CPSC5330 – Big Data Analytics
Lab 3: TFIDF (the beginning)

Over the course of the quarter you will build a simple search engine, where you will use Hadoop systems to ingest the documents, and a NoSQL data store to store the indexed documents.

For this first assignment we will do the first part of the document processing: computing a *TFIDF measur*e for the *terms* in the *document corpus*. Our document corpus will be a set of text files; we will use the textcorpora books for this example. You already know about *terms* and *tokens.* This document will explain TFIDF.

Although in real-life search or NLP projects the process of reducing a document to its *tokens* is subtle and complicated, for this assignment we will use the concept of *token* that has already been implemented in the MapReduce code you saw in class: the average word length per letter code converts its documents to a sequence of tokens as follows:

1. Split an input line into words on whitespace
2. For each word, downcase the word and remove all non-letter characters

That's a *token,* and you can further talk about the document as a set of the form (*token*, *count*) where for example in a particular document the token *giraffe* might appear 16 times. More usefully, since we compute these pairs for each document, the tuple really looks like *(document_id, token, count)*

Ultimately we want to compute the relevance between a document and a *query* (a query, like a document, is also a sequence of tokens, though queries are typically shorter than documents).

For *relevance* we will use a simple version of a concept called Term Frequency – Inverse Document Frequency (TF-IDF). This concept is supposed to measure how "important" or "significant" a term is in a document. The TF component says how often the term appears in the document. So if "giraffe" appears in the document frequently, it's likely that the document is somehow "about" giraffes or "relevant to" queries that contain the token giraffe. But you have to balance term frequency against how common the term is in the whole document corpus. Because instead of "giraffe," consider the term "the": that term occurs frequently in every document so the fact that a document has the term "the" in it lots of times might not mean the term is that important or relevant to any query.

Measuring TF-IDF keeps natural language people busy for years and years – there are many many variants. For our purposes we will use these simple definitions:

- We will define the TF of a *term* in a *document* to be the number of occurrences of the *term* in the *document* divided by the number of *terms* in the document. We need to divide by the number of terms because we don't want to reward a document simply for having a lot of terms.

- For IDF the *I* just stands for "inverse" so we will define the *document frequency* of a term as being the percentage of the documents in the corpus the term appears in. So for example, the term "the" might appear in 100% of the documents but "giraffe" might appear in 1% or less. Therefore “the” has high document frequency but low inverse document frequency IDF.

To review: term frequency is a function of a (*term*, *document*) pair – the term 'jesus' probably has higher TF in the King James Bible document than in Alice in Wonderland. Document frequency is a function of a term (only). The document frequency of the term "rabbit" is the percentage of the documents in the corpus containing the term "rabbit."

TF-IDF is also a function of a (*term*, *document*) pair – for this assignment it is the TF of the (*term*, *document*) pair divided by the DF of the *document*. The idea is it balances how frequent the term is in a document against how frequent that same term is in all the documents.

For this assignment you are going to compute TFIDF for every (term, document) pair in the document corpus.

You will produce the TFIDF stream in multiple steps, each a separate MapReduce job.

(In the instructions below I am using the term "stream" in sense of an HDFS directory containing part files, which contain tuples. In MapReduce these appear as a stream of tuples, hence the name.)

- Step 1: doc/term counts. This phase is essentially a word count, but remember you need to convert a word to a *term* using code supplied in class. Second complication is that for simple word count the map/reduce key is just the term, in this case the key is a pair (document_id, term). We will use the basename of the file containing the document as its ID. For example the file containing the text for Shakespere's Hamlet might be in an HDFS file named `/data/textcorpora/shakespere-hamlet.txt`, and we will use `shakespere-hamlet` as the ID for that document. You can get the file path basename for the document in the MapReduce code with the following line:

  ```
  import os
  docid = os.path.splitext(os.path.basename(os.getenv('map_input_file')))[0]
  ```

  In this step, for your MapReduce key, use the string "<doc_id>+<term>" to send to the reducer, so one key might be the string `"shakespere-hamlet+father".` Output for this step is a stream of tuples of the form ("<doc_id>+<term>", count). Sample output for this and the other phases is below.

