Data Visualization

Lucy Jingru Wang

November 24, 2017

\(\color{Navy}{Visualization\ in\ R}\)

\(\color{olive}{\textbf{Introduction/Motivation}}\)

Today, we work with various large datasets to better understand the market, decision-making, and risk. In RStudio, there are various tools to help us work with datasets – including dplyr, ggplot, data frames, and so on. In this blog, I will go over functions for what I consider the most interesting part of data interpretation in R – data visualization.
Part of what makes working with data so fun, in my opinion, is the task of putting it together into plots, diagrams, graphs ~~ \(\color{red}{i}\color{orange}{n\ }\)\(\color{purple}{c}\color{blue}{o}\color{green}{l}\color{magenta}{o}\color{red}{r}\) With plotted data, we can better visualize distributions, spread, and correlation between variables of a dataset.

\(\color{olive}{\textbf{Background Information}}\)

Packages we will be using in the tutorials below are:

\(\color{CadetBlue}{\textbf{Shiny & ggvis}}\)

Shiny is a RStudio package that allows us to create interactive applications which take in inputs from the user to execute various functions. When it comes to data analysis, the ggvis package is often used in companion with the Shiny package. \(ggvis\) comes with functions like layer_histogram(), layer_points, and layer_boxplots(). These functions do exactly as titled – plot histogram, points, and boxplots. The two packages are often used together in app.R documents, which output the shiny app itself.

\(\color{CadetBlue}{\textbf{RColorBrewer}}\)

RColorBrewer is a package for customizing the colors of graphs and displays. Within the package are three types of palettes: sequential, diverging, and qualitative. Sequential palette is best suited for data that is ordered from low to high. Diverging palette is best suited for data with outliers. Qualitative palette is best suited for categorical data.

\(\color{CadetBlue}{\textbf{tm:}}\)

The Textmining package's most important component is the Corpus constructor. The Corpus represents a collection of texts that are retrieved by the function DirSource, which takes in the directory location as a parameter. tm also has functions that eliminate whitespace from a file, convert characters to lower case, or "stem words" which reduce words to the stem alone.

\(\color{CadetBlue}{\textbf{wordcloud:}}\)

The wordcloud package, not so much for data analyzation, was developed more for the fun of plotting words in the form of a cloud. The package uses a random generator to retrieve words from a text document in order to form a colorful cloud.

\(\color{olive}{\textbf{Tutorials}}\)

\(\color{red}{\textbf{Make sure to scroll down and refer to the code as you are reading the tutorials}}\)

\(\color{CadetBlue}{\textbf{Creating a Customizable Scatterplot for Users:}}\)

Oftentimes in data visualization, data is displayed in different colors and opacities in order to make it more visible to users which variables are related and which are not.
ggvis makes this possible with its opacity, shape, and other arguments that can be passed in to alter the visualization of graphs. In this tutorial, we will learn how to decorate a plot, but with a user's input in Shiny App.

Objective: creating an interactive scatterplot from mtcars. Mtcars is a dataset embedded within the RStudio documentation, adapted from a 1974 Moter Trend magazine depicting the performance for 32 different automobiles of the time.

\(\color{CadetBlue}{\textbf{Steps:}}\)

- Create a new Shiny app file
- Import shiny and ggvis packages with library()
- Create a vector for colors, it may contain any color you wish (you may look online for color options in R)
- Inside titlePanel, put in a string argument with the title you want, an example is "Data Display"
- As you can see in the code below, the two outer functions are sidebarLayout and sidebarPanel.
- The sidebarLayout stands for the overall layout of your app, including your sidebar and mainPanel.
- The sidebarPanel represents the toolbars on the left side of your application that you will be implementing.
- The mainPanel represents the main display that is outputted.

\(\color{CadetBlue}{\textbf{Drop down menu for changing the color of plots}}\)
Inside our sidebarPanel, we are going to first use the selectInput function to create a dropdown menu that allows users to input the color they want the plot to be in.

- arguments used: inputId, label, choices, and selected
- The selectInput function will output a drop down menu in your application, and since we are implementing this inside the sidebarPanel, the dropdown menu will show up on the left.
- Here, we will set choices equal to the previously defined colors vector; these are the choices the user can pick from for color
- inputId is the name that we want to give the user's choice from the menu, later on when we define functions in our server, we can retrieve the user's choice with input$(**name of your inputId**)
- label is the title for this menu, which tells the user what it is for: "Color of Points" precisely tells users that their input will change the color of the plot's points
- selected is the default choice that the plot is set to, I chose 'Navy' just because it is the first color in my 'colors' vector, but you can choose any color within your colors vector.
- After you have filled the arguments listed above, there should be a drop-down menu on the left side of your app.

