plotly for R

Carson Sievert

Contents

Overview

This website explains and partially documents the R package **plotly**, a high-level interface to the open source JavaScript graphing library plotly.js (which powers plot.ly). The R package already has numerous examples and documentation on https://plot.ly/r and https://plot.ly/ggplot2, but this website provides more of a cohesive narrative to help explain fundamental concepts and recent developments. By reading from start to finish, readers new to R and plotly should be able to get up and running fairly quickly. That being said, advanced R and plotly users should still find the majority of this material useful and informative. I highly recommend copying/pasting examples into your R console, and modifying them as you read along, to aid the learning process.

Installation

If you have R installed, you can install the stable release of **plotly** by typing this in your R console:

```
install.packages("plotly")
```

Or you can install the development release via the devtools package:

```
if (!require("devtools")) install.packages("devtools")
devtools::install_github("ropensci/plotly")
```

The version of the R package used to build this site is:

```
packageVersion("plotly")
#> [1] '4.5.5.9000'
```

Get started

To ensure plotly is installed correctly, try loading the package and creating this example by pasting the code inside your R console.

```
library(plotly)
plot_ly(z = ~volcano)
```

plotly uses the htmlwidget framework, which allows plots to work seamlessly and consistently in various contexts (e.g., R Markdown documents, shiny apps, inside RStudio, or any other R command prompt) without an internet connection. IPython/Jupyter notebook users should wrap plots with the `embed_notebook()` function to embed them inline inside a notebook.

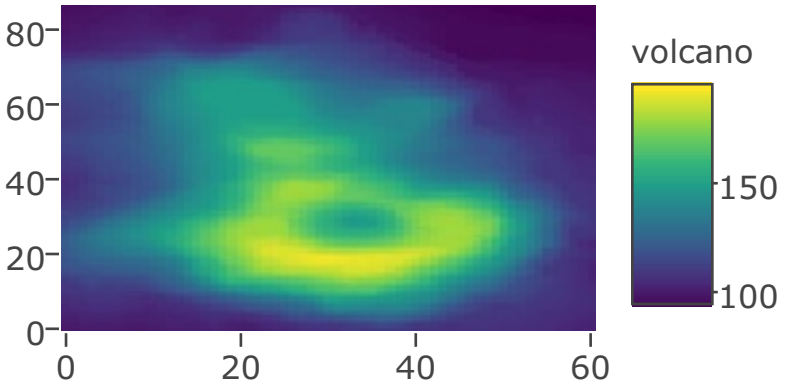


Figure 1:

plot.ly for collaboration

plot.ly subscribers can use the `plotly_POST()` function to publish plots from R onto plotly's web platform. This platform makes it very easy to host/share your graphs, collaborate with others, and is free to use for public graphs.[1] Once a plot is hosted on your account, others may copy/fork your graph to their account (with the right permissions) using a friendly user-interface. Here is a quick demonstration of that workflow from inside RStudio:

As long as you can view a plot hosted on http://plot.ly, you can bring the data behind with plot into R via the `get_figure()` function. This makes it easy to access and modify plots created with *any* plotly.js interface (e.g., Python, MATLAB, Julia, Scala, etc) from your R console.

Not only is this web-based user-interface to plotly.js useful for collaborating with others, but it is also useful for completing tasks that are cumbersome to do at the command-line. For instance, annotations can be added to any plot via a point-and-click interface:

[1] If you need privacy or customer support, pricing options

Chapter 1

Two approaches, one object

There are two main ways to initiate a plotly object in R. The `plot_ly()` function transforms *data* into a plotly object, while the `ggplotly()` function transforms a *ggplot object* into a plotly object (**?**); (**?**). Regardless how plotly object is created, printing it results in an interactive web-based visualization with tooltips, zooming, and panning enabled by default. It is also possible to enable more advanced interactive techniques, such as animation and linked highlighting. This chapter discusses some of the philosophy behind each approach, explores some of their similarities, and explains why understanding both approaches is extremely powerful.

The initial inspiration for the `plot_ly()` function was to support plotly.js chart types that **ggplot2** doesn't support, such as 3D surface and mesh plots. Over time, this effort snowballed into an interface to the entire plotly.js graphing library with additional abstractions inspired by the grammar of graphics (**?**). This newer "non-ggplot2" interface to plotly.js is currently not, and may never be, as fully featured as **ggplot2**. Since we can already translate a fairly large amount of ggplot objects to plotly objects, I'd rather not reinvent those same abstractions, and focus providing useful tools for advanced interactive graphics.

The next section uses a case study to introduce some of the similarities between `ggplotly()` and `plot_ly()`, dives into the cognitive framework behind `plot_ly()`, and also demonstrates how to extend `ggplotly()` with functions that can modify plotly objects.

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:

```
library(plotly)
txhousing
#> # A tibble: 8,602 × 9
#>      city  year month sales   volume median listings inventory  date
#>     <chr> <int> <int> <dbl>    <dbl>  <dbl>    <dbl>     <dbl> <dbl>
#> 1 Abilene  2000     1    72  5380000  71400      701       6.3  2000
#> 2 Abilene  2000     2    98  6505000  58700      746       6.6  2000
#> 3 Abilene  2000     3   130  9285000  58100      784       6.8  2000
#> 4 Abilene  2000     4    98  9730000  68600      785       6.9  2000
#> 5 Abilene  2000     5   141 10590000  67300      794       6.8  2000
#> 6 Abilene  2000     6   156 13910000  66900      780       6.6  2000
#> # ... with 8,596 more rows
```

In attempt to understand house price behavior over time, we could plot `date` on x, `median` on y, and group

the lines connecting these x/y pairs by `city`. Using **ggplot2**, we can *initiate* a ggplot object with the `ggplot()` function which accepts a data frame and a mapping from data variables to visual aesthetics. By just initiating the object, **ggplot2** won't know how to geometrically represent the mapping until we add a layer to the plot via one of `geom_*()` (or `stat_*()`) functions (in this case, we want `geom_line()`). In this case, it is also a good idea to specify alpha transparency so that 5 lines plotted on top of each other appear as solid black, to help avoid overplotting.

If you're new to **ggplot2**, the ggplot2 cheatsheet provides a nice quick overview. The online docs or R graphics cookbook are helpful for learning by example, and the ggplot2 book provides a nice overview of the conceptual underpinnings.

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

1.1.1 The `ggplotly()` function

Now that we have a valid **ggplot2** object, `p`, the **plotly** package provides the `ggplotly()` function which converts a ggplot object to a plotly object. By default, it supplies the entire aesthetic mapping to the tooltip, but the `tooltip` argument provides a way to restrict tooltip info to a subset of that mapping. Furthermore, in cases where the statistic of a layer is something other than the identity function (e.g., `geom_bin2d()` and `geom_hex()`), relevant "intermediate" variables generated in the process are also supplied to the tooltip. This provides a nice mechanism for decoding visual aesthetics (e.g., color) used to represent a measure of interest (e.g, count/value). In Figure **??**, the `subplot()` function from the **plotly** package (discussed in more detail in subplots), which accepts a collection of ggplot and/or plotly objects, helps to concisely display tooltips for a number of scenarios, and how to suppress them.

```
subplot(
  p, ggplotly(p, tooltip = "city"),
  ggplot(txhousing, aes(date, median)) + geom_bin2d(),
  ggplot(txhousing, aes(date, median)) + geom_hex(),
  nrows = 2, shareX = TRUE, shareY = TRUE,
  titleY = FALSE, titleX = FALSE
)
```

Although **ggplot2** does not have a `text` aesthetic, the `ggplotly()` function recognizes this aesthetic and displays it in the tooltip by default. In addition to providing a way to supply "meta" information, it also provides a way to customize your tooltips (do this by restricting the tooltip to the text aesthetic – `ggplotly(p, tooltip = "text")`)

The `ggplotly()` function translates most things that you can do in **ggplot2**, but not quite everything. To help demonstrate the coverage, I've built a plotly version of the ggplot2 docs. This version of the docs displays the `ggplotly()` version of each plot in a static form (to reduce page loading time), but you can click any plot to view its interactive version. The next section deomnstrates how to create plotly.js visualizations via the R package, without **ggplot2**, via the `plot_ly()` interface. We'll then leverage those concepts to extend `ggplotly()`.

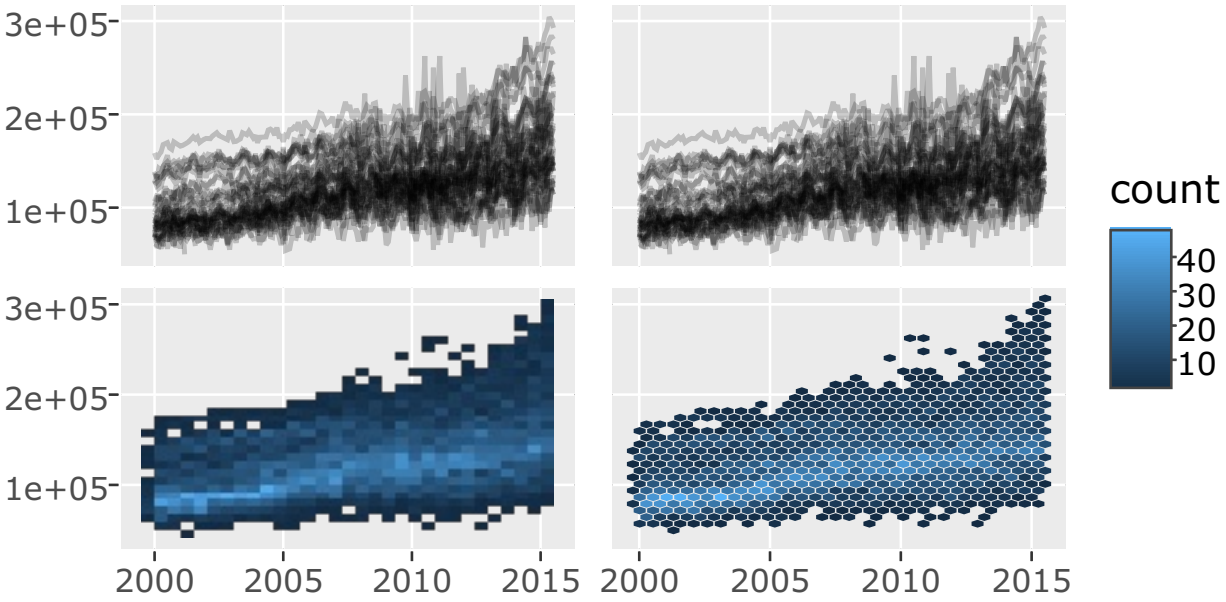


Figure 1.1: Monthly median house price in the state of Texas. The top row displays the raw data (by city) and the bottom row shows 2D binning on the raw data. The binning is helpful for showing the overall trend, but hovering on the lines in the top row helps reveal more detailed information about each city.

1.1.2 The `plot_ly()` interface

1.1.2.1 The Layered Grammar of Graphics

The cognitive framework underlying the `plot_ly()` interface draw inspiration from the layered grammar of graphics (**?**), but in contrast to `ggplotly()`, it provides a more flexible and direct interface to plotly.js. It is more direct in the sense that it doesn't call **ggplot2**'s sometimes expensive plot building routines, and it is more flexible in the sense that data frames are not required, which is useful for visualizing matrices, as shown in Get Started. Although data frames are not required, it is recommended to use them whenever possible, especially when constructing a plot with multiple layers or groups.

When a data frame is associated with a **plotly** object, it allows us to manipulate the data underlying that object in the same way we would directly manipulate the data. Currently, `plot_ly()` borrows semantics from and provides special plotly methods for generic functions in the **dplyr** and **tidyr** packages (**?**); (**?**). Most importantly, `plot_ly()` recognizes and preserves groupings created with **dplyr**'s `group_by()` function.

```
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, note the group attribute!
plotly_data(p)
#> Source: local data frame [8,602 x 9]
#> Groups: city [46]
#>
#>       city  year month sales   volume median listings inventory  date
#>      <chr> <int> <int> <dbl>    <dbl>  <dbl>    <dbl>     <dbl> <dbl>
#> 1 Abilene  2000     1    72  5380000  71400      701       6.3  2000
#> 2 Abilene  2000     2    98  6505000  58700      746       6.6  2000
#> 3 Abilene  2000     3   130  9285000  58100      784       6.8  2000
#> 4 Abilene  2000     4    98  9730000  68600      785       6.9  2000
```

```
#> 5 Abilene  2000     5   141 10590000  67300      794       6.8  2000
#> 6 Abilene  2000     6   156 13910000  66900      780       6.6  2000
#> # ... with 8,596 more rows
```

Defining groups in this fashion ensures `plot_ly()` will produce at least one graphical mark per group.[1] So far we've specified `x`/`y` attributes in the plotly object `p`, but we have not yet specified the geometric relation between these x/y pairs. 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, which is useful when plotting time series as shown in Figure **??**.

```
# add a line highlighting houston
add_lines(
  # plots one line per city since p knows city is a grouping variable
  add_lines(p, alpha = 0.2, name = "Texan Cities", hoverinfo = "none"),
  name = "Houston", data = filter(txhousing, city == "Houston")
)
```

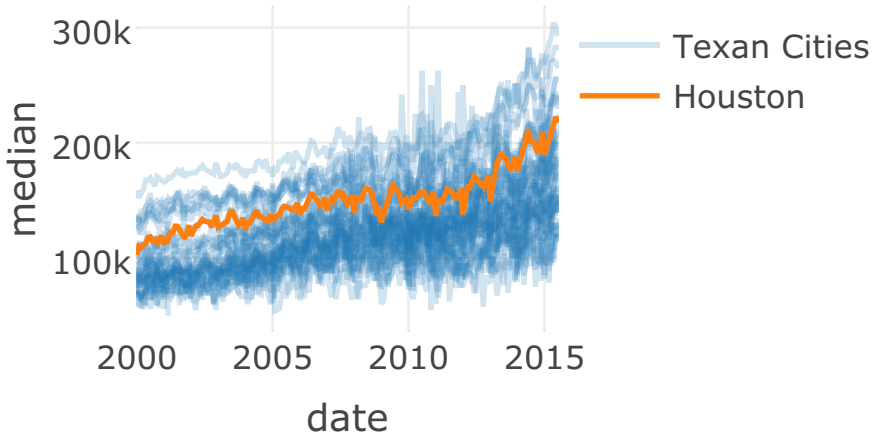


Figure 1.2: Monthly median house price in Houston in comparison to other Texan cities.

