Vector Space Retrieval System

Due date: October 7th

Task Description

Your task is to implement a basic vector space retrieval system using Hadoop. You will use the Cranfield collection to develop and test your system. The Cranfield collection is a standard IR text collection, consisting of 1400 documents from the aerodynamics field, in SGML format. It is available on KSOL in the file cranfield (one document on a line, the first word on the line is the document name). You need to implement the following steps in MapReduce. Note that there is not a unique way to implement the required functionality in MapReduce. You can use one MapReduce job or a sequence of MapReduce jobs per step, or you can combine two steps within one MapReduce job.

1. Implement a preprocessing Java class that transforms the corpus as follows:

    a. Eliminate SGML tags (e.g., <TITLE>, <DOC>, <TEXT>, etc.) – you should only keep the actual title and text (remove authors and references). You can use regular expressions in Java to do some of this, e.g.,

    ```
    String noHTMLString = htmlString.replaceAll("\\<.*?\\>", "");
    ```

    b. Tokenize each document (space and punctuation, use `StringTokenizer`); make the words case-insensitive (e.g., transform all letters to lower case), filter out stopwords (a list of stopwords is available on KSOL, you will have to store them in a `Hashtable`), filter out numbers and stem words using Porter stemmer. You can find a Java implementation of the Porter stemmer at: http://tartarus.org/~martin/PorterStemmer/

2. Implement an inverted index class for a vector space model, as discussed in class (except for the sort by document-id or term frequency, which you don't need to implement). Your inverted index class will have as input the preprocessed corpus obtained in step 1 (which will be stored in the HDFS). [Note that you could also build the inverted index while you preprocess the collection.] The output of this step will be words and their corresponding posting lists. That means, you will need to find

    a. word per document counts (term frequencies) and
    b. total number of documents per word (document frequencies)

    Once you have the inverted index, you will use another MapReduce job to calculate document lengths, assuming the TF-IDF representation for documents. To do this, you will also need:

    c. total number of documents
    d. max term frequency per document
    e. TF-IDF weights for word/doc combinations (which could be calculated ahead of time using another MapReduce job, or on the fly based on the index).

3. To check that the index you've constructed is correct, write a lookup class that takes as input the index (obtained in the previous step) and a word from the vocabulary, and prints out the information in the index, for example:

```
collision:5:(c0169,1),(c0101,1),(c0401,1),(c0447,1),(c1298,1),
```

(I abbreviated the names from "cranfield" to "c" to be able to fit everything on a line.)

or

```
collision (IDF=8.129283016944965) occurs in:

cranfield0101 1 times; |D|=10.613019143881187
cranfield0169 1 times; |D|=9.69299110828261
cranfield0401 1 times; |D|=10.891777182258325
cranfield0447 1 times; |D|=8.821482178534831
cranfield1298 1 times; |D|=13.326035118356959
```

Note that the results above are with stemming.

The lookup class will not use MapReduce, but it will interact with the HFDS to get the index information. [Try to avoid creating the index each time you call this class.]

4. Implement a query answering class that makes use of the inverted index built in step 2. This class does not use MapReduce, but as the lookup class in the previous step, it interacts with the HFDS to get the index information. Use the cosine metric to calculate the similarity between queries and documents and to rank the results. Calculate similarity incrementally as you go through the query terms. Use a hashtable to store partial results.

   When you develop your code, keep in mind that, ideally, a user should be able to query the system multiple times, once the index is created.

   Three query examples are provided below, together with the correct answers (as identified by "experts"). Run your code on these queries. Of course, you'd like your program to retrieve documents with high precision and recall. However, you'd need to incorporate quite a few more features into the system to get the exact correct answers, so don't be disappointed if your answers don't match exactly the correct answers. Some examples of possible answers you may get are also shown. Retrieve 10 documents and report the precision and recall points (also show the 10 documents retrieved). How would the precision/recall points change if you'd retrieve 100 documents?

