Reproducibility and Audit Risk

# Executive Summary

## Objective

This report contains the findings and recommendations relating to reproducibility issues and audit risks in a specific data analytics project. The report also includes references to the code which has been partly re-written as per some of the recommendations in this report.

## Approach

An existing data analytics project was identified which covers some of the key data science tasks that a data analyst/scientist has to perform on a day to day basis. The chosen project’s objectives are to identify the top factors that are related to a driver’s alertness or lack of based on the dataset provided and choosing an optimal classifier from a bag of models that can predict whether the driver is alert or not alert using the top factors/predictors with highest accuracy.

The key themes on which the reproducibility issues and risks were identified are as follows:

* Version controlling of source
* Collaboration/Portability/Reproducibility
* Code Readability and Layout
* Coding & Naming Standards
* Literate Programming

## Highlights

The R code in which the project was written, was put together without any version control system and did not have any dedicated repository which presented the first challenge of reprodroducing the correct version of the code. The nature of the code structure and building process resulted in multiple occurrences of redundant codes, confusing variable naming conventions, multiple coding styles, missing/misleading code comments and insufficient code documentation.

Instances of coding errors were also discovered as part of the process of reproducing the projects. The technical review was limited to finding reproducibility issues and risks and excludes any recommendation on the appropriate methodology of model selection or model validation techniques.

This report contains the various reproducibility/audit issues and risks that were found in the review process, along with a number of recommendations for improvements that includes using source control tool, following one coding style and using appropriate coding standards/naming conventions, implementing defensive programming and advantage of using software containers. An excerpt of the original code is also included in the report along with the recommended changes to code to make it reproducible and more auditable.

# Introduction

## Project Introduction

The project, under review, was undertaken as part of practical work-related project experience in one of the MDSI (Master Data Science & Innovation) subject named Advanced Data Analytics Algorithms.

The dataset is a simulated observations consisting of tests taken by 100 vehicle drivers of both genders, of different ages and ethnic backgrounds, who have been sampled a total of 500 times against 3 key sets of variables (driver’s physiological attributes combined with vehicular and environmental attributes acquired from the simulated environment). There are 33 attributes in the dataset in total.

* Physiological (8 features) defined simply as P1 to P8.
* Environmental (11 features) defined simply as E1 to E11.
* Vehicular (11 features) defined simply as V1 to V11.

Each driver trail has been recorded in a simulated driving environment for a period of 2 minutes and an observation recorded every 100 milliseconds. Each driver’s trail has been defined uniquely and labelled at TrialID and every observation within each trail is defined uniquely and labelled as ObsNum.

The key objectives of the project are:

* Identify the key factors that explains most of the driver’s alertness or lack of
* Fit an optimal classifier from a bag of models that can predict whether the driver is alert or not alert, with highest accuracy.

The nature of the project is purely exploratory and the deliverables includes an analysis of the importance of features and a recommendation of an appropriate model that can be used to make predictions about the alertness of the driver.

The original deliverable included a detailed technical and recommendation report in the form of a word document and the R code, which was used in the analysis, supplied as an attached single R file.

## Review Approach & Scope

The idea of this review exercise is to identify the reproducibility issues and risks that could potentially become issues for reproducing the analysis.

Imagine a situation, where a Data Science team came across the project report and found it useful enough to invest time and resources to implement the project as a deployable model, all they have with them is an analysis that was done 3 years ago and a version of R code which probably ran fine 3 years ago but is not guaranteed to run the same, given there is no information on the environment it was run and the dependencies it had at that time.

The approach of this review is to simulate the analysis using the piece of given R code and identify the imminent issues with reproducibility of the analysis and also, in the process, to identify any key risks that can become reproducibility issues in the near future. That would involve simulating the analysis in more than one environment and different conditions/dependencies. The key idea behind this review is not only to reproduce the analysis but also be able to make it more maintainable and auditable.

The scope of the review is identified the issues and risks from the perspective of:

* Version Control
* Code Readability and Layout
* Coding Standards
* Literate Programming
* Code Portability

The review does not make any recommendations on the appropriateness of the modelling methodology or the model training or validation approach.

## Findings

Each of the findings will talk about the potential issue or risk and a separate section on recommendations on how to mitigate these issues and risks. Examples of each of the issues/risk have been listed in the recommendation section.

