COM 4115/6115 Text Processing (2016/17)
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 ("COM3110∼COM4115∼COM6115") and let me know if not.

SUBMISSION DEADLINE: 3pm, Monday, 21 November, 2016

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: `read_documents.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: Inspect the files `documents.txt` and `queries.txt`. You will see they have a common format, where each document or query comes enclosed within (XML-*style*) open/close `document` tags, that also specify a numeric identifier for the document or query. The code in the `read_documents.py` allows convenient access to documents/queries via a simple iteration. In particular, if we create an instance of the `ReadDocuments` class (supplying the file name of the document collection or query set), then we can access the documents from that collection in the manner of a simple iteration, as illustrated in the following code example:

```
from read_documents import ReadDocuments
documents = ReadDocuments('documents.txt')
for doc in documents:
    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 employ the code in `read_documents.py` (i.e. by importing it) in building your own system (but this is optional).

The script `eval_ir.py` calculates system performance scores. It requires systems to produce a *results file* in a standard format, listing the documents deemed relevant for each query. Run the script with its 'help' option (`-h`) for instructions on its use, and on the required format of the results file. An example of a 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 pretty much at 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.
- **term weighting**: whether term weights in vectors are *binary*, or are *term frequencies*, or use the TF.IDF approach

Your retrieval system should be implemented in Python, and it should run on all platforms.

What to Submit

Your assignment work should be submitted *electronically* via 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 credit for code that can index the collection; more credit for further aspects implemented; and so on, up to producing a full retrieval system that can be evaluated against the test set. Some amount of credit will also be assigned in regard to the elegance of your code, 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`). The option `-I` is boolean, serving to 'switch' some aspect of behaviour (e.g. between the code reading a pre-existing index file or creating a new one). See the lecture slides on using the **getopt** module.

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, 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. Instead, 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. **Stemming**: The NLTK library provides an implemention of the Porter stemmer, whose use is illustrated in the following:

```
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()

wd = 'baking'
print wd, stemmer.stem(wd)
```

5. **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. one which maps *terms* to (embedded dictionaries that map from) *document ids* to *counts*. You will need an indexing function, to count terms in the document collection and populate this data structure.

6. **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. Ideally, your system would also work this way, suggesting the need for a function to print the inverted index out to a file (in some suitable format), and another function to read such a file to populate an empty index.

7. **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 index 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.

8. **Simple boolean retrieval**: A good way to test your inverted index is to use it for simple conjunctive *boolean* retrieval (in contrast to the *ranked* retrieval we aim to achieve overall). The inverted index records the set of documents that contain any one query term. If we compute the *intersection* of the document sets for several query terms (e.g. *parallel* and *computing*), this should identify precisely the documents containing *all* of the query terms.

9. **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 directly from the inverted index. 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.

10. **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:

    - It should be possible to select a query from the query set by its id number. You might, in addition, allow query terms to be supplied as a string on the command line.
    - 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 should be analysed just 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 $q_i d_i$. (To compute the *size* of document vectors, however, all terms with non-zero weights should be considered.)

11. **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.