   Examples of queries:

   - *what is the basic mechanism of the transonic aileron buzz*

     Correct answer: 64, 265, 65, 311, 496

However, your program might answer:

```
cranfield0496     Score:0.599356503074505
cranfield0643     Score:0.27092803909771346
cranfield0903     Score:0.23289463926954437
cranfield0199     Score:0.22771108152343675
cranfield0313     Score:0.15210907549217634
cranfield0660     Score:0.14512247138040954
cranfield0503     Score:0.1449328454793724
cranfield0440     Score:0.1346420956491418
cranfield0038     Score:0.12760239899147507
cranfield0468      Score:0.12458571617357031
```

- *papers on shock-sound wave interaction*

Correct answer: 64, 65, 496

However, your program might answer:

```
cranfield0064     Score:0.5129219210784176
cranfield0170     Score:0.3942509316602841
cranfield0132     Score:0.35093299710276193
cranfield1364     Score:0.3497747124586536
cranfield0256     Score:0.3225784553850413
cranfield0335     Score:0.31563155252128255
cranfield0345     Score:0.3064580438895609
cranfield0402     Score:0.29935598951282844
cranfield0798     Score:0.28197041823082636
cranfield0166    Score:0.28012299378031097
```

- *material properties of photoelastic materials*

Correct answer: 463, 462, 497.

However, your program might answer:

```
cranfield0462     Score:0.3450258491416232
cranfield1097     Score:0.2029090840572237
cranfield0761     Score:0.2026463146327092
cranfield0866     Score:0.16999794829182527
cranfield1025     Score:0.16376994110870036
cranfield1117     Score:0.16269181441783187
cranfield0463     Score:0.15736103176517224
cranfield1096     Score:0.15468591785684058
cranfield0553     Score:0.15354595832289342
cranfield1099    Score:0.15272054879724703
```

Submit: You should submit a zip archive, named **yourname.zip**, that contains the following (please use File Dropbox on KSOL to submit your assignment):

(10 points) A short report that

- Briefly describes the procedure and data structures that you used to implement the vector space retrieval system.
- List the MapReduce jobs in your code. For each MapReduce job, describe the inputs and the outputs (to map and reduce methods, respectively).
- Explains how to run your program in standalone mode and distributed mode (on Beocat), and show proof that your program does what it's supposed to do.
- Show the results for the 3 example queries, plus precision/recall points and discussion of results.
- Acknowledge any code you may have used or any resources that you may have consulted in order to complete your assignment.

(20 points) The source code of your program (including a short readme that explains how your code is organized).

General Recommendations:

Remember that you are not building a production system and the volume of data is relatively small. This suggests the following:

- You can choose data structures, etc. that illustrate the concepts but are straightforward to implement (e.g., you do not need to implement B trees, hashtables are good enough).
- Consider batch loading of data (e.g., no need to provide for incremental update).
- The user interface can be minimal.
- You can take a look at Cloud[9], an open source MapReduce library for Hadoop developed by a group at UMD, to get ideas about how you could implement various classes/methods in Hadoop. The Cloud[9] source code is available at the link below (for your convenience, I also put it on KSOL).

  http://www.umiacs.umd.edu/~jimmylin/Cloud9/docs/index.html

Random MapReduce Notes/Recommendations:

- Remember to remove output files from within your program, so that you don't have to do this by hand, in between different runs. Also, avoid hard-coding any parameters, especially input/output files (that will make it harder to run your code); instead, provide those as command line arguments.

- Remember that local aggregation in MapReduce can be very important with respect to performance, specifically the amount of data moved through the network should be minimal. You could implement combiners, but there are no guarantees that they are going to be executed. A better solution is to perform in-mapper-combining using associative arrays. The Initialize and Close methods we discussed in class are called setup and cleanup, in Hadoop, respectively.

- The posting lookup and query answering steps are meant to be completed outside of MapReduce/Hadoop. You will build the inverted index in Hadoop. If you set the

number of reducers to one, all the postings will be written to a single file. This will make the index easier to manipulate. Obviously, this isn't a scalable solution, but it will suffice for this exercise. Retrieve your index from HDFS (from within your code). Implement your index lookup and query answering engine to manipulate the inverted index stored on your local machine.

- You may want to test your MapReduce classes in standalone mode first (that will make it easier to debug using Eclipse).

