Week 10 Laboratory Activity

There are two parts in this week's lab. We will review some of the BASH (UNIX) commands (repeat some of the last week's lab activities), and then we will investigate basic tools for managing and processing large quantities of data by simulating the Hadoop's Map-Reduce paradigm. Note that the latter part is just a simulation that is meant as an introduction for theoretical understanding.

- Activity 1: Manipulating Large Files with Shell Commands
- Activity 2: Understanding Map-Reduce (Hadoop)

Activity 1: Manipulating large files with shell commands

Last week, you were briefly introduced to BASH. We will further review what you have done last week in this part of the tutorials (but using a different file for practice purposes). The benefit of using simple command line tools available in the BASH shell is to allow us to manipulate large data files without needing to ever load them fully into memory.

Start with your BASH shell (Windows users: Type 'bash' and open it as 'administrator', or 'Cygwin terminal' if you are already using it; MacOs user: MacOS terminal, Linux users: Linux terminal). You should be familiar with the following commands:

```
ls
pwd
cd ..
cd
```

Please ensure that you know what each of the commands above does. Download the bundle `tutorial_data.tar.gz` from Moodle and then copy it to your home (or anywhere you prefer) directory, for example:

```
cp /cygdrive/c/Users/Sicily/Downloads/tutorial_data.tar.gz .
```

The dot "`.`" notation is shorthand for the current directory, and since we didn't specify a different name for the file, it will be copied with the same name. Decompress the `gz` file to have a look at the contents. You can do that from the command line using:

```
tar -zxvf tutorial_data.tar.gz
```

Although the file is compressed using `gzip`, which you can decompress using `gunzip`, the `tar` command is introduced here (`man tar` for more information) which works in conjunction with the `gzip`/`gunzip` commands. You can also decompress it in two steps using

```
gunzip tutorial_data.tar.gz
tar -xvf tutorial_data.tar
```

And then change directory to the `books` sub-directory in the `tutorial_10` directory:

```
cd tutorial_10/books
```

We'll now have a look at "`less`" (again), which is a very simple program for viewing (but not editing) text files.

```
less book1.txt
```

Use the following commands to do basic navigation in file:

```
[up/down] - move one line the file
[space]   - move down a whole page
q - to quit
```

If you ever get into trouble while using less, just hit `[Control]+c` to kill the program.

Here are some other basic commands of the `less` commands (repeat from last week's tutorial) that allow you to skip to the end of a file or search for a string in the file.

```
[shift]+g - skip to end of file
/keyword  - search for the first occurrence of "keyword" within the file
/         - find the next occurrence of the keyword
```

Practice 1: What book is it? In which chapter did Alice go to a Mad Tea-Party? Have a look at the other books in the folder. What are their titles?

We can find out quickly what their titles are by running a "`grep`" command which checks each line of a file to see if it matches a particular pattern and returns the line if it does. Try the following two commands:

```
grep "Title" book2.txt
grep "Title" book*.txt
```

In the second command the asterisk "`*`" is a wild-card and matches against all the files `book1.txt` through to `book10.txt`, and the results are the same as if we called `grep` on each of those files in turn.

The third book is really long. It's Tolstoy's "War and Peace". Use the word count command "`wc`" to find out how long it is:

```
wc book3.txt
```

The command should return three numbers. Can you work out what the numbers mean (*mentioned in last week's tutorial*)? Google "wc unix" to check.

Another command line tool that is very useful is "`cat`", which does nothing more than load a file and output it to the terminal:

```
cat book1.txt
```

The `cat` command becomes particularly useful when combined with a Unix pipe operator "`|`". A pipe is used to connect programs using the syntax:

```
program1 | program2
```

All the pipe operator does is redirect the output of one program to be the input of another program. In the case above, it grabs the text that "`program1`" tries to print out to the screen and instead, passes it as input to "`program2`" (as though the user had typed it all in on the keyboard). Most shell commands are designed to be able to take input from a pipe and output to a pipe. We call them pipes, because they allow us to control the flow of data, just like a real pipe controls the flow of water.

The reason pipes are interesting to us when dealing with big files in data science applications is they allow us to work on data as a stream, rather than needing to load the data into memory in order to view or use it. To understand what this means, let's have a look at the rather large file. First change to the "`data`" directory and show its contents:

```
cd ../data
ls
```

It should contain a 12MB file called "`hourly_44201_2014-06.csv.gz`". Don't decompress the file. We'll do that on the pipe using the "`gunzip`" application:

```
cat hourly_44201_2014-06.csv.gz | gunzip | less
```

Notice how fast it loads! The reason is that only the start of the file is being accessed. The display only shows the start of the file and doesn't need to wait until the whole CSV file is loaded in order to display content. In fact, the whole CSV file isn't ever loaded if you quit the "`less`" program without ever scrolling to the end of the file!

