Post 02: Unique Data Visualizations Using 'ggplot2' and 'plotly' (with other packages)

Stat 133, Fall 2017

Woo Sik (Lewis) Kim

Introduction

We've worked with data a lot in this course - everything from cleaning them to dynamically displaying them in a `shiny` app. However, in terms of data visualization, we have been primarily focused on histograms, barcharts, and scatterplots. Although these forms of data visualization are arguably the most useful and versatile for almost any type of data, there are also many other forms of data visualization we don't see every day. These range from waffle plots, violin plots, and lollipop plots, to even 3D graphs.

In this post, I'll be expanding on what `ggplot2` can do (with the help of other packages), and also introduce you to a great package called `plotly` that's extremely light, easy-to-use, and aesthetically pleasing. We'll take a look at 5 different types of data visualizations we don't see every day: density plots, time series graphs, spatial data maps, 3D scattter plots, and 3D line graphs.

Before we get started, there are a couple of things to set up. Since `ggplot2` and `plotly` will be the main packages for this post, let's first download and install the packages (if you haven't before):

```
# Type these commands into your console.
install.packages('ggplot2')
install.packages('plotly')
```

Then, load the packages:

```
library(ggplot2)
library(plotly)
```

```
##
## Attaching package: 'plotly'
```

```
## The following object is masked from 'package:ggplot2':
##
##     last_plot
```

```
## The following object is masked from 'package:stats':
##
##     filter
```

```
## The following object is masked from 'package:graphics':
##
##     layout
```

Since we'll be using various different types of data (multiple .csv files), create a folder called "data" in your working directory and put the .csv files there. There will be links to download the datasets we use.

Now we're ready to get started on our data visualizations.

`ggplot2` - Part 1: Density Plots

A density plot visualizes the distribution of data over a continuous interval, or time period. In other words, it shows how "concentrated" data is at certain points. The peaks of a density plot help display where values are most concentrated over the interval. For our density plot, let's work with the built-in mpg dataset, and see how concentrated highway mileage is relative to car types.

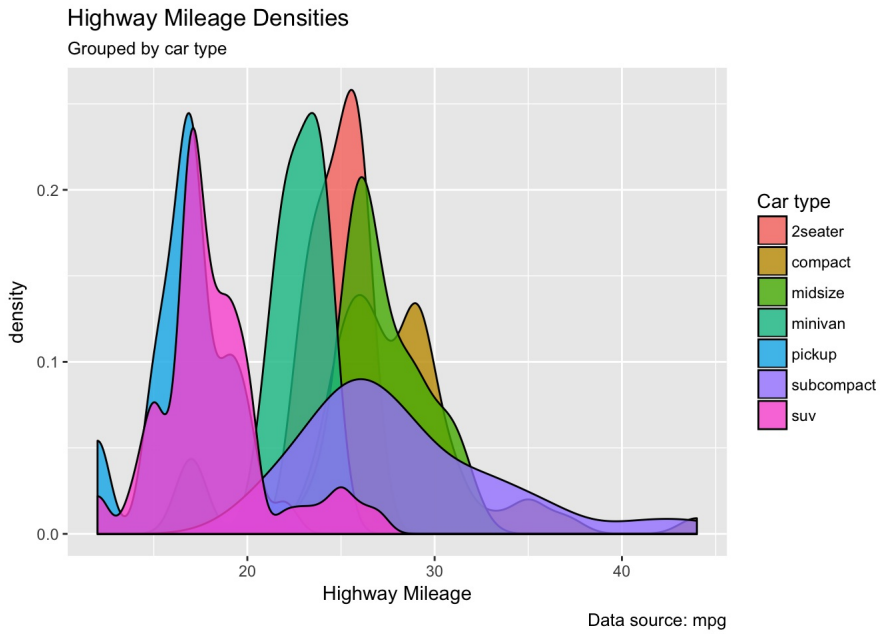
Creating a density plot with `ggplot2` is rather straightforward; follow this general pseudocode to get started.

```
plot <- ggplot(DATASET, aes(COLUMN FOR DENSITY))

plot + geom_density(aes(fill=factor(COLUMN TO ORGANIZE BY)), alpha=OPACITY) +
  labs(title, subtitle, caption, x, fill)
```

Extrapolating from the pseudocode above, we have `DATASET = mpg`, `COLUMN FOR DENSITY = hwy`, `COLUMN TO ORGANIZE BY = class`, `alpha = 0.8`. We'll also fill in the arguments of `labs()` with the appropriate plot title, subtitle, etc.

```
density_plot <- ggplot(mpg, aes(hwy))
density_plot + geom_density(aes(fill=factor(class)), alpha=0.8) +
    labs(title="Highway Mileage Densities",
         subtitle="Grouped by car type",
         caption="Data source: mpg",
         x="Highway Mileage",
         fill="Car type")
```



From our density plot, we can easily see the distribution shape (e.g. Normal, Left-Skew, Right-Skew, etc.) for each car type's highway mileages. This makes analyzing how concentrated data is at certain continuous intervals much easier and intuitive than, say, a histogram would.

