Information Retrieval and Filtering

Programming assignment

Due: 20 August, 2000

Build a simple information retrieval engine based on the vector space model. The program should provide three operations:

- **build** *file list*
  Takes as input a set of text files (each one is a document), and computes the indexing files required for searching on the texts.
- **search** *query terms* Takes a query, expressed as a list of words, and returns a ranked list of 20 document filenames giving the best documents matching the query (according to tf-idf). You do not have to implement the "+" operator.
- **show** *rank* Shows the file with given rank, as if using "more".

An example run on Unix could go as follows:

% *search_assignment*
Command: *build ~argamon/Courses/IR/data/*.txt*
...Database created...
Command: *search tax increase decrease*
Documents found:

1. **article-108.txt**
2. **article-522.txt**
3. **article-21.txt**
4. **article-1032.txt**
5. **article-433.txt**
6. **article-928.txt**
7. **article-19.txt**
8. **article-723.txt**
9. **article-954.txt**
10. **article-628.txt**
11. **article-300.txt**

Command: *show 1*
article-108.txt:
SWEDEN RAISES FUEL TAXES TO FUND DEFENCE SPENDING
STOCKHOLM, April 1 - Sweden announced tax increases on petrol and heating oil from July 1, 1987 to help finance a 1.7 pct rise in defence spending over the next five years.
...

Extra features

The basic system is worth 90 points as a final project. The following features can increase your grade (but not above 100).

- Operators + and -, indicating required and forbidden words, respectively. (10 points)
- Relevance feedback, using either the Rocchio formula or the probabilistic approach. (20 points)
- A graphic form-based user-interface, either using HTML/CGI or another graphic interface. (15 points)

Grading

Programs will be graded on correctness, efficiency, and clarity, including comments and documentation. You should hand in the following:

- Documentation describing how to install/run the program, as well as which extra options you have chosen to implement,
- A printout of the code, including useful comments,
- Print outs of three example searches, including: *tax increase* and *increase oil gas profits*.

Implementation Notes:

- Data text files:
  They can be found in ~argamon/Courses/IR/data/article-*.txt

- Stemming:
  All words must be stemmed to a root form before indexing and before retrieval. Stemming code in C++ is in Porter.C and Porter.H in the IR/data directory. A test program is given in Porter_t.C.

  This code also includes stopwords.

  A version of the code that does not do stopwords is in PorterPlain.{C,H}.

- Word statistics: Term frequency = tf(w,d): this is the fraction of word occurrences in a given document d that are the word w:

  ```
  tf(w,d) = Count(w,d) / Count(d)
  ```

  Inverse document frequency = idf(w): this is the log of the inverse of the fraction of documents that contain the word w:

  ```
  idf(w) = log(N/Num(w,d))
  ```

  where N is the number of documents and Num(w,d) is the number of documents that contain the word w.

  Normalization coefficient of a document = norm(d): this is the length of the document's tf-idf vector, given by:

  ```
  norm(d) = SQRT( SUM_i [ tf(w_i, d) * idf(w_i) ]^2 )
  ```

- BUILD: This operation creates several index files as follows (the precise format is not important):
  - Dictionary: Gives the internal index numbers of all words found in the corpus. Should be read into a hashtable. Words stored here are *after* stemming.

```
word     index
word     index
...      ...
```

  - File index: Stores a list of the filenames of each document, with the document's sequential index for internal use, and the document's normalization coefficient. This shouuld be scanned into an array, so that indices can be turned into filenames.

```
filename   index        norm
filename   index        norm
filename   index        norm
...          ...        ...
```

  - Inverted index: This file stores the documents in which each word occurs and their associated tf values. The documents in each list should be sorted by their index values to make intersections and unions more efficient. The file should be read into an array of linked lists, one linked list for each word in the corpus.

```
wordindex       docindex / tf   docindex / tf ...
wordindex       docindex / tf   docindex / tf ...
wordindex       docindex / tf   docindex / tf ...
...
```

- SEARCH: This operation creates a query structure, containing, for each word in the query:
  - The index of the stemmed word
  - Its search weight

The search is carried out by accumulating a list of documents containing the union of the documents for each search term. Each document d in this list will have a rank associated with it which is computed as:

$$\frac{\sum_i tf(w_i,d) * idf(w_i) * weight(w_i)}{norm(d)}$$

where w_i ranges over all terms in the search.

After the list is constructed, it should be sorted so that larger scores are first, and then it should be presented to the user.

Weighting search terms: For now, the weight of each search term should be set to be the word's idf value. Later, we will extend this to include other possible weights for search terms.

- SHOW: This should use the ranked list to determine which document the user wants to see, find the associated file, and then show it using "more".