Michael Bissell

STA 250
Advanced Statistical Computing

Final Project
Adventures in RHadoop

1. Introduction

In assignment #2 of the STA 250: Advanced Statistical Computing course, we had an introduction to the "Bag of Little Bootstraps" algorithm. Similar to the original concept of the bootstrap and other resampling methods, the core idea is that if we can generate enough random samples from our dataset, i.e. subsets, and compute values of our estimator(s), then we can approximate the sampling distribution of our estimators and in doing so, we can also estimate the standard error(s) of our estimator(s). This gives us a quality measure for our estimate(s).

The secret sauce that the Bag of Little Bootstraps brings to these existing and well known methods is that it is "embarrassingly parallel" in design and thus very amenable to large scale datasets where the original bootstrap may suffer from computational bottlenecks. In particular, the Bag of Little Bootstraps in our regression example can be programmed in such a way that the desired number of iterations used in the final computation of each estimate can be run completely in parallel and is independent of each other estimate, then simply combined in the final consolidation step. Whereas the original bootstrap method samples from the full data a large number of times, the Bag of Little Boostraps algorithm first subsets the full dataset in smaller chunks, each of which is then bootstrap resampled to get parameter estimates, and finally all estimates are then consolidated. This gives us and approximation to the sampling distribution of the estimates along with approximations of the standard errors. This makes the Bag of Little Bootstraps much more computationally efficient than the original bootstrap method and thus is extremely useful for very large datasets.

This report seeks to extend the work we did in that assignment by running some empirical studies on sensitivity of the estimates and standard errors to various choices in the number of subsets, the number of bootstrap samples, and the number of elements in each subset, namely, r, s, and γ in our algorithm, and seeks to replicate results found in the Kleiner, et. all paper, A Scalable Bootstrap for Massive Data (2011) as well as in Efron and Tibshirani, An Introduction to the Bootstrap (1993). In addition, we seek to frame these tradeoffs with respect to runtimes.

Lastly, we will take a quick tour with an experimental R package called 'segue' that allows one to connect their R session to Amazon Web Services Elastic MapReduce to run R scripts. We will just get our feet wet with a few basic examples to get up and running, document how we achieved this, and leave more complicated use cases for another time.

2. Methodology

We begin with a discussion of the algorithm construction within the context of our original regression problem. Our goal was to fit an ensemble of simple linear regressions on d=1,000 covariates with no intercept term, where the errors are distributed as $N(0, \sigma^2)$. We then use this as an approximation of the sampling distrubtion of $\widehat{\beta}$ and use the estimates returned to compute approximations for the standard errors of $\widehat{\beta_1}, \ldots, \widehat{\beta_{1000}}$.

Our workflow was to first subset the original full dataset into $s = 5$ distinct subsamples of size $b = n^{\gamma} = 1,000,000^{0.7} = 15849$ *without* replacement and selected from the full dataset of n=1,000,000 rows. Then for each of those distinct subsamples/subsets, we resampled $r = 50$ bootstrap subsamples n=1,000,000 times from the distinct subset *with* replacement, to ensure that we get a representative sample from the bootstrap. Each bootstrap subsample is selected using a multinomial probability to distribute n balls into b boxes where each box has equal probability, i.e. $1/b$. This gives us the weights (i.e. number of replications) to put on each of the b bootstrapped row indices in our subsample so that we get a bootstrap sample of size n. We then pass in just the unique bootstrap sample rows and the multinomial counts for each of those rows, i.e. weights, which dramatically improves the computational speed of the regression fitting procedure, lm() in R due to smaller matrix solves. This computational gain combined with the fact that we can run many regressions on smaller subsets in parallel is where the big computational gains are. The linear regressions computed using the lm() function return the

1,000 coefficient estimates for the 250 jobs, thus allowing us to compute standard errors for each of the $\widehat{\beta_1}, \ldots, \widehat{\beta_{1000}}$. As a reference point, the 250 jobs run in parallel with s=5, r=50 and $\gamma = 0.70$ took less than 5 minutes to run and produced standard errors of approximately 0.01, as expected.

To ensure the implementation was correctly using the $s = 5$ distinct subsets, each of which spawned $r = 50$ bootstrap samples, we pass around the random number generator seed to each of the 250 jobs. We use mod 5 on the job number to determine which of the s_index datasets the current job should be assigned to and then assign the seed as a function of the s_index making sure there is no overlap with the other s_index jobs in getting the distinct subsample of size b. Then we assign the r_index as $ceil(job\_num/r)$ allowing us to reset the seed as a function of the s_index and the r_index to ensure the correct s_index subsample has r_index distinct bootstrap samples.

