CS132a Information Retrieval spring 2021

Assignment 4: Building a simple Vector Space IR System

(Due Mar. 31, 2021)

Programming Problems

In this assignment you will implement a simple vector space information retrieval system supporting disjunctive ("OR") queries over terms, and apply it to the TREC corpus. The system should consist of two key components:

(1) An indexing module that constructs an inverted index, with term dictionary and postings lists for a corpus.
(2) A run-time module that implements a Web UI for searching the corpus, returning a ranked result list and presenting selected documents.

On LATTE you will find the file `cosi132a_hw4.zip` that provides:

1. A file `hw4.py` containing starter code to build the framework of your web app.
2. A file `utils.py` containing utility functions and search helper functions with `TODO` in them that you need to complete to make your web app work.
3. A file `text_processing.py` contains a class with `TODO` in them that you need to complete to normalize the text that is used to build the inverted index.
4. A file `inverted_index.py` contains functions with `TODO` in them that you need to complete to build the inverted index, disjunctive query over the inverted index and rank the results based on cosine similarity.
5. A file db.py contains functions with `TODO` in them that you need to complete to build a simple database to store WAPO documents.
6. A file `requirements.txt` listing all Python dependencies for this PA.
7. A folder `templates/` containing HTML templates.
8. A folder `pa4_data/` containing the documents your application will search for.

Before starting this assignment, activate the `cosi132a` conda environment and run `pip install -r requirements.txt`, using the file provided.
Finally, run `python hw4.py --build [INDEX-NAME]` to create the inverted index. Then run `python hw4.py -r` in your conda environment and type `http://127.0.0.1:5000/` in your browser to view and test your web application.

1. **Data Processing**
   The data we use for this assignment is still the same set of WAPO docs. The document loading and text processing parts are the same as with HW3. You should be able to reuse most of the code from your previous assignment. Note that unlike HW3, the function `get_normalized_tokens` in `TextProcessing` should return a list of normalized tokens instead of a set, because we need that information to compute term frequency. In addition, you will implement two static methods in `TextProcessing` to compute logarithmic TF and IDF.

2. **Sqlite Database for WAPO docs (Optional)**
   For this part you will use peewee to implement a simple Sqlite database to store your WAPO documents instead of pre-loading all the documents in the memory as in HW3. In the file `db.py`, you will need to define a data model (table schema), create the table and populate it with the processed documents from `load_wapo` and a simple query function to select a row based on the document id. Instructions and starter code are provided in `db.py`. Note that this part is optional.

3. **Query Matching**
   Your query should be disjunctive, i.e., a document must match at least one (non stopword) term in the query. If a term in the query is a stop word, your results page should indicate it by showing a message like "`Ignoring term: stopword1, stopword2, ...`" and then process the query without it. If a term in the query is not in your dictionary, your results page should indicate it by saying "`Unknown search term: term`". Your system should only search once when a new query is typed in, and the searching should be performed in the function `results` in `hw4.py`.

4. **Inverted Index Building and Weighting**
   `inverted_index.py` contains functions that you need to complete to build the inverted index, disjunctive query over the inverted index and rank the results based on cosine similarity. Weight *query terms* using logarithmic TF*IDF formula without length normalization. Weight *document terms* using log TF formula with cosine (length) normalization. Remember that computing the "length" of a document vector requires computing the square root of the sum of all (non-noiseword) term weights squared in the document. The length should be computed once per doc at index time and stored in a separate python shelf file keyed by document id.

5. **Cosine similarity ranking**
   Once you have computed the cosine similarity scores for all documents that match a query, find the top k scoring documents using a heap. As described in class, python provides a min-heap (heapq) which you can use for this. Sort documents by score and present 8 at a time (through the UI) in ranked order.

6. **Web UI**
   Folder `templates/` contains three HTML files: `home.html`, `results.html` and `doc.html`. `home.html` is given for free but you will need to fill in the other two to make these files functional. The functionalities of these HTML files are similar to HW3 except the following changes in `results.html`:
   - Above the result list, display the idf scores for each of the terms in the query.
   - To the left of each title in the result list, display its cosine similarity score in square brackets (rounded up to 4 digits after the decimal point.)
   - Below each document snippet, include a list of the query terms that were found in this document.

7. **Flask Application**
   In `hw4.py` you will write functions for URL routing and HTML rendering using Flask. This is also the main portal you will use to run your application. Instructions and specific requirements are included in `hw4.py`.

8. **Testing your system**
   Develop and test your system using a small corpus of 10~ hand-made documents before scaling up to your full corpus. Your hand-made documents should contain examples of things you want to test (tokenization, stemming or lemmatization, stop words, variants of the same term, etc.) Include your test corpus as a jsonl file (`test_corpus.jl`) and explain the tests you ran to verify that your system is working in your readme.pdf file.

Notes

- This assignment does NOT require writing CSS or Javascript, although you can still do so if you think that's easier for you or that can make your application look better.
- You can only use Python built-in packages and the ones listed in the requirements.txt for this assignment.
- A utility decorator `timer` is provided in `utils.py`. You may use it to test the running time of different modules for diagnostic purposes.
- Building the inverted index is the most time-consuming part, it took me ~8 minutes to build the index shelf file on a MacBook Pro (15-inch, 2019). Querying the index should be nearly instantaneous.
- You may feel a lag when you first load the home page every time. It's because the system needs to load all the TREC documents into your desktop memory. The loading will be faster when a database is used.

Submission

(1) A zip archive named `hw4_<last_name_first_name>.zip` containing:
   - `hw4.py` with your code and internal documentation, i.e., comments briefly describing what each data structure and function does.

- `utils.py` with your code and internal documentation.
- `text_processing.py` with your code and internal documentation.
- `inverted_index.py` with your code and internal documentation.
- `db.py` with your code and internal documentation. (optional)
- `templates/` with HTML files
- `README.pdf` with a system design description; a short write-up of how many hours you spent on this assignment, how difficult is this assignment and some general comments you have.

(2) Your `README.pdf` submitted separately.