17

Interactive Visualization in R

Adding interactivity to a visualization provides an additional mechanism through which data can be presented in an engaging, efficient, and communicative way. Interactions can allow users to effectively explore large data sets by *panning* and *zooming* through plots, or by *hovering* over specific plot geometry to gain additional details on demand.[1]

While `ggplot2` is the definitive, leading package for making static plots in R, there is not a comparably popular package for creating *interactive* visualizations. Thus this chapter briefly introduces three different packages for building such visualizations. Instead of offering an in-depth description (as with `ggplot2`), this chapter provides a high-level "tour" of these packages. The first two (*Plotly* and *Bokeh*) are able to add basic interactions to the plots you might make with `ggplot2`, while the third (*Leaflet*) is used to create interactive map visualizations. Picking among these (and other) packages depends on the type of interactions you want your visualization to provide, the ease of use, the clarity of the package documentation, and your aesthetic preferences. And because these open source projects are constantly evolving, you will need to reference their documentation to make the best use of these packages. Indeed, exploring these packages further is great practice in learning to use new R packages!

The first two sections demonstrate creating interactive plots of the `iris` data set, a canonical data set in the machine learning and visualization world in which a flower's species is predicted using features of that flower. The data set is built into the R software, and is partially shown in Figure 17.1.

For example, you can use `ggplot2` to create a static visualization of flower *species* in terms of the length of the petals and the sepals (the container for the buds), as shown in Figure 17.2:

```
# Create a static plot of the iris data set
ggplot(data = iris) +
  geom_point(mapping = aes(x = Sepal.Width, y = Petal.Width, color = Species))
```

The following sections show how to use the `plotly` and `rbokeh` packages to make this plot interactive. The third section of the chapter then explores interactive mapping with the `leaflet` package.

[1] Shneiderman, B. (1996). The eyes have it: A task by data type taxonomy for information visualizations. *Proceedings of the. 1996 IEEE Symposium on Visual Languages* (pp. 336–). Washington, DC: IEEE Computer Society. http://dl.acm.org/citation.cfm?id=832277.834354

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5.0	3.4	1.5	0.2	setosa

Figure 17.1 A subset of the `iris` data set, in which each observation (row) represents the physical measurements of a flower. This canonical data set is used to practice the machine learning task of *classification*—the challenge is to predict (*classify*) each flower's `Species` based on the other features.

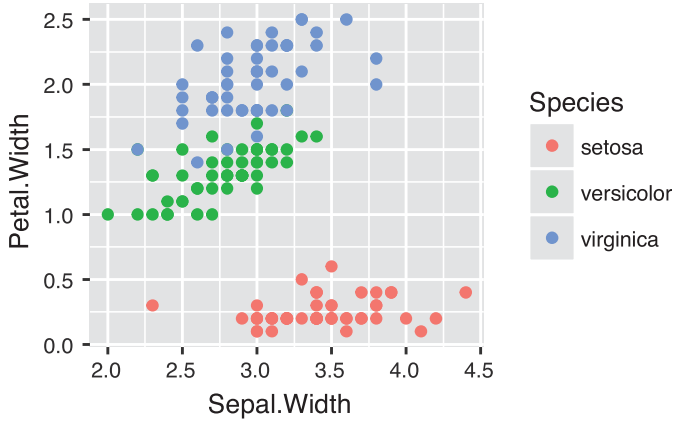

Figure 17.2 A static visualization of the `iris` data set, created using `ggplot2`.

17.1 The `plotly` Package

Plotly[2] is a piece of visualization software that provides open source APIs (programming libraries) for creating interactive visualizations in a wide variety of languages, including R, Python, Matlab, and JavaScript. By default, Plotly charts support a wide range of user interactions, including tooltips on hover, panning, and zooming in on selected regions.

Plotly is an external package (like `dplyr` or `ggplot2`), so you will need to install and load the package before you can use it:

```
install.packages("plotly") # once per machine
library("plotly")          # in each relevant script
```

[2] **Plotly**: https://plot.ly/r/

This will make all of the plotting functions you will need available.

With the package loaded, there are two main ways to create interactive plots. First, you can take any plot created using ggplot2 and "wrap" it in a Plotly plot,[3] thereby adding interactions to it. You do this by taking the plot returned by the ggplot() function and passing it into the **ggplotly()** function provided by the plotly package:

```
# Create (and store) a scatterplot of the `iris` data set using ggplot2
flower_plot <- ggplot(data = iris) +
  geom_point(mapping = aes(x = Sepal.Width, y = Petal.Width, color = Species))

# Make the plot interactive by passing it to Plotly's `ggplotly()` function
ggplotly(flower_plot)
```

This will render an interactive version of the iris plot! You can hover the mouse over any geometry element to see details about that data point, or you can click and drag in the plot area to zoom in on a cluster of points (see Figure 17.3).