\(\color{CadetBlue}{\textbf{Slider for Opacity}}\)

- Objective: implement a slider with a range of opacity levels from 0.1 to 1

- arguments used: inputId, label, min, max, value
- To create a slider for retrieving user input, we can use the function sliderInput
- Like selectInput, the arguments inputId, label are used the same way
- Unlike selectInput, the value argument is what gives the default slider value when the app is first opened; in the sample code below it is 1, which means the opacity of the plot is 1 when we first open the app.
- The min and max arguments give the range for opacity: 0.1 to 1

\(\color{CadetBlue}{\textbf{Radio Buttons for plot shapes}}\)

- Objective: implement a list of buttons for user to input the shape they want the plot points to be in.

- arguments used: inputId, label, choices
- To create buttons, we can use the radioButtons function
- Like the other functions, inputId, label, and choices are used
- choices points to a vector with all the possible shapes for plot points: circle, diamond, cross etc.
- The arguments should be defined in a meaningful way which allows the user to understand functionalities
- In the example below, I defined inputId as 'shape', label as 'Shapes of plot', and the choices as the shapes vector previously defined
- Although it's not very clear what inputId is, stick with me, it will be very useful in the next step!

```
library(shiny)
library(ggvis)

#vector of colors for user input
colors = c("Navy","Maroon","Gold",
           "Pink","Purple","Black",
           "Red","Green","Blue")

#vector of shapes for user input
shapes = c("circle","diamond","cross",
           "diamond","triangle-up","triangle-down")

ui <- fluidPage(

   # Application title
   titlePanel("Data display"),
   sidebarLayout(
      sidebarPanel(
        selectInput(inputId = "c", label ="Color of Points",
                    choices = colors, selected = "Navy"),
        sliderInput(inputId = "opac",label = "Opacity", min = 0.1,
                    max = 1,value = 1),
        radioButtons(inputId = "shape",label = "Shapes of plot",
                     shapes)
        ),
```

\(\color{CadetBlue}{\textbf{MainPanel}}\)

- The mainPanel, as mentioned before is where we want our scatterplot to display
- ggvisOutput is called on "plot", which we will define later; this function just outputs whatever methods are implemented inside "plot"
- The actual Shiny app code should not be split in this format, I am only doint this so it's easier to refer to the code

```
  mainPanel(
         ggvisOutput("plot")

   )
  )
)
```

\(\color{CadetBlue}{\textbf{Time to Define the Server}}\)

- Within the server is where we use ggvis to vary the plot depending on the user's input, retrieved with the previously defined inputID
- Here is where we define the "plot" that was previously called with ggvisOutput under the mainPanel
- Plot will be defined as a reactive expression
- Reactive definition from RStudio:
  "Reactive expressions are expressions that can read reactive values and call other reactive expressions. Whenever a reactive value changes, any reactive expressions that depended on it are marked as"invalidated" and will automatically re-execute if necessary. If a reactive expression is marked as invalidated, any other reactive expressions that recently called it are also marked as invalidated. In this way, invalidations ripple through the expressions that depend on each other."
- The ggvis function below is where we can set all the arguments that were "inputted" by the user with the widgets previously implemented inside sidePanel
- Inside plot, we will call ggvis on the dataset mtcars with %>%
- x = ~wt means that the x variable for this plot will be wt from the mtcars dataset and same goes for y
- How to use user input(Please refer to code chunk below as you are reading): input$c gives us the color that the user wants. If you scroll up, you can see that 'c' is what we had previously defined for the color dropdown menu's ID.
- Likewise, opacity will be input$opac and shape will be input$shape, all variables previously defined as ID.
- The final touch is layer_points(), which will reactively put points on the scatterplot based on user's preference for color, opacity, and shape.
- plot %>% bind_shiny will bind the 'plot' reactive expression we just defined as 'plot', which is then used in the mainPanel

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

   plot <- reactive({
     mtcars %>% ggvis(x = ~wt, y = ~mpg,
                      fill := input$c,
                      opacity := input$opac,
                      shape := input$shape) %>%
       layer_points()
   })
   plot %>% bind_shiny("plot")
}


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

\(\color{plum}{\textbf{Layer_() Functions}}\)

