2

Overview

This part of the book teaches you how to leverage the **plotly** R package to create a variety of interactive graphics. There are two main ways to creating a **plotly** object: either by transforming a **ggplot2** object (via `ggplotly()`) into a **plotly** object or by directly initializing a **plotly** object with `plot_ly()`/`plot_geo()`/`plot_mapbox()`. Both approaches have somewhat complementary strengths and weaknesses, so it can pay off to learn both approaches. Moreover, both approaches are an implementation of the Grammar of Graphics and both are powered by the JavaScript graphing library plotly.js, so many of the same concepts and tools that you learn for one interface can be reused in the other.

The subsequent chapters within this 'Creating views' part dive into specific examples and use cases, but this introductory chapter outlines some over-arching concepts related to **plotly** in general. It also provides definitions for terminology used throughout the book and introduces some concepts useful for understanding the infrastructure behind any **plotly** object. Most of these details aren't necessarily required to get started with **plotly**, but it will inevitably help you get 'un-stuck', write better code, and do more advanced things with **plotly**.

2.1 Intro to `plot_ly()`

Any graph made with the **plotly** R package is powered by the JavaScript library plotly.js[1]. The `plot_ly()` function provides a 'direct' interface to plotly.js with some additional abstractions to help reduce typing. These abstractions, inspired by the Grammar of Graphics and **ggplot2**,

[1] `https://github.com/plotly/plotly.js`

make it much faster to iterate from one graphic to another, making it easier to discover interesting features in the data (Wilkinson, 2005; Wickham, 2009). To demonstrate, we'll use `plot_ly()` to explore the `diamonds` dataset from **ggplot2** and learn a bit how **plotly** and plotly.js work along the way.

```
# load the plotly R package
library(plotly)

# load the diamonds dataset from the ggplot2 package
data(diamonds, package = "ggplot2")
diamonds
#> # A tibble: 53,940 x 10
#>   carat cut   color clarity depth table price     x
#>   <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int> <dbl>
#> 1 0.23  Ideal E     SI2      61.5    55   326  3.95
#> 2 0.21  Prem~ E     SI1      59.8    61   326  3.89
#> 3 0.23  Good  E     VS1      56.9    65   327  4.05
#> 4 0.290 Prem~ I     VS2      62.4    58   334  4.2
#> 5 0.31  Good  J     SI2      63.3    58   335  4.34
#> 6 0.24  Very~ J     VVS2     62.8    57   336  3.94
#> # ... with 5.393e+04 more rows, and 2 more variables:
#> #   y <dbl>, z <dbl>
```

If we assign variable names (e.g., `cut`, `clarity`, etc.) to visual properties (e.g., `x`, `y`, `color`, etc.) within `plot_ly()`, as done in Figure 2.1, it tries to find a sensible geometric representation of that information for us. Shortly we'll cover how to specify these geometric representations (as well as other visual encodings) to create different kinds of charts.

```
# create three visualizations of the diamonds dataset
plot_ly(diamonds, x = ~cut)
plot_ly(diamonds, x = ~cut, y = ~clarity)
plot_ly(diamonds, x = ~cut, color = ~clarity, colors = "Accent")
```

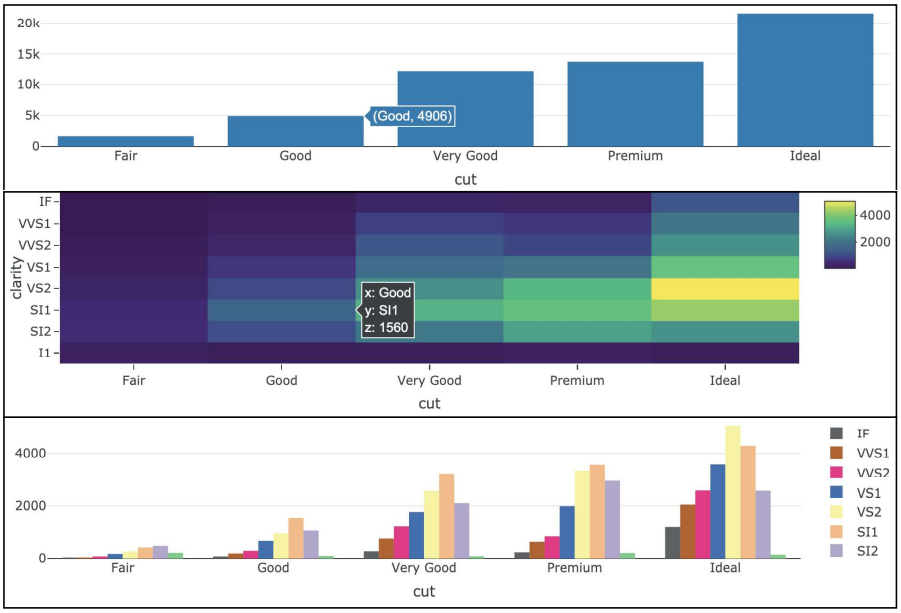

FIGURE 2.1: Three examples of visualizing categorical data with `plot_ly()`: (top) mapping `cut` to `x` yields a bar chart, (middle) mapping `cut` & `clarity` to `x` & `y` yields a heatmap, and (bottom) mapping `cut` & `clarity` to `x` & `color` yields a dodged bar chart.

The `plot_ly()` function has numerous arguments that are unique to the R package (e.g., `color`, `stroke`, `span`, `symbol`, `linetype`, etc.) and make it easier to encode data variables (e.g., diamond clarity) as visual properties (e.g., color). By default, these arguments map values of a data variable to a visual range defined by the plural form of the argument. For example, in the bottom panel of 2.1, `color` is used to map each level of diamond clarity to a different color, then `colors` is used to specify the range of colors (which, in this case, the `"Accent"` color palette from the **RColorBrewer** package, but one can also supply custom color codes or a color palette function like `colorRamp()`). Figure 2.2 provides a visual diagram of how this particular mapping works, but the same sort of idea can be applied to other visual properties like size, shape, linetype, etc.

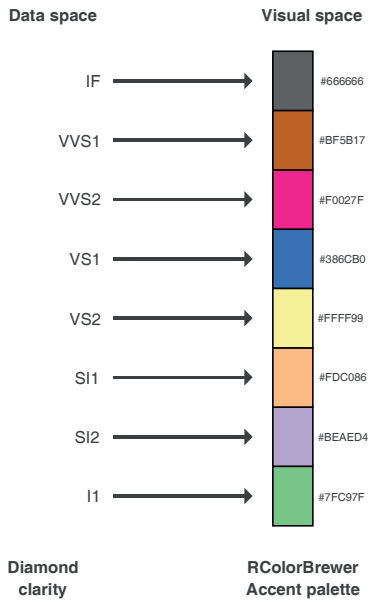

FIGURE 2.2: Mapping data values to a visual color range.

Since these arguments map data values to a visual range by default, you will obtain unexpected results if you try to specify the visual range directly, as in the top portion of Figure 2.3. If you want to specify the visual range directly, use the `I()` function to declare this value to be taken 'AsIs', as in the bottom portion of Figure 2.3. Throughout this book, you'll see lots of examples that leverage these arguments, especially in Chapter 3. Another good resource to learn more about these arguments (especially their defaults) is the R documentation page available by entering `help(plot_ly)` in your R console.

```
# doesn't produce black bars
plot_ly(diamonds, x = ~cut, color = "black")
# produces red bars with black outline
plot_ly(
  diamonds,
  x = ~cut,
  color = I("red"),
```

```
  stroke = I("black"),
  span = I(2)
)
```

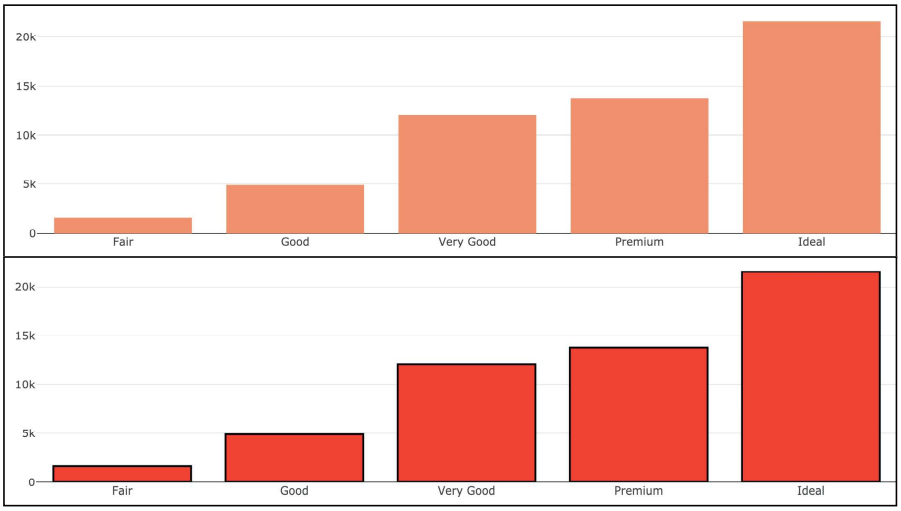

FIGURE 2.3: Using `I()` to supply visual properties directly instead of mapping values to a visual range. In the top portion of this figure, the value `'black'` is being mapped to a visual range spanned by `colors` (which, for discrete data, defaults to `'Set2'`).

The **plotly** package takes a purely functional approach to a layered grammar of graphics (Wickham, 2010).[2] The purely functional part means, (almost) every function anticipates a **plotly** object as input to its first argument and returns a modified version of that **plotly** object. Furthermore, that modification is completely determined by the input values to the function (i.e., it doesn't rely on any side effects, unlike, for example, base R graphics). For a quick example, the `layout()` function anticipates a **plotly** object in its first argument and its other arguments add and/or modify various layout components of that object (e.g., the title):

[2] If you aren't already familiar with the grammar of graphics or **ggplot2**, we recommend reading the Data Visualization chapter from the *R for Data Science* book. https://r4ds.had.co.nz/data-visualisation.html

```
layout(
  plot_ly(diamonds, x = ~cut),
  title = "My beatiful histogram"
)
```

For more complex plots that modify a **plotly** graph many times over, code written in this way can become cumbersome to read. In particular, we have to search for the innermost part of the R expression, then work outwards towards the end result. The `%>%` operator from the **magrittr** package allows us to rearrange this code so that we can read the sequence of modifications from left-to-right rather than inside-out (Bache and Wickham, 2014). The `%>%` operator enables this by placing the object on the left-hand side of the `%>%` into the first argument of the function of the right-hand side.

```
diamonds %>%
  plot_ly(x = ~cut) %>%
  layout(title = "My beatiful histogram")
```

In addition to `layout()` for adding/modifying part(s) of the graph's layout, there are also a family of `add_*()` functions (e.g., `add_histogram()`, `add_lines()`, etc.) that define how to render data into geometric objects. Borrowing terminology from the layered grammar of graphics, these functions add a graphical layer to a plot. A *layer* can be thought of as a group of graphical elements that can be sufficiently described using only 5 components: data, aesthetic mappings (e.g., assigning `clarity` to `color`), a geometric representation (e.g., rectangles, circles, etc.), statistical transformations (e.g., sum, mean, etc.), and positional adjustments (e.g., dodge, stack, etc.). If you're paying attention, you'll notice that in the examples thus far, we have not specified a layer! The layer has been added for us automatically by `plot_ly()`. To be explicit about what `plot_ly(diamonds, x = ~cut)` generates, we should add a `add_histogram()` layer:

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

As you'll learn more about in Chapter 5, **plotly** has both `add_histogram()` and `add_bars()`. The difference is that `add_histogram()` performs *statistics* (i.e., a binning algorithm) dynamically in the web browser, whereas `add_bars()` requires the bar heights to be pre-specified. That means, to replicate the last example with `add_bars()`, the number of observations must be computed ahead of time.

```
diamonds %>%
  dplyr::count(cut) %>%
  plot_ly() %>%
  add_bars(x = ~cut, y = ~n)
```

There are numerous other `add_*()` functions that calculate statistics in the browser (e.g., `add_histogram2d()`, `add_contour()`, `add_boxplot()`, etc.), but most other functions aren't considered statistical. Making the distinction might not seem useful now, but they have their own respective trade-offs when it comes to speed and interactivity. Generally speaking, non-statistical layers will be faster and more responsive at runtime (since they require less computational work), whereas the statistical layers allow for more flexibility when it comes to client-side interactivity, as covered in Chapter 16. Practically speaking, the difference in performance is often negligible. The more common bottleneck occurs when attempting to render lots of graphical elements at a time (e.g., a scatterplot with a million points). In those scenarios, you likely want to render your plot in Canvas rather than SVG (the default) via `toWebGL()`. For more information on improving performance, see Chapter 24.

In many scenarios, it can be useful to combine multiple graphical layers into a single plot. In this case, it becomes useful to know a few things about `plot_ly()`:

- Arguments specified in `plot_ly()` are *global*, meaning that any downstream `add_*()` functions inherit these arguments (unless `inherit = FALSE`).
- Data manipulation verbs from the **dplyr** package may be used to transform the `data` underlying a **plotly** object.[3]

Using these two properties of `plot_ly()`, Figure 2.4 demonstrates how we could leverage these properties of `plot_ly()` to do the following:

1. *Globally* assign `cut` to `x`.
2. Add a histogram layer (inherits the `x` from `plot_ly()`).
3. Use **dplyr** verbs to modify the `data` underlying the **plotly** object. Here we just count the number of diamonds in each `cut` category.
4. Add a layer of text using the summarized counts. Note that the global `x` mapping, as well as the other mappings local to this text layer (`text` and `y`), reflects data values from step 3.

```
library(dplyr)

diamonds %>%
  plot_ly(x = ~cut) %>%
  add_histogram() %>%
  group_by(cut) %>%
  summarise(n = n()) %>%
  add_text(
    text = ~scales::comma(n), y = ~n,
    textposition = "top middle",
    cliponaxis = FALSE
  )
```

[3] Technically speaking, these **dplyr** verbs are S3 generic functions that have a **plotly** method. In nearly every case, that method simply queries the data underlying the **plotly** object, applies the **dplyr** function, then adds the transformed data back into the resulting **plotly** object.

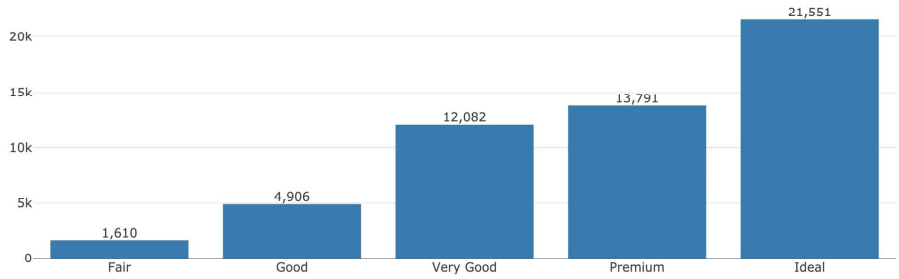

FIGURE 2.4: Using `add_histogram()`, `add_text()`, and **dplyr** verbs to compose a plot that leverages a raw form of the data (e.g., histogram) as well as a summarized version (e.g., text labels).

Before using multiple `add_*()` in a single plot, make sure that you actually want to show those layers of information on the same set of axes. If it makes sense to display the information on the same axes, consider making multiple **plotly** objects and combining them into a grid-like layout using `subplot()`, as described in Chapter 13. Also, when using **dplyr** verbs to modify the `data` underlying the **plotly** object, you can use the `plotly_data()` function to obtain the data at any point in time, which is primarily useful for debugging purposes (i.e., inspecting the data of a particular graphical layer).

```
diamonds %>%
  plot_ly(x = ~cut) %>%
  add_histogram() %>%
  group_by(cut) %>%
  summarise(n = n()) %>%
  plotly_data()
#> # A tibble: 5 x 2
#>   cut           n
#>   <ord>     <int>
#> 1 Fair       1610
#> 2 Good       4906
#> 3 Very Good 12082
#> 4 Premium   13791
#> 5 Ideal     21551
```

This introduction to `plot_ly()` has mainly focused on concepts unique to the R package **plotly** that are generally useful for creating most kinds of data views. The next section outlines how **plotly** generates plotly.js figures and how to inspect the underlying data structure that plotly.js uses to render the graph. Not only is this information useful for debugging, but it's also a nice way to learn how to work with plotly.js directly, which you may need to improve performance in **shiny** apps (Section 17.3.1) and/or for adding custom behavior with JavaScript (Chapter 18).

2.2 Intro to plotly.js

To recreate the plots in Figure 2.1 using plotly.js *directly*, it would take significantly more code and knowledge of plotly.js. That being said, learning how **plotly** generates the underlying plotly.js figure is a useful introduction to plotly.js itself, and knowledge of plotly.js becomes useful when you need more flexible control over **plotly**. As Figure 2.5 illustrates, when you print any **plotly** object, the `plotly_build()` function is applied to that object, and that generates an R list which adheres to a syntax that plotly.js understands. This syntax is a JavaScript Object Notation (JSON) specification that plotly.js uses to represent, serialize, and render web graphics. A lot of documentation you'll find online about plotly (e.g., the online figure reference[4]) implicitly refers to this JSON specification, so it can be helpful to know how to "work backwards" from that documentation (i.e., translate JSON into to R code). If you'd like to learn details about mapping between R and JSON, Chapter 19 provides an introduction aimed at R programmers, and Ooms (2014) provides a cohesive overview of the **jsonlite** package, which is what **plotly** uses to map between R and JSON.

[4] `https://plot.ly/r/reference/`

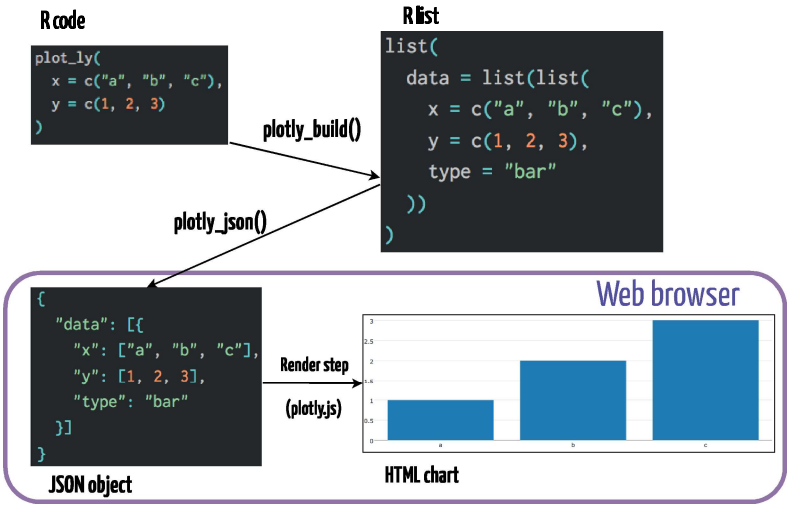

FIGURE 2.5: A diagram of what happens when you print a **plotly** graph.

For illustration purposes, Figure 2.5 shows how this workflow applies to a simple bar graph (with values directly supplied instead of a data column name reference like Figure 2.1), but the same concept applies for any graph created via **plotly**. As the diagram suggests, both the `plotly_build()` and `plotly_json()` functions can be used to inspect the underlying data structure on both the R and JSON side of things. For example, Figure 2.6 shows the `data` portion of the JSON created for the last graph in Figure 2.6.

```
p <- plot_ly(diamonds, x = ~cut, color = ~clarity, colors = "Accent")
plotly_json(p)
```

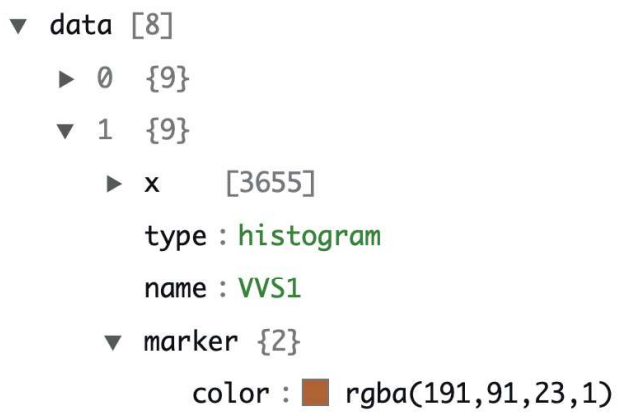

FIGURE 2.6: A portion of the JSON data behind the bottom plot of Figure 2.1. This dodged bar chart has eight layers of data (i.e., eight traces), one for each level of `clarity`.

In plotly.js terminology, a *figure* has two key components: `data` (aka, traces) and a `layout`. A *trace* defines a mapping from data and visuals.[5] Every trace has a *type* (e.g., histogram, pie, scatter, etc.) and the trace type determines what other attributes (i.e., visual and/or interactive properties, like `x`, `hoverinfo`, `name`) are available to control the trace mapping. That is, not every trace attribute is available to every trace type, but many attributes (e.g., the `name` of the trace) are available in every trace type and serve a similar purpose. From Figure 2.6, we can see that it takes multiple traces to generate the dodged bar chart, but instead of clicking through JSON viewer, sometimes it's easier to use `plotly_build()` and compute on the plotly.js figure definition to verify certain things exist. Since **plotly** uses the **htmlwidgets** standard[6], the actual plotly.js figure definition appears under a list element named `x` (Vaidyanathan et al., 2016).

[5] A trace is similar in concept to a layer (as defined in Section 2.1), but it's not quite the same. In many cases, like the bottom panel of Figure 2.1, it makes sense to implement a single layer as multiple traces. This is due to the design of plotly.js and how traces are tied to legends and hover behavior.

