Second Assignment — Information Retrieval

André Nascimento (73152) and Rui Lopes (73197)

November 12, 2018

Problem Statement

In a first work it was asked, for us, to produce an index which could retrieve an easy way to look for a term on document. Having in mind such work, it was now asked to perform a query over the output index, with a slight modification—the current index must consider not the frequency of a given term on a set of documents, but the tf–idf relation between terms and documents.

This work will have its indexer made by processing a corpus present on the Amazon Customer Reviews Dataset [1].

System Architecture

This project has a set of modules with inherent dependecies between them. As asked in the last assignment, one should have a corpus reader class, a document processor, a tokenizer, and an indexer. Moreover, we do applied another class to our set of classes, named `RankedRetrieval`, which is responsible of applying the query actions. A class diagram of this project is depicted in the figure 1.

The adition we did in this work, namely the `RankedRetrieval` Java class, is then, as already mentioned, responsible for execute queries on the index we have created. Such class has methods for looking for a given `word` on the index (method QUERYSEARCH(WORD)) and retrieve the document given an index (method GETDOCUMENT(DOCID)). Both methods are described in the sections below.

Assignment Questions Answered

In the assignment documentation, some questions were asked by the teacher, which we hereby answer. In a first question it is asked to modify our index in order to retrieve the tf–idf relation between terms and documents, followed by a second question, where some query tasks must be perfomed, having in mind the creation of a new class that ranks results over the index.

First Question

In this first exercise we debuted our development with the index made on the previous work. Such index was created by coupling the document identification, on a term's posting, with the frequency it appears on it, which will call from now on as ϕ. Now we want to replace ϕ by the tf–idf relation between terms and documents, in a normalized appearance.

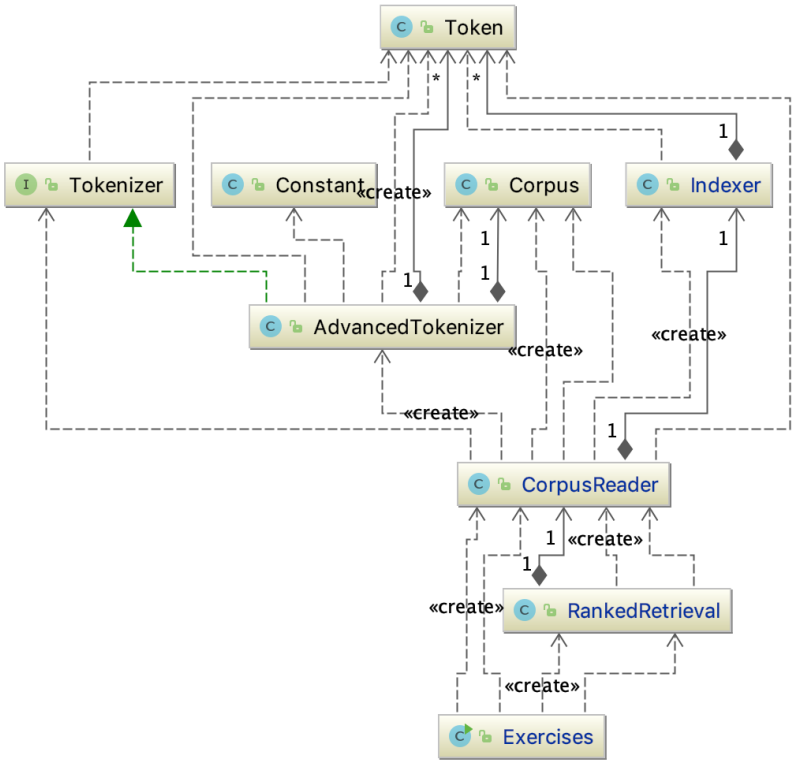

Figure 1: The Global System Architecture (Java Class Diagram)

To apply such logic to our problem, first we need to evaluate the *term frequency* of the term t on the document d, $tf_{t,d}$. This value is equal to the equation on (1), where $\phi_{t,d}$ is the pre-estimated frequency value of the term t on the document d, disregarding the computation of weights inside a document.

$$tf_{t,d} = 1 + \log_{10}(\phi_{t,d}) \tag{1}$$

Since we do want to complete our index in order to compare the placement of terms on the entire set of documents, one should now estimate the idf of the term t, idf_t. Such computation is made by relating the total number of indexed files, N, with the document frequency of a term t, DF_t, as we can verify in equation (2).

$$idf_t = \log_{10}\left(\frac{N}{DF_t}\right) \tag{2}$$

Although the equation (2) relates a term t with the set of documents we want it to happen having in mind the weight of the term inside a document. To compute so, we then have to multiply the value of tf by the idf.