When you move the mouse over a Plotly chart, you can see the suite of interaction types built into it through the menu that appears (see Figure 17.3). You can use these options to navigate and zoom into the data to explore it.

In addition to making ggplot plots interactive, you can use the Plotly API itself (e.g., calling its own functions) to build interactive graphics. For example, the following code will create an equivalent plot of the iris data set:

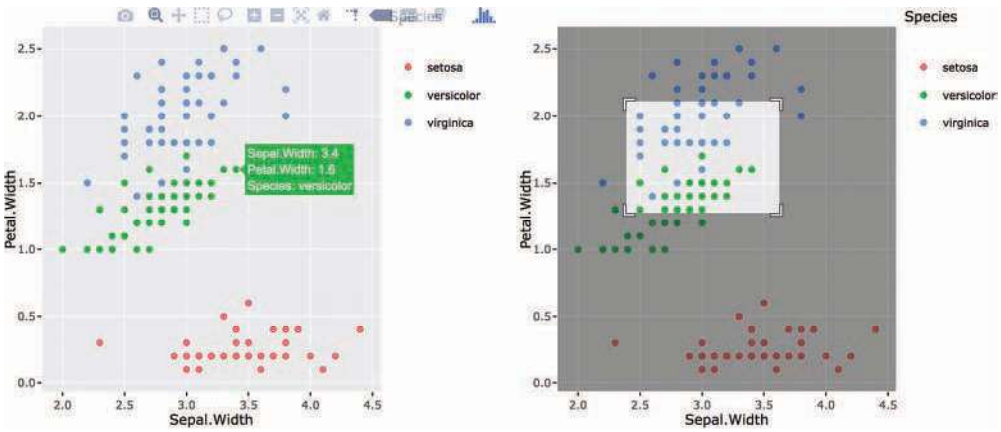

Figure 17.3 Plotly chart interactions: hover for tooltips (left), and brush (click + drag) to zoom into a region (right). More interactions, such as panning, are provided via the interaction menu at the top of the left-hand chart.

[3] **Plotly ggplot2 library**: https://plot.ly/ggplot2/ (be sure to check the navigation links in the menu on the left).

```
# Create an interactive plot of the iris data set using Plotly
plot_ly(
  data = iris,      # pass in the data to be visualized
  x = ~Sepal.Width, # use a formula to specify the column for the x-axis
  y = ~Petal.Width, # use a formula to specify the column for the y-axis
  color = ~Species, # use a formula to specify the color encoding
  type = "scatter", # specify the type of plot to create
  mode = "markers"  # determine the "drawing mode" for the scatter (points)
)
```

Plotly plots are created using the **`plot_ly()`** function, which is a sort of corollary to the `ggplot()` function. The `plot_ly()` function takes as arguments details about how the chart should be rendered. For example, in the preceding code, arguments are used to specify the data, the aesthetic mappings, and the plot type (that is, geometry). Aesthetic mappings are specified as *formulas* (using a tilde ~), indicating that the visual channel is a "function of" the data column. Also note that Plotly will try to "guess" values such as `type` and `mode` if they are left unspecified (and in which case it will print out a warning in the console).

For a complete list of options available to the `plot_ly()` function, see the official documentation.[4] It's often easiest to learn to make Plotly charts by working from one of the many examples.[5] We suggest that you find an example that is close to what you want to produce, and then read that code and modify it to fit your particular use case.

In addition to using the `plot_ly()` function to specify how the data will be rendered, you can add other chart options, such as titles and axes labels. These are specified using the **`layout()`** function, which is conceptually similar to the `labs()` and `theme()` functions from `ggplot2`. Plotly's `layout()` function takes as an argument a *Plotly chart* (e.g., one returned by the `plot_ly()` function), and then modifies that object to produce a chart with a different layout. Most commonly, this is done by *piping* the Plotly chart into the `layout()` function:

```
# Create a plot, then pipe that plot into the `layout()` function to modify it
# (Example adapted from the Plotly documentation)
plot_ly(
  data = iris,      # pass in the data to be visualized
  x = ~Sepal.Width, # use a formula to specify the column for the x-axis
  y = ~Petal.Width, # use a formula to specify the column for the y-axis
  color = ~Species, # use a formula to specify the color encoding
  type = "scatter", # specify the type of plot to create
  mode = "markers"  # determine the "drawing mode" for the scatter (points)
) %>%
  layout(
    title = "Iris Data Set Visualization",                  # plot title
    xaxis = list(title = "Sepal Width", ticksuffix = "cm"), # axis label + format
    yaxis = list(title = "Petal Width", ticksuffix = "cm")  # axis label + format
  )
```