The **plotly** package has a collection of `add_*()` functions, all of which inherit attributes defined in `plot_ly()`. These functions also inherit the data associated with the plotly object provided as input, unless otherwise specified with the `data` argument. I prefer to think about `add_*()` functions like a layer in **ggplot2**, which is slightly different, but related to a plotly.js trace. In Figure **??**, there is a 1-to-1 correspondence between layers and traces, but `add_*()` functions do generate numerous traces whenever mapping a discrete variable to a visual aesthetic (e.g., color). In this case, since each call to `add_lines()` generates a single trace, it makes sense to `name` the trace, so a sensible legend entry is created.

In the first layer of Figure **??**, there is one line per city, but all these lines belong a single trace. We *could have* produced one trace for each line, but this is way more computationally expensive because, among other things, each trace produces a legend entry and tries to display meaningful hover information. It is much more efficient to render this layer as a single trace with missing values to differentiate groups. In fact, this is exactly how the group aesthetic is translated in `ggplotly()`; otherwise, layers with many groups (e.g., `geom_map()`) would be slow to render.

1.1.2.2 The data-plot-pipeline

Since every **plotly** function modifies a plotly object (or the data underlying that object), we can express complex multi-layer plots as a sequence (or, more specifically, a directed acyclic graph) of data manipulations

[1] In practice, it's easy to forget about "lingering" groups (e.g., `mtcars %>% group_by(vs, am) %>% summarise(s = sum(mpg))`), so in some cases, you may need to `ungroup()` your data before plotting it.

and mappings to the visual space. Moreover, **plotly** functions are designed to take a plotly object as input, and return a modified plotly object, making it easy to chain together operations via the pipe operator (%>%) from the **magrittr** package (**?**). Consequently, we can re-express Figure **??** in a much more readable and understandable fashion.

```
allCities <- txhousing %>%
  group_by(city) %>%
  plot_ly(x = ~date, y = ~median) %>%
  add_lines(alpha = 0.2, name = "Texan Cities", hoverinfo = "none")

allCities %>%
  filter(city == "Houston") %>%
  add_lines(name = "Houston")
```

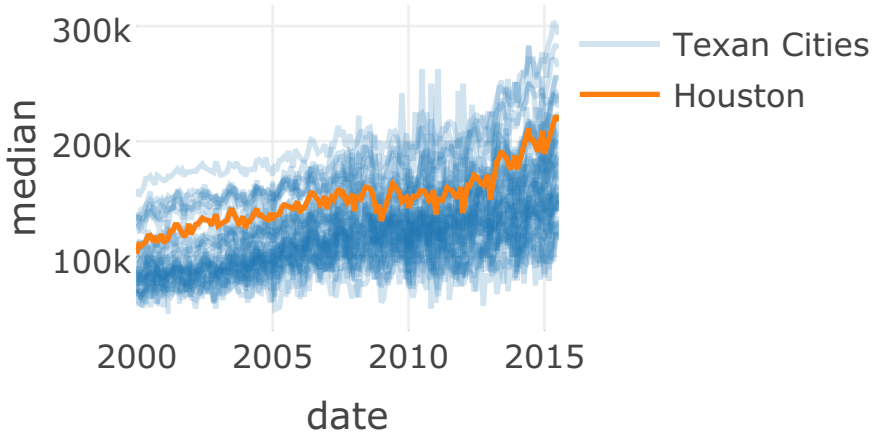


Figure 1.3:

Sometimes the directed acyclic graph property of a pipeline can be too restrictive for certain types of plots. In this example, after filtering the data down to Houston, there is no way to recover the original data inside the pipeline. The `add_fun()` function helps to work-around this restriction[2] – it works by applying a function to the plotly object, but does not affect the data associated with the plotly object. This effectively provides a way to isolate data transformations within the pipeline[3]. Figure **??** uses this idea to highlight both Houston and San Antonio.

```
allCities %>%
  add_fun(function(plot) {
    plot %>% filter(city == "Houston") %>% add_lines(name = "Houston")
  }) %>%
  add_fun(function(plot) {
    plot %>% filter(city == "San Antonio") %>% add_lines(name = "San Antonio")
  })
```

It is useful to think of the function supplied to `add_fun()` as a "layer" function – a function that accepts a plot object as input, possibly applies a transformation to the data, and maps that data to visual objects. To make layering functions more modular, flexible, and expressive, the `add_fun()` allows you to pass additional arguments to a layer function. Figure **??** makes use of this pattern, by creating a reusable function for layering both a particular city as well as the first, second, and third quartile of median monthly house sales (by city).

[2]Credit to Winston Chang and Hadley Wickham for this idea. The `add_fun()` is very much like `layer_f()` function in **ggvis**.
[3]Also, effectively putting a pipeline inside a pipeline

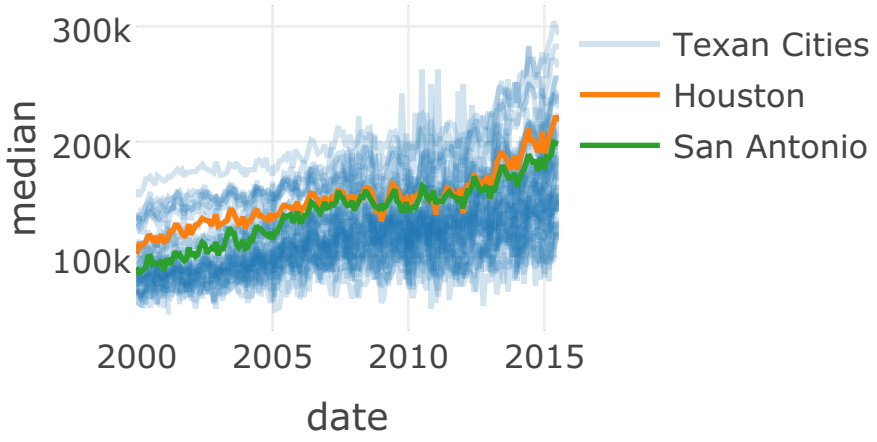


Figure 1.4: Monthly median house price in Houston and San Antonio in comparison to other Texan cities.

```
# reusable function for highlighting a particular city
layer_city <- function(plot, name) {
  plot %>% filter(city == name) %>% add_lines(name = name)
}

# reusable function for plotting overall median & IQR
layer_iqr <- function(plot) {
  plot %>%
    group_by(date) %>%
    summarise(
      q1 = quantile(median, 0.25, na.rm = TRUE),
      m = median(median, na.rm = TRUE),
      q3 = quantile(median, 0.75, na.rm = TRUE)
    ) %>%
    add_lines(y = ~m, name = "median", color = I("black")) %>%
    add_ribbons(ymin = ~q1, ymax = ~q3, name = "IQR", color = I("black"))
}

allCities %>%
  add_fun(layer_iqr) %>%
  add_fun(layer_city, "Houston") %>%
  add_fun(layer_city, "San Antonio")
```

A layering function does not have to be a data-plot-pipeline itself. Its only requirement on a layering function is that the first argument is a plot object and it returns a plot object. This provides an opportunity to say, fit a model to the plot data, extract the model components you desire, and map those components to visuals. Furthermore, since **plotly**'s `add_*()` functions don't require a data.frame, you can supply those components directly to attributes (as long as they are well-defined), as done in Figure **??** via the **forecast** package (**?**).

```
library(forecast)
layer_forecast <- function(plot) {
  d <- plotly_data(plot)
  series <- with(d,
    ts(median, frequency = 12, start = c(2000, 1), end = c(2015, 7))
  )
  fore <- forecast(ets(series), h = 48, level = c(80, 95))
  plot %>%
```

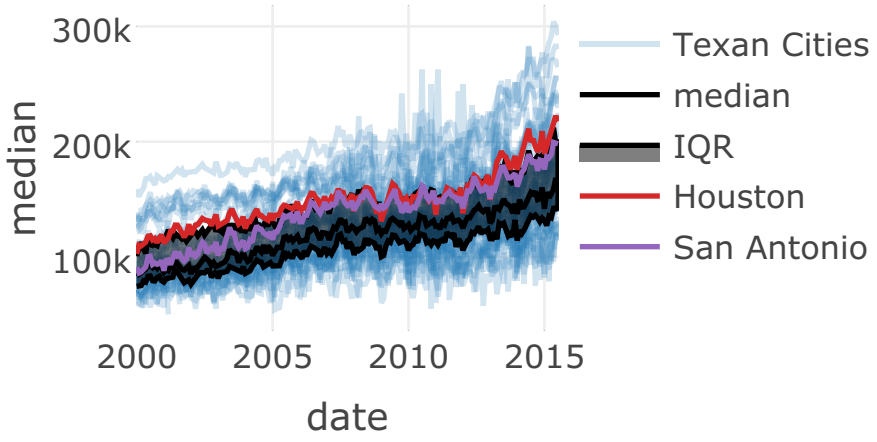


Figure 1.5: First, second, and third quartile of median monthly house pirce in Texas.

```
    add_ribbons(x = time(fore$mean), ymin = fore$lower[, 2],
                ymax = fore$upper[, 2], color = I("gray95"),
                name = "95% confidence", inherit = FALSE) %>%
    add_ribbons(x = time(fore$mean), ymin = fore$lower[, 1],
                ymax = fore$upper[, 1], color = I("gray80"),
                name = "80% confidence", inherit = FALSE) %>%
    add_lines(x = time(fore$mean), y = fore$mean, color = I("blue"),
              name = "prediction")
}

txhousing %>%
  group_by(city) %>%
  plot_ly(x = ~date, y = ~median) %>%
  add_lines(alpha = 0.2, name = "Texan Cities", hoverinfo = "none") %>%
  add_fun(layer_iqr) %>%
  add_fun(layer_forecast)
```

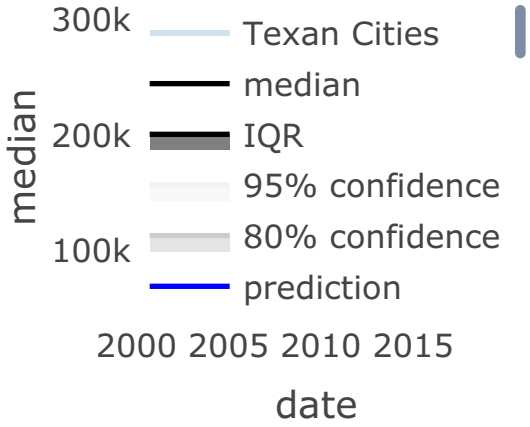


Figure 1.6: Layering on a 4-year forecast from a exponential smoothing state space model.

In summary, the "data-plot-pipeline" is desirable for a number of reasons: (1) makes your code easier to read and understand, (2) encourages you to think of both your data and plots using a single, uniform data structure, which (3) makes it easy to combine and reuse different pipelines, and (4) provides a natural

mechanism for implementing the pipeline(s) necessary in interactive graphics system with support for Linked Highlighting (**?**). As it turns out, we can even use these ideas when creating a plotly object via `ggplotly()`, as discused in the next section Extending `ggplotly()`.

1.2 Extending `ggplotly()`

1.2.1 Customizing the layout

Since the `ggplotly()` function returns a plotly object, we can manipulate that object in the same way that we would manipulate an object created with `plot_ly()`. A simple and obviously useful application of this is to specify interaction modes, like plotly.js' layout.hovermode.

```
p <- ggplot(fortify(gold), aes(x, y)) + geom_line()
gg <- ggplotly(p)
layout(gg, hovermode = "x")
```

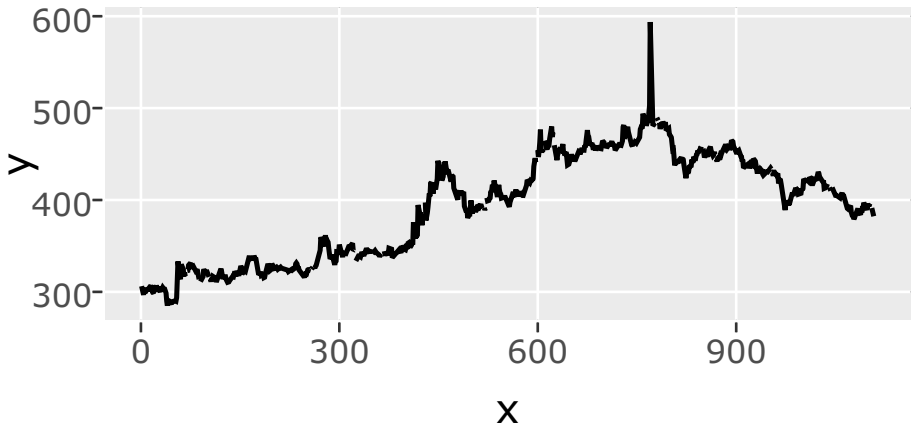


Figure 1.7:

We can also easily add a range slider to the x-axis, which allows you to zoom on the x-axis, without losing the global context. This is quite useful for quickly altering the limits of your plot to achieve an optimal aspect ratio for your data (**?**), without losing the global perspective.

```
rangeslider(gg)
```

Since a single plotly object can only have one layout, modifying the layout of `ggplotly()` is fairly easy, but it's trickier to add and modify layers.