`ggplot2` - Part 2: Time Series Graphs

A Time Series is a dataset with time attached as a variable. It shows how the y-variable fluctuates over a certain time interval - these time intervals can be as small as minutes, or as long as centuries. As long there's some variable attached to time, we're able to analyze it. Time series is especially useful for forecasting and prediction models; this statistical method is most notably used in finance, actuarial science, and machine learning. This section will serve as a *very* small and brief introduction to what time series graphs *look like*. We will not be going into specific details about the mathematics behind time series - only the visualization. If you're interested in learning more about this extremely useful analytical skill, consider taking Stat 153.

Now, let's take a look at how to create time series graphs in R using `ggplot2`, with a dataset of UK road casualties from 1969-1984. We'll see how the number of casualties fluctuates over 15 years (the intervals aren't separated by 1 year, but rather much smaller subintervals; 0.08333 years for our dataset, to be exact). We're actually also able to predict future trends with the package `forecast`, but we won't be going into that. If you're interested in learning more about `forecast`, check out this link: forecast package

Since the data we'll be using is uploaded in a csv format in GitHub, we can read in the .csv file directly from the link (see the code chunk below for the link). But, if you want to download the dataset, the .csv file is available here: UK Road Casualties

Most general time series graphs in `gplot2` follows this general pseudocode:

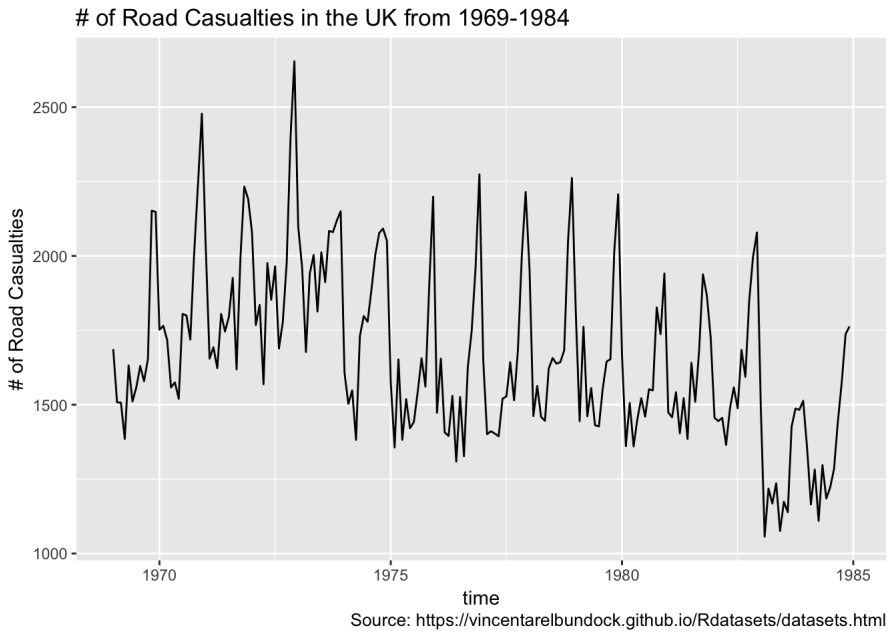
```
ggplot(DATASET, aes(x=TIME)) +
  geom_line(aes(y=COLUMN TO ANALYZE)) +
  labs(title, caption, y)
```

Of course, the most important thing to realize here is that your dataset *needs* to have time has a variable. Otherwise, time series just doesn't make any sense.

