Assignment 1

Filipe Pires [85122], Joo Alegria [85048]

Information Retrieval
Department of Electronics, Telecommunications and Informatics
University of Aveiro

October 2, 2019

Introduction

This report aims to describe the work developed for the first assignment of the discipline of 'Information Retrieval', explaining the overall processing pipeline. We include a short description of each class developed and of the respective methods, as well as the instructions on how to run our code.

The program implemented in Python version 3 has the purpose of indexing documents given as input in a compressed format. This document indexer consists of a corpus reader, a tokenizer and the actual indexer.

Along with the description of the solution, we also answer to a few questions proposed for the assignment (*1*).

1. Architecture

In order to maintain a modular architecture, we resorted to the Python library *Abstract Base Classes* (or simply ABC) (*2*). This module provided us with the infrastructure for defining abstract classes for each of our own modules. As seen in Fig.1, the 3 base classes all derive from an ABC.

The choice of this form of architecture was due to several reasons: modules allow the reduced coupling between system components, making it easy to replace or add new ones; creating variations of the solution (as we will see on the `Tokenizer` class) becomes far simpler and easier to manage; adopting this program structure also helps making the extension/adaptability of the program to other corpora structures easier to implement.

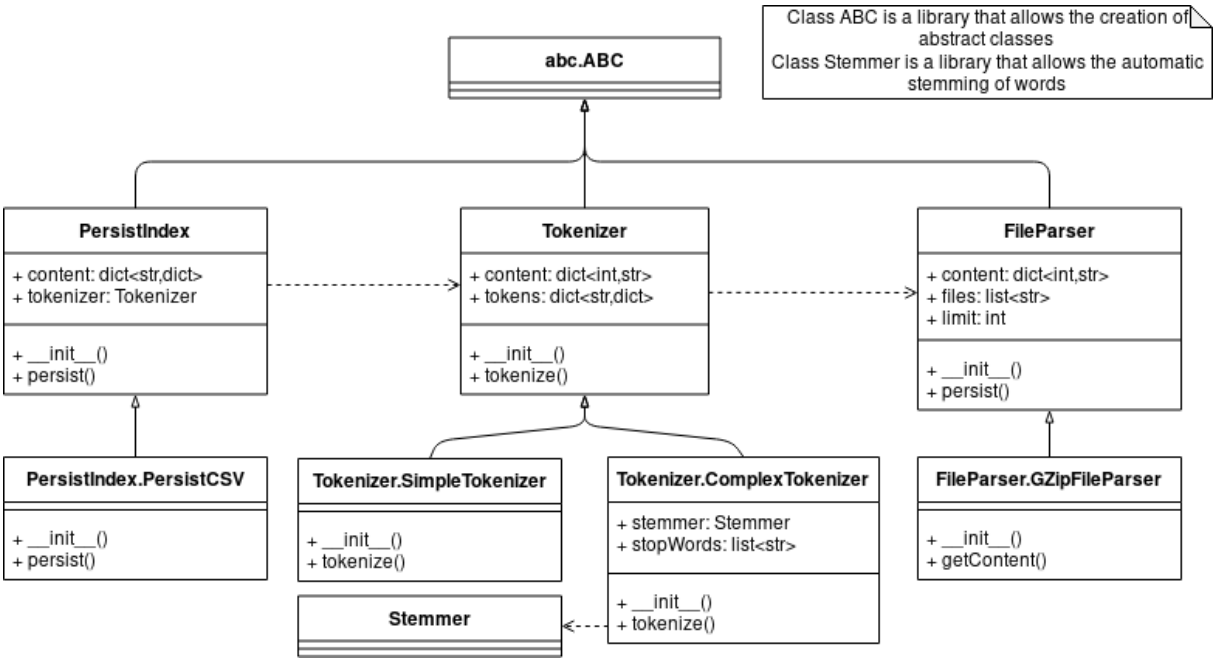

Figure 1: Program's class diagram.

The class `FileParser` is an abstract class for all file parsers. In our case, we only needed to develop 1 derived class to parse files on .gz format. It is `GZipFileParser` that actually reads the various compressed files passed to the program and dinamically processes them, meaning that the program will line by line decompress the file to retrieve the text bytes and with that information will create an dictionary that makes direct relation between a document id("PMID") and it's title("TI"). To improve our reading performance we used a reading buffer that significantly decreases the execution time.

`Tokenizer` is the abstract base class for our 2 Tokenizer implementations. This class depends on the `FileParser` as it uses the dictionary created by the file processing already mentioned. The `SimpleTokenizer` contains only one method called *tokenize()*, defined by

the parent, that replaces all non-alphabetic characters by a space, lowercases the entire content, splits on whitespace, and ignores all tokens(resulting strings after the split) with less than 3 characters. The `ComplexTokenizer`, on the other hand, is, as the name indicates, a more complex implementation aimed to return a more valuable set of tokens. This class integrates a stopword filter to ignore words considered not relevant and a stemmer to standardize all words, i.e. it normalizes the terms (or tokens) according to an algorithm that considers the English language by conflating words with the same linguistic base form. The stemmer used was PyStemmer (*3*), which is an implementation of Porter Stemmer, as recommended on the assignment instructions. For this reason, `ComplexTokenizer` depends on the class `Stemmer`. In addition this tokenizer also implements custom rules conserning pontuation, dates, emails, etc. We opted to implement restrictions in a way that will maintain dates, emails, phone numbers, money amounts and numbers in general, but in the other hand will eliminate any kind of pontuation(i.e. ,;.:!?" ...) and will separate if any whitespace, hyphen or underscore, considering the resulting terms as tokens too.