The pipe is buffered (with a buffer size of usually around 64KB), which means that each program on the pipe will only produce new output (the next 64KB of data) when the previous output (on the buffer) has been emptied by the subsequent program. So, unless any of the programs needs to load all the data into memory for some reason, the overall command line will execute without ever loading the data all into memory all at once. Each subsequent program on the pipe simply accesses the data it needs as and when it is ready to process it.

We don't need to finish the pipe with the "`less`" command. If we just wanted to know how many lines there were in a file, we could instead pipe the results to the word count "`wc`" command:

```
cat hourly_44201_2014-06.csv.gz | gunzip | wc
```

It takes much longer to execute this command line than the previous one (that ended with less).

Practice 2: Why do you think that it takes longer?

If we would like to see just the start or end of the file (without loading it interactively in less) we could use the "`head`" and "`tail`" commands, similar to R:

```
cat hourly_44201_2014-06.csv.gz | gunzip | head
cat hourly_44201_2014-06.csv.gz | gunzip | tail
```

Notice how much longer the command line with tail takes to execute than does the one with head!

If all we are wanting to do with the file is extract the first say 1000 lines of the file, then we could use the head command to do this by appending the flag `-n1000`. We could then save these lines to a file rather than printing them out on the screen by doing a redirect with the "`>`" symbol:

```
cat hourly_44201_2014-06.csv.gz | gunzip | head -n1000 > first_1000_lines.txt
```

A pipe "`|`" passes data to another program, while a redirect "`>`" saves data to a file.

A very handy tool is "`awk`" which is a program for processing a text file one line at a time. When using awk all we need to do is

1) specify the delimiter it should use to break up each line (into columns), and
2) tell it what to do with each line.

For example, we can use awk to select a subset of the columns as follows:

```
cat hourly_44201_2014-06.csv.gz | gunzip | awk -F',' '{print $6,$7,$14}' | less
```

Here we have used the flag `-F','` to tell `awk` that the columns are separated by commas, not whitespace (which it assumes by default). And we have passed it the instruction `{print $6,$7,$14}`, which tells it to print the value in columns 6, 7 and 14 for each line of input it sees.

If we are happy with the new table we have produced, we could save it to a file, but imagine we would only like to save a small sample of the file, say 500 lines starting from line 1001. We can tell awk to apply the print command only to those lines as follows (some of the commands below are on a single line, it's on two lines because of the limited space on this page):

```
cat hourly_44201_2014-06.csv.gz | gunzip | awk -F',' 'NR>1000 && NR<=1500
{print $6,$7,$14}' > 3columns_500rows.txt
```

Often, we'd like a random sample of the data. Again, this is a very simple `awk` task. If we want to sample 1% of the rows, we can use the random number function "`rand()`" to produce a value in the range 0 to 1 and only output a line if the value is less than 1/100:

```
cat hourly_44201_2014-06.csv.gz | gunzip | awk -F',' 'rand()<1/100 {print
$6,$7,$14}' | less
```

We can also select only rows that have a particular value in one of the columns. For instance, we could select measurements around the Los Angeles area (34.0522, -118.2437) by restricting the latitude and longitude coordinates, as follows:

```
cat hourly_44201_2014-06.csv.gz | gunzip | awk -F',' '$6>=34 && $6<=34.5 &&
$7>=-118.5 && $7<=-118 {print $6,$7,$14}' | less
```

There are many other programs available in the BASH shell. Particularly useful is the "`sort`" command, which can sort a column of values either lexicographically (/alphabetically) or numerically (by using the `-n` flag):

```
cat hourly_44201_2014-06.csv.gz | gunzip | awk -F',' '{print $14}' | sort -
n | less
```

Be patient! Notice how long it takes for the values to appear? That's because the whole file has to be loaded into memory before they can be sorted. The sort operation itself can also take some time if the list of numbers is very large.

If you have multiple columns in your output, you can tell the sort command to sort on a particular column by using the `-k` flag, (sort `-k2,2` will sort based on the second column, while sort `-k2,4` will sort based on column 2 then 3 then 4). You can find out more about any unix command by either googling it or opening its man page:

```
man sort
```

The `man` page has the same controls as `less`.

