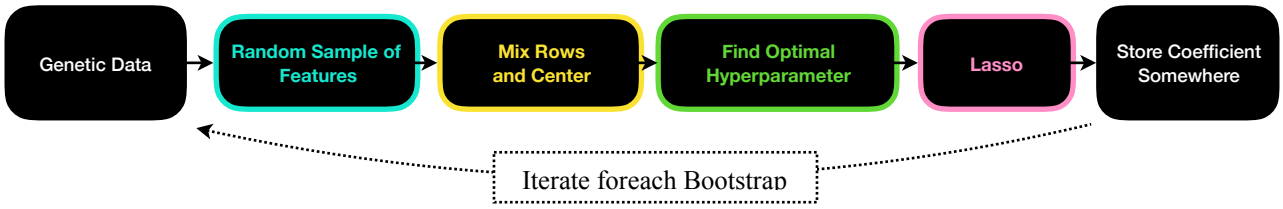
Spark for Random Lasso Implementation in R

Introduction

This report outlines an array of novel approaches and relevant findings regarding **parallelizing Random Lasso** using **Apache Spark** in the **R language**. The goal of this report is not to present a clearcut solution for the problem at hand, but to present several avenues of attack and discuss the reasons for the changing nature of this problem.

Parallelize Random Lasso

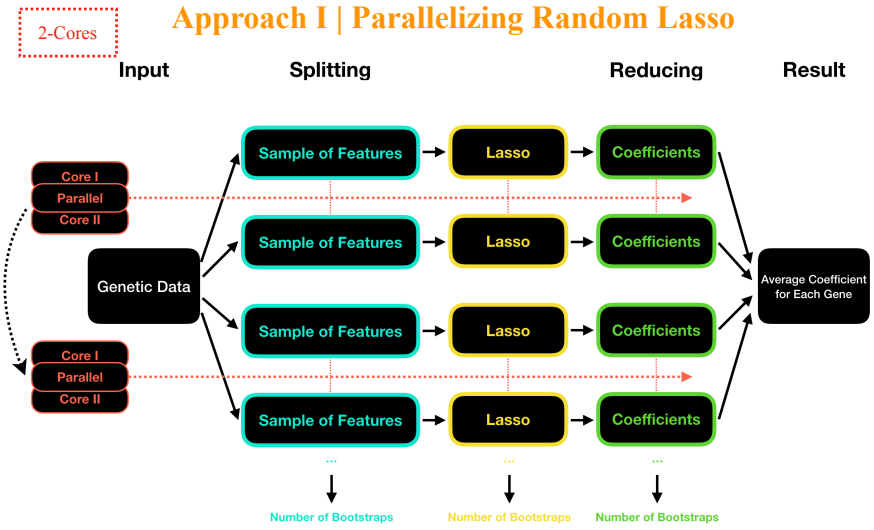
Random Lasso is a heavyweight algorithm involving several iterations of computationally burdensome tasks, e.g. transposing, matrix multiplication, hyper-parameter tuning. Below is a general outline of the processes.



Fortunately, each iteration of tasks, henceforth known as **bootstraps**, is **highly independent**. Below are a few way to parallelize Random Lasso:

I. [Matthew] We can run each bootstrap above independently in parallel.
II. [Daniel] We can run the sampling in parallel first then run lasso in parallel later.
III. We can possibly run Lasso for <u>just</u> one bootstrap in parallel — depending on library.

Note, you can not run a combination of these! It is not possible to run parallel computing within anouther layer of parallel computing. For example, if a **two core computer** was running the **two** bootstraps of lasso in parallel, we couldn't further divide up the tasks with two cores within the lasso function since this would then require **four** cores.



Function Programming

Scala is not a purely functional language, but often it is better to use a functional style of programming — especially for RDD manipulation. Function programming languages are difficult for new programmers to learn, since programmers are mainly taught imperative languages.

R is both an imperative and functional language. You can use either, however many high-level R programmers prefer to use the functional style. The challenge is often to convert for-loops into their functional counterpart, which is a form of vectorization.

Vectorization

Speed in R

Vectorization > For-Loops

I completely vectorized the Random Lasso algorithm, so there are no for-loops. **This was half the battle, as this is Random Lasso in its functional form.** For functional programming we use apply( ) and reduce( ) in vanilla R. **Notice that apply( ) and reduce( ) sound similar to MapReduce.** I plan to implement MapReduce in a similar style as apply( ) and reduce( ).

Vanilla R

```
beta.hat <- lapply(1:bootstraps, part1, x, y)
importance.measure <- Reduce(mean, beta.hat)
```

part1: a function that samples, mixes, and centers features and then runs lasso, **returns** coefficients.
mean: finds the mean of a vector, **returns** the mean.

lapply: in this example, returns a list of coefficients for each random sampling
Reduce: finds the mean of each list column-wise, as if they were stacked up into a matrix.