This report aims to address the question of how should one choose the values of r, s and γ so that we can have a well-balanced tradeoff between computational simplicity, speed, and accuracy in our estimates. Because we start with a regression problem with which we already know the answer and since the standard errors of the β coefficients are known to be approximately 0.01 for this simulated dataset, we start with that as a very simplistic baseline from which we can empirically judge performance for other values of r, s, and γ. We start with investigating the choice of γ for fixed values of s=5 and r=50 by running the Bag of Little Bootstraps algorithm over $\gamma = 0.50, 0.55, 0.60, 0.65, 0.70, 0.75, 0.80, 0.85, 0.90$. We then examine plots of the standard errors of the β coefficients as well as look at metrics such as the interquartile range of the standard errors and the variance of the standard errors, again because we simplistically assume they are all approximately 0.01. Similarly, we examine choices of r and s by fixing $\gamma = 0.70$ and allowing pairs of r and s such that $r \times s = 250$, i.e. (5,50), (10,25), (25,10), and (50,5). Finally, we must make these decisions taking into consideration total runtime for different sets of parameters. Our methodology with respect to timing is to time a single process with fixed values of s=5 and r=50 while letting γ range over values 0.50 to 0.90 since this algorithm is inherently parallel and thus with an uncongested cluster with an appropriate numbers of cores could run many of these jobs and thus for timing purposes it would be best to time a single job so as not to have results influence by other jobs.

3. Results

As mentioned above, we investigate the choice of γ by using fixed values of s=5 and r=50, then run the Bag of Little Bootstraps algorithm over $\gamma = 0.50, 0.55, 0.60, 0.65, 0.70, 0.75, 0.80, 0.85, 0.90$. Again, we start with the simplistic assumption that the correct standard errors of $\widehat{\beta}$ are all approximately 0.01 and thus we look at summary statistics of the full set of $\widehat{\beta}$'s. The first thing we see in Table 1 below is that the summary statistics for $\gamma = 0.50$ look suspicious but results from the fact that taking a bootstrap sample of size n=1,000,000 from such a small subset of b=1,000 and doing this 250 times means that we will get virtually no variability in our estimates. Next, we notice that the interquartile range of the standard errors and the standard deviation of the standard errors begins declining as we increase γ from 0.55 to 0.75 and then begins to increase slightly for values of γ between 0.80 and 0.90. This would suggest that there is some sweet spot in the trade off between a tighter IQR and computational demands from the data size somewhere around $\gamma = 0.70$ range. However, we do point out that this difference is in the 5th decimal place and would be very hard to distinguish as statistically different. We see the same basic pattern with the standard deviation. We then look at the IQR for each value of γ relative to $\gamma = 0.70$ and we can see that there is approximately a 6 % reduction in IQR from when $\gamma = 0.55$ to $\gamma = 0.70$ with a potential gain of another 2% gain going to $\gamma = 0.75$. Again, we see the same basic pattern with the standard deviation.

Empirical Results for γ									
Gamma	0.50	0.55	0.60	0.65	0.70	0.75	0.80	0.85	0.90
b = n^{γ}	1000	1995	3981	7943	15849	31623	63096	125893	251189
Min	3.44E-13	0.00860	0.00855	0.00864	0.00839	0.00872	0.00867	0.00824	0.00828
Q1	8.50E-13	0.00966	0.00959	0.00967	0.00964	0.00964	0.00962	0.00961	0.00976
Med	1.12E-12	0.00997	0.00992	0.00997	0.00993	0.00995	0.00993	0.00990	0.01006
Mean	1.21E-12	0.00998	0.00992	0.00998	0.00994	0.00995	0.00994	0.00991	0.01008
Q3	1.51E-12	0.01031	0.01022	0.01029	0.01025	0.01024	0.01025	0.01022	0.01040
Max	2.91E-12	0.01135	0.01161	0.01152	0.01140	0.01141	0.01144	0.01172	0.01155
SD	4.73E-13	0.00048	0.00047	0.00046	0.00046	0.00046	0.00045	0.00047	0.00048
IQR	6.55E-13	0.00065	0.00063	0.00062	0.00061	0.00060	0.00063	0.00061	0.00064
IQR relative to $\gamma = 0.70$	1.07E-09	1.06360	1.02610	1.01140	1.00000	0.98530	1.02610	1.00330	1.05060
SD relative to $\gamma = 0.70$	1.74E-06	1.04421	1.02274	1.00789	1.00000	1.01289	0.98340	1.02601	1.04973
Runtime (sec)	4.185	7.71	12.79	25.09	47.01	47.09	176.04	412.19	856.89

Table 1: Summary statistics for various values of γ and fixed values of r and s

In the figures shown below, we see index plots of the standard errors for various values of γ along with dashed lines at the 5th and 95th percentiles. Again, it is very difficult to discern a practical difference in these plots.