Since our dataset is already formatted to be compatible with a time series graph, there isn't much to do with `ggplot2` other than simply using `geom_line()` and `labs()`:

```
UK_casualties <- read.csv("https://raw.githubusercontent.com/vincentarelbundock/Rdatasets/master/csv/datasets/UKDriverDeaths.csv")

ggplot(UK_casualties, aes(x=time)) +
  geom_line(aes(y=UKDriverDeaths)) +
  labs(title="# of Road Casualties in the UK from 1969-1984",
       caption="Source: https://vincentarelbundock.github.io/Rdatasets/datasets.html",
       y="# of Road Casualties")
```



Source: https://vincentarelbundock.github.io/Rdatasets/datasets.html

Here, we can see that there was an overall downward trend in the # of road casualties . This is expected, as there would have been safer cars and better regulations over the 15-year timespan. Even though this is a very small and simple introduction to visualizing a time series dataset, you can see that these type of graphs can be extremely useful in analyzing how a certain variable changed over time, how it currently changes over time, and how it will most likely change over time.

`ggplot2` - Part 3: Spatial Data Maps

So far, we've been working with quantitative numbers and how to visualize them in different ways. But, what if we want to work with spatial data? In other words, what if we want to display and analyze specific locations on a map? For instance, what if we want to display a certain number of landmarks/buildings (such as the buildings at UC Berkeley), and cluster them on a satellite map of the region?

To create these "map graphs", we'll need to download and install a package called `ggmap` and `ggalt`. These packages work in conjunction with `ggplot2`, and lets us highlight and encircle certain features on a section of Google Maps, which we download.

`ggmap` and `ggalt` aren't uploaded on CRAN; this means that you won't be able to download them using `install.packages()`. We'll need to use `devtools` to download the other packages from GitHub. To get started, install these packages using this code:

```
install.packages('devtools')
devtools::install_github("dkahle/ggmap")
devtools::install_github("hrbrmstr/ggalt")
```

After installing them, let's load the necessary packages:

```
# No need to load devtools.
library(ggmap)
```

```
## Google Maps API Terms of Service: http://developers.google.com/maps/terms.
```

```
## Please cite ggmap if you use it: see citation("ggmap") for details.
```

```
##
## Attaching package: 'ggmap'
```

```
## The following object is masked from 'package:plotly':
##
##     wind
```

```
library(ggalt)
```

Now, we're ready to make our map graph. To do this, we're going to be primarily using `geocode()` and `qmap()`, along with `ggplot2` features we've seen before: `geom_points()` and `encircle()`.

For those who need a refresher, `geom_points()` circles certain features on a graph/chart. For our map, it'll be highlighting a coordinate (longitude and latitude) on a Google Map section.

`encircle()` "circles" (or "clusters") a group of data points with a colored line. For our map, `encircle()` will group together certain locations (coordinates) on a Google Map section.

`geocode()` is a function that takes in a name of a location (a string), and returns the longitude/latitude coordinates of that location.

`qmap()` is a wrapper for `ggmap` and `get_map()`. This simply means that it downloads the map of a location from a source (a server). `qmap()` follows the following general structure:

`qmap(location, zoom, source, maptype)`

• `location` is a string that tells `qmap()` what part of Google Maps you'll be using. It can be an address, such as "ADDRESS", or it can be a name, such as "University of California, Berkeley". For our map, let's use UC Berkeley.

• `zoom` is exactly what it sounds like - it tells `qmap()` how far into the map to zoom in. The higher the `zoom` value, the further it zooms. For our map, let's use 16. You can use trial-and-error to see what zoom level is right for your map.

• `source` tells `qmap()` what server to download the map from - in most cases, it'll be Google, and `source` will be set to `source = "google"`.

• `maptype` is the type of map we'll be download: it can be a satellite map, a road map, or a hybrid map (values are "satellite", "roadmap", and "hybrid"). For our map, let's use a satellite map.

Now, we're ready to create the map! Follow the comments in the code chunk for explanations.

```
# A vector of certain UC Berkeley building names.
ucb_buildings <- c("Barrows Hall", "Dwinelle Hall", "Soda Hall", "Latimer Hall")

# Getting the coordinate data for the previous UCB buildings using geocode().
building_locations <- geocode(ucb_buildings)
```

```
## Source : https://maps.googleapis.com/maps/api/geocode/json?address=Barrows%20Hall
```

```
## Source : https://maps.googleapis.com/maps/api/geocode/json?address=Dwinelle%20Hall
```

```
## Source : https://maps.googleapis.com/maps/api/geocode/json?address=Soda%20Hall
```

