PhUSE US Connect 2018

Paper DV07

A Gentle Introduction To R Graphics From A SAS Programmer's Perspective

Nate Mockler, Biogen, Cambridge, USA
Saranya Duraisamy, Biogen, Cambridge, USA

ABSTRACT

Like a good craftsman being able to use a variety of tools, a skilled statistical programmer should know many techniques for visualizing complex clinical data. While SAS has impressive graphical capabilities, it is prudent to have another tool in your toolbox. R – an open source language, has made enormous strides in the past few years to make publication-quality graphs. This talk will serve as introduction of R's graphics capabilities, the grammar of graphics that serves as the basis of the popular "ggplot2" capability, and interactive dashboards using R/Shiny. This will be discussed from the perspective of a SAS programmer wanting to learn R as an additional language.

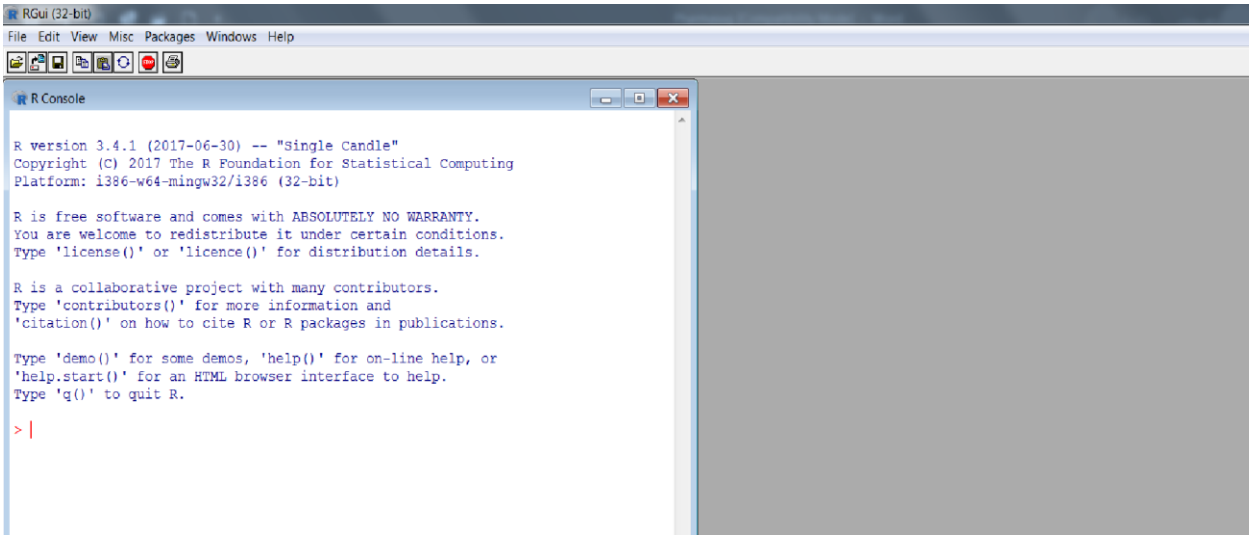
INTRODUCTION

No carpenter would want to be stuck with 1 tool; no band would want to be solely with 1 instrument. A well-rounded statistical analyst should be familiar enough with both languages in order to work with whatever problem comes their way. Especially with the rise of "big data", and "machine learning", at least having a base level knowledge means that if an opportunity comes, you can quickly dive in.

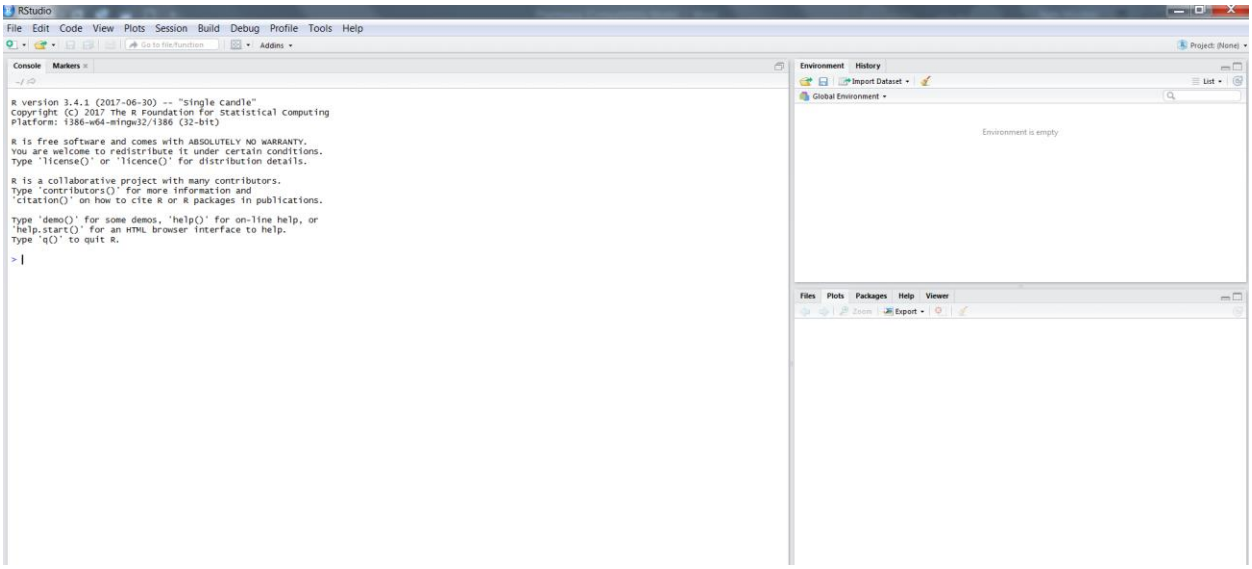
TALK LIKE A PIRATE DAY

R is an open-source language. That means not only is it free as in beer, it's free as in speech. You could go, get the raw code and compile it yourself, not that I recommend that. It's a collaborative enterprise between academics, people in industry, and hobbyist determined to enhance the science of analytics. CRAN (The Comprehensive R Archive Network) is the home of the binaries, and the thousands of packages contained within.

Looking at the command-line can be intimidating, especially for people who have grown up with the comfort of the SAS Enhanced Editor. Fortunately (this will be a pattern), someone has realized this problem and come up with a solution. There are numerous GUIs that will allow you to view workspaces, submit code, and debug much easier. The most popular one at this time is RStudio, which is under very active development. Plus, it is free and open-source as well:



Output 1: Command line! Ahh, scary!



Output 2: Phew, that's better.

One of the biggest differences between SAS and R is that R is an object-orientated language (Technically, SAS has an object-oriented part, but that's going to get complicated). SAS is more of a procedural language. In SAS, the only object that you interact with is a dataset. The dataset is comprised of observations and variables. Every PROC interacts with a dataset, and if you want to manipulate the output, you're going to have to output it to a dataset. In R, that's not the case.
There are vectors, matrices, lists (that can contain lists within it), and the closest analogue to a dataset, the data.frame. We'll be using these mostly for the remainder of this, but the ecosystem of R is much larger than SAS.

