CS 5123: Cloud Computing and Distributed Systems
Assignment – 3: TF-IDF in Spark

This assignment is to be done *individually*. The assignment will utilize core PySpark programming. Students can execute their programs in either Google Colab or the CS-X Spark cluster. The assignment comprises of ***two tasks*** and a ***bonus task***. **NOTE:** Learn the basics of PySpark programming and Spark's *Transformation* and *Action* functions before starting this assignment.

Total Points: 100
Percentage towards final: 10%

TF-IDF on Wikipedia:

The goal of this programming assignment is to develop a simple search engine to get relevant documents based on simple TF-IDF measures. The goals of this assignment are:

1. Understand TF-IDF steps and how it works in Spark
2. Construct TF-IDF index in Spark
3. Develop a system for users to submit queries to the TF-IDF index and retrieve relevant results
4. Examine the output on English Wikipedia dump

This assignment is more involved than Assignment – 2, given the nature of new Spark programming, and big data dump.

Data:

We provide two datasets for this assignment (**Assignment3Data_small.zip and Assignment3Data.zip**). All datasets are crawled from latest updates made on selected articles in English Wikipedia corpus. The actual corpus is about 2.2 GB spread across 700,000 files – one per article. However, HDFS is designed to operate efficiently on small number of large files rather than many small files. Students will be using pre-processed version of the sampled Wikipedia corpus stored in .txt file, with 1000s of small Wikipedia articles combined in to a single .txt file. Even with the pre-processed data, students are warned to expect high volume of noise and special cases in the data.

Each page in the original Wikipedia dump is represented in XML as follows:

```
<page>
<title>Page title</title>
(other meta-data we do not care about)
<revision>
<text optionalAttributes> Page content goes here
</text>
</revision>
</page>
```

This has been further pre-processed such that each article/page is on the same line. This makes students work easier to use the default *textFile()* function in Spark to read an input file. With a simple *map()*, Spark reads the entire article when reading a line. Each article is compressed in a single line like:

<docid>Page ID</docid><text>(Page content)</text>

The page content also has all newlines converted to spaces to ensure all lines stays on one line. Page IDs are attached to each and every article. Students have to consider this as document ID (doc. ID) when calculating TF/IDF/TF-IDF. The Page content will contain page title also.

Assignment3Data_small.zip contains very small data – 1 .txt file (with roughly 10,000 articles). Use this data to develop and unit test your program.

Assignmen3Data.zip contains 10 .txt files (each file with approximately 10,000 articles). ***Use this data only when you are confident that your program works without any bugs***. Program with bugs can cause severe runtime performance issues like slow execution, system crash, etc. when executing on big data

Task – 1: Build Term-Frequency (TF) index [*60 points*]
Create a program named **TF.py** for tasks given in Step – 1 and Step – 2.

Step – 1: Text processing
The Wikipedia data comes with a lot of special characters, stop words, and words that does not have any meaning. We are going to do (i) Case folding (UPPERCASES to LOWERCASES), (ii) Stop words removal (using broad stop-words list from https://github.com/stopwords-iso/stopwords-en/blob/master/stopwords-en.txt), and (iii) punctuations removal (remove all non-alphabetic characters and words). In addition to the above tasks, we need to perform the following couple of tasks:

a) Remove URLs

Articles in Wikipedia are heavily linked to one another. Thus, each page has links pointing to other pages. We will remove these URLs to reduce the complexity of dictionary creation. In the given data URLs are prefixed with **http://….. or https://…. or www.**. You can completely remove any URLs, if appeared in the data.

b) Remove tags

Although the data is processed, it still contains some HTML/XML tags due to minor formatting issues in the data. You have to programmatically handle skip these tags before creating indexes. Following are some examples:

<body> Corona virus </body> ➔ Corona virus…
As reported by CNN…
 ➔ As reported by CNN…

Step – 2: Term Frequency index construction

Assuming, we have the processed data and document ID for each page, we can easily construct a Term Frequency matrix. An easiest approach would be to get file names (doc. IDs) of each word and calculating the frequency of each word in each document. We cannot construct the actual matrix required to store term frequencies in Spark. Instead, we can construct RDD as a tuple (something similar to a dictionary) to store term frequencies. A tuple will be of the format:

(word, [(doc-1, *freq*), (doc-2, *freq*), (doc-3, *freq*), …])

where each word contains a list and each element in the list is again a tuple of type **(doc. ID, *freq*),** where **doc. ID** is the ID of the Wikipedia article in the which the **word** appears and ***freq*** is the *log-weighted frequency* of the **word** that appears in **doc. ID.** [*log-weighted TF* = $1 + \log_{10} TF$]

Spark will write data types also along with the data, if we write this RDD into a file as it is, which will not be efficient to process when handling user queries. Thus, we will convert the complete tuple into a *string* and then write the string into a file. We will use the following separators to convert tuple(s) into a string:

- **@** - to separate **word** from the list
- **#** - to separate **doc. ID** and ***freq***
- **+** - to separate each tuple in the list

The choice of these separators is arbitrary – so we can use same set of separators to recover all values. Thus, the above RDD should be transformed into:

word@doc-1#*freq*+doc-2#*freq*+doc-3#*freq*+…

Write this RDD to **TF_index** directory.