[6] The **htmlwidgets** package provides a foundation for other packages to implement R bindings to JavaScript libraries so that those bindings work in various contexts (e.g., the R console, RStudio, inside **rmarkdown** documents, **shiny** apps, etc.). For more info and examples, see the website `http://www.htmlwidgets.org`.

```
# use plotly_build() to get at the plotly.js definition
# behind *any* plotly object
b <- plotly_build(p)

# Confirm there 8 traces
length(b$x$data)
#> [1] 8

# Extract the `name` of each trace. plotly.js uses `name` to
# populate legend entries and tooltips
purrr::map_chr(b$x$data, "name")
#> [1] "IF" "VVS1" "VVS2" "VS1" "VS2" "SI1" "SI2" "I1"

# Every trace has a type of histogram
unique(purrr::map_chr(b$x$data, "type"))
#> [1] "histogram"
```

Here we've learned that **plotly** creates 8 histogram traces to generate the dodged bar chart: one trace for each level of `clarity`.[7] Why one trace per category? As illustrated in Figure 2.7, there are two main reasons: to populate a tooltip and legend entry for each level of `clarity` level.

[7] Although the x-axis is discrete, plotly.js still considers this a histogram because it generates counts in the browser. Learn more about the difference between histograms and bar charts in Chapter 5.

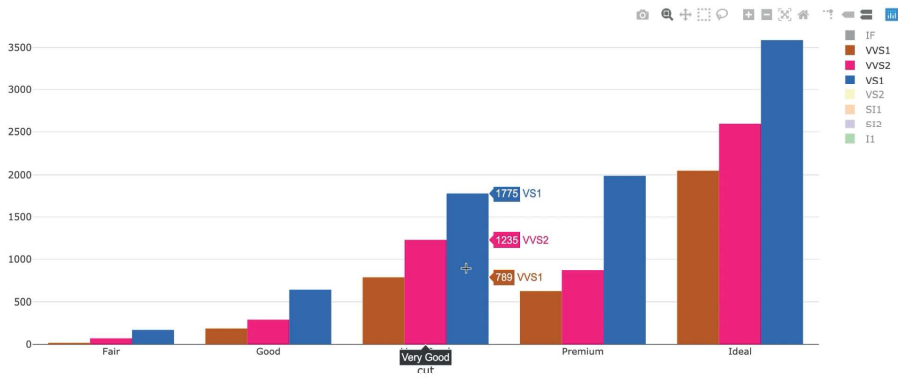

FIGURE 2.7: Leveraging two interactive features that require one trace per level of `clarity`: (1) Using 'Compare data on hover' mode to get counts for every level of `clarity` for a given level of `cut`, and (2) using the ability to hide/show clarity levels via their legend entries. For a video demonstration of the interactive, see `https://bit.ly/intro-show-hide-preview`. For the interactive, see `https://plotly-r.com/interactives/intro-show-hide.html`

If we investigated further, we'd notice that `color` and `colors` are not officially part of the plotly.js figure definition; the `plotly_build()` function has effectively transformed that information into a sensible plotly.js figure definition (e.g., `marker.color` contains the actual bar color codes). In fact, the `color` argument in `plot_ly()` is just one example of an abstraction the R package has built on top of plotly.js to make it easier to map data values to visual attributes, and many of these are covered in Chapter 3.

2.3 Intro to `ggplotly()`

The `ggplotly()` function from the **plotly** package has the ability to translate **ggplot2** to **plotly**. This functionality can be really helpful for quickly

adding interactivity to your existing **ggplot2** workflow.[8] Moreover, even if you know `plot_ly()` and plotly.js well, `ggplotly()` can still be desirable for creating visualizations that aren't necessarily straightforward to achieve without it. To demonstrate, let's explore the relationship between `price` and other variables from the well-known `diamonds` dataset.

Hexagonal binning (i.e., `geom_hex()`) is useful way to visualize a 2D density[9], like the relationship between `price` and `carat` as shown in Figure 2.8. From Figure 2.8, we can see there is a strong positive linear relationship between the *log* of carat and price. It also shows that for many, the carat is only rounded to a particular number (indicated by the light blue bands) and no diamonds are priced around $1500. Making this plot interactive makes it easier to decode the hexagonal colors into the counts that they represent.

```
p <- ggplot(diamonds, aes(x = log(carat), y = log(price))) +
  geom_hex(bins = 100)
ggplotly(p)
```

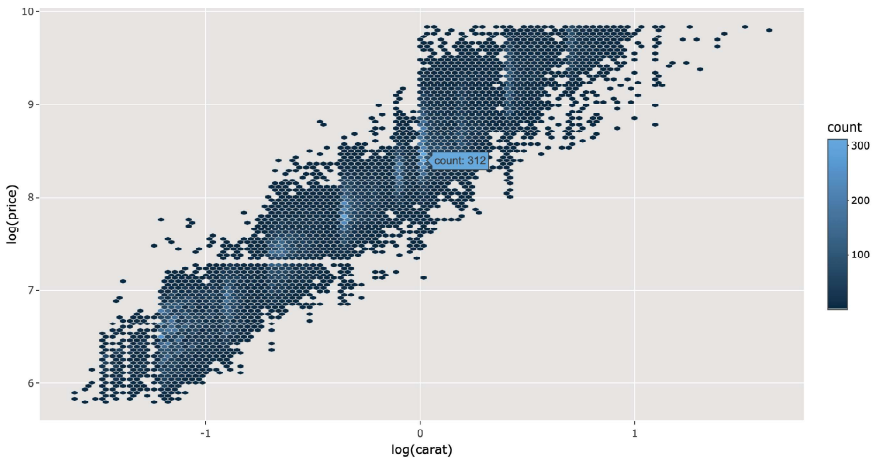

FIGURE 2.8: A hexbin plot of diamond carat versus price.

[8] This section is not meant to teach you **ggplot2**, but rather to help point out when and why it might be preferable to `plot_ly()`. If you're new to **ggplot2** and would like to learn it, see Section 1.3.3.

[9] See Section 7 for approaches using `plot_ly()`

I often use `ggplotly()` over `plot_ly()` to leverage **ggplot2**'s consistent and expressive interface for exploring statistical summaries across groups. For example, by including a discrete `color` variable (e.g., `cut`) with `geom_freqpoly()`, you get a frequency polygon for each level of that variable. This ability to quickly generate visual encodings of statistical summaries across an arbitrary number of groups works for basically any geom (e.g., `geom_boxplot()`, `geom_histogram()`, `geom_density()`, etc.) and is a key feature of **ggplot2**.

```
p <- ggplot(diamonds, aes(x = log(price), color = clarity)) +
    geom_freqpoly()
ggplotly(p)
```

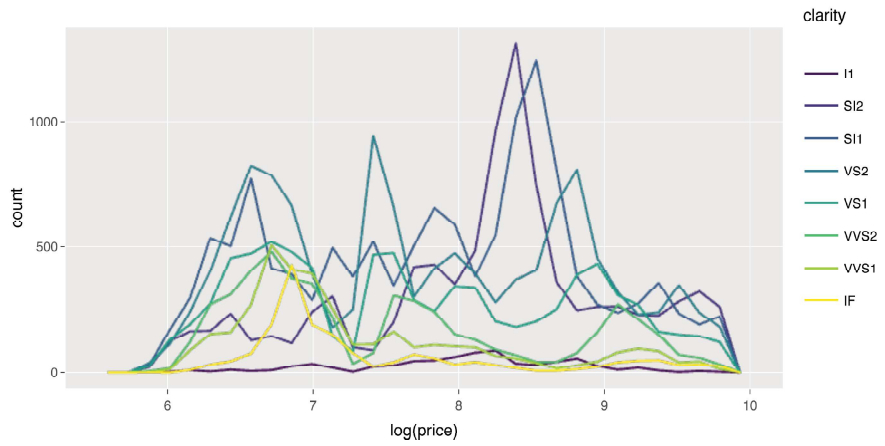

FIGURE 2.9: Frequency polygons of diamond price by diamond clarity. This visualization indicates there may be significant main effects.

Now, to see how `price` varies with both `cut` and `clarity`, we could repeat this same visualization for each level of `cut`. This is where **ggplot2**'s `facet_wrap()` comes in handy. Moreover, to facilitate comparisons, we can have `geom_freqpoly()` display relative rather than absolute frequencies. By making this plot interactive, we can more easily compare particular levels of clarity (as shown in Figure 2.10) by leveraging the legend filtering capabilities.

```
p <- ggplot(diamonds, aes(x = log(price), color = clarity)) +
    geom_freqpoly(stat = "density") +
    facet_wrap(~cut)
ggplotly(p)
```

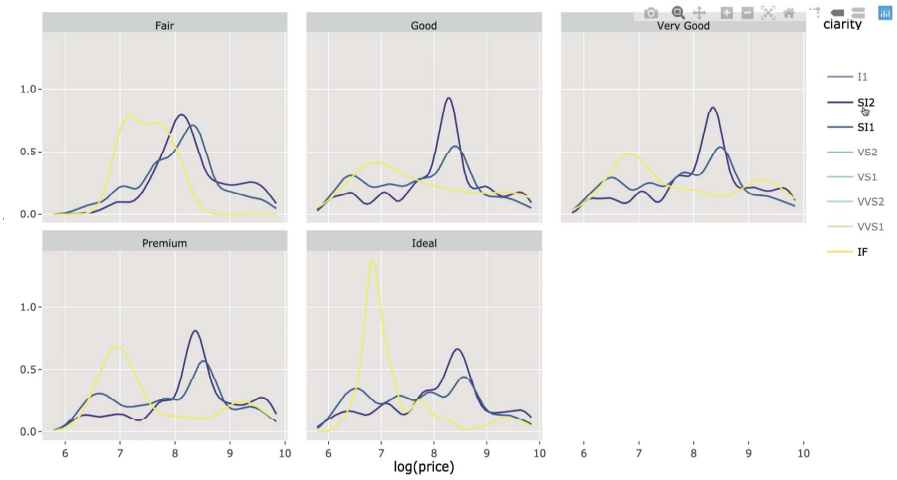

FIGURE 2.10: Diamond price by clarity and cut. For a video demonstration of the interactive, see `https://bit.ly/freqpoly-facet`. For the interactive, see `https://plotly-r.com/interactives/freqpoly-facet.html`

In addition to supporting most of the 'core' **ggplot2** API, `ggplotly()` can automatically convert any **ggplot2** extension packages that return a 'standard' **ggplot2** object. By standard, I mean that the object is comprised of 'core' **ggplot2** data structures and not the result of custom geoms.[10] Some great examples of R packages that extend **ggplot2** using core data structures are **ggforce**, **naniar**, and **GGally** (Pedersen, 2019; Tierney et al., 2018; Schloerke et al., 2016).

Figure 2.11 demonstrates another way of visualizing the same information found in Figure 2.10 using `geom_sina()` from the **ggforce** pack-

[10] As discussed in Chapter 34, `ggplotly()` can actually convert custom geoms as well, but each one requires a custom hook, and many custom geoms are not yet supported.

age (instead of `geom_freqpoly()`). This visualization jitters the raw data within the density for each group allowing us not only to see where the majority observations fall within a group, but also across all groups. By making this layer interactive, we can query individual points for more information and zoom into interesting regions. The second layer of Figure 2.11 uses **ggplot2**'s `stat_summary()` to overlay a 95% confidence interval estimated via a Bootstrap algorithm via the **Hmisc** package (Harrell Jr et al., 2019).

```
p <- ggplot(diamonds, aes(x=clarity, y=log(price), color=clarity)) +
    ggforce::geom_sina(alpha = 0.1) +
    stat_summary(fun.data = "mean_cl_boot", color = "black") +
    facet_wrap(~cut)

# WebGL is a lot more efficient at rendering lots of points
toWebGL(ggplotly(p))
```

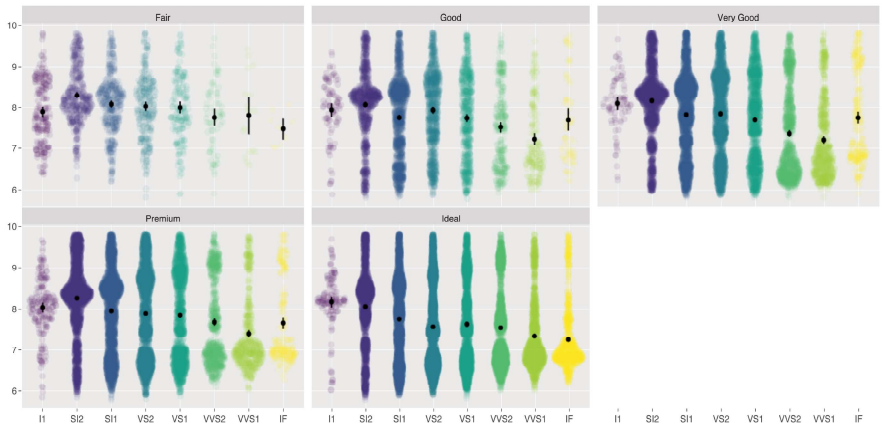

FIGURE 2.11: A sina plot of diamond price by clarity and cut.

As noted by Wickham and Grolemund (2016), it's surprising that the diamond price would decline with an increase of diamond clarity. As it turns out, if we account for the carat of the diamond, then we see that better diamond clarity does indeed lead to a higher diamond price, as shown in Figure 2.12. Seeing such a strong pattern in the residuals of

simple linear model of carat vs. price indicates that our model could be greatly improved by adding `clarity` as a predictor of `price`.

```
m <- lm(log(price) ~ log(carat), data = diamonds)
diamonds <- modelr::add_residuals(diamonds, m)
p <- ggplot(diamonds, aes(x = clarity, y = resid, color = clarity)) +
    ggforce::geom_sina(alpha = 0.1) +
    stat_summary(fun.data = "mean_cl_boot", color = "black") +
    facet_wrap(~cut)
toWebGL(ggplotly(p))
```

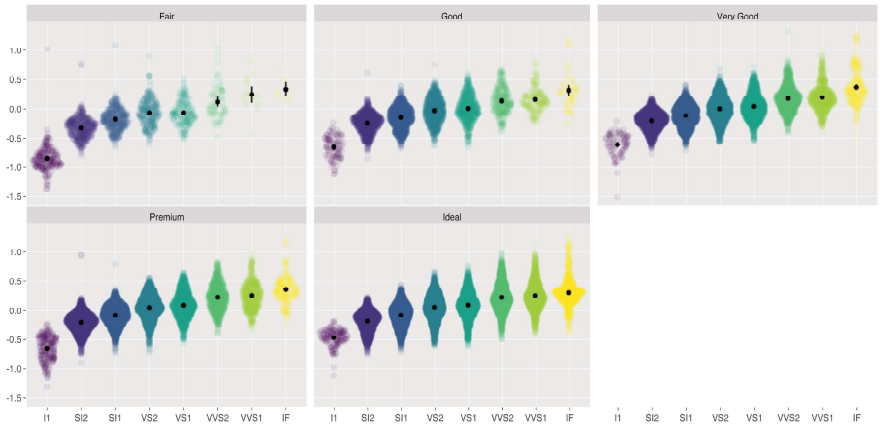

FIGURE 2.12: A sina plot of diamond price by clarity and cut, after accounting for carat.

As discussed in Section 16.4.7, the **GGally** package provides a convenient interface for making similar types of model diagnostic visualizations via the `ggnostic()` function. It also provides a convenience function for visualizing the coefficient estimates and their standard errors via the `ggcoef()` function. Figure 2.13 shows how injecting interactivity into this plot allows us to query exact values and zoom in on the most interesting regions.

```
library(GGally)
m <- lm(log(price) ~ log(carat) + cut, data = diamonds)
gg <- ggcoef(m)
# dynamicTicks means generate new axis ticks on zoom
ggplotly(gg, dynamicTicks = TRUE)
```

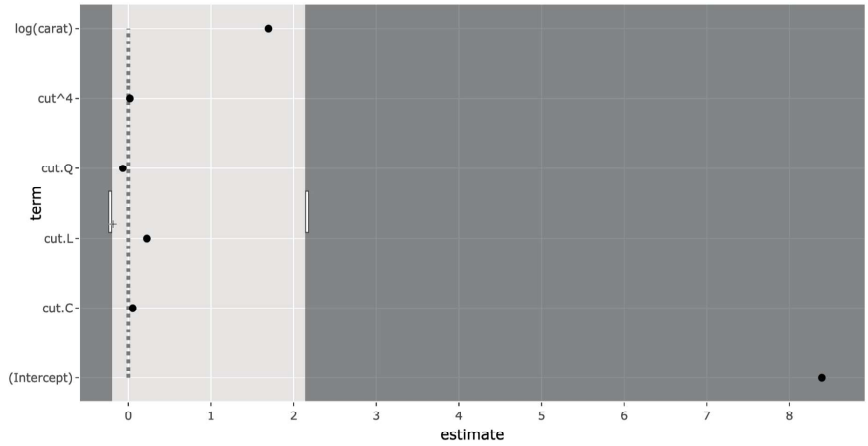

FIGURE 2.13: Zooming in on a coefficient plot generated from the `ggcoef()` function from the **GGally** package. For a video demonstration of the interactive, see `https://bit.ly/GGally`. For the interactive, see `https://plotly-r.com/interactives/ggally.html`

Although the `diamonds` dataset does not contain any missing values, it's a very common problem in real data analysis problems. The **naniar** package provides a suite of computational and visual resources for working with and revealing structure in missing values. All the **ggplot2** based visualizations return an object that can be converted by `ggplotly()`. Moreover, **naniar** provides a custom geom, `geom_miss_point()`, that can be useful for visualizing missingness structure. Figure 2.14 demonstrates this by introducing fake missing values to the diamond price.

```
library(naniar)
# fake some missing data
diamonds$price_miss <- ifelse(diamonds$depth>60, diamonds$price, NA)
p <- ggplot(diamonds, aes(x = clarity, y = log(price_miss))) +
    geom_miss_point(alpha = 0.1) +
    stat_summary(fun.data = "mean_cl_boot", colour = "black") +
    facet_wrap(~cut)
toWebGL(ggplotly(p))
```

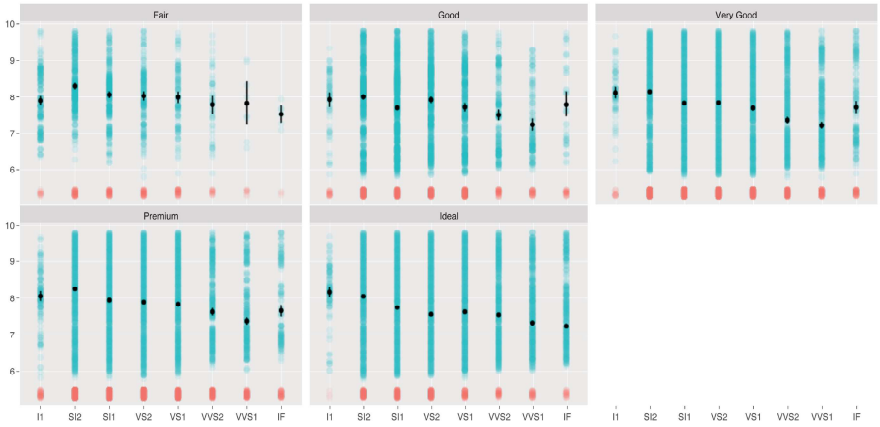

FIGURE 2.14: Using the `geom_miss_point()` function from the **naniar** package to visualize missing values in relation to non-missing values. Missing values are shown in red.

In short, the **ggplot2** ecosystem provides a world-class exploratory visualization toolkit, and having the ability to quickly insert interactivity such as hover, zoom, and filter via `ggplotly()` makes it even more powerful for exploratory analysis. In this introduction to `ggplotly()`, we've only seen relatively simple techniques that come for free out-of-the-box, but the true power of interactive graphics lies in linking multiple views. In that part of the book, you can find lots of examples of linking multiple (`ggplotly()` and `plot_ly()`) graphs purely client-side as well as with **shiny**.

It's also worth mentioning that `ggplotly()` conversions are not always perfect and **ggplot2** doesn't provide an API for interactive features, so sometimes it's desirable to modify the return values of `ggplotly()`. Chapter 33 talks generally about modifying the data structure underlying `ggplotly()` (which, by the way, uses the same a plotly.js figure definition as discussed in Section 2.2). Moreover, Section 25.2 outlines various ways to customize the tooltip that `ggplotly()` produces.

3

Scattered foundations

As we learned in Section 2.2, a plotly.js figure contains one (or more) trace(s), and every trace has a type. The trace type scatter is great for drawing low-level geometries (e.g., points, lines, text, and polygons) and provides the foundation for many `add_*()` functions (e.g., `add_markers()`, `add_lines()`, `add_paths()`, `add_segments()`, `add_ribbons()`, `add_area()`, and `add_polygons()`) as well as many `ggplotly()` charts. These scatter-based layers provide a more convenient interface to special cases of the scatter trace by doing a bit of data wrangling and transformation under-the-hood before mapping to scatter trace(s). For a simple example, `add_lines()` ensures lines are drawn according to the ordering of `x`, which is desirable for a time series plotting. This behavior is subtly different than `add_paths()` which uses row ordering instead.

```
library(plotly)
data(economics, package = "ggplot2")

# sort economics by psavert, just to
# show difference between paths and lines
p <- economics %>%
  arrange(psavert) %>%
  plot_ly(x = ~date, y = ~psavert)

add_paths(p)
add_lines(p)
```

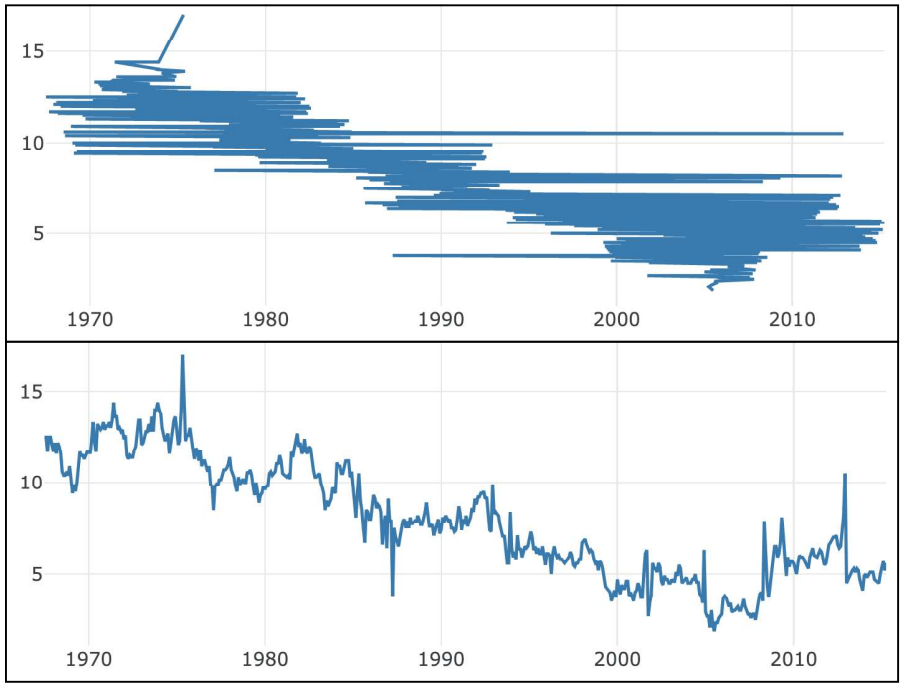

FIGURE 3.1: The difference between `add_paths()` and `add_lines()`: The top panel connects observations according to the ordering of `psavert` (personal savings rate), whereas the bottom panel connects observations according to the ordering of `x` (the date).

