Homework 9: MapReduce, Hadoop and Spark

Due Wednesday, April 11, 11:59 pm

Worth 15 points

Read this first. A few things to bring to your attention:

1. **Important:** If you have not already done so, please request a Flux Hadoop account. Instructions for doing this can be found on Canvas.
2. Start early! If you run into trouble installing things or importing packages, it's best to find those problems well in advance, not the night before your assignment is due when we cannot help you!
3. **Make sure you back up your work!** I recommend, at a minimum, doing your work in a Dropbox folder or, better yet, using `git`, which is well worth your time and effort to learn.

Instructions on writing and submitting your homework.

Failure to follow these instructions will result in lost points. Much of this homework will involve running code on the Data Science Cluster (Fladoop). As with previous homeworks, you will hand in a .zip archive that includes a jupyter notebook file. You should follow the same naming conventions laid out in previous homeworks. In addition to a jupyter notebook, you will also be creating several files on the Fladoop cluster that we will ask you to include in your homework submission, so make sure you have a way to copy files from the Fladoop cluster to your machine, e.g., using the command `scp`. If you do not know how to do this, please speak with the instructor or your GSI. In cases where we ask you to execute operations on the command line, please copy-paste the commands that you run from your shell to a NBConvert cell in your jupyter notebook. NBConvert cells display text in monospace, so it "looks" like code, but jupyter will not attempt to execute this code.

Refer to the instructions on the course webpage for the usual additional information: `http://www-personal.umich.edu/~klevin/teaching/Winter2018/STATS701/hw_instructions.html`. Please direct any questions to either the instructor or your GSI.

1 Warmup: counting words with `mrjob` (3 points)

In this problem, you'll get a gentle introduction to `mrjob` and running `mrjob` on the Fladoop cluster. I have uploaded a large text file to the Fladoop cluster. Your job is to count how many times each word occurs in this file.

1. Write an `mrjob` job that takes text as input and counts how many times each word occurs in the text. Your script should strip punctuation like full stops, commas and

semicolons, but you may treat hyphens, apostrophes, etc. as you wish. Simplest is to treat, e.g., “John’s” as two words, “John” and “s”, but feel free to do more complicated processing if you wish. Your script should ignore case, so that “Cat” and “cat” are considered the same word. Your output should be a collection of (word,count) pairs.

2. To test your code, I have uploaded a simple text file to the course webpage:

   `http://www-personal.umich.edu/~klevin/teaching/Winter2018/STATS701/simple.txt` .

   Download this file and test your code either on your local machine or on the Fladoop grid. The file is small enough that you should be able to check by hand whether your code is behaving correctly. Save the output of your script on this small file to a file called `simple_word_counts.txt` and include it in your submission. **Note:** use the redirect arrow `>` to send the Hadoop output to a file. This will only send the `stdout` output to the file, while still printing the Hadoop error/status messages to the terminal.

3. Once you are confident in the correctness of your program, run your `mrjob` job on the file

   `hdfs:///var/stat701w18/moby_dick.txt`

   on the Fladoop grid (this file is the Project Gutenberg plain text version of Herman Melville’s novel *Moby Dick*). Note that this file is on `hdfs`, not the local file system, so you’ll have to run your script accordingly. Save the output to a file called `word_counts.txt`, and include it in your submission.

4. Zipf’s law states, roughly, that if one plots word frequency against frequency rank (i.e., most frequent word, second most frequent word, etc.), the resulting line is (approximately) linear on a log-log scale. Using the information in `word_counts.txt`, make a plot of word frequency as a function of word rank on a log-log scale for all words in the file

   `hdfs:///var/stat701w18/moby_dick.txt`

   Give an appropriate title to your plot and include axis labels.

5. How “Zipfian” does the resulting plot look (It suffices for you to state whether or not your plot looks approximately like a line)? You can read more about Zipf’s law and about power laws generally at the respective Wikipedia pages (`https://en.wikipedia.org/wiki/Zipf's_law`, `https://en.wikipedia.org/wiki/Power_law`). For more about power laws, I recommend this survey paper by Mark Newman, a faculty member here at University of Michigan `https://arxiv.org/pdf/cond-mat/0412004.pdf`.

2 Computing Sample Statistics with `mrjob` (6 points)

In this problem, we’ll compile some very basic statistics summarizing a toy dataset. The file

`http://www-personal.umich.edu/~klevin/teaching/Winter2018/STATS701/populations_small.txt` contains a collection of (class,value) pairs, one per line, with each line taking the form `class_label,value`, where `class_label` is a nonnegative integer and `value` is a float. Each pair corresponds to an observation, with the class labels corresponding to different populations, and the values corresponding to some measured quantity.