Step – 3: Query processing

Write a query program (named **TF_query.py**) that can get any number of queries from the user – this means that the program should not terminate after retrieving results of one query. The program should do the following:

1. Up on starting the program, it should load the TF_index as RDD before receiving a query from the user (NOTE: the *string* should be converted back to a dictionary)
2. When the user submits a query, the program should assign an ID (**qID**) to the query and the program should also perform all text pre-processing, given in Step-1, to the query
3. Instead of following the procedure discussed in the class, we will follow a following simple process to get ranks of each document for a given query:
    a. Filter the RDD such that it contains only words that are present in the given query
    b. Once filtered, we will get words and their corresponding term frequencies that are required for query processing. Now transform the RDD so that it will be of format: **(doc. ID, *freq*)**

c. For each doc. ID, sum all ***freq*** values. Now we have single **weight** for each valid **doc. ID**
d. Sort the RDD by value in decreasing order
e. Write **only top 10 results** into a directory named **qID**

Grading Rubric:

1. Logic to remove stop-words, punctuations, and case folding – **5 points**
2. Logic to remove HTML TAGs and URLs – **5 points**
3. Logic to create TF index – **15 points**
4. Successful index creation (no errors and correct naming) – **10 points**
5. Text processing on query - **5 points**
6. Query execution - **5 points**
7. Output (including correct results and naming conventions) – **10 points**
8. Program README and documentation – **5 points**

We will penalize 5 points for each single error (outputs not in the required format, errors/exception, and unsuccessful program execution)**.**

Task – 2: Query after cosine normalization [*40 points*]

This task adds cosine normalization module to the above indexing model. A document vector can be cosine normalized by dividing each component in the vector by Euclidean length (check the tutorial in Canvas for this). Program the following:

1. Copy TF.py (rename it to **CTF.py**) and modify it to calculate Euclidean length of each document. Use Euclidean length of documents to calculate cosine normalized TF values of words. (**NOTE:** This step can be done in multiple ways. The simplest way would be calculating Euclidean length when identifying term frequencies itself. The other way would be to use *Shared Variables* concept. Make a wise decision on what to share use in a *Shared Variable*, if you are using it). Normalize term frequencies with the corresponding Euclidean length
2. This converts the RDD in Task – 1 to:
   (word, ([(doc-1, normalized_*tf*), (doc-2, normalized_*tf*), …]))
   Write this RDD (as instructed in the previous task) to **CTF_index** directory for further query processing
3. Follow the same procedure given in **Task – 1** to add separators and write the output to **CTF_index** directory
4. Write a query program (**named CTF_query.py**), similar to Task – 1. We will follow **ltn** (logarithmic term-frequency, tf-idf, and no normalization) for queries and **lnc** (logarithmic term-frequency, no tf-idf, and cosine normalization) for documents. Your program should handle the following for each query (with **qID**):
   a. Retrieve cosine normalized vector of the query words from **CTF_index**
   b. Construct TF-IDF ($\mathbf{C_t}$) for each word in the query (NOTE: **TF** of a word in the query may **NOT** be always **1**). You can calculate IDF and TF-IDF values from **CTF_index**

c. Perform DOT product on two vectors ($\mathbf{C_t}$ and $\mathbf{CTF_d}$) and sum each component in the vector to get a score for each document **d**
d. Sort the documents in decreasing order of the score and write **only top 10** Doc. IDs to **qID** directory

Grading Rubric:

1. Logic to create cosine normalized TF index – **15 points**
2. Successful index creation (no errors and correct naming) – **5 points**
3. Query execution - **5 points**
4. Output (including correct results and naming conventions) – **10 points**
5. Program README and documentation – **5 points**

Each single error (outputs not in the required format, errors/exception, and unsuccessful program execution) **will be penalized -5 points.**

Bonus task: Query after cosine normalization on TF-IDF index [*10 points*]
Copy CTF.py and rename it to **CTFIDF.py**. Modify the program in such a way that it handles the following:

1. In addition to calculating *log-weighted term frequencies* of each word (**w**) in CTF_index, calculate **Document frequency ($\mathbf{df_w}$)** and **Inverse document frequency ($\mathbf{idf_w}$)** by using **$\mathbf{idf_w}$ =** $log_{10}\frac{N}{1+df_w}$, and get the RDD as triple:
   (word, ([(doc-1, *freq*), (doc-2, *freq*), (doc-3, *freq*), …]), (idf))
2. Now multiply each ***freq*** value with **idf** value to get **TF-IDF** score of each word in each Wikipedia article, which transform the RDD into:
   (word, ([(doc-1, *tf-idf*), (doc-2, *tf-idf*), (doc-3, *tf-idf*), …]))
3. Follow the same procedure given in **Task – 2** to cosine normalize the construct the normalized TFIDF index of format:
   (word, ([(doc-1, weighted_*tf-idf*), (doc-2, weighted_*tf-idf*), …]))
4. Follow the same procedure given in **Task – 1** to add separators and write the output to **CTFIDF_index** directory

Copy the query program from CTF_query.scala/CTF_query.py (rename it to **CTFIDF_query.py**). Modify the program in such a way that it works on **CTFIDF_index** but **NOT ON CTF_index**. All other operations are same. The program should not end until the user terminates it. For each query (with **qID**), the program should give **only top 10** best articles.

Assignment Submission:

Check **Assignment3_mata.pdf** file to get details on what to submit, file formatting and some tips.

This assignment is due on 03/24/2020 at 11:59pm. Any updates about the due date will be announced in Canvas.