COM 4115/6115 Text Processing (2013/14)
Assignment: Document Retrieval

Task in brief: To implement and test a basic document retrieval system.

Submission: Submit your assignment work *electronically* via MOLE. Precise instructions for what files to submit are given later in this document. Please check that you can access the relevant MOLE unit (listed as "COM3110∼COM4115∼COM6115") and let me know if not.

SUBMISSION DEADLINE: 3pm, Monday, 25 November, 2013

Penalties: Standard departmental penalties apply for late hand-in and for plagiarism

Materials Provided

Download the file `4115_6115_Assignment_Files.zip` from the module homepage, which unzips to give a folder containing the following data and code files, for use in the assignment:

data files: `documents.txt`, `queries.txt`, `cacm_gold_std.txt`, `stoplist.txt`
`example_results_file.txt`

code files: `Collection.py`, `eval_ir.py`

The file `documents.txt` contains a collection of documents which record publications in the CACM (*Communications of the Association for Computing Machinery*). Each document is a short record of a CACM paper, including its title, author(s), and abstract — although one or other of these (especially abstract) may be absent for a given document. The file `queries.txt` contains a set of IR queries for use against this collection. (These are 'old-style' queries, where users might write an entire paragraph describing their interest.) The file `cacm_gold_std.txt` is a 'gold standard' identifying the documents that have been judged relevant to each query. These files constitute a *standard test set* that has been used for evaluating IR systems (although it is now somewhat dated, not least by being very small by modern standards).

Code files: If you inspect the files `documents.txt` and `queries.txt`, you will see that they have a common format, where each document or query comes enclosed within (XML-style) open and close `document` tags, that also specify a numeric identifier for the document/query. The Python class `Collection.py` provides convenient access to documents/queries in the manner of a simple iteration. In particular, if we create an instance of this class (supplying the name of the file containing the document collection or query set), then the `.docs()` method of that instance returns an iterator that will successively return the documents/queries, as illustrated in the following code example:

```
import Collection
file = "documents.txt"
collection = Collection.Collection(file)
for doc in collection.docs():
    print "ID: ", doc.docid
    print doc.lines[0]
```

Here, the document is returned as an instance of a `Document` class, that has two attributes: an attribute `docid` whose (integer) value is the numeric identifier of the document (or query), and an attribute `lines` whose value is a list of strings for the lines of text in the document.

The example code above prints the identifier and *first* line of each document in the collection. You are free to use the `Collection` class as part of your system (but this is optional).

The Python script `eval_ir.py` calculates system performance scores. It requires systems to produce a *results file* in a standard format, listing the documents it has returned for each query. Execute the script with its 'help' option (`-h`) for instructions on using the script, and on the required format of the results file. An example results file is provided as `example_results_file.txt`, so you can try the scorer out. (This file, *btw*, is real output from a previous student assignment, and its performance is at pretty much the upper limit of what is achievable on this task.)

Task Description

Your task is to implement a document retrieval system, based on the *vector space model*, and to evaluate its performance over the CACM test collection under alternative configurations, arising from choices that might include the following:

- **stoplist**: whether a *stoplist* is used or not (to exclude less useful terms)
- **stemming**: whether or not stemming is applied to terms. (A python version of the Porter stemmer is available from: `tartarus.org/~martin/PorterStemmer`).
- **term weighting**: whether just *term frequency* is used, or the TF.IDF approach

You should implement your retrieval system in Python, and you should ensure that it will run under a linux/unix evironment (such as that provided by the department), as it will be tested under such an environment when your work is marked.

What to Submit

Your assignment work is to be submitted *electronically* using MOLE, and should include:

1. Your Python code, plus a README file explaining how to run it. Credit will be given in regard to the extent of the implementation achieved, e.g. some partial credit for code that can index the collection and store the index to a file; more credit for achieving boolean retrieval using the index; and so on, up to producing a full ranked retrieval system that can be evaluated against the test set. Some amount of credit will also be assigned in regard to the elegance/comprehensibility of your code, and its presentation (i.e. comments, etc), and its ease of use (according to the instructions in the README).
2. A short report (as a pdf file), which should *NOT EXCEED 3 PAGES IN LENGTH*. The report should include a brief description of your system, including the extent of the implementation achieved (this is especially important if you have not completed the entire implementation). The report should also present the performance results you have collected, under different configurations, and any conclusions that you draw from your analysis of these results. Graphs/tables may be used in presenting your results, if this aids exposition.

Subtasks and a Possible Breakdown of Work

To help you break the work down into more manageable portions, the following notes suggest a subdivision of the work into specific subtasks, and a possible sequencing thereof. You are

not required to follow this suggested work programme, but you should read the notes anyway (since they contain various instructions/useful suggestions).

1. **Command line options/flags**: It is strongly preferred that you use *command line options* to parameterise your code's behaviour, for specifying input/output files, etc, as in e.g.:

   ```
   python myCode.py -s stoplist.txt -c documents.txt -i index.txt -I
   ```

   This example has options for the stoplist (`-s`), the collection (`-c`), and for naming the index file (`-i`). These options all have an argument string. The option `-I` is boolean, serving to 'switch' some aspect of behaviour (e.g. whether the code should read a pre-existing index file or create a new one). The processing of command line options is supported by the **getopt** module — see the Python Library Reference pages for details. (Alternatively, the **optparse** module provides greater functionality, but is more complicated to use.)

