Assignment 1

Filipe Pires [85122], João Alegria [85048]

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

October 1, 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.

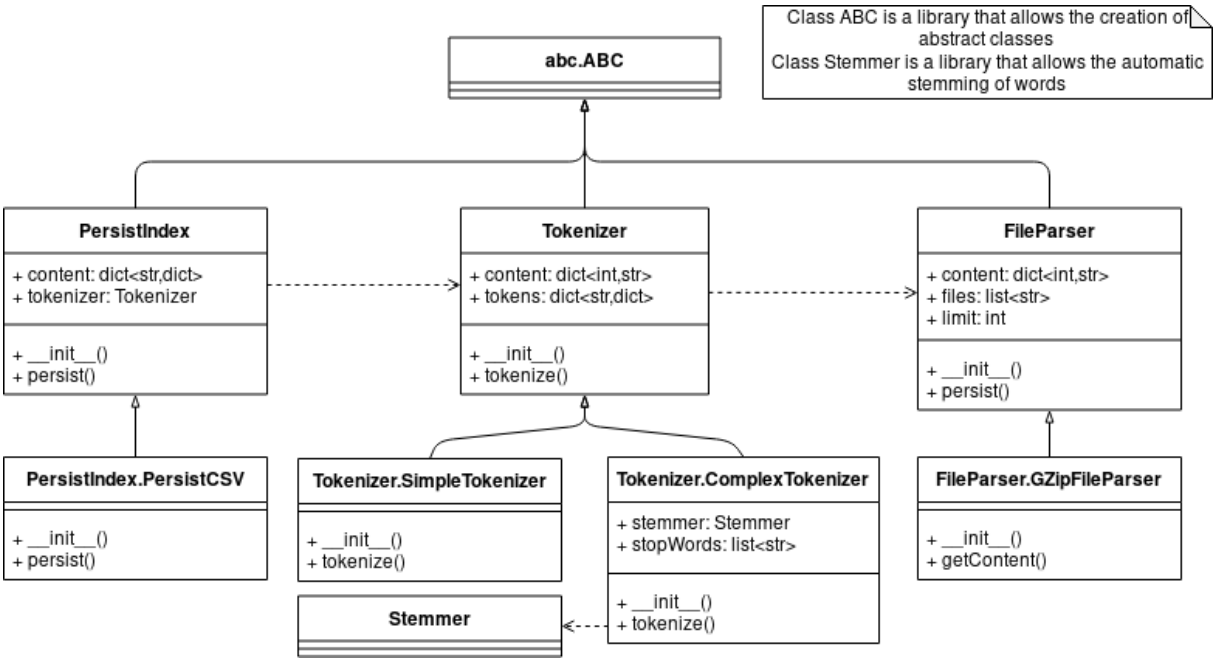

Figure 1: Program's class diagram.

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 .....

`Tokenizer` is the abstract base class for our 2 Tokenizer implementations. This class depends on the `FileParser` as it uses its ..... The `SimpleTokenizer` contains only one method called *tokenize()* that replaces all non-alphabetic characters by a space, lowercases tokens, splits on whitespace, and ignores all tokens 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`.

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 in a map structure (a Python dictionary) using the method *persist()*.

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 attempts to .....

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 `9'12''` (nine minutes and twelve seconds) and an index size on disk of `500.9 Mbs` with the `SimpleTokenizer` and of `14'15''` (fourteen minutes and fifteen seconds) and an index size on disk of `484.3 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 of both input files combined was of `N` unique terms.

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

. . . . . .

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

The 10 terms with highest document frequency are the following:

. . . . . .

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; these terms are less than 10, we proceed on filtering for documents with document frequency of 1 less document 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)