**No Source Control Tool**

The source code, documents and data, all together were zipped up and stored on google drive. Although, this is one form of source control but it is not an ideal way to version controlling the source. The project artifacts lacked a proper structure and different versions of code were found, stored as copies of the same file. There is no way of identifying which artifact is the latest version, which raises a concern whether the R code that was finally submitted was even a true copy of the latest version of the code. This is nothing stopping to carry on the analysis if there is no source control, but it is a key risk which in future could cause inconsistencies in the code and leaves you without the ability to undo and redo changes. Also, one loses the ability to track changes over time in case one needs to retrospectively look at the changes to review what worked and what didn’t.

**Inappropriate Central/Remote Repository**

Google Drive is not an ideal remote repository tool for hosting code repositories. The key thing missing in that setting is the ability to collaborate. Remote repositories specifically designed for controlling versions of code not only helps in sharing code among other developer/team members but also are ideal provides features of looking back in time for changes by all users and merging changes from two different users. Some of the code repository providers even have more advanced functionality of continuous integration and continuous deployment of the source. Again, having a central remote repository is not an issue but a key audit risk. Also, if several developers intend to work at the same time, then it will become an issue to consolidate everyone’s changes without investing significant time and effort and the resulting code still might be error prone.

**Missing info on package versions/ R version/ OS**

The project R code was built using the latest version of dependent packages at the time of writing the original code. It has been 3 years since then and today I was not able to re-install one of the packages mentioned in the source code. The dependent packages follow their own development cycle and might have seen various revisions since the original code was written. Fortunately, the package that I was not able to re-install is actually no longer needed in the code. Recording the version of the dependencies is a must and poses an imminent issue in the near future. Moreover, the machine/operating system it was built on has also changed, from a Window 10 OS to a macOS, the code did not travel well. Even though the code ran fine, I was not able to reconcile the results as documented previously to the current results that I am getting. Also, the version of R itself has progressed over the course of 3 years. Versions are important and requires attention of a proper version management tool that can snapshot the current state/version of OS/ dependent packages/ R version.

**Inconsistent Coding Practices**

Inconsistent style made it hard to understand what the different variables are used for and the naming convention had been arbitrary which breaks the flow of the reader/auditor who is trying to understand what the code is supposed to do. A neatly formatted code, consistent naming conventions makes it easier to read and understand the code. Also, as pointed out earlier, there are instances of code where some of the redundant code exists and could have been reviewed and removed from the final version of the code. Even though the code might run, but it presents a huge audit risk and costlier maintenance operations as the reader would have to spend a lot of time understanding the code.

**No particular programming paradigm**

Even though the project was exploratory in nature, there are instances where a functional programming approach could have been used. While it might work fine for sharing ad hoc analysis but it is essential to follow either an object oriented approach or a functional programming approach to build an industry standard solution. Hence, from reproducibility perspective I was able to reproduce the code but it poses a significant risk if the same code is adopted to build an industry standard model.

**Insufficient documentation**

As mentioned earlier, the code lacked functional programming and even where there was a function written, it lacked proper documentation. It was really hard to understand what the function does until you actually execute it. A function is meant to be a reusable piece of code and without proper documentation, a reader will be left with no choice but to write their own version of function, making the existing code redundant. Again, this is not an immediate issue but over time it might pose as an audit risk and the code becoming redundant. Even though the supporting report has some form of documentation, code itself is not self explanatory.

**Key Person Risk**

Only one person knew about the code and other artifacts, the developer. No other person can takeover and run the code independently. It simply cannot be reproduced without considerable effort.

## Recommendations

**Use source control**

Source control tools like *git* for maintaining local repositories and *Github* or *BitBucket* for maintaining remote repositories are highly recommended. It is really easy to reproduce not only the latest code but also have the ability to go back in time and restore an older version of the code. These tools help in collaborating and working in a large team. They provide reviewers/auditor with the capability to compare two versions and make informed decisions based on the differences.

The re-written code has been maintained at github:  
<https://github.com/anuj-kapil/driveralertness>

**Adopt one style of coding**

Adopt one style for coding and consistently use it across the code. Follow one set of naming conventions and avoid giving useless and random names. It makes the code easy to understand.