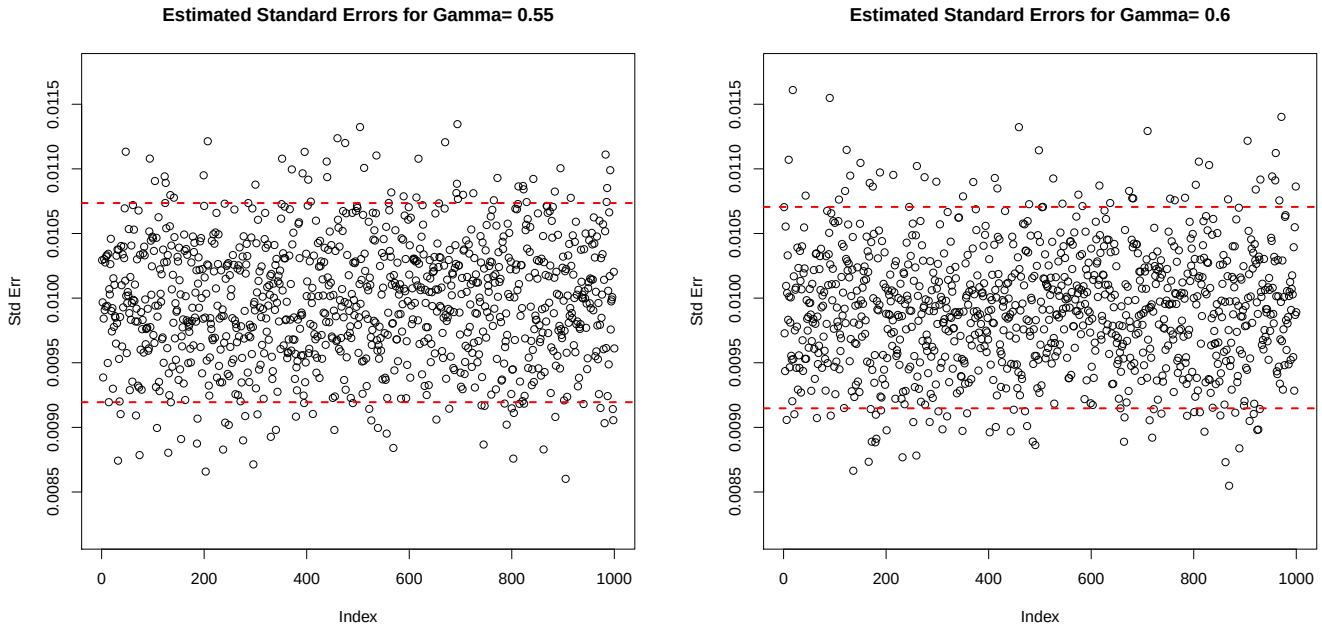

Figure 1: Plots of Standard Errors for Various Gamma

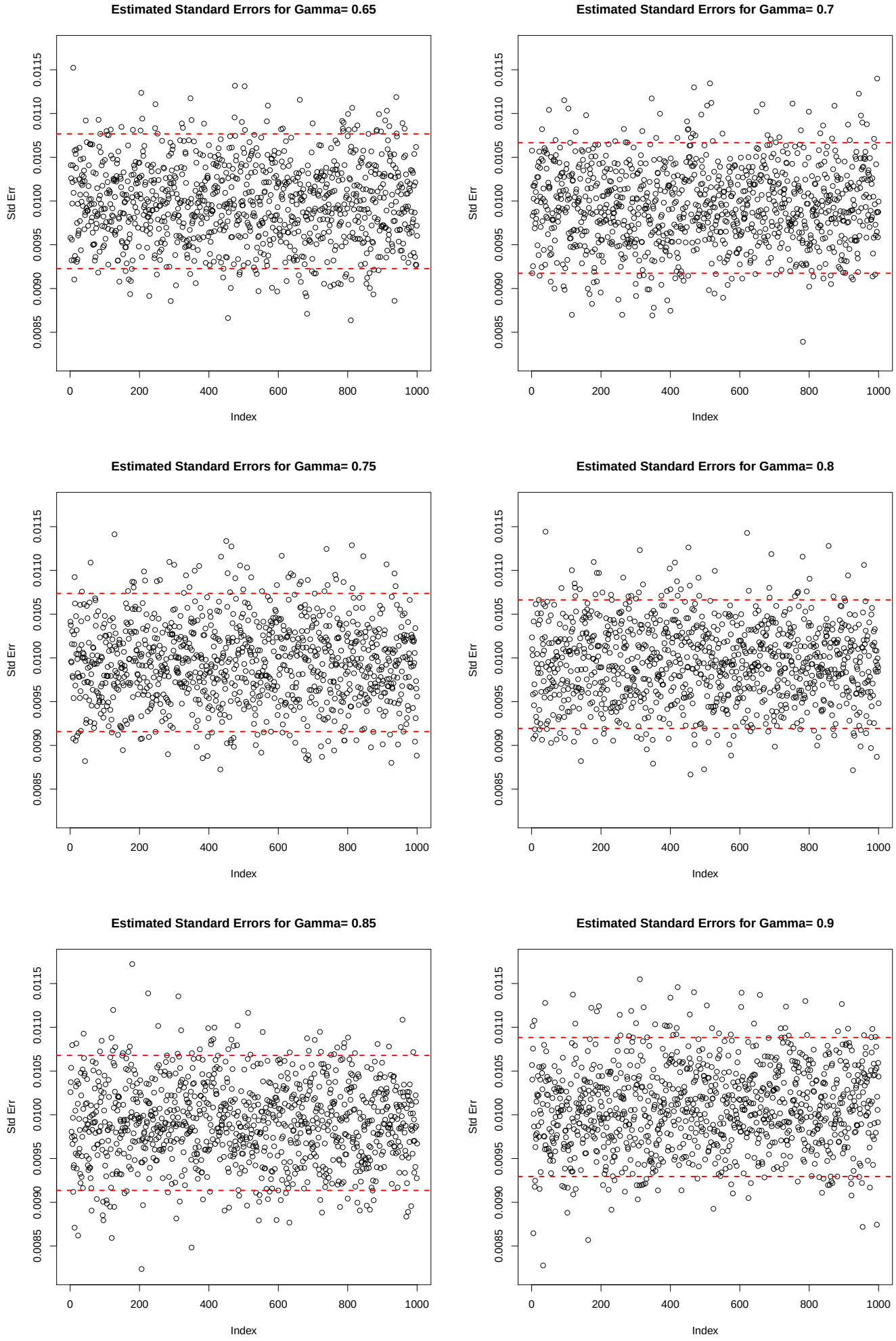

Figure 2: Plots of Standard Errors for Various Gamma

Thus, all else equal, the choice of γ in the range of 0.60 to 0.70 will likely produce reasonable results with a good balance in the trade-off in computation time required. This lends empirical support to this range of γ as intuitive in light of the statement in Kleiner paper that "a simple and standard calculation (Efron and Tibshirani, 1993) shows that each bootstrap resample contains approximately 0.632n distinct points."

Referring back to Table 1 above, we now turn our attention to runtimes. Here we see that there are exponential increases in runtimes for values of γ from 0.80 and above. As marked on the plot in Figure 3 below, $\gamma = 0.70$ again seems to be a reasonable trade off between speed and estimate quality.

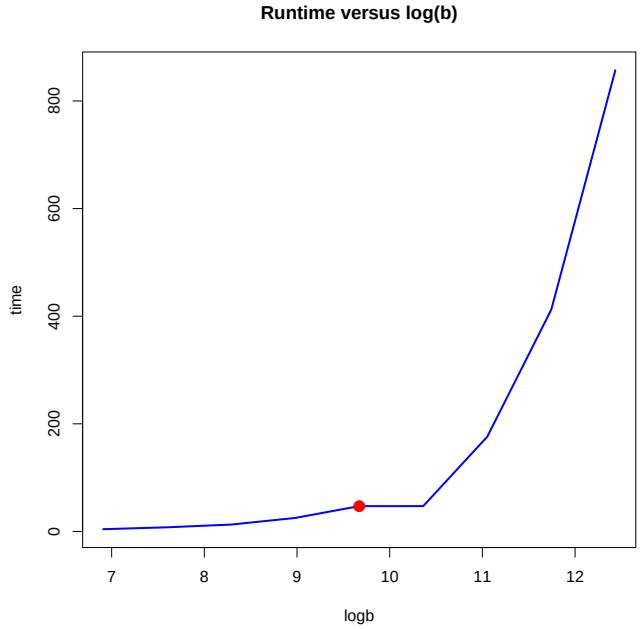

Figure 3: Plot of Runtime versus log(b)

