Assignment 1A

Due Friday September 15 2017.

This assignment will introduce you to working with the MapReduce framework in Hadoop.

The biggest goal of this assignment is to clear the basic technical hurdles so we can get down to some real work. See the tech instructions for the course, and instructions for our cluster in particular.

Put some files in HDFS

The files that Hadoop jobs use as input (and produce as output) are stored in the cluster's HDFS (Hadoop Distributed File System). There are a few things you need to know to work with it.

The `hdfs dfs` command *(https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html)* (which is a synonym for `hadoop fs` commands, which you might also see in docs) is used to interact with the file system. By convention, your home directory is `/user/<USERID>/` and you should keep your files in there. For example, on the cluster, either of these commands will list the files in your home directory:

```
hdfs dfs -ls /user/<USERID>
hdfs dfs -ls
```

Create a directory to hold input files for our first job and copy some files into it. There are some files to work with on the gateway node in `/home/bigdata/`. (These are the texts from the NLTK Gutenberg corpus *(http://www.nltk.org/book/ch02.html#gutenberg-corpus)* plus one extra that we'll see later.) It is much easier to separate the files of a data set into their own directory: the easiest way to specify input is "all of the files in this directory."

```
hdfs dfs -mkdir wordcount-2
hdfs dfs -copyFromLocal /home/bigdata/wordcount/* wordcount-2/
hdfs dfs -mkdir wordcount-1
hdfs dfs -cp wordcount-2/a* wordcount-1/
```

This creates a directory `wordcount-2` with the full data set, and `wordcount-1` with a subset that you can run smaller and quicker tasks on while testing. (The full data set here isn't very large, but this is probably a good habit: experiment with small data sets where you can iterate quickly.)

If you'd like to experiment with the same files off the cluster, you can download them: wordcounts-1 *(http://cmpt732.csil.sfu.ca/datasets/wordcount-1.zip)* , wordcounts-2 *(http://cmpt732.csil.sfu.ca/datasets/wordcount-2.zip)* .

Compile a job

The "word count" problem is a common place to start with MapReduce, and we will do that here. The idea is that you have a large collection of text files and would like to count the number of times each word is used in the text.

Start with the provided WordCount code. **Have a look at the code**: it is going to count the number of occurrences of each word in the files (coming to conclusions like "'the' appears 2827 times"). Can you see how that's going to happen?

This needs to be compiled to `.class` files and then combined into a `.jar` file that can be submitted to the cluster. See CompilingHadoop and/or EclipseHadoop.

Build a `.jar` file containing the WordCount class and the two inner classes. Copy the JAR to somewhere in your home directory on the cluster (also described in the instructions above).

Run a job

When we run this job, it takes two arguments on the command line: the directories for input and output files. (Those are handled in the lines above that access `arg[0]` and `arg[1]`.

The command to **submit the job** to the cluster will be like one of these (depending if you have a `package` declared for your class or not):

```
yarn jar wordcount.jar ca.sfu.whatever.WordCount wordcount-1 output-1
yarn jar wordcount.jar WordCount wordcount-1 output-1
```

If all happens to go well, you can **inspect the output** the job created:

```
hdfs dfs -ls output-1
hdfs dfs -cat output-1/part-r-00000 | less
```

And remove it if you want to run again:

```
hdfs dfs -rm -r output-1
```

There was one file created in the output directory because there was one reducer responsible for combining all of the map output (one is the default). We can change the configuration so three reducers run:

```
yarn jar wordcount.jar WordCount -D mapreduce.job.reduces=3 \
    wordcount-1 output-2
```

(or equivalently, could do `job.setNumReduceTasks(3)` in the `run` method.) Re-run the job with three reducers and have a look at the output. [?]

You can also specify zero reducers: in that case, Hadoop will simply dump the output from the mappers. This of this as a chance to debug the intermediate output. Try that:

```
yarn jar wordcount.jar WordCount -D mapreduce.job.reduces=0 \
    wordcount-1 output-3
```

Have a look at this output. Is it what you expected the mapper to be producing? [?]

Modify WordCount

Copy the example above into a new class `WordCountImproved` (in `WordCountImproved.java`). In this class, we will make a few improvements.

Simplify the code

The original code counts using the Java `Integer` values, which are 32-bit integers. Maybe we wanted to count **lots** of words (this is "big data" after all). **Update the mapper** so it produces `(Text, LongWritable)` pairs, so we are working with 64-bit integers. (You don't have to worry about the reducer: we're just about to throw it away.)

It turns out that the `IntSumReducer` reducer in the original `WordCount` class was instructive but unnecessary. Since this is such a common usage, Hadoop includes an `IntSumReducer` that does exactly the same thing, and org.apache.hadoop.mapreduce.lib.reduce.LongSumReducer *(http://cmpt732.csil.sfu.ca/hadoop-javadoc/org/apache/hadoop/mapreduce/lib/reduce/LongSumReducer.html)* that does the same for `Longs`. **Remove the inner class** `IntSumReducer` and use the pre-made `LongSumReducer` instead.