1.2.2 Adding layers

Since `ggplotly()` returns a plotly object, and plotly objects have data associated with them, we can effectively associate data from a ggplot object with a plotly object, before or after summary statistics have been applied. Since each ggplot layer owns a data frame, it is useful to have some way to specify the particular layer of data of interest, which is the point of the `layerData` argument in `ggplotly()`. Also, when a particular layer applies a summary statistic (e.g., `geom_bin()`), or applies a model (e.g., `geom_smooth()`) to the data, it might be useful to access the output of that transformation, which is the point of the `originalData` argument in `ggplotly()`.

```
p <- ggplot(mtcars, aes(x = wt, y = mpg)) +
   geom_point() + geom_smooth()
```

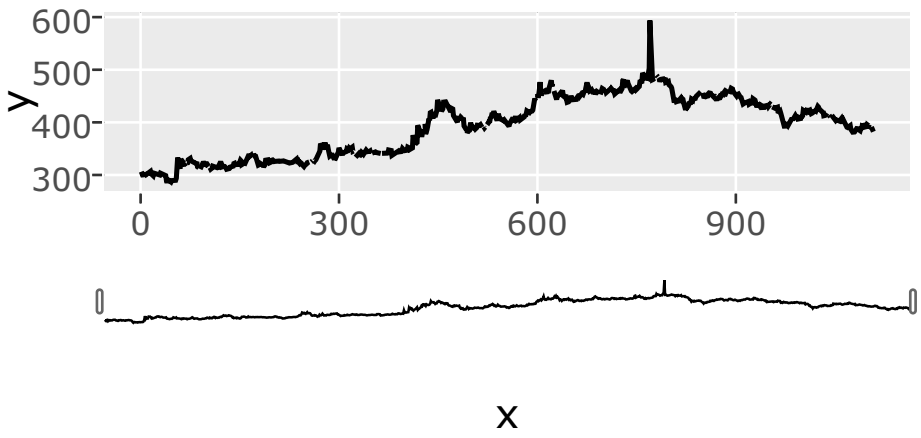


Figure 1.8:

```
p %>%
  ggplotly(layerData = 2, originalData = FALSE) %>%
  plotly_data()
#> # A tibble: 80 × 13
#>       x     y  ymin  ymax    se PANEL group  colour   fill  size linetype
#> * <dbl> <dbl> <dbl> <dbl> <dbl> <int> <int>   <chr>  <chr> <dbl>    <dbl>
#> 1  1.51  32.1  28.1  36.0  1.92     1    -1 #3366FF grey60     1        1
#> 2  1.56  31.7  28.2  35.2  1.72     1    -1 #3366FF grey60     1        1
#> 3  1.61  31.3  28.1  34.5  1.54     1    -1 #3366FF grey60     1        1
#> 4  1.66  30.9  28.0  33.7  1.39     1    -1 #3366FF grey60     1        1
#> 5  1.71  30.5  27.9  33.0  1.26     1    -1 #3366FF grey60     1        1
#> 6  1.76  30.0  27.7  32.4  1.16     1    -1 #3366FF grey60     1        1
#> # ... with 74 more rows, and 2 more variables: weight <dbl>, alpha <dbl>
```

This is the dataset ggplot2 uses to actually draw the fitted values (as a line) and standard error bounds (as a ribbon). Figure **??** uses this data to add additional information about the model fit; in particular, it adds a vertical lines and annotations at the x-values that are associated with the highest and lowest amount uncertainty in y.

```
p %>%
  ggplotly(layerData = 2, originalData = F) %>%
  add_fun(function(p) {
    p %>% slice(which.max(se)) %>%
      add_segments(x = ~x, xend = ~x, y = ~ymin, yend = ~ymax) %>%
      add_annotations("Maximum uncertainty", ax = 60)
  }) %>%
  add_fun(function(p) {
    p %>% slice(which.min(se)) %>%
      add_segments(x = ~x, xend = ~x, y = ~ymin, yend = ~ymax) %>%
      add_annotations("Minimum uncertainty")
  })
```

Although it is not used in this example, it worth noting that when adding plotly layers to the output of `ggplotly()`, it will inherit the global mapping by default, which may or may not be desired, but the `inherit` argument in any of the `add_*()` functions may be set to `FALSE` to avoid this behavoir.

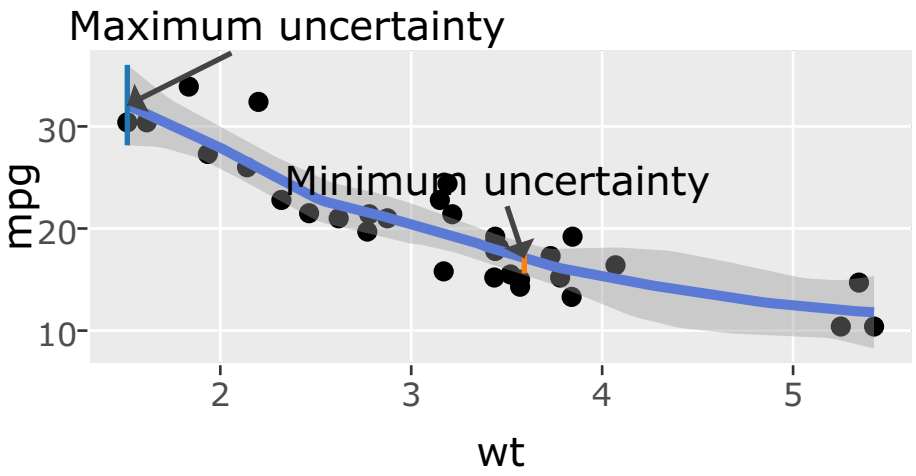


Figure 1.9: Leveraging data associated with a `geom_smooth()` layer to display additional information about the model fit.

1.2.3 Modifying layers

As mentioned previously, `ggplotly()` translates each ggplot2 layer into one or more plotly.js traces. In this translation, it is forced to make a number of assumptions about trace attribute values that may or may not be appropriate for the use case. The `style()` function is useful in this scenario, as it provides a way to modify trace attribute values in a plotly object. Before using it, you may want to inspect the actual traces in a given plotly object using the `plotly_json()` function. This function uses the **listviewer** package to display a convenient interactive view of the JSON object sent to plotly.js (**?**). By clicking on the arrow next to the data element, you can see the traces (data) behind the plot. In this case, we have three traces: one for the `geom_point()` layer and two for the `geom_smooth()` layer.

```
plotly_json(p)
```

Say, for example, we'd like to display information when hovering over points, but not when hovering over the fitted values or error bounds. The ggplot2 API has no semantics for making this distinction, but this is easily done in plotly.js by setting the hoverinfo attribute to `"none"`. Since the fitted values or error bounds are contained in the second and third traces, we can hide the information on just these traces using the `traces` attribute in the `style()` function:

```
style(p, hoverinfo = "none", traces = 2:3)
```

1.3 Choosing an interface

1. ggplot2 requires data frame(s) and can be inefficient (especially for time series).
2. ggplot2 does not have a functional interface (making it awkward to combine with modern functional interfaces such as dplyr), and does not satisfy referential transparency (making it easier to program with – for more details, see)
3. `ggplotly()` tries to replicate *exactly* what you see in the corresponding static ggplot2 graph. To do so, it sends axis tick information to plotly as tickvals/ticktext properties, and consequently, axis ticks do not update on zoom events.
4. ggplot2's interface wasn't designed for interactive graphics. Directly extending the grammar to support more advanced types of interaction (e.g., linked brushing) is a risky endeavor.

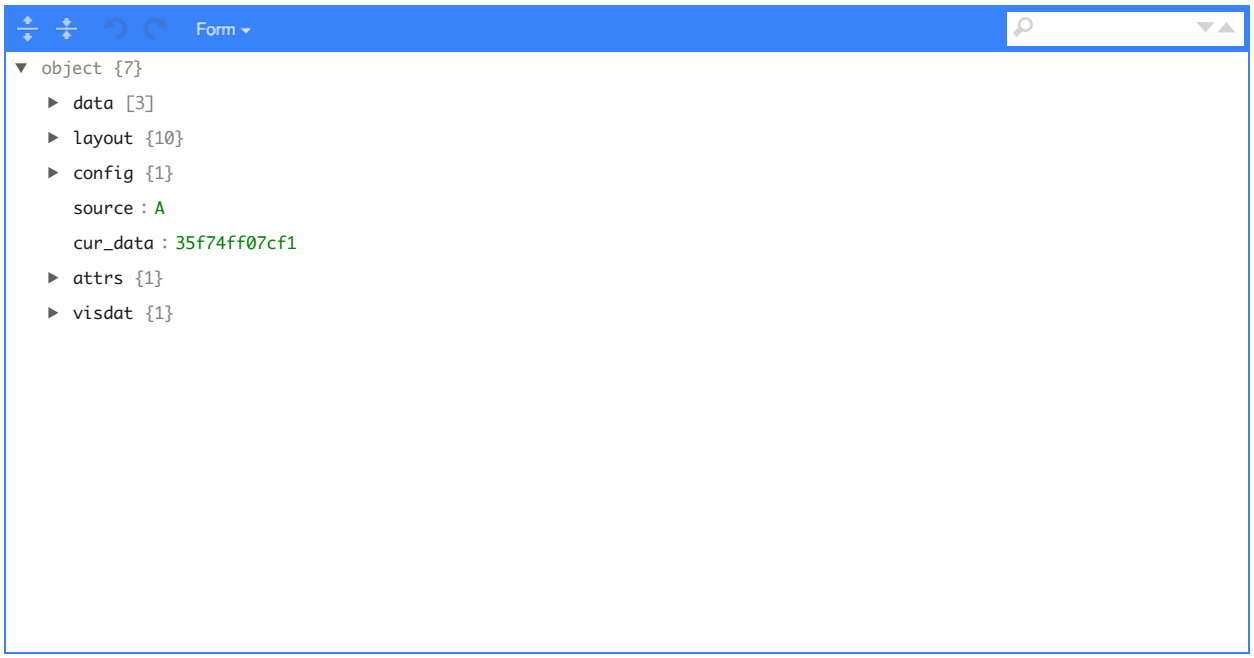


Figure 1.10: Using listviewer to inspect a plotly object.

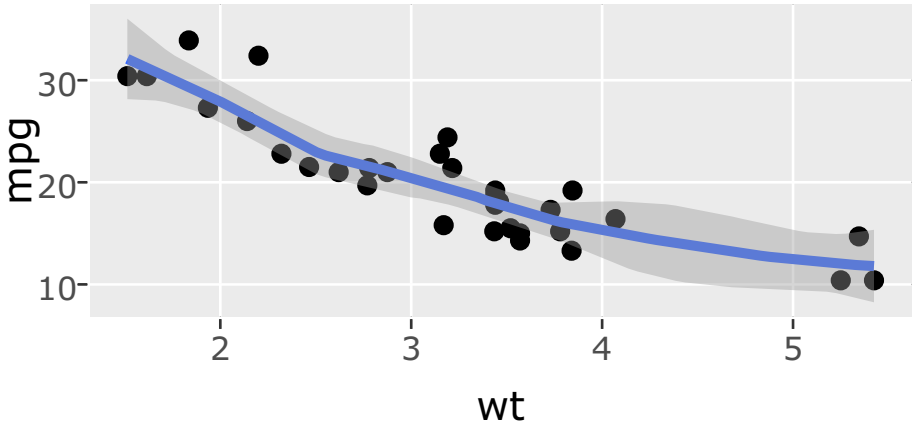


Figure 1.11:

Chapter 2

The plotly cookbook

This chapter demonstrates the capabilities of `plot_ly()` through a series of examples. The `plot_ly()` function does provide a direct interface to plotly.js, so anything in the figure reference can be specified via `plot_ly()`, but this chapter will focus more on the semantics unique to the R package that can't be found on the figure reference. Along the way, we will touch on some best practices in visualization.

2.1 Scatter traces

A plotly visualization is composed of one (or more) trace(s), and every trace has a `type`. The default trace type, "scatter", can be used to draw a large amount of geometries, and actually powers many of the `add_*()` functions such as `add_markers()`, `add_lines()`, `add_paths()`, `add_segments()`, `add_ribbons()`, and `add_polygons()`. Among other things, these functions make assumptions about the mode of the scatter trace, but any valid attribute(s) listed under the scatter section of the figure reference may be used to override defaults.

You may notice some arguments are related to items in the in the figure reference, but are not listed (e.g., `color`/`colors`, `symbol`/`symbols`, `linetype`/`linetypes`, `size`/`sizes`). These arguments (documented on the help page `help(plot_ly)`) are unique to the R package and make it easier to scale data values to visual aesthetics. Generally speaking, the singular form of the argument defines the domain of the scale (data) and the plural form defines the range of the scale (visuals). To make it easier to alter default visual aesthetics (e.g., change all points from blue to black), interprets "AsIs" values (values wrapped with the `I()` function) as values that already live in visual space, and thus do not need to be scaled. The next section on scatterplots explores detailed use of the `color`/`colors`, `symbol`/`symbols`, & `size`/`sizes` arguments. The section on lineplots explores detailed use of the `linetype`/`linetypes`.

2.1.1 Scatterplots

The scatterplot is useful for visualizing the correlation between two quantitative variables. If you supply a numeric vector for x and y in `plot_ly()`, it defaults to a scatterplot, but you can also be explicit about adding a layer of markers/points via the `add_markers()` function. A common problem with scatterplots is overplotting, meaning that there are multiple observations occupying the same (or similar) x/y locations. There are a few ways to combat overplotting including: alpha transparency, hollow symbols, and 2D density estimation. Figure **??** shows how alpha transparency and hollow symbols can provide an improvment over the default.

```
subplot(
  plot_ly(mpg, x = ~cty, y = ~hwy, name = "default"),
  plot_ly(mpg, x = ~cty, y = ~hwy) %>% add_markers(alpha = 0.2, name = "alpha"),
```

```
  plot_ly(mpg, x = ~cty, y = ~hwy) %>% add_markers(symbol = I(1), name = "hollow")
)
```

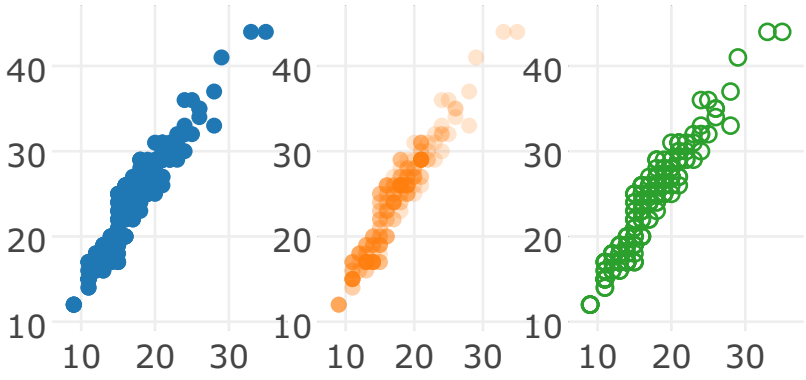


Figure 2.1: Three versions of a basic scatterplot

In Figure **??**, hollow circles are specified via `symbol = I(1)`. By default, the `symbol` argument (as well as the `color`/`size`/`linetype` arguments) assumes value(s) are "data", which need to be mapped to a visual palette (provided by `symbols`). Wrapping values with the `I()` function notifies `plot_ly()` that these values should be taken "AsIs". If you compare the result of `plot(1:25, 1:25, pch = 1:25)` to Figure **??**, you'll see that `plot_ly()` can translate R's plotting characters (pch), but you can also use plotly.js' symbol syntax, if you desire.

```
subplot(
  plot_ly(x = 1:25, y = 1:25, symbol = I(1:25), name = "pch"),
  plot_ly(mpg, x = ~cty, y = ~hwy, symbol = ~cyl, symbols = 1:3, name = "cyl")
)
```

