Realtime tuning of the ADaM parameters with an example of the big actigraphy dataset

Mehrnoosh Oghbaie, Johnson& Johnson, New York, US

ABSTRACT

As part of the Digital Health toolset, ActiGraph GT9X (GT9X, n.d.) is a high-resolution liquid crystal display (LCD) 3-axis accelerometer for measuring continuous secondary movement and position over a period of time. A common approach is to analyze the summarized time series. While summarized data allows hypothesized testing, a lot of information gets lost during this process, including optimum summarization rules and parameters and how the summary data is affected by these parameters' changes. In this paper, we discuss programming strategies and techniques to handle digital health data to speed up big data analytics and parameter sensitivity analysis. We developed an interactive R shiny interface with user-friendly features and visualization that enables users to change the parameters and rules and see their effect in real-time. To achieve that, we borrowed object-oriented programming as well as functional programming, recursion, R6 object and Apache Spark machine.

INTRODUCTION

An actigraph is a device that is usually worn on the wrist and is used to monitor subjects' movement, sleep patterns and their activity during wake time. Actigraph is used in clinical trials for a variety of purposes including evaluating medication sleep side effects or levels of activity among patients with sleep disorders. Figure 1 shows the distribution of therapeutic area in clinical studies that utilized actigraph devices. It's notable that behavioral and mental disorders studies, followed by nervous system diseases seem to utilize actigraph data more than other areas.

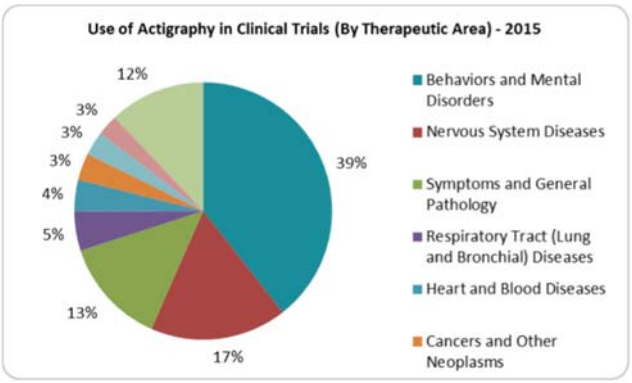

Figure 1: Use of Actigraphy in Clinical Trial by Therapeutic Area (Ilancheran & Zammit, 2016)

Technical Difficulties

Actigraph data often includes epoch level data which is normally minute level or 30 second's level. Epoch level data could be anything from 1-20Gb depends on the number of participants and length of the study. Storing, analyzing, transforming, and summarizing this size of data could pose computational challenges. Producing aDaM file from vendor data could often takes up to 4 hours for statistical programmers. Finding the optimum parameters for aDaM file production is a separate problem. How can we effectively compare different scenarios where in each scenario we change one or more parameters?

High level Solution

In brief, we succeeded to reduce the run time of aDaM file production from 3-4 hours to less than a minute by utilizing Spark and R6 object-oriented programming (OOP). Furthermore, we used the pipeline in the backend of a shiny app that enables users to change the dynamic parameters and refresh the shiny view and download pictures with name and captions that contain enough

data for comparison. The app consists of four main modules: group stats, weekly stats, daily stats and individual stats.

Why Spark?

Hadoop MapReduce is a framework that uses parallel, distributed algorithms for big data processing. It enables developers to write big, parallelized programs without worrying about work distribution or interruption. However, Hadoop-MapReduce runs a job in a sequential multi-process; at each step requires to read the data from cluster and return the result to HDFS. Thus, MapReduce jobs could run slow.

Spark is an open-source framework designed for machine learning, real-time analysis, and machine learning algorithms. While it doesn't have its own storage system, it runs on other storage systems such as HDFS, Amazon Redshift, Cassandra, and others. Spark address MapReduce latency by in-memory processing where data is read into memory once and result written back when operation is done. Also, Spark uses an in-memory cache that speeds up algorithms that repeatedly run a function on the same dataset. This makes Spark multiple times faster than MapReduce.

Spark addresses the limitations by doing processing in-memory, reducing the number of steps in a job, and by reusing data across multiple parallel operations.

Spark has five main sub modules:

- Spark Core
- Spark SQL
- Spark Streaming
- Spark MLlib
- Spark GraphX

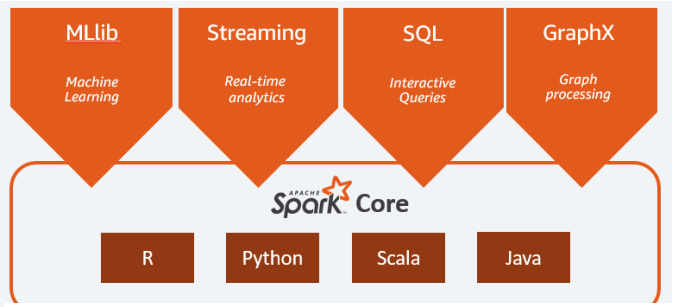

Figure 2 (Introduction to Apache Spark, n.d.)