Sample names and style in old code:

fsModels2 <- c("glm","gbm","treebag","ridge","lasso","rf","xgbLinear")  
myFs2<-fscaret(trainDataset, testDataset, myTimeLimit = 40, preprocessData = TRUE,  
 Used.funcRegPred = fsModels2, with.labels = TRUE,  
 supress.output=FALSE, no.cores = 2, installReqPckg = TRUE)

Sample names and style in new code:

list\_of\_models <- c("glm", "gbm", "treebag", "ridge", "lasso", "rf", "xgbLinear")  
  
feature\_selection\_models <- fscaret(trainDataset, testDataset,  
 myTimeLimit = 40, preprocessData = TRUE,  
 Used.funcRegPred = fsModels2, with.labels = TRUE,  
 supress.output = FALSE, no.cores = 2, installReqPckg = TRUE  
)

Styler can be used in R to style the code

Old Style: (Hard to read)

data<-data[c("TrialID", "ObsNum", "P1", "P2", "P3", "P4", "P5", "P6", "P7", "P8", "E1", "E2", "E3", "E4", "E5", "E6", "E7", "E8", "E9", "E10", "E11", "V1", "V2", "V3", "V4", "V5", "V6", "V7", "V8", "V9", "V10", "V11", "IsAlert")]

New Style: (Easy to read)

data <- data[c(  
 "TrialID", "ObsNum", "P1", "P2", "P3", "P4", "P5",  
 "P6", "P7", "P8", "E1", "E2", "E3", "E4", "E5", "E6", "E7", "E8",  
 "E9", "E10", "E11", "V1", "V2", "V3", "V4", "V5", "V6", "V7",  
 "V8", "V9", "V10", "V11", "IsAlert"  
)]

**Use a package management tool**

It is essential to record the version of the packages used in the R code. Dependent packages are continuously developed and newer versions are often susceptible of breaking the functionality of the code in which they are imported. **packrat** in R is used to maintain a snapshot of the dependent packages loaded in your project. The basic idea is that instead of using the default location to install packages that are shared across all R code, each project gets its own private library of packages. That way, package versions are independent from project to project.

When you want to share a project with other collaborators, you may want to ensure everyone is working with the same environment – otherwise, code in the project may unexpectedly fail to run because of changes in behavior between different versions of the packages in use. You can use **packrat** to help make this possible.

When using **packrat**, the packages used in your project will be recorded into a lockfile, ‘packrat.lock’. Because ‘packrat.lock’ records the exact versions of R packages used within a project, if you share that file with your collaborators, they will be able to use packrat::restore() to install exactly those packages into their own library.

Sample packrat snapshot

# Initialize packrat  
packrat::init(  
 infer.dependencies = FALSE,  
 enter = TRUE,  
 restart = FALSE  
)  
  
# First, Let's install all the packages that are required in the project  
  
# Second, Take snapshot of all the loaded packages  
packrat::snapshot(  
 snapshot.sources = FALSE,  
 ignore.stale = TRUE,  
 infer.dependencies = FALSE  
)

The above code automatically updates the ‘packrat.lock’ file and anyone who hase this ‘packrat.lock’ and **packrat** package installed on their system, can restore the snapshot from ‘packrat.lock’ file.

Sample packrat restore

# Restore packages using the 'packrat.lock' file  
packrat::restore()

**Functional Programming**

Any problem can be broken down into smaller chunks of problems and functions can be written to solve those chunks of problems. Any repetitive code can be converted into a function which helps reduce the chances of introducing errors and makes the program more readable.

Sample repetitive code in old source:  
The below code was written for each of the 33 variables/features to plot their histograms and box plots.

##### Histogram and Box Plots #####  
  
png(height=3000, width=3000, pointsize=40, file="hist\_boxplot\_indvarP1\_P4.png")  
par(mfrow=c(4,2))  
  
hist(data[, 3], main = "Histogram of P1", xlab="Physiological 1", ylab="Frequency", col = "blue")  
boxplot(data[, 3], main = "Boxplot of P1", xlab="Physiological 1", col = "blue")  
  
hist(data[, 4], main = "Histogram of P2", xlab="Physiological 2", ylab="Frequency", col = "blue")  
boxplot(data[, 4], main = "Boxplot of P2", xlab="Physiological 2", col = "blue")  
  
hist(data[, 5], main = "Histogram of P3", xlab="Physiological 3", ylab="Frequency", col = "blue")  
boxplot(data[, 5], main = "Boxplot of P3", xlab="Physiological 3", col = "blue")  
  
hist(data[, 6], main = "Histogram of P4", xlab="Physiological 4", ylab="Frequency", col = "blue")  
boxplot(data[, 6], main = "Boxplot of P4", xlab="Physiological 4", col = "blue")  
  
dev.off()

The repetitive code turned into a function:  
The code looks more cleaner and easier to understand. Provides consistent output on each invocation. Moreover, now there is less scope of introducing errors.