Section 2.1 introduced 'aesthetic mapping' arguments (unique to the R package) which make it easier to map data to visual properties (e.g., `color`, `linetype`, etc.). In addition to these arguments, **dplyr** groupings can be used to ensure there is at least one geometry per group. The top panel of Figure 3.1 demonstrates how `group_by()` could be used to effectively wrap the time series from Figure 3.1 by year, which can be useful for visualizing annual seasonality. Another approach to generating at least one geometry per 'group' is to provide categorical variable to a relevant aesthetic (e.g., `color`), as shown in the bottom panel of Figure 3.1.

```
library(lubridate)
econ <- economics %>%
```

```
  mutate(yr = year(date), mnth = month(date))

# One trace (more performant, but less interactive)
econ %>%
  group_by(yr) %>%
  plot_ly(x = ~mnth, y = ~uempmed) %>%
  add_lines(text = ~yr)

# Multiple traces (less performant, but more interactive)
plot_ly(econ, x = ~mnth, y = ~uempmed) %>%
  add_lines(color = ~ordered(yr))

# The split argument guarantees one trace per group level (regardless
# of the variable type). This is useful if you want a consistent
# visual property over multiple traces
# plot_ly(econ, x = ~mnth, y = ~uempmed) %>%
#   add_lines(split = ~yr, color = I("black"))
```

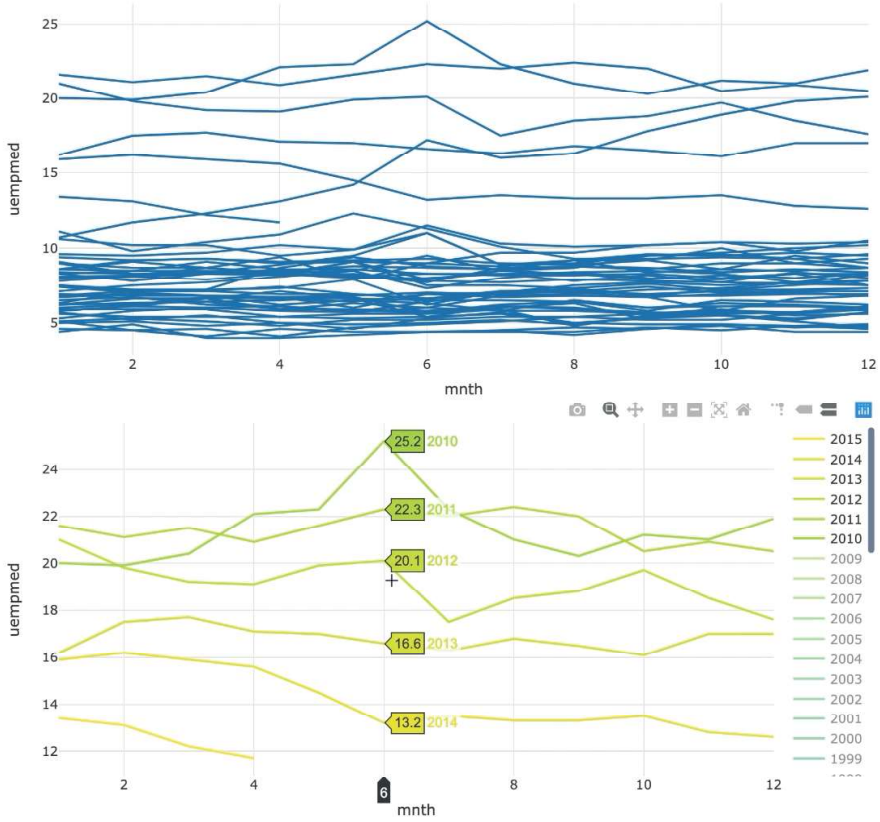

FIGURE 3.2: Drawing multiple lines using **dplyr** groups (top panel) versus a categorical `color` mapping (bottom panel). Comparatively speaking, the bottom panel has more interactive capabilities (e.g., legend-based filtering and multiple tooltips), but it does not scale as well with many lines. For a video demonstration of the interactive, see `https://bit.ly/scatter-lines`. For the interactive, see `https://plotly-r.com/interactives/scatter-lines.html`

Not only do these plots differ in visual appearance, they also differ in interactive capabilities, computational performance, and underlying implementation. That's because, the grouping approach (top panel of Figure 3.2) uses just one plotly.js trace (more performant, less interactive), whereas the `color` approach (bottom panel of Figure 3.2) generates one trace per line/year. In this case, the benefit of having

multiple traces is that we can perform interactive filtering via the legend and compare multiple y-values at a given x. The cost of having those capabilities is that plots begin to suffer from performance issues after a few hundred traces, whereas thousands of lines can be rendered fairly easily in one trace. See Chapter 24 for more details on scaling and performance.

These features make it easier to get started using plotly.js, but it still pays off to learn how to use plotly.js directly. You won't find plotly.js attributes listed as explicit arguments in any **plotly** function (except for the special `type` attribute), but they are passed along verbatim to the plotly.js figure definition through the ... operator. The scatter-based layers in this chapter fix the `type` plotly.js attribute to `"scatter"` as well as the `mode`[1] (e.g., `add_markers()` uses `mode='markers'` etc.), but you could also use the lower-level `add_trace()` to work more directly with plotly.js. For example, Figure 3.3 shows how to render markers, lines, and text in the same scatter trace. It also demonstrates how to leverage *nested* plotly.js attributes, like `textfont`[2] and `xaxis`[3]; these attributes contain other attributes, so you need to supply a suitable named list to these arguments.

```
set.seed(99)
plot_ly() %>%
 add_trace(
   type = "scatter",
   mode = "markers+lines+text",
   x = 4:6,
   y = 4:6,
   text = replicate(3, praise::praise("You are ${adjective}! ")),
   textposition = "right",
   hoverinfo = "text",
   textfont = list(family = "Roboto Condensed", size = 16)
 ) %>%
 layout(xaxis = list(range = c(3, 8)))
```

[1] https://plot.ly/r/reference/#scatter-mode
[2] https://plot.ly/r/reference/#scatter-textfont
[3] https://plot.ly/r/reference/#layout-xaxis

FIGURE 3.3: Using the generic `add_trace()` function to render markers, lines, and text in a single scatter trace. This `add_trace()` function, as well as any `add_*()` function allows you to directly specify plotly.js attributes.

If you are new to plotly.js, I recommend taking a bit of time to look through the plotly.js attributes that are available to the scatter trace type and think how you might be able to use them. Most of these attributes work for other trace types as well, so learning an attribute once for a specific plot can pay off in other contexts as well. The online plotly.js figure reference, `https://plot.ly/r/reference/#scatter`, is a decent place to search and learn about the attributes, but I recommend using the `schema()` function instead for a few reasons:

- `schema()` provides a bit more information than the online docs (e.g., value types, default values, acceptable ranges, etc.).
- The interface makes it a bit easier to traverse and discover new attributes.
- You can be absolutely sure it matches the version used in the R package (the online docs might use a different – probably older – version).

```
schema()
```

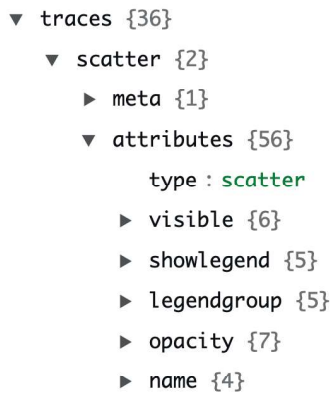

FIGURE 3.4: Using `schema()` function to traverse through the attributes available to a given trace type (e.g., scatter).

The sections that follow in this chapter demonstrate various types of data views using scatter-based layers. In an attempt to avoid duplication of documentation, a particular emphasis is put on features only currently available from the R package (e.g., the aesthetic mapping arguments).

3.1 Markers

This section details scatter traces with a `mode` of `"markers"` (i.e., `add_markers()`). For simplicity, many of the examples here use `add_markers()` with a numeric x and y axis, which results in scatterplot: a common way to visualize the association between two quantitative variables. The content that follows is still relevant markers displayed non-numeric x and y (aka dot pots) as shown in Section 3.1.6.

3.1.1 Alpha blending

As Unwin (2015) notes, scatterplots can be useful for exposing other important features including: casual relationships, outliers, clusters, gaps, barriers, and conditional relationships. A common problem with scatterplots, however, is overplotting, meaning that there are multiple observations occupying the same (or similar) x/y locations. Figure 3.5 demonstrates one way to combat overplotting via alpha blending. When dealing with tens of thousands of points (or more), consider using `toWebGL()` to render plots using Canvas rather than SVG (more in Chapter 24), or leveraging 2D density estimation (Section 7.2).

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

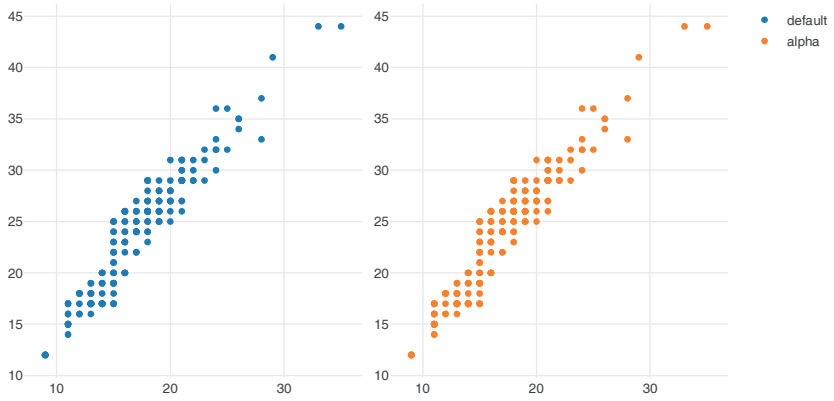

FIGURE 3.5: Combating overplotting in a scatterplot with alpha blending.

3.1.2 Colors

As discussed in Section 2.2, mapping a discrete variable to `color` produces one trace per category, which is desirable for its legend and hover properties. On the other hand, mapping a *numeric* variable to `color` produces one trace, as well as a colorbar[4] guide for visually decoding 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 (Berkeley Institute for Data Science, 2016). Viridis is also the default colorscale for ordered factors.

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

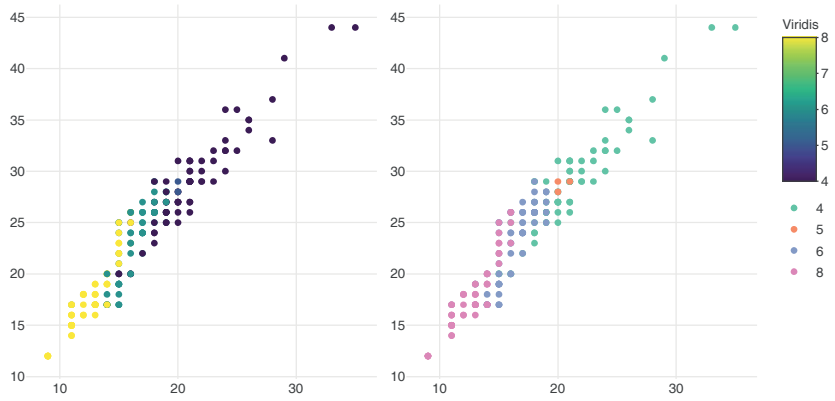

FIGURE 3.6: Variations on a numeric color mapping.

[4] https://plot.ly/r/reference/#scatter-marker-colorbar

There are numerous ways to alter the default color scale via the `colors` argument. This argument excepts one of the following: (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 conscious of using a sequential colorscale for numeric variables (and ordered factors) as shown in Figure 3.7, and a qualitative colorscale for discrete variables as shown in Figure 3.8.

```
col1 <- c("#132B43", "#56B1F7")
col2 <- viridisLite::inferno(10)
col3 <- colorRamp(c("red", "white", "blue"))
subplot(
  add_markers(p, color = ~cyl, colors = col1) %>%
    colorbar(title = "ggplot2 default"),
  add_markers(p, color = ~cyl, colors = col2) %>%
    colorbar(title = "Inferno"),
  add_markers(p, color = ~cyl, colors = col3) %>%
    colorbar(title = "colorRamp")
) %>% hide_legend()
```

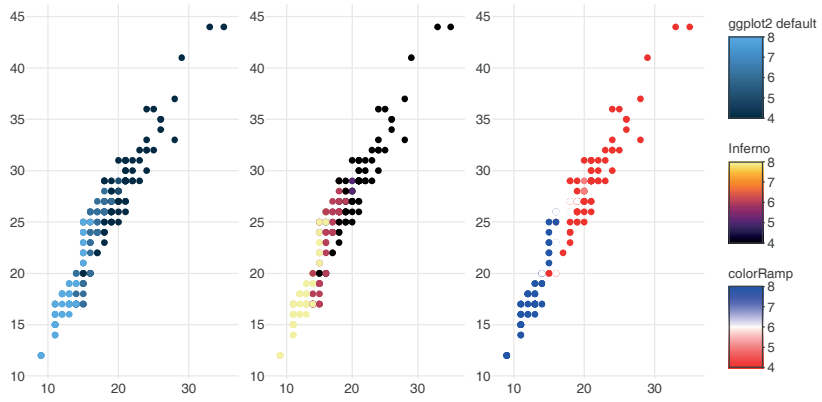

FIGURE 3.7: Three variations on a numeric color mapping.

```
col1 <- "Accent"
col2 <- colorRamp(c("red", "blue"))
col3 <- c(`4` = "red", `5` = "black", `6` = "blue", `8` = "green")
subplot(
  add_markers(p, color = ~factor(cyl), colors = col1),
  add_markers(p, color = ~factor(cyl), colors = col2),
  add_markers(p, color = ~factor(cyl), colors = col3)
) %>% hide_legend()
```

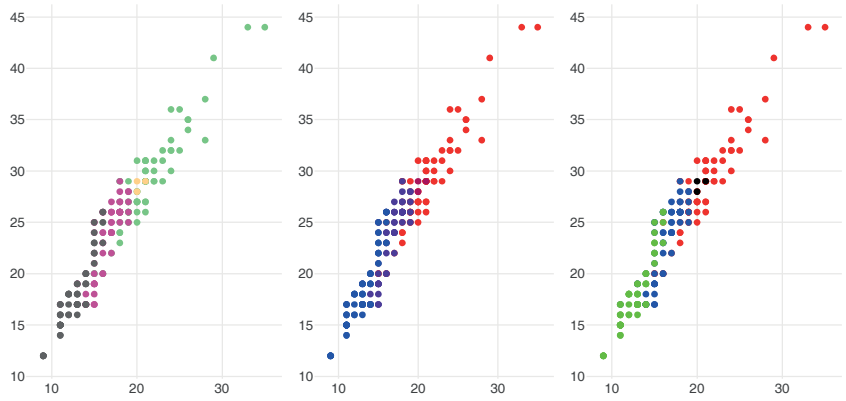

FIGURE 3.8: Three variations on a discrete color mapping.

As introduced in Figure 2.3, color codes can be specified manually (i.e., avoid mapping data values to a visual range) by using the `I()` function. Figure 3.9 provides a simple example using `add_markers()`. Any color understood by the `col2rgb()` function from the **grDevices** package can be used in this way. Chapter 27 provides even more details about working with different color specifications when specifying colors manually.

```
add_markers(p, color = I("black"))
```

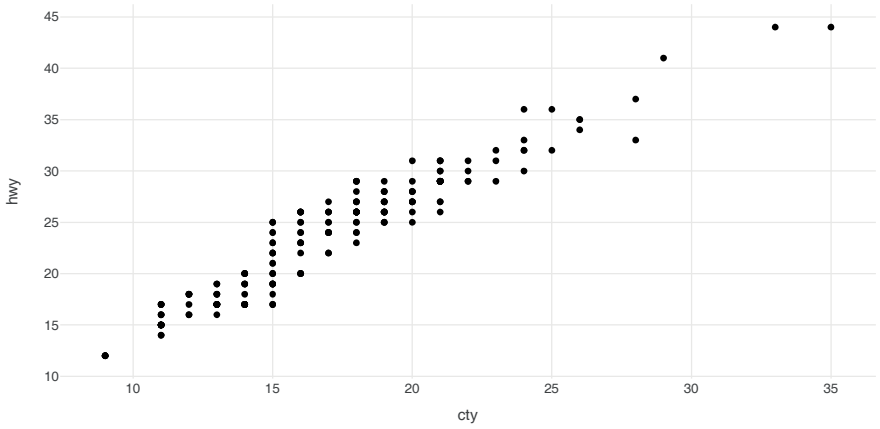

FIGURE 3.9: Setting a fixed color directly using `I()`.

The `color` argument is meant to control the 'fill-color' of a geometric object, whereas `stroke` (Section 3.1.4) is meant to control the 'outline-color' of a geometric object. In the case of `add_markers()`, that means `color` maps to the plotly.js attribute `marker.color`[5] and `stroke` maps to `marker.line.color`[6]. Not all, but many, marker symbols have a notion of stroke.

3.1.3 Symbols

The `symbol` argument can be used to map data values to the `marker.symbol` plotly.js attribute. It uses the same semantics that we've already seen for `color`:

- A numeric mapping generates trace.
- A discrete mapping generates multiple traces (one trace per category).
- The plural, `symbols`, can be used to specify the visual range for the mapping.
- Mappings are avoided entirely through `I()`.

For example, the left panel of Figure 3.10 uses a numeric mapping, and the right panel uses a discrete mapping. As a result, the left panel is

[5] `https://plot.ly/r/reference/#scatter-marker-color`
[6] `https://plot.ly/r/reference/#scatter-marker-line-color`

linked to the first legend entry, whereas the right panel is linked to the bottom three legend entries. When plotting multiple traces and no color is specified, the plotly.js colorway[7] is applied (i.e., each trace will be rendered a different color). To set a fixed color, 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"))
)
```

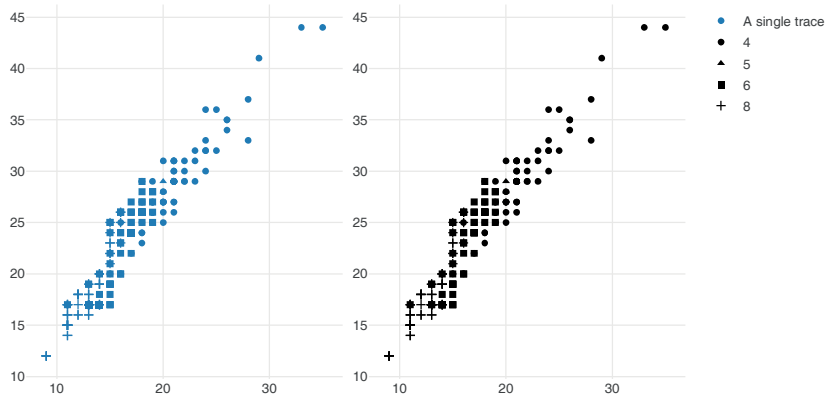

FIGURE 3.10: Mapping symbol to a numeric variable (left panel) and a factor (right panel).

There are two ways to specify the visual range of `symbols`: (1) numeric codes (interpreted as a `pch` codes) or (2) a character string specifying a valid `marker.symbol` value. Figure 3.11 uses pch codes (left panel) as well as their corresponding `marker.symbol` name (right panel) to specify the visual range.

[7] `https://plot.ly/r/reference/#layout-colorway`

```
subplot(
  add_markers(p, symbol = ~cyl, symbols = c(17, 18, 19)),
  add_markers(
    p, color = I("black"),
    symbol = ~factor(cyl),
    symbols = c("triangle-up", "diamond", "circle")
  )
)
```

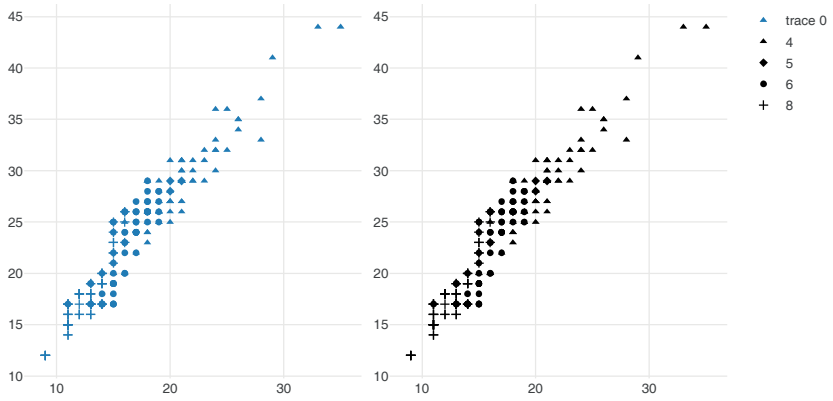

FIGURE 3.11: Specifying the visual range of `symbols`.

These `symbols` (i.e., the visual range) can also be supplied directly to `symbol` through `I()`. For example, Figure 3.12 fixes the marker symbol to a diamond shape.

