COM3110 Text Processing (2020/21)
Assignment: Document Retrieval

Task: Complete the implementation of a basic document retrieval system in Python.

Submission: Submit your assignment work *electronically* via the module's Blackboard page, so check you have access to this, and notify the module lecturer if not. Precise instructions for what files to submit are given later in this document.

SUBMISSION DEADLINE: 3pm, Friday, Week 7 (13 November, 2020)

Penalties: Standard departmental penalties apply for late hand-in and use of unfair means.

Materials Provided

Download the file `Document_Retrieval_Assignment_Files.zip` from the module homepage. It unzips to a folder that contains 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 lectures, standard IR systems create 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 stemming was applied or not (e.g. giving files such as `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 provided only for information/inspection. They are not required for the work you must do, and should *not* be accessed by your code.)

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.

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 Blackboard, and should include:

1. Your Python code, as a modified version of the file `my_retriever.py`. Do *NOT* submit any other code files. Your implementation should not depend on any changes made to the file `ir_engine.py`. (Any such dependency will cause your code to fail at testing time, as this will involve placing your code alongside a 'fresh' copy of the other files that are needed.) Your code file should not open any other files *at all*. Rather, it should take its inputs, and return its results solely through its interactions with the code in `ir_engine.py`.

2. A short report (as a **pdf** file), which should *NOT EXCEED 2 PAGES IN LENGTH* (excluding a title page, should you wish to have one). The report may include a brief description of the extent of the implementation achieved (this is only really 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.

Assessment Criteria

A total of 30 marks are available for the assignment and will be assigned based on the following criteria.

Implementation and Code Style (20 marks)
How many of the alternative weighting schemes have been correctly implemented? How efficient is the implementation (i.e. how quickly are results returned)? Have appropriate Python constructs been used? Is the code comprehensible and clearly commented?

Report (10 marks)
Is the report a clear and accurate description of the implementation? How complete and accurate is the discussion of the performance of the system under a range of configurations? What inferences can be drawn about the performance of the IR system from these results?

Guidance on Use of Python Libraries

The use of certain low level libraries is fine (e.g. `math` to compute sqrt). The use of intermediate level libraries like `numpy` and `pandas` is discouraged. Our experience is that students using these libraries do not use them effectively and end up producing code that is less clear and less efficient than those who simply implement from the ground up and thus retain clear control and understanding over what they are doing.

The use of high level libraries that implement some of the core functionality you are asked to produce (e.g. `scikit-learn` or `whoosh` or other implementations of the vector space model or aspects of it, computing IDF weights, etc) will be seriously penalised – this is the stuff you are meant to do yourself!

If in doubt about whether to use any 3rd party code, ask.

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. **Classic Error:** Many students make the mistake of "recreating the document collection" from the index, i.e. building a structure which associates with each document id the set of terms that the document contains. They then carry out document retrieval by searching through the set of terms associated with each document id in this structure to see which documents contain the query terms. Doing this defeats the whole point of building an

inverted index and does NOT scale to very large document collections. This error will be heavily penalised, so do take care to avoid it.

4. 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*.

5. 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 $q_i d_i$. (HOWEVER, when we compute the *size* of document vectors, *all terms* with non-zero weights should be considered.)

6. **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.