- Step 2: term count per document. This step reads the documents again and just calculates the number of terms per document. It is essentially the word count pattern except (a) the doc_id is passed to the reducer, and (b) words in the input are converted to terms before being sent to the reducer. Output for this step is a stream of tuples of the form (doc_id, term_count)

- Step 3: split key from Step 1. In Step 1 we needed a composite key with both doc_id and term. However to get TF we need to join the doc/term count (Step 1) with the document count (Step 2) to get TF. So Step 3 will be a map-only job that will simply take the tuples from Step 1 of the form ("<doc_id>+<term>", count) and produce three-tuples of the form (doc_id, term, count)

- Step 4: compute DF. This Map Reduce job will go back to the documents, but this time the mapper will construct tuples of the form (term, doc_id) and the reducer will emit tuples of the form (term, unique_doc_id_count). Hint: the mapper will look the same as mappers you have already seen, but the reducer's job is different. Rather than summing a quantity or taking max from a stream of numbers, it needs to "remember" the set of unique values it has seen for each key. The code for doing so is very short and simple, but it will require some thought.

- Step 5: compute TF-IDF. You have three streams available: from Step 2: (doc_id, term_count), from Step 3: (doc_id, term, count) and from Step 4: (term, unique_doc_id_count). The goal is a stream (doc_id, term, tf-idf). You can compute TF-IDF from these three. You will use Hive to first build three table abstractions over these three streams, then you can compute your desired TF-IDF tuples using a single select. You can then have Hive Create a Table based on this select, and that will make the table appear as a stream in HDFS storage.

You get your final TFIDF values by running these five steps in sequence, then looking in HDFS at the output location of the last step.

HINTS!

Here is some sample output from the first four MapReduce jobs. Your output might be in a different order.

```
# hdfs dfs -cat /data-output/1-term-count/* | head -n 5
austen-emma+a	3073
austen-emma+abbey	23
austen-emma+abbeymill	7
austen-emma+abbeyoh	1
austen-emma+abbots	1
```

```
# hdfs dfs -cat /data-output/2-term-count-document/* | head -n 5
austen-emma	158128
austen-persuasion	83259
austen-sense	118620
austin-persuasion	83259
bible-kjv	790029
```

```
# hdfs dfs -cat /data-output/3-split-doc-term/* | head -n 5
austen-emma	freak	1
austen-emma	free	5
austen-emma	freed	2
austen-emma	freedom	3
austen-emma	freeze	1
```

```
# hdfs dfs -cat /data-output/4-df/* | head -n 5
a	19
aaron	2
aaronites	1
aarons	2
ab	1
```

More hints. The first four streams were generated by MapReduce jobs, so I have four scripts that look like this:

```
# cat step-2-terms-per-doc
#!/bin/bash
./hadoop-streaming 2-term-count-document /data/textcorpora /data-output/2-term-count-document
```

And then I have a directory 2-term-count-document that has files mapper.py and reducer.py.

The fifth stage is different in that it is done by Hive, not Hadoop.

I created a file with Hive commands, the beginning of which looks like this:

```
# head -n 5 step-5-tfidf-hive
DROP TABLE IF EXISTS doc_term_count;
CREATE EXTERNAL TABLE doc_term_count(doc_id STRING, term STRING, count int)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
LOCATION '/data-output/split-doc-term';
```

This Hive script has 4 create table statements like the one above, then the final step is this:

```
DROP TABLE IF EXISTS tfidf;
CREATE TABLE tfidf
ROW FORMAT DELIMITED FIELDS TERMINATED by '\t'
STORED AS TEXTFILELOCATION '/data-output/tfidf'
AS
   select doc_term_count.doc_id, doc_term_count.term,
   1000000 * (count / doc_length) / df.df as tfidf
   from doc_term_count, doc_length, df
   where doc_term_count.doc_id = doc_length.doc_id and
   doc_term_count.term = df.term;
```

You notice I multiplied the real TFIDF value by 10^6 just because the computed TFIDF values were very small. The magnitude of the values won’t make any difference in terms of computing relevance.

I can execute this script just by saying

```
     hive < step-5-tfidf-hive
```

After the script ends, I can view the output like this:

```
# hdfs dfs -cat /data-output/5-tfidf/* | head -n 5
austen-emma     freak   2.1079968970285674
austen-emma     free    1.6642080766015006
austen-emma     freed   1.8068544831673437
austen-emma     freedom 1.5809976727714259
austen-emma     freeze  0.7904988363857128
```