Now let's have a look at a much bigger file: `hourly_WIND_2015.csv.gz`, which contains wind data readings across the US for all of 2015. It is 87MB compressed and 2.5GB uncompressed.

This file is compressed using the `gzip` format, so again we'll use the `gunzip` command to decompress it, but instead of generating a 2.5GB file, we'll output the data to a pipe. We tell the `gunzip` program to do this by using the "`-c`" flag. We'll then take the output and view it in `less`.

```
gunzip -c hourly_WIND_2015.csv.gz | less
```

Practice 3: How many lines does this file contain? Use `wc` to find out without decompressing the whole file.

Let's extract the wind measurements ("`Sample Measurement`" column) for the state of California (i.e. where the column "`State Name`" has the value "`California`") and save them to a text file. Note the use of the double quotes to wrap the string "California" inside the `awk` command.

```
gunzip -c hourly_WIND_2015.csv.gz | awk -F',' '$22=="\"California\"" ||
NR==1 {print $6,$7,$14}' > california_wind.txt
```

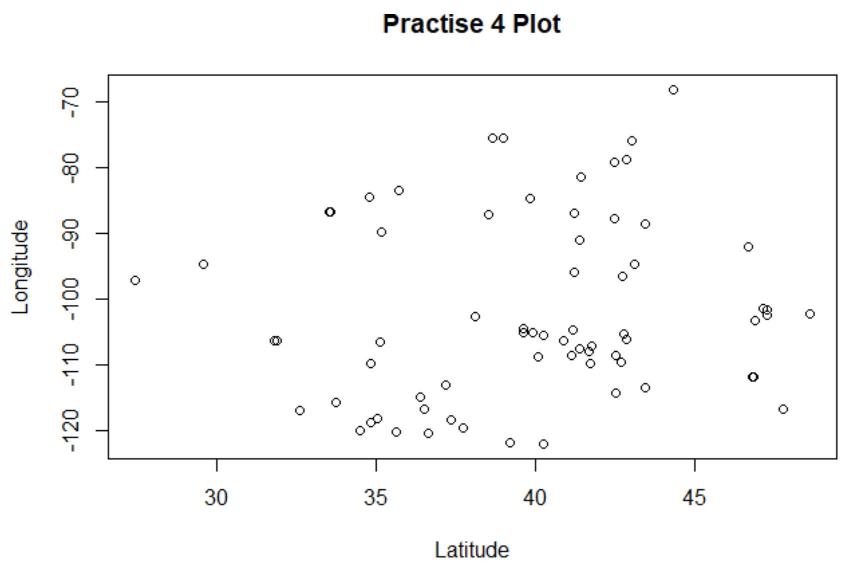
This could take a while! The file you are creating is quite big. *Practise 4: What does the part "`|| NR==1`" in the `awk` command do? Why did we run that?*

Now, we want to move the file back to where our R can read and process it. Let's load and visualise it with R.

```
df <- read.table('california_wind.txt', header = TRUE)
head(df)
tail(df)
summary(df$Sample.Measurement)
```

Practice 5: From the file `hourly_WIND_2015.csv.gz`, extract the latitude and longitude values for locations with extremely high wind measurements, say greater than 30 Knots (about 55 km/h), into a new file (hint: you should actually only get approximately 1,500 data points (rows)). Return to R, load in the data and plot the latitude and longitude values against each other as a scatter plot to see the locations where the extreme measurements were recorded.

Your resultant plot should look similar to:



Activity 2: Understanding Map-Reduce (i.e. Hadoop)

We will look at very simple examples of Map and Reduce operations that can be used within the Hadoop framework for distributed processing of big data. This part of the tutorial makes use of code from Michael Noll's blog post Writing an Hadoop Map Reduce Program in Python.

Change to the scripts directory and have a look at the two files (written by Michael Noll):

```
cd ../scripts
ls
less mapper.py
less reducer.py
```

These are Python scripts which can be executed directly from the shell!

The first line of each script is "`#!/usr/bin/env python`" tells the shell interpreter that Python code will follow. The script is written to take data from standard input (either the keyboard or more likely a pipe) and output text to standard output (the screen, or a pipe).