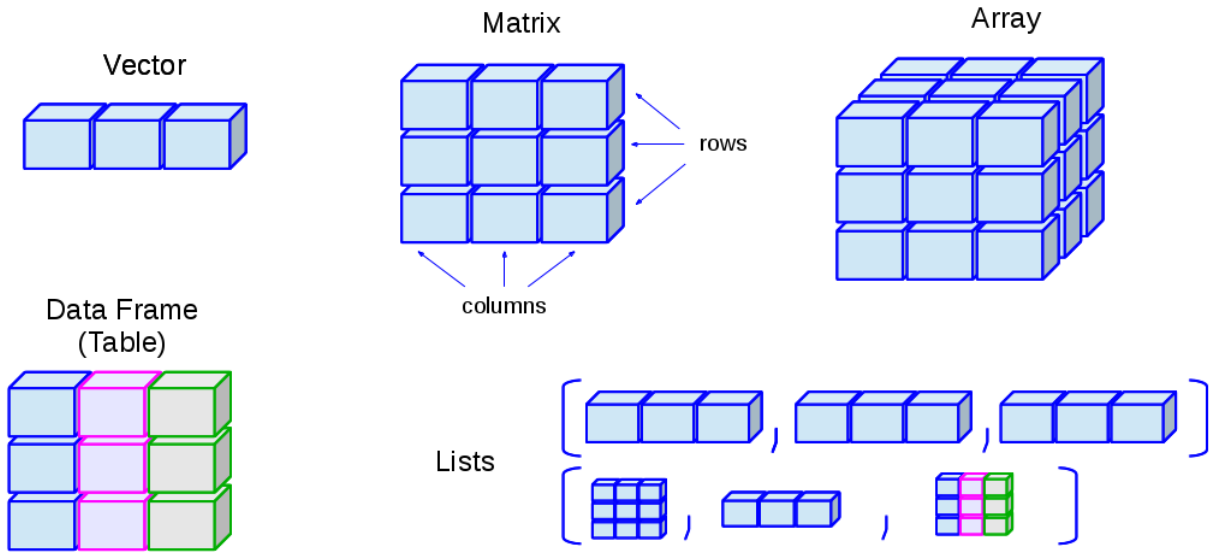


Figure 1: Objects in R (Each color corresponds to a type)

Figure 2: Objects in SAS (That's a dataset)

THE LIFE-CHANGING MAGIC OF TIDYING UP

Of course, since R is open-source, it gives the opportunity for individuals to create packages that greatly extend the functionality. That leads us to Hadley Wickham, chief scientist at RStudio, and professor at Auckland University. One article called him "The Man who Revolutionized R"
https://priceonomics.com/hadley-wickham-the-man-who-revolutionized-r/
His compilation of packages is called the "tidyverse". Some examples of this are:

- ggplot2, for data visualisation.
- dplyr, for data manipulation.
- tidyr, for data tidying.
- readr, for data import.
- purrr, for functional programming.
- tibble, for tibbles, a modern re-imagining of data frames.

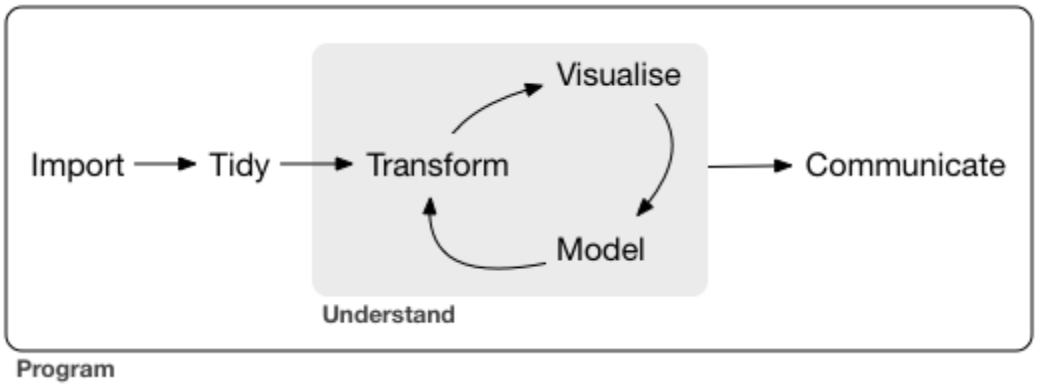


Figure 3: The Project Loop – tidy style (Wickham, 2017)

Wickham describes Tidy data as:
Tidying your data means storing it in a consistent form that matches the semantics of the dataset with the way it is stored. In brief, when your data is tidy, each column is a variable, and each row is an observation. Tidy data is important because the consistent structure lets you focus your struggle on questions about the data, not fighting to get the data into the right form for different functions. (Wickham, 2017)

Of course, coming from SAS, most of our data is in this format, but dealing with data such as JSON, XML, web scraping, etc won't be, so we need to get the data in this format.
Some of these packages that we'll explore are critical in getting the data in the format to analyze, and most R users these days use at least one of his packages. The packages we'll discuss are:
Tidyr: Just in case you missed PROC TRANSPOSE, it's here, with commands such as gather() and spread()
Ggplot2: We'll talk about this more in the next section, but it allows you to create complicated visualizations that you are unable to in SAS or even base R
Dplyr: Allows you subset observations, and rows, summarize, make new variables and combine data sets.

One of the best things about the tidyverse is the pipe, which allows you to string functions together without having to a) Create some nesting doll nightmare of parentheses or b) Create a lot of intermediate steps that can make the code unreadable. The key is that the object on the left is the first/only argument on the right
For example,

```
iris %>%
group_by(Species)
```

is equivalent to:

```
group_by(Iris, Species)
```

See how that's more readable? The benefits start to become more evident once you string multiple ones together.

```
car_data <-
  mtcars %>%
  subset(hp > 100) %>%
  aggregate(. ~ cyl, data = ., FUN = . %>% mean %>% round(2)) %>%
  transform(kpl = mpg %>% multiply_by(0.4251)) %>%
  print
```

Is equivalent to:

```
car_data <-
  transform(aggregate(. ~ cyl,
                      data = subset(mtcars, hp > 100),
                      FUN = function(x) round(mean(x, 2))),
            kpl = mpg*0.4251)
```

Which is much harder to read and debug.

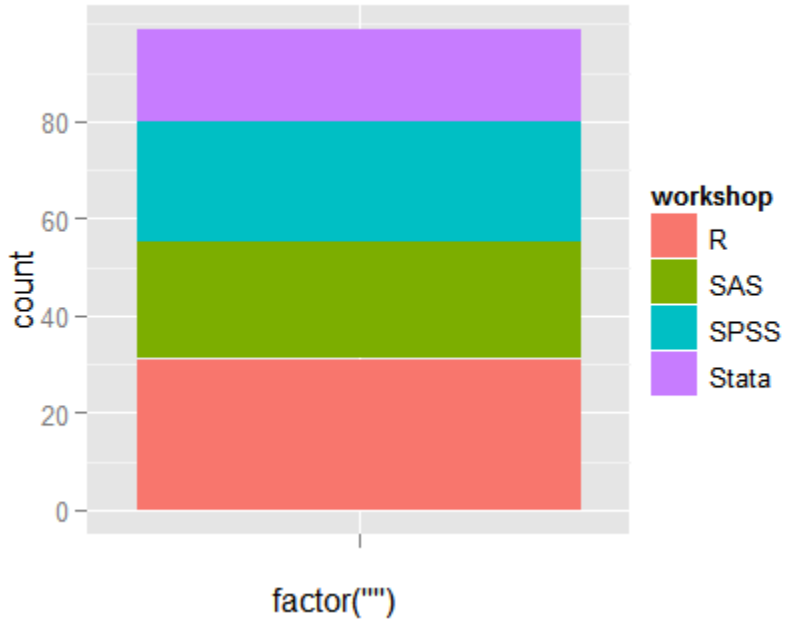
One of the more one-to-one equivalents are PROC TRANSPOSE. This is done by the spread and collapse function.

If you're a fan of PROC SQL for joining, you're in luck, since dplyr has various join functions:
Left_join (a, b, by = "x1") is equivalent to PROC SQL;
Select * from a left join b on a.x1 = b.x1. The other types of joins (right, inner, outer) work accordingly.