[4] **Plotly: R Figure Reference**: https://plot.ly/r/reference/
[5] **Plotly: Basic Charts** example gallery: https://plot.ly/r/#basic-charts

The chart created by this code is shown in Figure 17.4. The `xaxis` and `yaxis` arguments expect *lists* of axis properties, allowing you to control many aspects of each axis (such as the `title` and the `ticksuffix` to put after each numeric value in the axis). You can read about the structure and options to the other arguments in the API documentation.[6]

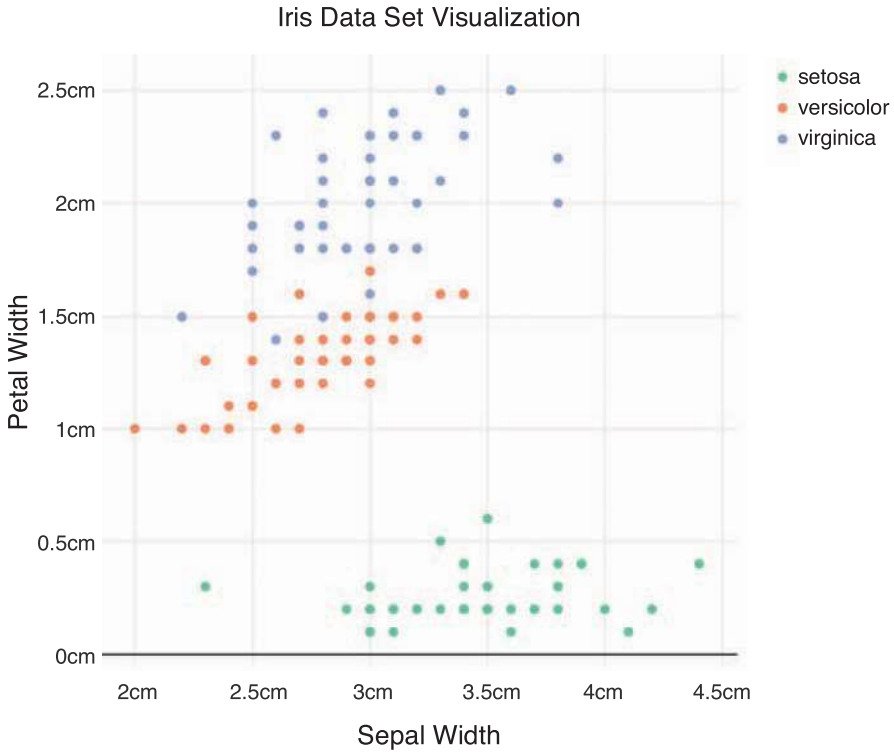

Figure 17.4 A Plotly chart with informative labels and axes added using the `layout()` function.

17.2 The rbokeh Package

Bokeh[7] is a visualization package that provides a similar set of interactive features as Plotly (including hover tooltips, drag-to-pan, and box zoom effects). Originally developed for the Python programming language, Bokeh can be used in R through the **rbokeh** package.[8] While not as popular as Plotly, Bokeh's API and documentation can be more approachable than Plotly's examples.

[6] **Plotly layout**: https://plot.ly/r/reference/#layout
[7] **Bokeh**: http://bokeh.pydata.org
[8] **rbokeh, R Interface for Bokeh**: http://hafen.github.io/rbokeh/

As with other packages, you will need to install and load the `rbokeh` package before you can use it. At the time of this writing, the version of `rbokeh` on CRAN (what is installed with `install.packages()`) gives warnings—but not errors!—for R version 3.4; installing a development version from the package's maintainer Ryan Hafen fixes this problem.

```
# Use `install_github()` to install the version of a package on GitHub
# (often newer)
devtools::install_github("hafen/rbokeh") # once per machine
library("rbokeh") # in each relevant script
```

You create a new plot with Bokeh by calling the **`figure()`** function (which is a corollary to the `ggplot()` and `plot_ly()` functions). The `figure()` function will create a new plotting area, to which you add layers of plot elements such as plot geometry. Similar to when using geometries in `ggplot2`, each layer is created with a different function—all of which start with the `ly_` prefix. These layer functions take as a first argument the plot region created with `figure()`, so in practice they are "added" to a plot through piping rather than through the addition operator.