I plan to implement MapReduce similar to the lapply( ) and Reduce( ) example. We can run lapply( ) in parallel, since these processes are completely independent. I experimented with this using **mclapply( )** from the **library(parallel).**

Parallelizing without Spark

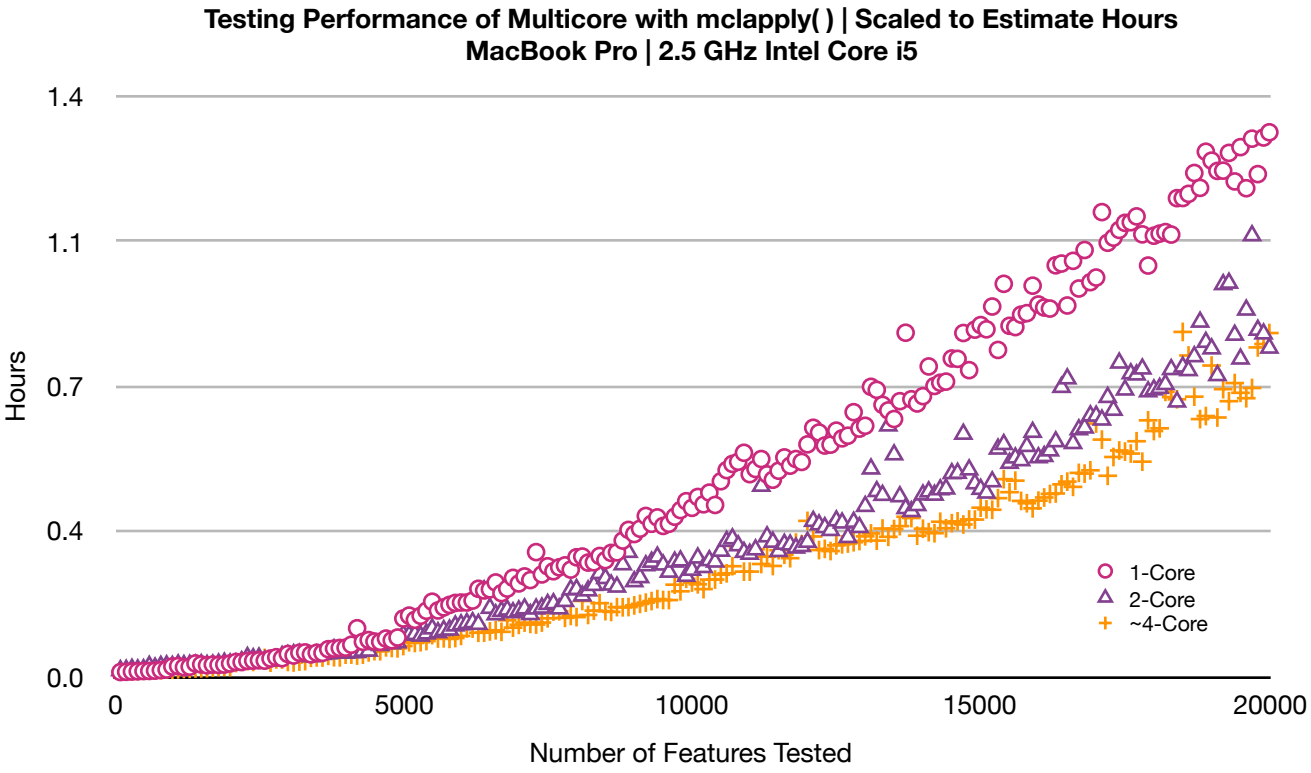
```
library(parallel)

beta.hat <- mclapply(1:bootstraps, part1, x, y, mc.cores = 2)
importance.measure <- Reduce(mean, beta.hat)
```

mc.cores: number of cores to run each function call within of lapply( ) in parallel.

Note, **mclapply( )** runs identical to **lapply( )**, except each function call is run in parallel depending on how many cores or specified. For example, lapply( ) returns a vector coefficients **one** at a time into a list in the example above, but mclapply( ) returns a vector coefficients **two** at a time into a list in the example above.

I can set up mclapply( ) to automatically detect the number or cores in a computer, so researchers can use Random Lasso in parallel with seamlessly setup. However, Spark will be needed for more complicated systems using distributed computing and big data. We can see some early results below:



Samples: One-tenth of Features
Note, tested on a 2-core computer with 2-virtual-cores.

Packages

In this section we go over the main differences between the two most popular Spark packages in R. They can not be used together. Later, we list some helper libraries used with Sparklyr.

SparkR

- An older and less supported package for using Spark in R.
- Supports RDD

- Has hidden :::Map( ) and :::Reduce( ) functions.