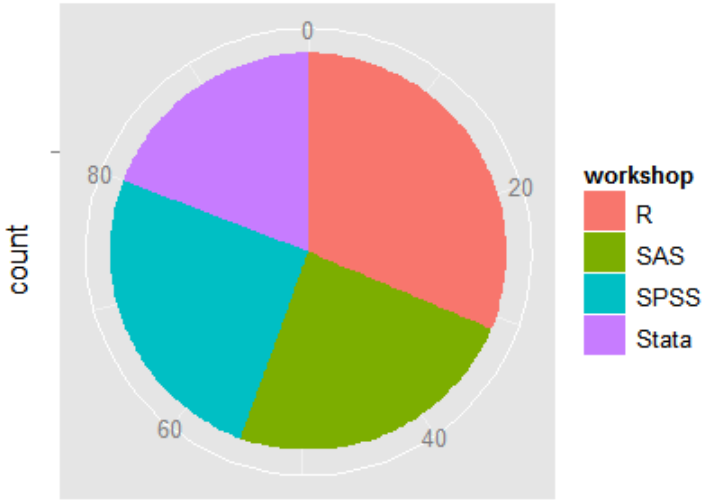
THE PLOT THICKENS

Now onto what many consider R's greatest strengths, its ability to do data visualization. While SAS can do a lot of visualizations, it can be limited even with using GTL. R has primarily 3 graphics packages: base, lattice, and ggplot2. For this exercise, we're going to focus on ggplot2.

GGPLOT2 is a package part of the tidyverse based on the "Grammar of Graphics". The grammar of graphics basically puts all visualization in a common language. It allows you to take a plot such as this



Output 3: A Stacked Bar Chart



Output 4: A Pie Chart; we're not so different, you and I

What the grammar of graphics shows that these are the exact same plot, and to change this only take 1 option change (from linear to a polar scale). An exploration of the grammar of graphics is beyond the scope of this paper, but I recommend exploring the ggplot2 documentation. The concept behind ggplot2 is based on stacking elements on top of each other, like a sandwich:

Figure 4: Delicious and Illustrative!

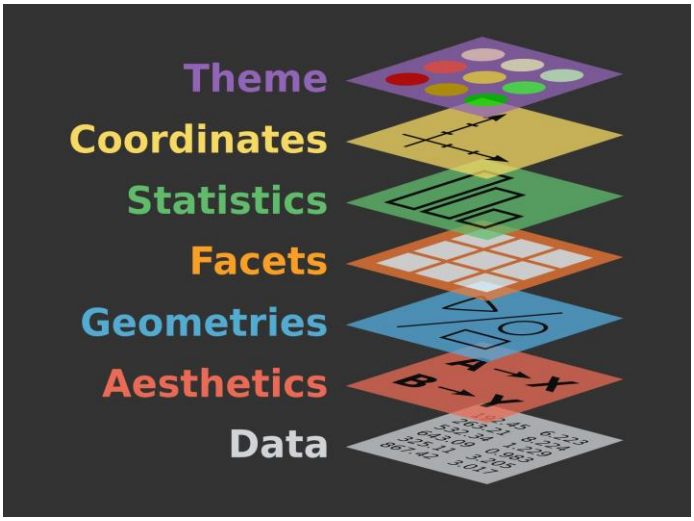
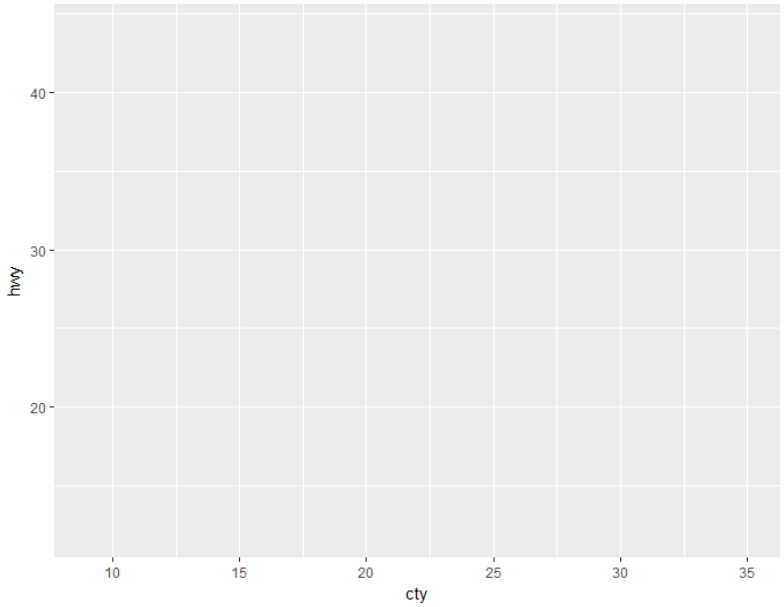


Figure 5: Less edible, but slightly more informative.

So now we're going to stack a graph together. This is in contrast to SAS, where all your options are done at once. Let's start with a basic graph:

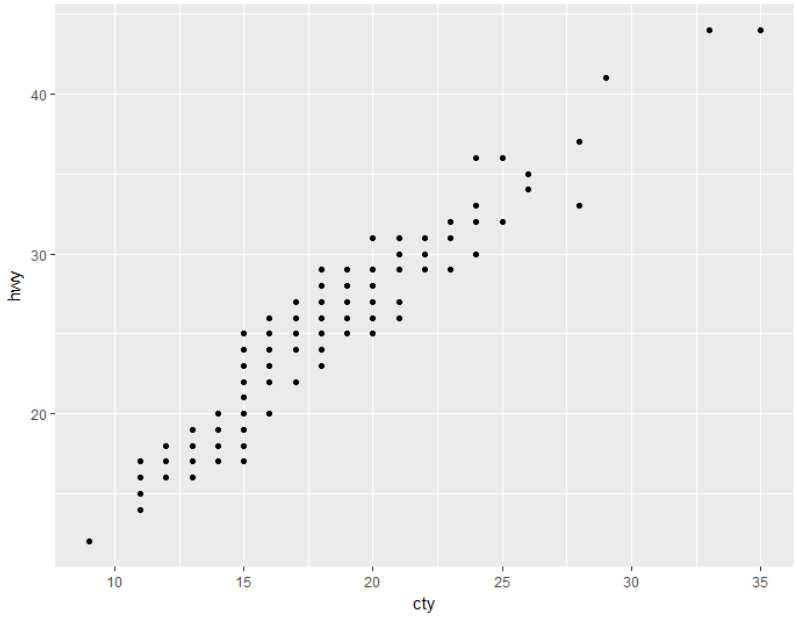
```
g <- ggplot(data = mpg, aes(x = cty, y = hwy))
```



Output 5: Not Much to look at, yet…

Easy enough, right? This creates a graph object that we can modify. Now let's add points:

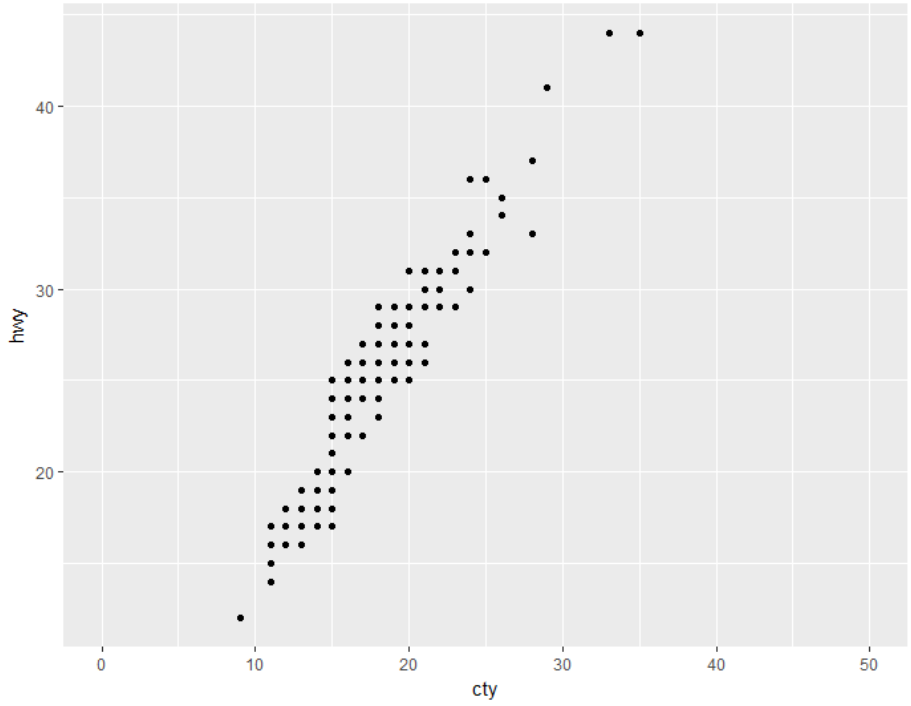
```
g <- ggplot(data = mpg, aes(x = cty, y = hwy)) + geom_point()
```



Output 6: Getting There….

