Interactive graphics

EDH7916

Benjamin Skinner

In this supplemental lesson, we'll make a few interactive plots using the plotly[1] library. Why might you want to make interactive plots? Aside from the fact that they are very cool, they can help you make a stronger data presentation.

How so?

First, the *wow!* factor of interactive plots can bring people in. Rather than passively reading a report or watching you present your findings, your audience can actively participate in their own understanding. Even if your interactive plot isn't any special, the process of a person using/"playing" with it means you've got their attention, at least for a moment. A well designed plot, like a well-designed interactive exhibit at a museum, can use that attention to instruct.

Second, interactive features allow you to plot along new dimensions. With a static 2D graph, we start hitting a wall at about 4 dimensions at once: x-axis, y-axis, color/shape, small multiples, *etc.* If we are smart about it, we can fit *a lot* of information into one figure, but it's difficult. With interactive features, we can add a 3rd dimension, dynamic visuals that change over time, tooltips to show point-specific information, and buttons that allow the user to change the underlying data. Interactive graphics can allow for more options since they can be changed on the fly.

The good news is that plotly in R works very similarly to ggplot. In fact, it's often trivial to convert a static ggplot figure into an interactive plotly figure. In this lesson, however, I'll show you how to use plotly's particular interface, which is a little different.

As a note, most of what I'm showing you here is just code from the R plotly website[2] that I've modified to replicate some figures we've made before. As usual, what I'm showing you only scratches the surface of what's possible. If you want to make different types of interactive figures or modify what you see here, check out the plotly web page: plotly.com/r/[3].

NOTE I've included a PDF link at the top of this page as always, but interactive plots don't play well with PDFs so you won't see the figures there. Be sure to run the code in RStudio or use the website to see the interactive figures.

Setup

Unless you have already done so, don't forget to install plotly library using `install.packages("plotly")`.

```
## ---------------------------
## libraries
## ---------------------------

library(tidyverse)
```

[1] https://plotly.com/r/
[2] https://plotly.com/r/
[3] https://plotly.com/r/

```
## ── Attaching packages ─────────────────────────────────────── tidyverse 1.3.1 ──

## ✔ ggplot2 3.3.5     ✔ purrr   0.3.4
## ✔ tibble  3.1.6     ✔ dplyr   1.0.8
## ✔ tidyr   1.2.0     ✔ stringr 1.4.0
## ✔ readr   2.1.2     ✔ forcats 0.5.1

## ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
```

```
library(haven)
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
```

As we did with the the lesson on making graphics[4], we'll use two sets of data: `hsls_small.dta` and `all_schools.csv`.

Note that since we have two data files this lesson, we'll give them unique names instead of the normal `df`:

- `df_hs` := `hsls_small.dta`
- `df_ts` := `all_schools.csv`

And as always, we are working in the `./scripts` subdirectory.

```
## ---------------------------
## directory paths
## ---------------------------

## assume we're running this script from the ./scripts subdirectory
dat_dir <- file.path("..", "data")
tsc_dir <- file.path(dat_dir, "sch_test")
```

```
## ---------------------------
## input data
## ---------------------------

## assume we're running this script from the ./scripts subdirectory
## read_dta() ==> read in Stata (*.dta) files
## read_csv() ==> read in comma separated value (*.csv) files
df_hs <- read_dta(file.path(dat_dir, "hsls_small.dta"))
df_ts <- read_csv(file.path(tsc_dir, "all_schools.csv"))
```

```
## Rows: 24 Columns: 5
## ── Column specification ────────────────────────────────────────────────────────
```

[4] https:/equant.github.io/edh7916/lessons/plotting.html

```
## Delimiter: ","
## chr (1): school
## dbl (4): year, math, read, science
##
## □  Use `spec()` to retrieve the full column specification for this data.
## □  Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Plots using plotly

Pretty much whatever plot you can make in ggplot(), you can make using plotly. We'll start with histograms to get our bearings.

Histogram

Just like ggplot starts with the function `ggplot()`, plotly starts with `plot_ly()` (notice the underscore in the function name). Like you have seen with `ggplot()`, `plot_ly()` starts with the data you want to use.

Unlike `ggplot()`, you don't use an aesthetics—`aes()`—function to map the data to axes. Instead, you set the values directly, but use a tilde (~) next to the variable name. Think of the tilde as saying, *I want this variable to be interactive in some way*. For our first histogram, we're going to plot the distribution of math test scores: `~x1txmtscor`.

Also unlike ggplot(), we set the figure type (geom in ggplot-speak) inside the `plot_ly()` function using the argument `type`. In this case, `type = "histogram"`).

We could just call `plot_ly()` and let the plot pull up in our plot window, but we'll first save it in an object, `p`. We'll then call `p`.

One thing to notice about plotly figures is that they are `*.html` files by default. This is the markup language for websites (**H**yper **T**ext **M**arkup **L**anguage). If you are using RStudio, the plot should pull up in your plot viewer. Outside of RStudio, it will open you in your web browser.

