R for SAS Programmers

Michael Walshe

# Setting the Stage

Are you a SAS programmer picking up R? Have you struggled with missing values, out of memory errors, or just how to switch your brain from SAS to R? Then this blog post is for you!

If you’re familiar with SAS, you know that it’s an excellent language for high-performance analytics. However, many industries are exploring R alongside SAS for its open-source flexibility and expansive community, building on their existing statistical know-how. So for SAS veterans looking to add another weapon to their analytical arsenal, today we’re going to teach you how to navigate the world of R - highlighting the similarities, smoothing out the bumps, and you’ll be a practiced “useR” before you know it!

One thing to note here is what we will *not* cover - this will not be a complete introduction to R! What I’m going to give is an overview of R for the SAS programmer. For a detailed introduction to R from the fundamentals up, take a look at our [R courses](https://katalyzedata.com/courses/r-training/). If you’re looking for a course aimed at SAS programmers, then make an enquiry [here](https://katalyzedata.com/training/learn-r-for-sas-programmers/).

# A Brief History of R

R first appeared on the scene in 1993, although the syntax and style is heavily inspired by a predecessor S - developed by Bell Labs in 1976. It was started by two professors at the University of Auckland, to aid in using and teaching statistics. This has led to a common phrase: that R is “for statisticians, by statisticians”. Since then, there have been several major milestones:

* 1993: First public announcement of R
* 1997: R became a GNU project, and is released as free and open-source
* 2000: R version 1.0 was released
* 2011: RStudio first released, a very popular IDE for R
* 2018: Pharmaceutical companies start to work together to use R for clinical trials, showing that open-source software can be used and foster collaboration even in heavily regulated industries

One of the major differences between R and SAS is that in R a lot of the functionality and fun features are provided by the user community in the form of **packages**. These are portable bundles of code, documentation, and data, which are distributed via [CRAN](https://cran.r-project.org/). An excellent feature of CRAN compared to other open-source package repositories is that every package is checked and curated by a human alongside a suite of automated checks – currently there are >20’000 curated packages available!

# Diving In

Let’s get started by looking at some of the basic operations in R, as compared to SAS. In SAS, the essential unit of data is the *dataset*, composed of *observations* and *variables*. In R, the the essential atomic structure is the vector, a homogeneous sequence of elements (e.g. c(1, 2, 3, 4) or c("A", "C", "A")). The next level up is the data.frame, which analogously to SAS’s dataset is a tabular data format with *rows* and *columns*.

Before we get bogged down in minutia, let’s look at an example of doing some basic processing of a dataset in SAS, and then R. We’ll use the following demog data set for both:

head(demog)

#> # A tibble: 6 × 15  
#> age gender height weight status children cars salary grade hours dept   
#> <dbl> <chr> <dbl> <dbl> <chr> <dbl> <dbl> <dbl> <chr> <dbl> <chr>  
#> 1 22 M 64 60 S 0 1 13592. low 35 A   
#> 2 26 F 87 74 M 1 1 8870. low 35 B   
#> 3 29 M 55 62 D 2 1 12672. low 35 C   
#> 4 28 M 84 74 M 3 1 23761. high 35 C   
#> 5 23 F 84 88 P 1 1 10513. low 35 C   
#> 6 30 M 87 76 S 0 1 7521. low 25 D   
#> # ℹ 4 more variables: overtime <dbl>, cc <dbl>, staffno <chr>, budget <dbl>

* **Using SAS:**

%let rc = %sysfunc(dlgcdir(%nrstr(%%USERPROFILE%%)\source\katalyze-data\content\r-for-sas-programmers-blog));  
  
proc import file="data/demog.csv" out=work.demog replace;  
run;  
  
data demog2;  
 set demog(keep=height weight salary gender);  
 where gender = "M";  
 height\_m = (height \* 2.54) / 100;  
 bmi = weight / (height\_m \*\*2);  
run;  
  
proc corr data=demog2;  
 var bmi salary;  
run;

Line 3

Read in our CSV using the import procedure, and save it to a dataset called demog.