Now let's say we want to modify the x axis to be from 0 to 50. Rather than modify the original statement, we add a scale statement:

```
g <- ggplot(data = mpg, aes(x = cty, y = hwy)) + geom_point() +
scale_x_continuous(limits = c(0,50))
```

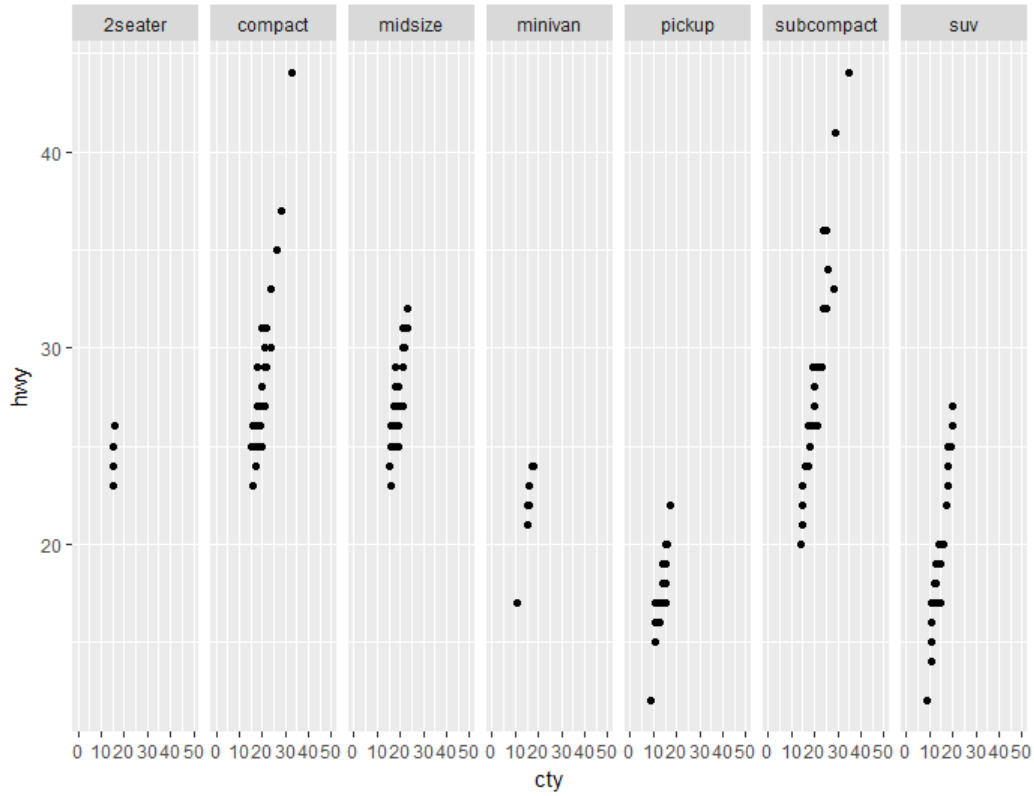


Output 7: Almost….

Okay, that sounds good. Now let's say we want to create a separate graph (the equivalent of a by statement) for each type (the variable is class) of car. We can stack on that stack, and do:

```
g <- ggplot(data = mpg, aes(x = cty, y = hwy)) + geom_point() +
scale_x_continuous(limits = c(0,50)) + facet_wrap(. ~ class)
```

To get our final graph:



Output 8: Finally, our graph!

There is also work we can do with themes, legends, etc… using the same concepts. Please see the documentation for more information.

WHAT MAKES A GOOD DASHBOARD?

Before starting a new dashboard, we should explore quickly what makes a good dashboard. To do that, let's first explore what makes a *bad* dashboard. One of the thought leaders in the area of information is Stephen Few, and his company Perceptual Edge. One of his white papers *Common Pitfalls in Dashboard Design* lists 13 pitfalls (Few, 2018):

- Exceeding the boundaries of a single screen
- Supplying inadequate context for the data
- Displaying excessive detail or precision
- Expressing measures indirectly
- Choosing inappropriate media of display
- Introducing meaningless variety
- Using poorly designed display media
- Encoding quantitative data inaccurately
- Arranging the data poorly
- Ineffectively highlighting what's important
- Cluttering the screen with useless decoration
- Misusing or overusing color
- Designing an unappealing visual display

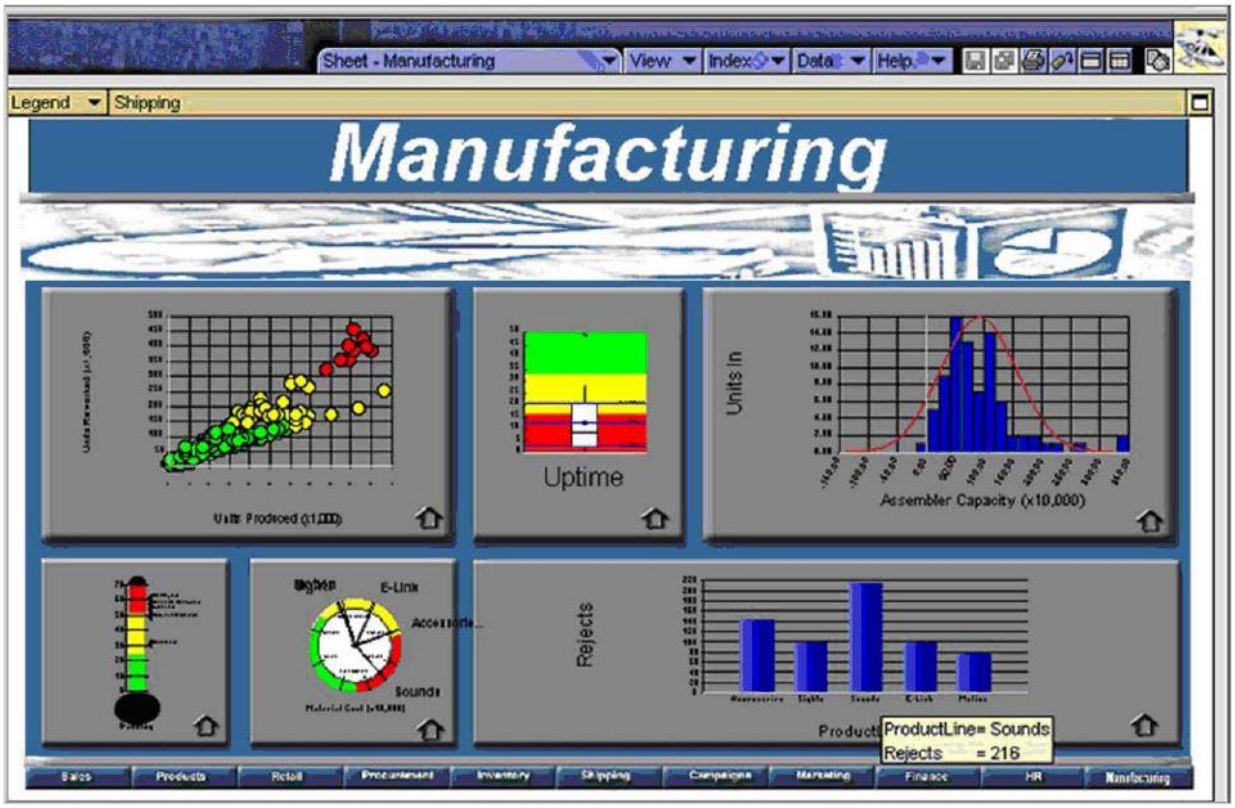


Figure 6: This thing is hideous, and unhelpful (Few, 2018a)