```
plot_ly(mpg, x = ~cty, y = ~hwy) %>%
  add_markers(symbol = I(18), alpha = 0.5)
```

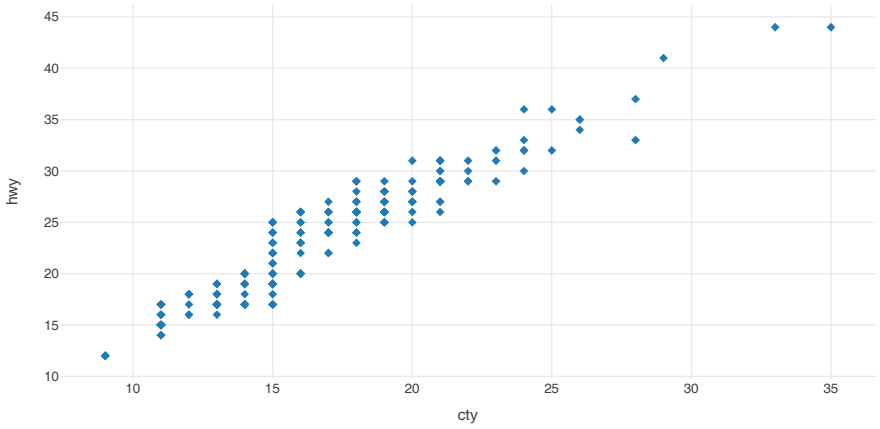

FIGURE 3.12: Setting a fixed symbol directly using `I()`.

If you'd like to see all the symbols available to **plotly**, as well as a method for supplying your own custom glyphs, see Chapter 28.

3.1.4 Stroke and span

The `stroke` argument follows the same semantics as `color` and `symbol` when it comes to variable mappings and specifying visual ranges. Typically you don't want to map data values to `stroke`, you just want to specify a fixed outline color. For example, Figure 3.13 modifies Figure 3.12 to simply add a black outline. By default, the `span`, or width of the stroke, is zero, you'll likely want to set the width to be around one pixel.

```
plot_ly(mpg, x = ~cty, y = ~hwy, alpha = 0.5) %>%
  add_markers(symbol = I(18), stroke = I("black"), span = I(1))
```

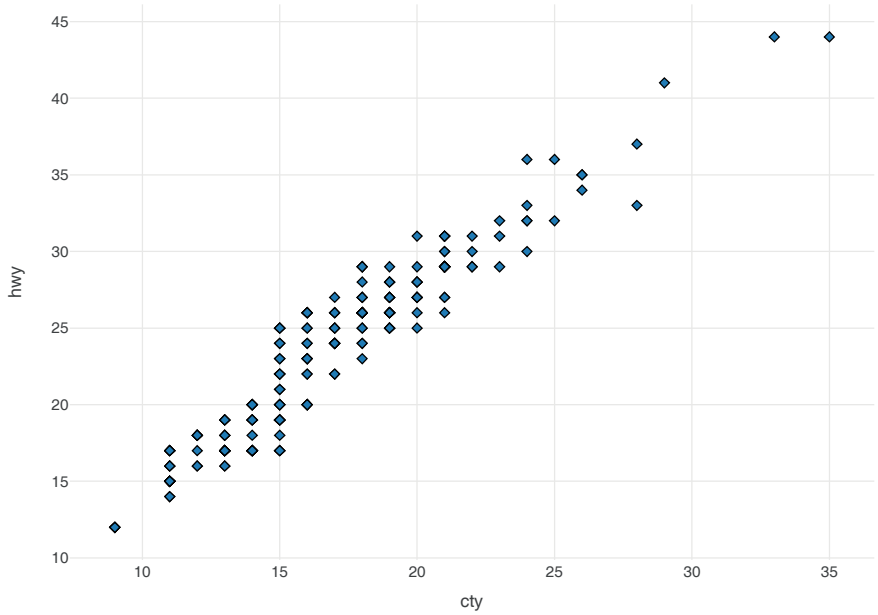

FIGURE 3.13: Using `stroke` and `span` to control the outline color as well as the width of that outline.

3.1.5 Size

For scatterplots, the `size` argument controls the area of markers (unless otherwise specified via sizemode[8]), and *must* be a numeric variable. The `sizes` argument controls the minimum and maximum size of circles, in pixels:

```
p <- plot_ly(mpg, x = ~cty, y = ~hwy, alpha = 0.3)
subplot(
  add_markers(p, size = ~cyl, name = "default"),
  add_markers(p, size = ~cyl, sizes = c(1, 500), name = "custom")
)
```

[8] https://plot.ly/r/reference/#scatter-marker-sizemode

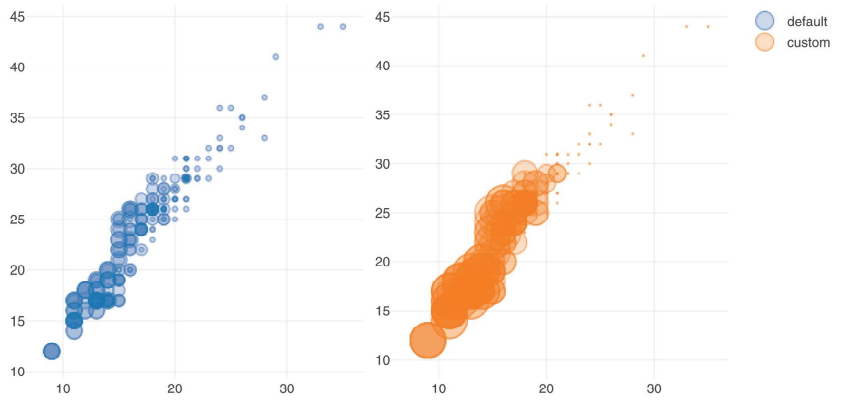

FIGURE 3.14: Controlling the size range via `sizes` (measured in pixels).

Similar to other arguments, `I()` can be used to specify the size directly. In the case of markers, `size` controls the `marker.size`[9] plotly.js attribute. Remember, you always have the option to set this attribute directly by doing something similar to Figure 3.15.

```
plot_ly(mpg, x = ~cty, y = ~hwy, alpha = 0.3, size = I(30))
```

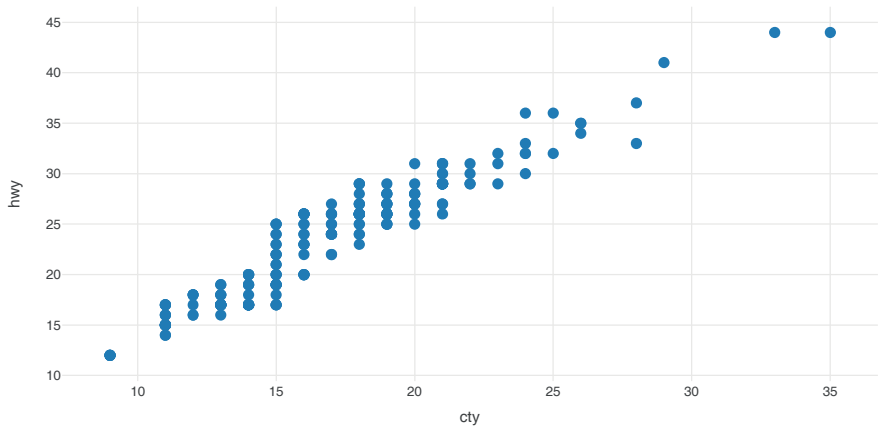

FIGURE 3.15: Setting a fixed marker size directly using `marker.size`.

[9] `https://plot.ly/r/reference/#scatter-marker-size`

3.1.6 Dotplots and 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 preferable to pie charts since comparing position along a common scale is much easier than comparing angle or area (Cleveland and McGill, 1984; Heer and Bostock, 2010). Furthermore, dotplots can be preferable to bar charts, especially when comparing values within a narrow range far away from 0 (Few, 2006). 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 (Messing, 2012).

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 (Lander, 2016) 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 would be better to use error objects[10]. Figure 3.16 uses the `tidy()` function from the **broom** package (Robinson, 2016) to obtain a data frame with one row per model coefficient, and produce a coefficient plot with error bars along the x-axis.

```
# Fit a full-factorial linear model
m <- lm(
  Sepal.Length ~ Sepal.Width * Petal.Length * Petal.Width,
  data = iris
)

# (1) get a tidy() data structure of covariate-level info
# (e.g., point estimate, standard error, etc.)
# (2) make sure term column is a factor ordered by the estimate
# (3) plot estimate by term with an error bar for the standard error
```

[10] https://plot.ly/r/reference/#scatter-error_x

```
broom::tidy(m) %>%
  mutate(term = forcats::fct_reorder(term, estimate)) %>%
  plot_ly(x = ~estimate, y = ~term) %>%
  add_markers(
    error_x = ~list(value = std.error),
    color = I("black"),
    hoverinfo = "x"
  )
```

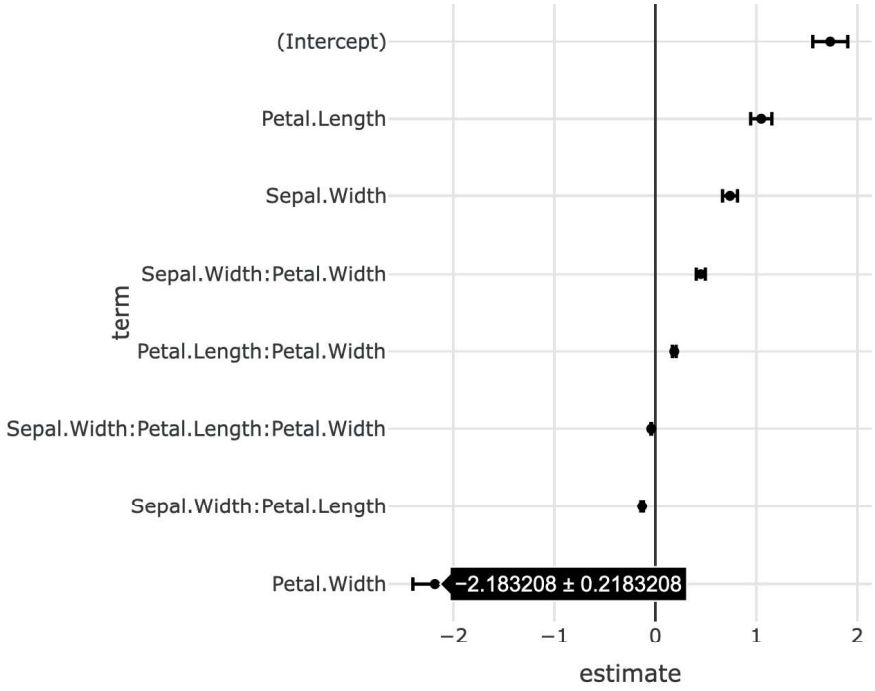

FIGURE 3.16: A coefficient plot.

3.2 Lines

Many of the same principles we learned about aesthetic mappings with respect to markers (Section 3.1) also apply to lines.[11] Moreover, at the start of this chapter (namely Figure 3.2) we also learned how to use **dplyr**'s `group_by()` to ensure there is at least one geometry (in this case, line) per group. We also learned the difference between `add_paths()` and `add_lines()`; the former draws lines according to row ordering, whereas the latter draw them according to `x`. In this chapter, we'll learn about `linetype`/`linetype`, an aesthetic that applies to lines and polygons. We'll also discuss some other important chart types that can be implemented with `add_paths()`, `add_lines()`, and `add_segments()`.

3.2.1 Linetypes

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()`. As Figure 3.17 demonstrates, once we have the data 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[13] or plotly.js dash values[14].

```
library(dplyr)
top5 <- txhousing %>%
```

[11] At the time of writing, the plotly.js attributes `line.width` and `line.color`[12] do not support multiple values, meaning a single line trace can only have one width/color in 2D line plot, and consequently numeric `color`/`size` mappings won't work. This isn't necessarily true for 3D paths/lines and there will likely be support for these features for 2D paths/lines in WebGL in the near future.

[13] `https://github.com/wch/r-source/blob/e5b21d/src/library/graphics/man/par.Rd#L726-L743`

[14] `https://plot.ly/r/reference/#scatter-line-dash`

```
  group_by(city) %>%
  summarise(m = mean(sales, na.rm = TRUE)) %>%
  arrange(desc(m)) %>%
  top_n(5)

tx5 <- semi_join(txhousing, top5, by = "city")

plot_ly(tx5, x = ~date, y = ~median) %>%
  add_lines(linetype = ~city)
```

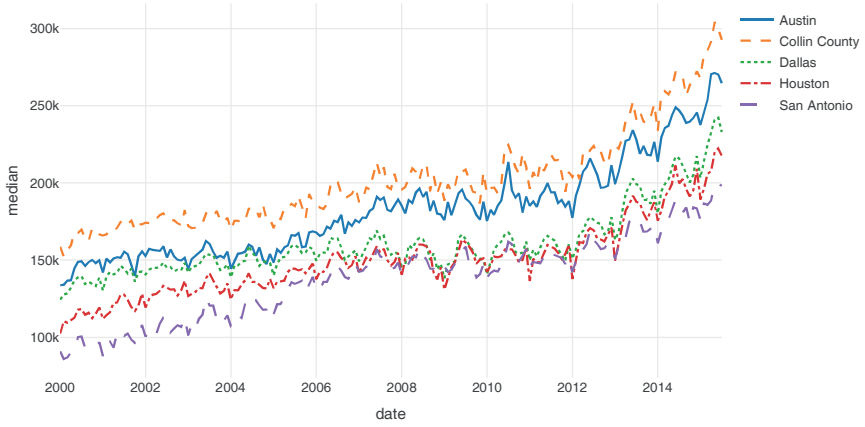

FIGURE 3.17: Using `color` and/or `linetype` to differentiate groups of lines.

If you'd like to control exactly which linetype is used to encode a particular data value, you can provide a named character vector, like in Figure 3.18. Note that this is similar to how we provided a discrete colorscale manually for markers in Figure 3.8.

```
ltys <- c(
  Austin = "dashdot",
  `Collin County` = "longdash",
  Dallas = "dash",
  Houston = "solid",
```

```
  `San Antonio` = "dot"
)

plot_ly(tx5, x = ~date, y = ~median) %>%
  add_lines(linetype = ~city, linetypes = ltys)
```

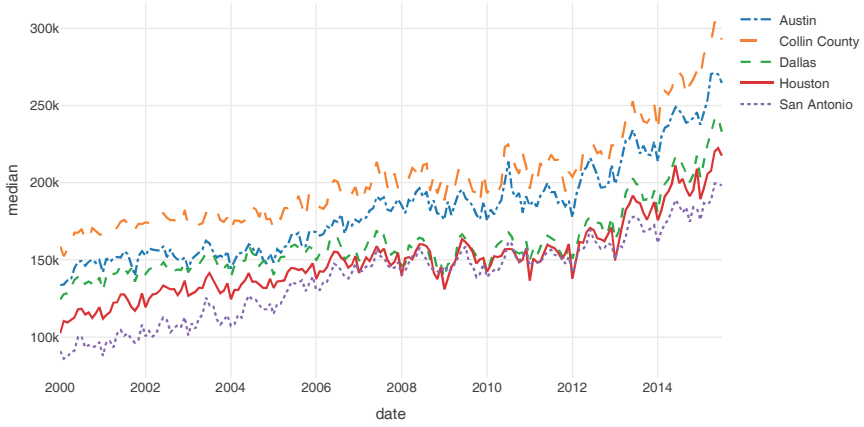

FIGURE 3.18: Providing a named character vector to linetypes in order to control exactly what linetype gets mapped to which city.

3.2.2 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 numerous useful chart types, including slopegraphs, dumbell charts, candlestick charts, and more. Slopegraphs and dumbell charts are useful for comparing numeric values across numerous categories. Candlestick charts are typically used for visualizing change in a financial asset over time.

Segments can also provide a useful alternative to `add_bars()` (covered in Chapter 5), especially for animations. In particular, Figure 14.5 of Section 14.2 shows how to implement an animated population pyramid using segments instead of bars.

3.2.2.1 Slopegraph

!range@range}

The slope graph, made popular by Tufte (2001), is a great way to compare the change in a measurement across numerous groups. This change could be along either a discrete or a continuous axis. For a continuous axis, the slopegraph could be thought of as a decomposition of a line graph into multiple segments. The **slopegraph** R package provides a succinct interface for creating slopegraphs with base or **ggplot2** graphics and also some convenient datasets which we'll make use of here (Leeper, 2017). Figure 3.19 recreates an example from Tufte (2001), using the `gdp` dataset from **slopegraph**, and demonstrates a common issue with labelling in slopegraphs; it's easy to have overlapping labels when anchoring labels on data values. For that reason, this implementation leverages **plotly** ability to interactively edit annotation positions. See Chapter 12 for similar examples of 'editing views'.

```
data(gdp, package = "slopegraph")
gdp$Country <- row.names(gdp)

plot_ly(gdp) %>%
  add_segments(
    x = 1, xend = 2,
    y = ~Year1970, yend = ~Year1979,
    color = I("gray90")
  ) %>%
  add_annotations(
    x = 1, y = ~Year1970,
    text = ~paste(Country, "  ", Year1970),
    xanchor = "right", showarrow = FALSE
  ) %>%
  add_annotations(
    x = 2, y = ~Year1979,
    text = ~paste(Year1979, "  ", Country),
    xanchor = "left", showarrow = FALSE
  ) %>%
  layout(
```

```
    title = "Current Receipts of Govermnent as a Percentage of GDP",
    showlegend = FALSE,
    xaxis = list(
      range = c(0, 3),
      ticktext = c("1970", "1979"),
      tickvals = c(1, 2),
      zeroline = FALSE
    ),
    yaxis = list(
      title = "",
      showgrid = FALSE,
      showticks = FALSE,
      showticklabels = FALSE
    )
  ) %>%
  config(edits = list(annotationPosition = TRUE))
```

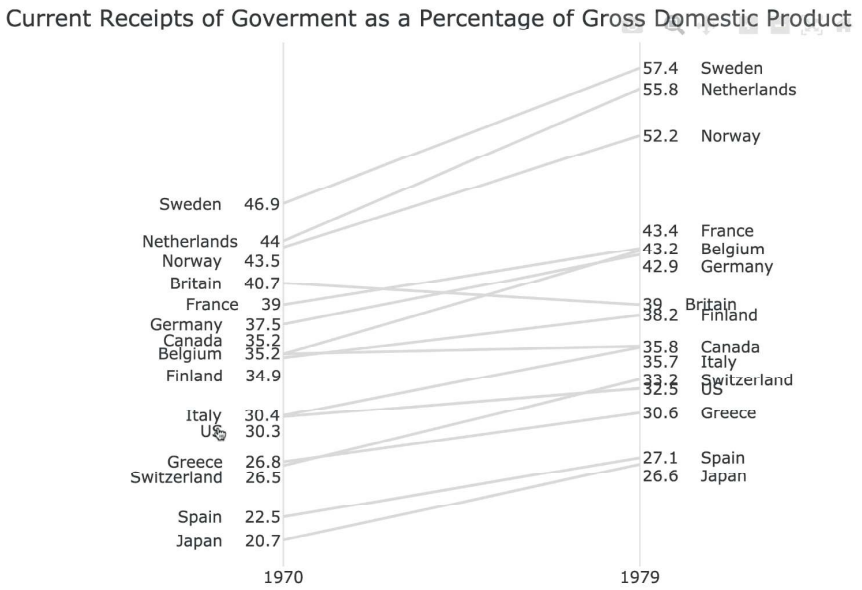

FIGURE 3.19: Interactively editing the label positioning in a slopegraph. For a video demonstration of the interactive, see `https://bit.ly/Slopegraph`. For the interactive, see `https://plotly-r.com/interactives/slopegraph.html`

3.2.2.2 Dumbell

So called dumbell charts are similar in concept to slope graphs, but not quite as general. They are typically used to compare two different classes of numeric values across numerous groups. Figure 3.20 uses the dumbell approach to show average miles per gallon city and highway for different car models. With a dumbell chart, it's always a good idea to order the categories by a sensible metric; for Figure 3.20, the categories are ordered by the city miles per gallon.

```
mpg %>%
  group_by(model) %>%
  summarise(c = mean(cty), h = mean(hwy)) %>%
  mutate(model = forcats::fct_reorder(model, c)) %>%
  plot_ly() %>%
  add_segments(
```

```
    x = ~c, y = ~model,
    xend = ~h, yend = ~model,
    color = I("gray"), showlegend = FALSE
  ) %>%
  add_markers(
    x = ~c, y = ~model,
    color = I("blue"),
    name = "mpg city"
  ) %>%
  add_markers(
    x = ~h, y = ~model,
    color = I("red"),
    name  = "mpg highway"
  ) %>%
  layout(xaxis = list(title = "Miles per gallon"))
```

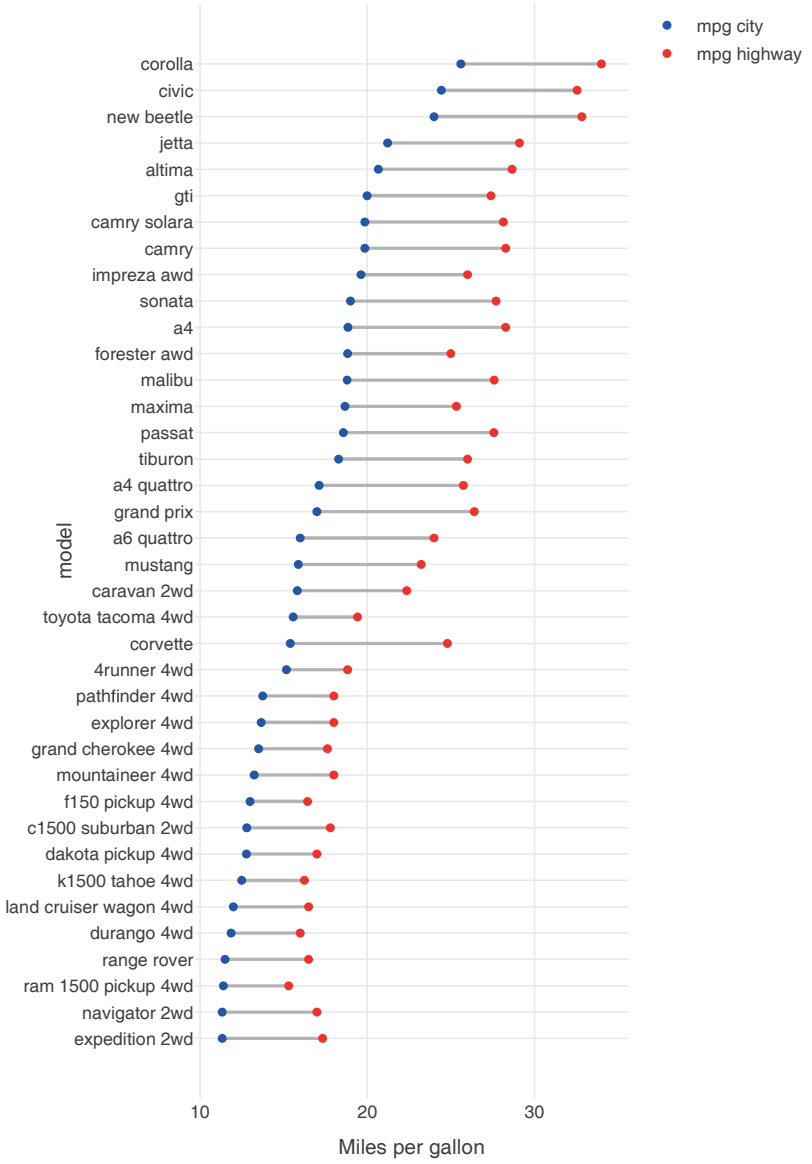

FIGURE 3.20: A dumbell chart of mile per gallon city vs. highway by model of car.

3.2.2.3 Candlestick

Figure 3.21 uses the **quantmod** package (Ryan, 2016) to obtain stock price data for Microsoft and plots two segments for each day: one to encode the opening/closing values, and one to encode the daily high/low. This implementation uses `add_segments()` to implement the candlestick chart, but more recent versions of plotly.js contain a candlestick[15] and ohlc[16] trace types, both of which are useful for visualizing financial data.