Line 6

Use a data step to edit our dataset, first filtering to just observations where gender is equal to "M" and only keeping the variables height, salary, weight, and gender on the input.

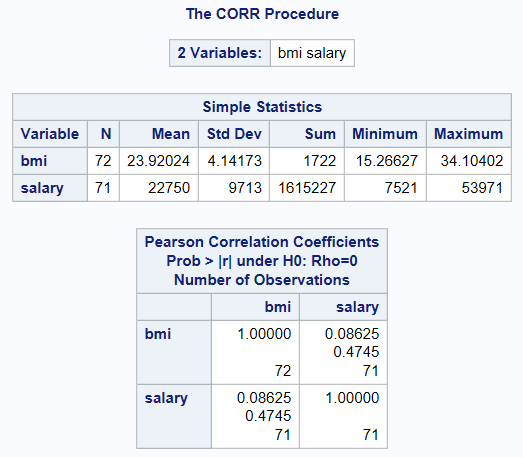
Line 9

Calculate our new BMI column, using an intermediate height\_m column.

Line 13

Compute the correlation between BMI and salary, using the corr procedure.

This produces the following output:



* **Using R:**

demog <- read.csv("data/demog.csv")  
  
demog2 <- demog[  
 demog$gender == "M",  
 c("height", "weight", "salary")  
]  
  
demog2$height\_m <- (demog2$height \* 2.54) / 100  
demog2$bmi <- demog2$weight / (demog2$height\_m \*\*2)  
  
result <- cor.test(  
 ~ bmi + salary,  
 data = demog2,  
 na.action="na.omit"  
)

Line 1

Import our CSV to a dataframe, using the read.csv function, and saving it in the *variable* demog. This is equivalent to our import procedure in SAS, though for different file formats there are many different read functions. Note also that to create a variable we use an assignment operator: <-.

Line 3

Filter to just the rows where gender is equal to "M" and select just 3 columns. We achieve this by indexing using square brackets, passing in a logical expression to filter the rows, and then a vector of column names to select some columns.

Lines 8-9

Add two new columns to our dataframe: height in metres and BMI. This all happens in open code, not in a datastep! We can refer to columns using dataframe$column.

Line 11

Compute the correlation between bmi and salary, using the cor.test function. This uses a **formula** of the form ~ bmi + salary, a nice way in R of defining relationships between variables for statistics and modelling. We save the output of cor.test to a variable result.

This produces the following output:

print(result)

#>   
#> Pearson's product-moment correlation  
#>   
#> data: bmi and salary  
#> t = 0.71916, df = 69, p-value = 0.4745  
#> alternative hypothesis: true correlation is not equal to 0  
#> 95 percent confidence interval:  
#> -0.1500696 0.3132541  
#> sample estimates:  
#> cor   
#> 0.08625406

**So what’s the difference?**

We can already see a few of the key differences, but let’s enumerate the most important ones:

* **No data steps.** In SAS, the main way to do data wrangling and calculations is in a data step, which is an implicit loop over all observations in your data set. In R, instead we perform *vectorised* calculations on an entire column. You can manually loop over vectors and dataframes in R, but it’s generally not required.
* **No procedures.** If we’re not using a data step in SAS, then we’re probably using a PROC. In R, we use functions. These are similar in some ways - as packaged units of code with various keywords and parameters. However, functions are both more flexible and less all-encompassing than procedures.
* **Different outputs.** In SAS, we received a printed output of different tables and statistics, by default quite a lot of output! In R, we received as output a **list** that we stored in the variable result. A list in R is a heterogeneous sequence of elements, and is often the way that statistical outputs are returned. We could access different elements of our list using the $ operator similarly to a dataframe (e.g. result$p.value ⟹ 0.4744709), or we can just print the list to see a specially summarised output of the key statistics.
* **Missing Values.** In SAS, missing values are represented by either . or by a blank string "". In R, missing values are represented by NA (i.e. **N**ot **A**vailable). We can see that in our cor.test function, we specified na.action="na.omit" to remove missing values from our calculation, in SAS missing values are automatically omitted from most functions and procedures.