```r
## create basic histogram with plotly
p <- plot_ly(data = df_hs, x = ~x1txmtscor, type = "histogram")
```

```r
## show
p
```

This first histogram is pretty plain, but run your mouse over it. You should notice a tooltip next to your mouse. There are two numbers. These represent the x-axis range of the histogram bin and the number of observations falling into that bin (y-axis). So if you see, `(49.5 - 50, 571)`, that's telling you that 571 observations have test scores within 49.5 and 50. With ggplot(), we don't get that level of detail.

You should also see a number of controls on the upper part of the figure. These allow you to zoom in and and save the figure. If you click and drag on the figure, you can zoom in on a certain region. Take a few moments to play around with the figure and controls.

If we want to show a density histogram rather than a frequency histogram, we only need to add the option `histnorm = "probability"` as an argument. All else can stay the same.

```r
## create histogram plotly
p <- plot_ly(data = df_hs,
             x = ~x1txmtscor,
             histnorm = "probability",
             type = "histogram")
```

```r
## show
p
```

Notice that the y-axis is now on the probability scale. Similarly, the tooltip now shows the probability of the bin rather than the count of observations that fall into it.

> **QUICK EXERCISE** *Create an interactive histogram using another continuous variable in the* `hsls_small` *data set. You can make either a frequency or density histogram.*

Now that we've made a couple, let's make a nicer looking histogram. First, within the `plot_ly()` function, we'll add the argument `marker` and some options to make our histogram bars look a little different. One thing to notice about plotly options is that they tend to fall into lists. This is likely due to the fact that plotly works with other programming languages like Python and is splitting the difference between idioms. With `marker` we add two named list items: `color` and `line`, which also takes a list. With these options, we can change the fill (inside) color of the bars and edge color and line thickness.

Next, we add nicer labels using the `layout()` function. Notice that unlike ggplot, which connects items with a plus (+) sign, plotly uses the magrittr pipe (`%>%`). Inside `layout()`, we set the overall title and the x-axis label (which also use the `title` name).

Finally, we improve the tooltip. With argument `hovertemplate` we `paste()` together a new string. Anything static in the string, like `"Bin width:"` will stay static in the tooltip. So the values for bin width and probability change, we include the variables inside `%{<var>}`. For example, notice that next to `"Bin width: "` we include `%{x}`. This means that that value should change along values of `x`, our test score. It's a little tricky here since we don't actually have a `y` value that we set explicitly. That said, we know that `y` is probability in this histogram. So that we don't have a redundant name on our tooltip, we set `name = ""`.

```r
## create histogram plotly w/
## 1. better labels
## 2. add title
## 3. change color of bars
## 4. add outline to bars
## 5. improve the tooltip
p <- plot_ly(data = df_hs,
             x = ~x1txmtscor,
             histnorm = "probability",
             type = "histogram",
             marker = list(color = "#E28F41",
                           line = list(color = "#6C9AC3",
                                       width = 2)),
             name = "",
             hovertemplate = paste("Bin width: %{x}",
                                   "<br>", # add HTML <br> for line break
                                   "Probability: %{y}")) %>%
    layout(title = "Distribution of math test scores",
           xaxis = list(title = "Math test score"))
```

```r
## show
p
```

That's much nicer looking! Notice how much nicer and more user friendly the tooltip is now, too.

Two-way histogram

Plotting the difference in a continuous distribution across groups is a common task. As we've done before, let's see the difference between student math scores between students with parents who have any postsecondary degree and those without. Since we're using data that was labeled in Stata, we'll see the labels when we use count().

