CSC 369: Distributed Computing
Spring 2020
Lab 8-1
Due: Wednesday, June 3, 10:00am

This is an individual assignment.

When executing it, please remember - you may make choices, as part of this assignment, that are different from mine, and result in a somewhat different result. This is OK. I am looking for you to understand the nature of Spark operations, and to utilize them effectively. Your code will be examined and counted as correct even when the outputs do not quite match (as long as you did the right thing).

It is **vastly preferable** for you to complete this assignment before Lab 8-2 is assigned on Wednesday, June 3 in lab. However, I will have the usual grace period for submissions to give you an opportunity to juggle multiple responsibilities.

Spark For Text Analysis.

We will be working with the Project Guttenberg documents located in /data/Guttenberg directory on HDFS.

We will use sc.`TextFile()` in situations where we need to import individual documents. We also will use `sc.WholeTextFiles(directoryPath)` to read in the entire collection of documents.

Information retrieval is a part of the overarching field of Knowledge Discovery in Data (which also contains Machine Learning/Data Mining as its constituent parts) that deals with analysis of textual documents and search for information in them. Information retrieval relies on two basic principles: *vectorization* and *similarity computation* to operate.

Vectorization is a representation of textual documents as numeric vectors of different features.

Similarity computations are computations that are interpreted as indicating how similar two vectorized representations of documents are to each other.

In this lab we will incrementally build some Spark methodology for both vectorizing and computing similarities between different textual documents. The lab is set as a collection of challenges - you need to produce pySpark code to meet each challenge, and accomplishing it will provide a stepping stone for the next challenge. We will start by working on some of the challenges together. The remaining challenges are for you to complete on your own.

Step 1. Finding 100 most popular words in an individual document.

This is a similar exercise to something you needed to do with MapReduce on Hadoop. It is so **on purpose.** Let us first take some document from the Guttenberg collection (we can start with "Alice in Wonderland", which is **`/data/Guttenberg/11-0.txt)`**

Our first task is to find, for this document the top 100 most popular words.

Path to solution. To accomplish this task, we decompose the problem a bit. The RDD provided to us by `sc.textFile()` contains individual lines in the document as individual RDD elements. First we need to break them into individual words. An example exists from our Wednesday, May 27 lab. After we have a list of words in the document, we can use standard grouping and aggregation techniques. The output of this process needs to be an RDD that contains 100 most popular words in the document together with the raw counts for these words.

Step 2. Not all words are created equal.

One thing we will notice when we find top 100 most popular words in a single document is that a lot of these words are not very interesting: they are articles, prepositions, verbs like "is" or "are", or "does", and so on. Our interest in describing the documents is to find "meaningful" words that are popular.

For this we need to filter out the words that we consider to be stopwords. Let us assume that a list of stopwords is a text document that contains one word per line, and is located in your home directory under the filename `stowords.txt (`you can build your own list of stopwords for this exercise, or "borrow" someone else's list). (For your convenience, I put a `stopwords.txt` file with over 400 stopwords in `/data` on HDFS. Please note, this file contains empty lines, your if you are processing it, you would need to run a filter that removes those empty lines).

Note: you may want to add some words like "guttenberg" to this list, although there is no need to go overboard here.

Write pySpark transformations that remove all words in the stopword list from consideration, and combine these transformations with your work on **Step 1.** to produce a list of "meaningful" 100 most popular words in a document.

In addition to stopword removal, write functions for Spark transformations that perform some basic cleanup:

- remove all non alphanumeric characters found in individual words (if they have not been removed on previous stages)
- remove empty words ("") from the list of words (if they are still there)

Organize your **Step 1** and **Step 2** code into a Python file **`top100.py`**. In that file, create function `getTop100(filename)` which is passed the filename of a file (assume a full HDFS path - this way, we can apply this function to a lot of different documents). and which returns back an RDD containing top 100 "meaningful" words from the file with their frequencies. The program itself should find top 100 words for Alice in Wonderland, and print it.

Step 3. Extend finding Top 100 meaningful words to the entire collection of documents.

On this stage you will extend the code you developed on Steps 2 and 3 to find the top 100 words of the entire Guttenberg collection (all 11 documents we have of it).

You will use `sc.wholeTextFiles(directoryPath)` to create the initial RDD containing all the data in our Guttenberg text collection.

After that, you will adapt the code you developed for Step 1 and Step 2 to produce an RDD containing the list of top 100 most frequent "meaningful" words in the entire collection.

Write a Python program **`frequentWords.py`** which both prints the top 100 most frequent words you found (together with their frequencies, and organized in descending order by frequency), **and** saves to HDFS (in your home or test directory) an RDD representing that data, **but sorted in alphabetical order.** You can save the document as a Pickle file, or in any other format - as long as your programs below can adequately read it.

Notes. One thing you will notice about the text in your RDD elements after `sc.wholeTextFiles()` is the explicit presence of `"\r\n"` combinations signifying the newlines. Make sure your transformations take them out (in some cases this can happen automatically, but there are also solutions where you may need to take care of these explicitly).

Step 4. Simple Vectorization.

Now, it is the turn of vectorization. A typical vector representation of a textual document uses individual words as features, and assigns each words a *weight* that corresponds to the importance of the word to the document. A very popular way to compute the word (term) weights is to combine two key characteristics of a given word:

- ***term frequency (tf) -*** *the frequency (i.e., how many times it appears) of a term in a specific document*
- ***document frequency (df) -*** *the number of documents in a document collection which contain at least one occurrence of the term.*

In practice for the weight of a term in a document to be high, we want *term frequency to be **high*** (the term shows up frequently), and the *document frequency* to be **low** (the term is exclusive to this particular document).