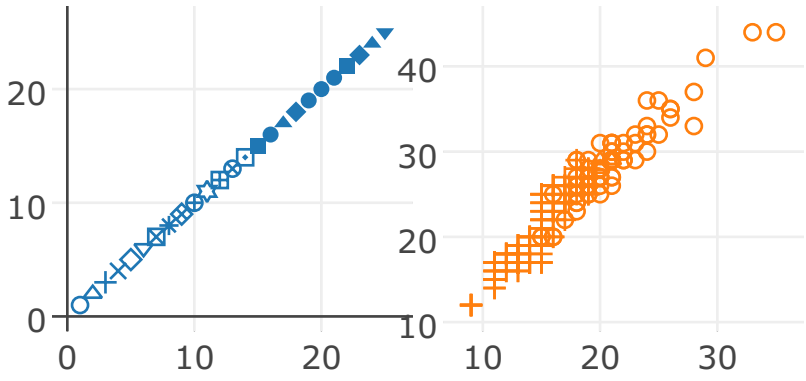


Figure 2.2: Specifying symbol in a scatterplot

When mapping a numeric variable to `symbol`, it creates only one trace, so no legend is generated. If you do want one trace per symbol, make sure the variable you're mapping is a factor, as Figure **??** demonstrates. When plotting multiple traces, the default plotly.js color scale will apply, but you can set the color of every trace generated from this layer with `color = I("black")`, or similar.

```
p <- plot_ly(mpg, x = ~cty, y = ~hwy, alpha = 0.3)
subplot(
  add_markers(p, symbol = ~cyl, name = "A single trace"),
  add_markers(p, symbol = ~factor(cyl), color = I("black"))
)
```

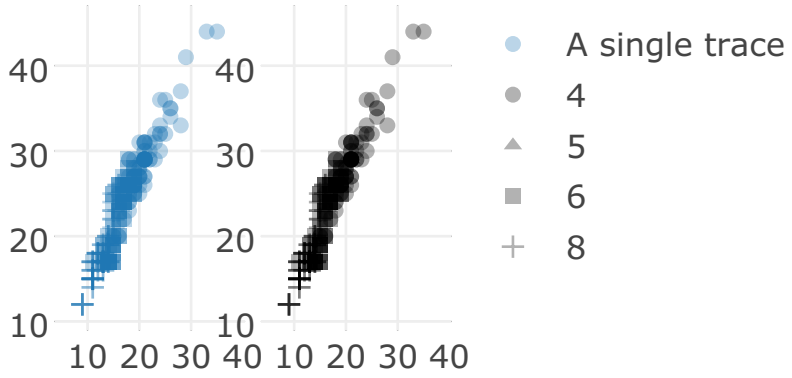


Figure 2.3: Mapping symbol to a factor

The `color` argument adheres to similar rules as `symbol`:

- If numeric, `color` produces one trace, but colorbar is also generated to aide the decoding of colors back to data values. The `colorbar()` function can be used to customize the appearance of this automatically generated guide. The default colorscale is viridis, a perceptually-uniform colorscale (even when converted to black-and-white), and perceivable even to those with common forms of color blindness (**?**).
- If discrete, `color` produces one trace per value, meaning a legend is generated. If an ordered factor, the default colorscale is viridis (**?**); otherwise, it is the "Set2" palette from the **RColorBrewer** package (**?**)

```
p <- plot_ly(mpg, x = ~cty, y = ~hwy, alpha = 0.5)
subplot(
  add_markers(p, color = ~cyl, showlegend = FALSE) %>%
    colorbar(title = "Viridis", len = 1/2, y = 1),
  add_markers(p, color = ~factor(cyl))
) %>% layout(showlegend = TRUE)
```

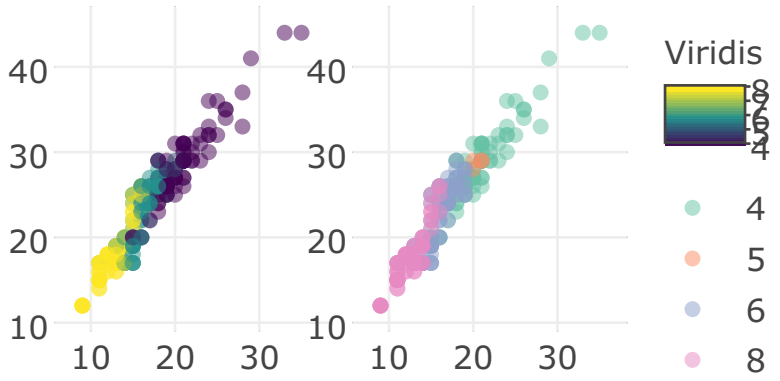


Figure 2.4: Variations on a numeric color mapping.

There are a number of ways to alter the default colorscale via the `colors` argument. This argument excepts: (1) a color brewer palette name (see the row names of `RColorBrewer::brewer.pal.info` for valid names), (2) a vector of colors to interpolate, or (3) a color interpolation function like `colorRamp()` or `scales::colour_ramp()`. Although this grants a lot of flexibility, one should be concious of using a sequential colorscale for numeric variables (& ordered factors) as shown in **??**, and a qualitative colorscale for discrete variables as shown in **??**. (TODO: touch on lurking variables?)

```
subplot(
  add_markers(p, color = ~cyl, colors = c("#132B43", "#56B1F7")) %>%
    colorbar(title = "ggplot2 default", len = 1/3, y = 1),
  add_markers(p, color = ~cyl, colors = viridisLite::inferno(10)) %>%
    colorbar(title = "Inferno", len = 1/3, y = 2/3),
  add_markers(p, color = ~cyl, colors = colorRamp(c("red", "white", "blue"))) %>%
    colorbar(title = "colorRamp", len = 1/3, y = 1/3)
)
```

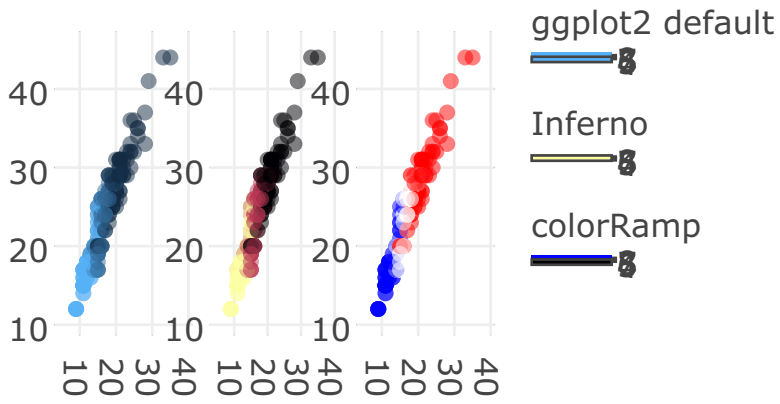


Figure 2.5: Three variations on a numeric color mapping

```
subplot(
  add_markers(p, color = ~factor(cyl), colors = "Pastel1"),
  add_markers(p, color = ~factor(cyl), colors = colorRamp(c("red", "blue"))),
  add_markers(p, color = ~factor(cyl),
              colors = c(`4` = "red", `5` = "black", `6` = "blue", `8` = "green"))
) %>% layout(showlegend = FALSE)
```

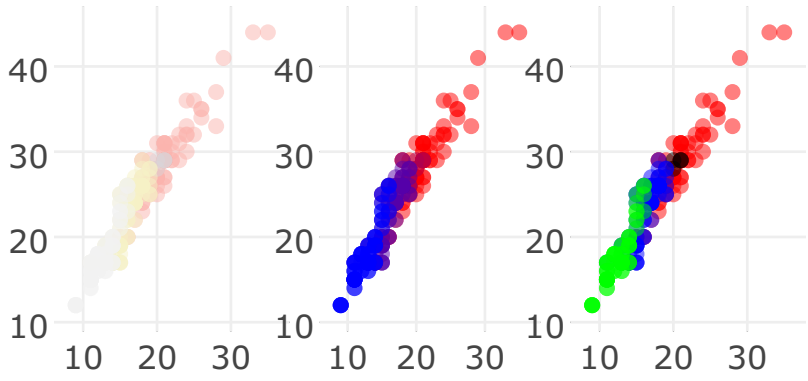


Figure 2.6: Three variations on a discrete color mapping

For scatterplots, the `size` argument controls the area of markers (unless otherwise specified via sizemode), and *must* be a numeric variable. The `sizes` argument controls the minimum and maximum size of circles, in pixels:

```
subplot(
  add_markers(p, size = ~cyl, name = "default"),
  add_markers(p, size = ~cyl, sizes = c(1, 500), name = "custom")
```

```
)
```

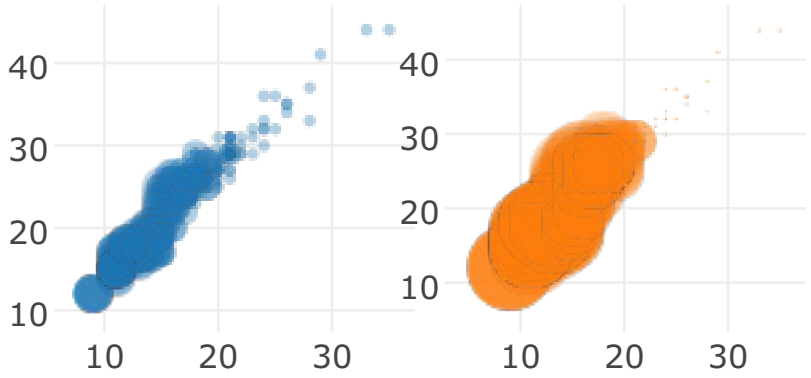
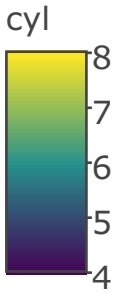


Figure 2.7:

2.1.1.1 3D scatterplots

To make a 3D scatterplot, just add a `z` attribute:

```
plot_ly(mpg, x = ~cty, y = ~hwy, z = ~cyl) %>%
  add_markers(color = ~cyl)
```



Webgl is not supported by your browser - visit http://get.webgl.org for more info

Figure 2.8: A 3D scatterplot

2.1.1.2 Scatterplot matrices

Scatterplot matrices *can* be made via `plot_ly()` and `subplot()`, but `ggplotly()` has a special method for translating ggmatrix objects from the **GGally** package to plotly objects (**?**). These objects are essentially a matrix of ggplot objects and are the underlying data structure which powers higher level functions in **GGally**,

such as `ggpairs()`, a function for making a generalized pairs plot (a generalization of the scatterplot matrix to incorporate categorical data) (**?**). Figure **??** shows an interactive version of the generalized pairs plot made via `ggpairs()` and `ggplotly()`. In Linking views without shiny, we explore how this framework can be extended to enable linked brushing in the generalized pairs plot.

```
pm <- GGally::ggpairs(iris)
ggplotly(pm)
```

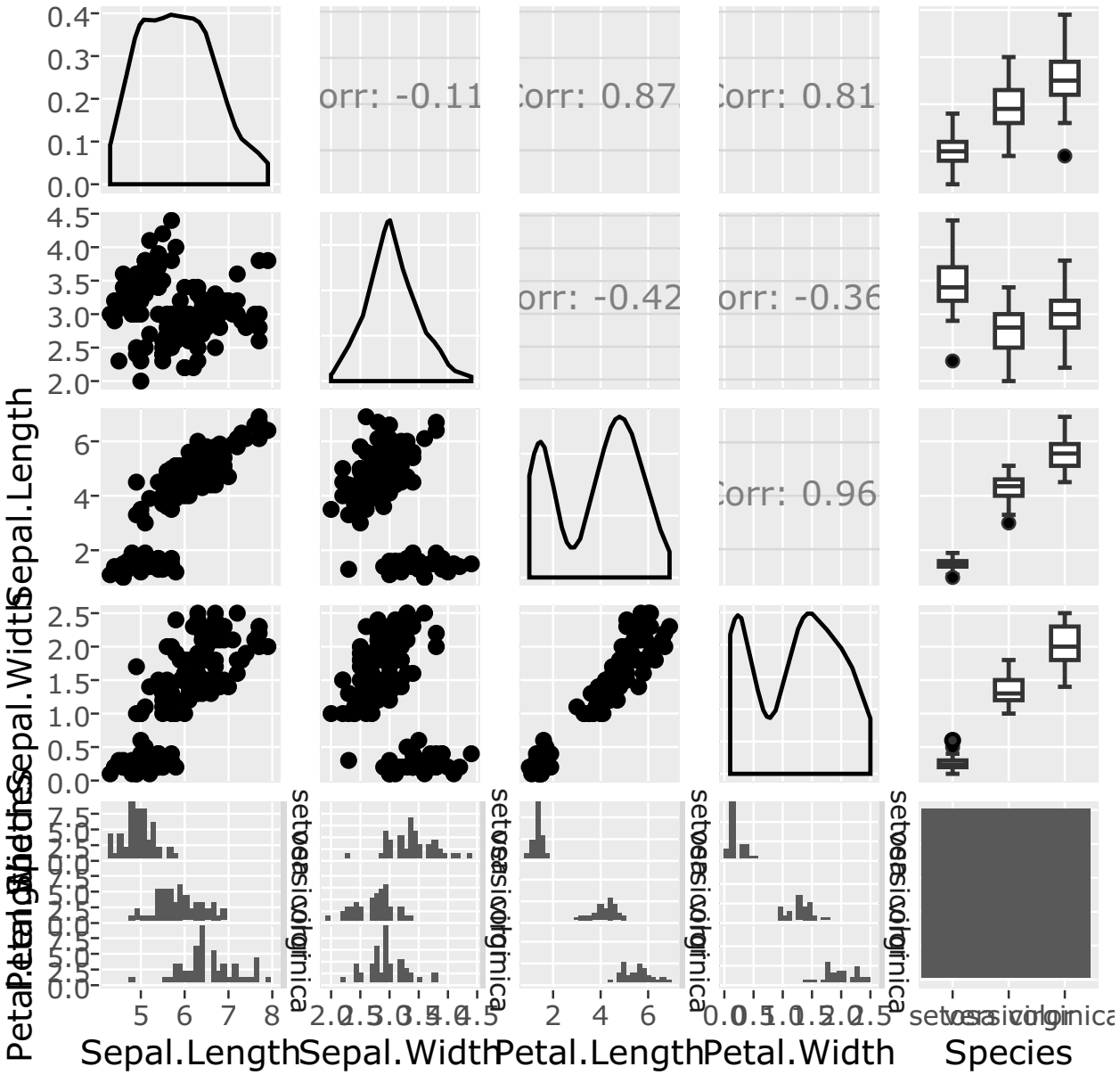


Figure 2.9: An interactive version of the generalized pairs plot made via the `ggpairs()` function from the **GGally** package

2.1.2 Dotplots & error bars

A dotplot is similar to a scatterplot, except instead of two numeric axes, one is categorical. The usual goal of a dotplot is to compare value(s) on a numerical scale over numerous categories. In this context, dotplots

are preferrable to pie charts since comparing position along a common scale is much easier than comparing angle or area (**?**); (**?**). Furthermore, dotplots can be preferrable to bar charts, especially when comparing values within a narrow range far away from 0 (**?**). Also, when presenting point estimates, and uncertainty associated with those estimates, bar charts tend to exaggerate the difference in point estimates, and lose focus on uncertainty (**?**).