Sparklyr: Big data enabler for R

Sparklyr enables users to interact with Spark using familiar R interfaces, such as dplyr, broom or DBI. Users gain access to Spark's distributed Machine Learning libraries through Sparklyr and can run distributed R code inside Spark environment.

We mainly used Spark SQL module for fast import and manipulation of the data. Users can access Spark SQL through dplyr and run predefined Spark functions or customized functions. We used dplyr and it's main five SQL functions for our project:

- select() ~ SELECT
- filter() ~ WHERE
- arrange() ~ ORDER
- summarise() ~ aggregators: sum, min, sd, etc.
- mutate() ~ operators: +, *, log, etc.

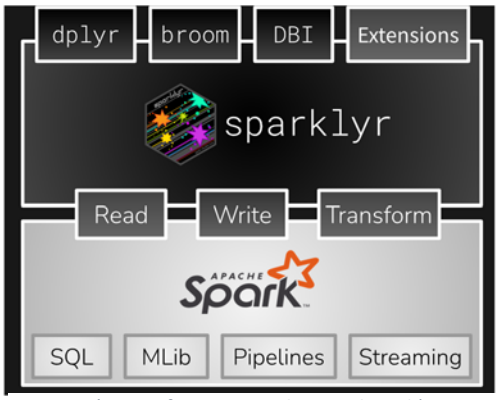

Figure 3 (R interface to Apache Spark, n.d.)

Notes about functional programming in R

Motivation

R is mainly a functional programming (FP) language and thus provides many options for creation of functions. Yet the main motivation for FP is cleaning and summarizing the data before serious analysis (Wickham, Advanced R, n.d.).

Anonymous functions

In R, functions aren't automatically bound to a name. If you choose not to give the function a name, you get an anonymous function.

Functionals instead of for loop

Functionals (apply, lapply, sapply,etc) reduce bugs in your code by better communicating intent. They are written in C and implemented in base R are for most part well tested and they're used by so many people.

```
lapply(mtcars, function(x) length(unique(x)))
```

To purr or not to purr

One of the best-known functions in purrr library is map. The map functions transform their input by applying a function to each element of a list or atomic vector and returning an object of the same length as the input (Attalides, n.d.). One of the features that purrr library provides is to split the data to pieces, divide the task to multiple sub tasks that can be written with the appropriate map functions and extract the output in the desired format.

```
# A more realistic example: split a data frame into pieces, fit a
# model to each piece, summarise and extract R^2
mtcars |>
  split(mtcars$cyl) |>
  map(\(df) lm(mpg ~ wt, data = df)) |>
  map(summary) |>
  map_dbl("r.squared")
#>         4         6         8
#> 0.5086326 0.4645102 0.4229655
```

Parallel programming foreach

Many computational tasks can be run faster with parallel computing. Different pieces of a computational task can be executed simultaneously across multiple processors or cores. Parallel computing is more effective when the size of data is bigger than certain size (Privé, n.d.). There is multiple factor that we must keep in mind when deciding on whether using parallel computing or not:

- Task start up and termination cost.
- Synchronization problems
- Cost of communication among multiple tasks.
- Software overhead of libraries, parallel compiler, and supportive OS.
- Complexity of designing, coding, debugging and maintenance.

Foreach is one of the easiest packages to start parallel programming with. It looks very much like a for loop but functions more like lapply.

```
library(foreach)

foreach (i = 1:3) %do% {sqrt(i) }

        [[1]] [1] 1

        [[2]] [1] 1.414214

        [[3]] [1] 1.732051

lapply (1:3, function(i) {sqrt(i)})

        [[1]] [1] 1

        [[2]] [1] 1.414214

        [[3]] [1] 1.732051
```

However, foreach has features that makes it more suitable for constructing tables such as. combine options. It provides users with the option to integrate output in format that desired in the same line of command.

```
foreach (i = 1:3, .combine = 'c') %do% {sqrt(i)}

        [1] 1.000000 1.414214 1.732051

library(foreach); library(parallel)

foreach (i = 1:3, .combine = 'c') %do% {sqrt(i)}

        [1] 1.000000 1.414214 1.732051

cl <- parallel::makeCluster(2)

 doParallel::registerDoParallel(cl)

foreach (i = 1:3, .combine = 'c') %dopar% {sqrt(i)}

        [1] 1.000000 1.414214 1.732051

 parallel::stopCluster(cl)
```