Lastly, we turn our attention to varying values of r and s while holding γ fixed. In order to keep the comparisons relatively apples-to-apples, we fixed the total number of jobs at 250, fixed the value of $\gamma = 0.70$, and then for convenience ran the batch jobs with values of s and r, in that order, with the following pairs, (5,50), (10,25), (25,10), and (50,5). As can be seen in Table 2 below, there does not seem to be much difference among the pairs of s and r of (5,50), (10,25), (25,10) in terms of IQR or SD. However, it does seem clear that pair of s and r of (50,5) clearly does not perform well. This makes some intuitive sense. For the (s,r) pair (5,50), what we are really doing is averaging a larger number of subsamples, r=50, from each of the s=5 subsets, this gives us a reasonably high quality estimate. Howver, the pair (50,5) is really averaging only r=5 subsamples over s=50 subsets, thus we have very small subsample sizes and no surprise that this results in a poor quality estimate. As a result, one should take r to relative large compare to s.

s	5	10	25	50
r	50	25	10	5
Min	0.00839	0.00842	0.00849	0.00778
Q1	0.00964	0.00962	0.00942	0.00906
Med	0.00993	0.00990	0.00971	0.00937
Mean	0.00994	0.00991	0.00972	0.00939
Q3	0.01025	0.01022	0.01002	0.00973
Max	0.01140	0.01135	0.01109	0.01094
SD	0.00046	0.00046	0.00046	0.00049
IQR	0.00061	0.00060	0.00061	0.00066
IQR Relative to s=5, r = 25	1.02167	1.00000	1.01000	1.10833
SD Relative to s=10, r = 25	1.00366	1.00000	1.01535	1.08013

Table 2: Summary Statistics for various values of r and s for fixed γ

4. Conclusions

The key points that a reader should take away from this report is that in the context of our Bag of Little Boostraps regression problem, it seems that choices of γ in the 0.60 to 0.70 produce reasonable quality estimates with relatively reasonable computational runtimes and this works particularly well when the number of bootstrap subsamples r is large relative to the number of subset datasets s.

5. Possible Extensions

One extension that could add value is to break down the runtimes into more detailed steps since it seemed from just a simple tail -f on the .out file that the lm() step was the real bottleneck.

Another obvious extension is to run the analysis on other datasets and other models and see if the same general rules of thumb still hold.

6. Self-Criticisms

One obvious criticism is that this is a purely ad hoc, empirical study that does not use an formal statistical tests and relies mainly on exploratory analysis. A key assumption is that this problem had a known answer from a simulated dataset and we assumed that all the standard errors were on the same scale and thus look at the total variation in all $\widehat{\beta}$'s simultaneously. Clearly this would not be possible if the variation in the $\widehat{\beta}$'s was not on the same scale and thus this is very problem specific.

---

1. Introduction to the R package 'segue'

"Segue has a simple goal: Parallel functionality in R; two lines of code; in under 15 minutes. No shit."

Who could possibly resist installing a package with such a cool tag line? The key feature of the segue packages is that it lets you use Elastic MapReduce as a backend for lapply-style operations. This is essentially a parallel computing approach similar to using a cluster such as Gauss only it uses AWS EMR as the backend. Note that this is not meant to be a full-blown MapReduce paradigm.

The code base for the package is not available on CRAN, but rather is hosted on Google Code along with a Google Forum as it appears to still be a very experimental package. It can be found here:
http://code.google.com/p/segue/
https://groups.google.com/forum/#!forum/segue-r

I began by following the instructions found in the following Jeffrey Breen article and will attempt to add additional details relevant to our servers.
http://jeffreybreen.wordpress.com/2011/01/10/segue-r-to-amazon-elastic-mapreduce-hadoop/

(a) Preliminaries

Note: segue works only on Mac and Linux, but not Windows.

Note: I had problems installing and running segue on R.3.x.x on Pearson and Lipschitz. But was successful in installing and running segue on the Macro server with R.2.13.x. Additionally note that even on the Macro server, if you simply type R and entering into an interactive terminal session, you will fire up R 3.0.2 as the default. You may need to go to /usr/local/lib64/R/bin/R to launch R.2.15 on Macro server.

Note: you will need AWS credentials

Note: segue requires the packages rJava, caTools and bitops if they are not currently installed.

(b) Install the 'seque' package in R.

Note: segue works only on Mac and Linux, but not Windows.