So what makes a good dashboard? Few describes a good dashboard as one that:

- Easily Scan the Big Picture
- Zoom in on important specifics
- Link to Supporting Details

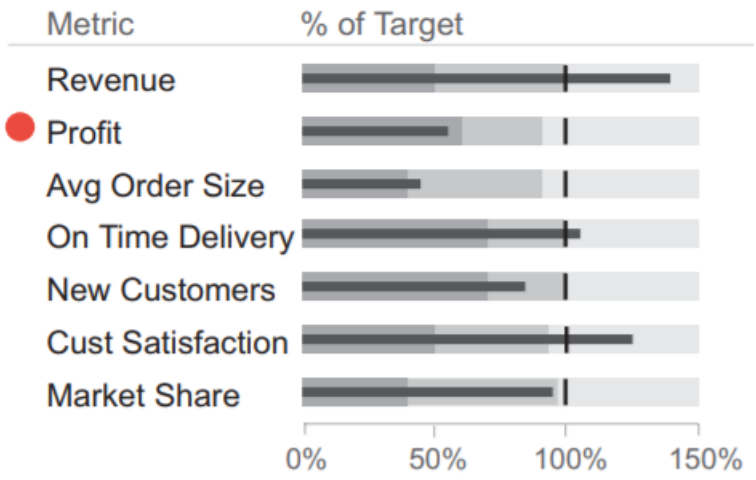


Figure 7: I can easily tell Profit is Bad (Few, 2018b)

Now that we have that discussed, let's discuss Shiny.

INTRO TO SHINY

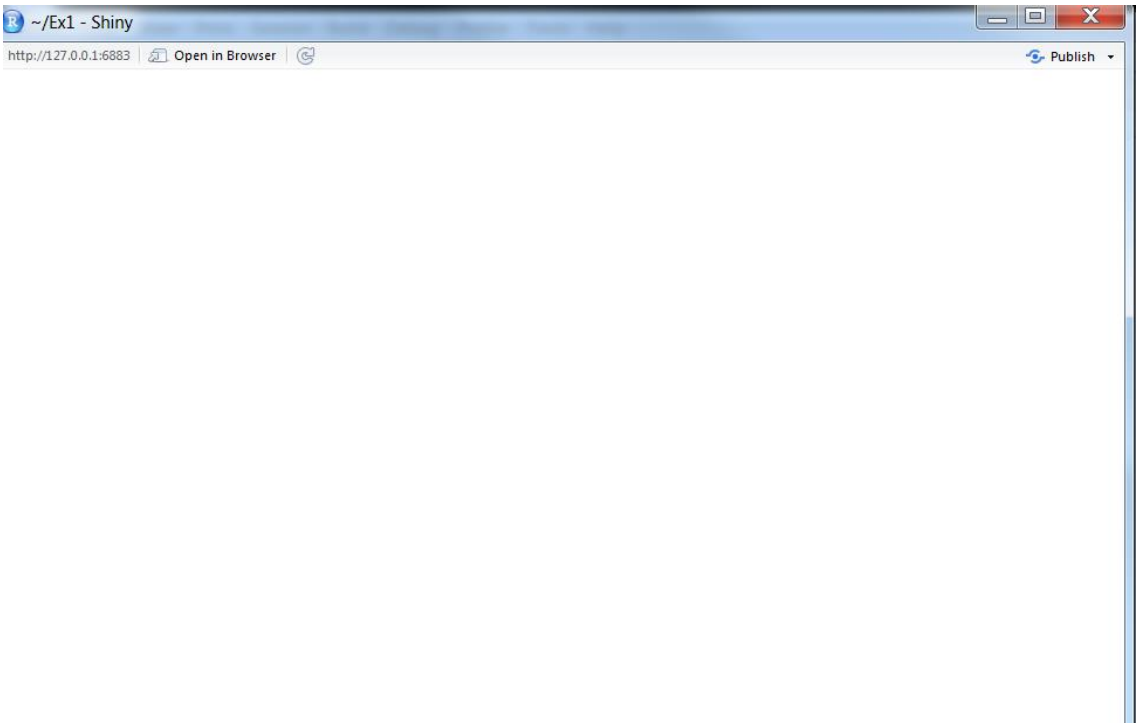
Shiny is an R package that makes it easy to build interactive web apps straight with R. Created with Javascript and CSS, we can create web apps right from R. The best part is that we can do it right within the language – no additional work required.

The way that the package works is that it runs a function – shinyApp, that takes a ui object and server function. The server can be your own computer (the default), or uploaded to a cloud-based service that will let other users look at it.

The structure of a basic shiny function is shown below:

```
library(shiny)
ui <- fluidPage(
)
server <- function(input, output) {
}
# Run the application
shinyApp(ui = ui, server = server)
```

The first line loads in the shiny library. Following that, it creates the ui object with nothing in it. It initiates the server function as well – calculating nothing, and then runs the application.



Output 9: Not Much to Look at, yet…

Not very exciting, but it will be the canvas in which you will paint your dashboard (figuratively: don't paint your screen, please).
Most of the work in getting a Shiny output is defined in the relationship between the UI object and the Server function, as shown below:

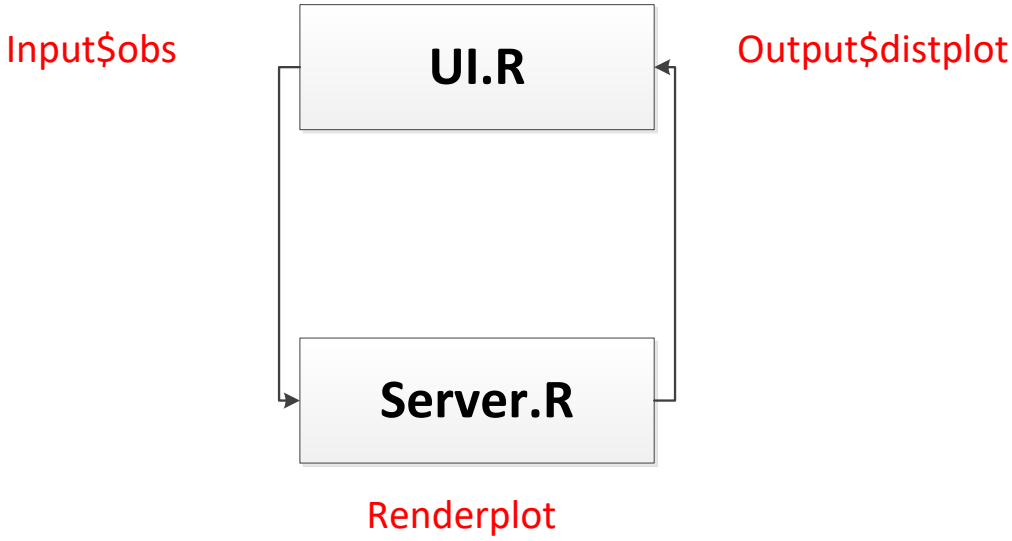


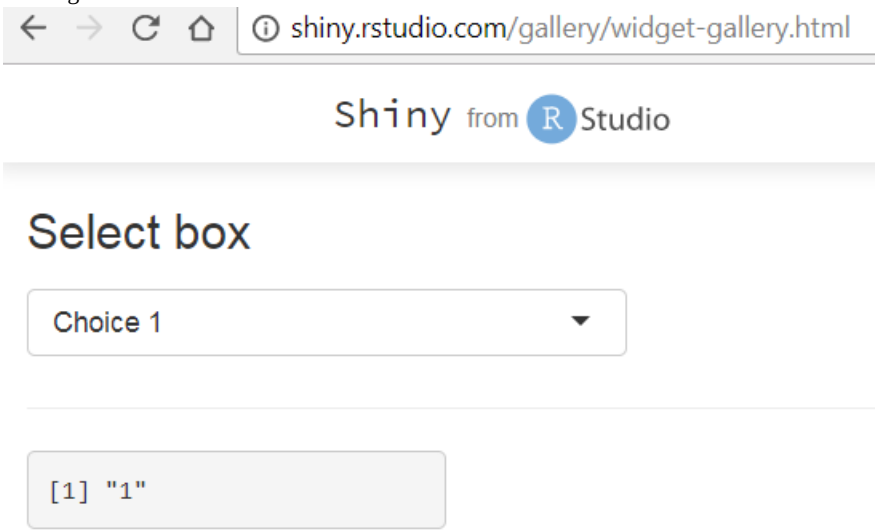
Figure 8: UI and Server – like Chocolate and Peanut Butter

The key to understanding shiny is in between the relationship between the UI object and the Server function. The UI object is responsible for accepting the inputs of the user, and communicating the outputs. The Server object is responsible to doing all the "R Stuff" (generating the plots, tables, doing the calculations, etc). Understanding that is the key to working with Shiny.

Another key to understanding Shiny are "widgets", functions that will create html objects and communicate the inputs that the user select. An example is

```
selectInput("select", label = h3("Select box"),
    choices = list("Choice 1" = 1, "Choice 2" = 2, "Choice 3" = 3),
     selected = 1),
```

which generates:



Output 10: A box that gives you three choices

Now that we have that, let's create a simple dropbox and a table.

A SIMPLE DROPBOX AND A TABLE

Let's review what we have:

```
library(shiny)
ui <- fluidPage(
)
server <- function(input, output) {
}
# Run the application
shinyApp(ui = ui, server = server)
```

Let's first add some data to the workspace. This will not change any of the output:

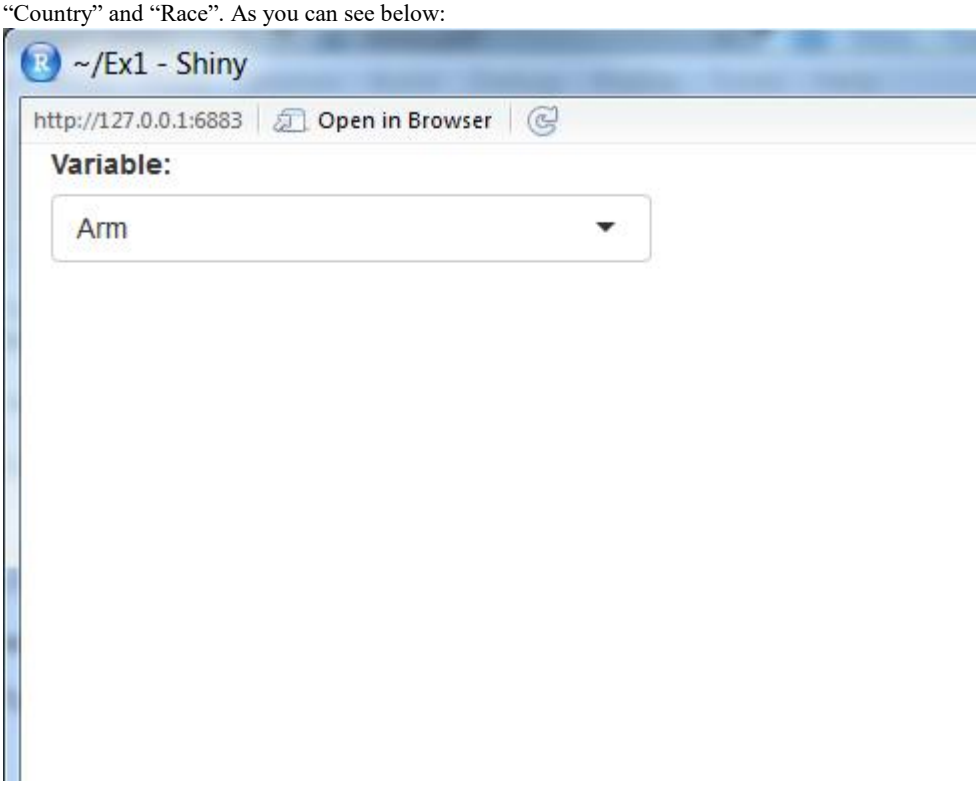
```
library(shiny)
library(haven)
dm <- read_sas("J:/drug/study/R_training/dm.sas7bdat")

ui <- fluidPage(
)

server <- function(input, output) {
}


# Run the application
shinyApp(ui = ui, server = server)
```

Now that we have done that, the next step is to create the input. We'll use the selectInput function discussed prior and add it to the ui object.

```
library(shiny)
library(haven)

dm <- read_sas("J:/drug/study/R_training/dm.sas7bdat")

ui <- fluidPage(  selectInput("variable", "Variable:",
                              c("Arm" = "ARM",
                                "Country" = "COUNTRY",
                                "Race" = "RACE"))
                )
```

```
server <- function(input, output) {

}

# Run the application
shinyApp(ui = ui, server = server)
```

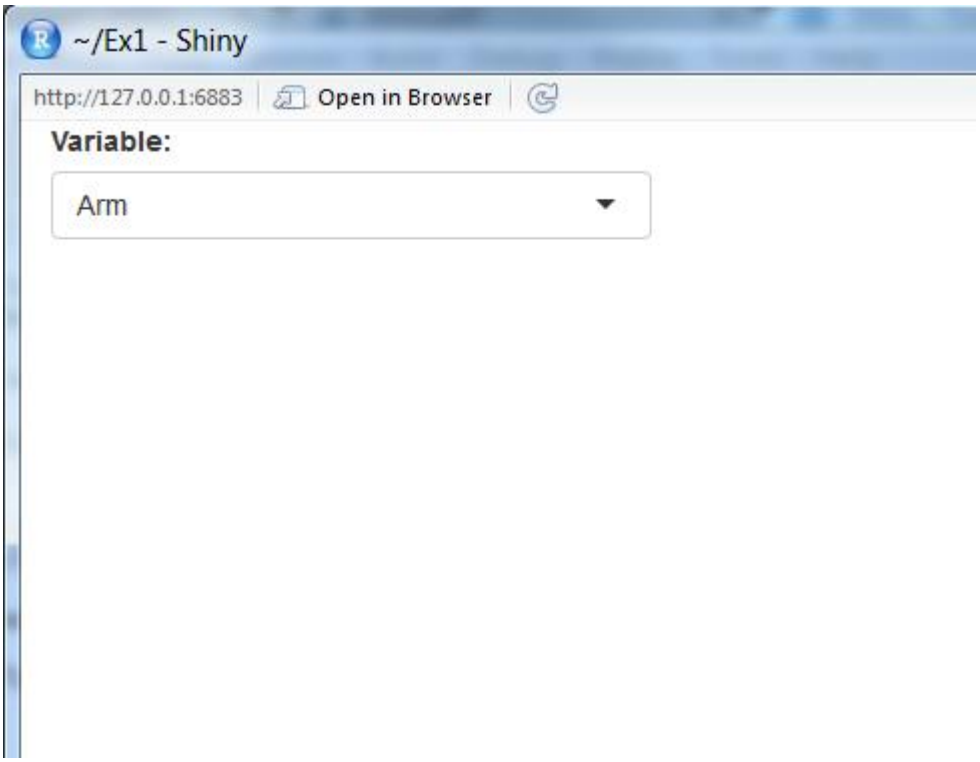
This now creates an attribute of the "input" object called variable that is the option that the user selects between "Arm", "Country" and "Race". As you can see below:



Output 11: Slow Progress….