Don’t forget to benchmark

It’s always helpful to test your function performance between implementing it in your pipeline. Below you see a good example when parallel programming increases the time instead of decreasing. As mentioned in previous section, there can be overhead, or communication cost associated with parallel programming. Parallel programming efficiency is better observed as the size of the data increases.

```
library(microbenchmark)

microbenchmark(

        foreach(i = 1:3, .combine = 'c') %do% {sqrt(i)},

        foreach(i = 1:3, .combine = 'c') %dopar% {sqrt(i)}

)
```

```
Unit: milliseconds
                                                    expr       min
    foreach(i = 1:3, .combine = "c") %do% {     sqrt(i) }  1.299025
 foreach(i = 1:3, .combine = "c") %dopar% {     sqrt(i) } 41.397892
        lq      mean    median        uq      max neval cld
  1.338756  1.467948  1.379935  1.404856 10.33015   100  a
 41.709773 41.996617 41.971969 42.163705 44.16577   100   b
```

Loops that should be left as is

Some loops have no natural functional equivalent such as:

- modifying in place
- recursive functions
- while loops

While we can use functionals to tackle these problems, it will never give us a clean and understandable code. You'll end up writing a code that is not understandable by others.

Object Oriented Programming and R6 objects

Object-**O**riented **P**rogramming revolves around **objects**. We create these objects from blueprints named **classes**. Each class has **properties** and **methods** that describe how each object behaves. (Wickham, R6, n.d.)

OOP revolves around four principles:

- Encapsulation
- Inheritance
- Abstraction
- Polymorphism

Solution at a Glance!

Using Spark

Figure4 shows a typical pipeline for accessing data in aws S3 bucket, manipulate and visualizing it.

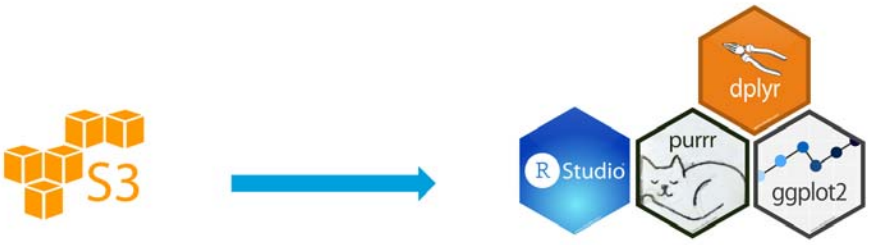

Figure 4: Typical R pipeline for reading data from S3 bucket and manipulate it

Adding Spark to the pipeline and reading data from Spark schema reduces the import time drastically. In addition, certain functions can be run much faster if written in Spark in Spark or replaced with predefined.

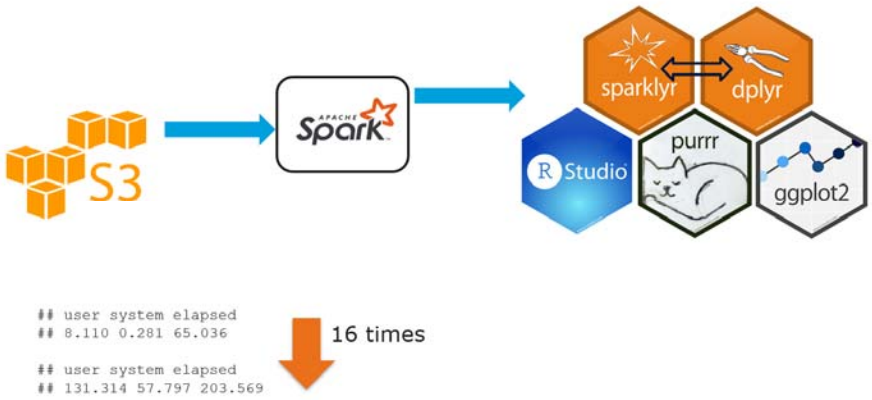

Figure 5: R pipeline with Spark

Dividing functions to static & dynamic

One strategy to find a better solution is to list the dynamic parameters and their effect in the output tables. While we finalize the dynamic parameters, we can think of dynamic and static functions that correspond to the task at hand. In this text, we use the expression dynamic for functions that change in dynamic parameters can alter the output and static for functions that return output regardless of the dynamic parameters.