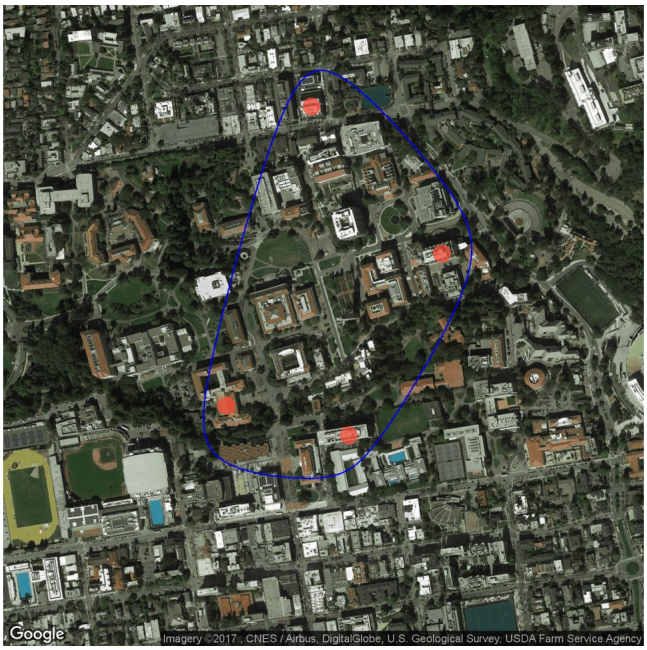
```
## Source : https://maps.googleapis.com/maps/api/geocode/json?address=Latimer%20Hall
```

```
# Downloading a satellite Google map of UC Berkeley, with a zoom level of 16.
ucb_sat_map <- qmap("University of California, Berkeley", zoom=16, source = "google", maptype="satellite")
```

```
## Source : https://maps.googleapis.com/maps/api/staticmap?center=University+of+California,+Berkeley&zoom=16&size=
640x640&scale=2&maptype=satellite&language=en-EN
```

```
## Source : https://maps.googleapis.com/maps/api/geocode/json?address=University%20of%20California%2C%20Berkeley
```

```
# Using ggplot2 features on the downloaded satellite map: geom_point() highlights the building locations with a CO
LOR dot, and geom_encircle() clusters the building locations with a blue line.
ucb_sat_map + geom_point(aes(x=lon, y=lat),
data = building_locations,
alpha = 0.7,
size = 4,
color = "tomato") +
geom_encircle(aes(x=lon, y=lat),
data = building_locations, size = 1, color = "blue")
```



As you can see, our code created "map graph" that highlights 4 iconic UC Berkeley buildings on a satellite map: Dwinelle Hall, Barrows Hall, Soda Hall, and Latimer Hall. Then, it clustered them with a line for easy visualization of the area between these buildings.

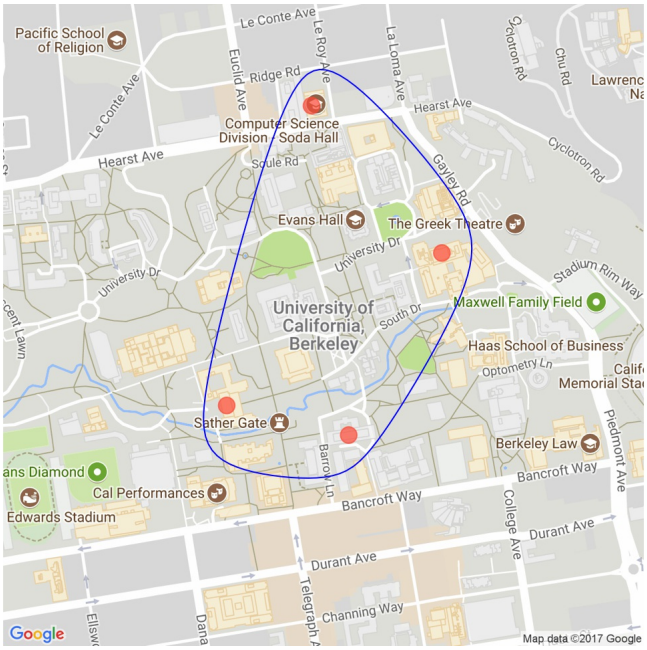
What if we want to display them on a road map? All we need to do is change up a couple of things in our code:

```
# Downloading a road Google map of UC Berkeley, with a zoom level of 16.
ucb_road_map <- qmap("University of California, Berkeley", zoom=16, source = "google", maptype="roadmap")
```

```
## Source : https://maps.googleapis.com/maps/api/staticmap?center=University+of+California,+Berkeley&zoom=16&size=640x640&scale=2&maptype=roadmap&language=en-EN
```

```
## Source : https://maps.googleapis.com/maps/api/geocode/json?address=University%20of%20California%2C%20Berkeley
```

```
# Using ggplot2 features on the downloaded road map:
ucb_road_map + geom_point(aes(x=lon, y=lat),
data = building_locations,
alpha = 0.7,
size = 4,
color = "tomato") +
geom_encircle(aes(x=lon, y=lat),
data = building_locations, size = 1, color = "blue")
```



These type of spatial graphs can get much more complex, such as adding multiple clusters, and can be very useful in analyzing spatial intersections and help us deduce spatial information such as which areas are most populated and which areas are most empty.