```
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()
```

[15] https://plot.ly/r/reference/#candlestick

[16] https://plot.ly/r/reference/#ohlc

FIGURE 3.21: A candlestick chart built with `add_segments()`. Note how the `color` mapping, which is a logical vector (`TRUE` if the closing value was higher than opening), creates two traces: a red trace indicating a drop in price and a green trace indicating a rise in price.

3.2.3 Density plots

In Chapter 5, we leverage 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()`. We can leverage the `density()` function for computing kernel density estimates in a similar way, and route the results to `add_lines()`, as is done in Figure 3.22.

```
kerns <- c("gaussian", "epanechnikov", "rectangular",
           "triangular", "biweight", "cosine", "optcosine")
p <- plot_ly()
for (k in kerns) {
  d <- density(economics$pce, kernel = k, na.rm = TRUE)
  p <- add_lines(p, x = d$x, y = d$y, name = k)
}
p
```

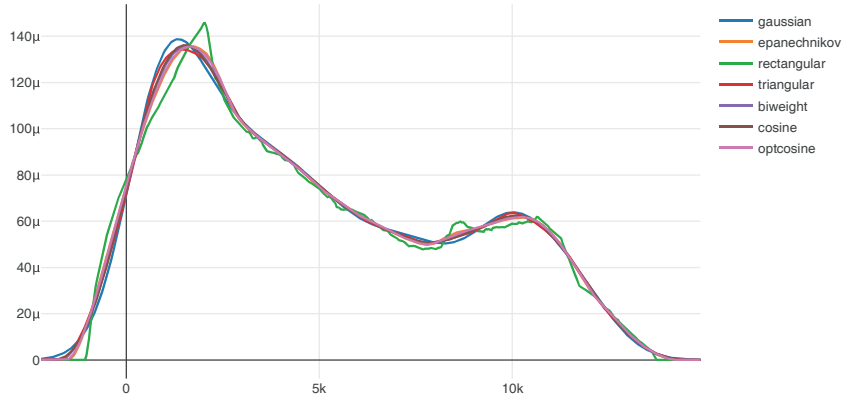

FIGURE 3.22: Various kernel density estimates.

3.2.4 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 (Cleveland and McGill, 1984). Usually each line represents every measurement for a given row (or observation) in a dataset. It's true that plotly.js provides a trace type, parcoords, specifically for parallel coordinates that offer desirable interactive capabilities (e.g., highlighting and reordering of axes).[17] However, it can also be useful to learn how to use `add_lines()` to implement parallel coordinates, as it can offer more flexibility and control over the axis scales.

When measurements are on very different scales, some care must be taken, and variables must be transformed to be put on a common scale. As Figure 3.23 shows, even when variables are measured on a similar scale, it can still be informative to transform variables in different ways.

[17] See `https://plot.ly/r/parallel-coordinates-plot/` for some interactive examples.

```
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),
  nrows = 3, shareX = TRUE
) %>% hide_legend()
```

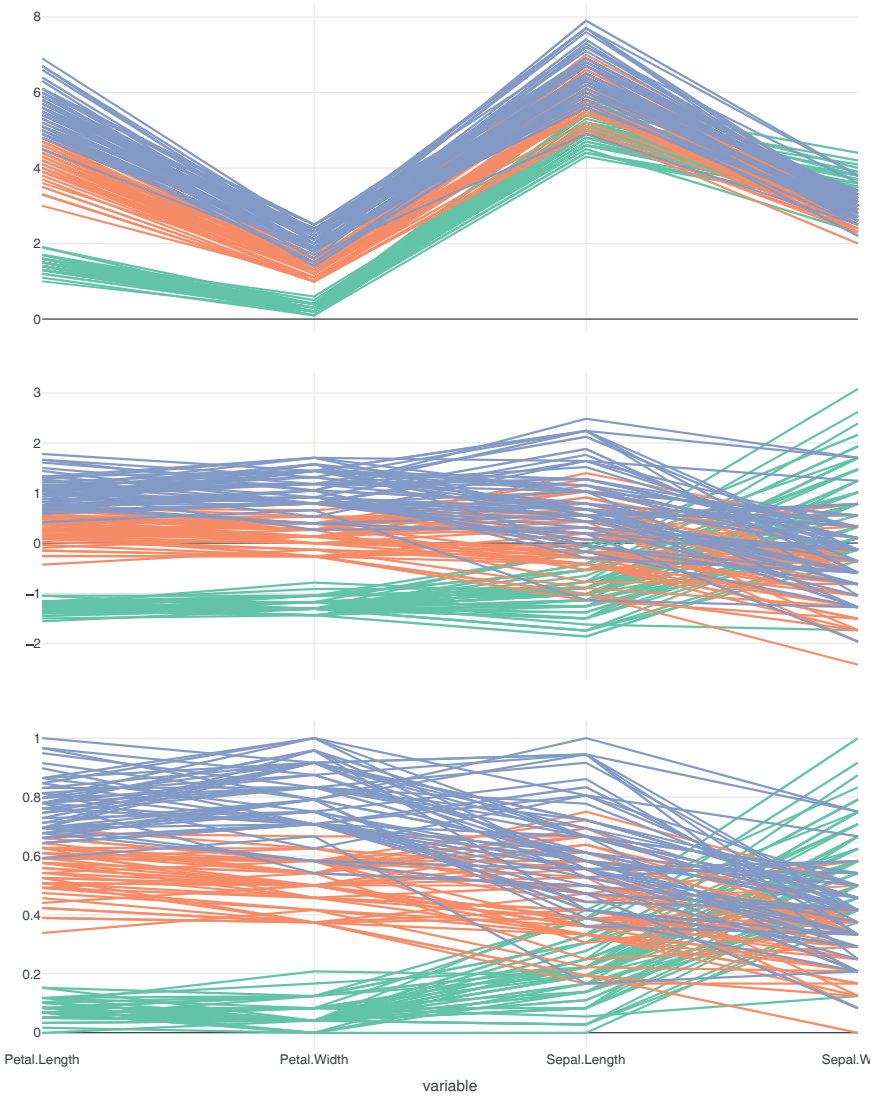

FIGURE 3.23: Parallel coordinate plots of the Iris dataset. The top panel shows all variables on a common scale. The middle panel scales each variable to have mean of 0 and standard deviation of 1. In the bottom panel, each variable is scaled to have a minimum of 0 and a maximum of 1.

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()`. Thanks to the linked highlighting framework, parallel coordinates created in this way could be linked to lower dimensional (but sometimes higher resolution) graphics of related data to guide multi-variate data exploration. The **pedestrians** package provides some examples of linking parallel coordinates to other views such as a grand tour for exposing unusual features in a high-dimensional space (Sievert, 2019a).

3.3 Polygons

The `add_polygons()` function is essentially equivalent to `add_paths()` with the fill[18] attribute set to "toself". Polygons form the basis for other, higher-level scatter-based layers (e.g., `add_ribbons()` and `add_sf()`) that don't have a dedicated plotly.js trace type. Polygons can be used to draw many things, but perhaps the most familiar application where you *might* want to use `add_polygons()` is to draw geo-spatial objects. If and when you use `add_polygons()` to draw a map, make sure you fix the aspect ratio (e.g., `xaxis.scaleanchor`[19]) and also consider using `plotly_empty()` over `plot_ly()` to hide axis labels, ticks, and the background grid. On the other hand, Section 4.2 shows you how to make custom maps using the **sf** package and `add_sf()`, which is a bit of work to get started, but is absolutely worth the investment.

```
base <- map_data("world", "canada") %>%
  group_by(group) %>%
  plotly_empty(x = ~long, y = ~lat, alpha = 0.2) %>%
  layout(showlegend = FALSE, xaxis = list(scaleanchor = "y"))

base %>%
  add_polygons(hoverinfo = "none", color = I("black")) %>%
```

[18] https://plot.ly/r/reference/#scatter-fill
[19] https://plot.ly/r/reference/#layout-xaxis-scaleanchor

```
  add_markers(text = ~paste(name, "<br />", pop), hoverinfo = "text",
              color = I("red"), data = maps::canada.cities)
```

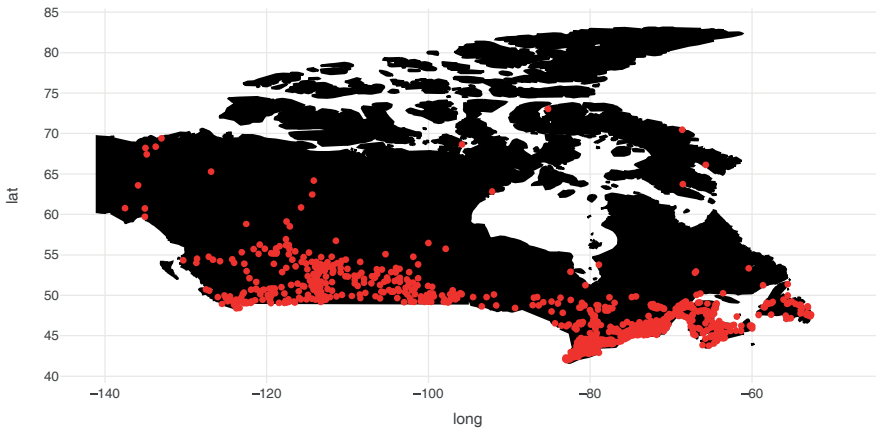

FIGURE 3.24: Using `add_polygons()` to make a map of Canada and major Canadian cities via data provided by the **maps** package.

As discussion surrounding Figure 4.10 points out, scatter-based polygon layers (i.e., `add_polygons()`, `add_ribbons()`, etc.) render all the polygons using one plotly.js trace by default. This approach is computationally efficient, but it's not always desirable (e.g., can't have multiple fills per trace, interactivity is relatively limited). To work around the limitations, consider using `split` (or `color` with a discrete variable) to split the polygon data into multiple traces. Figure 3.25 demonstrates using `split` which will impose plotly.js's colorway to each trace (i.e., subregion) and leverage `hoveron` to generate one tooltip per sub-region.

```
add_polygons(base, split = ~subregion, hoveron = "fills")
```

FIGURE 3.25: Using `split` to render polygons with different fills and interactive properties.

3.3.1 Ribbons

Ribbons are useful for showing uncertainty bounds as a function of x. The `add_ribbons()` function creates ribbons and requires the arguments: `x`, `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 convenient form for visualization. Figure 3.26 shows the fitted values and uncertainty bounds from a linear model object.

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

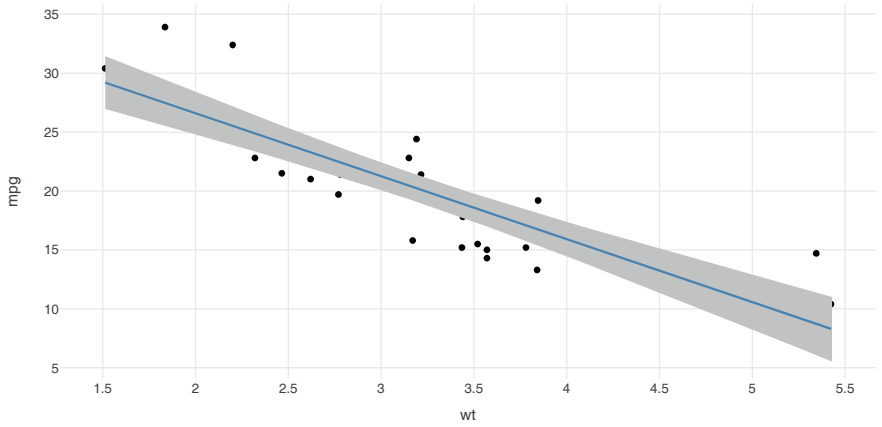

FIGURE 3.26: Plotting fitted values and uncertainty bounds of a linear model via the **broom** package.

4

Maps

There are numerous ways to make a map with **plotly**; each with its own strengths and weaknesses. Generally speaking, the approaches fall under two categories: integrated or custom. Integrated maps leverage plotly.js's built-in support for rendering a basemap layer. Currently there are two supported ways of making integrated maps: either via Mapbox[1] or via an integrated d3.js powered basemap. The integrated approach is convenient if you need a quick map and don't necessarily need sophisticated representations of geo-spatial objects. On the other hand, the custom mapping approach offers complete control since you're providing all the information necessary to render the geo-spatial object(s). Section 4.2 covers making sophisticated maps (e.g., cartograms) using the **sf** R package, but it's also possible to make custom **plotly** maps via other tools for geo-computing (e.g., **sp**, **ggmap**, etc.).

It's worth noting that **plotly** aims to be a general purpose visualization library, and thus, doesn't aim to be the most fully featured geo-spatial visualization toolkit. That said, there are benefits to using **plotly**-based maps since the mapping APIs are very similar to the rest of plotly, and you can leverage the larger **plotly** ecosystem (e.g., linking views client-side like Figure 16.23). However, if you run into limitations with **plotly**'s mapping functionality, there is a very rich set of tools for interactive geospatial visualization in R[2], including but not limited to: **leaflet**, **mapview**, **mapedit**, **tmap**, and **mapdeck** (Lovelace et al., 2019).

[1] https://www.mapbox.com/

[2] https://geocompr.robinlovelace.net/adv-map.html#interactive-maps

4.1 Integrated maps

4.1.1 Overview

If you have fairly simple latitude/longitude data and want to make a quick map, you may want to try one of **plotly**'s integrated mapping options (i.e., `plot_mapbox()` and `plot_geo()`). Generally speaking, you can treat these constructor functions as a drop-in replacement for `plot_ly()` and get a dynamic basemap rendered behind your data. Furthermore, all the scatter-based layers we learned about in Section 3 work as you'd expect it to with `plot_ly()`.[3] For example, Figure 4.1 uses `plot_mapbox()` and `add_markers()` to create a bubble chart:

```
plot_mapbox(maps::canada.cities) %>%
  add_markers(
    x = ~long,
    y = ~lat,
    size = ~pop,
    color = ~country.etc,
    colors = "Accent",
    text = ~paste(name, pop),
    hoverinfo = "text"
  )
```

[3] Unfortunately, non-scatter traces currently don't work with `plot_mapbox()`/`plot_geo()` meaning that, for one, raster (i.e., heatmap) maps are not natively supported.

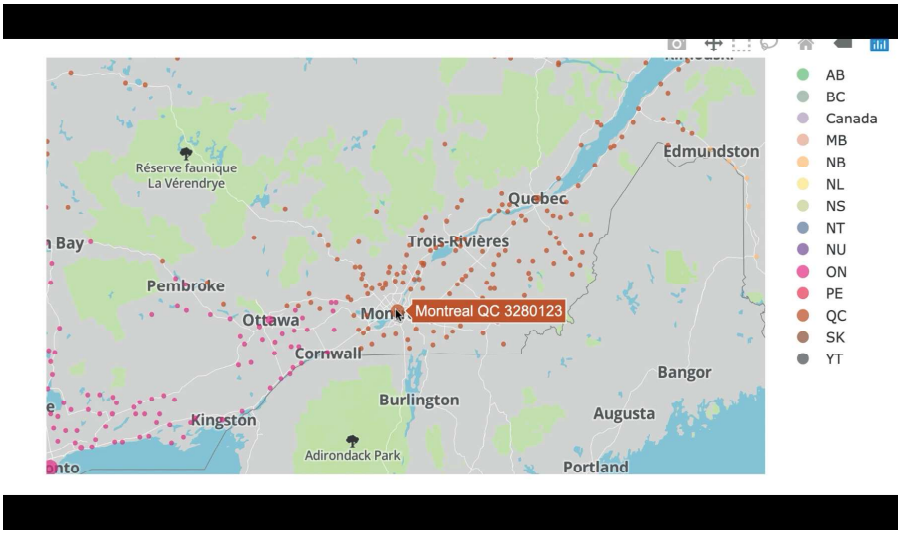

FIGURE 4.1: A mapbox powered bubble chart showing the population of various cities in Canada. For a video demonstration of the interactive, see `https://bit.ly/mapbox-bubble`. For the interactive, see `https://plotly-r.com/interactives/mapbox-bubble.html`

The Mapbox basemap styling is controlled through the `layout.mapbox.style`[4] attribute. The **plotly** package comes with support for 7 different styles, but you can also supply a custom URL to a custom mapbox style[5]. To obtain all the pre-packaged basemap style names, you can grab them from the official plotly.js `schema()`:

```
styles <- schema()$layout$layoutAttributes$mapbox$style$values
styles
#>  [1] "basic"             "streets"
#>  [3] "outdoors"          "light"
#>  [5] "dark"              "satellite"
#>  [7] "satellite-streets" "open-street-map"
#>  [9] "white-bg"          "carto-positron"
```

[4] `https://plot.ly/r/reference/#layout-mapbox-style`
[5] `https://docs.mapbox.com/help/tutorials/create-a-custom-style/`

```
#> [11] "carto-darkmatter"  "stamen-terrain"
#> [13] "stamen-toner"      "stamen-watercolor"
```

Any one of these values can be used for a mapbox style. Figure 4.2 demonstrates the satellite earth imagery basemap.

```
layout(
  plot_mapbox(),
  mapbox = list(style = "satellite")
)
```

FIGURE 4.2: Zooming in on earth satellite imagery using `plot_mapbox()`. For a video demonstration of the interactive, see `https://bit.ly/mapbox-satellite`. For the interactive, see `https://plotly-r.com/interactives/satellite.html`

Figure 4.3 demonstrates how to create an integrated plotly.js dropdown menu to control the basemap style via the `layout.updatemenus`[6] attribute. The idea behind an integrated plotly.js dropdown is to supply a list of buttons (i.e., menu items) where each button invokes a plotly.js method

[6] `https://plot.ly/r/reference/#layout-updatemenus-items-updatemenu-buttons`

with some arguments. In this case, each button uses the relayout[7] method to modify the `layout.mapbox.style` attribute.[8]

```
style_buttons <- lapply(styles, function(s) {
  list(
    label = s,
    method = "relayout",
    args = list("mapbox.style", s)
  )
})
layout(
  plot_mapbox(),
  mapbox = list(style = "dark"),
  updatemenus = list(
    list(y = 0.8, buttons = style_buttons)
  )
)
```

[7] https://plot.ly/javascript/plotlyjs-function-reference/

[8] To see more examples of creating and using plotly.js's integrated dropdown functionality to modify graphs, see https://plot.ly/r/dropdowns/

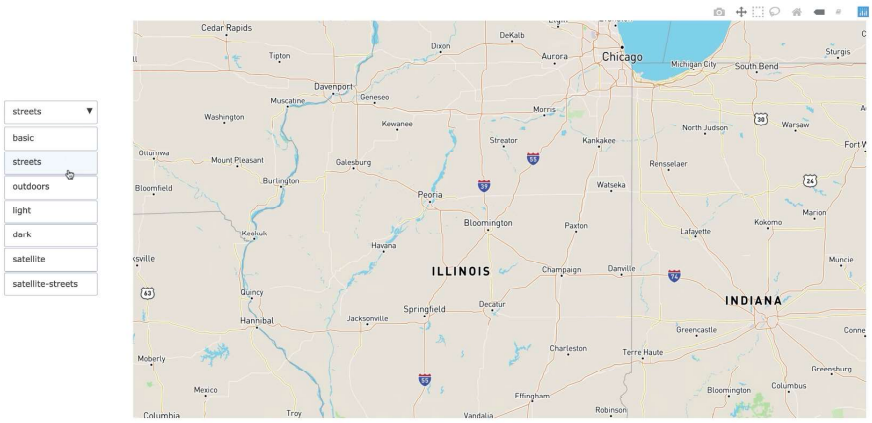

FIGURE 4.3: Providing a dropdown menu to control the styling of the mapbox baselayer. For a video demonstration of the interactive, see `https://bit.ly/mapbox-style-dropdown`. For the interactive, see `https://plotly-r.com/interactives/mapbox-style-dropdown.html`

The other integrated mapping solution in **plotly** is `plot_geo()`. Compared to `plot_mapbox()`, this approach has support for different mapping projections, but styling the basemap is limited and can be more cumbersome. Figure 4.4 demonstrates using `plot_geo()` in conjunction with `add_markers()` and `add_segments()` to visualize flight paths within the United States. Whereas `plot_mapbox()` is fixed to a mercator projection, the `plot_geo()` constructor has a handful of different projections available to it, including the orthographic projection which gives the illusion of the 3D globe.