A popular application for dotplots (with error bars) is the so-called "coefficient plot" for visualizing the point estimates of coefficients and their standard error. The `coefplot()` function in the **coefplot** package (**?**) and the `ggcoef()` function in the **GGally** both produce coefficient plots for many types of model objects in R using **ggplot2**, which we can translate to plotly via `ggplotly()`. Since these packages use points and segments to draw the coefficient plots, the hover information is not the best, and it'd be better to use error objects. Figure **??** uses the `tidy()` function from the **broom** package (**?**) to obtain a data frame with one row per model coefficient, and produce a coefficient plot with error bars along the x-axis.

```
m <- lm(Sepal.Length ~ Sepal.Width * Petal.Length * Petal.Width, data = iris)
# arrange by estimate, then make term a factor to order categories in the plot
d <- broom::tidy(m) %>%
  arrange(desc(estimate)) %>%
  mutate(term = factor(term, levels = term))
plot_ly(d, x = ~estimate, y = ~term) %>%
  add_markers(error_x = ~list(value = std.error)) %>%
  layout(margin = list(l = 200))
```

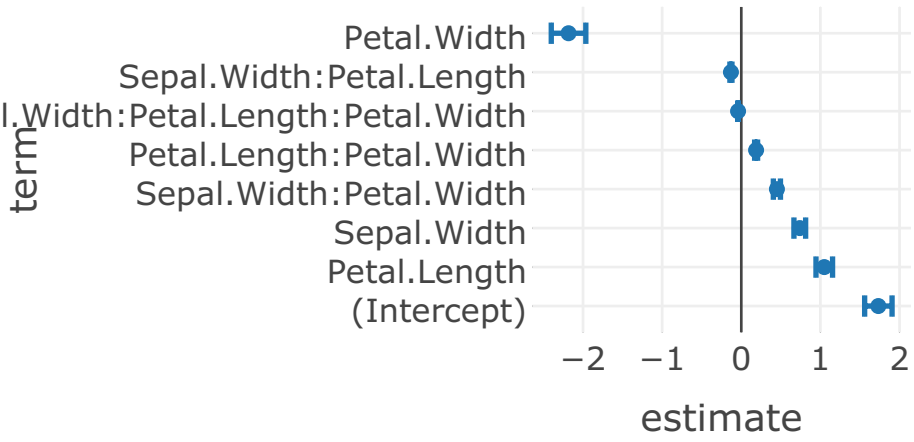


Figure 2.10: A coefficient plot

2.1.3 Line plots

This section surveys useful applications of `add_lines()` and `add_paths()`. The only difference between these functions is that `add_lines()` connects x/y pairs from left to right, instead of the order in which the data appears. Both functions understand the `color`, `linetype`, and `alpha` attributes[1], as well as groupings defined by `group_by()`.

Figure **??** uses `group_by()` to plot one line per city in the `txhousing` dataset using a *single* trace. Since there can only be one tooltip per trace, hovering over that plot does not reveal useful information. Although plotting many traces can be computationally expensive, it is necessary in order to display better information on hover. Since the `color` argument produces one trace per value (if the variable (`city`) is discrete), hovering on Figure **??** reveals the top ~10 cities at a given x value. Since 46 colors is too many to perceive in a single plot, Figure **??** also restricts the set of possible `colors` to black.

[1] plotly.js currently does not support data arrays for `scatter.line.width` or `scatter.line.color`, meaning a single line trace can only have one width/color in 2D line plot, and consequently numeric `color`/`size` mappings won't work

```
plot_ly(txhousing, x = ~date, y = ~median) %>%
  add_lines(color = ~city, colors = "black", alpha = 0.2)
```

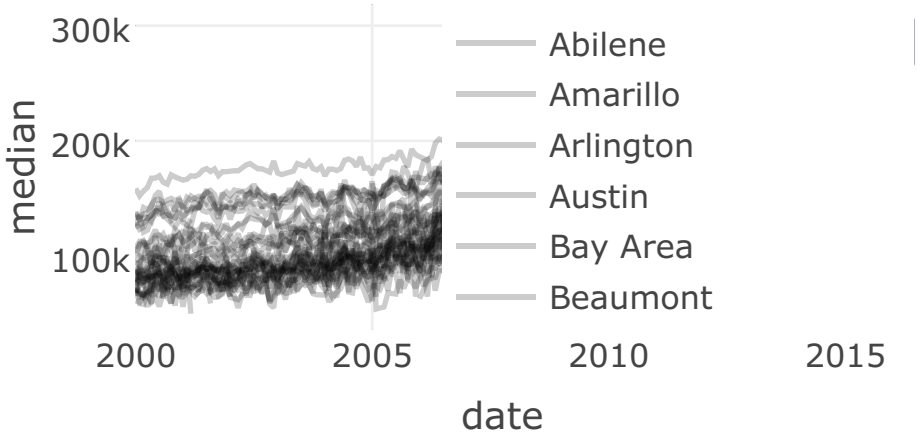


Figure 2.11: Median house sales with one trace per city.

Generally speaking, it's hard to perceive more than 8 different colors/linetypes/symbols in a given plot, so sometimes we have to filter data to use these effectively. Here we use the **dplyr** package to find the top 5 cities in terms of average monthly sales (`top5`), then effectively filter the original data to contain just these cities via `semi_join()`. Once we have the data is filtered, mapping city to `color` or `linetype` is trivial. The color palette can be altered via the `colors` argument, and follows the same rules as scatterplots. The linetype palette can be altered via the `linetypes` argument, and accepts R's `lty` values or plotly.js dash values.

```
library(dplyr)
top5 <- txhousing %>%
  group_by(city) %>%
  summarise(m = mean(sales, na.rm = TRUE)) %>%
  arrange(desc(m)) %>%
  top_n(5)

p <- semi_join(txhousing, top5) %>%
  plot_ly(x = ~date, y = ~median)

subplot(
  add_lines(p, color = ~city),
  add_lines(p, linetype = ~city),
  shareX = TRUE, nrows = 2
)
```

2.1.3.1 Density plots

In Bars & histograms, we leveraged a number of algorithms in R for computing the "optimal" number of bins for a histogram, via `hist()`, and routing those results to `add_bars()`.

```
kerns <- c("gaussian", "epanechnikov", "rectangular",
           "triangular", "biweight", "cosine", "optcosine")
p <- plot_ly()
for (k in kerns) {
  d <- density(txhousing$median, kernel = k, na.rm = TRUE)
```

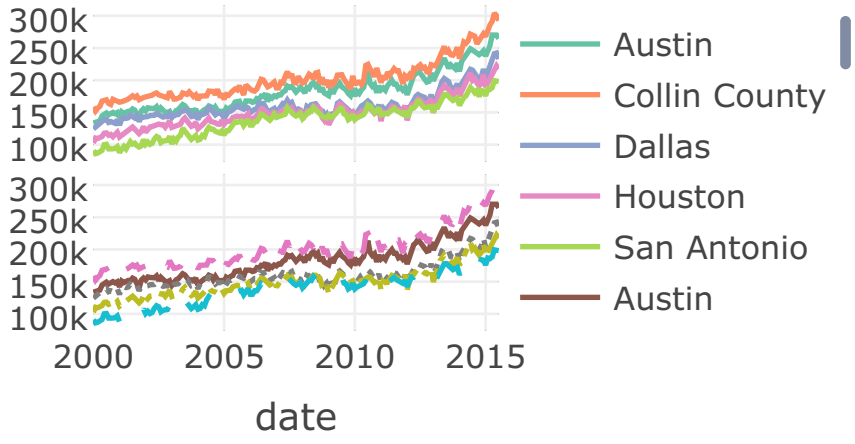


Figure 2.12:

```
  p <- add_lines(p, x = d$x, y = d$y, name = k)
}
layout(p, xaxis = list(title = "Median monthly price"))
```

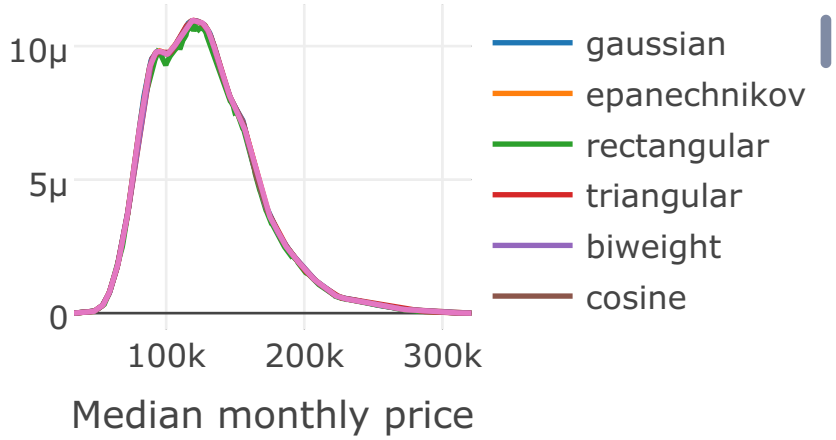


Figure 2.13:

2.1.3.2 Parallel Coordinates

One very useful, but often overlooked, visualization technique is the parallel coordinates plot. Parallel coordinates provide a way to compare values along a common (or non-aligned) positional scale(s) – the most basic of all perceptual tasks – in more than 3 dimensions (**?**). Usually each line represents every measurement for a given row (or observation) in a data set. When measurements are on very different scales, some care must be taken, and variables must transformed to be put on a common scale. As Figure **??** shows, even when variables are measured on a similar scale, it can still be a informative to transform variables in different ways.

```
iris$obs <- seq_len(nrow(iris))
iris_pcp <- function(transform = identity) {
  iris[] <- purrr::map_if(iris, is.numeric, transform)
  tidyr::gather(iris, variable, value, -Species, -obs) %>%
    group_by(obs) %>%
```

```
    plot_ly(x = ~variable, y = ~value, color = ~Species) %>%
    add_lines(alpha = 0.3)
}
subplot(
  iris_pcp(),
  iris_pcp(scale),
  iris_pcp(scales::rescale)
) %>% hide_legend()
```

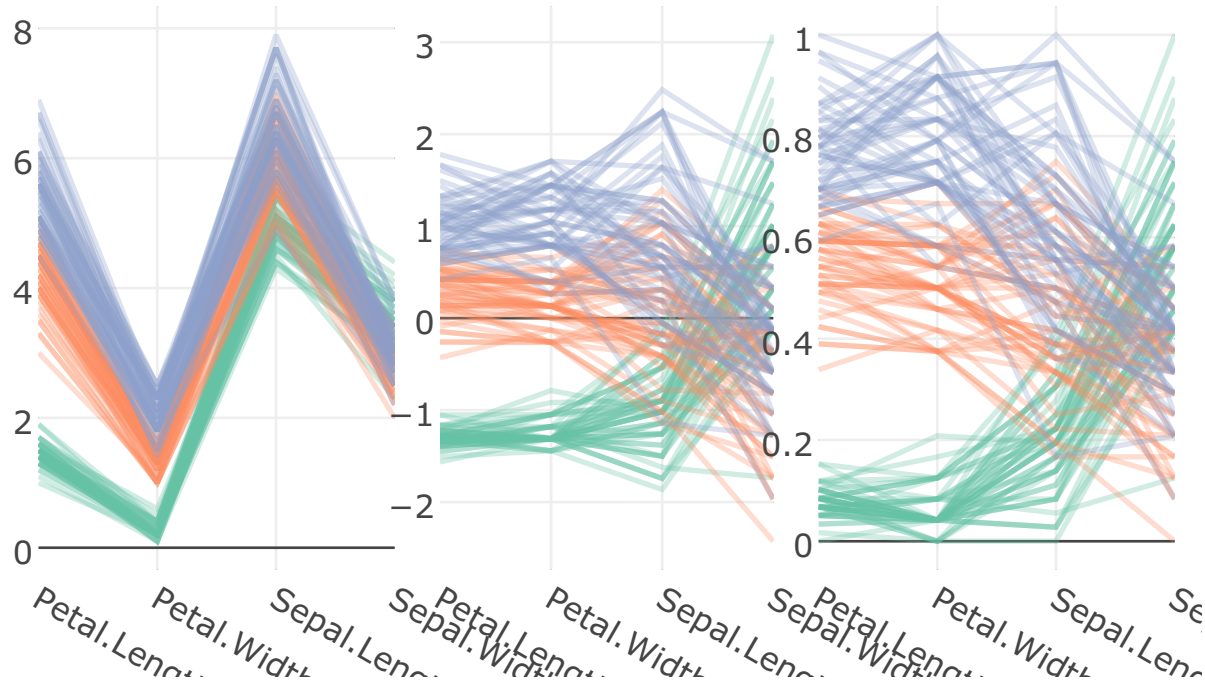


Figure 2.14: Parallel coordinates plots of the Iris dataset. On the left is the raw measurements. In the middle, each variable is scaled to have mean of 0 and standard deviation of 1. On the right, each variable is scaled to have a minimum of 0 and a maximum of 1.

It is also worth noting that the **GGally** offers a `ggparcoord()` function which creates parallel coordinate plots via **ggplot2**, which we can convert to plotly via `ggplotly()`. In linked highlighting, parallel coordinates are linked to lower dimensional (but sometimes higher resolution) graphics of related data to guide multi-variate data exploration.