We are going to assemble the term frequency vectors for each document in the collection for the top 100 most frequent overall terms. Notice that you already have most of the code to compute the term frequency vectors from your solutions for Steps 1,2,3. We will resuse/repurpose this code here.

We are also going to construct a vector of document frequencies of each of the top 100 terms.

Notes. To construct the document frequency vector, let's remember *inverted indexes.* When you initially read your Guttenberg collection using **`sc.wholeTextFiles()`** the data represents a direct index - each element in the resulting RDD has the file name (full HDFS path to it) as the key, and individual words as values. An inverted index will use individual words as keys, and for each word - it will index the documents it appears in. Eventually we need just one entry per Guttenberg text file in the inverted index for each word (an analog of MongoDB's `addToSet()` aggregation).

A few RDD transformations may prove useful. Using **`flatMapValues()`** and **`mapValues()`** instead of **`flatMap()`** and **`map()`** respectively keeps the keys (e.g., the file names) intact during the process of cleaning. You can use already prepared (and pickled - it is actually the easiest way to store it) list of top 100 words to filter out all other words (generally speaking intersection and join operations may be useful, depending on the path you take). Finally, you can use `distinct()` to eliminate duplicates in the most opportune moment.

Let us write two programs. The first program, **`df.py`**, constructs and saves to HDFS an RDD containing the document frequencies of the words from the top100 list.

The second program, **`tf.py`** constructs and saves 11 RDDs containing term frequencies for each of the 11 Guttenberg documents. You can hard-code the names and locations of the 11 RDDs.

Step 5. Similarity computation.

We are at the final step of our process. When documents are represented in terms of term frequency and inverse document frequency, the usual way to compare them to each other and to construct similarity scores is to use ***cosine similarity.***

Given two vectors $d_1 = (w_1, .., w_n)$ and $d_2 = (v_1, .., v_2)$ the ***cosine similarity*** between these two vectors is their normalized dot product, i.e.:

$$cos(d_1, d_2) = \frac{\sum_{i=1}^{n} w_i v_i}{\sum_{i=1}^{n} w_i^2 \sum_{i=1}^{n} v_{i^2}}$$

Each vector for us will represent a single document, and will be formed as follows:

$$w_i = tf_i \cdot idf_i$$

Here, tf_i is the term frequency for the word *i* in the document **d,** and idf_i is the **inverse document frequency** of the word *i* in your collection which is computed (in our specific case) as:

$$idf_i = log_2 \frac{N+1}{df_i}$$

where *N* is the number of documents in the Guttenberg collection (11), and df_i is the ***document frequency for the word i*** that you computed on previous step.

This, for example, if a word "little" occurs in all 11 documents and occurs 24 times in "Alice in Wonderland", the the weight of the word "little" in "Alice in Wonderland" is going to be:

$$w_{little} = 24 \cdot log_2 \frac{12}{11} = 3.0127\ldots$$

Because all our term frequencies and document frequencies are positive, and because the inverse document frequency is guaranteed to be non-negative, the cosine similarity can range from 0 (two documents are NOTHING LIKE EACH OTHER) to 1 (two documents are essentially copies of the same document - although in our case, since we are only looking at the similarity of the top 100 words, the latter is not quite the case).

Last task. Let us write the final program, **`cosineSim.py.`** This program shall retrieve the document frequencies RDD, and the 11 term frequencies RDDs from where your **`idf.py`** and **`tf.py`** programs saved them.

It shall then compute the cosine similarity between every pair of documents in our Guttenberg collection.

Finally, it shall form an RDD that consists of the key representing the pair of documents being compared, and the value -- the computed cosine similarity between them based on the top 100 words.

Your program shall then:

- Pickle the prepared RDD to the file `similarities.pickle` in your home HDFS directories.
- Pretty print the output in the form of a table looking like this:

```
           docId1  docId2   docId3 docId5 docId6   docId7    docId8 docId9 docId10  docID11
docId1        1    sim12     sim13  ..                             ..                sim1_11
docId2      sim21    1      …                            …                           sim2_11
docId3                                     ….
docId5
docId6
docId7
docId8
docId9
docId10
docID11    sim11_1 sim11_2 …          …                                    …            1
```

(you can choose to output either the full matrix, or just the upper diagonal matrix of similarities).

Notes. You can compute all cosine similarities separately here, and then create the appropriate RDD via **`sc.parallelize().`** When working on this step, please remember to carry with you the identity of each document - the file name (without the path) is probably the best and the most succinct key, however, you are welcome to hardcode as keys short references to book titles.

To compute logarithms, you can import NumPy (pyspark is a full pass-through Python interpreter). To get the dot products, you can use the inner join transformation effectively to combine term frequencies and (inverse) document frequencies. The dot product itself has sums in it - look into **`reduceByKey()`** or some other aggregation transformation.

Deliverables and Submission.

Submit all programs you built for this assignment: **`top100.py, frequentWords.py, tf.py, idf.py, cosineSim.py.`** If you place all the functions you defined for the pySpark computations into a separate Python file (a separate package, e.g.), submit it as well.

Include a README file specifying how to run the programs. In general all your programs shall be runnable in one uninterrupted sequence, and individual programs must successfully run using **`spark-submit`** tool.

Submit using handin from the unixN.csc.calpoly.edu machines.

```
$ handin dekhtyar lab08-1 <files>
```

Good Luck!