Tables	Main functions	Dynamic parameters
• Epochsummarydata (minute)	• importMetadata	• minimum.WearMinutes
• Subjectdaystats (daily)	• importCsvDatafromS3	• minimum_days
• Subjectsleepperiodmetrics (daily)	• cleanDaylySubStats	• exclude_weekends
	• calculateWklySubStats	• earliestInBedTime
	• calculateSleepPercent	• latestInBedTime
	• cleanDaylySubSleepMetrics	• latestOutBedTime
	• calculateWklySubSleepMetrics	• wearpct.min
	• mergeFiles	• mergedPeriodThreshold

Figure 6: List of tables, min functions and dynamic parameters

Using R6 to implement functions

We can write attributes and methods that fit the task at hand by grouping the small functions under umbrella of main functions with names that are easily understood. For example, we have two tables with daily statistics that we wish to extract weekly data from. We use expressive names like `cleanDaylySubStats()` or `calculateWeeklySubStats()` as you can see in figure 7.

```
actigraph <- Template.project$
    new()$
    importMetadata(metadata)$
    importCsvDatafromS3(sc, bucket, prefix,  transfer_date)$
    cleanDaylySubStats(minimum.WearMinutes)$
    calculateWklySubStats(minimum_days, exclude_weekends)$
    calculateSleepPercent()$
    cleanDaylySubSleepMetrics(earliestInBedTime, latestInBedTime, latestOutBedTime,
    wearpct.min, mergedPeriodThreshold)$
    calculateWklySubSleepMetrics(minimum_days, exclude_weekends)$
    mergeFiles()
```

Figure 7: R pipeline written with R6 class

Passing R6 object to shiny module through reactiveValues

Our shiny app consists of four main modules as shown in figure 8: Group stats, Weekly stats, daily stats and individual stats. R6 object is stored in S3 bucket and saved in a reactiveValues before being passed to shiny modules. New instance of the object will be constructed in each module and attributes (tables) replaced with the default ones.

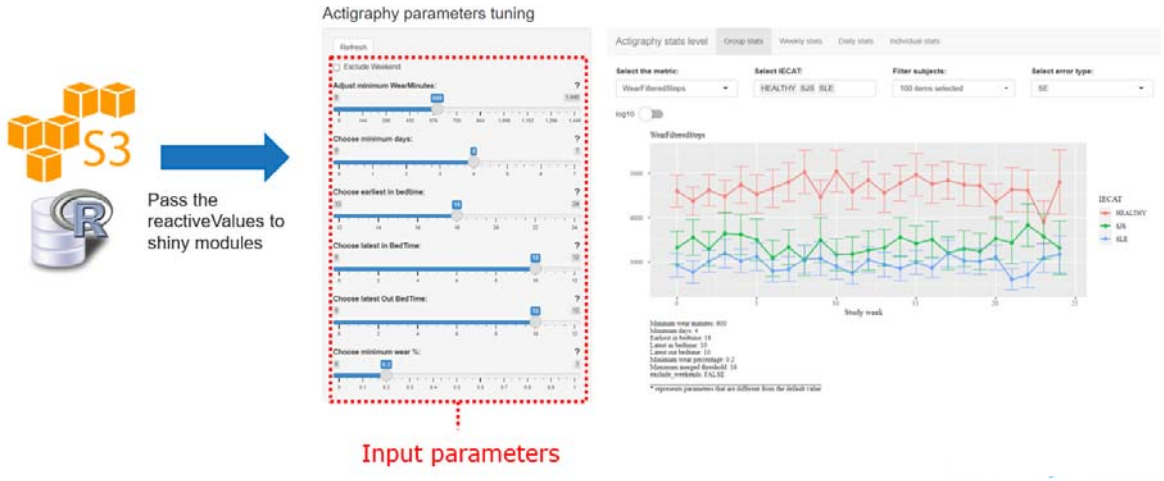

Figure 8: Passing R6 object to shiny app through reactiveValues

Updating R6 object with new parameters

Upon pressing the refresh button, the detected parameters' changes will be detected by shiny app, reactiveValues will be updated and shared will the four modules. Each module contains a plot that has detailed caption for potential comparison and can be downloaded for future reference.

References

Attalides, N. (n.d.). *Purrr or not to purrr*. Retrieved from R bloggers: https://www.r-bloggers.com/2018/05/to-purrr-or-not-to-purrr/

GT9X. (n.d.). Retrieved from Actigraph: https://www.actigraph.nl/en/product/72/actigraph-link-gt9x.html

Ilancheran, M., & Zammit, G. K. (2016, February). Impact of Actigraphy in Clinical Research. *Applied Clinical Trials*.

Introduction to Apache Spark. (n.d.). Retrieved from aws: https://aws.amazon.com/big-data/what-is-spark/#:~:text=Spark%20is%20an%20open%20source,Couchbase%2C%20Cassandra%2C%20and%20others

Privé, F. (n.d.). *A guide to parallelism in R*. Retrieved from https://privefl.github.io/blog/a-guide-to-parallelism-in-r/

R interface to Apache Spark. (n.d.). Retrieved from Spark: https://spark.rstudio.com/

Wickham, H. (n.d.). *Advanced R*. Retrieved from http://adv-r.had.co.nz/Functional-programming.html

Wickham, H. (n.d.). *R6*. Retrieved from https://adv-r.hadley.nz/r6.html

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at: Johnson& Johnson Raritan (1003 Highway 202), New Jersey, United States.

Email: moghbaie@its.jnj.com