Using `plotly`

`plotly` is a package that provides simple and easy-to-use data analytics and visualization tools. It provides online graphing, analytivs, and statistics tools, and works not only with R, but also with Python, MATLAB, Perl, Julia, Arduino, and REST.

`plotly` also lets us create interactive HTML widgets with just a few lines of code. It's interactivity is similar to that of `shiny`, but `plotly` doesn't involve any user inputs; rather, it provides a "panel" with soft buttons that the user can click to customize how they look at graphs and plots. `plotly` is extremely intuitive and easy to pick up: there isn't much to write in terms of code, and is intuitive enough that it doesn't need a lot of explanations for different arguments.

Here's the cheat sheet for `plotly` if you need more information on what `plotly` can do: plotly cheat sheet

---

`plotly` - Part 1: Dumbbell Plots

Let's get started with `plotly` and create a simple dumbbell plot. A dumbbell plot is a plot that shows how a certain data changes over an interval, such as time (e.g. how the approval ratings for a candidate changes over a year). A single point on a dumbbell plot consists of 3 parts: a beginning point (time 1), an end point (time 2), and a horizontal line segment that connects these 2 points. A dumbbell plot can be very useful in analyzing how certain variables increase or decrease over time relative to others (e.g. increases in CO2 vs. decreases in ozone over 100 years).

Here's the general pseudocode for a `plotly` dumbbell plot:

```
# Order the column properly with a factor:
DATASET$COLUMN <- factor(DATASET$COLUMN, levels=as.character(DATASET$COLUMN))

plot <- plot_ly(DATASET, color = COLOR)) %>%
  # Add the line segment that connects 2 end points: xend and yend.
  add_segments(x = TIME_1, xend = TIME_2, y = COLUMN, yend = COLUMN, showlegend) %>%
  # Add markers for each ends of the line segments (time 1 and time 2)
  add_markers(x = TIME_1, y = COLUMN, name = TIME_1 NAME, color = COLOR) %>%
  add_markers(x = TIME_2, y = COLUMN, name = TIME_2 NAME, color = COLOR) %>%

  # Configure layout.
  layout(TITLE, xaxis = list(title = XAXIS NAME), margin)

plot
```

We'll be using a dataset for the approval ratings (% of total polulation that approves) for fictional political candidates. The dataset is available

here: Candidate Approval Ratings. Don't forget to put your .csv file in your "data" folder.

Now, we're ready to create our dumbbell plot. Follow the comments in the code chunk for explanations. You'll also notice that `plotly` comes with a soft button bar when you hover over the graph. You can interact with them to customize how you observe the data.

```
ratings <- read.csv("data/approvalratings.csv")
# To correctly order the candidates' names:
ratings$candidate <- factor(ratings$candidate, levels=as.character(ratings$candidate))

dumbbell_plot <- plot_ly(ratings, color = I("gray80")) %>%
  # Add the line segment that connects 2 end points: xend and yend.
  add_segments(x = ~rating_2005, xend = ~rating_2006, y = ~candidate, yend = ~candidate, showlegend = FALSE) %>%
  # Add markers for each ends of the line segments: in this case, the 2005 and 2006 ratings.
  add_markers(x = ~rating_2005, y = ~candidate, name = "2005 Ratings", color = I("pink")) %>%
  add_markers(x = ~rating_2006, y = ~candidate, name = "2006 Ratings", color = I("blue")) %>%
  layout(
    title = "Changes in Candidates' Approval Ratings (from 2005 to 2006)",
    xaxis = list(title = "% Approval Rating"),
    margin = list(l = 65)
  )

dumbbell_plot
```

As you can see, we can easily compare how the approval ratings for the political candidates changed over a year. Their decreases and increases are neatly organized onto 1 plot, and makes data analysis much easier.

`plotly` - Part 2: 3D Data Visualizations

In addition to unique 2D graphs such as dumbbell plots, `plotly` also provides a very unique way to analyze data that many other packages don't: it allows you to compare up to 3 variables at once. What does this mean? Well, this means that you're able to create *interactive, 3D* data visualizations with `plotly`.