Objective: display simulated data on different types of plots

- Below, we will practice with several layer_ () functions from ggvis
- First, define a vector 'b' as rep(1:100,10), this just means a vector with 10 of each number from 1 to 50
- The purpose of creating the vector 'b' is to use it as a "population" for sampling
- After the "population" is created, we will then sample twenty times and create a data frame
- Make sure to set seed as 3, so that your output will be the same as mine if you are following this tutorial word by word.
- To create data frame, we can use data.frame() with arguments x and y
- x stands for the nth sample w/replacement while y stands for the outcome of that sample.
- y below is defined as sample(b,20,replace=TRUE), which means sample 20 times from the 'b' vector with replacement

- Once your simulation data frame has been created, we can now plot.
- Like our previous encounterance with ggvis, we will again use %>% to denote that ggvis is retrieving variables 'x' and 'y' from the df2
- Lastly, the functions layer_paths(),layer_points(),layer_bars(), and layer_model_predictions() will be used.
- As we learned before, the layer_points() function will output a scatterplot
- layer_paths will draw lines between the points, layer_bars will draw a barplot, and layer_model_predictions will output a "model" that you define
- For layer_model_predictions(), you may use "loess" or "lm" to instantiate the model argument
- I personally feel that the barplot is best for visualization because it gives me a sense of which values had higher/lower frequency in the stimulation. However, depending on the project you are working with, one type of visualization may be better than another.

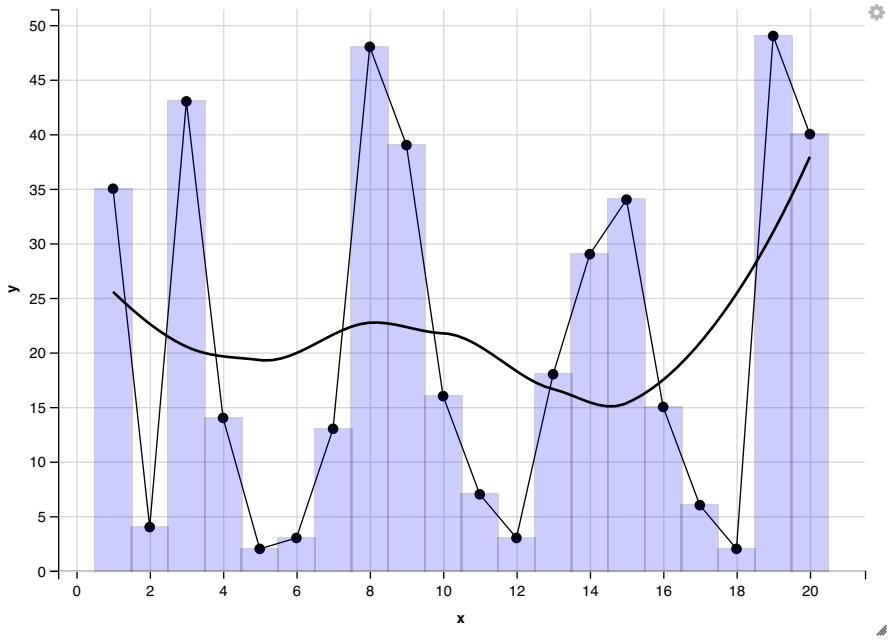
```
library(ggvis)
```

```
## Warning: package 'ggvis' was built under R version 3.4.2
```

```
b = rep(1:50,10)
set.seed(3)
df2<- data.frame(x = 1:20, y = sample(b,20,replace=TRUE))

df2 %>% ggvis(~x,~y) %>%
  layer_paths()%>%
  layer_points()%>%
  layer_bars(fill:= 'blue', opacity := 0.2)%>%
  layer_model_predictions(model = 'loess')
```

```
## Guessing formula = y ~ x
```



\(\color{CadetBlue}{\textbf{Coin Simulation}}\)

In this exercise, we are going to find the answer to a Math10A (Calculus course @ UCB) question by simulation

Which is more likely, getting 60 or more heads in 100 tosses of a fair coin or getting 225 or more heads in 400 tosses of a fair coin?