We will now execute these scripts as programs in the shell. Before we can do that we need to tell the filesystem that these files are executable files (scripts), as opposed to standard read-only or read-write files. To do this type:

```
chmod a+x mapper.py
```

Once they are made executable you can execute the commands by prefixing them with "`./`", which just tells the interpreter where to find the executable (i.e. in the current directory). Try running the mapper code.

```
./mapper.py
```

What output does the program then produce? It will not do anything, but simply wait for input from you. Type a few sentences and then hit [Control]+d, to signal the "end of the input file":

Let's apply the program to a bigger input file. We'll use `book1.txt` from the books directory:

```
cat ../books/book1.txt | ./mapper.py | less
```

The output is just the set of terms in the order they appear in the text, with each word (followed by a 1) on a different line. We can then sort the lines by the terms in order to group the same terms together:

```
cat ../books/book1.txt | ./mapper.py | sort -k1,1 | less
```

After which we can use the `reducer.py` script to count up the number of consecutive instances of each word.

```
cat ../books/book1.txt | ./mapper.py | sort -k1,1 | ./reducer.py | less
```

So here we have three consecutive operations. A **Map** step, which maps an input sequence (long vector) of terms into an output sequence (another long vector), a **Sort** step, which sorts the sequence, and a **Reduce** step, which takes the sequence (long vector) and creates a shorter sequence (short vector) from it. When using Hadoop to parallelise computation, one needs to define exactly those types of steps.

Now we could apply this sequence to all of the files in the books directory, using the asterisk wild-card notation to get cat to concatenate all of the documents together:

```
cat ../books/book*.txt | ./mapper.py | sort -k1,1 | ./reducer.py | less
```

But the mapper operation doesn't need to see all of the documents in the directory in order. It can be run each document or indeed any part of a given document separately!

So instead of running one Map program on the files sequentially, we could apply the Map program in parallel on each of the books in the directory, producing an intermediate output for each. Copy the following 10 commands into the command terminal all at once:

```
cat ../books/book1.txt | ./mapper.py > intermediate1.txt &
cat ../books/book2.txt | ./mapper.py > intermediate2.txt &
cat ../books/book3.txt | ./mapper.py > intermediate3.txt &
cat ../books/book4.txt | ./mapper.py > intermediate4.txt &
cat ../books/book5.txt | ./mapper.py > intermediate5.txt &
cat ../books/book6.txt | ./mapper.py > intermediate6.txt &
cat ../books/book7.txt | ./mapper.py > intermediate7.txt &
cat ../books/book8.txt | ./mapper.py > intermediate8.txt &
cat ../books/book9.txt | ./mapper.py > intermediate9.txt &
cat ../books/book10.txt | ./mapper.py > intermediate10.txt &
```

The ampersand '`&`' character at the end of each line tells the shell to spawn a new process to execute the command so that we run each command line in parallel, rather than one after the other.

In this case, the commands all execute very quickly so it's hard to notice that they are running in parallel, but imagine you had millions of such files. Executing these individual Map operations in parallel, across thousands of machines (by dividing up the corpus of documents across them), allows us to scale the computation to massive quantities of data.

Having performed the Map step on the individual pieces of data, we can now perform the Reduce step, which combines the intermediate outputs together. In this case, we will simply concatenate the intermediate results back together, before running the sort and reduce commands:

```
cat intermediate*.txt | sort -k1,1 | ./reducer.py | less
```

Are the results the same as before? You can check by creating an output file for each and comparing them using the "`diff`" command. (Will leave it to you to find out how "`diff`" works).

So what is the take-home message?

- In the Map-Reduce paradigm, the data is divided up across many servers and local computation (called a Map step) is performed at each node to process the data, then aggregation operations (the Reduce step) are performed to combine the data generated at the various nodes of the network. This allows for massive quantities of data to be processed.

- The Hadoop framework provides the functionality to do the above-mentioned parallelisation of execution across a cluster of computers, by providing (i) a common file system across the machines, and (ii) a MapReduce job scheduler which allocates data and Map or Reduce jobs to different nodes on the network.

- In order to use Hadoop, the user needs to define Map and Reduce tasks, which look exactly like the Python scripts discussed in the example above. Indeed, you can read about how to use these very Python scripts with a Hadoop implementation on Michael Noll's blog: Writing an Hadoop Map Reduce Program in Python.

- Finally, note that in many/most MapReduce scenarios, not only the Map step but also the Reduce step is executed in parallel. In the typical use case of building a search engine, the Map step combines the three operations discussed above (to produce term counts for each document separately) and the Reduce step involves building posting lists for individual terms, which are lists identifiers for documents that contain the term. The posting lists for different terms can be computed independently.