```r
## see the counts for each group
df_hs %>% count(x1paredu)
```

```
## # A tibble: 7 × 2
##                                       x1paredu     n
##                                      <dbl+lbl> <int>
## 1  1 [Less than high school]                   1010
## 2  2 [High school diploma or GED]              5909
## 3  3 [Associate's degree]                      2549
## 4  4 [Bachelor's degree]                       4102
## 5  5 [Master's degree]                         2116
## 6  7 [Ph.D/M.D/Law/other high lvl prof degree]  1096
## 7 NA                                           6721
```

Now that we see our options, we'll make a new variable, `pared_coll` that equals 1 if either of the student's parents have any college degree and 0 otherwise. We'll store this smaller data frame in a new object, `plot_df`, that we'll us to plot a new two-way histogram.

```r
## need to set up data
plot_df <- df_hs %>%
  ## select the columns we need
  select(x1paredu, x1txmtscor) %>%
  ## can't plot NA so will drop
  drop_na() %>%
  ## create new variable that == 1 if parents have any college
  mutate(pared_coll = ifelse(x1paredu >= 3, 1, 0)) %>%
  ## drop (using negative sign) the original variable we don't need now
  select(-x1paredu)

## show
plot_df
```

```
## # A tibble: 16,429 × 2
##    x1txmtscor pared_coll
##     <dbl+lbl>      <dbl>
##  1       59.4          1
##  2       47.7          1
##  3       64.2          1
##  4       49.3          1
##  5       62.6          1
##  6       58.1          1
##  7       49.5          0
##  8       54.6          1
##  9       53.2          0
## 10       63.8          1
## # … with 16,419 more rows
```

Unlike with ggplot, we won't just add a `group = ...` aesthetic to the plot. Instead, we'll add two separate histograms, one for each `pared_coll` group and then tell plotly to `barmode = "overlay"` in the `layout()` function. To only plot one `pared_coll` group at a time, notice that we use `plot_df %>% filter(pared_coll == ...)` in each data argument, replacing `...` with `0` and `1`.

We include `hovertemplate` arguments in each `add_histogram()` function. We also include redundant `text` and `name` arguments. With the `text` argument, we can include the `pared_coll` status in the tooltip with `%{text}`. With `name`, the legend will be properly labeled. However, adding `name` back in means that we get that annoying extra label on our tooltip. To remove it, we add `"<extra></extra>"` at the end of our `hovertemplate`.

```
## two way histogram: add one at a time
p <- plot_ly(alpha = 0.5) %>%
  ## first: when parents don't have college degree
  add_histogram(data = plot_df %>% filter(pared_coll == 0),
                x = ~x1txmtscor,
                histnorm = "probability",
                type = "histogram",
                name = "No college",
                hovertemplate = paste("No college",
                                      "<br>",
                                      "Bin width: %{x}",
                                      "<br>",
                                      "Probability: %{y}",
                                      "<extra></extra>")) %>%
  ## second: when parents have college degree
  add_histogram(data = plot_df %>% filter(pared_coll == 1),
                x = ~x1txmtscor,
                histnorm = "probability",
                type = "histogram",
                name = "Some college or more",
                hovertemplate = paste("Some college or more",
                                      "<br>",
                                      "Bin width: %{x}",
                                      "<br>",
                                      "Probability: %{y}",
                                      "<extra></extra>")) %>%
  ## tell plotly to overlay the histograms
  layout(barmode = "overlay",
         title = "Distribution of math test scores",
         xaxis = list(title = "Math test score"),
         legend = list(title = list(text = "Parent's education level")))
```

```
## show
p
```

QUICK EXERCISE *Create an interactive two-way histogram using another grouping variable in the* `hsls_small` *data set (something other than parental college). You can use an existing indicator variable or create your own.*

Box plot

Here we'll replicate the box plot that we've made before. Since the xaxis boxes and the different colors represent the same things—a different level of parental educational expectation—we'll turn off the legend with `showlegend = FALSE`.

```
## box plot using as_factor()
p <- plot_ly(data = df_hs,
             color = ~as_factor(x1paredu),
             y = ~x1txmtscor,
```

```r
             type = "box") %>%
  layout(title = "Math score by parental expectations",
         xaxis = list(title = "Parental expectations"),
         yaxis = list(title = "Math score"),
         showlegend = FALSE)
```

```r
## show
p
```

Nothing too fancy here. Our factor labels are a bit long, so we could probably make a cleaner figure if shortened them. The biggest benefit of the interactive boxplot is that the tooltip shows the values of the various levels: `min`, `lower fence`, `Q1`, `median`, `Q3`, `upper fence`, and `max`. That said, it's a bit busy, so you'll have to decide whether the benefit outweighs the cost.

Scatter

Interactive scatter plots can be really useful if for no other reason than they allow you to zoom in and pick out specific values, which is great for identifying outliers.

As with a prior lesson, we'll limit our data to only a 10% sample. Remember that because it's a random process, your data and resulting figures will look a little different from mine.