Now we can use the input from the user to create a table

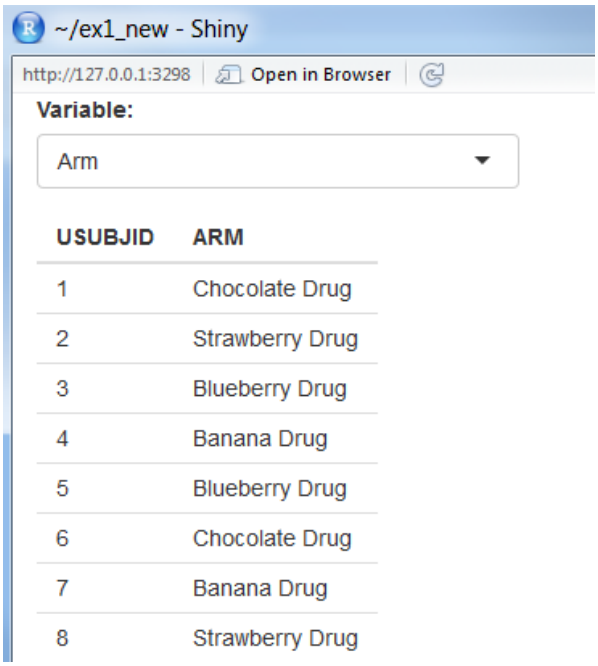
```
library(shiny)
library(haven)

dm <- read_sas("J:/drug/study/R_training/dm.sas7bdat")

ui <- fluidPage(  selectInput("variable", "Variable:",
                              c("Arm" = "ARM",
                                "Country" = "COUNTRY",
                                "Race" = "RACE"))
                )

server <- function(input, output) {
  output$data <- renderTable({
    dm[, c("USUBJID", input$variable), drop = FALSE]
  }, rownames = TRUE)
}

# Run the application
shinyApp(ui = ui, server = server)
```



Output 12: I thought there was supposed to be a table here?

I'm sure you note that there is no table here, that is because after the table is generated on the server side, it has to be rendered on the UI side. So, we have to use the RenderTable statement:

```
library(shiny)
library(haven)
dm <- read_sas("J:/drug/study/R_training/dm.sas7bdat")

ui <- fluidPage(
  selectInput("variable", "Variable:",
              c("Arm" = "ARM",
                "Country" = "COUNTRY",
                "Race" = "RACE")),
  tableOutput("data")
)

server <- function(input, output) {
  output$data <- renderTable({
    dm[, c("USUBJID", input$variable), drop = FALSE]
  }, rownames = TRUE)
}

# Run the application
shinyApp(ui = ui, server = server)
```



Output 13: We did it! An interactive table

EXTENDING THE FUNCTIONALITY

First, congrats on creating your first project with Shiny. I'd shake your hand if I was there:

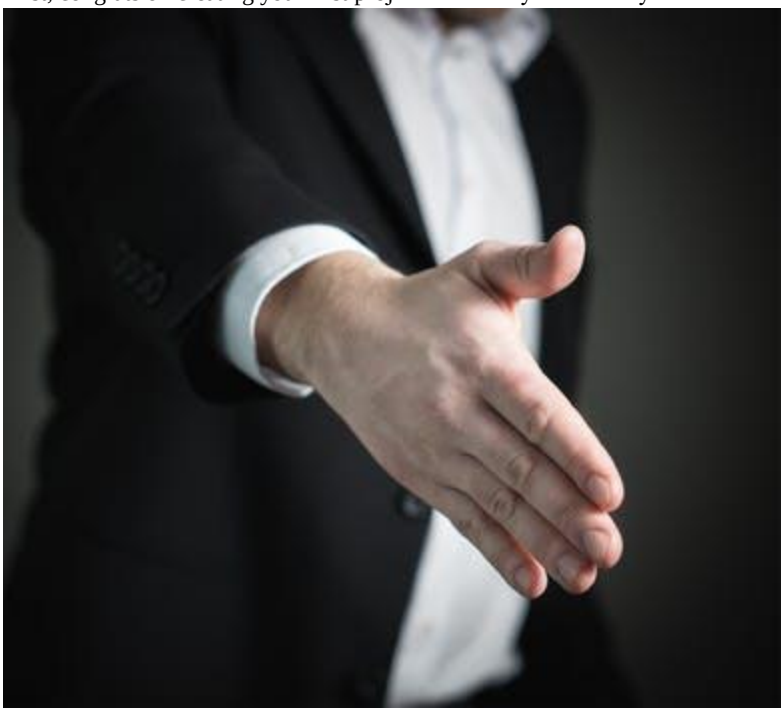


Figure 9: This is not my actual hand.

Everything that was just done is the core of Shiny. Everything else is just adding additional inputs and outputs, or nesting it within tabs or dashboards. If you can understand the basic loop we just did, everything else will be a breeze.

As an example, let's replace the table with a histogram. To do this, we'll use 2 different functions: tableOutput to replace plotOutput, and renderTable to replace renderPlot.

So let's take the code we just generated:

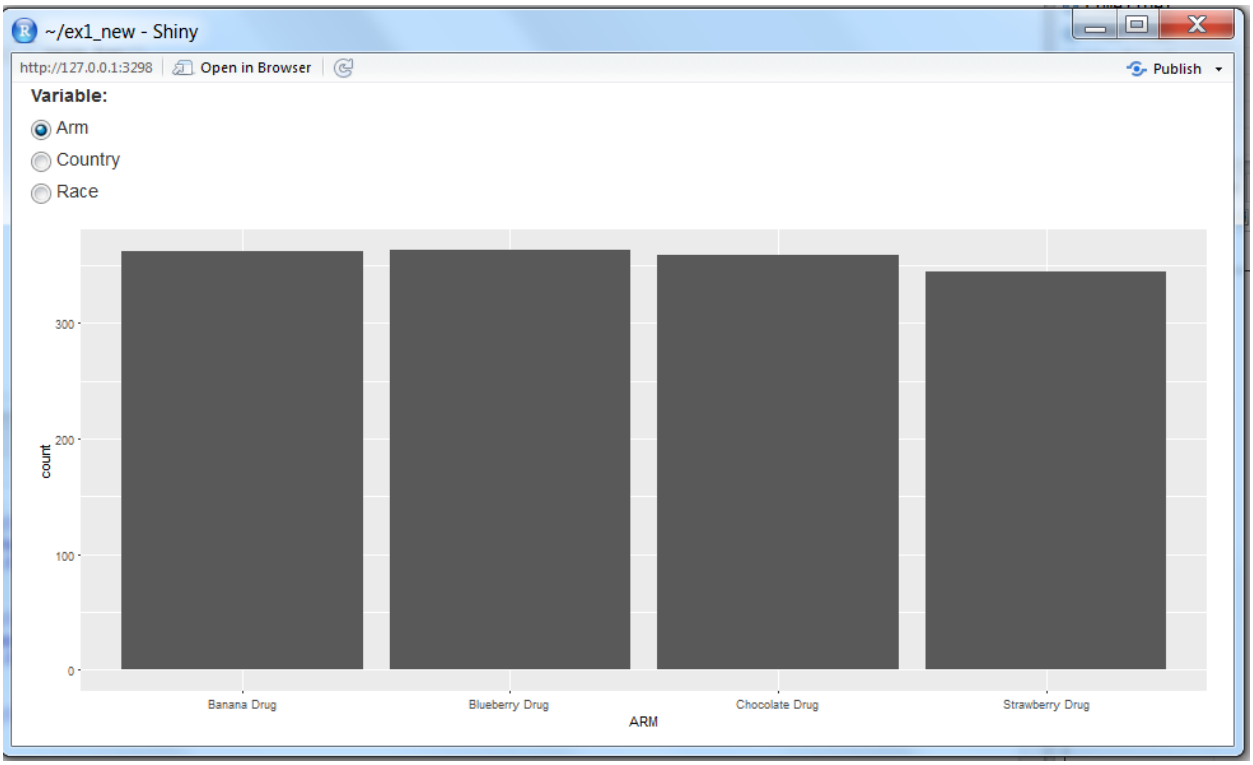
```
library(shiny)
```

```
library(haven)
dm <- read_sas("J:/drug/study/R_training/dm.sas7bdat")

ui <- fluidPage(
  selectInput("variable", "Variable:",
              c("Arm" = "ARM",
                "Country" = "COUNTRY",
                "Race" = "RACE")),
  tableOutput("data")
)

server <- function(input, output) {
  output$data <- renderTable({
    dm[, c("USUBJID", input$variable), drop = FALSE]
  }, rownames = TRUE)
}

# Run the application
shinyApp(ui = ui, server = server)
```