```
library(plotly)
library(dplyr)
# airport locations
air <- read.csv(
  'https://plotly-r.com/data-raw/airport_locations.csv'
)
# flights between airports
flights <- read.csv(
  'https://plotly-r.com/data-raw/flight_paths.csv'
```

```
)
flights$id <- seq_len(nrow(flights))

# map projection
geo <- list(
  projection = list(
    type = 'orthographic',
    rotation = list(lon = -100, lat = 40, roll = 0)
  ),
  showland = TRUE,
  landcolor = toRGB("gray95"),
  countrycolor = toRGB("gray80")
)

plot_geo(color = I("red")) %>%
  add_markers(
    data = air, x = ~long, y = ~lat, text = ~airport,
    size = ~cnt, hoverinfo = "text", alpha = 0.5
  ) %>%
  add_segments(
    data = group_by(flights, id),
    x = ~start_lon, xend = ~end_lon,
    y = ~start_lat, yend = ~end_lat,
    alpha = 0.3, size = I(1), hoverinfo = "none"
  ) %>%
  layout(geo = geo, showlegend = FALSE)
```

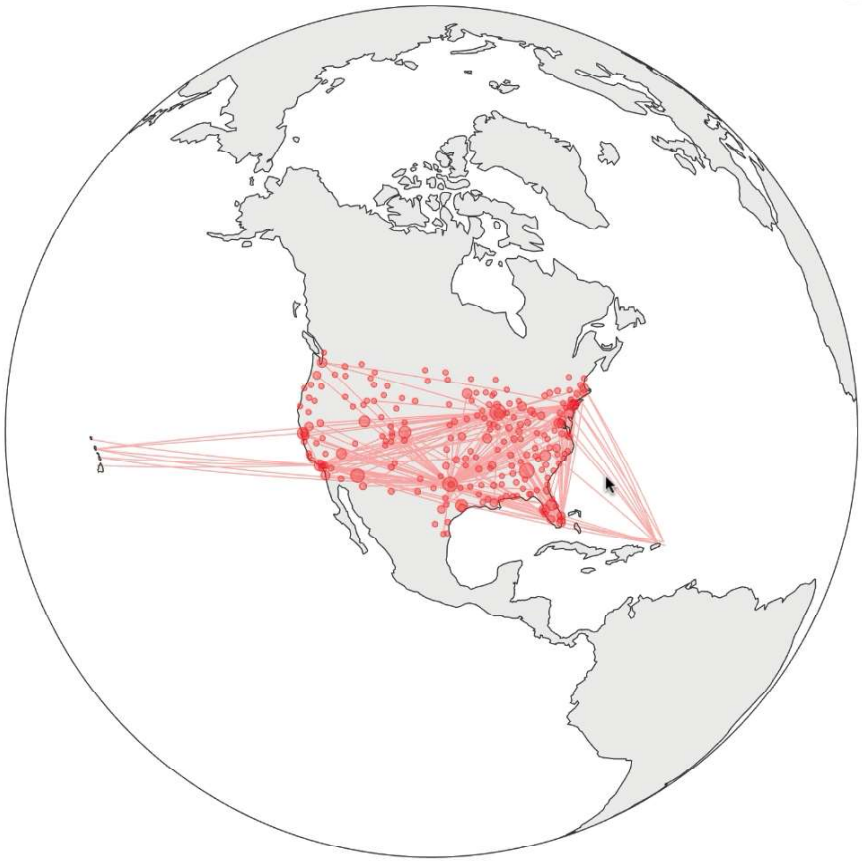

FIGURE 4.4: Using the integrated orthographic projection to visualize flight patterns on a '3D' globe. For a video demonstration of the interactive, see `https://bit.ly/geo-flights`. For the interactive, see `https://plotly-r.com/interactives/geo-flights.html`

One nice thing about `plot_geo()` is that it automatically projects geometries into the proper coordinate system defined by the map projection. For example, in Figure 4.5 the simple line segment is straight when using `plot_mapbox()`, yet curved when using `plot_geo()`. It's possible to achieve the same effect using `plot_ly()` or `plot_mapbox()`, but the relevant marker/line/polygon data has to be put into an **sf** data structure before rendering (see Section 4.2.1 for more details).

```
map1 <- plot_mapbox() %>%
  add_segments(x = -100, xend = -50, y = 50, yend = 75) %>%
  layout(
    mapbox = list(
      zoom = 0,
      center = list(lat = 65, lon = -75)
    )
  )

map2 <- plot_geo() %>%
  add_segments(x = -100, xend = -50, y = 50, yend = 75) %>%
  layout(geo = list(projection = list(type = "mercator")))

library(htmltools)
browsable(tagList(map1, map2))
```

FIGURE 4.5: A comparison of **plotly**'s integrated mapping solutions: `plot_mapbox()` (top) and `plot_geo()` (bottom). The `plot_geo()` approach will transform line segments to correctly reflect their projection into a non-Cartesian coordinate system.

4.1.2 Choropleths

In addition to scatter traces, both of the integrated mapping solutions (i.e., `plot_mapbox()` and `plot_geo()`) have an optimized choropleth trace type (i.e., the choroplethmapbox[9] and choropleth[10] trace types). Comparatively speaking, choroplethmapbox is more powerful because you can fully specify the feature collection using GeoJSON, but the choropleth trace can be a bit easier to use if it fits your use case.

Figure 4.6 shows the population density of the U.S. via the choropleth trace using the U.S. state data from the **datasets** package (R Core Team, 2016). By simply providing a `z`[11] attribute, `plotly_geo()` objects will try to create a choropleth, but you'll also need to provide `locations`[12] and a `locationmode`[13]. It's worth noting that the `locationmode` is currently limited to countries and US states, so if you need to plot a different geo-unit (e.g., counties, municipalities, etc.), you should use the choroplethmapbox trace type and/or use a "custom" mapping approach as discussed in Section 4.2.

```
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, span = I(0),
    locations = state.abb, locationmode = 'USA-states'
  ) %>%
  layout(geo = g)
```

[9] https://plot.ly/r/reference/#choroplethmapbox
[10] https://plot.ly/r/reference/#choropleth
[11] https://plot.ly/r/reference/#choropleth-z
[12] https://plot.ly/r/reference/#choropleth-locations
[13] https://plot.ly/r/reference/#choropleth-locationmode

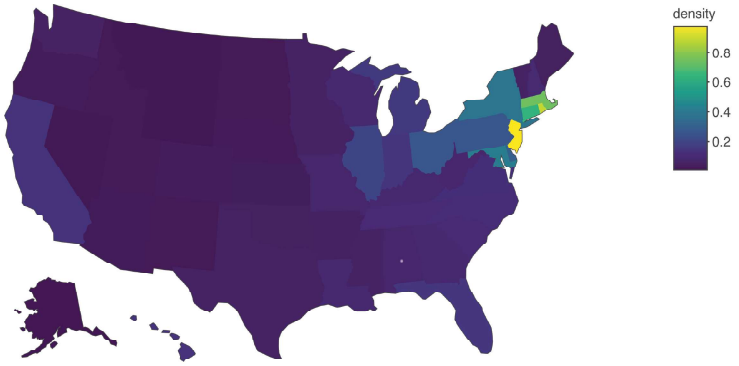

FIGURE 4.6: A map of U.S. population density using the `state.x77` data from the **datasets** package.

Choroplethmapbox is more flexible than choropleth because you supply your own GeoJSON definition of the choropleth via the `geojson` attribute. Currently this attribute must be a URL pointing to a geojson file. Moreover, the `location` should point to a top-level id attribute of each feature within the geojson file. Figure 4.7 demonstrates how we could visualize the same information as Figure 4.6, but this time using choroplethmapbox.

```
plot_ly() %>%
  add_trace(
    type = "choroplethmapbox",
    # See how this GeoJSON URL was generated at
    # https://plotly-r.com/data-raw/us-states.R
    geojson = paste(c(
      "https://gist.githubusercontent.com/cpsievert/",
      "7cdcb444fb2670bd2767d349379ae886/raw/",
      "cf5631bfd2e385891bb0a9788a179d7f023bf6c8/",
      "us-states.json"
    ), collapse = ""),
    locations = row.names(state.x77),
    z = state.x77[, "Population"] / state.x77[, "Area"],
```

```
    span = I(0)
  ) %>%
  layout(
    mapbox = list(
      style = "light",
      zoom = 4,
      center = list(lon = -98.58, lat = 39.82)
    )
  ) %>%
  config(
    mapboxAccessToken = Sys.getenv("MAPBOX_TOKEN"),
    # Workaround to make sure image download uses full container
    # size https://github.com/plotly/plotly.js/pull/3746
    toImageButtonOptions = list(
      format = "svg",
      width = NULL,
      height = NULL
    )
  )
```

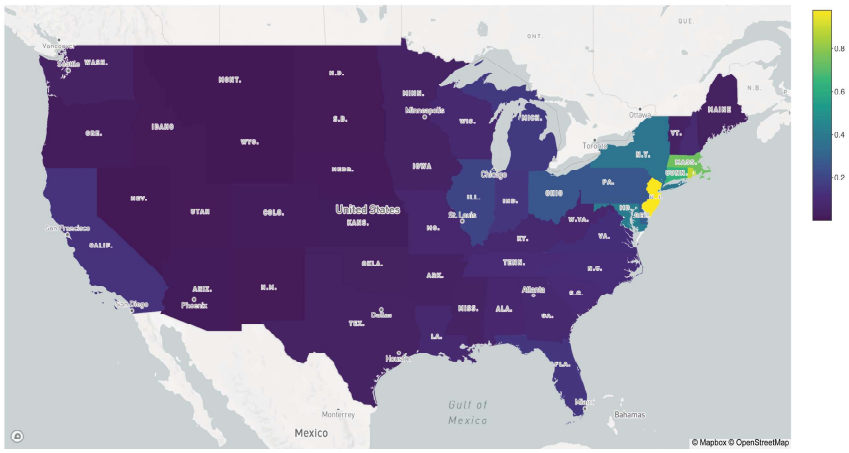

FIGURE 4.7: Another map of U.S. population density, this time using choroplethmapbox with a custom GeoJSON file.

Figures 4.6 and 4.7 aren't an ideal way to visualize state population a graphical perception point of view. We typically use the color in choropleths to encode a numeric variable (e.g., GDP, net exports, average SAT score, etc.) and the eye naturally perceives the area that a particular color covers as proportional to its overall effect. This ends up being misleading since the area the color covers typically has no sensible relationship with the data encoded by the color. A classic example of this misleading effect in action is in US election maps – the proportion of red to blue coloring is not representative of the overall popular vote (Newman, 2016).

Cartograms are an approach to reducing this misleading effect and grant another dimension to encode data through the size of geo-spatial features. Section 4.2.2 covers how to render cartograms in **plotly** using **sf** and **cartogram**.

4.2 Custom maps

4.2.1 Simple features (sf)

The **sf** R package is a modern approach to working with geo-spatial data structures based on tidy data principles (Pebesma, 2018; Wickham, 2014b). The key idea behind **sf** is that it stores geo-spatial geometries in a list-column[14] of a data frame. This allows each row to represent the real unit of observation/interest — whether it's a polygon, multi-polygon, point, line, or even a collection of these features — and as a result, works seamlessly inside larger tidy workflows.[15] The **sf** package itself does not really provide geo-spatial data; it provides the framework and utilities for storing and computing on geo-spatial data structures in an opinionated way.

[14] `https://jennybc.github.io/purrr-tutorial/ls13_list-columns.html`

[15] This is way more intuitive compared to older workflows based on, say using `ggplot2::fortify()` to obtain a data structure where a row represents a particular point along a feature and having another column track which point belongs to each feature.

There are numerous packages for accessing geo-spatial data as simple features data structures. A couple of notable examples include **rnaturalearth** and **USAboundaries**. The **rnaturalearth** package is better for obtaining any map data in the world via an API provided by `https://www.naturalearthdata.com/` (South, 2017). The **USAboundaries** package is great for obtaining map data for the United States at any point in history (Mullen and Bratt, 2018). It doesn't really matter what tool you use to obtain or create an **sf** object; once you have one, `plot_ly()` knows how to render it:

```
library(rnaturalearth)
world <- ne_countries(returnclass = "sf")
class(world)
#> [1] "sf"    "data.frame"
plot_ly(world, color = I("gray90"), stroke = I("black"), span = I(1))
```

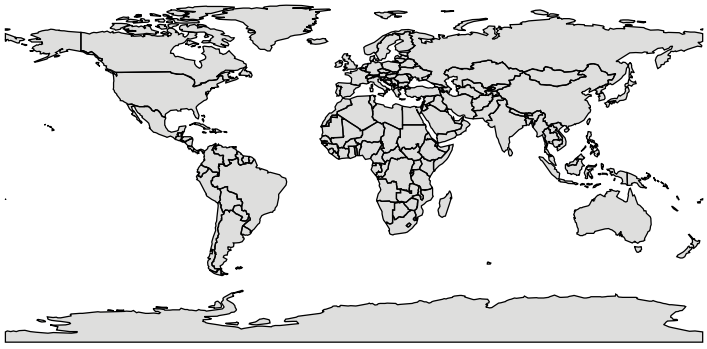

FIGURE 4.8: Rendering all the world's countries using `plot_ly()` and the `ne_countries()` function from the **rnaturalearth** package.

How does `plot_ly()` know how to render the countries? It's because the geo-spatial features are encoded in special (geometry) list-column. Also, meta-data about the geo-spatial structure are retained as special attributes of the data. Figure 4.9 augments the print method for **sf** to data frames to demonstrate that all the information needed to render

the countries (i.e., polygons) in Figure 4.8 is contained within the `world` data frame. Note also that **sf** provides special **dplyr** methods for this special class of data frame so that you can treat data manipulation as if it were a 'tidy' data structure. One thing about this method is that the special 'geometry' column is always retained; if we try to just select the `name` column, then we get both the name and the geometry.

```
library(sf)
world %>%
  select(name) %>%
  print(n = 4)
```

```
Simple feature collection with 177 features and 1 field
geometry type:  MULTIPOLYGON
dimension:      XY
bbox:           xmin: -180 ymin: -90 xmax: 180 ymax: 83.64513
epsg (SRID):    4326
proj4string:    +proj=longlat +datum=WGS84 +no_defs
First 4 features:
                 name                       geometry
0         Afghanistan MULTIPOLYGON (((61.21082 35 ...
1              Angola MULTIPOLYGON (((16.32653 -5 ...
2             Albania MULTIPOLYGON (((20.59025 41 ...
3 United Arab Emirates MULTIPOLYGON (((51.57952 24 ...
```

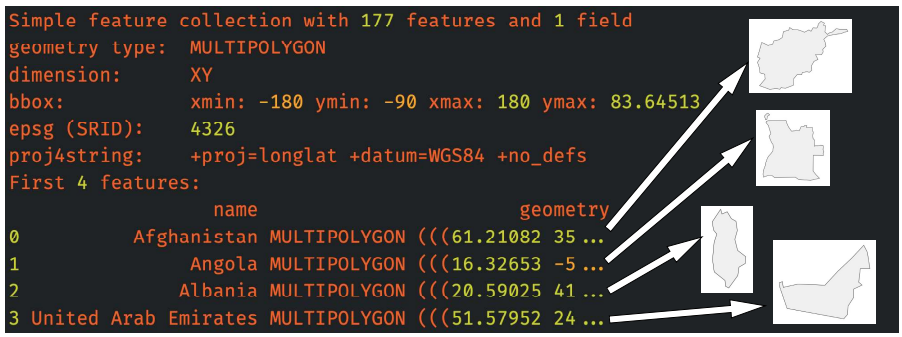

FIGURE 4.9: A diagram of a simple features data frame. The geometry column tracks the spatial features attached to each row in the data frame.

There are actually 4 different ways to render **sf** objects with **plotly**: `plot_ly()`, `plot_mapbox()`, `plot_geo()`, and via **ggplot2**'s `geom_sf()`. These functions render multiple polygons using a *single* trace by default, which is fast, but you may want to leverage the added flexibility of multiple traces. For example, a given trace can only have one `fillcolor`, so it's impossible to render multiple polygons with different colors using a single trace. For this reason, if you want to vary the color of multiple polygons, make sure the `split` by a unique identifier (e.g., `name`), as done in Figure 4.10. Note that, as discussed for line charts in Figure 3.2, using multiple traces automatically adds the ability to filter `name` via legend entries.

```
canada <- ne_states(country = "Canada", returnclass = "sf")
plot_ly(canada, split = ~name, color = ~provnum_ne)
```

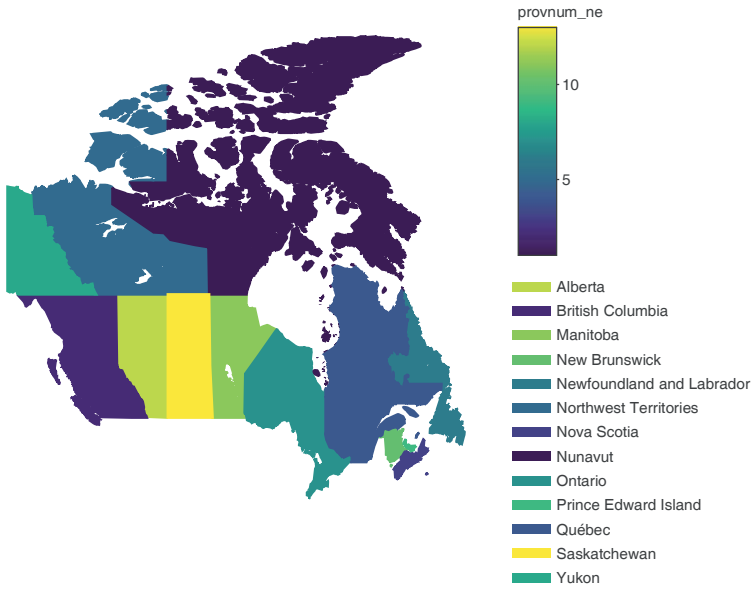

FIGURE 4.10: Using `split` and `color` to create a choropleth map of provinces in Canada.

Another important feature for maps that may require you to `split` multiple polygons into multiple traces is the ability to display a different hover-on-fill for each polygon. By providing `text` that is unique within each polygon and specifying `hoveron='fills'`, as in Figure 4.11, the tooltip behavior is tied to the trace's fill (instead of being displayed at each point along the polygon).

```
plot_ly(
  canada,
  split = ~name,
  color = I("gray90"),
  text = ~paste(name, "is \n province number", provnum_ne),
```

```
  hoveron = "fills",
  hoverinfo = "text",
  showlegend = FALSE
)
```

FIGURE 4.11: Using `split`, `text`, and `hoveron='fills'` to display a tooltip specific to each Canadian province.

Although the integrated mapping approaches (`plot_mapbox()` and `plot_geo()`) can render **sf** objects, the custom mapping approaches (`plot_ly()` and `geom_sf()`) are more flexible because they allow for any well-defined mapping projection. Working with and understanding map projections can be intimidating for a causal map maker. Thankfully, there are nice resources for searching map projections in a human-friendly interface, like `http://spatialreference.org/`. Through this website, one can search desirable projections for a given portion of the globe and extract commands for projecting their geo-spatial objects into that projection. As shown in Figure 4.12, one way to per-

form the projection is to supply the relevant PROJ4 command to the `st_transform()` function in **sf** (PROJ contributors, 2018).

```
# filter the world sf object down to canada
canada <- filter(world, name == "Canada")
# coerce cities lat/long data to an official sf object
cities <- st_as_sf(
  maps::canada.cities,
  coords = c("long", "lat"),
  crs = 4326
)

# A PROJ4 projection designed for Canada
# http://spatialreference.org/ref/sr-org/7/
# http://spatialreference.org/ref/sr-org/7/proj4/
moll_proj <- "+proj=moll +lon_0=0 +x_0=0 +y_0=0 +ellps=WGS84
+units=m +no_defs"

# perform the projections
canada <- st_transform(canada, moll_proj)
cities <- st_transform(cities, moll_proj)

# plot with geom_sf()
p <- ggplot() +
  geom_sf(data = canada) +
  geom_sf(data = cities, aes(size = pop), color = "red", alpha = 0.3)
ggplotly(p)
```

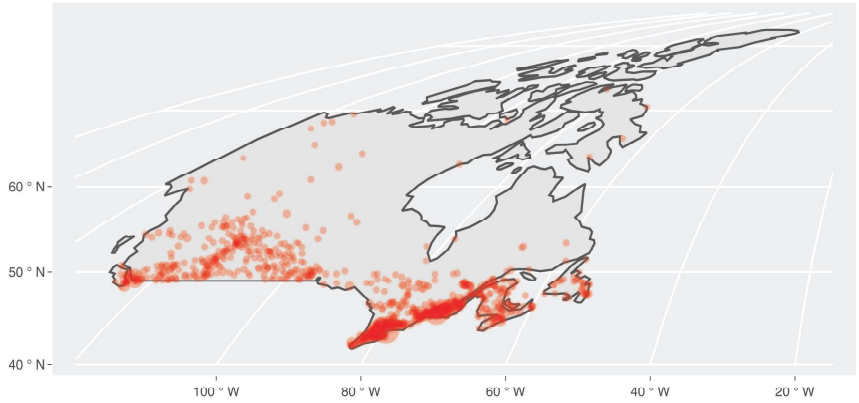

FIGURE 4.12: The population of various Canadian cities rendered on a custom basemap using a Mollweide projection.

Some geo-spatial objects have an unnecessarily high resolution for a given visualization. In these cases, you may want to consider simplifying the geo-spatial object to improve the speed of the R code and responsiveness of the visualization. For example, we could recreate Figure 4.8 with a much higher resolution by specifying `scale = "large"` in `ne_countries()`; this gives us a **sf** object with over 50 times more spatial coordinates than the default scale. The higher resolution allows us to zoom in better on more complex geo-spatial regions, but it allow leads to slower R code, larger HTML files, and slower responsiveness. Sievert (2018b) explores this issue in more depth and demonstrates how to use the `st_simplify()` function from **sf** to simplify features before plotting them.