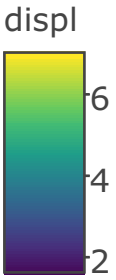
2.1.3.3 3D line plots

To make a 3D line plot, just add a `z` attribute (in addition to `x` and `y`):

```
plot_ly(mpg, x = ~cty, y = ~hwy, z = ~cyl) %>%
  add_lines(color = ~displ)
```

2.1.4 Segments

The `add_segments()` function essentially provides a way to connect two points ((`x`, `y`) to (`xend`, `yend`)) with a line. Segments form the building blocks for many useful chart types, including candlestick charts, a popular way to visualize stock prices. Figure **??** uses the **quantmod** package (**?**) to obtain stock price data



Webgl is not supported by your browser - visit http://get.webgl.org for more info

Figure 2.15: A 3D scatterplot

for Microsoft and plots two segments for each day: one to encode the opening/closing values, and one to encode the daily high/low.

```
library(quantmod)
msft <- getSymbols("MSFT", auto.assign = F)
dat <- as.data.frame(msft)
dat$date <- index(msft)
dat <- subset(dat, date >= "2016-01-01")

names(dat) <- sub("^MSFT\\.", "", names(dat))

plot_ly(dat, x = ~date, xend = ~date, color = ~Close > Open,
        colors = c("red", "forestgreen"), hoverinfo = "none") %>%
  add_segments(y = ~Low, yend = ~High, size = I(1)) %>%
  add_segments(y = ~Open, yend = ~Close, size = I(3)) %>%
  layout(showlegend = FALSE, yaxis = list(title = "Price")) %>%
  rangeslider()
```

2.1.5 Ribbons

Ribbons are useful for showing uncertainy bounds as a function of x. The `add_ribbons()` function creates ribbons and requires the arguments: `ymin` and `ymax`. The `augment()` function from the **broom** package appends observational-level model components (e.g., fitted values stored as a new column `.fitted`) which is useful for extracting those components in a form that is convenient for visualization.

```
m <- lm(mpg ~ wt, data = mtcars)
broom::augment(m) %>%
  plot_ly(x = ~wt, showlegend = FALSE) %>%
  add_markers(y = ~mpg, color = I("black")) %>%
  add_ribbons(ymin = ~.fitted - 1.96 * .se.fit,
```

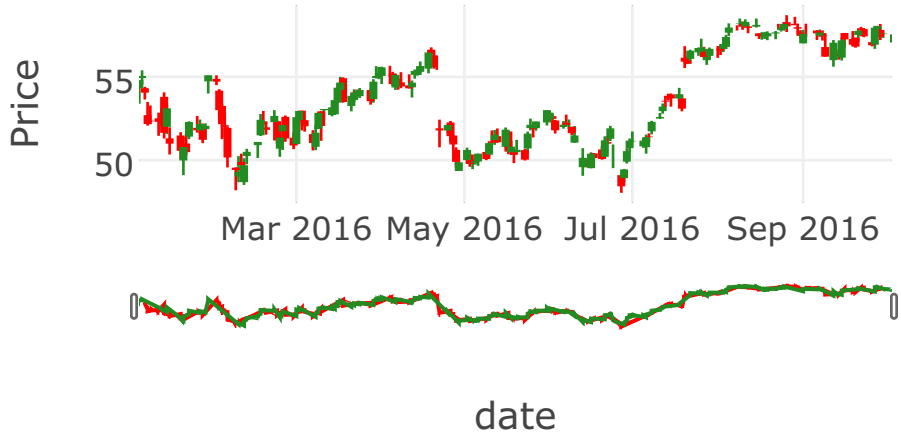


Figure 2.16: A candelstick chart

```
            ymax = ~.fitted + 1.96 * .se.fit, color = I("gray80")) %>%
  add_lines(y = ~.fitted, color = I("steelblue"))
```

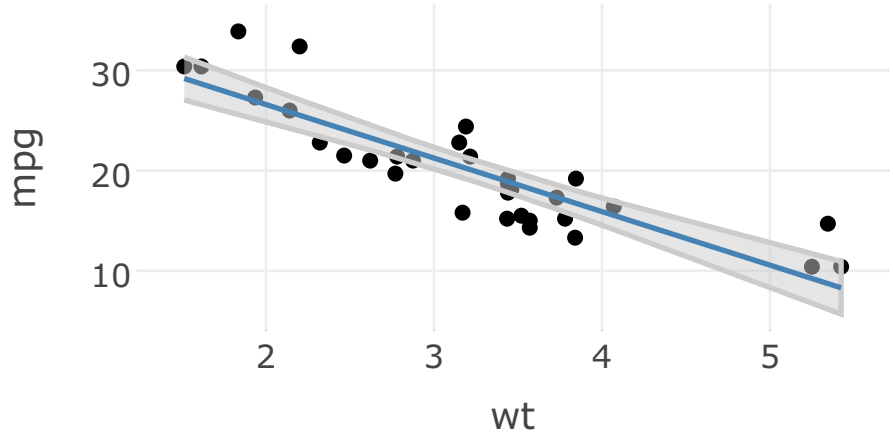


Figure 2.17:

2.1.6 Polygons

The `add_polygons()` function is essentially equivalent to `add_paths()` with the fill attribute set to "toself". Polygons from the basis for other, higher-level, geometries such as `add_ribbons()`, but can be useful in their own right.

```
map_data("world", "canada") %>%
  group_by(group) %>%
  plot_ly(x = ~long, y = ~lat, alpha = 0.2) %>%
  add_polygons(hoverinfo = "none", color = I("black")) %>%
  add_markers(text = ~paste(name, "<br />", pop), hoverinfo = "text",
              color = I("red"), data = maps::canada.cities) %>%
  layout(showlegend = FALSE)
```

[](bookdown_files/figure-latex/map-canada, "A map of Canada using the default cartesian coordinate system."-1.pdf)

2.2 Maps

2.2.1 Using scatter traces

As shown in polygons, it is possible to create maps using plotly's default (cartesian) coordinate system, but plotly.js also has support for plotting scatter traces on top of either a custom geo layout or a mapbox layout. Figure **??** compares the three different layout options in a single subplot.

```
dat <- map_data("world", "canada") %>% group_by(group)

map1 <- plot_ly(dat, x = ~long, y = ~lat) %>%
  add_paths(size = I(1)) %>%
  add_segments(x = -100, xend = -50, y = 50, 75)

map2 <- plot_mapbox(dat, x = ~long, y = ~lat) %>%
  add_paths(size = I(2)) %>%
  add_segments(x = -100, xend = -50, y = 50, 75) %>%
  layout(mapbox = list(zoom = 0,
      center = list(lat = ~median(lat), lon = ~median(long))
   ))

# geo() is the only object type which supports different map projections
map3 <- plot_geo(dat, x = ~long, y = ~lat) %>%
  add_markers(size = I(1)) %>%
  add_segments(x = -100, xend = -50, y = 50, 75) %>%
  layout(geo = list(projection = list(type = "mercator")))

subplot(map1, map2) %>%
  subplot(map3, nrows = 2) %>%
  hide_legend()
```

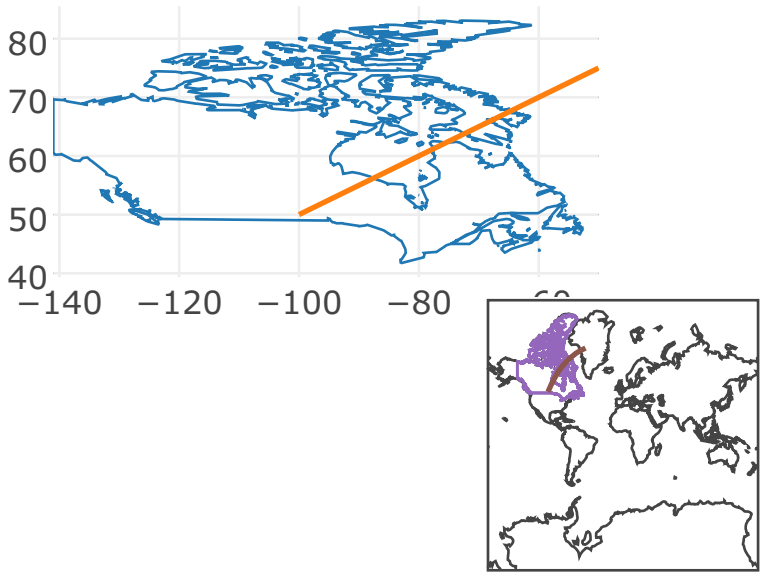


Figure 2.18: A few maps

Any of the `add_*()` functions found under scatter traces should work as expected on plotly-geo (initialized via `plot_geo()`) or plotly-mapbox (initialized via `plot_mapbox()`) objects. You can think of `plot_geo()`

and `plot_mapbox()` as special cases (or more opiniated versions) of `plot_ly()`. For one, they won't allow you to mix scatter and non-scatter traces in a single plot object, which you probably don't want to do anyway. In order to enable Figure **??**, plotly.js *can't* make this restriction, but since we have `subplot()` in R, we *can* make this restriction without sacrificing flexibility.

2.2.2 Choropleths

In addition to scatter traces, plotly-geo objects can also create a choropleth trace/layer. Figure **??** shows the population density of the U.S. via a choropleth, and also layers on markers for the state center locations, using the U.S. state data from the **datasets** package (**?**). By simply providing a `z` attribute, plotly-geo objects will try to create a choropleth, but you'll also need to provide `locations` and a `locationmode`.

```
density <- state.x77[, "Population"] / state.x77[, "Area"]

g <- list(
  scope = 'usa',
  projection = list(type = 'albers usa'),
  lakecolor = toRGB('white')
)

plot_geo() %>%
  add_trace(
    z = ~density, text = state.name,
    locations = state.abb, locationmode = 'USA-states'
  ) %>%
  add_markers(
    x = state.center[["x"]], y = state.center[["y"]],
    size = I(2), symbol = I(8), color = I("white"), hoverinfo = "none"
  ) %>%
  layout(geo = g)
```

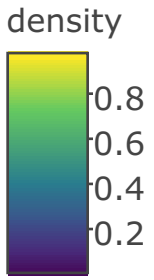


Figure 2.19: A map of U.S. population density using the `state.x77` data from the **datasets** package.

2.3 Bars & histograms

The `add_bars()` and `add_histogram()` functions wrap the bar and histogram plotly.js trace types. The main difference between them is that bar traces require bar heights (both `x` and `y`), whereas histogram traces require just a single variable, and plotly.js handles binning in the browser.[2] And perhaps confusingly, both

[2]This has some interesting applications for linked highlighting as it allows for summary statistics to be computed on-the-fly based on a selection

of these functions can be used to visualize the distribution of either a numeric or a discrete variable. So, essentially, the only difference between them is where the binning occurs.

Figure ?? compares the default binning algorithm in plotly.js to a few different algorithms available in R via the `hist()` function. Although plotly.js has the ability to customize histogram bins via xbins/ybins, R has diverse facilities for estimating the optimal number of bins in a histogram that we can easily leverage.[3] The `hist()` function alone allows us to reference 3 famous algorithms by name (?); (?); (?), but there are also packages (e.g. the **histogram** package) which extend this interface to incorporate more methodology (?). The `price_hist()` function below wraps the `hist()` function to obtain the binning results, and map those bins to a plotly version of the histogram using `add_bars()`.

```
p1 <- plot_ly(diamonds, x = ~price) %>% add_histogram(name = "plotly.js")

price_hist <- function(method = "FD") {
  h <- hist(diamonds$price, breaks = method, plot = FALSE)
  plot_ly(x = h$mids, y = h$counts) %>% add_bars(name = method)
}

subplot(
  p1, price_hist(), price_hist("Sturges"),  price_hist("Scott"),
  nrows = 4, shareX = TRUE
)
```

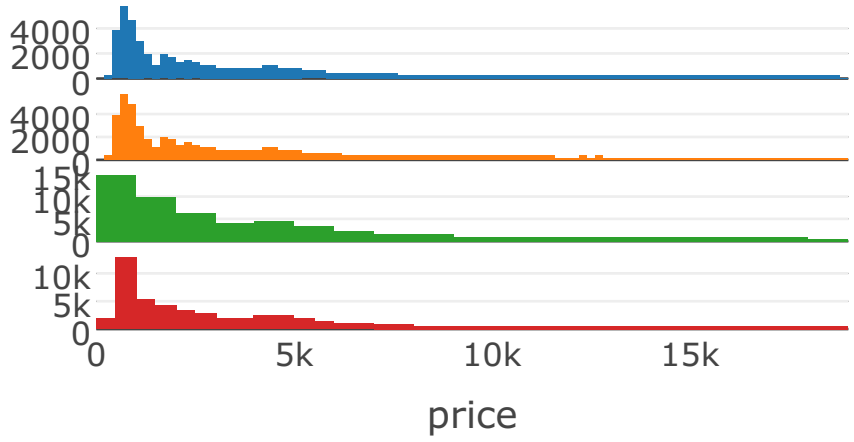


Figure 2.20: plotly.js's default binning algorithm versus R's `hist()` default

Figure ?? demonstrates two ways of creating a basic bar chart. Although the visual results are the same, its worth noting the difference in implementation. The `add_histogram()` function sends all of the observed values to the browser and lets plotly.js perform the binning. It takes more human effort to perform the binning in R, but doing so has the benefit of sending less data, and requiring less computation work of the web browser. In this case, we have only about 50,000 records, so there is much of a difference in page load times or page size. However, with 1 Million records, page load time more than doubles and page size nearly doubles.[4]

```
p1 <- plot_ly(diamonds, x = ~cut) %>% add_histogram()

p2 <- diamonds %>%
  dplyr::count(cut) %>%
```

[3] Optimal in this context is the number of bins which minimizes the distance between the empirical histogram and the underlying density.

[4] These tests were run on Google Chrome and loaded a page with a single bar chart. Here are the results for `add_histogram()` and here are the results for `add_bars()`

```
  plot_ly(x = ~cut, y = ~n) %>%
  add_bars()

subplot(p1, p2) %>% hide_legend()
```

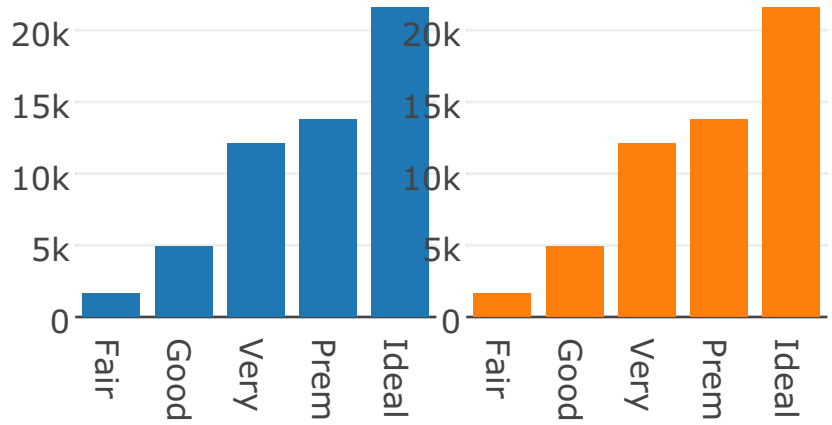


Figure 2.21: Number of diamonds by cut.

2.3.1 Multiple numeric distributions

It is often useful to see how the numeric distribution changes with respect to a discrete variable. When using bars to visualize multiple numeric distributions, I recommend plotting each distribution on its own axis, rather than trying to overlay them on a single axis.[5]. This is where the `subplot()` infrastructure, and its support for trellis displays, comes in handy. Figure **??** shows a trellis display of diamond price by diamond color. Note how the `one_plot()` function defines what to display on each panel, then a split-apply-recombine strategy is employed to generate the trellis display.

```
one_plot <- function(d) {
  plot_ly(d, x = ~price) %>%
    add_annotations(
      ~paste("Clarity:", unique(clarity)), x = 0.5, y = 1,
      xref = "paper", yref = "paper", showarrow = FALSE
    )
}

diamonds %>%
  split(.$clarity) %>%
  lapply(one_plot) %>%
  subplot(nrows = 2, shareX = TRUE, titleX = FALSE) %>%
  hide_legend()
```

2.3.2 Multiple discrete distributions

Visualizing multiple discrete distributions is difficult. The subtle complexity is due to the fact that both counts and proportions are important for understanding multi-variate discrete distributions. Figure **??** presents diamond counts, divided by both their cut and clarity, using a grouped bar chart.