```r
## sample 10% to make figure clearer
df_hs_10 <- df_hs %>%
    ## drop observations with missing values for x1stuedexpct
    drop_na(x1stuedexpct) %>%
    ## sample
    sample_frac(0.1)
```

To make the scatter plot, we'll start by setting `type = "scatter"` and `mode = "markers"`.

```r
## scatter
p <- plot_ly(data = df_hs_10,
             x = ~x1ses,
             y = ~x1txmtscor,
             type = "scatter",
             mode = "markers")
```

```r
## show
p
```

Like the other plotly figures, we can hover over specific points to get their values. Let's do another version, but cleaned up: better axes, better tooltip, and better colors. One new thing: we set `zeroline = FALSE` in the `layout()` function. This prevents the figure from drawing a bold vertical line at 0. You can remove the option and have the bold line if you want, but it seemed unnecessary here so I've dropped it.

```r
## scatter
p <- plot_ly(data = df_hs_10,
             x = ~x1ses,
             y = ~x1txmtscor,
             type = "scatter",
             mode = "markers",
             ## set point options
             ## size: radius
             ## color: fill color
             ## line: circumference line width and color
             marker = list(size = 10,
```

```
                           color = "#E28F41",
                           line = list(color = "#6C9AC3",
                                       width = 2)),
             ## turn off side name on hover
             name = "",
             hovertemplate = paste("SES: %{x}",
                                   "<br>", # add <br> for line break
                                   "Math: %{y}")) %>%
  layout(title = "Math scores as function of SES",
         xaxis = list(title = "SES",
                      zeroline = FALSE),
         yaxis = list(title = "Math score"))
```

```
## show
p
```

Let's add another dimension: color representing whether the student planned to graduate from college or not. First, we'll check the levels of our key variable, `x1stuedexpct`).

```
## see student base year plans
df_hs %>%
  count(x1stuedexpct)
```

```
## # A tibble: 12 × 2
##                                    x1stuedexpct     n
##                                       <dbl+lbl> <int>
##  1  1 [Less than high school]                      93
##  2  2 [High school diploma or GED]               2619
##  3  3 [Start an Associate's degree]               140
##  4  4 [Complete an Associate's degree]           1195
##  5  5 [Start a Bachelor's degree]                 115
##  6  6 [Complete a Bachelor's degree]             3505
##  7  7 [Start a Master's degree]                   231
##  8  8 [Complete a Master's degree]               4278
##  9  9 [Start Ph.D/M.D/Law/other prof degree]      176
## 10 10 [Complete Ph.D/M.D/Law/other prof degree]  4461
## 11 11 [Don't know]                               4631
## 12 NA                                            2059
```

We see that `x1stuedexpct >= 6` means a student plans to earn a Bachelor's degree or higher. But since we need to account for the fact that 11 means "I don't know", we need to make sure our test includes `x1stuedexpct < 11`. Remember from a prior lesson that we can connect these two statements together with the operator `&`. Let's create our new variable.

```
## create variable for students who plan to graduate from college
df_hs_10 <- df_hs_10 %>%
  mutate(plan_col_grad = ifelse(x1stuedexpct >= 6 & x1stuedexpct < 11,
                                "Yes",        # if T: "Yes"
                                "No"))        # if F: "No"
```

Now that we have our new variable `plan_col_grad`, we can add it the `color` aesthetic. How do we specify the colors? Notice that we created an object called `pal` with two Hex color codes. In the next line, we named these colors in the vector with "Yes" and "No" to match our new `plan_col_grad` variable. We add this to the `colors` argument. That will assign the first color in pal to all the `Yes`s and the second color in pal to all the `No`s.

This time, we're also making our scatterplot a little differently. We're using `add_trace()` this time. Most of the arguments are the same as before, but are in the `add_trace()` function rather than in the `plot_ly()` function.