and make those replacements:

```
library(shiny)
library(haven)

dm <- read_sas("J:/drug/study/R_training/dm.sas7bdat")

ui <- fluidPage(  radioButtons("variable", "Variable:",
                               c("Arm" = "ARM",
                                  "Country" = "COUNTRY",
                                  "Race" = "RACE")),
                  plotOutput("plot")
               )

server <- function(input, output) {
  output$plot <- renderPlot({
    ggplot(dm, aes_string(input$variable)) + geom_bar()
  })
}

# Run the application
shinyApp(ui = ui, server = server)
```

You get an interactive plot just like that:



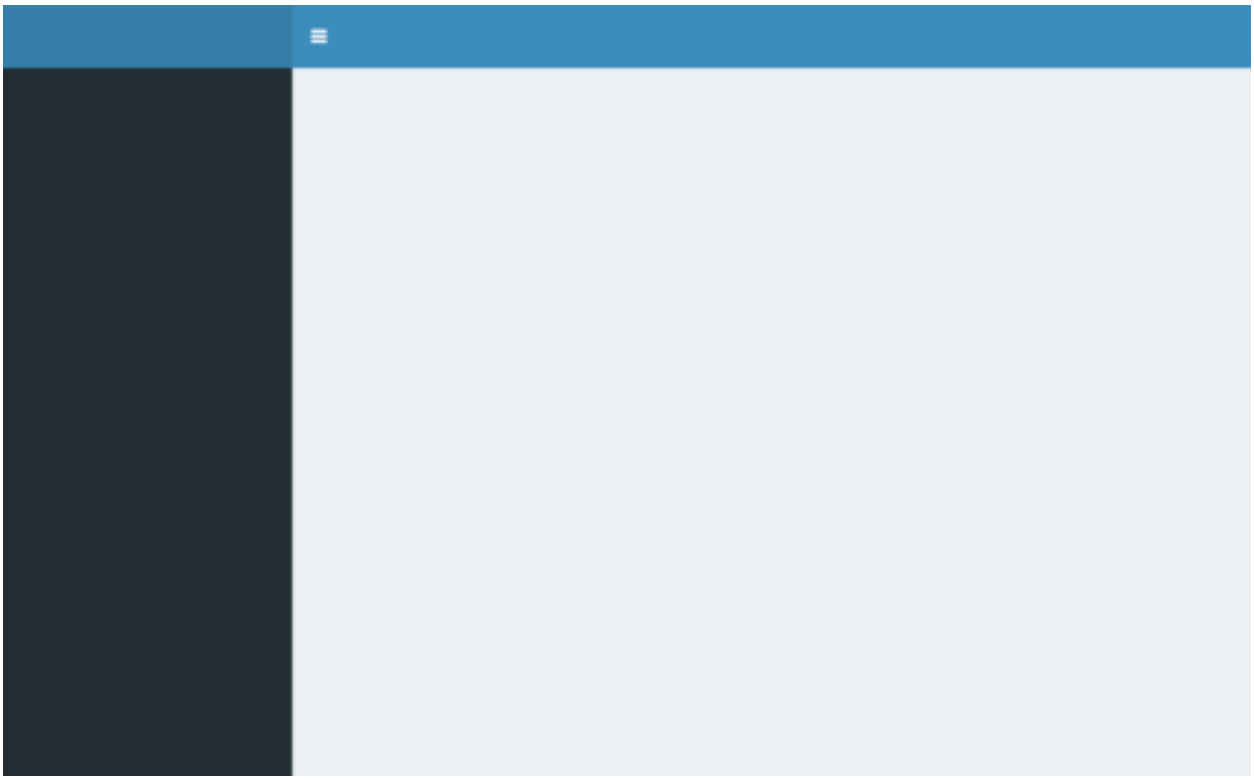
Output 14: Easy as Pie

DASHBOARD CONFESSIONALS

Now that you understand how to create and modify both table and plot objects, let's actually look at a dashboard. For this, we will need the shinydashboard package. Running the following code:

```
library(shiny)
library(shinydashboard)

ui <- dashboardPage(
  dashboardHeader(),
  dashboardSidebar(),
  dashboardBody()
)

server <- function(input, output) { }

shinyApp(ui, server)
```

which should look familiar to you, generates the following:



Output 15: Stephen Few Would Not Be Happy

The only differences are the UI object has another object within in… a dashboard object. This dashboard object has three parts: header (which contains the title), Sidebar (which deals with moving through different "pages" of a dashboard and is beyond the score of this paper), and Body. Body is basically the output that we already have. Consequently, things that we put in ui we now put in DashboardBody. The good part is that the program will now handle aligning the layout, so you can focus on more importantly matters, like deciding what actually to put in there.

CONCLUSION

This is but a taste of what R can do. From machine-learning to interactive dashboards, R has a lot of uses. Combined with the fact that it is free and open-source, as well as the fact anyone can contribute a package means whatever your question is, there probably is an R package that will do it. Any SAS Programmer should be at least be familiar with how to use it to add their tool belt..

REFERENCES

Muenchen, Robert A. 2011. *R for SAS and SPSS Users*. New York, NY :Springer

Priceanomics. 2015. "Hadley Wickham, the Man Who Revolutionized R." Accessed February 1, 2018. https://priceonomics.com/hadley-wickham-the-man-who-revolutionized-r/

Wickham, Hadley and Garret Grolemund. 2017. R for Data Science. Sepastopol, CA: O'Reilly Media

Few, Stephen. "Common Pitfalls in Dashboard Design." Accessed Feburary 1, 2018. Available at http://www.perceptualedge.com/articles/Whitepapers/Common_Pitfalls.pdf .

Few, Stephen. "Rich Data, Poor Data: Designing Dashboards to Inform" Accessed Feburary 1, 2018. Available at http://www.perceptualedge.com/articles/Whitepapers/Rich_Data_Poor_Data.pdf

CONTACT INFORMATION

Author Name: Nate Mockler
Company: Biogen
Email: Nate.Mockler@biogen.com

Author Name: Saranya Duraisamy
Company: Biogen
Email: Saranya.Duraisamy@biogen.com