[5]It's much easier to visualize multiple numeric distributions on a single axis using lines

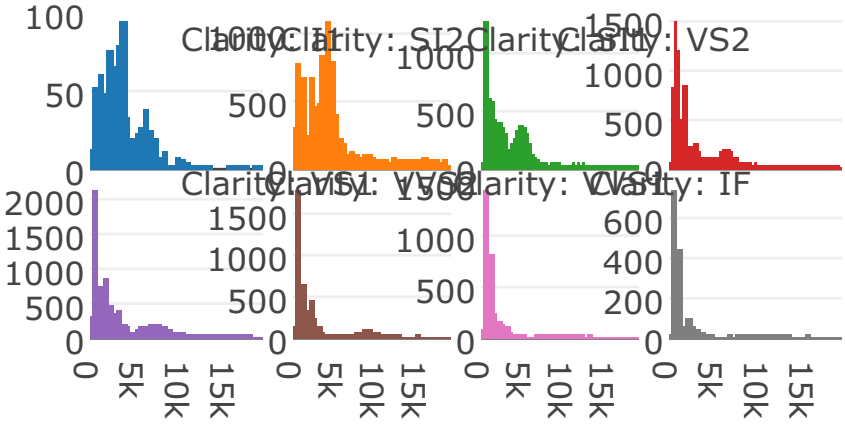


Figure 2.22: A trellis display of diamond price by diamond color.

```
plot_ly(diamonds, x = ~cut, color = ~clarity) %>%
  add_histogram()
```

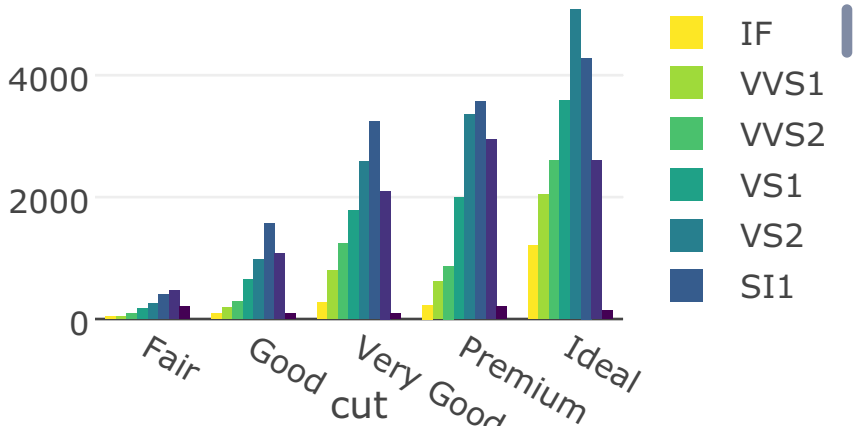


Figure 2.23: A grouped bar chart

Figure ?? is useful for comparing the number of diamonds by clarity, given a type of cut. For instance, within "Ideal" diamonds, a cut of "VS1" is most popular, "VS2" is second most popular, and "I1" the least popular. The distribution of clarity within "Ideal" diamonds seems to be fairly similar to other diamonds, but it's hard to make this comparison using raw counts. Figure ?? makes this comparison easier by showing the relative frequency of diamonds by clarity, given a cut.

```
# number of diamonds by cut and clarity (n)
cc <- count(diamonds, cut, clarity)
# number of diamonds by cut (nn)
cc2 <- left_join(cc, count(cc, cut, wt = n))
cc2 %>%
  mutate(prop = n / nn) %>%
  plot_ly(x = ~cut, y = ~prop, color = ~clarity) %>%
  add_bars() %>%
  layout(barmode = "stack")
```

This type of plot, also known as a spine plot, is a special case of a mosaic plot. In a mosaic plot, you can scale both bar widths and heights according to discrete distributions. For mosaic plots, I recommend using

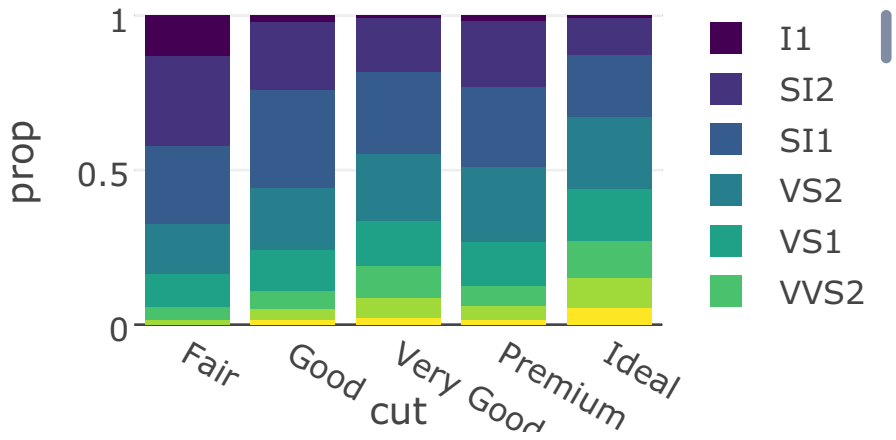


Figure 2.24: A stacked bar chart showing the proportion of clarity within

the **ggmosaic** package (**?**), which implements a custom **ggplot2** geom designed for mosaic plots, which we can convert to plotly via `ggplotly()`. Figure **??** show a mosaic plot of cut by clarity. Notice how the bar widths are scaled proportional to the cut frequency.

```
library(ggmosaic)
p <- ggplot(data = cc) +
  geom_mosaic(aes(weight = n, x = product(cut), fill = clarity))
#> Error in as.vector(y): attempt to apply non-function
ggplotly(p)
```

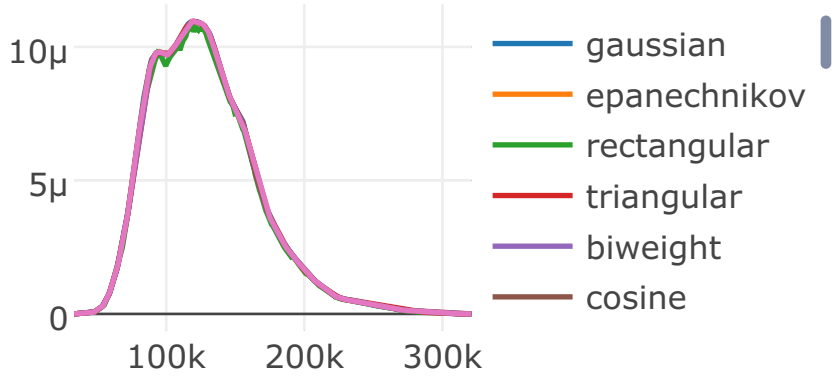


Figure 2.25: Using ggmosaic and ggplotly() to create advanced interactive visualizations of categorical data

2.4 Boxplots

Boxplots encode the five number summary of a numeric variable, and are more efficient than trellis displays of histograms for comparing many numeric distributions. The `add_boxplot()` function requires one numeric variable, and guarantees boxplots are oriented correctly, regardless of whether the numeric variable is placed on the x or y scale. As Figure **??** shows, on the axis orthogonal to the numeric axis, you can provide a discrete variable (for conditioning) or supply a single value (to name the axis category).

```
p <- plot_ly(diamonds, y = ~price, color = I("black"),
             alpha = 0.1, boxpoints = "suspectedoutliers")
p1 <- p %>% add_boxplot(x = "Overall")
```

```
p2 <- p %>% add_boxplot(x = ~cut)
subplot(
  p1, p2, shareY = TRUE,
  widths = c(0.2, 0.8), margin = 0
) %>% hide_legend()
```

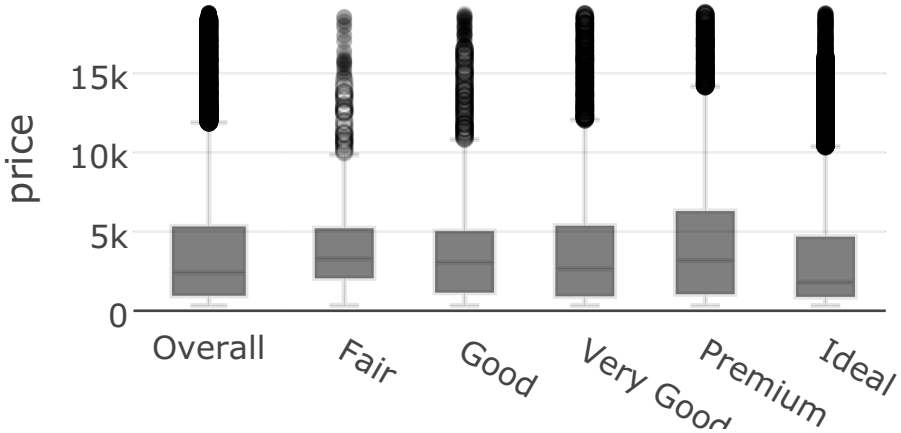


Figure 2.26: Overall diamond price and price by cut.

If you want to partition by more than one discrete variable, I recommend mapping the interaction of those variables to the discrete axis, and coloring by the nested variable, as Figure **??** does with diamond clarity and cut.

```
plot_ly(diamonds, x = ~price, y = ~interaction(clarity, cut)) %>%
  add_boxplot(color = ~clarity) %>%
  layout(yaxis = list(title = ""), margin = list(l = 100))
```

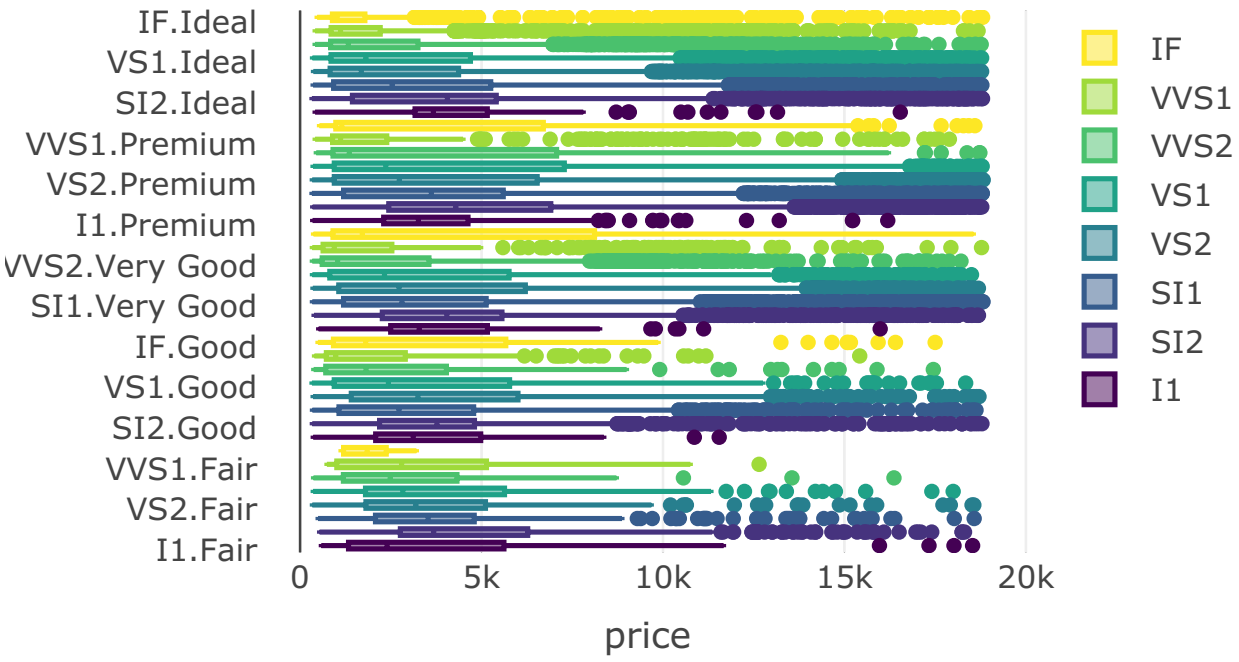


Figure 2.27: Diamond prices by cut and clarity.

It is also helpful to sort the boxplots according to something meaningful, such as the median price. Figure **??** presents the same information as Figure **??**, but sorts the boxplots by their median, and makes it immediately clear that diamonds with a cut of "SI2" have the highest diamond price, on average.

```
d <- diamonds %>%
  mutate(cc = interaction(clarity, cut))

# interaction levels sorted by median price
lvls <- d %>%
  group_by(cc) %>%
  summarise(m = median(price)) %>%
  arrange(m) %>%
  .[["cc"]]

plot_ly(d, x = ~price, y = ~factor(cc, lvls)) %>%
  add_boxplot(color = ~clarity) %>%
  layout(yaxis = list(title = ""), margin = list(l = 100))
```

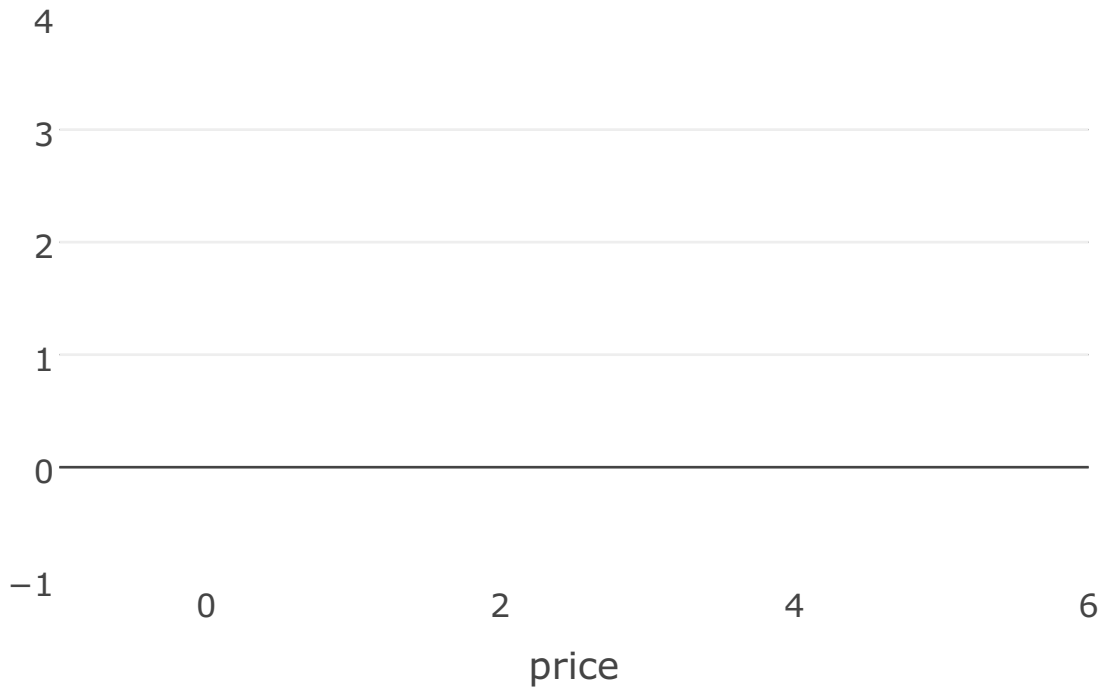


Figure 2.28: Diamond prices by cut and clarity, sorted by price median.