```
## set color palette with names
pal <- c("#E28F41","#6C9AC3")
pal <- setNames(pal, c("Yes", "No"))

## scatter plot
p <- plot_ly() %>%
  add_trace(data = df_hs_10,
            x = ~x1ses,
            y = ~x1txmtscor,
            color = ~plan_col_grad,   # color changes by college plans
            colors = pal,             # using pal from above
            type = "scatter",
            mode = "markers",
            hovertemplate = paste("SES: %{x}",
                                  "<br>",
                                  "Math: %{y}",
                                  "<extra></extra>")) %>%
  layout(title = "Math scores as function of SES",
         xaxis = list(title = "SES",
                      zeroline = FALSE), # turn off bold zero line
         yaxis = list(title = "Math score"),
         legend = list(title = list(text = "Plans to graduate from college?")))
```

```
## show
p
```

There's quite a bit of overlap, but we can see that students who plan to graduate from college tend to have a higher SES and math scores.

3D

Let's take full advantage of plotly and make a 3D plot. It would be ideal if we had a third continuous variable to plot against, but with our limited data set, we'll settle for the number of months between HS graduation and first college enrollment on the z axis.

To add the third dimension repeats most of the same code. We're back using `plot_ly()` for most of the set up, adding `z = ~x4hs2psmos` and a new line to our tooltip. We call the points with the `add_markers()` function.

Labeling the axes is slightly different. For a 3D plot, all the x, y, and z-axis labels need to go inside the `scene` argument list. But other than that, the labels are the same.

```
## set color palette with names
pal <- c("#E28F41","#6C9AC3")
pal <- setNames(pal, c("Yes", "No"))

## scatter along three dimensions; most settings inside plot_ly() function now
p <- plot_ly(data = df_hs_10,
             x = ~x1ses,
             y = ~x1txmtscor,
             z = ~x4hs2psmos,
             color = ~plan_col_grad,
             colors = pal,
             ## add extra row to tool tip
```

```r
            hovertemplate = paste("SES: %{x}",
                                  "<br>",
                                  "Math: %{y}",
                                  "<br>",
                                  "HS to College: %{z} months",
                                  "<extra></extra>"),
            ## make marker a little smaller
            marker = list(size = 3)) %>%
  ## now tell plot_ly to add points as markers
  add_markers() %>%
  ## in 3D, set axis titles inside scene() argument
  layout(title = "Math scores as function of SES",
         scene = list(xaxis = list(title = "SES"),
                      yaxis = list(title = "Math score"),
                      zaxis = list(title = "Months between HS and college")),
         legend = list(title = list(text = "Plans to graduate from college?")))
```

```r
## show
p
```

Click on the figure and move it around. You'll see that in addition to zooming in and out, you can rotate the figure to better see the relationship across the three variables. For this particular example, the relationship is difficult to see (3D figures aren't always the best option!). Most students who enrolled in college did so pretty soon after graduating high school. Just by looking, there doesn't seem to be much difference between the two groups (though some more formal testing might be in order).

QUICK EXERCISE *Create an interactive 3D scatter plot using SES, math scores, and a categorical variable with more than two groups. This means your legend should more than two items and your graph should have more than two colors. Note that you likely need to increase the number of colors in* `pal` *(you can find more hex codes at https://htmlcolorcodes.com) or remove the argument altogether.*

Line graph

We can also make line graphs with plotly. For these, we'll once again use our school test score data. We won't go through all the iterations in the first graphing lesson, but know that you can convert those to interactive figures as well.

We'll begin by showing our data (which is wide-ish) and creating a fully long version of the data.

```r
## show test score data
df_ts
```

```
## # A tibble: 24 × 5
##    school        year  math  read science
##    <chr>        <dbl> <dbl> <dbl>   <dbl>
##  1 Bend Gate     1980   515   281     808
##  2 Bend Gate     1981   503   312     814
##  3 Bend Gate     1982   514   316     816
##  4 Bend Gate     1983   491   276     793
##  5 Bend Gate     1984   502   310     788
##  6 Bend Gate     1985   488   280     789
##  7 East Heights  1980   501   318     782
##  8 East Heights  1981   487   323     813
```

```
## 9 East Heights 1982 496 294 818
## 10 East Heights 1983 497 306 795
## # … with 14 more rows
```

```
## reshape data long (as we've done in a prior lesson)
df_ts_long <- df_ts %>%
  pivot_longer(cols = c("math","read","science"), # cols to pivot long
               names_to = "test",                  # where col names go
               values_to = "score") %>%            # where col values go
  group_by(test) %>%
  mutate(score_std = (score - mean(score)) / sd(score)) %>%
  group_by(test, school) %>%
  arrange(year) %>%
  mutate(score_year_one = first(score),
         ## note that we're using score_year_one instead of mean(score)
         score_std_sch = (score - score_year_one) / sd(score)) %>%
  ungroup