Let's say we're analyzing the NBA dataset previously used in this class. So far, we've only been comparing up to 2 variables and displaying them on a 2D scatterplot. But what if we wanted to see if there is any relationship between 3 variables (player statistics)? `plotly` allows us to do that.

The NBA dataset is available here: NBA Player Statistics

Similar to the data set for the dumbbell plot, place the NBA .csv file in "data".

A 3D scatter plot for `plotly` follows the following pseudocode:

```
plot <- plot_ly(DATASET, x = ~COLUMN1, y = ~COLUMN2, z = ~COLUMN3, opacity=OPACITY, color=COLOR) %>%
  add_markers() %>%
  layout(scene = list(xaxis = list(title = XAXISNAME),
                      yaxis = list(title = YAXISNAME),
                      zaxis = list(title = ZAXISNAME)))

plot
```

Following this pseudocode, let's create our 3D scatter plot that compares Age, Salary, and Games Played (GP) in player statistics. You can click and drag the scatterplot to move it around.

```
nbadata <- read.csv("data/nba2017-player-statistics.csv")

scatter_3d <- plot_ly(nbadata, x = ~Age, y = ~Salary, z = ~GP, opacity=0.5, color=I("blue")) %>%
  add_markers() %>%
  layout(scene = list(xaxis = list(title = 'Age'),
                      yaxis = list(title = 'Salary'),
                      zaxis = list(title = 'Games Played')))

scatter_3d
```

Now, with `plotly`, it's easier than ever to observe and analyze the relationship between 3 variables.

`plotly` is also useful analyzing mathematical data as well. Let's say we want to visualize the function f(x, y) = sin(x) + cos(y) on x, y in [0, 60].

First, let's create the function and put the points in the graph into a data frame:

```
f <- function(x, y) {
  return (sin(x) + cos(y))
}

x <- c()
y <- c()
z <- c()

for (n in 0:60) {
  x <- c(x, n)
  y <- c(y, n)
  z <- c(z, f(n, n))
}

points <- data.frame(x, y, z)
```

A 3D `plotly` line graph follows the following pseudocode:

```
plot <- plot_ly(DATA, x = ~XCOLUMN, y = ~YCOLUMN, z = ~ZCOLUMN, type = 'scatter3d', mode = 'lines',
opacity = OPACITY, line = list(width = WIDTH, reverscale = FALSE))

plot
```

Now, using the pseudocode, and our point data "points" for our function, let's graph the function f(x, y) = sin(x) + cos(y) (on x, y in [1, 60]). Similar to the 3D scatterplot, you can also click and drag the function around to interact with it.

```
function_plot <- plot_ly(points, x = ~x, y = ~y, z = ~z, type = 'scatter3d', mode = 'lines',
opacity = 1, line = list(width = 6, reverscale = FALSE))

function_plot
```

In addition to 3D scatter plots, `plotly` makes it extremely easy to create 3D line graphs (with just a couple of lines of code) for both real-world and mathematical data as well. These 3D line graphs can be used to create lines of best fit, or demonstrate random walks. The fact that you can interact with it (i.e. click and drag, zoom-in/out) makes analysis much more intuitive and powerful.

Take-Home Message

The take-home message for this post is rather straightforward: data visualization in R is *much* more broad and has much more potential than simply displaying histograms, barcharts, and scatterplots. There is just so much you can do. There are numerous packages that allow you to display an appropriate visualization for your data, whether that be 2D or 3D (or even 4D).

Data visualization in R is essentially unlimited in its methodologies, and there are much more complicated and skill-intensive techniques than the ones we've seen today to represent your data appropriately. All these different packages such as `ggplot2`, `ggmap`, and `plotly` make data visualization a form of art, a personal expression of how you want to share your data. Hopefully this post has served as a short but informative introduction to just what you can do to display your data in R, and now you too can express yourself in your data in limitless ways.

Sources:

• https://www.statmethods.net/graphs/density.html

• https://datavizcatalogue.com/methods/density_plot.html

• https://en.wikipedia.org/wiki/Time_series

• https://www.statmethods.net/advstats/timeseries.html

• https://github.com/dkahle/ggmap

• https://github.com/hrbrmstr/ggalt

• https://plot.ly/r/getting-started/

• https://plot.ly/r/dumbbell-plots/#dot-and-dumbbell-plots

• https://plot.ly/r/3d-scatter-plots/

• https://plot.ly/r/3d-line-plots/

• http://r-statistics.co/Top50-Ggplot2-Visualizations-MasterList-R-Code.html