Similar to `add_histogram()`, `add_boxplot()` sends the raw data to the browser, and lets plotly.js compute summary statistics. Unfortunately, plotly.js does not yet allow precomputed statistics for boxplots.[6]

2.5 2D distributions

2.5.1 Rectangular binning in plotly.js

The **plotly** package provides two functions for displaying rectangular bins: `add_heatmap()` and `add_histogram2d()`. For numeric data, the `add_heatmap()` function is a 2D analog of `add_bars()` (bins

[6] Follow the issue here https://github.com/plotly/plotly.js/issues/242

must be pre-computed), and the `add_histogram2d()` function is a 2D analog of `add_histogram()` (bins can be computed in the browser). Thus, I recommend `add_histogram2d()` for exploratory purposes, since you don't have to think about how to perform binning. It also provides a useful `zsmooth` attribute for effectively increasing the number of bins (currently, "best" performs a bi-linear interpolation, a type of nearest neighbors algorithm), and nbinsx/nbinsy attributes to set the number of bins in the x and/or y directions. Figure **??** compares three different uses of `add_histogram()`: (1) plotly.js' default binning algorithm, (2) the default plus smoothing, (3) setting the number of bins in the x and y directions.

```
p <- plot_ly(diamonds, x = ~log(carat), y = ~log(price))
subplot(
  add_histogram2d(p) %>%
    colorbar(title = "default", len = 1/3, y = 1) %>%
    layout(xaxis = list(title = "default")),
  add_histogram2d(p, zsmooth = "best") %>%
    colorbar(title = "zsmooth", len = 1/3, y = 2/3 - 0.05) %>%
    layout(xaxis = list(title = "zsmooth")),
  add_histogram2d(p, nbinsx = 60, nbinsy = 60) %>%
    colorbar(title = "nbins", len = 1/3, y = 1/3 - 0.1) %>%
    layout(xaxis = list(title = "nbins")),
  shareY = TRUE, titleX = TRUE
)
```

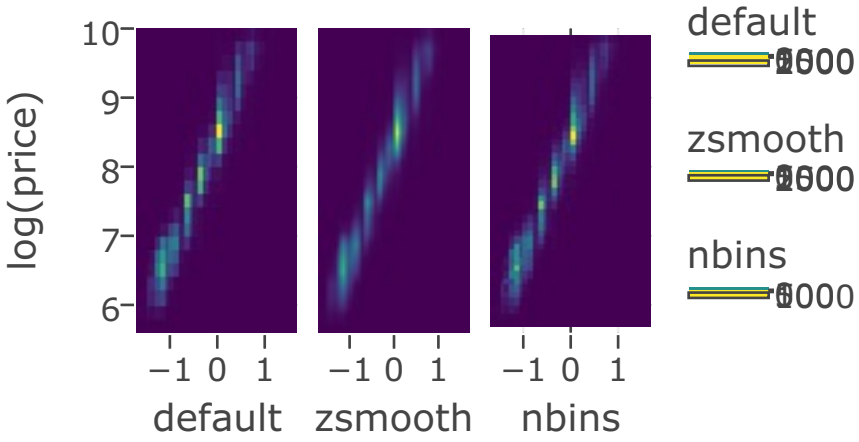


Figure 2.29: Three different uses of `histogram2d()`

2.5.2 Rectangular binning in R

In Bars & histograms, we leveraged a number of algorithms in R for computing the "optimal" number of bins for a histogram, via `hist()`, and routing those results to `add_bars()`. There is a surprising lack of research and computational tools for the 2D analog, and among the research that does exist, solutions usually depend on characteristics of the unknown underlying distribution, so the typical approach is to assume a Gaussian form (**?**). Kernel density estimation is another non

Practically speaking, that assumption is not very useful, but thankfully , and specifically the `kde2d()` function from the **MASS** package, provides a well-supported rule-of-thumb.

```
kde_count <- function(x, y, ...) {
  kde <- MASS::kde2d(x, y, ...)
  df <- with(kde, setNames(expand.grid(x, y), c("x", "y")))
  df$count <- with(kde, c(z) * length(x) * diff(x)[1] * diff(y)[1])
```

```
  data.frame(df)
}

kd <- with(diamonds, kde_count(log(carat), log(price), n = 30))
plot_ly(kd, x = ~x, y = ~y, z = ~count) %>%
  add_heatmap() %>%
  colorbar(title = "Number of diamonds")
```

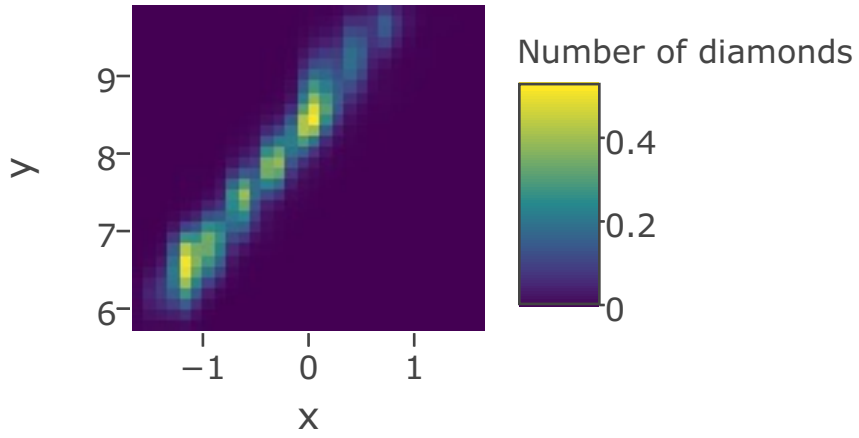


Figure 2.30: 2D Density estimation via the `kde2d()` function

2.5.3 Basic heatmaps

The `add_heatmap()` function can also handle categorical x and y variables.

```
corr <- cor(diamonds[vapply(diamonds, is.numeric, logical(1))])
plot_ly(x = rownames(corr), y = colnames(corr), z = corr) %>%
  add_heatmap() %>%
  colorbar(limits = c(-1, 1))
```

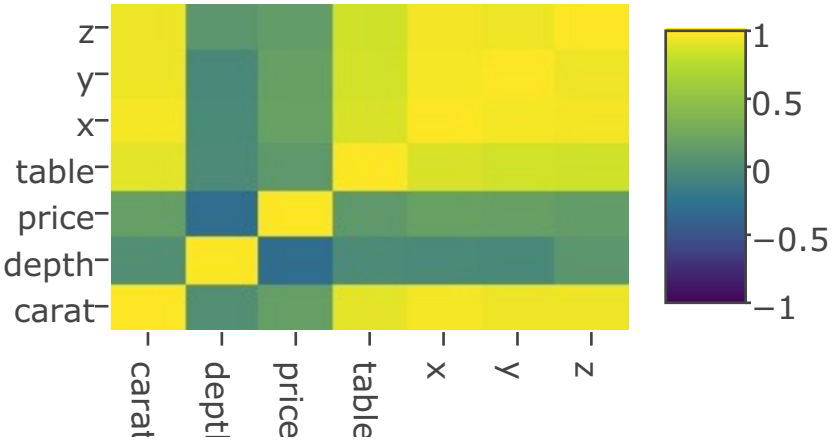


Figure 2.31:

A heatmap with categorical axes is basically a two-way table. The `Titanic` dataset is a four-way table that we can put into a tidy data frame using the `tidy()` function from the **broom** package.

```
broom::tidy(Titanic) %>%
  plot_ly(x = ~interaction(Age, Survived), y = ~interaction(Class, Sex)) %>%
  add_heatmap(z = ~Freq)
```

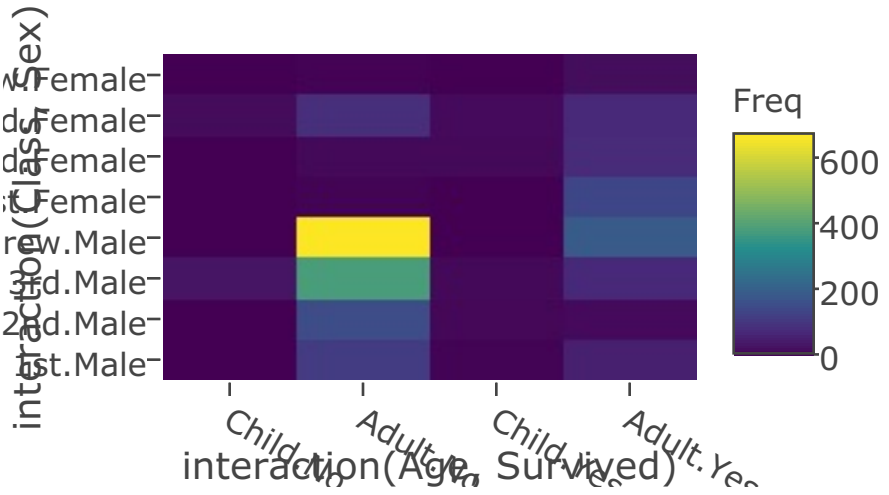


Figure 2.32:

```
#one_heatmap <- function(d) {
#  plot_ly(d, x = ~Class, y = ~Sex) %>%
#    add_heatmap(z = ~Freq) %>%
#    add_annotations(
#      ~unique(paste(Age, Survived, sep = ", ")), x = 0.5, y = 1,
#      yanchor = "bottom", xanchor = "middle",
#      xref = "paper", yref = "paper", showarrow = FALSE
#    )
#}
#
#tidy(Titanic) %>%
#  group_by(Age, Survived) %>%
#  do(p = one_heatmap(.)) %>%
#  subplot(
#    nrows = 2, margin = c(0.01, 0.01, 0.04, 0.04),
#    shareX = TRUE, shareY = TRUE,
#    titleY = FALSE, titleX = FALSE
#  )
```

2.5.4 Contours

```
plot_ly(diamonds, x = ~cut, y = ~clarity) %>%
  add_histogram2dcontour()
```

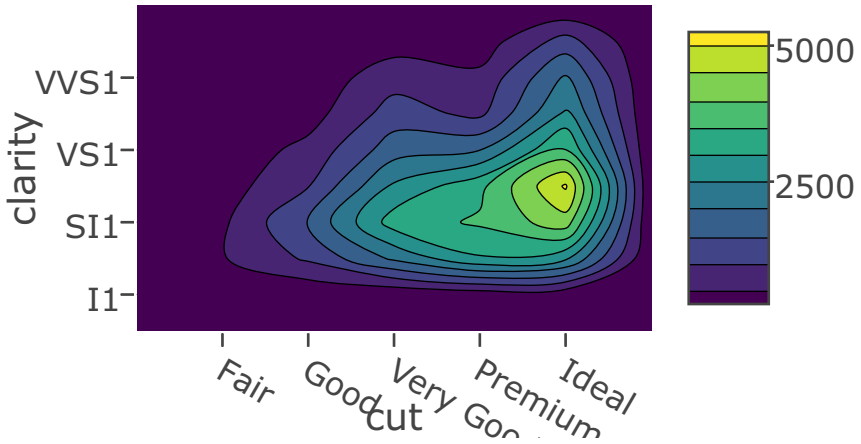


Figure 2.33:

2.6 3D plots

2.6.1 Surface

2.6.2 Mesh

2.7 Annotations

The `add_annotations()` function

Chapter 3

Arranging multiple views

The `subplot()` function provides a flexible interface for arranging multiple **plotly** plots in a single view. It is more flexible than most trellis display frameworks (e.g., ggplot2's `facet_wrap()`) as you don't have to condition on a value of common variable in each display (**?**). Its capabilities and interface is similar to the `grid.arrange()` function from the **gridExtra** package, which allows you to arrange multiple **grid** grobs in a single view, effectively providing a way to arrange (possibly unrelated) **ggplot2** and/or **lattice** plots in a single view (**?**); (**?**); (**?**). The simplest way to use it is to pass plotly objects directly to `subplot()`.

```
library(plotly)
p1 <- plot_ly(economics, x = ~date, y = ~unemploy, name = "unemploy") %>% add_lines()
p2 <- plot_ly(economics, x = ~date, y = ~uempmed, name = "uempmed") %>% add_lines()
subplot(p1, p2)
```

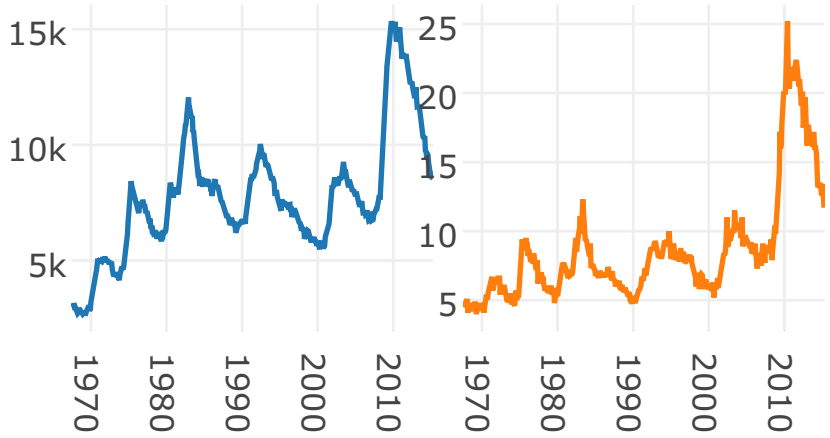


Figure 3.1:

Although `subplot()` accepts an arbitrary number of plot objects, passing a *list* of plots can save typing and redundant code when dealing with a large number of plots. To demonstrate, let's create one time series for each variable in the `economics` dataset and share the x-axis so that zoom/pan events are synchronized across each series:

```
vars <- setdiff(names(economics), "date")
plots <- lapply(vars, function(var) {
  plot_ly(economics, x = ~date, y = as.formula(paste0("~", var)), name = var) %>%
    add_lines()
})
```