Finally we have the `PersistIndex` abstract class. Our implementation, `PersistCSV`, is the class responsible for persisting the ids of the documents where a given token appears and the number of occurrences in each of those documents, for all of the tokens from the entire vocabulary of the input files. This is done by storing the data present in a dictionary constructed by one of the previous tokenizers in a text file using the method *persist()*. To try to improve our performance we adopted an batch processing like system, were we store the token, all the document ID's where it appeared and the numbers of occurences on each document at the same time. After testing we observed that this approach improve our performance and execution time.

2. Data Flow

The file `Indexer.py` serves as the command to be executed in order to indexing all documents passed as arguments. This command has the following format:

```
$ python3 Indexer.py [-h] [-o outputFile] [-l limit] \\
  [-t tokenizer] inputFile1 [inputFile2]+
```

Here, `-h` is the option that presents the manual for the command usage. Options `-o` and `-l` allow the definition of the output file's name and of the limit for the number of lines to be processed in each input file. Option `-t` makes possible for the user to choose the type of tokenizer to be used. The alternatives are: 'simple' for the use of the `SimpleTokenizer` class, and 'complex' for the `ComplexTokenizer` class. The previous arguments are all optional and the actual values for these arguments must appear right after the respective options. The final argument(s) of the command must be the name(s) of the input file(s) to be indexed.

Once the arguments are validated, the program passes the file(s) name(s) to the FileParser that already is capable of processing multiple files. Once all the documents have been parsed, a choosen Tokenizer will access the processing resulting and will tokenize everything, that in this case is the title of each document. Lastly a PersistIndex class will persist the information created by the tokenizer in a file, make possible further consultation and analisys of the index.

Below we present the actual implementation in Python of the function responsible for parsing the information source file.

. . . . . .

3. Discussion

To test the capabilities of the developed software, we indexed 2 large compressed files made available along with the assignment description with the names `2004_TREC_ASCII_MEDLINE_1.gz` and `2004_TREC_ASCII_MEDLINE_2.gz`. Each file is a collection (corpus) of documents. In this chapter we discuss our implementation's performance over these files and answer the following questions:

a) What is the total indexing time and final index size on disk?

b) What is the vocabulary size?

c) What are the ten first terms (in alphabetic order) that appear in only one document (document frequency = 1)?

d) What are the ten terms with highest document frequency?

To answer question a), we simple added the command `time` in the beginning of the execution of the `Indexer.py` (e.g. `$time python3 Indexer.py (...)`). This command returned us 3 measured times during the execution, but we were interested only in one of them - the Elapsed real time (in seconds) since the execution start. After all code optimizations, the best value we could achieve was of `5'48''` (five minutes and forty-eight seconds) and an index size on disk of `470.3 Mbs` with the `SimpleTokenizer` and of `10'52''` (ten minutes and fifty-two seconds) and an index size on disk of `406.1 Mbs` with the `ComplexTokenizer`.

For the remaining questions, we developed a small script called `IndexAnalyzer.py` that processes the output file passed as argument. The calculated size of the vocabulary generated by the SimpleTokenizer was `346221` unique terms. The calculated size of the vocabulary generated by the ComplexTokenizer was `N` unique terms.

The 10 first terms (ordered alphabetically) with document frequency of 1 of the SimpleTokenizer are the following:

```
'aaaat', 'aaab', 'aaac', 'aaacr', 'aaact', 'aaaction', 'aaad',
'aaaga', 'aaagat', 'aaah'
```

The 10 first terms (ordered alphabetically) with document frequency of 1 of the ComplexTokenizer are the following:

```
XXX
```

These were retrieved first filtering out all terms with document frequency above 1 and then ordering the remaining tokens alphabetically with the use of the *quicksort* algorithm.

The 10 terms with highest document frequency with the SimpleTokenizer are the following:

```
'and' with 2044099 occurs; 'the' with 2044099 occurs;
'with' with 2044099 occurs; 'for' with 2044099 occurs;
```

```
'from' with 2044099 occurs; 'patients' with 2044099 occurs;
'human' with 2044099 occurs; 'cell' with 2044099 occurs;
'cells' with 2044099 occurs; 'study' with 2044099 occurs
```

To retrieve these terms, we stored the maximum number of occurrences during the output file's reading process and then search for the terms with that document frequency; if these terms are less than 10, we proceed on filtering for documents with the following highest frequency and repeat the process until the 10 terms are found.

Conclusions

Lorem ipsum ...

References

1. S. Matos, *IR: Assignment 1*, University of Aveiro, 2019/20.

2. Abstract Base Classes, Python.org, *https://docs.python.org/2/library/abc.html*, (visited in 01/10/2019)

3. Snowball Stem PyStemmer, GitHub.com, *https://github.com/snowballstem/pystemmer*, (visited in 01/10/2019)