Group 98497 and 98039

RI2022 - Assigment1 Feedback

December 13, 2022

1 Overview

Table 1: Overview of the issues found on the code submitted by Group 98497 and 98039. Documentation regards the overall description of the code. Modularity regards the decoupling, future expandability and overall organization of the code. Efficiency regards the algorithmic complexity of the code written, while good code regards some good coding practicing. Correctness takes into consideration if the module does what is supposed to do.

		Major	Inter.	Minor	P. Points
Code Analysis	**Tokenizers**				
	- Documentation	0	0	0	0
	- Modularity	0	0	0	0
	- Efficiency and good code	0	2	0	1
	- Correctness	0	0	2	0
	Reader				
	- Documentation	0	0	0	0
	- Modularity	0	0	0	0
	- Efficiency and good code	0	1	0	1
	- Correctness	0	1	2	0

Legend:
P. Points - Stands for Positives Points, which represent details that I found interesting and are worth to mention.

Changes done to execute the code:

- We needed to create a requirements.txt, since none was provided and the code had missing dependencies.

Constrains in the code:

- The collections must be in their compressed format.

Main API comment:

- The main API was correctly extended.

2 Code Analysis

2.1 Tokenizers

They implemented the PubMedTokenizer class that is responsible for the tokenization of the text. The main algorithm splits the text into words and regex is used to remove non alphanumeric characters. The filters applied are, in this order, stopwords, minimum token length, lowercase, and stemmer.

2.1.1 Documentation

The code has reasonable documentation, however, some functions could have a better documentation with a description.

2.1.2 Modularity

No additional modularity was added besides the already provided classes.

2.1.3 Efficiency and good code

Positive points:

- They implemented a global caching mechanism for the stemmer, which is by far the smartest move to do. Although their implementation as a minor flaw that it is not bounded by a limit max on terms stored. This means that through the execution of the program the caching will be gradually "eating" available memory for the indexer. So, my tip is to use a LRU (least recently used) type of cache, that will only keep the most frequent terms in memory, forgetting the rarest ones. This method should have a similar performance to the one implemented but will use considerable less memory.

Intermediate issues:

- At line 130, they are instantiating a new stemmer each time that a new term is seen. Thanks to the implemented caching system this turns out to not be a severe problem. However, it still a inefficient implementation. Since the class PotterStemmer does not implement the Singleton pattern, the correct implementation is to init the stemmer in the constructor.
- The stopwords are loaded into a list instead of a set. The list has lookup time of $\mathcal{O}(n)$ (in average), while set is $\mathcal{O}(1)$ (in average).

2.1.4 Correctness

Minor issues:

- The order of the filters is not entirely correct, since it may miss some stopwords. Also note that, the min token length filter should be the first to be applied in order to maximize performance.
- The program crash if a stopwords file is not specified, this is a BUG. The optional arguments should always have a default value.

2.2 Reader

They implemented the PubMedRead class, that **only** reads the document collection in their compressed format (.gz). The file is read sequentially and for each line the PMID is stored as a key and the title abstract (title+abstract) as values of a dictionary.

2.2.1 Documentation

Due to the simplicity of the code, I consider the code to be self-explanatory. However, class functions should have a description.

2.2.2 Modularity

No additional modularity was added besides the already provided classes.

2.2.3 Efficiency and good code

Positive points:

- Although the current implementation requires an extra attention to the memory management, they successful handle this problem, by creating a *MemoryManager* class abstraction that approximates the number of bytes that the data is occupying (which is faster than asking for the RAM available through the psutil lib).

Intermediate issues:

- The reader is "wasting" a lot of memory to store a set documents in memory. It is much more efficient to just read and yield each document at the time, this way the python program only has 1 document in memory at each step, leaving more available memory to the indexer. Note that the python and/or OS will already ensure that big chunks of the file in disk are read and properly cached.

2.2.4 Correctness

Intermediate issues:

- The implementation seems a bit strange since each document that is read is stored in a dictionary that is returned. This also makes the memory handling a bit more challenging, since now the reader must take into consideration the memory available.

Minor issues:

- The program crash if *–reader.memory_threshold* is not specified. This is a BUG since optional flags should always work with default values.
- The *title* and *abstract* strings are being directly concatenated. If the title do not finish in space or any punctuation, the first word of the *abstract* would be appended to the first word of the *title*.