In order to be able to reasonably compare values between terms in documents we then need to normalize them. This way, we can do operations with ease, as we will need to be able to sum the weights of terms in order to compute the entire set term's weight. To normalize such quantities, we then do the following, as stated by the equation (3).

$$\text{normalized}\,(tf_{t,d} \cdot idf_t) = \frac{tf_{t,d} \cdot idf_t}{\|\text{doc}_d\|} \tag{3}$$

In the equation (3), doc_d means the following, as represented in the equation (4).

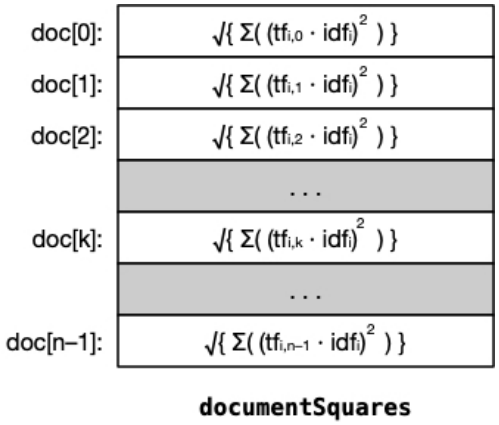

Figure 2: Auxiliar structure to compute document norm in (4)

$$\|\text{doc}_d\| = \sqrt{\sum_{i=0}^{k} tf_{i,d} \cdot idf_t^2} = \sqrt{tf_{0,d} \cdot idf_0^2 + tf_{1,d} \cdot idf_1^2 + \cdots + tf_{k-1,d} \cdot idf_{k-1}^2} \tag{4}$$

This normalization method will, in our case, be similar in both document and query actions. In sum, and in the SMART notation, we can then say that we follow an LTC.LTC weighting rule.

In terms of execution, since the last work, we now preserve the iteration to perfom the indexing of the documents. At the time of running this first iteration over the documents, we find the value of N, that is, the number of indexed documents. Since we need to estimate the $tf - idf$ relation, which its normalized computation requires (4), we then execute a second iteration over the set of documents to compute the norm of all documents. Such results are then preserved on a special data structure, which we call `documentSquares`, where its indexes are the document identification, as represented in (4).

After the latter step we run an ending iteration, where we only consult the meaningful data on `documentSquares` and the main dictionary, in order to compute the $tf - idf$ relation, as required by the designed index.

At the end we get a dictionary which keys are the terms of the *corpi* and its contents are their postings list, that is, list which have pairs document identification and its $tf - idf$ relation, already normalized. This dictionary, for memory usage issues, is partitioned in runtime in an array of blocks, already sorted by the term's first character.

A short representation of what is done within the indexing tasks can be found depicted on the figure 2.

Second Question

Having in mind that we already have a dictionary representing the output of an indexing task partitioned along several blocks on the disk, we now can perform some queries over this data structure. Plus, these queries must retrieve not only a result, but the best result, as results should be ranked by the definition of a new class, which we have named as `RankRetrieval`.

Such class has two methods called QUERYSEARCH(WORD)—which looks for a given WORD on the index— and GETDOCUMENT(DOCID)—which retrieves the document line where the term was found with a higher score.

Below we describe how does the method QUERYSEARCH(WORD) works.