Sparklyr

- A newer and more supported package for using Spark in R.
- Integrated with R Studio, has Spark UI features.
- Does not support RDD, instead possibly use **rddlist** package:
  - rddlist(sc, X, cache = TRUE)
  - lapply_rdd(X, FUN, cache = TRUE)
- Instead of RDD sparklyr uses **ML Pipelines**
- Support for major R packages: **dplyr**, **purr**, DBI, broom, etc.
- Support for **H20** package, a powerful new machine learning library.

Documentation: https://spark.rstudio.com/reference/

Dplyr

This package is used in **sparklyr**, and is primarily for adding a syntactical style that functional programmer like to use. Primarily this adds **pipes** → %>% an example below:

Normal Syntax:

```
sample <- sdf_sample(all_data, replacement = TRUE)
```

Dplyr Syntax:

```
sample <- all_data %>%
     sdf_sample(replacement = TRUE)
```

Naturally, you may think this is unnecessary, and it is! But in order to understand online examples given by advanced R programmers, you need to understand this syntactical style. I personally am adopting this syntactic style.

RSparkling

H2O.ai is a new framework the released a new generation ML libraries for R and Sparklyr. H2O offers **"the best high performance machine learning"** in the Hadoop eco-system right now:

Sparkling Water = H20 + Apache Spark

Documentation: http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/glm.html

Quick Install

```
install.packages("sparklyr")
install.packages("dplyr")
install.packages("purrr")
install.packages("rsparkling")
```

Challenges

There are many cookie-cutter (simple) tutorials on how to use Spark for an array of machine learning and regression applications. Random Lasso is a uniquely different flavor regression analysts, therefore has some challenges:

- Spark focuses on row manipulation not column manipulation. Partitioning the data is done by rows. Many Spark functions are designed for manipulating the rows and not columns. Someone times it is difficult to find an equivalent column-wise function.
- Sparklyr does not use the traditional map( ) and reduce( ) functions. However, there are still ways to accomplish our parallel and distributed goals in Sparklyr: https://www.quora.com/Does-one-have-to-write-MapReduce-jobs-when-using-R-packages-in-SparkR
- When selecting a column in Scala you use the actual name of the column and not the index, we are using many columns which can be a challenge at first. To select all columns use "." period.
- There is a lack of examples in sparklyr, since it is relatively new. Pyspark and Scala have many examples provided by Apache.

Despite these challenges, sparklyr is a powerful tool for statical analysis in R, and we are prepared to tackle these challenges!

Code

Loading CSV into Spark Dataframe

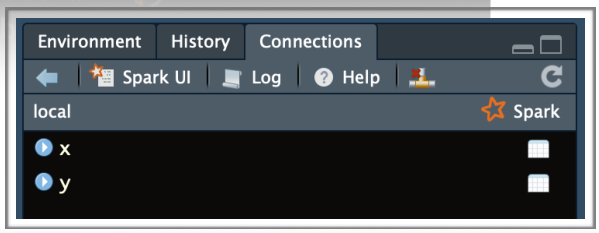
```
setwd("~/Dropbox/KSULasso/R/SparkSandbox/")
x = spark_read_csv(sc, name = "x", path = "x")
y = spark_read_csv(sc, name = "y", path = "y")
```