For example, the following code shows how to recreate the `iris` visualization using Bokeh (shown in Figure 17.5):

```
# Create an interactive plot of the iris data set using Bokeh
figure(
  data = iris,                          # data for the figure
  title = "Iris Data Set Visualization" # title for the figure
) %>%
  ly_points(
    Sepal.Width,    # column for the x-axis (without quotes!)
    Petal.Width,    # column for the y-axis (without quotes!)
    color = Species # column for the color encoding (without quotes!)
  ) %>%
  x_axis(
    label = "Sepal Width",       # label for the axis
    number_formatter = "printf", # formatter for each axis tick
    format = "%s cm",            # specify the desired tick labeling
  ) %>%
  y_axis(
    label = "Petal Width",       # label for the axis
    number_formatter = "printf", # formatter for each axis tick
    format = "%s cm",            # specify the desired tick labeling
  )
```

The code for adding layers is reminiscent of how geometries act as layers in `ggplot2`. Bokeh even supports non-standard evaluation (referring to column names without quotes) just like `ggplot2`—as opposed to Plotly's reliance on formulas. However, formatting the axis tick marks is more verbose with Bokeh (and is not particularly clear in the documentation).

The plot that is generated by Bokeh (Figure 17.5) is quite similar to the version generated by Plotly (Figure 17.4) in terms of general layout, and offers a comparable set of interaction utilities through a

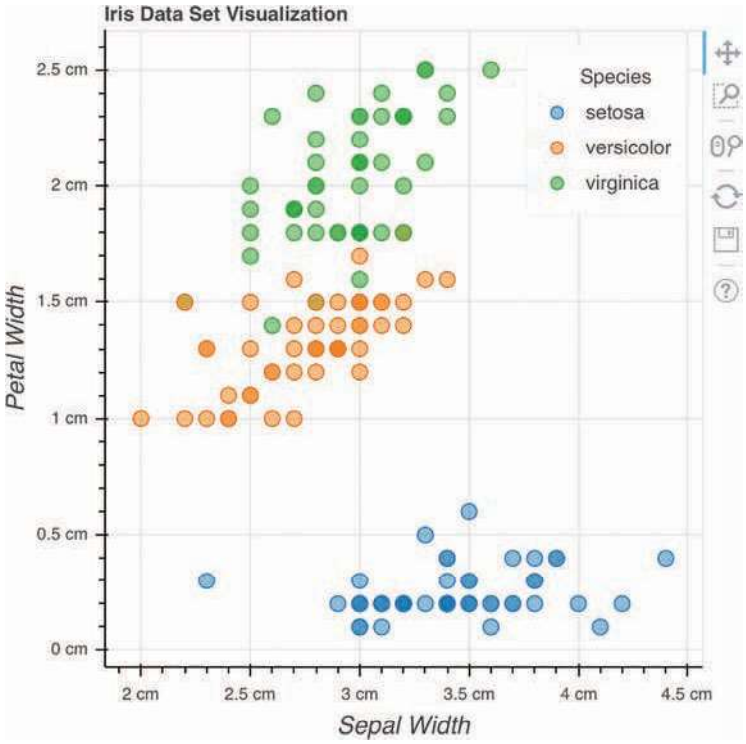

Figure 17.5 A Bokeh chart with styled axes. Note the interaction menu to the right of the chart.

toolbar to the right of the chart. Thus you might choose between these packages based on which coding style you prefer, as well as any other aesthetic or interactive design choices of the packages.

17.3 The `leaflet` Package

Leaflet[9] is an open source JavaScript library for building interactive maps, which you can use in R through the **`leaflet`** package.[10] Maps built with Leaflet have rich interactivity by default, including the ability to pan, zoom, hover, and click on map elements and markers. They can also be customized to support formatted labels or respond to particular actions. Indeed, many of the interactive maps you see accompanying online news articles are created using Leaflet.

As with other packages, you will need to install and load the `leaflet` package before you can use it:

```
install.packages("leaflet") # once per machine
library("leaflet")          # in each relevant script
```

[9] **Leaflet**: https://leafletjs.com
[10] **Leaflet for R**: https://rstudio.github.io/leaflet/

You can create a new Leaflet map by calling the **`leaflet()`** function. Just as calling `ggplot()` will create a blank canvas for constructing a plot, the `leaflet()` function will create a blank canvas on which you can build a map. Similar to the other visualization packages, Leaflet maps are then constructed by adding (via pipes) a series of layers with different visual elements to constitute the image—including map tiles, markers, lines, and polygons.

The most important layer to add when creating a Leaflet map are the **map tiles**, which are added with the **`addTiles()`** function. Map tiles are a series of small square images, each of which shows a single piece of a map. These tiles can then be placed next to each other (like tiles on a bathroom floor) to form the full image of the map to show. Map tiles power mapping applications like Leaflet and Google Maps, enabling them to show a map of the entire world at a wide variety of levels of zoom (from street level to continent level); which tiles will be rendered depends on what region and zoom level the user is looking at. As you interactively navigate through the map (e.g., panning to the side or zooming in or out), Leaflet will automatically load and show the appropriate tiles to display the desired map!

