COM3110 Text Processing (2016/17)
Assignment: Document Retrieval

Task in brief: To complete a basic document retrieval system and evaluate its performance.

Submission: Submit your assignment work *electronically* via MOLE. Precise instructions for what files to submit are given later in this document. Please check you have access to 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 `3110_Assignment_Files.zip` from the module homepage, which unzips to give a folder containing a number of code and data files, for use in the assignment.

Data files: The materials provided include a file `documents.txt`, which contains a collection of documents that 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 three files together 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).

As discussed in class, a standard IR system creates an *inverted index* over a document collection (such as `documents.txt`), to allow efficient identification of the documents relevant to a query. Various choices are made in preprocessing documents before indexation (e.g. whether a stoplist is used, whether terms are stemmed, etc) with various consequences (e.g. for the effectiveness of retrieval, the size of the index, etc). To simplify the task, the files provided include several *precomputed index files* for the document collection, which were generated according to different *preprocessing choices*, i.e. whether a stoplist was used or not, and whether (porter) stemming was applied or not (e.g. such as the file `index_nostoplist_nostemming.txt`, and so on). Correspondingly preprocessed versions of the queries are also provided (e.g. such as the file `queries_nostoplist_nostemming.txt`, and so on). (As such, the original 'non-preprocessed' files `documents.txt` and `queries.txt` are not really required for the work you must do — they have only been been provided for information/inspection.)

Code files: The materials provided include the code file `ir_engine.py`, which is the 'outer shell' of a retrieval engine, that loads an index and preprocessed query set, and then 'batch processes' the queries, i.e. uses the index to compute the 10 best-ranking documents for each query, which it prints to a results file. Run this program with its *help* option (`-h`) for information on its command line options. These include flags for whether stoplisting and/or stemming are applied during preprocessing (which are used to determine which of the index

and query files to load). Another option allows the user to set the name of the file to which results are written. A final option allows the user to select the *term weighting scheme* used during retrieval, with a choice of `binary`, `tf` (term frequency) and `tfidf` modes.

The Python script `eval_ir.py` calculates system performance scores, by comparing the collection gold standard (`cacm_gold_std.txt`) to a system *results file* (which lists the ids of the documents the system returns for each query). Execute the script with its *help* option (`-h`) for instructions on use. 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.)

The program `ir_engine.py` can be executed to generate a results file, but you will find that it scores *zero* for retrieval performance. The program does implement various aspects of required functionality, i.e. it processes the command line, loads the selected index file into a suitable data structure (a two-level dictionary), loads the preprocessed queries, runs a batch process over the queries, and prints out the results to file. However, it does **not** include a sensible implementation of the functionality for computing what are the most relevant documents for a given query, based on the index. This functionality is to be provided by the class `Retrieve` which `ir_engine.py` imports from the file `my_retriever.py`, but the current definition provided in that file is just a '*stub*' which returns the same result for every query (which is just a list of the numbers 1 to 10, as if these were the ids of the documents selected as relevant).

Task Description

Your task is to complete the definition of the `Retrieve` class, so that the overall IR system performs retrieval based on the *vector space model*. Ideally, your implementation should allow retrieval under alternative term weighting schemes, as selected using the "`-w`" command line flag, i.e. under *binary*, *term frequency* and *TFIDF* schemes. You should then evaluate the performance of the system over the CACM test collection under alternative configurations, arising from alternative preprocessing choices and the available term weighting schemes.

What to Submit

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

1. Your Python code, as a modified version of the file `my_retriever.py`. Your implementation should *NOT* depend on any changes made to the file `ir_engine.py`. (Any such dependency will mean that your code fails at testing time, as testing will be done by placing your modified `my_retriever.py` alongside a 'fresh' copy of all the other files that are needed.)

   Credit will be given in regard to the extent of the implementation achieved (e.g. how many of the alternative weighting schemes you have implemented), and also in regard to the elegance/comprehensibility of your code, and its presentation (e.g. commenting, etc).

2. A short report (as a **pdf** file), which should *NOT EXCEED 3 PAGES IN LENGTH*. The report should include a brief description of the extent of the implementation achieved (this is especially important if you have not completed a sufficient implementation for performance testing), and should present the performance results you have collected under different configurations, and any conclusions you draw from your analysis of these results. Graphs/tables may be used in presenting your results, to aid exposition.

Notes and Comments

1. Study the code in the file `ir_engine.py`, particularly with a view to understanding: (i) how the retrieval index is stored within the program (as a two-level dictionary structure, mapping *terms* to *doc-ids* to *counts*), (ii) the representation of individual queries (as a dictionary mapping *query-terms* to *counts*), and (iii) how the code of the `Retrieve` class, that you are to complete, is called from the main program.

2. When retrieving relevant documents for an individual query, the set of *candidate* documents to be considered for retrieval are those containing at least one of the terms from the query, i.e. the candidate set is the *union* of the document sets for the individual query terms. Having computed this set, similarity scores can be computed for each, and used to rank the candidates, so that the top 10 can be returned.

3. The vector space model views documents as vectors of term weights, and computes similarity between documents as the cosine of the angle between their vectors. Stated in terms of the comparison of a document and a query, this 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 when we compute scores to *rank* the candidate documents *for a single query* (so that the top N can be returned), the component $\sqrt{\sum_{i=1}^{n} q_i^2}$ (for the size of the query vector) is constant across comparisons, and so can be *dropped* without affecting how candidates are *ranked*.

4. 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.)

5. **Computing required values**: Various numeric values that derive from the document collection are required for calculating term weights and similarity scores. These values can be computed from the inverted index. The required values are:

   a. The total number of documents in the collection ($|D|$) — which can be computed by gathering together the full set of document identifiers that are present in the index

   b. 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

   c. The inverse document frequency $\mathsf{log}(|D|/df_w)$ of each term w, computed from the above

   d. 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.