Go get the tar ball from Google Code repository that will be used to install the segue package:
http://code.google.com/p/segue/downloads/list

The fastest way to get the tar ball is to log into the Macro (or other department server) and do mget:

```
wget http://segue.googlecode.com/files/segue_0.05.tar.gz
```

Alternatively, you could download the segue_0.05.tar.gz file directly to your local directory and then use WinSCP to transfer it onto one of the servers or, once on the server, use scp.

Now do the actual installation for R by running the following command line directly on the server (not in an interactive R session):

```
R CMD INSTALL segue_0.05.tar.gz
```

Alternatively, you can open an interactive R session from the Linux command line by simply typeing 'R' to get a terminal R session loaded, or you may have to use /usr/local/lib64/R/bin/R to launch R.2.15 on Macro server ro similar directory paths on other servers. Then install the package using the command line below (with obvious edits for the path to the tar ball):

```
install.packages("/home/mbissell/Stuff/FinalProject/segue_0.05.tar.gz",
                 repos=NULL, type="source")
```

You may get a similar error message as follows. Select 'y' to use a personal library and 'y' to create a personal library directory:

```
Warning in install.packages("/home/mbissell/Stuff/FinalProject/segue_0.05.tar.gz",  :
  'lib = "/usr/local/R-3.0.2/lib64/R/library"' is not writable
Would you like to use a personal library instead?  (y/n) y
Would you like to create a personal library
~/R/x86\_64-unknown-linux-gnu-library/3.0
to install packages into?  (y/n) y
```

You may also get error messages as follows. Identify which dependencies are not already installed and install them from CRAN using install.packages:

```
ERROR: dependencies rJava, caTools are not available for package segue
* removing /home/mbissell/R/x86\_64-unknown-linux-gnu-library/2.15/segue
Warning message:
In install.packages("/home/mbissell/Project/segue\_0.05.tar.gz",  :
  installation of package /home/mbissell/Project/segue\_0.05.tar.gz had non-zero exit status
```

I also had to install the 'rJava', 'caTools', and 'bitops' packages.

(c) Load the segue package.

First load R:
/usr/local/lib64/R/bin/R to launch R.2.15 on Macro server

```
> library(segue)
Loading required package: rJava
Loading required package: caTools
Loading required package: bitops
Segue did not find your AWS credentials. Please run the setCredentials() function.
```

(d) Set your Amazon Credentials (this appears to be needed for each R session).

You can find instructions on how to get your security credentials here:
https://console.aws.amazon.com/iam/home?#security_credential
Note: that you need to have your own account or be an administrator for your account to generate and save keys. Fortunately, AWS has a free tier that you can sign up for.
Follow the instructions on the AWS security credentials website and generate an access key and a secret key. They should pop up in a dialog box and also allow you to save them a file. They should look something like this:
-AccessKey AKIAIOSFODNN7EXAMPLE
-SecretKey wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY

Now go back to your R session and set your credentials by running the following command (with obvious edits for your own actual credentials):

```
setCredentials("AKIAIOSFODNN7EXAMPLE",
               "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY",
               setEnvironmentVariables=TRUE)
```

(e) Create a cluster on AWS EMR directly from your R session.

```
> myCluster <- createCluster(numInstances=5)
STARTING - 2011-01-04 15:07:53
STARTING - 2011-01-04 15:08:24
STARTING - 2011-01-04 15:08:54
STARTING - 2011-01-04 15:09:25
STARTING - 2011-01-04 15:09:56
STARTING - 2011-01-04 15:10:27
STARTING - 2011-01-04 15:10:58
```

```
BOOTSTRAPPING - 2011-01-04 15:11:28
BOOTSTRAPPING - 2011-01-04 15:11:59
BOOTSTRAPPING - 2011-01-04 15:12:30
BOOTSTRAPPING - 2011-01-04 15:13:01
BOOTSTRAPPING - 2011-01-04 15:13:32
BOOTSTRAPPING - 2011-01-04 15:14:03
BOOTSTRAPPING - 2011-01-04 15:14:34
BOOTSTRAPPING - 2011-01-04 15:15:04
WAITING - 2011-01-04 15:15:35
Your Amazon EMR Hadoop Cluster is ready for action.
Remember to terminate your cluster with stopCluster().
Amazon is billing you!
```