```
sum(rapply(world$geometry, nrow))
#> [1] 10586

world_large <- ne_countries(scale = "large", returnclass = "sf")
sum(rapply(world_large$geometry, nrow))
#> [1] 548121
```

Analogous to the discussion surrounding 3.2, it pays to be aware of the tradeoffs involved with rendering **plotly** graphics using one or many traces, and to be knowledgeable about how to leverage either approach. Specifically, by default, **plotly** attempts to render all simple features in a single trace, which is performant, but doesn't have a lot of interactivity.

```
plot_mapbox(
  world_large,
  color = NA,
  stroke = I("black"),
  span = I(0.5)
)
```

For those interested in learning more about geocomputation in R with **sf** and other great R packages like **sp** and **raster**, Lovelace et al. (2019) provide lots of nice and freely available learning resources (Pebesma and Bivand, 2005; Hijmans, 2019).

4.2.2 Cartograms

Cartograms distort the size of geo-spatial polygons to encode a numeric variable other than the land size. There are numerous types of cartograms and they are typically categorized by their ability to preserve shape and maintain contiguous regions. Cartograms have been shown to be an effective approach to both encode and teach about geo-spatial data, though the effects certainly vary by cartogram type (Nusrat et al., 2016). The R package **cartogram** provides an interface to several popular cartogram algorithms (Jeworutzki, 2018). A number of other R packages provide cartogram algorithms, but the great thing about **cartogram** is that all the functions can take an **sf** (or **sp**) object as input and return an **sf** object. This makes it incredibly easy to go from raw spatial objects, to transformed objects, to visual. Figure 4.13 demonstrates a continuous area cartogram of US population in 2014 using a rubber sheet distortion algorithm from Dougenik et al. (1985).

```
library(cartogram)
library(albersusa)

us_cont <- cartogram_cont(usa_sf("laea"), "pop_2014")

plot_ly(us_cont) %>%
  add_sf(
    color = ~pop_2014,
    split = ~name,
    span = I(1),
    text = ~paste(name, scales::number_si(pop_2014)),
    hoverinfo = "text",
    hoveron = "fills"
  ) %>%
  layout(showlegend = FALSE) %>%
  colorbar(title = "Population \n 2014")
```

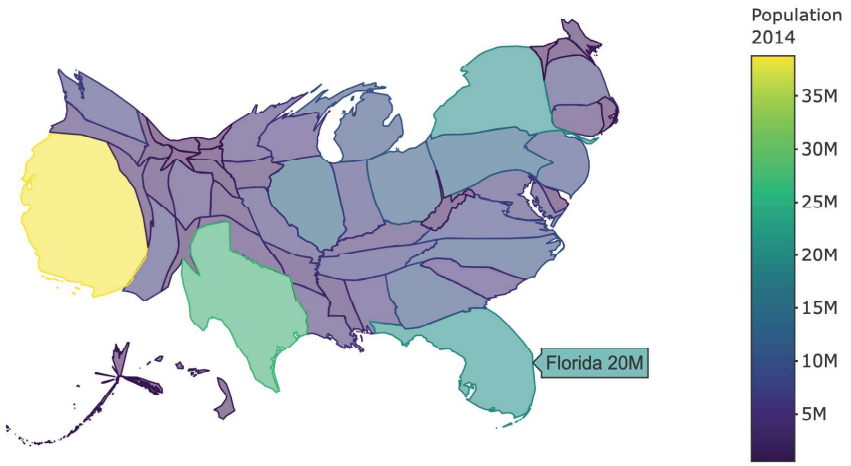

FIGURE 4.13: A cartogram of the U.S. population in 2014. A cartogram sizes the area of geo-spatial objects proportional to some metric (e.g., population).

Figure 4.14 demonstrates a non-continuous Dorling cartogram of US population in 2014 from Dorling D. (1996). This cartogram does not try to preserve the shape of polygons (i.e., states), but instead uses circles to represent each geo-spatial object, then encodes the variable of interest (i.e., population) using the area of the circle.

```
us <- usa_sf("laea")
us_dor <- cartogram_dorling(us, "pop_2014")

plot_ly(stroke = I("black"), span = I(1)) %>%
  add_sf(
    data = us,
    color = I("gray95"),
    hoverinfo = "none"
  ) %>%
  add_sf(
    data = us_dor,
    color = ~pop_2014,
    split = ~name,
    text = ~paste(name, scales::number_si(pop_2014)),
    hoverinfo = "text",
    hoveron = "fills"
  ) %>%
  layout(showlegend = FALSE)
```

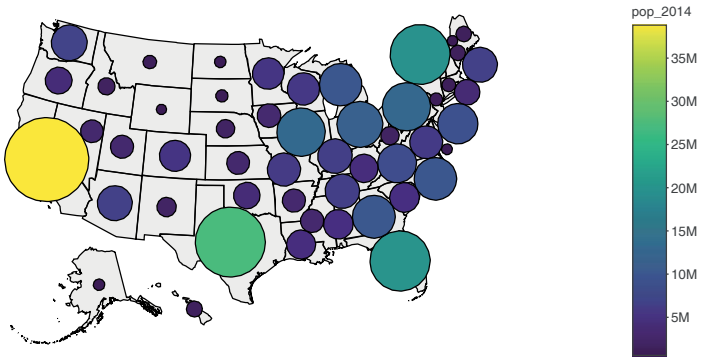

FIGURE 4.14: A Dorling cartogram of the U.S. population in 2014. A Dorling cartogram sizes the circles proportional to some metric (e.g., population).

Figure 4.15 demonstrates a non-continuous cartogram of the U.S. population in 2014 from Olson (1976). In contrast to the Dorling cartogram, this approach does preserve the shape of polygons. The implementation behind Figure 4.15 is to simply take the implementation of Figure 4.14 and change `cartogram_dorling()` to `cartogram_ncont()`.

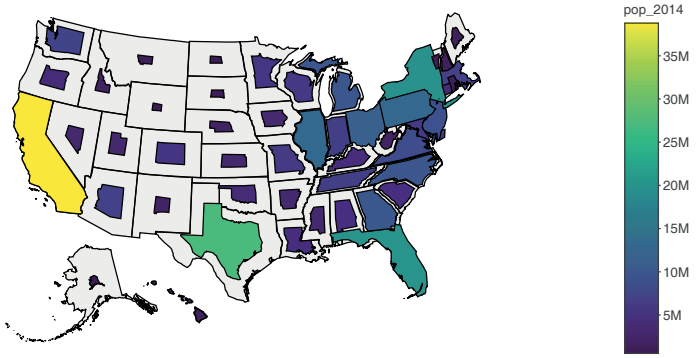

FIGURE 4.15: A non-contiguous cartogram of the U.S. population in 2014 that preserves shape.

A popular class of contiguous cartograms that do not preserve shape are sometimes referred to as tile cartograms (aka tilegrams). At the time of writing, there doesn't seem to be a great R package for *computing* tilegrams, but Pitch Interactive provides a nice web service where you can generate tilegrams from existing or custom data `https://pitchinteractiveinc.github.io/tilegrams/`. Moreover, the service allows you to download a TopoJSON file of the generated tilegram, which we can read in R and convert into an **sf** object via **geojsonio** (Chamberlain and Teucher, 2018). Figure 4.16 demonstrates a tilegram of U.S. Population in 2016 exported directly from Pitch's free web service.

```
library(geojsonio)
tiles <- geojson_read("~/Downloads/tiles.topo.json", what = "sp")
tiles_sf <- st_as_sf(tiles)
plot_ly(
  tiles_sf, color = ~name,
  colors = "Paired", stroke = I("transparent")
)
```

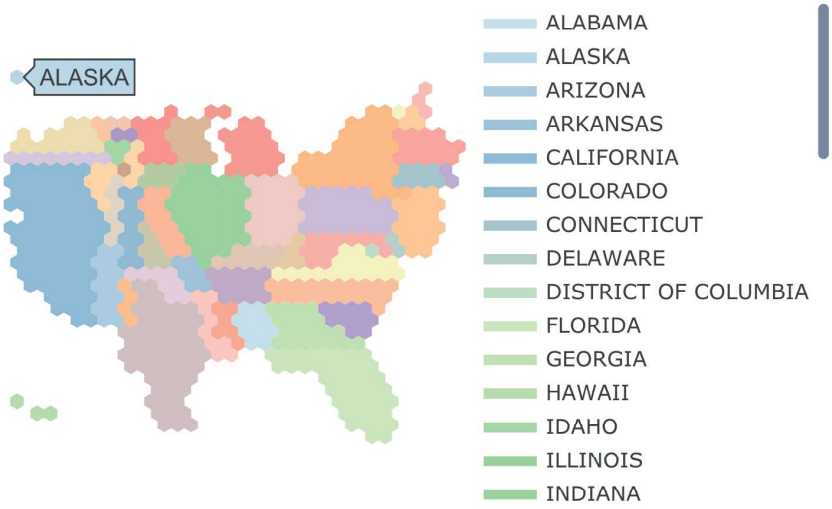

FIGURE 4.16: A tile cartogram of the U.S. population in 2016.


Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

5

Bars and histograms

The `add_bars()` and `add_histogram()` functions wrap the bar[1] and histogram[2] 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.[3] And perhaps confusingly, both 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 5.1 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`[4]/`ybins`[5], R has diverse facilities for estimating the optimal number of bins in a histogram that we can easily leverage.[6] The `hist()` function alone allows us to reference 3 famous algorithms by name (Sturges, 1926; Freedman and Diaconis, 1981; Scott, 1979), but there are also packages (e.g., the **histogram** package) which extend this interface to incorporate more methodology (Mildenberger et al., 2009). 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()`.

[1] `https://plot.ly/r/reference/#bar`

[2] `https://plot.ly/r/reference/#histogram`

[3] As we'll see in Section 16.1, and specifically Figure 16.6, using a 'statistical' trace type like `add_histogram()` enables statistical graphical queries.

[4] `https://plot.ly/r/reference/#histogram-xbins`

[5] `https://plot.ly/r/reference/#histogram-ybins`

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

```
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
)
```

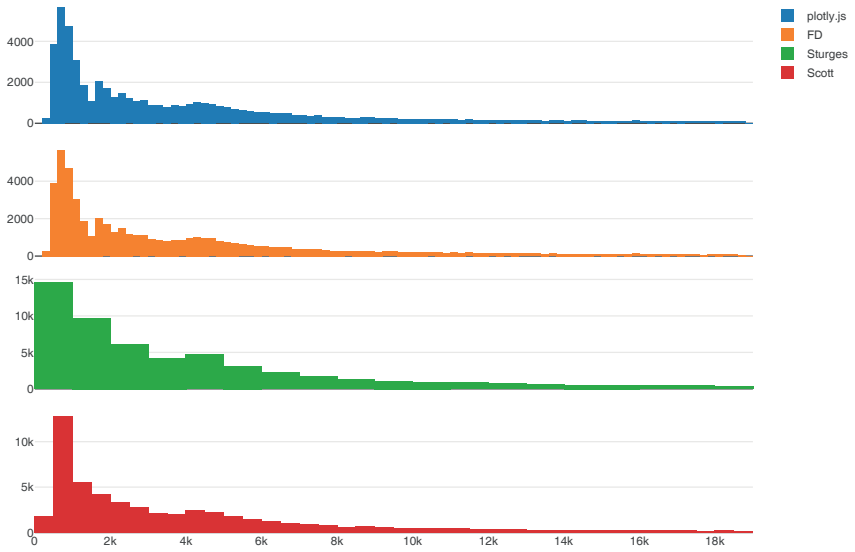

FIGURE 5.1: plotly.js's default binning algorithm versus R's `hist()` default.

Figure 5.2 demonstrates two ways of creating a basic bar chart. Although the visual results are the same, it is 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 not 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.[7]

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

p2 <- diamonds %>%
  count(cut) %>%
  plot_ly(x = ~cut, y = ~n) %>%
  add_bars()

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

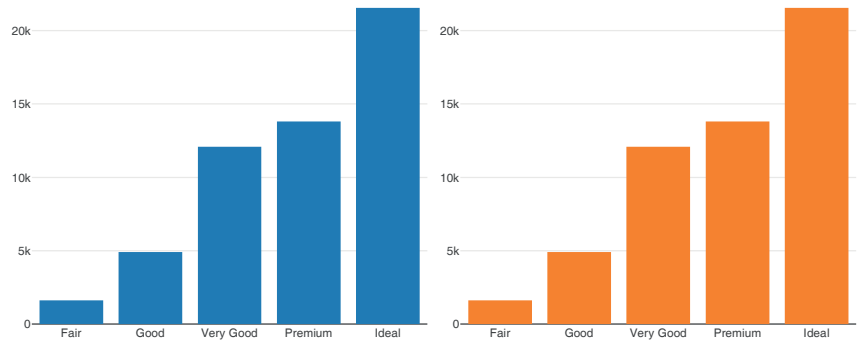

FIGURE 5.2: Number of diamonds by cut.

[7] These tests were run on Google Chrome and loaded a page with a single bar chart. See `https://www.webpagetest.org/result/160924_DP_JBX` for `add_histogram()` and `https://www.webpagetest.org/result/160924_QG_JA1` for `add_bars()`.

5.1 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 using a small multiples display, rather than trying to overlay them on a single axis.[8] Chapter 13, and specifically Section 13.1.2.3, discusses small multiples in more detail, but Figure 13.9 demonstrates how it is done with `plot_ly()` and `subplot()`. Note how the `one_plot()` function defines what to display on each panel, then a split-apply-recombine (i.e., `split()`, `lapply()`, `subplot()`) strategy is employed to generate the trellis display.

```
one_plot <- function(d) {
  plot_ly(d, x = ~price) %>%
    add_annotations(
      ~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()
```

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

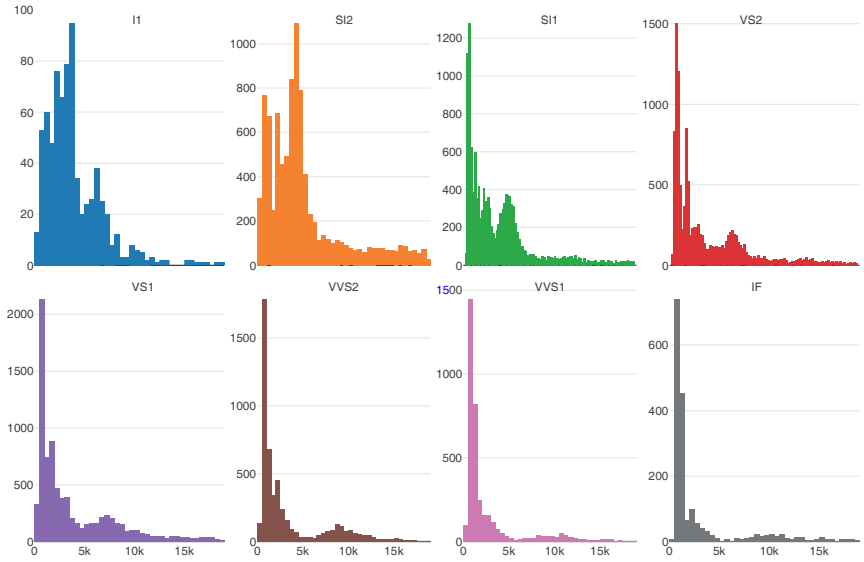

FIGURE 5.3: A trellis display of diamond price by diamond clarity.

5.2 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 5.4 presents diamond counts, divided by both their cut and clarity, using a grouped bar chart.

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

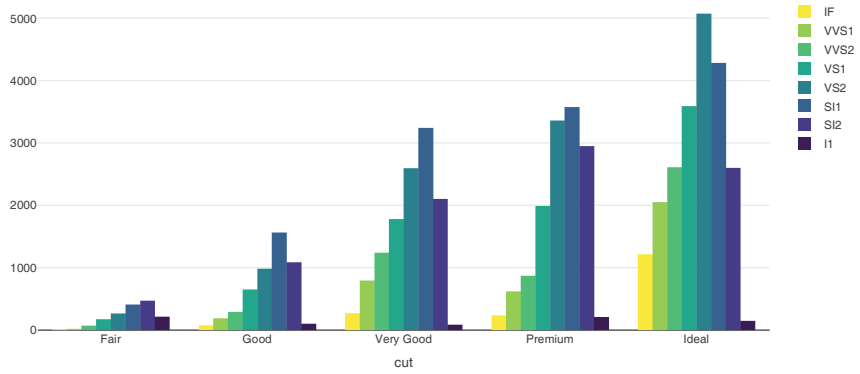

FIGURE 5.4: A grouped bar chart of diamond counts by cut and clarity.

Figure 5.4 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 5.5 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, name = 'nn'))
cc2 %>%
  mutate(prop = n / nn) %>%
  plot_ly(x = ~cut, y = ~prop, color = ~clarity) %>%
  add_bars() %>%
  layout(barmode = "stack")
```

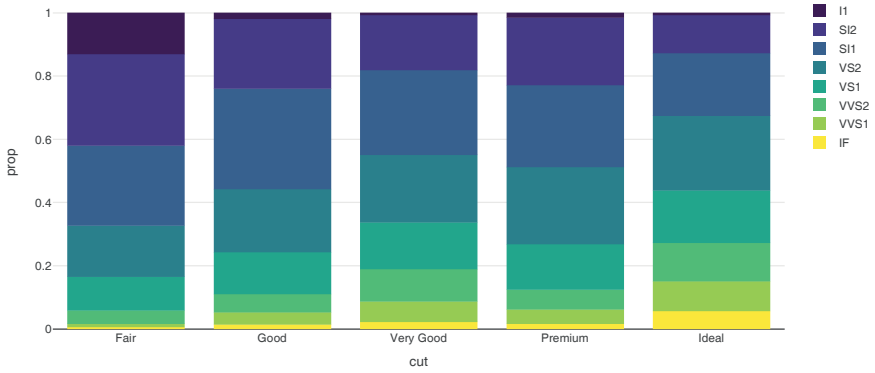

FIGURE 5.5: A stacked bar chart showing the proportion of diamond clarity within cut.

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 the **ggmosaic** package (Jeppson et al., 2016), which implements a custom **ggplot2** geom designed for mosaic plots, which we can convert to plotly via `ggplotly()`. Figure 5.6 shows 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))
ggplotly(p)
```

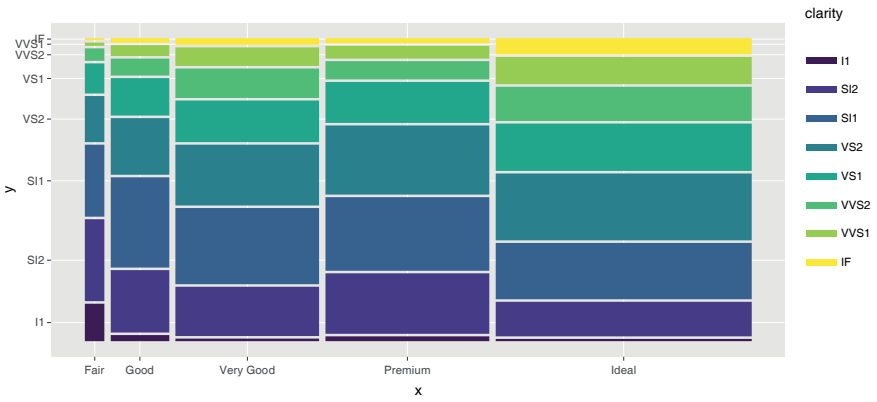

FIGURE 5.6: Using **ggmosaic** and `ggplotly()` to create advanced interactive visualizations of categorical data.

6

Boxplots

Boxplots encode the five number summary of a numeric variable, and provide a decent way to compare many numeric distributions. The visual task of comparing multiple boxplots is relatively easy (i.e., compare position along a common scale) compared to some common alternatives (e.g., a trellis display of histograms, like Figure 5.1), but the boxplot is sometimes inadequate for capturing complex (e.g., multi-modal) distributions (in this case, a frequency polygon, like Figure 2.9 provides a nice alternative). The `add_boxplot()` function requires one numeric variable, and guarantees boxplots are oriented[1] correctly, regardless of whether the numeric variable is placed on the x or y scale. As Figure 6.1 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()
```

[1] https://plot.ly/r/reference/#box-orientation

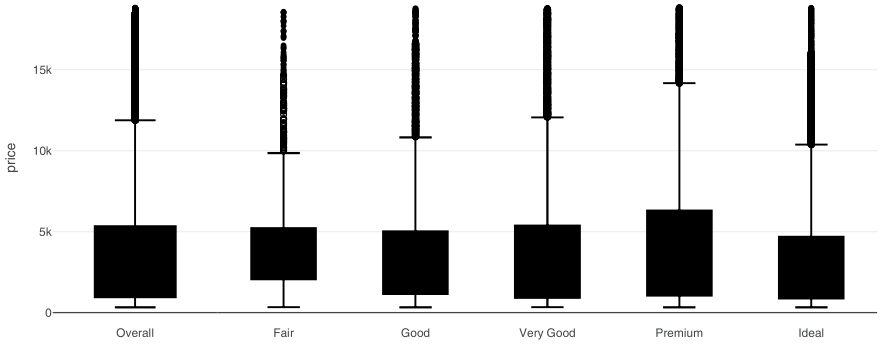

FIGURE 6.1: Overall diamond price and price by cut.

If you want to partition by more than one discrete variable, you could use the interaction of those variables to the discrete axis, and coloring by the nested variable, as Figure 6.2 does with diamond clarity and cut. Another approach would be to use a trellis display, similar to Figure 13.9.

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

FIGURE 6.2: Diamond prices by cut and clarity.

