Assignment 2
ITCS 6190/8190: Cloud Computing for Data Analysis

Due by 11:59:59pm on Thursday, September 29, 2016

The goal of this programming assignment is to become familiar with Hadoop. We will do this in the context of information retrieval. Please monitor Canvas for example input and output, additional details, and links to resources.

Information retrieval (IR) is concerned with finding material (e.g., documents) of an unstructured nature (usually text) in response to an information need (e.g., a query) from large collections. One approach to identify relevant documents is to compute scores based on the matches between terms in the query and terms in the documents. For example, a document with words such as *ball*, *team*, *score*, *championship* is likely to be about sports. It is helpful to define a weight for each term in a document that can be meaningful for computing such a score. We describe below popular information retrieval metrics such as term frequency, inverse document frequency, and their product, term frequency-inverse document frequency (TF-IDF), that are used to define weights for terms.

Term Frequency:

Term frequency is the number of times a particular word *t* occurs in a document *d*.

TF(t, d) = No. of times *t* appears in document *d*

Since the importance of a word in a document does not necessarily scale linearly with the frequency of its appearance, a common modification is to instead use the logarithm of the raw term frequency.

$WF(t,d) = 1 + \log_{10}(TF(t,d))$ if $TF(t,d) > 0$, and 0 otherwise **(equation#1)**

We will use this logarithmically scaled term frequency in what follows.

Inverse Document Frequency:

The inverse document frequency (IDF) is a measure of how common or rare a term is across all documents in the collection. It is the logarithmically scaled fraction of the documents that contain the word, and is obtained by taking the logarithm of the ratio of the total number of documents to the number of documents containing the term.

$IDF(t) = \log_{10}$ (Total # of documents / # of documents containing term t) **(equation#2)**

Under this IDF formula, terms appearing in all documents are assumed to be stopwords and subsequently assigned IDF=0. We will use the smoothed version of this formula as follows:

$\text{IDF}(t) = \log_{10}(1 + \text{Total \# of documents} / \text{\# of documents containing term } t)$ (equation#3)
Practically, smoothed IDF helps alleviating the out of vocabulary problem (OOV), where it is better to return to the user results rather than nothing even if his query matches every single document in the collection.

TF-IDF:

Term frequency–inverse document frequency (TF-IDF) is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus of documents. It is often used as a weighting factor in information retrieval and text mining.

$\text{TF-IDF}(t, d) = \text{WF}(t,d) * \text{IDF}(t)$ (equation#4)

Your tasks:

1. You should first install and get Hadoop running (in pseudo-distributed mode) on your computer using the installation resources provided on Canvas. You should next run the example WordCount program from Assignment 1 to ensure your Hadoop installation is working properly.

2. You should then modify the WordCount program so it outputs the wordcount for each distinct word in each file. So the output of this DocWordCount program should be of the form '**word#####filename count**', where '#####' serves as a delimiter between **word** and **filename** and **tab** serves as a delimeter between **filename** and **count**. Submit your source code in a file named DocWordCount.java.

3. Make a copy of DocWordCount.java, rename it TermFrequency.java, and modify it to compute the logarithmic Term Frequency **WF(t,d)** **(***equation#1*).

   For example:
   If the output of DocWordCount.java was "**Hadoop#####file1.txt 1000**", the matching output of TermFrequency.java would be "**Hadoop#####file1.txt4**", i.e, it computes the logarithm of the word count value.

   The expected output format of TermFrequency.java is
   > **"word<delimiter>file w"**
   > where *w* is the term frequency of 'word' in **'file'**.

   Explanation:
   > Consider two simple files file1.txt and file2.txt.
   > $ echo "Hadoop is yellow Hadoop" > file1.txt
   > $ echo "yellow Hadoop is an elephant" > file2.txt
   > Running 'DocWordCount.java' and 'TermFrequency.java' on these two files will give an output similar to that below, where ##### is a delimiter.

Output of DocWordCount.java

```
yellow#####file2.txt  1
Hadoop#####file2.txt        1
is#####file2.txt       1
elephant#####file2.txt      1
yellow#####file1.txt  1
Hadoop#####file1.txt        2
is#####file1.txt       1
an#####file2.txt       1
```

Output of TermFrequency.java

```
yellow#####file2.txt  1.0
Hadoop#####file2.txt        1.0
is#####file2.txt       1.0
elephant#####file2.txt      1.0
yellow#####file1.txt  1.0
Hadoop#####file1.txt        1.3010299956639813
is#####file1.txt       1.0
an#####file2.txt       1.0
```

4. Make a copy of the file TermFrequency.java, rename it TFIDF.java, and modify it so it runs two mapreduce jobs, one after another. The first mapreduce job computes the Term Frequency as described above. The second job takes the output files of the first job as input and computes TF-IDF values. The map and reduce phases of the second pass are explained below.

   Map Phase:
   The output files of the Term Frequency step is given as input (each line at a time) to the Mapper for the TF-IDF computation.

