COM3110: Text Processing (2023/2024)
Assignment: Document Retrieval

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

SUBMISSION DEADLINE: 3pm, Friday, 17 November, 2023

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 Blackboard course. It unzips to a folder that contains a number of code and data files, for use in the assignment.

Document Collection and Benchmarking

The materials 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 records a single CACM paper, including its title, author(s), and abstract – although some of these (esp. abstract) may be absent for a given record. The file `queries.txt` contains a set of IR queries for use against this collection. (These are 'old-style' queries, where users might write a full paragraph stating their need.) 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*, or *benchmark*, that has been used for evaluating IR systems (although it is now rather dated, not least by being very small by modern standards).

The script `eval_ir.py` calculates system performance scores, by comparing the gold standard to a system *results file*, of which an example is provided as `example_results_file.txt`. Score this file by calling: "`python eval_ir.py cacm_gold_std.txt example_results_file.txt`" (This file, BTW, is real system output, and its scores are close to the upper limit of what is achievable on this task.) Execute the script with its *help* option (`-h`) for instructions on use.

Task Code Files

Standard IR systems create an *inverted index* over a document collection, 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).

The script `IR_engine.py` is the 'outer shell' of a retrieval engine. It loads a (preprocessed) index and query set, from the file `IR_data.pickle` (which must be in the same folder). It then 'batch processes' the queries, to compute the 10 best-ranking documents for each, which it prints to a results file. Run this program with its `-h` option, to list its command line options. These include options for whether stoplisting and/or stemming are applied in preprocessing, an option to set the name of the results file, and an option to specify the *term weighting scheme* used during retrieval (with a choice of `binary`, `tf` (term frequency) and `tfidf` modes).

The `IR_engine.py` script can be executed to produce a results file, but this will score *zero* for retrieval performance, as it does **not** yet have functionality for computing the most relevant documents for a given query. This functionality is to be provided by the class `Retrieve` which `IR_engine.py` imports from the file `my_retriever.py`. The `for_query` method of this class

(which performs retrieval for a single query) has a *dummy* definition, which just returns a list of the numbers 1 to 10 for every query (as if these were the ids of the relevant documents).

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*. Your code should allow alternative *term weighting schemes* (selected by the "`-w`" option) to be used, i.e. *binary* vs. *term frequency* vs. *TFIDF*. Evaluate your system's performance over the CACM test collection under the various configurations that arise from the alternative choices for preprocessing and for term weighting.

What to Submit

Submit your assignment work via the dropbox on Blackboard (under `Assessment`), by 3pm on Friday, 17 November, 2023. Your assignment submission should include:

1. Your code, as a modified copy of the file `my_retriever.py`. Do *NOT* submit any other code files. Your implementation should not depend on any changes made to `IR_engine.py`. (Any such dependency will cause failure at testing time, as a 'fresh' copy of the other files needed will be used.) Your code file should not open *any other files*. Rather, it should take its inputs, and return its results, solely by its interactions with the code in `IR_engine.py`.

2. A short report (as a **pdf** file), which *must NOT exceed 2 pages in length* (any title page or references do not count against this limit). The report should include a brief, high-level description of your implementation and should present the performance results collected under different configurations, and any conclusions you draw from these results. Graphs/tables may be used in presenting your results, to aid exposition.

Assessment Criteria

There are 25 marks available for the assignment, assigned according to the following criteria:

Implementation and Code Style (15 marks): How many term weighting schemes were correctly implemented? Is the code efficient (i.e. are results returned quickly)? Have appropriate Python constructs been used? Is the code comprehensible and suitably commented?

Report (10 marks): Is the report a clear and accurate description of the implementation? Are system performance results for a full range of configurations presented and discussed? Are sensible inferences drawn about the performance from these results?

Guidance on the 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. We find that students using them often end up producing code that is less clear and less efficient than those who simply implement from the ground up (who 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 `IR_engine.py`, to understand how the code of the `Retrieve` class (that you must complete) is called from the main program.

2. Within the program, the retrieval index is stored as a two-level dictionary structure, i.e. which maps *terms* to *doc-ids* to *counts*. (You might use a modified copy of `IR_engine.py` as a means to probe the data structure, to ensure that you understand this vital point.)

3. The queries are stored as a list of pairs of the form (*query-id, preprocessed-text*), e.g. with text such as `['parallel','languages','languages','for','parallel','computation']` for query 10 with no stemming or stoplisting, but when both are in use, it instead becomes `['parallel','languag','languag','parallel','comput']`. (Given such repeated terms, it makes sense, in your code, to convert this representation to a dictionary of term counts.)

4. Only documents containing at least one term from the query can achieve similarity scores above zero. All other documents can be ignored. The *inverted index* records, for any term, the documents in which the term appears. You should use the index to identify the *candidate documents* for which to compute similarity scores, to determine the top ranked candidates for return.

5. The vector space model views documents as vectors of term weights, and computes similarity as the cosine of the angle between the vectors. As a comparison between document and query vectors, this is calculated as shown on the right. However, when we compute scores to <u>*rank*</u> the candidate documents *for a single query*, 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*.

$$cos(\vec{q}, \vec{d}) = \frac{\sum_{i=1}^{n} q_i d_i}{\sqrt{\sum_{i=1}^{n} q_i^2} \sqrt{\sum_{i=1}^{n} d_i^2}}$$

6. Although the vector space model *envisages* documents as vectors with values *for every term of the collection*, we *don't* actually need to construct these (enormous) 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.)

7. Although you are asked to expand the code in `my_retriever.py`, so that the `for_query` method performs ranked retrieval, this does not mean that you should only add code to the definition of this method. As always, other methods can be added to perform coherent subtasks (and making your code more readable). This is illustrated by the definition (in `my_retriever.py`) of a method that computes, from the index, the number of documents in the collection. Other examples of methods that might reasonably be added include:

   a. A method to precompute the *inverse document frequency* value of each term in the collection, again based just on the inverted index. Thus, the index maps each term to the documents that contain it, whose number determines its document frequency.

   b. A method to precompute the *document vector size* for each document in the collection. Note that 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.