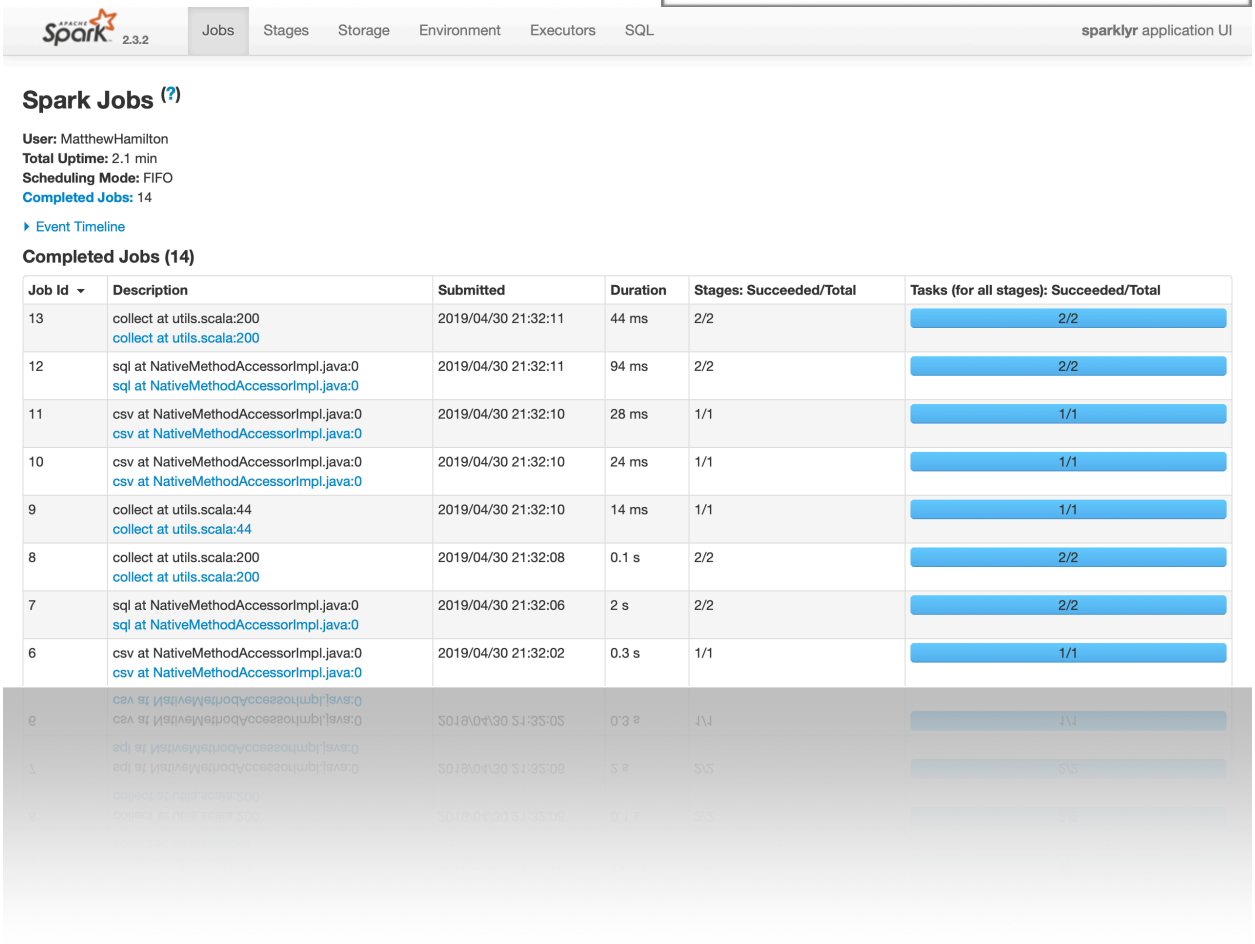
Sampling & Lasso for One Bootstrap

```
all_data <- copy_to(sc, all_data_R, overwrite = TRUE)

coefficients <- all_data %>%
  select(c(1, sample(features, samples, replace = FALSE))) %>%
  sdf_sample(replacement = TRUE) %>%
  ml_linear_regression(response = dependent ~ .,
                       fit_intercept = TRUE,
                       lastic_net_param = 1)
```

UI of Sparklyr in R-Studio





Spark Jobs (?)

User: MatthewHamilton
Total Uptime: 2.1 min
Scheduling Mode: FIFO
Completed Jobs: 14

▸ Event Timeline

Completed Jobs (14)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
13	collect at utils.scala:200 collect at utils.scala:200	2019/04/30 21:32:11	44 ms	2/2	2/2
12	sql at NativeMethodAccessorImpl.java:0 sql at NativeMethodAccessorImpl.java:0	2019/04/30 21:32:11	94 ms	2/2	2/2
11	csv at NativeMethodAccessorImpl.java:0 csv at NativeMethodAccessorImpl.java:0	2019/04/30 21:32:10	28 ms	1/1	1/1
10	csv at NativeMethodAccessorImpl.java:0 csv at NativeMethodAccessorImpl.java:0	2019/04/30 21:32:10	24 ms	1/1	1/1
9	collect at utils.scala:44 collect at utils.scala:44	2019/04/30 21:32:10	14 ms	1/1	1/1
8	collect at utils.scala:200 collect at utils.scala:200	2019/04/30 21:32:08	0.1 s	2/2	2/2
7	sql at NativeMethodAccessorImpl.java:0 sql at NativeMethodAccessorImpl.java:0	2019/04/30 21:32:06	2 s	2/2	2/2
6	csv at NativeMethodAccessorImpl.java:0 csv at NativeMethodAccessorImpl.java:0	2019/04/30 21:32:02	0.3 s	1/1	1/1

MapReduce

Daniel:

An issue encountered with the sampling of the overall dataset is that if we have each node sample from the overall dataset, each node has to render the overall dataset to extra only a fraction. The pseudo code suggests sampling the bootstraps before sending to each node so that the overall dataset is only rendered once. The current dataset we are using is relatively small. When dealing with very large datasets in the thousands or millions of samples, the problem of rendering the overall dataset on each node will slow down the speed.

```
Map(key=bootstrapNumber, value=bootstrapData)
    #apply algorithm of (Random) Lasso
    emit(key=covariants, value=coefficients)
Reduce(key=covariants, value=coefficients)
    #apply transformation to row with covariant headers
    emit(key=rowID, value=1DVector(allCoefficients))
```