Steps of operation:
* set p as the probability of getting a head on a coin flip - 0.5 * set N as the number of tosses: N stands for the 100 tosses, and N2 the 400 tosses trial
* set outcomes as 0 or 1, 0 stands for head and 1 stands for tails
* simulation1 will be an empty vector in which we can then use to store the tosses in
* using a for loop from 1 to 1000, we will do a simulation of 1000 trials of 100 flips per trial first
* 'toss_100' is first set as the outcome of flips by flipping 100 times
* 'toss_100' is then changed into a data frame with flip value (head/coin) and frequency
* we will put the frequency of 0's (heads) into the simulations vector
* Eventually when the for-loop finishes running, the simulations1 will contain the number of heads per 100 flips for the 1000 trials
* The exact same procedure is repeated for simulations2, which is 1000 trials of 400 flips
* As we used before, we are going to use ggvis again to plot, but this time a histogram
* What is the answer to the stat question?? First try to answer it yourself, and then scroll down for the answer.

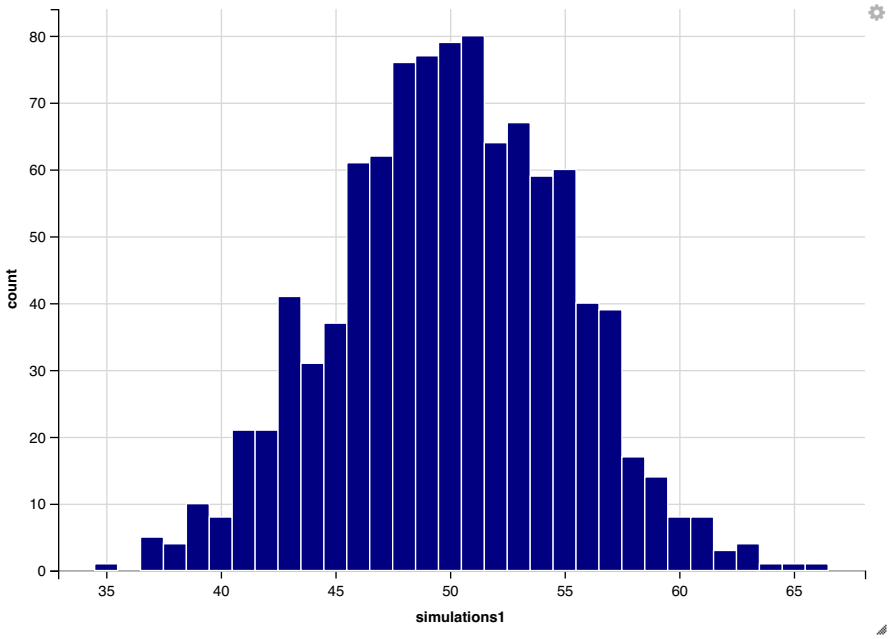
```
p <- 0.5
N <- 100
N2 <- 400
outcomes <-c(0,1)

simulations1 <- c()
for(n in 1:1000) {
  toss_100 <- sample(outcomes,size=N,replace = TRUE, prob = c(p,1-p))
  toss_100 <- as.data.frame(table(toss_100))
  simulations1[n] <- toss_100$Freq[toss_100==0]
}

simulations1 <- data.frame(simulations1)

simulations1 %>%
  ggvis(~simulations1, fill:= "navy", stroke:= "white")%>%
  layer_histograms()
```

```
## Guessing width = 1 # range / 31
```

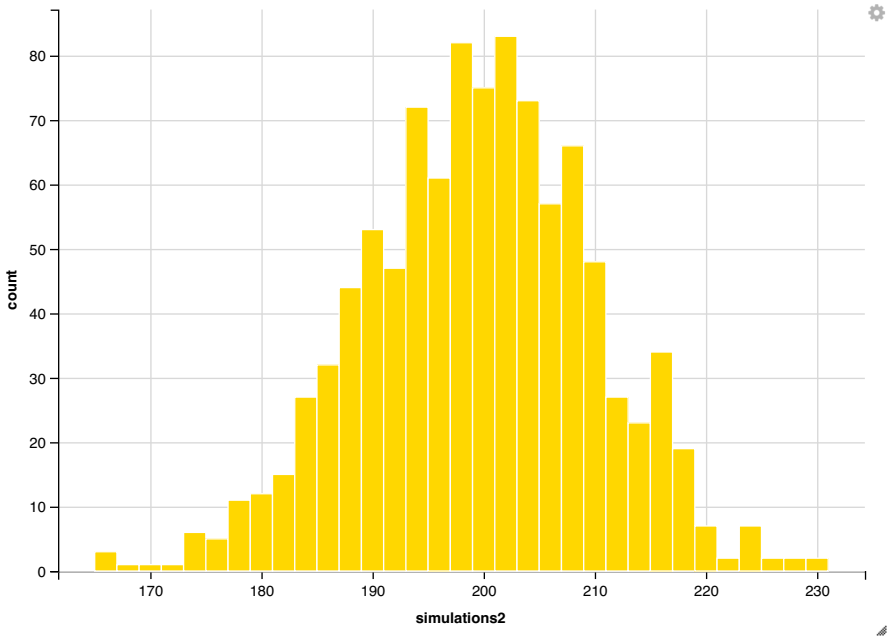



```
simulations2<- c()
for(n in 1:1000) {
  toss_400 <- sample(outcomes,size=N2,replace = TRUE, prob = c(p,1-p))
  toss_400 <- as.data.frame(table(toss_400))
  simulations2[n] <- toss_400$Freq[toss_400==0]
}

simulations2 <- data.frame(simulations2)

simulations2 %>%
  ggvis(~simulations2, fill := "gold", stroke:="white")%>%
  layer_histograms()
```

```
## Guessing width = 2 # range / 33
```




```
length((simulations1[simulations1>=60]))
```

```
## [1] 26
```

```
length((simulations2[simulations2>=225]))
```

```
## [1] 9
```