> **Fun Fact:** It takes 366,503,875,925 tiles (each 256×256 pixels) to map the entire globe for the (standard) 20 different zoom levels!

There are many different sources of map tiles that you can use in your maps, each of which has its own appearance and included information (e.g., rivers, streets, and buildings). By default, Leaflet will use tiles from *OpenStreetMap*,[11] an open source set of map tiles. OpenStreetMap provides a number of different tile sets; you can choose which to use by passing in the name of the tile set (or a URL schema for the tiles) to the `addTiles()` function. But you can also choose to use another map tile provider[12] depending on your aesthetic preferences and desired information. You do this by instead using the `addProviderTiles()` function (again passing in the name of the tile set). For example, the following code creates a basic map (Figure 17.6) using map tiles from the *Carto*[13] service. Note the use of the `setView()` function to specify where to center the map (including the "zoom level").

```
# Create a new map and add a layer of map tiles from CartoDB
leaflet() %>%
  addProviderTiles("CartoDB.Positron") %>%
  setView(lng = -122.3321, lat =       47.6062 =   ) # center the map on Seattle
```

The rendered map will be *interactive* in the sense that you can drag and scroll to pan and zoom—just as with other online mapping services!

After rendering a basic map with a chosen set of map tiles, you can add further layers to the map to show more information. For instance, you can add a layer of **shapes** or **markers** to help answer questions about events that occur at specific geographic locations. To do this, you will need to pass the data to map into the `leaflet()` function call as the `data` argument (i.e., `leaflet(data = SOME_DATA_FRAME)`). You can then use the `addCircles()` function to add a layer of circles to the

[11] **OpenStreetMap** map data service: https://www.openstreetmap.org
[12] **Leaflet-providers preview** http://leaflet-extras.github.io/leaflet-providers/preview/
[13] **Carto** map data service: https://carto.com

Figure 17.6 A map of Seattle, created using the `leaflet` package. The image is constructed by stitching together a layer of *map tiles*, provided by the Carto service.

map (similar to adding a geometry in `ggplot2`). This function will take as arguments the data columns to map to the circle's location aesthetics, specified as formulas (with a ~).

```
# Create a data frame of locations to add as a layer of circles to the map
locations <- data.frame(
  label = c("University of Washington", "Seattle Central College"),
  latitude = c(47.6553, 47.6163),
  longitude = c(-122.3035, -122.3216)
)

# Create the map of Seattle, specifying the data to use and a layer of circles
leaflet(data = locations) %>% # specify the data you want to add as a layer
  addProviderTiles("CartoDB.Positron") %>%
  setView(lng = -122.3321, lat = 47.6062, zoom = 11) %>% # focus on Seattle
  addCircles(
    lat = ~latitude,   # a formula specifying the column to use for latitude
    lng = ~ longitude, # a formula specifying the column to use for longitude
    popup = ~label,    # a formula specifying the information to pop up
    radius = 500,      # radius for the circles, in meters
    stroke = FALSE     # remove the outline from each circle
  )
```

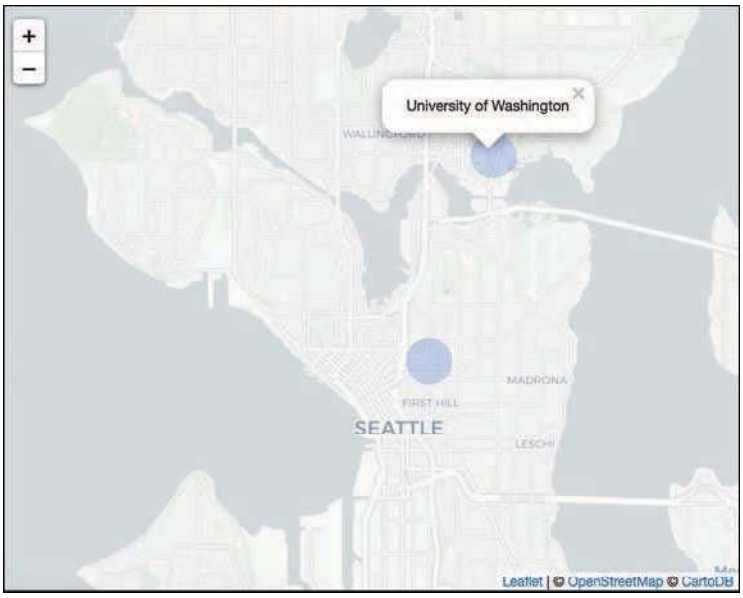

Figure 17.7 A map showing two universities in Seattle, created by adding a layer of markers (`addCircles()`) on top of a layer of map tiles.