(f) Now run some code (in parallel of course).

```
> # first, let's generate a 10-element list of 999 random numbers + 1 NA:

myList <- NULL
set.seed(1)
for (i in 1:10){
   a <- c(rnorm(999), NA)
   myList[[i]] <- a
   }

> # now run it on the cluster
> outputEmr   <- emrlapply(myCluster, myList, mean,  na.rm=T)
RUNNING - 2011-01-04 15:16:57
RUNNING - 2011-01-04 15:17:27
RUNNING - 2011-01-04 15:17:58
WAITING - 2011-01-04 15:18:29
>

> # since this is a toy test case, we can run it locally to compare:
> outputLocal  <- lapply(myList, mean, na.rm=T)

> all.equal(outputEmr, outputLocal)
[1] TRUE



> estimatePi <- function(seed){
   set.seed(seed)
   numDraws <- 1e6

   r <- .5 #radius... in case the unit circle is too boring
   x <- runif(numDraws, min=-r, max=r)
   y <- runif(numDraws, min=-r, max=r)
   inCircle <- ifelse( (x^2 + y^2)^.5 < r , 1, 0)

   return(sum(inCircle) / length(inCircle) * 4)
 }

> seedList <- as.list(1:1e3)

> myEstimates <- emrlapply( myCluster, seedList, estimatePi )
RUNNING - 2011-01-04 15:22:28
RUNNING - 2011-01-04 15:22:59
RUNNING - 2011-01-04 15:23:30
RUNNING - 2011-01-04 15:24:01
RUNNING - 2011-01-04 15:24:32
RUNNING - 2011-01-04 15:25:02
RUNNING - 2011-01-04 15:25:34
RUNNING - 2011-01-04 15:26:04
RUNNING - 2011-01-04 15:26:39
RUNNING - 2011-01-04 15:27:10
RUNNING - 2011-01-04 15:27:41
RUNNING - 2011-01-04 15:28:11
RUNNING - 2011-01-04 15:28:42
RUNNING - 2011-01-04 15:29:13
```

```
RUNNING - 2011-01-04 15:29:44
RUNNING - 2011-01-04 15:30:14
RUNNING - 2011-01-04 15:30:45
RUNNING - 2011-01-04 15:31:16
RUNNING - 2011-01-04 15:31:47
WAITING - 2011-01-04 15:32:18

> stopCluster(myCluster)
> head(myEstimates)
[[1]]
[1] 3.142512

[[2]]
[1] 3.140052

[[3]]
[1] 3.138796

[[4]]
[1] 3.145028

[[5]]
[1] 3.14204

[[6]]
[1] 3.142136

> # Reduce() is R's Reduce() -- look it up! -- and not related to the cluster:
> myPi <- Reduce(sum, myEstimates) / length(myEstimates)

> format(myPi, digits=10)
[1] "3.141586544"

> format(pi, digits=10)
[1] "3.141592654"
```

I worked through the first two code examples and the actually got segfaults in R:

```
 *** caught segfault ***
address 0xc0006, cause 'memory not mapped'

Possible actions:
1: abort (with core dump, if enabled)
2: normal R exit
3: exit R without saving workspace
4: exit R saving workspace
Selection:
 *** caught segfault ***
address 0x88, cause 'memory not mapped'

Possible actions:
1: abort (with core dump, if enabled)
2: normal R exit
3: exit R without saving workspace
4: exit R saving workspace
Selection:
 *** caught segfault ***
address 0xcf58ed68, cause 'memory not mapped'

Possible actions:
1: abort (with core dump, if enabled)
2: normal R exit
3: exit R without saving workspace
4: exit R saving workspace
Selection: Segmentation fault
```

References

B. Efron and R. Tibshirani. An Introduction to the Bootstrap. Chapman and Hall, 1993.

Kleiner, A., Talwalkar, A., Sarkar, P., Jordan, M.I. (2011) A Scalable Bootstrap for Massive Data. arXiv: 1112.5016