# Putting R to Work

## Data Manipulation in the {tidyverse}

What we’ve seen so far is using just the functionality available in so-called “Base R”, without any extra packages installed. This isn’t typical, most users will install a variety of packages to aid in their work. One of the most popular sets of packages is the [{tidyverse}](https://www.tidyverse.org/), which is a collection of packages that share a common philosophy and style. Let’s load the {tidyverse} and see how it changes our workflow.

# Loading just two packages from the tidyverse  
library(readr) # For better file reading  
library(dplyr) # For better data wrangling  
  
result <-   
 read\_csv("data/demog.csv") |>  
 filter(gender == "M") |>  
 select(height, weight, salary) |>  
 mutate(height\_m = (height \* 2.54) / 100,  
 bmi = weight / (height\_m \*\*2)) |>   
 cor.test(~ bmi + salary,  
 data = \_,  
 na.action = "na.omit")

I won’t go through this line-by-line, but we’ve achieved the same result as before, this time writing {tidyverse} style code. This style is characterised by a few key features:

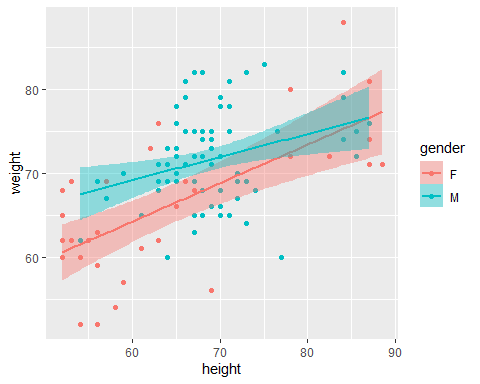
* Using the pipe operator |> to chain together functions, making the code more readable and easier to follow. Otherwise, this would look something like: mutate(filter(select(read\_csv("file"), ...), ...), ...).
* Using functions with a consistent naming style, and consistent arguments. This makes it easier to remember how to use functions, and to learn new ones.
* Using lazy evaluation, which is a way of referring to columns without putting them in quotes (i.e. "height") or using the $ operator

These functions may seem familiar to those of you used to SQL (available through PROC SQL in SAS). This is no coincidence, the {dplyr} package was partially inspired by SQL, and is a great way to do data wrangling in R. Our output is the same as before, just produced in a (in my opinion) more readable way.

## Plotting in R and SAS

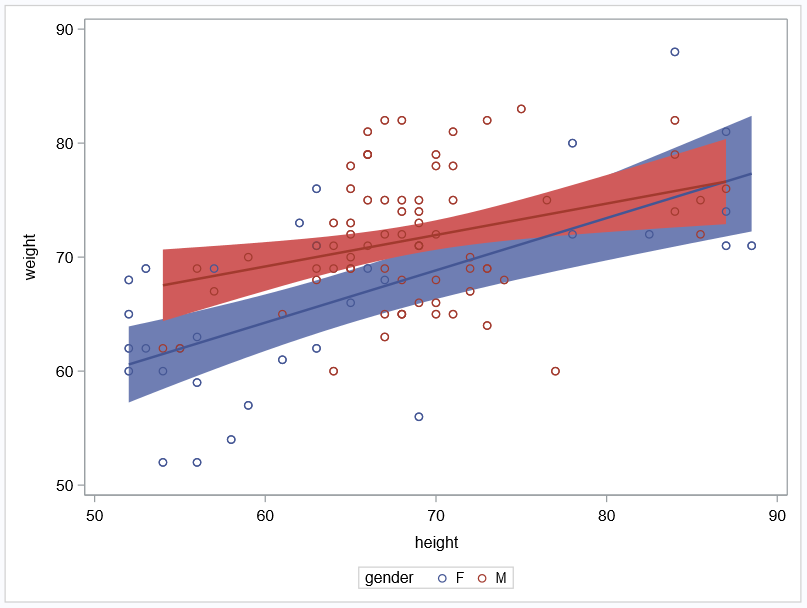
Now that we’ve seen a little more of the basics of data manipulation in R, let’s take a look at plotting. Here, we’ll create a simple plot of the demog dataset from before using the immensely popular [{ggplot2}](https://ggplot2.tidyverse.org/) package. This package helps us create plots by layering different components together, a similar approach to the proc sgplot procedure in SAS.