> **Caution**: Interactive visualization packages such as `plotly` and `leaflet` are limited in the number of markers they can display. Because they render *scalable vector graphics* (SVGs) rather than raster images, they actually add a new visual element for each marker. As a result they are often unable to handle more than a few thousand points (something that isn't an issue with `ggplot2`).

The preceding code also adds interactivity to the map by providing *popups*—information that pops up *on click* and remains displayed—as shown in Figure 17.7. Because these popups appear when users are interacting with the circle elements you created, they are specified as another argument to the `addCircles()` function—that is, as a value of the formula for which column to map to the popup. Alternatively, you can cause labels to appear *on hover* by passing in the `label` argument instead of `popup`.

17.4 Interactive Visualization in Action: Exploring Changes to the City of Seattle

This section demonstrates using an interactive visualization in an attempt to evaluate the claim that *"The City of Seattle is changing"* (in large part due to the growing technology industry) by analyzing construction projects as documented through building permit data[14] downloaded from

[14] **City of Seattle Land use permits**: https://data.seattle.gov/Permitting/Building-Permits/76t5-zqzr

	PermitNum	PermitClass	PermitClassMapped	PermitTypeMapped	PermitTypeDesc	Description
1	6243602-CN	Commercial	Non-Residential	Building	New	Install portable office building (unit...
2	6408217-CN	Single Family/Duplex	Residential	Building	New	Establish use as and Construct new...
3	6285442-CN	Single Family/Duplex	Residential	Building	New	Establish use as and construct new ...
4	6343245-CN	Multifamily	Residential	Building	New	Establish use as and construct six-...
5	6547255-CN	Single Family/Duplex	Residential	Building	New	Establish use as and construct new ...
6	6271097-CN	Commercial	Non-Residential	Building	New	Establish use as car wash and mino...
7	6454733-CN	Single Family/Duplex	Residential	Building	New	Establish use as and construct new ...
8	6213875-PH	Multifamily	Residential	Building	New	Phased project: Construction of a r...
9	6583694-CN	Single Family/Duplex	Residential	Building	New	Construct East duplex, per plan (Es...
10	6312868-CN	Single Family/Duplex	Residential	Building	New	Establish use as and construct new ...
11	6464007-CN	Single Family/Duplex	Residential	Building	New	Establish use as and Construct new...
12	6363269-CN	Single Family/Duplex	Residential	Building	New	Construct CNTR single family resid...

Figure 17.8 City of Seattle data on permits for buildings in Seattle, showing the subset of new permits since 2010.

the City of Seattle's open data program. A subset of this data is shown in Figure 17.8. The complete code for this analysis is also available online in the book code repository.[15]

First, the data needs to be loaded into R and filtered down to the subset of data of interest (new buildings since 2010):

```
# Load data downloaded from
# https://data.seattle.gov/Permitting/Building-Permits/76t5-zqzr
all_permits <- read.csv("data/Building_Permits.csv", stringsAsFactors = FALSE)

# Filter for permits for new buildings issued in 2010 or later
new_buildings <- all_permits %>%
  filter(
    PermitTypeDesc == "New",
    PermitClass != "N/A",
    as.Date(all_permits$IssuedDate) >= as.Date("2010-01-01") # filter by date
  )
```

Before mapping these points, you may want to get a higher-level view of the data. For example, you could aggregate the data to show the number of permits issued per year. This will again involve a bit of data wrangling, which is often the most time-consuming part of visualization:

```
# Create a new column storing the year the permit was issued
new_buildings <- new_buildings %>%
  mutate(year = substr(IssuedDate, 1, 4)) # extract the year

# Calculate the number of permits issued by year
by_year <- new_buildings %>%
  group_by(year) %>%
  count()
```

[15] **Interactive visualization in action**: https://github.com/programming-for-data-science/in-action/tree/master/interactive-vis

```
# Use plotly to create an interactive visualization of the data
plot_ly(
  data = by_year, # data frame to show
  x = ~year,      # variable for the x-axis, specified as a formula
  y = ~n,         # variable for the y-axis, specified as a formula
  type = "bar",   # create a chart of type "bar" -- a bar chart
  alpha = .7,     # adjust the opacity of the bars
  hovertext = "y" # show the y-value when hovering over a bar
) %>%
  layout(
    title = "Number of new building permits per year in Seattle",
    xaxis = list(title = "Year"),
    yaxis = list(title = "Number of Permits")
  )
```

The preceding code produces the bar chart shown in Figure 17.9. Keep in mind that the data was downloaded before the summer of 2018, so the observed downward trend is an artifact of when the visualization was created!