Require: w as word to be looked for
1: $\hat{w} \leftarrow$ STEMMER(w)
2: max $\leftarrow -1$
3: wantedDocument $\leftarrow 0$
4: **while** THEREISANOTHERBLOCKTOREAD(WITHFIRSTLETTEROF($\hat{w}$)) **do**
5: dictionaryBlock $\leftarrow$ READNEWBLOCK()
6: higherDocumentID $\leftarrow$ GETKEYOFHIGHERVALUE(dictionaryBlock)
7: **if** MAXVALUEOF(dictionaryBlock) $>$ max **then**
8: max $\leftarrow$ MAXVALUEOF(dictionaryBlock)
9: wantedDocument $\leftarrow$ higherDocumentID
10: **end if**
11: **end while**
12: **return** GETDOCUMENT(wantedDocument)

Evaluation

Having finished the development of both index changes and the `RankRetrieval` class, we experimented some queries in order to validate our solution. As we are mainly working with two different datasets (Wireless reviews—around 4 GB of data,— and Watches reviews—around 400 MB of data), we started by testing our solution in the smallest set—the Watches.

Throughout the development of this work, before running our first tests we were expecting to have queries finding documents with the *right* amount of the lookup term within, not with the most times within. For instance, searching for a term *watch* over a set of documents where the document "watch watch watch" is present would not retrieve such document, but one where the proportion of the term *watch* and the others is quite *normal*. This happens due to the application of the relation $tf - idf$ between the terms and the documents on the total set of documents, where logarithm operations are then applicable, attenuating any effects of overuse of a certain term over a document.

Our first was then the execution of the query "watch" over the index of the "Watches" dataset.

> US 14642241 R1IPS15FFH8VG1 B00EO6EM96 952792417 Kenneth Cole Reaction Metal Diver's Style Men's watch #RK3230 **Watches** 4 0 0 N Y very nice **watch**! I have become a fan of Kenneth Cole **watches** for some time now... I own eight of them in various colors and functionality... mainly telling time, and color coordination with various suits, casual, and sports attire... I needed a look that had red in its makeup, and was still tastefully designed... this **watch** met that need... it also has a nice heavier feel to it that i like... Also, I get nice compliments from the ladies... :) 2014-02-17

As one should verify, the presence of the term *watch* is not overused, being this document the more ranked one. There are more documents where the number of times that the term *watch* appears is bigger in comparison to the total of terms inside the same document. Although that would be a great result in the last work, now, as we are establishing a relation $tf - idf$ between the terms and its existence on the entire set of documents, such result cannot be the best ranked one.

One of the most common terms in this set is *time*. Below is the example of such query.

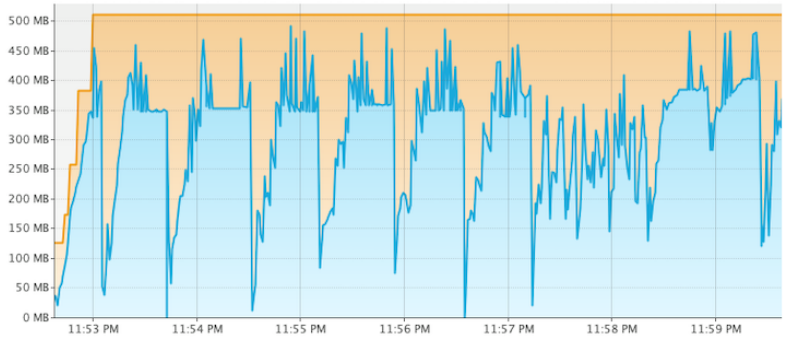

Figure 3: Memory usage between three iterations on Watches dataset

> US 23869115 R3I7HIQ01HNS2J B005F0Z4QK 52061931 Ultmost WA-281-EN Ana-Digi Talking Alarm Watch, English Watches 2 1 1 N Y Not such a good **time** Was expecting better quality. I know, I know, you get what you paid for. My pieve is with the actual **time** vs **time** viewed. The viewing **time** when using the analog feature does not keep up with the actual **time** but if you use the speaking **time** it does speak the real **time**. Not sure why or how this occurs only know how annoying it is. 2012-12-23

Again, there are plenty of documents where the number of terms *time* is more common. But since its proportion with the size of the document is overused.

The more common term that exists on this set is *black*, which leads us to the following result.

> US 48999136 RVTDAJ0TLTN5S B00470S8SK 620897482 Swatch **Black** Rebel Mens Watch SUOB702 Watches 4 0 0 N Y All **blacks** i like the all **blacks**.

my second swatch with **black** band, **black** hands, **black** face, **black** frame. would have been better if the adjustment knob was also **black**. 2013-03-27

After validating such results with the Watches dataset, we also did some testings over the Wireless dataset. The query we made, since it is a very common term, was *galaxy*, which retrieved the following result.

