Homework 4: regular expressions using `grep` and distributed computing using Slurm

Due October 30, 11:59 pm

This homework will give you a chance to practice using regular expressions in `grep`, and will introduce you to distributed computing using Slurm. Create a directory `4` on your desktop in which to perform these exercises.

1 Regular expressions in `grep`

The exercises below will give you some practice at using extended grep regular expressions. The file `http://www.greenteapress.com/thinkpython/code/words.txt` contains a list of about 100,000 English words. Download it into the directory `4` with

```
cd ~/Desktop/4
wget http://www.greenteapress.com/thinkpython/code/words.txt
sudo apt install dos2unix # install "dos2unix" program
dos2unix words.txt # Convert DOS/Windows "\r\n" line breaks to Linux "\n"
```

The exercises below will have you write regular expressions for matching certain words in `words.txt`. For each problem, you will turn in two files: one containing your regular expression, called `regexZ.txt`, and another containing the words from `words.txt` that your regex matched, called `matchesZ.txt`, where `Z` is the problem number. For example, for problem 1 below, save your regex in a text file `regex1.txt`, and then use `cat words.txt | egrep -f regex1.txt > matches1.txt` to write the matches from `words.txt` to your matches file. We will check your regular expressions by checking your matches file against a solution file and running `cat words.txt | egrep -f regexZ.txt` to verify that your regular expression produces those matches. You can use this same pattern to check that you have written your regular expression into your submission files correctly.

The file `words.txt` is all lower-case, so you do not need to worry about capitalization in the exercises below. You should not require any `grep` flags aside from `-E` (i.e., equivalent to running `egrep`) and the `-f` flag for reading the pattern from your regex file instead of the command line.

1. Write a regular expression that matches any string containing exactly four consecutive consonants. For the purposes of this **specific** problem, the vowels are `a, e, i, o, u, y`. All other letters are consonants.

2. Write a regular expression that matches any string that contains no instances of the letter `e`.

3. Write a regular expression that matches any string that begins and ends with a vowel and has no vowels in between. For the purposes of this **specific** problem, `y` is neither consonant nor vowel, so consonants are the 20 letters that are not one of `a, e, i, o, u, y` and vowels are `a, e, i, o, u`. The words need not begin and end with the *same* vowel, so `angle` is a valid match.

4. Write a regular expression that matches any string whose last two characters are the first two characters in reverse order. So, for example, your regex should match `repeater` and `stats`, but not `neoprene`. **Hint:** be careful of the cases in which the word is length less or equal to 3. You may handle the case of a single character (e.g., `a`), as you like, but please give a brief explanation for your choice one way or the other in a file called `explanation4.txt`, to be included in your submission.

2 Parallel Computing with Slurm

These exercises will provide a gentle introduction to using the Slurm scheduler on the statistics department high-performance computing cluster. **Note:** you may wish to skip to "What to turn in" below, and turn in an incomplete version of your homework, just in case you run into trouble using the compute cluster. We will grade the last version of your homework submitted before the deadline, so you may, if you like, submit an incomplete version of your work now, and submit a more complete version later.

Reminder: your files must be in a `workspace` directory for your jobs to run correctly!

1. Let's start with a simple exercise. We will write a distributed program for computing the lightest weight three-speed (i.e., three-gear) car in the `mtcars` data set from R.

   (a) The first step is to log onto the cluster. Log on to `lunchbox.stat.wisc.edu` and then run `srun --pty /bin/bash` to begin working on a *compute note* instead of on the login node `lunchbox`. Download `example.sh` to the current directory by running

   ```
   wget http://pages.stat.wisc.edu/~jgillett/605/HPC/examples/5mtcarsPractice/example.sh
   ```

   Read through `example.sh` to make sure you understand it. Try running it and then try changing some small things in the file and see what happens.

   (b) Write a script `getData.sh`, which creates three files, `mtcars1.csv`, `mtcars2.csv`, and `mtcars3.csv`, by running the following code;

```
Rscript -e "write.csv(x=mtcars[ 1:10, ], file='mtcars1.csv'); \
            write.csv(x=mtcars[11:20, ], file='mtcars2.csv'); \
            write.csv(x=mtcars[21:32, ], file='mtcars3.csv');"
```

(The "\" escapes the newline, so that bash treats the three lines of text as a single line of code.) We'll pretend these three tiny files are large– in practice, it would make little sense to use parallel jobs for such a small data set.

(c) Write a script `jobArray.sh` to process the `.csv` file (with header) corresponding to the `SLURM_ARRAY_TASK_ID` given to it. That is, when `${SLURM_ARRAY_TASK_ID}` is `1`, `jobArray.sh` should process `mtcars1.csv` (and similarly for `2` and `3`).

The processing consists of retaining only the weight (`wt`) column and only the rows corresponding to 3-speed cars (gear) and writing these weights to a file called either `out1.csv`, `out2.csv`, or `out3.csv` depending on the value of `${SLURM_ARRAY_TASK_ID}`.

(d) Write a script `findLightest.sh` to combine your three output files (`out1.csv`, `out2.csv`, `out3.csv`) into one stream and write the lightest weight to a file `lightest.txt`. **Hint:** Remember that the `cat` command can take multiple files as command line arguments.