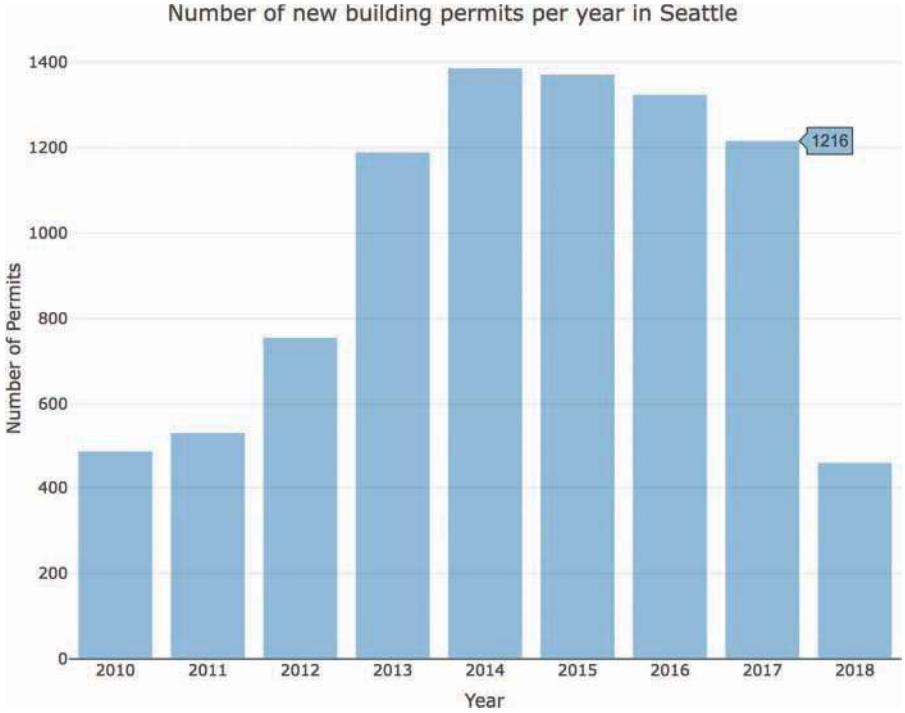

Figure 17.9 The number of permits issued for new buildings in Seattle since 2010. The chart was built before the summer of 2018.

After understanding this high-level view of the data, you likely want to know *where* buildings are being constructed. To do so, you can take the previous map of Seattle and add an additional layer of circles on top of the tiles (one for each building constructed) using the `addCircles()` function:

```
# Create a Leaflet map, adding map tiles and circle markers
leaflet(data = new_buildings) %>%
  addProviderTiles("CartoDB.Positron") %>%
  setView(lng = -122.3321, lat = 47.6062, zoom = 10) %>%
  addCircles(
    lat = ~Latitude,     # specify the column for `lat` as a formula
    lng = ~Longitude,    # specify the column for `lng` as a formula
    stroke = FALSE,      # remove border from each circle
    popup = ~Description # show the description in a popup
  )
```

The results of this code are shown in Figure 17.10—it's a lot of new buildings. And because the map is interactive, you can click on each one to get more details!

While this visualization shows all of the new construction, it leaves unanswered the question of *who benefits* and *who suffers* as a result of this change. You would need to do further research into the number of affordable housing units being built, and the impact on low-income and homeless communities. As you may discover, building at such a rapid pace often has a detrimental effect on housing security in a city.

As with `ggplot2`, the visual attributes of each shape or marker (such as the size or color) can also be driven by data. For example, you could use information about the permit classification (i.e., if the

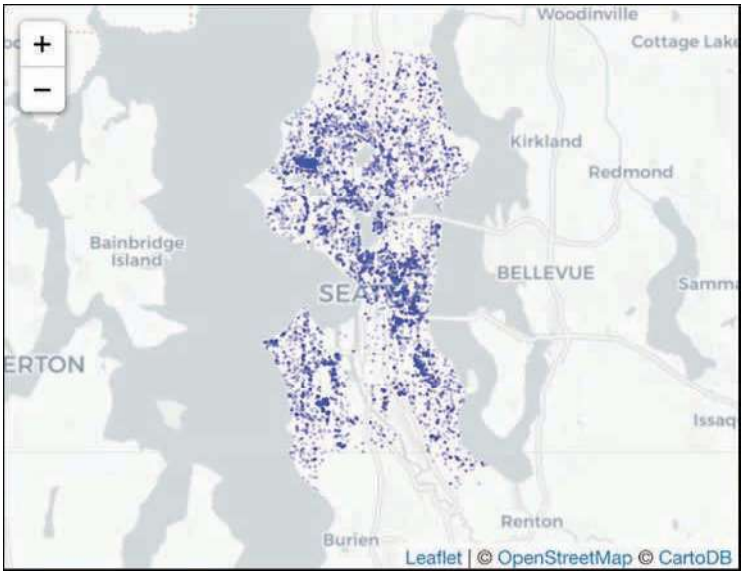

Figure 17.10 A Leaflet map of permits for new buildings in Seattle since 2010.