2. **Stoplist**: You might begin by defining a function (or class) to read and store the list of stop words (supplied in `stoplist.txt`). Remember to strip linebreaks from the input (or you will fail to match words correctly). Despite its name, you should **not** use a *list* to store the stopwords, as stopword testing must be fast/efficient (since every word in the collection must be tested this way during indexing). Python's `set` data structure is a suitable option.

3. **Tokenisation / preprocessing**: You can choose your own approach to tokenising the input, but a simple approach should suffice, e.g. simply extracting the maximal alphabetic sequences from text using a suitable regex, and mapping them to lowercase.

4. **Index data structure**: Your inverted index stores the counts of terms in different documents. A suitable data structure is a two-level dictionary, i.e. a dictionary which maps from *terms* to (embedded dictionaries that map from) *document ids* to *counts*. You will need an indexing function, to traverse the document collection to populate this data structure.

5. **Storing / loading the index**: In general, a retrieval system's index is not computed dynamically each time the system is run. Rather the index, once computed, is stored to disk, to be reloaded from there as required. This is how your retrieval system should work, so you will need a function that prints the inverted index you have computed out to a file in some suitable format, and another function to read such a file to populate an empty index. An obvious format to use stores information for one term per line, each such line starting with the term itself and being followed by as many docid/count pairs as needed.

6. **One program or two?**: One choice is whether to have two separate programs for indexing and retrieval or just one, i.e. one program to indexes the collection and write the index to file, and another program to read this index in, and use it to retrieve documents, *or* a single program that does both tasks as different parameterised behaviours. Clearly, however, there is common functionality required by both indexing and retrieval stages (e.g. use of stoplist, tokenisation, etc.), and it is preferable to avoid redundancy in your code.

7. **Simple boolean retrieval**: At this stage, you should be able to test your inverted index by adding functionality for simple (conjunctive) boolean retrieval. Queries might be entered via the command line (e.g. `... -q "parallel computing"`). Your preprocessing function could extract the non-stoplist terms from the query string, and for each such term, the set of documents containing that term is given by the inverted index. Intersecting these sets gives the set of documents that contain *all* of the query terms. You might just return this set of document ids, or traverse the collection to print them out.

8. **Computing required values**: Various numeric values that derive from the document collection are required later for term weighting and for calculating query/document similarity. Computing these values when the collection is indexed implies the need to store them alongside the index file. Rather than increasing the number of record files that need to be created/reloaded, a simpler approach is to compute these values instead over the inverted index (i.e. after it has been loaded from the index file). The required values are:
   - The total number of documents in the collection ($|D|$) — which can be computed by gathering together the full set of document identifiers for the collection
   - The document frequency df_w of each term w — which is easily computed, as the index maps each term to the documents that contain it
   - The inverse doc frequency $\mathsf{log}(|D|/df_w)$ of each term w, computed from the above
   - The *size* of each document vector, $|\vec{d}| = \sqrt{\Sigma_{i=1}^{n} d_i^2}$, i.e. the sum of squared weights for terms appearing in the document. This can be computed for all documents at the same time, in a single pass over the index. Where TF.IDF term weighting is used, the IDF values must be computed *before* the document vector sizes are calculated.

9. **Ranked retrieval for single queries**: You should now be in a position to write a function to do ranked retrieval for single queries. Some comments:

   - Queries might either be selected from the query set (by id number), or entered on the command line. (`Hint:` these two cases might be unified by wrapping up command line query strings as an instance of the `Document` class.)
   - Note that, unlike typical 'web queries', queries in the official query set may contain multiple occurrences of terms, whose counts must be taken into account in computing similarity values, i.e. the query must itself be analysed as documents in the collection have been, i.e. with (potentially) stemming, stoplisting, counting of terms, and so on.
   - The set of documents to be considered for ranked retrieval are those containing at least one term from the query, i.e. the *union* of the document sets for the individual query terms. Similarity scores are computed for each such candidate, and used to rank them, so some top N can be returned. Recall from class that query/document similarity is calculated as:
   $$sim(\vec{q}, \vec{d}) = cos(\vec{q}, \vec{d}) = \frac{\sum_{i=1}^{n} q_i d_i}{\sqrt{\sum_{i=1}^{n} q_i^2}\sqrt{\sum_{i=1}^{n} d_i^2}}$$
   Note, however, that the component $\sqrt{\sum_{i=1}^{n} q_i^2}$ is *constant for a given query* $\vec{q}$, and so can be *dropped* without affecting the *ranking* of candidates for that query.
   - Although the vector space model *envisages* documents as vectors with term weight values for every term of the collection, we *do not* actually need to construct these vectors. In practice, only terms with non-zero weights will contribute. For example, in computing the product $\sum_{i=1}^{n} q_i d_i$, we need only consider the terms that are present in the query; for all other terms q_i is zero, and so also is $q_i d_i$. (When we compute the *size* of document vectors, however, all terms with non-zero weights should be considered.)

10. **Retrieving for the full query set**: Finally, extend your code so that it can 'batch process' the entire query set, returning results in the format required by the scoring script.