- Remember that the Hadoop system divides the (large) input data set into logical "records" and then calls map() once for each record. How much data constitutes a record depends on the input data type. For text files, a record is a single line of text. The input key is the byte-offset of the line within the whole file and the value is the text on the line. Note that we use the byte-offset to refer to a line because the line number is not available to us. Again, large files are broken up into smaller chunks, which are passed to mappers in parallel; these chunks are broken up on the line ends nearest to specific byte boundaries. Since there is no easy correspondence between lines and bytes, a mapper over the second chunk in the file would need to have read all of the first chunk to establish its starting line number – defeating the point of parallel processing! However, given that we know each chunk's size, we can easily find the byte-offset for each chunk.

  To get the current filename, use the following code snippet:

  ```
  FileSplit fileSplit = (FileSplit)reporter.getInputSplit();
  String fileName = fileSplit.getPath().getName();
  ```

- To complete your assignment, you'll have to create and manipulate postings lists, which are complex objects that have their own internal structure. You have two choices to represent postings lists. First, you can encode them as Java strings wrapped inside Hadoop Text objects. The downside is that when manipulating postings, you'll have to do a lot of string-based operations (e.g., splits). This approach will work, but it's pretty ugly. The second approach is to write your own custom Writable. Take a look at some Writables in Cloud9 (edu.umd.cloud9.io package) to see some examples. If you decide to adopt the second option (write your own Writable), this exercise is a good opportunity to learn about different output formats. An OutputFormat (see Hadoop API) describes how output key-value pairs are written to HDFS. By default, Hadoop uses TextOutputFormat, which writes out the key-value pairs in human-readable text format. This is good for you, but can be annoying if you want to further manipulate the output programmatically—since you'll have the read in the text file and parse the key-value pairs back into Java objects (even if you have your own custom Writables). As an alternative, you might want to consider SequenceFileOutputFormat. You can specify that format with a method in JobConf:

  ```
  conf.setOutputFormat(SequenceFileOutputFormat.class);
  ```

  If you do this, the output of your MapReduce job will be stored in one or more SequenceFiles. The advantage of SequenceFiles is that they store key-value pairs in a machine-readable format, i.e., as serialized byte representations of the Java objects (not human readable, but can be programmatically manipulated quite easily).

The SequenceFile API provides methods for reading key-value pairs—saving you the effort of having to manually parse plain text files. Of course, SequenceFiles aren't very useful if you are using Text objects as output values.

Along the same lines, you might also want to take a look at MapFileOutputFormat, which writes the output key-value pairs to... as you've guessed, MapFiles! These files have the additional advantage of supporting random access to the keys. You should learn to use SequenceFiles, but MapFiles are likely more useful for this exercise. See the API for details.

- In several instances you will need to pass one of more files (or objects) to all mappers - for example, you will need to pass the stopwords file (or equivalently, the hashtable containing the stopwords) to all preprocessing mappers. A possible solution is to copy the stopwords.txt to the HDFS before running the job, and then read it into an appropriate data structure in the Mapper's setup method. E.g:

```
/*
 * http://stackoverflow.com/questions/13501276/best-practice-to-pass-copy-of-object-to-
all-mappers-in-hadoop
 * **/
```

When you configure the job:

```
conf.set("job.stopwords.path", args[x]);
```

In the Mapper:

```
public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
private final static IntWritable one = new IntWritable(1);
private Text word = new Text();
private HashMap <String,Object>stopwords = null;

@Override
public void setup(Context context) {
    Configuration conf = context.getConfiguration();
    String location = conf.get("job.stopwords.path");
    if (location != null) {
        BufferedReader br = null;
        try {
            FileSystem fs = FileSystem.get(conf);
            Path path = new Path(location);
            if (fs.exists(path)) {
                stopwords = new HashMap<String, Object>();
                FSDataInputStream fis = fs.open(path);
                br = new BufferedReader(new InputStreamReader(fis));
                String line = null;
                while ((line = br.readLine()) != null && line.trim().length() > 0) {
                    stopwords.put(line, null);
                }
            }
        }
        catch (IOException e) {
            //handle
        }
        finally {
            IOUtils.closeStream(br);
        }
```

```
    }
}
```

Acknowledgments: This assignment is adapted from similar assignments by Mihalcea, Arms, Allan, Lin.