## show
df_ts_long
```

```
## # A tibble: 72 × 7
##    school        year test    score score_std score_year_one score_std_sch
##    <chr>        <dbl> <chr>   <dbl>     <dbl>          <dbl>         <dbl>
##  1 Bend Gate     1980 math      515     1.40             515             0
##  2 Bend Gate     1980 read      281    -0.863            281             0
##  3 Bend Gate     1980 science   808     0.759            808             0
##  4 East Heights  1980 math      501     0.115            501             0
##  5 East Heights  1980 read      318     1.34             318             0
##  6 East Heights  1980 science   782    -0.735            782             0
##  7 Niagara       1980 math      514     1.31             514             0
##  8 Niagara       1980 read      292    -0.208            292             0
##  9 Niagara       1980 science   787    -0.448            787             0
## 10 Spottsville   1980 math      498    -0.161            498             0
## # … with 62 more rows
```

First, we'll only plot one school, Bend Gate. As we did above, we won't add a grouping argument for each test. Instead, we'll add a unique `add_trace()` for each test—math, reading, and science—which are added to the `y` argument. In the first `add_trace()`, we'll include our wide data and `filter(school == "Bend Gate")`. We don't need to include data in the other functions since it will be included by default. We set `type = "scatter"` and `mode = "lines"` to make our line graphs with `x = ~year`. Finally, we add the argument `hovermode = "x unified"` to the `layout()` function, which connects our line information in the tooltip.

```
## scatter plot
p <- plot_ly() %>%
  ## add data for Bend Gate only; y == math
  add_trace(data = df_ts %>% filter(school == "Bend Gate"),
            x = ~year,
            y = ~math,
            name = "Math",
            type = "scatter",
            mode = "lines") %>%
  ## repeated, but this time y == reading
  add_trace(x = ~year,
            y = ~read,
            name = "Reading",
```

```
           type = "scatter",
           mode = "lines") %>%
  ## repeated, but this time y == science
  add_trace(x = ~year,
           y = ~science,
           name = "Science",
           type = "scatter",
           mode = "lines") %>%
  ## add unified hovermode so that tooltip for all test scores pops ups
  layout(title = "Test scores at Bend Gate: 1980 - 1985",
        xaxis = list(title = "Year"),
        yaxis = list(title = "Score"),
        legend = list(title = list(text = "Test")),
        hovermode = "x unified")
```

```
## show
p
```

Notice how the tooltip includes information for all tests in the same year, no matter which line you hover over? That's due to the `hovermode = "x unified"` argument. You can omit that argument and the tooltip will work as normal (local to the point under your mouse pointer), but in this case, it makes sense to link them.

How to add all the school values in one plot like we did with ggplot's `facet_wrap()`? It's a little trickier, but we can do it with a loop (see the lesson on functional programming[5] for a review of loops).

First, we'll store our school names in an object, `schools`. Next, we'll initialize a blank list with `plot_list <- list()` that will store our school-specific plots. Inside the loop, we'll filter the data with `filter(school == i)` where `i` will equal whichever school we're on in the loop: Bend Gate the first iteration, East Heights the second, and so on.

So that we don't have repeated legends in the final figure, we do a few things. First, we add the argument `legendgroup = ~test` so that plotly knows the tests (math, reading, and science) are grouped together within school. Next, we set `showlegend = if_first`. What's `if_first`? It's a Boolean (TRUE/FALSE) that's only `TRUE` on the first loop iteration, when `i == 1`, and `FALSE` otherwise. This means that we only include a legend with the first school figure. Since we're going to put them all together at the end, that one will suffice for all of them. Finally, we use `add_annotations()` with various options so that the name of each school is printed on the top if its own figure (`text = i`).

Once the loop has run, we use the function `subplot()` to join the plots into one figure. With argument `nrows = 2`, we'll end up with a 2x2 grid.