Test your job to make sure it still has the same behaviour. Now we have a bigger problem…

Redefine "words"

Having a closer look at the output, I notice things like this:

```
$ hdfs dfs -cat /user/<USERID>/output-1/part-r-00000 | grep -i "^better"
Better  5
better  255
better' 1
better, 27
better,"        2
better--grown   1
better--or      1
better. 23
better."        7
better.--       2
better.--Now    1
better: 1
better; 9
```

All of these are really instances of the word "better", but with some punctuation making them count as different words (and also instances of "grown", "or", and "now" that weren't counted). This is the fault of the `StringTokenizer` used in the example which just breaks up the line on any whitespace, which apparently isn't quite the right concept of a "word".

And now we open the shockingly-complicated door to international words and characters. First: if you don't have a really good grasp on what the word "Unicode" means, read The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets *(http://www.joelonsoftware.com/articles/Unicode.html)* . Go ahead: we'll wait. [And if you knew those things, try Dark Corners of Unicode *(http://eev.ee/blog/2015/09/12/dark-corners-of-unicode/)* .]

Okay now… it turns out that not everybody speaks English. You might easily decide that words are things that contain characters A–Z and a–z, but these are words that don't meet that definition: "garçon" "hyvä" "Привет".

Lots of languages have lots of rules for word breaks, and I don't know most of them. The solution is to admit ignorance and find a library for the job: Java's built-in BreakIterator *(https://docs.oracle.com/javase/7/docs/api/java/text/BreakIterator.html)* . It's about as easy to use as most Java built-ins but here's a really big hint (lightly adapted from the example in the BreakIterator docs):

```
import java.util.Locale;
import java.text.BreakIterator;

Locale locale = new Locale("en", "ca");
BreakIterator breakiter = BreakIterator.getWordInstance(locale);

int start = breakiter.first();
breakiter.setText(line);
for (int end = breakiter.next();
      end != BreakIterator.DONE;
      start = end, end = breakiter.next()) {
  word = line.substring(start,end).trim();
  if ( word.length() > 0 ) {
    System.out.println(word);
  }
}
```

Update your mapper so that it **ignores punctuation** by using BreakIterator. Make it **ignore case** by applying `.toLowerCase(locale)` to each word. (Languages have different rules for case conversions too: `toLowerCase` knows them.)

At this point, you should get results more like this:

```
$ hdfs dfs -cat output-4/part* | grep -i "^better"
better  338
```

Unicode is Complicated

I snuck a file `a_unicode_text.txt` into our test data. It contains a bunch of "words" prefixed with "zzzz" so they sort to the bottom:

```
$ hdfs dfs -cat output-1/part-* | tail -n 15
zealously       2
zigzags 1
zzzzHeLlo       1
zzzzI   1
zzzzfi  1
zzzzhEllo       1
zzzzöo  1
zzzzẋ̣  1
zzzzẋ̣  1
zzzzöo  1
zzzzПривет      1
zzzzпривет      1
zzzz1       مرحبا
zzzzI   1
zzzzfi  1
```

There are strings there that look the same to me: "zzzzẋ̣" and "zzzzẋ̣" look the same, as do "zzzzi" and "zzzzi", as do "zzzzöo" and "zzzzöo" but they don't even sort beside each other.

The problem is in Unicode normalization: the Unicode code points U+006F U+0308 ("Latin Small Letter O" and "Combining Diaeresis") are conceptually identical to U+00F6 ("Latin Small Letter O with Diaeresis") but compare as different because they are encoded as different bytes.

On the other hand, "zzzzi" and "zzzzi" contain characters that I think are semantically-distinct: U+0069 ("Latin Small Letter I") vs U+2170 ("Small Roman Numeral One").

You can make sure these compare properly by normalizing your strings: converting character sequences with multiple representations to a unique representation. In this case, NFD normal form seems most correct.

Update your code so it **converts your keys into NFD normal form** with java.text.Normalizer *(https://docs.oracle.com/javase/7/docs/api/java/text/Normalizer.html)* . Then after running:

```
$ hdfs dfs -cat output-5/part-* | tail -n 15
youthful        5
z       1
zeal    11
zealous 4
zealously       2
zigzags 1
zzzzfi  1
```

```
zzzzhello       2
zzzzi   1
zzzzöo  2
zzzzẋ.  2
zzzzпривет      2
zzzz1      مرحبا
zzzzi   1
zzzzfi   1
```

Cleanup

We'll ask you to tidy up after yourself for these assignments. There should be plenty of space on the cluster, but as files get bigger, we'd like to keep as few copies as possible lying around:

```
hdfs dfs -rm -r output*
```

Questions

In a file `answers.txt`, answer these questions:

1. Are there any parts of the original `WordCount` that still confuse you? If so, what?
2. How did the output change when you submitted with `-D mapreduce.job.reduces=3`? Why would this be necessary if your job produced large output sets?
3. How was the `-D mapreduce.job.reduces=0` output different?

Development Environment

All of your jobs will eventually run on the cluster, but that's not always the best way to develop code: the turnaround time to upload and submit into the cluster can be uncomfortably long.

Spend some time getting a development environment that you like. That will probably be HadoopStandalone (running as a process on your machine) which is much faster to start and process small data sets than the full cluster.

See also the YourComputer instructions.

Submission

Submit your `WordCountImproved.java` and `answers.txt` to Assignment 1A in CourSys.

Updated Wed Sept. 06 2017, 15:55 by ggbaker.