##### Histogram and Box Plots #####  
  
# Function to plot the histogram and box plots   
# Plots for a particular column (x) that exists in the data.frame (dat)  
hist\_box\_plots <- function(dat, columnname) {  
 hist(dat[, columnname], main = paste0("Histogram of ", columnname), xlab = columnname, ylab = "Frequency", col = "blue")  
 boxplot(dat[, columnname], main = paste0("Boxplot of ", columnname), xlab = columnname, col = "blue")  
}  
# List of all variables for which we need histograms and box plots  
variables <- c(  
 "P1", "P2", "P3", "P4", "P5", "P6", "P7", "P8",  
 "E1", "E2", "E3", "E4", "E5", "E6", "E7", "E8", "E9", "E10", "E11",  
 "V1", "V2", "V3", "V4", "V5", "V6", "V7", "V8", "V9", "V10", "V11"  
)  
  
# The following statement are to set the graphics to be displayed in a particular order  
# This below statement tells the graphic engine to display the four images on one page  
layout\_matrix <- matrix(c(1:4), nrow = 2, ncol = 2, byrow = T)  
layout(mat = layout\_matrix)  
  
# This function basically loops all the variables and invokes hist\_box\_plots for each one of them  
# Prints the histogram for all the variables selected above  
lapply(variables, hist\_box\_plots, dat = data)

**Defensive Programming**

Assert statement was introduced to interrupt the code execution early in case of input inconsistencies. This helps in building a fail-fast system and catches errors sooner than later and saves time and computation. It is highly recommended to add assertion in the code as it provides quality assurance and prevents inconsistencies in the results. It is better to error the program than to produce irrelavent results.

The following sample *stopifnot* assertion was added in code:  
This code prevents loading a totally different data saved with the same file name.

##### Reading the file #####  
data <- read.csv("Data/fordTrain.csv",  
 header = TRUE, stringsAsFactors = FALSE, na.strings = c("NA", ""),  
 strip.white = TRUE, blank.lines.skip = TRUE, skip = 0  
)  
  
required\_names <- c(  
 "TrialID", "ObsNum", "P1", "P2", "P3", "P4", "P5",  
 "P6", "P7", "P8", "E1", "E2", "E3", "E4", "E5", "E6", "E7", "E8",  
 "E9", "E10", "E11", "V1", "V2", "V3", "V4", "V5", "V6", "V7",  
 "V8", "V9", "V10", "V11", "IsAlert"  
)  
  
#Check if all the required column names listed above exists in the data that is read, otherwise throw error  
stopifnot(all(required\_names %in% names(data)))

**Documentation**

While creating a function, it is essential to document what that function does and what input it takes and what to expect as an output from function and if possible include some examples of function usage. The **roxygen2** package in R converts such documentation into user manual which is available as function help. All you need to do is, for each of the function, generate a skeleton documentation and fill it will the required information. Once you build the package or generate the documentation using **roxygen2** package, **roxygen2** reads all the skeleton information and generates a manual for each of the function. This is usually what you see when you invoke function names preceded by a question mark (Example: ?sum)

A sample documentation is shown below:

#' Histogram and Box Plots Combined  
#'  
#' @param dat The data.frame containing the data  
#' @param columnname The variable in the data.frame whose plots are required  
#'  
#' @return NULL  
#' @export  
#'  
#' @examples  
#' None recorded  
hist\_box\_plots <- function(dat, columnname) {  
 hist(dat[, columnname], main = paste0("Histogram of ", columnname), xlab = columnname, ylab = "Frequency", col = "blue")  
 boxplot(dat[, columnname], main = paste0("Boxplot of ", columnname), xlab = columnname, col = "blue")  
}

**Automate unit-tests**

While writing functions, a developer often tests the functions with some sample inputs. Record those sample tests into a function and run them every time someone makes a change to the function. This ensures no one breaks the existing functionality of the function.

Sample test:

# Test to check if the data is loaded fine  
test\_that("Data loading", {  
   
 expect\_true(exists("data"))  
   
})

**Reproducibility**

Containers are a piece of software that mimics a computer. When run on a physical computer, they can be thought of as a computer inside a computer. They can be thought of as virtual machine running on top of an actual machine but instead of virtualizing the hardware stack, containers virtualize at the operating system level, with multiple containers running atop the OS kernel directly. Containers offer a logical packaging mechanism in which applications can be abstracted from the environment in which they actually run. This means developers can focus on developing the application without worrying about the underlying operating system level dependencies. Containers provides consistent environments, runs virtually everywhere and runs in isolation.