(e) Write a script `mtcars_submit.sh` to use `sbatch` to implement this directed acyclic graph (with data "flowing" from top to bottom):

```
               getData.sh
             /      |      \
            /       |       \
jobArray.sh    jobArray.sh   jobArray.sh
            \       |       /
             \      |      /
              \     |     /
             findLightest.sh
```

Run, test, and debug your code.

(f) You can check your work with this command:

```
Rscript -e 'print(min(mtcars$wt[mtcars$gear == 3]))' # Should be 2.465
```

2. Read `http://stat-computing.org/dataexpo/2009/the-data.html`, which links to and describes data on all U.S. flights in the period 1987-2008. Find out, for departures from Madison, how far you can get in one flight, and what is the average departure delay for each day of the week. To do this, write a program `airlines_submit.sh` and supporting scripts to:

(a) Run 22 parallel jobs, one for each year from 1987 to 2008. The first job should:

i. download the 1987 data via
`wget http://pages.stat.wisc.edu/~jgillett/605/HPC/airlines/1987.csv.bz2`

ii. unzip the 1987 data via `bzip2 -d 1987.csv.bz2`

iii. use a short bash pipeline to extract from `1987.csv` the columns `DayOfWeek`, `DepDelay`, `Origin`, `Dest`, and `Distance`; and retain only the rows whose `Origin` is `MSN` (Madison's airport code); and write a much smaller file, `MSN1987.csv`.

The other 21 jobs should handle the other years analogously.

(b) Collect the Madison data from your 22 `MSN*.csv` files into a single `allMSN.csv` file, and write a set of jobs to answer the following two questions:

- How far can you get from Madison in one flight? Write a line like `MSN,ORD,109` to answer. This line says, "You can fly 109 miles from Madison (MSN) to Chicago (ORD)." But 109 isn't the farthest you can get from Madison in one flight; write the correct line. (Hint: I used a `bash` pipeline to do this.) Save the result in `farthest.txt`.
- What is the average departure delay for each day of the week? Write a pair of lines like these (the whitespaces between the columns should be tabs) to a file `delays.txt`:

```
Mo  Tu  We  Th  Fr  Sa  Su
8.3 5.0 4.3 5.5 9.5 2.1 3.5
```

  Of course, these are not the correct numbers. It's up to you to derive the correct ones. **Hint:** I used R's `tapply()` to do this.

What to turn in

On your VM, create a directory `NetID_hw4`, where `NetID` is your NetID. Create a subdirectory `NetID_hw4/mtcars`. Copy into that subdirectory the following files (see "Retrieving files with `scp`" below for instructions on how to copy files from the cluster to your local machine):

- `getData.sh`
- `jobArray.sh`
- `findLightest.sh`
- `mtcars_submit.sh`
- `lightest.txt`

Next, create a subdirectory `NetID_hw4/airlines`. Copy into that subdirectory the following files:

- `airlines_submit.sh`
- `farthest.txt`

- `delays.txt`
- Any other supporting files required to run your script.

We should be able to recreate your `farthest.txt` and `delays.txt` files by running `airlines_submit.sh`.

Finally, create a file `README` in the directory `NetID_hw4` with a line `NetID,FamilyName,GivenName`, where `NetID` is your `NetID`, `FamilyName` is your family name, and so on. If you collaborated with any other students on this homework, add additional lines of this form, one for each of your collaborators. So, for example, if George Box, with NetID `gepbox` worked with John Bardeen with NetID `jbardeen`, George's `README` file should look like

```
gepbox,Box,George
jbardeen,Bardeen,John
```

From the parent directory of `NetID_hw4`, run

```
tar cvf NetID_hw4.tar NetID_hw4
```

and upload `NetID_hw4.tar` as your HW4 submission on Canvas.

You can verify that you compressed your submission (i.e., created the `.tar` file) correctly by downloading your submission file from Canvas, and then

1. Create a directory to test in, say, `mkdir test_HW4`
2. Move your downloaded `.tar` file into that test directory, and `cd` into that test directory.
3. Extract the `.tar` file with `tar xvf NetID_hw4.tar`. This will create a new directory, which should be called `NetID_hw4` (where `NetID` is your NetID).
4. List the contents to make sure all your files are there: `ls NetID_hw4`.

Retrieving files with `scp`

When we work with compute clusters or other remote resources, we need tools for moving files back and forth from our local machine to the cluster. `scp` ("secure copy") is the most common tool for the job (though see also `rsync` for another commonly-used copy program). The basic form of a call to `scp` is just like the command line program `cp`: `scp source target` copies the file `source` to a file called `target`. What makes `scp` special is that `source` or `target` may be on another machine entirely. For example, to copy a file from `lunchbox.stat.wisc.edu` to your local machine, run the command

```
scp username@lunchbox.stat.wisc.edu:/path/to/the/file.txt path/to/save/in.txt
```

on your *local* machine. On the other hand, to copy the file `foo.txt` from your local machine to `lunchbox.stat.wisc.edu`, run the command

```
scp foo.txt username@lunchbox.stat.wisc.edu:/path/to/save/foo.txt
```

again on your local machine. See `man scp` for more information.