```
## set up vector of school names
schools <- c("Bend Gate", "East Heights", "Niagara", "Spottsville")

## init list to hold plots
plot_list <- list()

## loop through schools to make each plot at a time
for (i in schools) {

  ## TRUE if first school, otherwise FALSE;
  ## use so we only end up with one legend
  if_first <- (i == 1)
```

[5] https://equant.github.io/edh7916/lessons/programming.html

```r
  ## store scatter plot in list using index i
  plot_list[[i]] <- plot_ly() %>%
    ## filter to school i (one school at a time)
    add_trace(data = df_ts_long %>% filter(school == i),
              x = ~year,
              y = ~score_std_sch,
              color = ~test,
              legendgroup = ~test,
              text = ~score,
              showlegend = if_first,   # only TRUE the first time
              type = "scatter",
              mode = "lines",
              ## notice that we include scaled and actual score in hover
              hovertemplate = paste("Year: %{x}",
                                    "<br>",
                                    "Score (scaled): %{y}",
                                    "<br>",
                                    "Score (actual): %{text}")) %>%
    layout(xaxis = list(title = "Year"),
           yaxis = list(title = "Score"),
           legend = list(title = list(text = "Test"))) %>%
    ## settings to add school name to title of each subplot
    add_annotations(text = i,
                    x = 0,
                    y = 1,
                    yref = "paper",
                    xref = "paper",
                    xanchor = "middle",
                    yanchor = "top",
                    showarrow = FALSE,
                    font = list(size = 15))
}

## combine subplots into on main plot like ggplot() facet_wrap()
p <- subplot(plot_list[[1]], plot_list[[2]], plot_list[[3]], plot_list[[4]],
             nrows = 2)
```

```r
## show
p
```

Okay! The figure is a little scrunched on my website, but it might look better by itself on a larger monitor. We could also play with some sizing, but since HTML is sized on the fly and unique to each screen, that may not be worth our time.

What is particularly cool about this figure is that even though we plot the first-year standardized scores, we can include the real scores in the tooltip. That let's us have the best of both worlds: the line graph shows the relationship we care about (how have scores changed relative to each other and the first year?), but have the real scores at our finger tips.

Tables

Lastly, we can also use plotly to make a table. While perhaps not as exiting as the figures, the table is interactive in that it prints pretty (includes a scroll bar) and that the user can drag to reorder columns. As a demo, we'll make a table using the raw test score data underlying the line graph we just made.

To make a table, you need to three key components:

1. The data you want to use (`df_ts`)
2. The header you want plus its formatting
3. The formatting for the table cells

For 1 and 3, most of the formatting is simply to change the background and font colors. For the `values` in the `header` list, we'll use the names from our wide-ish data frame. So that they are capitalized, we'll use `str_to_title()`, which capitalizes the first letter only: `str_to_title(names(df_ts))`.

For the data in the `cells`, well convert our data frame to a matrix using `as.matrix()`. Because of a quirk of the program, we'll need to transpose the matrix with `t()`. Finally, so we don't repeat the column names as the first (remember, we've already included them in header), we'll `unname()` the data frame. All together, it's `t(as.matrix(unname(df_ts)))`.

```
## make interactive table
tab <- plot_ly(
  type = "table",
  ## adjust how header row looks
  header = list(
    ## use str_to_title() with tibble column names: math --> Math
    values = c(str_to_title(names(df_ts))),
    ## left align school names, and center all other columns (hence -1)
    align = c("left", rep("center", ncol(df_ts) - 1)),
    ## make vertical lines thicker
    line = list(width = 1, color = "black"),
    ## fill color with blue
    fill = list(color = "rgba(108, 154, 195, 0.8)"),
    ## set font to sans serif with bigger size and white color
    font = list(family = "Arial", size = 14, color = "white")),
  ## adjust how cells look
  cells = list(
    ## use df_ts, but...
    ## 1. convert to matrix,
    ## 2. transpose using t(),
    ## 3. drop names (already in header row)
    values = t(as.matrix(unname(df_ts))),
    ## same alignment as header
    align = c("left", rep("center", ncol(df_ts) - 1)),
    ## same vertical line setup as header
    line = list(color = "black", width = 1),
    ## fill first column different color to stand out
    fill = list(color = c("rgba(108, 154, 195, 0.5)",
                          "rgba(226, 143, 65, 0.5)")),
    ## same font as header, but smaller and different color
    font = list(family = "Arial", size = 12, color = c("black"))
  ))
```

```
## show
tab
```

If you hover over a cell and then click, you should be able to drag and drop the columns into a new order. It seems a bit much for this small data set, but having an "active" data table that you can manipulate could come in handy during meetings or your own research process.