While **packrat** took care of all the R dependencies on other packages, containers takes care of other dependencies on operating systems.  
For our project, I have created two docker images. Docker image with the snapshot of all the dependent packages was created first using the base image of rocker/studio. rocker/studio has all the necessary libraries to run R and RStudio IDE, on top of which I have installed all the dependencies required for our project. Earlier, we took a snapshot of all the dependent packages using packrat::snapshot() and used packrat::restore() to build a docker image with all our projects’ R dependencies installed.

# Only run this after making packrat/packrat.lock by  
# running manage\_packages.R  
  
# Pull the base image that contains R and RStudio  
FROM rocker/rstudio:3.6.1  
  
# zlib is an external dependency for one of our dependent packages  
RUN apt-get update && apt-get install zlib1g-dev  
  
# Copies the packrat.lock we had generated using packrat snapshot onto docker container  
COPY ./packrat/packrat.lock packrat/  
  
# Installs packrat into docker container  
RUN install2.r packrat  
  
# Installs projects R dependencies into docker container  
RUN Rscript -e 'packrat::restore()'  
  
# Modify Rprofile.site so R loads packrat library by default  
RUN echo '.libPaths("/packrat/lib/x86\_64-pc-linux-gnu/3.6.1")' >> /usr/local/lib/R/etc/Rprofile.site

Now we have an image onto which all our project’s R dependencies have been baked.

Using this image as a base image, I have created another docker image which has our project’s code baked into it. The second image has been configured in GitHub continuous integration workflow to regularly update with our project package on each push to the repository.

# Pull the base image that contains R, RStudio and R dependencies  
FROM anujkapil/packratmanaged:1.1  
  
# Add the analysis file to the docker image  
ADD R/main.R /home/rstudio/packtest/R/  
ADD Data/fordTest.csv /home/rstudio/packtest/Data/fordTest.csv  
ADD Data/fordTrain.csv /home/rstudio/packtest/Data/fordTrain.csv  
ADD RMD/main.RMD /home/rstudio/packtest/RMD/  
ADD RMD/Data/fordTest.csv /home/rstudio/packtest/RMD/Data/fordTest.csv  
ADD RMD/Data/fordTrain.csv /home/rstudio/packtest/RMD/Data/fordTrain.csv  
ADD RMD/main.docx /home/rstudio/packtest/RMD/  
ADD RMD/main.md /home/rstudio/packtest/RMD/

The code for the GitHub workflow can be found here:  
<https://github.com/anuj-kapil/driveralertness/blob/master/.github/workflows/dockerimage.yml>

This ensures that every time there is a change to the project application, the docker image gets refreshed with the newly pushed changes. This is called continous integration.

Alternate approach can be configured, using just one dockerfile, to manage any updates to dependent packages in our project. Let’s assume someone adds a dependent package in our project, then two docker images solution won’t work as our first Docker image is a static snapshot of all the dependent packages that our project needs today. This defeats the purpose of using **packrat** as on each build of project package, **packrat** takes a new snapshot of the dependent packages which isn’t getting updated in the project’s first Docker image. But a single docker image can be configured to do both the snapshot restore as well as latest project package on each push to the repository.

Also, the GitHub actions could be configured to run the testthat tests on the project repository to unit test the project source code.

The re-written code can be found here:  
<https://github.com/anuj-kapil/driveralertness/blob/master/R/main.R> <https://github.com/anuj-kapil/driveralertness/blob/master/RMD/main.RMD> <https://github.com/anuj-kapil/driveralertness/blob/master/RMD/main.md>

The code has been re-written using R-Markdown that binds together the report and code, which were two separate artefacts originally. R-Markdown is a better approach of narrating your analysis alongwith the snippets of code. With dockerised version of the shareable R-Mardown code, any one can reproduce the same analysis.

The code to run the docker container for the re-written code is:

docker run -e PASSWORD=lizard --rm -p 8787:8787 anujkapil/driver-alertness-analysis:1.1

An attempt was made to re-write the code as an R package and deploy an API to validate the model. However, due to issue with R package creation and issue with installing the package on docker, I couldn’t reach to the point where an API could have been written. Well, all the code that gets written today, becomes a technical debt tomorrow.