It is also helpful to sort the boxplots according to something meaningful, such as the median price. Figure 6.3 presents the same information as Figure 6.2, 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) %>%
  pull(cc)

plot_ly(d, x = ~price, y = ~factor(cc, lvls)) %>%
```

```
  add_boxplot(color = ~clarity) %>%
  layout(yaxis = list(title = ""))
```

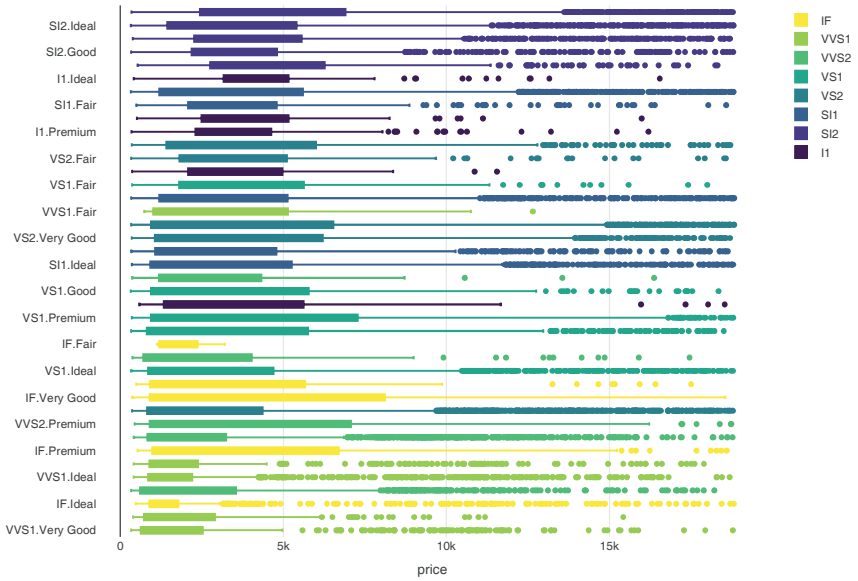

FIGURE 6.3: Diamond prices by cut and clarity, sorted by price median.

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.[2]

[2] Follow the issue here `https://github.com/plotly/plotly.js/issues/1059`

7

2D frequencies

7.1 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 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`[1] attribute for effectively increasing the number of bins (currently, "best" performs a bi-linear interpolation[2], a type of nearest neighbors algorithm), and `nbinsx`[3]/`nbinsy`[4] attributes to set the number of bins in the x and/or y directions. Figure 7.1 compares three different uses of `add_histogram()`: (1) plotly.js's default binning algorithm, (2) the default plus smoothing, (3) setting the number of bins in the x and y directions. It is also worth noting that filled contours, instead of bins, can be used in any of these cases by using `add_histogram2dcontour()` instead of `add_histogram2d()`.

```
p <- plot_ly(diamonds, x = ~log(carat), y = ~log(price))
subplot(
  add_histogram2d(p) %>%
    colorbar(title = "default") %>%
    layout(xaxis = list(title = "default")),
```

[1] https://plot.ly/r/reference/#histogram2d-zsmooth
[2] https://en.wikipedia.org/wiki/Bilinear_interpolation
[3] https://plot.ly/r/reference/#histogram2d-nbinsx
[4] https://plot.ly/r/reference/#histogram2d-nbinsy

```
  add_histogram2d(p, zsmooth = "best") %>%
    colorbar(title = "zsmooth") %>%
    layout(xaxis = list(title = "zsmooth")),
  add_histogram2d(p, nbinsx = 60, nbinsy = 60) %>%
    colorbar(title = "nbins") %>%
    layout(xaxis = list(title = "nbins")),
  shareY = TRUE, titleX = TRUE
)
```

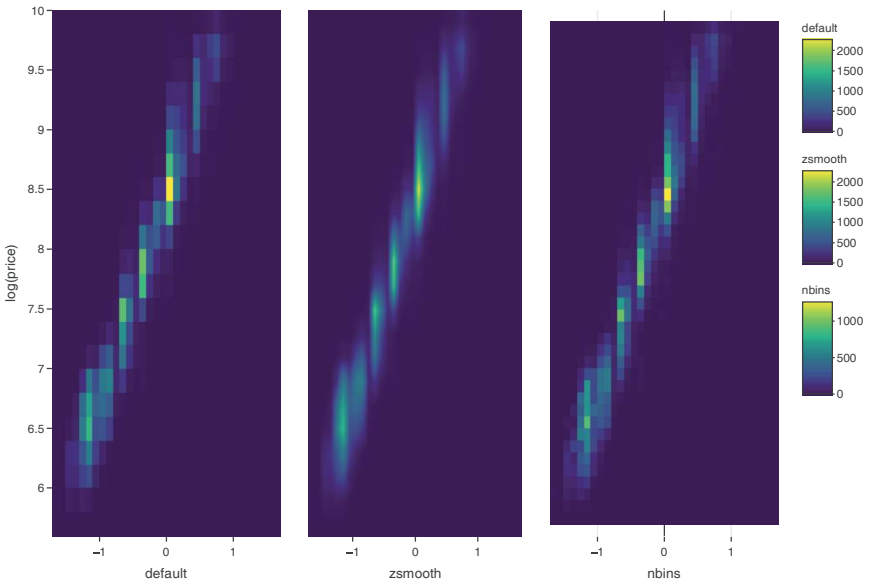

FIGURE 7.1: Three different uses of `histogram2d()`.

7.2 Rectangular binning in R

In Chapter 5, 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 (Scott, 1992). Practically speaking, that assumption is not very useful, but 2D kernel density estimation provides a useful alternative that tends to be more robust to changes in distributional form. Although kernel density estimation requires choice of kernel and a bandwidth parameter, the `kde2d()` function from the **MASS** package provides a well-supported rule-of-thumb for estimating the bandwidth of a Gaussian kernel density (Venables and Ripley, 2002). Figure 7.2 uses `kde2d()` to estimate a 2D density, scales the relative frequency to an absolute frequency, then uses the `add_heatmap()` function to display the results as a heatmap.

```
kde_count <- function(x, y, ...) {
  kde <- MASS::kde2d(x, y, ...)
  df <- with(kde, setNames(expand.grid(x, y), c("x", "y")))
  # The 'z' returned by kde2d() is a proportion,
  # but we can scale it to a count
  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")
```

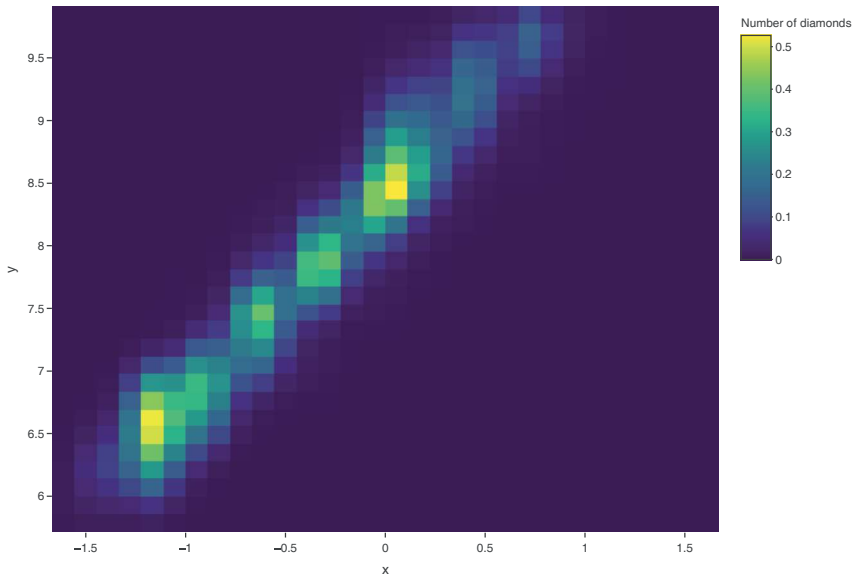

FIGURE 7.2: 2D density estimation via the `kde2d()` function.

7.3 Categorical axes

The functions `add_histogram2d()`, `add_histogram2dcontour()`, and `add_heatmap()` all support categorical axes. Thus, `add_histogram2d()` *can* be used to easily display 2-way contingency tables, but since it is easier to compare values along a common scale rather than compare colors (Cleveland and McGill, 1984), I recommend creating grouped bar charts instead. The `add_heatmap()` function can still be useful for categorical axes, however, as it allows us to display whatever quantity we want along the z axis (color).

Figure 7.3 uses `add_heatmap()` to display a correlation matrix. Notice how the `limits` arguments in the `colorbar()` function can be used to expand the limits of the color scale to reflect the range of possible correlations (something that is not easily done in plotly.js).

```
corr <- cor(dplyr::select_if(diamonds, is.numeric))
plot_ly(colors = "RdBu") %>%
  add_heatmap(x = rownames(corr), y = colnames(corr), z = corr) %>%
  colorbar(limits = c(-1, 1))
```

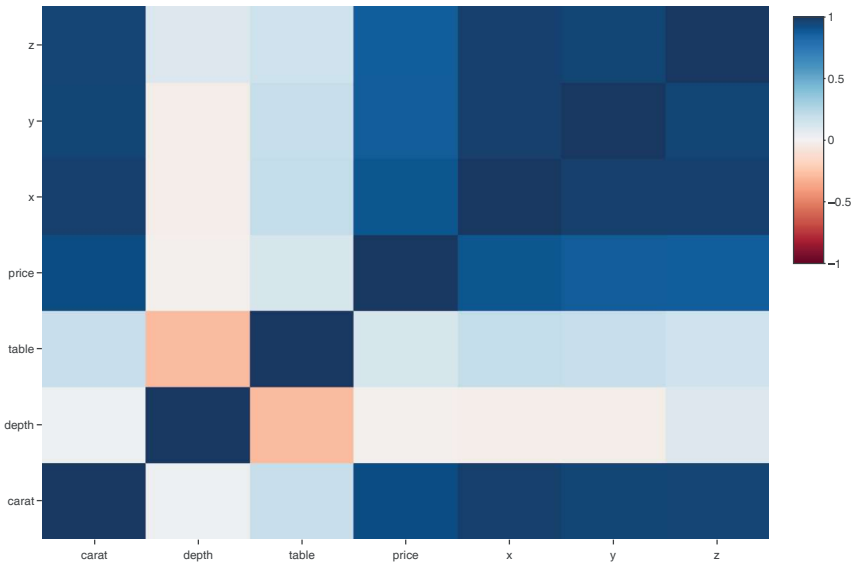

FIGURE 7.3: Displaying a correlation matrix with `add_heatmap()` and controlling the scale limits with `colorbar()`.


Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

8

3D charts

8.1 Markers

As it turns out, by simply adding a `z` attribute `plot_ly()` automatically renders markers, lines, and paths in three dimensions. That means, all the techniques we learned in Sections 3.1 and 3.2 can be re-used for 3D charts:

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

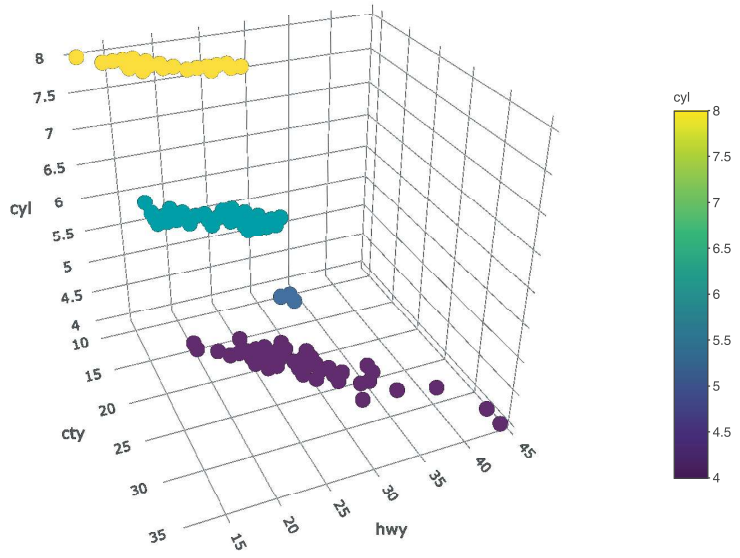

FIGURE 8.1: A 3D scatterplot.

8.2 Paths

To make a path in 3D, use `add_paths()` in the same way you would for a 2D path, but add a third variable `z`, as Figure 8.2 does.

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

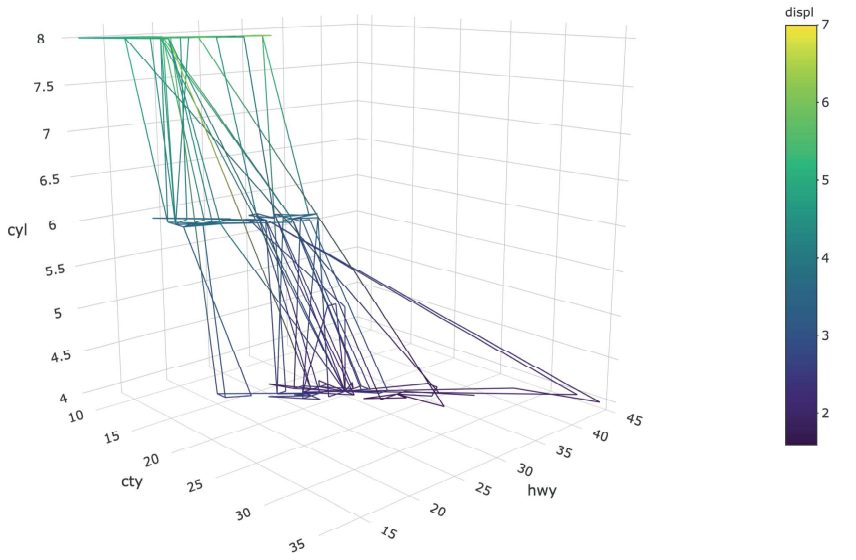

FIGURE 8.2: A path with color interpolation in 3D.

8.3 Lines

Figure 8.3 uses `add_lines()` instead of `add_paths()` to ensure the points are connected by the x axis instead of the row ordering.

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

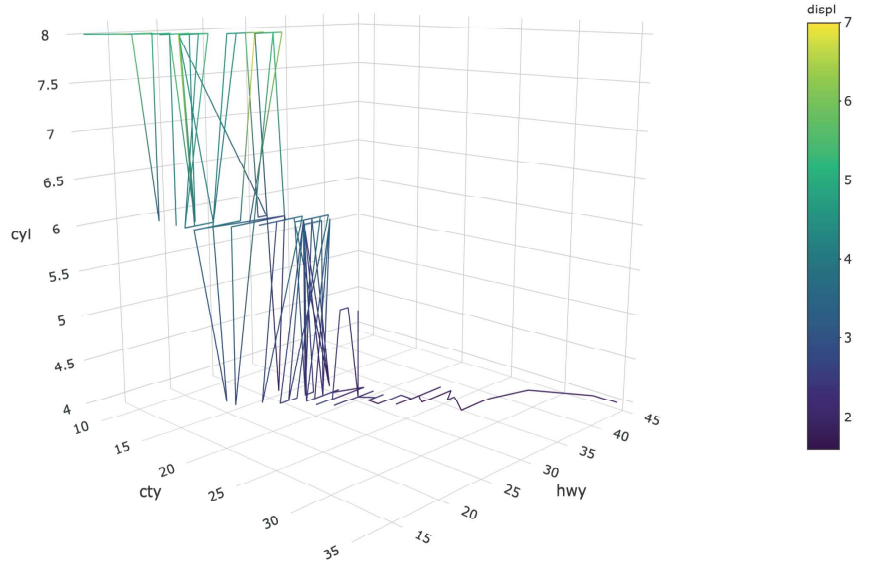

FIGURE 8.3: A line with color interpolation in 3D.

As with non-3D lines, you can make multiple lines by specifying a grouping variable.

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

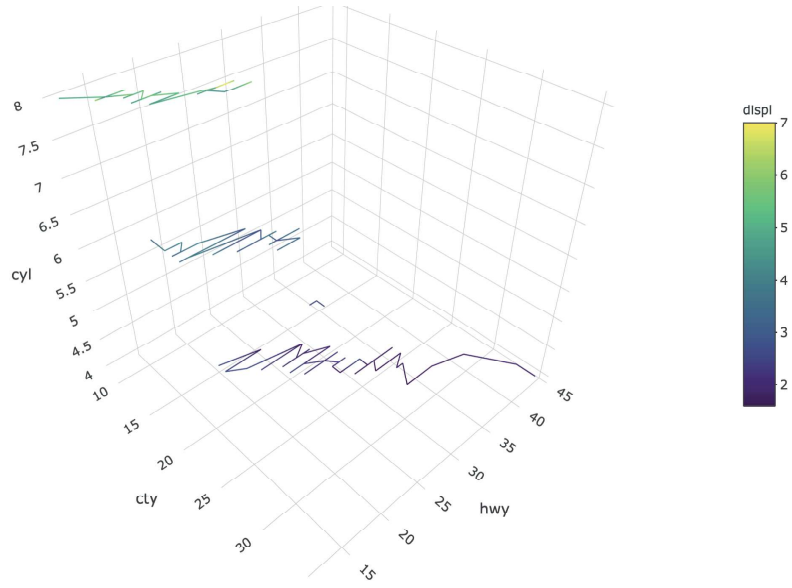

FIGURE 8.4: Using `group_by()` to create multiple 3D lines.

8.4 Axes

For 3D plots, be aware that the axis objects are a part of the `scene`[1] definition, which is part of the `layout()`. That is, if you wanted to set axis titles (e.g., Figure 8.5), or something else specific to the axis definition, the relation between axes (i.e., `aspectratio`[2]), or the default setting of the camera (i.e., `camera`[3]); you would do so via the `scence`.

```
plot_ly(mpg, x = ~cty, y = ~hwy, z = ~cyl) %>%
  add_lines(color = ~displ) %>%
  layout(
    scene = list(
      xaxis = list(title = "MPG city"),
```

[1] https://plot.ly/r/reference/#layout-scene
[2] https://plot.ly/r/reference/#layout-scene-aspectratio
[3] https://plot.ly/r/reference/#layout-scene-camera

```
      yaxis = list(title = "MPG highway"),
      zaxis = list(title = "Number of cylinders")
    )
  )
```

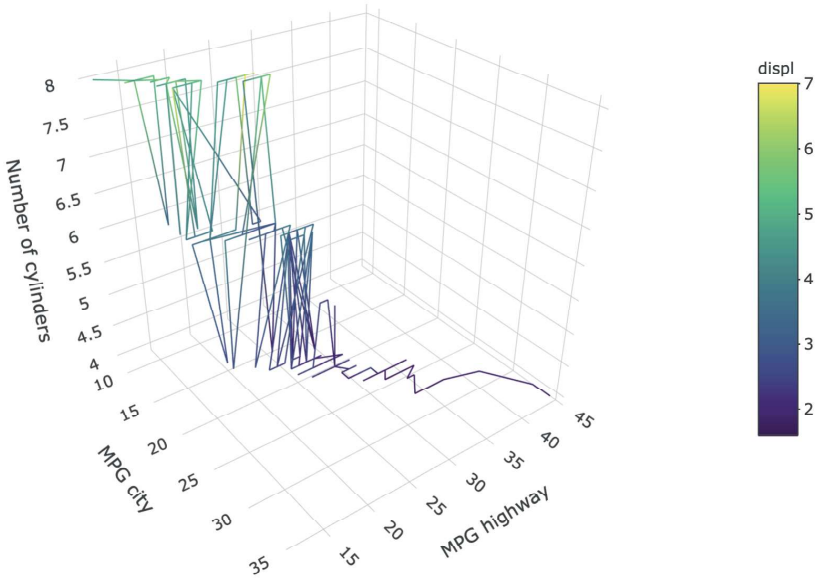

FIGURE 8.5: Setting axis titles on a 3D plot.

8.5 Surfaces

Creating 3D surfaces with `add_surface()` is a lot like creating heatmaps with `add_heatmap()`. In fact, you can even create 3D surfaces over categorical x/y (try changing `add_heatmap()` to `add_surface()` in Figure 7.3)! That being said, there should be a sensible ordering to the x/y axes in a surface plot since plotly.js interpolates z values. Usually the 3D surface is over a continuous region, as is done in Figure 8.6 to display the height of a volcano. If a numeric matrix is provided to z as in Figure 8.6, the x and y attributes do not have to be provided, but if they are,

the length of x should match the number of columns in the matrix and y should match the number of rows.

```
x <- seq_len(nrow(volcano)) + 100
y <- seq_len(ncol(volcano)) + 500
plot_ly() %>% add_surface(x = ~x, y = ~y, z = ~volcano)
```

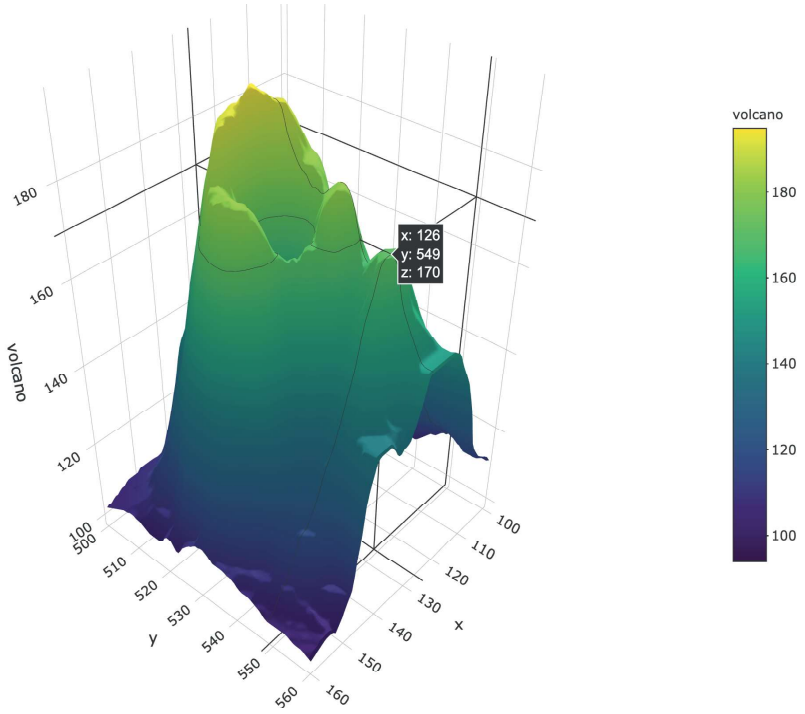

FIGURE 8.6: A 3D surface of volcano height.