library(ggplot2)  
  
ggplot(demog, aes(height, weight, colour=gender, fill=gender)) +  
 geom\_point() +   
 geom\_smooth(method="lm")



If we wanted to create a similar output in SAS, we could do:

proc sgplot data=demog;  
 scatter x=height y=weight / group=gender;  
 reg x=height y=weight / group=gender clm;  
run;



Here we can see that the {ggplot2} code is a little less verbose, and also as we build up more complex plots it’s easier in R to add different layers and components. R also has a robust ecosystem of tools for producing reports and dashboards - for example this blog post was written in [Quarto](https://quarto.org/)!

# Integration with SAS and Common Pitfalls

## How do I make R SASsy?

Integration between SAS and R is a topic deserving of an entire blog post ([let us know](mailto::info@katalyzedata.com) if you’d like to see one!), but here we’ll just cover the basics. If you’re looking to talk to with SAS *from* R, there are a few options:

* **Reading in SAS datasets.**. The [{haven}](https://haven.tidyverse.org/) package provides functions to read in SAS datasets, including formats and labels.
* **Calling SAS from R.** The [{swat}](https://github.com/sassoftware/R-swat) package provides us a way to call SAS from R, specifically the CAS server in Viya. If you’re looking to call standard SAS, you could also use the [SASPy](https://support.sas.com/en/software/saspy.html) Python package via the R package [{reticulate}](https://rstudio.github.io/reticulate/), which integrates R and Python

If you’re looking to talk to R from SAS, then the main way is via PROC IML. This is a SAS procedure for performing matrix and vector operations, and includes functionality to call R functions or scripts.

## Watch your step!

One of the most common pitfalls for SAS programmers learning R is the different mode of thinking about data. In SAS, we’re used to thinking about looping over observations, and performing calculations on each one. In R, generally the best method is to perform a *vectorised* calculation over an entire column. If you do try to perform calculations in a loop, it can be slower if you don’t prepare correctly, and will often be much slower if the vectorised method is implemented in a faster language (such as C or C++, as many functions are in R).

Another common mistake that we’ve already seen is the difference in missing values. More than just the representation and treatment in functions, however, is the difference during comparisons. In SAS, a missing value is treated as negative infinity for comparison (e.g . < anything is always true). In R, a missing value during comparison will propagate! So NA < anything is NA. This can be a source of bugs if you’re not careful.

The final pitfall that I’ll mention here is that of data sizes. In SAS, datasets of any size can be manipulated on any system without issue, as long as you have enough disk space. In R, however, all data is loaded into memory and is limited by available RAM. This means that if you’re working with very large datasets, you may need to be careful about how you load and manipulate them. There are a variety of packages that can help with this, two that I’ll demonstrate here are [{arrow}](https://arrow.apache.org/docs/r/) alongside [{duckdb}](https://duckdb.org/docs/api/r).

We’ll use a large dataset of taxi trips in New York City. I’ve downloaded the first 4 months of 2016, which is approximately 8GB as a CSV! This is just about manageable for a modern laptop, but it would quickly get painful to analyse any more data, let alone the full history.

If you want to take a look at the code to download it and benchmark some basic manipulation using the standard tidyverse, expand the section below. For now, we’ll just note the time taken to do some calculations and aggregations.

# Download and unpack the data  
library(arrow)  
library(aws.s3) # n.b. need to set environment variable with AWS account details  
library(dplyr)  
library(fs)  
library(lubridate)  
library(purrr)  
library(readr)  
library(stringr)  
library(tibble)  
  
# Download each file if it doesnt already exist  
downloaded\_csvs <-   
 get\_bucket\_df("s3://nyc-tlc/", region="us-east-1") |>   
 as\_tibble() |>   
 filter(str\_detect(Key, r"(^csv\_backup/yellow\_tripdata\_2016-0[1-4].csv?)")) |>   
 pull(Key) |>  
 map\_chr(  
 \(x) {  
 fpath <- file.path("data", "nyctaxi", x)  
 if (!file\_exists(fpath)) {  
 save\_object(  
 object = x,  
 bucket = "s3://nyc-tlc/",  
 file = fpath  
 )  
 }  
 fpath  
 }  
 )  
  