   The output key value pairs expected from Map phase are:
   <"yellow ", "file2.txt=1.0"> <"Hadoop", "file2.txt=1.0"> <"is", "file2.txt=1.0"> <"elephant", "file2.txt=1.0"> <"yellow", "file1.txt=1.0"> <"Hadoop", "file1.txt=1.3010299956639813"> <"is", "file1.txt=1.0"> <"an", "file2.txt=1.0">

   ** Each angular brace pair(<>) represents a key-value pair. For example, in <"Hadoop", "file1.txt=1.3010299956639813">, **Hadoop** is a key and **file1.txt=1.3010299956639813** is a value.*

   Reduce Phase:
   Input: The output pairs of Mapper are sorted according to the keys and given to Reducer. The **keys are words.** And for each word, the value (a.k.a **postings list**) is a list of (**file name=WF**) pairs.

```
<"Hadoop", ["file1.txt=1.3010299956639813", "file2.txt=1.0"]>
<"is", ["file1.txt=1.0", "file2.txt=1.0"]>
<"yellow", ["file1.txt=1.0", "file2.txt=1.0"]>
<"an", ["file2.txt=1.0"]>
<"elephant", ["file2.txt=1.0"]>
```

** The above input is not seen, and is only intended to help clarifying the input to the reduce phase).*
** You will need to pass as input the total number of files in the collection to the reducer in order to calculate the IDF. (this is a hint, or you can do it your own way). Just make sure you don't hardcode it in your submission.*

TF-IDF:
The reducer will first calculate the IDF for each word (*equation#3*) by looping on its postings list, the output of this calculation should be as follows: (*this output is not seen, and is only intended to help clarify the IDF results through the Reduce phase, you can use it for debugging*)

```
yellow  0.30102999566
Hadoop          0.30102999566
is      0.30102999566
elephant        0.47712125472
an      0.47712125472
```

Finally TFIDF.java should output for each word/file pair the TF-IDF score (equation#4).

```
is#####file2.txt  0.30102999566
is#####file1.txt  0.30102999566
yellow#####file2.txt      0.30102999566
yellow#####file1.txt      0.30102999566
elephant#####file2.txt  0.47712125472
Hadoop#####file2.txt    0.30102999566
Hadoop#####file1.txt    0.39164905394
an#####file2.txt          0.47712125472
```

**Where ##### is a delimiter. "#####" is used so that it does not match the existing text/words in the files.*

5. To close the loop, you will develop a job which implements a simple batch mode search engine. The job (Search.java) accepts as input a user query and outputs a list of documents with scores that best matches the query (a.k.a **search hits**). The map and reduce phases of this job are explained below.

   User query:

Your program will accept through the command line a space separated user query. You will need to pass the user query to the mapper. Use the same approach you used in step 4 to pass the total # of files in the collection.

Map Phase:
The output files of the TFIDF.java would be the input for the mapper (one line at a time). Recall that each line contains word, filename, and TD-IDF score.

The mapper will first check if one of the user query terms matches the input word in the line, if so, the mapper will output a key-value pair. They **key** is the **filename** and the **value** is the **TD-IDF** score.

E.g., if user query is "yellow Hadoop", the key-value pairs from the mapper would be:
<"file2.txt ", "0.30102999566"> <"file1.txt", "0.30102999566"> <"file2.txt", "0.30102999566"> <"file1.txt", "0.39164905394">

Reduce Phase:
Input: The output pairs of Mapper are sorted according to the keys and given to Reducer. The **key is filename.** And for each filename, the **value is a list of TF-IDF scores**.

<"file1.txt", ["0.30102999566","0.39164905394"]>
<"file2.txt", ["0.30102999566", "0.30102999566"]>

** The above input is not seen, and is only intended to help clarifying the input to the reduce phase).*

Final Scoring:
The reducer will accumulate the scores of each document and output the filename along with its accumulated score.

E.g., if user query is "yellow Hadoop", the key-value pairs from the reducer would be:

```
file2.txt        0.60205999132
file1.txt        0.6926790496
```

** As you notice. the search hits might not be sorted in descending order of scores.*

6. **(BONUS-10%)** Write a ranker job (Rank.java) which ranks the search hits in descending order by their accumulated score, just like Google and Bing do. The job would accept the output of Search.java and output the search hits ranked by scores as follows (in case user query is "yellow Hadoop"):

```
file1.txt        0.6926790496
```

file2.txt	**0.60205999132**

7. Submit the source code for DocWordCount.java, TermFrequency.java, TFIDF.java, Search.java, and Rank.java (**optionally**) along with a README file with execution instructions in a single zip file using the Assignment 2 submission link on Canvas.

Guidelines:
Please make sure your source code is adequately commented, and also make sure each of your files has your name and email id included at the top of the file. Please also include a README file where you provide any execution instructions.

The assignment is due by 11:59:59pm on Thursday, September 29, 2016. Submission will be on Canvas and instructions will be posted on the Assignment web page.

Please monitor Canvas for example input and output, additional details, and links to resources.

Assignments are to be done individually. See course syllabus for late submission policy and academic integrity guidelines.