses different tokenization. In documents and queries you should truncate any term longer than 5 characters (for example, “America” becomes “ameri”; “americans” also becomes “ameri”; "golf" remains "golf"). I call this 5-stemming. Using this second index produce another ranked output file named *yourlastname*-b.txt and email those results as well. If you like, you may opt-out of doing this stemming experiment – the penalty is only 15% of the assignment. Also, there are publicly available stemming algorithms (*e.g.,* Porter’s Snowball stemmer). You may use a different stemmer instead of 5-stemming if you are interested in doing so, but if you do, then briefly describe it (*i.e.,* explain it, or provide a URL).

Comments
- If you have trouble processing the entire collection, then it is better to have results on a portion (50% or 25% than none at all).
- Cosine scoring is rather more work than dot product.
- I recommend testing results on a very small collection. For example, see if your results agree with my examples from the lecture notes on the small “fruit” collection.
- In my opinion this is the most difficult of the programming assignments in the course. So I suggest an early start rather than waiting until the last few days.