\(\color{CadetBlue}{\textbf{Answer}}\)

If we look at each histogram, it is clear that getting 60 or more in 100 flips is more likely. This also makes sense from a statistical perspective; Think about it like this, if we have a smaller sample size, then the spread is greater and thus it is more likely to have flip proportions that's farther from 0.5.

\(\color{Plum}{\textbf{Concluding Tutorial: WordCloud}}\)

- To wrap this up, we will end with a WordCloud tutorial
- WordCloud, unlike ggVis, is not a tool that we would use for data plotting/interpretation, instead it is just a fun visual for words
- First make sure to download and load the package wordcloud
- download and load the packages 'tm' and 'RColorBrewer'
- create a directory called text, and put in a text file with any messages from the internet or elsewhere, but make sure there are repreating words for visual purposes
- using the vector color_pallets, store colors that are listed below
- type in the functions used in the below code snippet
- Explanation of the functions below:

1. Corpus will turn the text directory into a collection of written texts that can me represented and manipulated with the tm package functions, this is a very significant conversion within the tm package
2. tm_map(text, ____) functions: just as titled, the stripWhitespace takes out white spaces from the words, the tolower will switch all words to lowercase, and the removeWords functions to remove insignificant words.
3. Lastly, we can make a wordcloud by using the brewer.pal, which is a package with various colors, and if we sample from the color_palettes, each time we execute the below code, we will get different colors for our cloud.
4. the scale functios to scale size of tex the max words represents the maximum repeats.
5. Run the code and a cool cloud will display
6. Ta-daaa

```
library(wordcloud)
```

```
## Loading required package: RColorBrewer
```

```
library(tm)
```

```
## Loading required package: NLP
```

```
library(RColorBrewer)

color_palettes <- c("Accent","Dark2","Paired","Pastel1","Set1","Set2","Set3")

text <- Corpus(DirSource("text"))

text <- tm_map(text, stripWhitespace)

text <- tm_map(text, tolower)

text <- tm_map(text, removeWords, stopwords("english"))

wordcloud(text, scale = c(5,0.5), max.words = 50, random.order = FALSE, rot.per = 0.35, use.r.layout=FALSE, colors
= brewer.pal(7,name = sample(color_palettes,1)))
```

\(\color{Plum}{\textbf{Take Home Message: }}\)

Hopefully you now know how to use basic plots in ggvis and how to make a wordCloud.
The objective of this blog was to teach you how to use Shiny with ggvis as well as create plots from various types of data. In statistics, we work a lot with data, and a huge part to interpreting is visualization, particularly for people who don't understand statistics. Suppose you are trying to explain the meaning of a dataset and correlation to your client, but they have zero knowledge in stats. It will be much easier just to convince them of your conclusion with a plot rather than explaining all the mathematical concepts - that is, however, only one use for visualization packages like ggvis. As you explore other fields, you will discover numerous other uses for data visualization.

\(\color{Plum}{\textbf{References: }}\)

https://www.datacamp.com/community/tutorials/make-histogram-ggvis-r https://www.r-bloggers.com/normal-distribution-functions/
https://ggvis.rstudio.com/ggvis-basics.html https://georeferenced.wordpress.com/2013/01/15/rwordcloud/ https://www.r-bloggers.com/more-explorations-of-shiny/ https://cran.r-project.org/web/packages/RColorBrewer/RColorBrewer.pdf
https://shiny.rstudio.com/reference/shiny/latest/reactive.html