1. Write a `mrjob` program called `mr_summary_stats.py` that takes as input a sequence of (label,value) pairs like in the file at `http://www-personal.umich.edu/~klevin/teaching/Winter2018/STATS701/populations_small.txt`, and outputs a collection of (label, number of samples, sample mean, sample variance) 4-tuples, in which one 4-tuple appears for each class label in the data, and mean and variance are the *sample* mean and variance, respectively, of all the values for that class label. Thus, if 25 unique class labels are present in the input then your program should output 25 lines, one for each class label. **Note:** I don't care whether you use n or $n-1$ in the denominator of your sample variance formula—just be clear which one you are using. **Note:** you don't need to do any special formatting of the Hadoop output. That is, your output is fine if it consists of lines of the form `label [number,mean,variance]` or similar.

   Think carefully about what your key-value pairs should be here, as well as what your mappers, reducers, etc should be. Should there be more than one step in your job? Sit down with pen and paper first! **Hint:** to compute the sample mean and sample variance of a collection of numbers, it suffices to know their sum, the sum of their squares, and the size of the collection. **Hint:** there are many ways to solve this problem, but you will likely find it useful to use the `lambda` and `reduce` statements in Python.

2. Download the small file at `http://www-personal.umich.edu/~klevin/teaching/Winter2018/STATS701/populations_small.txt`. Run your `mrjob` script on this file, either on your local machine or on Fladoop, and write the output to a file called `summary_small.txt`. Please include this file in your submission. Inspect your program's output and verify that it is behaving as expected.

3. I have uploaded to the Fladoop cluster a much larger data file, located on the HDFS file system at `hdfs:///var/stat701w18/populations_large.txt`. Once you are *sure* that your script is doing what you want, run it on this file. Be sure to use the `-r hadoop` command to tell `mrjob` to run on the Hadoop server rather than on the login node. Save the output to a file called `summary_large.txt`. Download this file and include it in your submission. Don't forget to include in your notebook file a copy-paste of your shell session on Fladoop.

4. Use `matplotlib` and the results in `summary_large.txt` to create a plot displaying 95% confidence intervals for the sample means of the populations given by the class labels in file `hdfs:///var/stat701w18/populations_large.txt`. You will probably want to make something similar to a boxplot for this, but feel free to get creative if you think you have a better way to display the information.

3 Graph Processing: Counting Triangles with `PySpark` (6 points)

A classic task in graph processing is called "triangle counting". If you have never heard of graphs, that's okay! It suffices to know that a graph is a set of *nodes* (also called *vertices*), pairs of which are joined by *edges* (see `https://en.wikipedia.org/wiki/Graph_theory` for more). A *triangle* in graph theory is a set of three nodes, say $\{a, b, c\}$, such that all three nodes are joined by edges. Triangle counting is closely related to a fundamental task for social media companies, who may wish to suggest new "friends" to users based on their existing social network. In this problem, you'll implement triangle counting in the MapReduce framework using PySpark. We should note that in practice, the MapReduce framework is rather poorly-suited to the problem of counting triangles, but it's a good problem to get you practice with the framework, so we'll leave that be.

The input for this problem will be a collection of files representing users' friend lists in a social network. Each user in the network is assigned a numeric ID, and that user's friend list is contained in a file called `n.txt`, where `n` is the user's ID. Each such file contains a single space-separated line, of the form

```
n f1 f2 ...  fK
```

where `n` is the node and `f1,f2,...,fK` are the IDs of the friends of `n`. So, if node 1 is friends with nodes 2,5 and 6, there will be a file `1.txt`, containing only the line `1 2 5 6`. If node 10 has no friends, then there will be a file `10.txt`, containing only the line `10`, or perhaps no file at all. Note that just because an ID appears in a friend list, that doesn't necessarily mean that there will be a file listing that user's friends, but you may assume (1) **symmetry:** if 100 is a friend of 200, then 200 is a friend of 100. (2) **no duplication:** each friend appears in a given friend list at most once (i.e., every file will contain a given number at most once).

Once again, before you dive in and write a bunch of code, sit down and think about the problem. What is the right "fundamental unit" of the problem? What should your keys and values look like? **Hint:** the simplest solution to this problem involves multiple steps, involving a standard map-reduce pattern and a subsequent filtering operation.

1. Write a PySpark job that takes the described input and produces a list of all the triangles in the network, one per line. Each triangle should be listed as a space-separated line `node1 node2 node3`, with the entries sorted numerically. So, if nodes 2, 5 and 15 form a triangle, the output should include the triple (2,5,15), but *not* (2,15,5), (15,2,5), etc.

2. Test your script on the set of 5 simple files in the HDFS directory

   ```
   hdfs:///var/stat701w18/fof/friends.simple
   ```

   which is small enough that you should be able to work out by hand what the correct output is. How many triangles are there? List them in a file called `small_triangle_list.txt` and include it in your submission.

3. Once you are confident that your script is correct, run it on the larger data set, stored on HDFS at `hdfs:///var/stat701w18/fof/friends1000` Save the list of triangles to a file called `big_triangle_list.txt`, and include it in your submission. Don't forget to include in your notebook file a copy-paste of the commands you used to launch your job along with their outputs.