> US 49906433 R1H4ZTIYYT6KMJ B007Z0WEH0 333315966 SAMRICK - PURPLE - High Capacitive Aluminium Stylus Pen for Samsung S5300 **Galaxy** Pocket - i9220 **Galaxy** Note - i8160 **Galaxy** Ace 2 - S6500 **Galaxy** Mini 2 - i9070 **Galaxy** S Advance - i9100 **Galaxy** S 2 - i9300 **Galaxy** S 3 - S5360 **Galaxy** Y - i8150 **Galaxy** W - S5830 **Galaxy** Ace - S5570 **Galaxy** Mini - i9000 **Galaxy** S - i9250 **Galaxy** Nexus - S7500 **Galaxy** Ace Plus - i9001 **Galaxy** S Plus - S5670 **Galaxy** Fit - S5660 Ga Wireless 5 0 0 N Y Purple Stylus My favorite color is purple. It works great. I use it on my ipod and touch screen cell phone. Love it. 2012-12-23

In terms of complexity and time, the procedures on the Watches dataset revealed the following usage, as depicted in the figure 3.

The Wireless dataset revealed the following results, as depicted in figure 4.

Instructions to Run the Code

In order to test the program, one should take in consideration the following command line arguments: `java -jar RankedRetrieval.jar <operation> <word>`, where you can choose

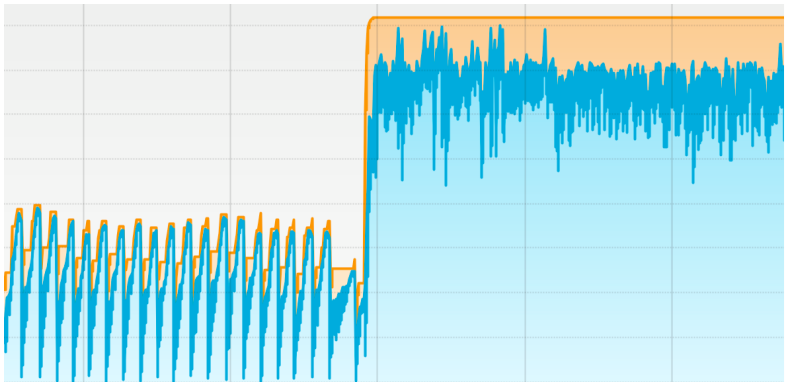

Figure 4: Memory usage between three iterations on Wireless dataset

between the operation 'Read Corpus, Index and Calculate Weights'(0) or 'Ranked Retrival Query'(1). For instance, to execute indexing run the following code:

```
java -Xmx512m -XX:+UseConcMarkSweepGC -jar RankedRetrieval.jar 0
```

To execute a word query run the following code:

```
java -Xmx512m -XX:+UseConcMarkSweepGC -jar RankedRetrieval.jar 1 "watch"
```

Note that in order to execute this program, you must create three directories at the same level as the `RankedRetrieval.jar` file—the `dataset` directory, where you should put the one and only dataset file you want to index; the `indexes` directory, where the program will output the index directories/blocks; the `reindexes` directory, where the program will output the index directories/blocks of calculated weights.

More, you must execute first the index and weigth calculation of the given dataset to proceed testing the program.

Conclusion

Our results did go the way we thought they would be. In fact, as we did not parallelize any of the made tasks, our solution lacks some optimizations in terms of both time and memory complexity.

Nevertheless, even with such downsides, our work reached all the goals the assignment has established and we could learn how to build a proper indexer from it.

References

[1] Amazon Custom Reviews Dataset. Last time accessed in October 22nd, 2018. Available in: (https://s3.amazonaws.com/amazon-reviews-pds/readme.html).

[2] Java's Runtime Class documentation. Last time accessed in October 22nd, 2018. Available in: (https://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html).

[3] Snowball, an implementation of the Porter Stemmer. Last time accessed in October 22nd, 2018. Available in: (http://snowball.tartarus.org/).

[4] VisualVM, a profiling tool for Idea IntelliJ. Last time accessed in October 22nd, 2018. Available in (https://visualvm.github.io/).