Code Appendix

```
# Fit simple linear regression on d=1,000 covariates with *no* intercept
# Errors ~ N(0,sigma^2)
# Goal is to find SE(beta.hat.1),...,SE(beta.hat.1000)

# Data description:
# blb_lin_reg_data.txt: n=1,000,000 observations. Each row corresponds to an observation,
# each column to a variable. First d=1,000 columns are covariates (X1,...,X1000),
# final column corresponds to the response variable y
# Mini dataset d=40 covariates and n=10,000 observations

# Clear out everything in memory for a nice clean run and easier debugging
rm(list=ls())

s = 5         # s = number of distinct subsamples
               ##### r must match up in BLB_lin_reg_process.R
r = 50          # r = number of bootstrap samples to do for each distinct subsample
               ##### s must match up in BLB_lin_reg_process.R

mini <- FALSE   # use the mini dataset or the full dataset?

#================================ Setup for running on Gauss... ================================#

args <- commandArgs(TRUE)

cat("Command-line arguments:\n")
print(args)

####
# sim_start ==> Lowest possible dataset number
###

####################
sim_start <- 1000
####################

if (length(args)==0){
  sim_num <- sim_start + 1
  set.seed(121231)
} else {
  # SLURM can use either 0- or 1-indexing...
  # Lets use 1-indexing here...
  sim_num <- sim_start + as.numeric(args[1])

  # Get the job number from the argument 1:(s*r) (i.e. 1:5*50 = 1:250)
  job.num = as.numeric(args[1])

  # Get the s_index by using mod s.
  # Also, if job.num mod s == 0, then it is subsample dataset s (i.e. 5)
  s_index = job.num %% s
  if (s_index == 0){
    s_index = s
  }

  # Get the r_index by using ceiling(job.num/s).
  # Also, if job.num mod r == 0, then it is bootstrap sample r (i.e. 50) within subsample dataset s
  r_index = ceiling(job.num / s)
  if (r_index == 0){
    r_index = r
  }

  # The first seed must be a function of the s_index to ensure that the subsample is the same
  # for same values of the s_index
  sim_seed <- (762*(s_index) + 121231)
  set.seed(sim_seed)
}

# Some checks that go into the .out file
cat(paste("\nAnalyzing dataset number ",sim_num,"...\n\n",sep=""))
cat(paste("\nRunning s_index ",s_index," r_index ",r_index," seed ",sim_seed," job.num ",job.num,"...\n\n",sep=""))


#================================ Run the simulation study ================================#

# Load packages:
library(BH)
library(bigmemory.sri)
library(bigmemory)
library(biganalytics)

# I/O specifications:
# datapath = "C:/Users/Michael/Documents/Michael UC Davis/STA 250 Adv Stat Computing/HW2/"
datapath <- "/home/pdbaines/data/"
outpath <- "output/"


# mini or full?
if (mini){
   rootfilename <- "blb_lin_reg_mini"
} else {
   rootfilename <- "blb_lin_reg_data"
}

# Filenames:
datafile = paste0(datapath, rootfilename, ".desc")

# Set up I/O stuff:

# Attach big.matrix :
data.full = attach.big.matrix(datafile)

# Remaining BLB specs:
n = nrow(data.full)     # n = nrows of the full dataset
```

```
d = dim(data.full)[2]-1 # d = number of covariates
gamma = 0.90
b = ceiling(n^gamma)    # b = size of each subset b<n, taken without replacement from the full dataset
                        # then resample n points from b<n

# Extract the subset:
# Get b row indices from 1:n without replacement to use for each of the s subsamples
# based on the seed set above and use the same seed for same values of s_index
row.indices = sample(1:n, size=b, replace=FALSE)
X.sub.sample = data.full[row.indices,1:d]
Y.sub.sample = data.full[row.indices,d+1]

# Checks the subsets:
# dim(X.sub.sample)
# dim(Y.sub.sample)
# outfile = paste0("dump/","BLB_sample_indices_",sprintf("%02d",s_index),"_",sprintf("%02d",r_index),".txt")
# write.table(x=cbind(sim_seed,row.indices),file=outfile, sep=",", col.names=TRUE, row.names=FALSE)

# Reset simulation seed:
# The seed for the bootstrap sample must be different for each bootstrap sample r within
# each subsample s, so make the seed a function of s_index and r_index with no overlap
sim_seed <- (762*(s_index) + 121231 + r_index)
set.seed(sim_seed)

# Bootstrap dataset:
# Select bootstrap sample of size n with replacement from subsample
# Do this using rmultinom to distribute n balls into b boxes
# where each box has equal probability, i.e. 1/b
# This gives us the weights (i.e. # of replications) to put on each of the b row indices in
# our subsample so that we get a bootstrap sample of size n
data.weights = rmultinom(1, size = n, rep(1/b, b))

# Check the weights
#outfile = paste0("dump/","BLB_bootstrap_weights_",sprintf("%02d",s_index),"_",sprintf("%02d",r_index),".txt")
#write.table(x=data.weights,file=outfile, sep=",", col.names=TRUE, row.names=FALSE)

# Fit the linear regression using lm and get the coefficients:
model = lm(Y.sub.sample ~ 0 + X.sub.sample, weights=data.weights)
beta.hat = model$coefficients

# Output file:
outfile = paste0("output/","coef_",sprintf("%02d",s_index),"_",sprintf("%02d",r_index),".txt")

# Save estimates to file:
write.table(x=beta.hat,file=outfile, sep=",", col.names=TRUE, row.names=FALSE)

cat("done. :)\n")

q("no")
```