# Convert all the CSVs to parquet, for later duckdb manipulation  
open\_dataset("data/nyctaxi/csv\_backup", format="csv") |>  
 mutate(month = month(tpep\_pickup\_datetime)) |>  
 group\_by(month) |>   
 write\_dataset("data/nyctaxi/nycp", format="parquet")  
  
# Load the data in R lazily  
# fread is typically faster than readr::read\_csv, but lazy representations  
# can make up the difference!  
nyc\_taxi <- read\_csv(  
 downloaded\_csvs,  
 id="file\_name",  
 lazy=TRUE,  
 col\_types = cols(  
 tpep\_pickup\_datetime = col\_datetime(format = "%Y-%m-%d %H:%M:%S"),  
 tpep\_dropoff\_datetime = col\_datetime(format = "%Y-%m-%d %H:%M:%S"),  
 store\_and\_fwd\_flag = col\_character(),  
 .default=col\_double()  
 )  
)  
  
tictoc::tic("Using standard methods")  
  
agg\_taxi <- nyc\_taxi |>  
 filter(VendorID == 2) |>   
 mutate(  
 # Compute the "manhatten" distance between two points... :P  
 trip\_manhatten\_distance = (  
 abs(dropoff\_longitude - pickup\_longitude) +  
 abs(dropoff\_latitude - pickup\_latitude)  
 ),  
 # Calculate trip duration, in seconds  
 trip\_duration = as.duration(tpep\_pickup\_datetime - tpep\_dropoff\_datetime)  
 ) |>   
 group\_by(passenger\_count) |>   
 summarise(  
 avg\_trip\_distance = mean(trip\_manhatten\_distance, na.rm=TRUE),  
 avg\_trip\_duration = mean(trip\_duration, na.rm=TRUE),  
 avg\_fare\_amount = mean(fare\_amount, na.rm=TRUE)  
 )  
  
tictoc::toc()  
  
rm(agg\_taxi)

#> Using standard methods: 432.22 sec elapsed

Now let’s demonstrate doing the same manipulation with {duckdb}:

library(arrow)  
library(duckdb)  
library(tictoc)  
  
tic("Using duckdb & arrow") # Basic timing  
  
agg\_taxi2 <- open\_dataset("data/nyctaxi/nycp") |>   
 to\_duckdb() |>   
 filter(VendorID == 2) |>   
 mutate(  
 trip\_manhatten\_distance = (  
 abs(dropoff\_longitude - pickup\_longitude) +  
 abs(dropoff\_latitude - pickup\_latitude)  
 ),  
 trip\_duration = datediff("s", tpep\_pickup\_datetime, tpep\_dropoff\_datetime)  
 ) |>   
 group\_by(passenger\_count) |>   
 summarise(  
 avg\_trip\_distance = mean(trip\_manhatten\_distance, na.rm=TRUE),  
 avg\_trip\_duration = mean(trip\_duration, na.rm=TRUE),  
 avg\_fare\_amount = mean(fare\_amount, na.rm=TRUE)  
 ) |>  
 collect()  
  
res <- toc()

#> Using duckdb & arrow: 9.91 sec elapsed

I won’t walk through the code - just notice that it was 43 times faster, all without running out of memory! This is another topic that deserves a blog post of it’s own, and it’s worth saying that this isn’t the only method for working with large datasets in R.

# What’s next?

This has introduced you to some common functionality in R, and how it compares to SAS. If you’re looking for a fully featured introduction to R, either from base principles or focused on advanced concepts, then take a look at our [R courses](https://katalyzedata.com/courses/r-training/). If you’d be interested in an R course aimed at SAS programmers, with more tips and tricks for those transitioning, then make an enquiry [here](https://katalyzedata.com/training/learn-r-for-sas-programmers/).