Lab: Plotly

Contents

This tutorial comes from Carson Sievert's Plotly for R Master Class.

1.1 A case study of housing sales in Texas

The `plotly` package depends on `ggplot2` which bundles a data set on monthly housing sales in Texan cities acquired from the TAMU real estate center. After the loading the package, the data is "lazily loaded"" into your session, so you may reference it by name:

```r
library(plotly)
txhousing
```

Let's see if there's any pattern in house price behavior over time:

```r
p <- ggplot(txhousing, aes(x = date, y = median)) +
  geom_line(aes(group = city), alpha = 0.2)
class(p)
```

It's be nice if we could see which city each line corresponds to when we hover. `plotly` makes this easy! Just wrap your `ggplot` object in the `ggplotly()` function:

```r
ggplotly(p)
```

If we just want the city name, we can specify exactly what to put in the tooltip:

```r
ggplotly(p, tooltip = "city")
```

We can also build `plotly` objects directly using the `plot_ly()` function along with `dplyr`-like syntax. Why would we want to? Well, for one thing, `plot_ly()` recognizes and preserves groupings created with `dplyr`'s `group_by()` function:

```r
library(dplyr)
tx <- group_by(txhousing, city)

# initiate a plotly object with date on x and median on y
p <- plot_ly(tx, x = ~date, y = ~median)

# plotly_data() returns data associated with a plotly object
plotly_data(p)
```

Since we didn't specify any mapping, the plot defaults to a scatterplot:

```r
p
```

Let's change that to a line chart. Similar to `geom_line()` in `ggplot2`, the `add_lines()` function connects (a group of) x/y pairs with lines in the order of their x values and returns the transformed `plotly` object:

```
add_lines(p, alpha = 0.2, name = "Texan Cities")
```

Want to highlight a particular line? Filtering works, and since each `add_lines()` call returns a pointer to the modified `plotly` object, we can chain calls together with pipes:

```
p <- txhousing %>%
  group_by(city) %>%
  plot_ly(x = ~date, y = ~median) %>%
  add_lines(alpha = 0.2, name = "All TX Cities", hoverinfo = "none") %>%
  filter(city == "Houston") %>%
  add_lines(name = "Houston")
p
```

Want to zoom in without losing context? Try a `rangeslider()`:

```
rangeslider(p)
```

And just so you don't think we're limited to line charts:

```
p2 <- ggplotly(ggplot(txhousing, aes(date, median)) + geom_bin2d())
p2
```

Check out The Plotly Cookbook for more details on specific plotly visualization types ("traces").

Combining views

With `htmlwidgets`

Since `plotly` objects inherit properties from an `htmlwidget` object, any method that works for arranging `htmlwidgets` also works for plotly objects. In some sense, an `htmlwidget` object is just a collection of HTML tags, and the htmltools package provides some useful functions for working with HTML tags RStudio and Inc. 2016. The `tagList()` function gathers multiple HTML tags into a tag list, and when printing a tag list inside of a knitr/rmarkdown document Xie 2016; Allaire et al. 2016, it knows to render as HTML:

```
library(htmltools)
tagList(p, p2)
```

This renders two plots, each in its own row spanning the width of the page, because each `htmlwidget` object is an HTML `<div>` tag. More often than not, it is desirable to arrange multiple plots in a given row, and there are a few ways to do that. A very flexible approach is to wrap all of your plots in a flexbox (i.e., an HTML `<div>` with `display: flex` Cascading Style Sheets (CSS) property). The `tags$div()` function from htmltools provides a way to wrap a `<div>` around both tag lists and `htmlwidget` objects, and set attributes, such as style.

```
tags$div(
  style = "display: flex; flex-wrap: wrap",
  tags$div(p, style = "width: 45%; padding: 1em;"),
  tags$div(p2, style = "width: 45%; padding: 1em;")
)
```

With `shiny`

Another way to arrange multiple `htmlwidget` objects on a single page is to leverage the `fluidPage()`, `fluidRow()`, and `column()` functions from the `shiny` package:

```r
library(shiny)
fluidPage(
  fluidRow(
    column(6, p), column(6, p2)
  )
)
```

With `subplot()`

We could also use `plotly`'s built-in `subplot()` function to generate a single plotly object with a common y-axis

```r
subplot(
  p, p2,
  shareY = TRUE
)
```

Linking multiple views with `crosstalk`

`crosstalk` is the R implementation of the powerful crossfilter JS library. Though this dataset isn't very large, we'll use it to create a `SharedData` object that allows us to propagate interaction.

```r
library(crosstalk)
shared_data <- txhousing %>%
  filter(city %in% c("Houston", "Dallas", "Galveston")) %>%
  SharedData$new(~year)
```

As far as `ggplotly()` and `plot_ly()` are concerned, `SharedData` object(s) act just like a data frame, but with a special `key` attribute attached to graphical elements. Since both interfaces are based on the layered grammar of graphics, `key` attributes can be attached at the layer level, and those attributes can also be shared across multiple views. Let's leverage both of these features to link multiple views of median house sales in various Texan cities:

```r
p <- ggplot(shared_data, aes(month, median)) +
  geom_line(aes(group = year)) +
  facet_wrap(~ city)

ggplotly(p, tooltip = "year") %>%
  highlight(color = "red")
```

We can also link different views, like those from `ggpairs`. Let's look at the `iris` dataset because it's easy on the eyes:

```r
shared_iris <- SharedData$new(iris)
p <- GGally::ggpairs(shared_iris, aes(color = Species), columns = 1:4)
highlight(ggplotly(p), on = "plotly_selected")
```

Really, we can link any collection of `htmlwidgets`:

```r
library(leaflet)
```

```r
shared_quakes <- SharedData$new(quakes)

p <- plot_ly(shared_quakes, x = ~depth, y = ~mag) %>%
  add_markers(alpha = 0.5) %>%
  highlight("plotly_selected", dynamic = TRUE)

map <- leaflet(shared_quakes) %>%
  addTiles() %>%
  addCircles()

bscols(widths = c(6, 6), p, map)
```

We'll see more about `leaflet` maps this afternoon.

Your turn!

Try some of these ideas out on your own dataset and see what you can come up with!