permit is for a home versus a commercial building) to color the individual circles. To effectively map this (categorical) data to a set of colors in Leaflet, you can use the **`colorFactor()`** function. This function is a lot like a scale in `ggplot2`, in that it returns a specific mapping to use:

```
# Construct a function that returns a color based on the PermitClass column
# Colors are taken from the ColorBrewer Set3 palette
palette_fn <- colorFactor(palette = "Set3", domain = new_buildings$PermitClass)
```

The `colorFactor()` function returns a new function (here called `palette_fn()`) that maps from a set of data values (here the unique values from the `PermitClass` column) to a set of colors—it performs an aesthetic mapping. You can use this function to specify how the circles on the map should be rendered (as with `ggplot2` geometries, further arguments can be used to customize the shape rendering):

```
# Modify the `addCircles()` method to specify color using `palette_fn()`
addCircles(
  lat = ~Latitude, # specify the column for `lat` as a formula
  lng = ~Longitude, # specify the column for `lng` as a formula
  stroke = FALSE, # remove border from each circle
  popup = ~Description, # show the description in a popup
  color = ~palette_fn(PermitClass) # a "function of" the palette mapping
)
```

To make these colors meaningful, you will need to add a legend to your map. As you might have expected, you can do this by adding another layer with a legend in it, specifying the color scale, values, and other attributes:

```
# Add a legend layer in the "bottomright" of the map
addLegend(
  position = "bottomright",
  title = "New Buildings in Seattle",
  pal = palette_fn, # the color palette described by the legend
  values = ~PermitClass, # the data values described by the legend
  opacity = 1
)
```

Putting it together, the following code generates the interactive map displayed in Figure 17.11.

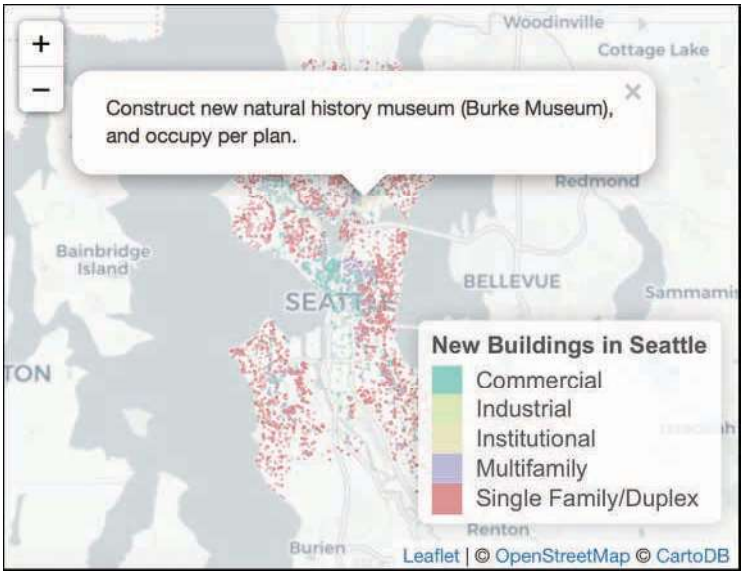

Figure 17.11 A Leaflet map of permits for new buildings in Seattle since 2010, colored by construction category.

```
# Create a Leaflet map of new building construction by category
leaflet(data = new_buildings) %>%
  addProviderTiles("CartoDB.Positron") %>%
  setView(lng = -122.3321, lat = 47.6062, zoom = 10) %>%
  addCircles(
    lat = ~Latitude, # specify the column for `lat` as a formula
    lng = ~Longitude, # specify the column for `lng` as a formula
    stroke = FALSE, # remove border from each circle
    popup = ~Description, # show the description in a popup
    color = ~palette_fn(PermitClass), # a "function of" the palette mapping
    radius = 20,
    fillOpacity = 0.5
  ) %>%
  addLegend(
    position = "bottomright",
    title = "New Buildings in Seattle",
    pal = palette_fn, # the palette to label
    values = ~PermitClass, # the values to label
    opacity = 1
  )
```

In summary, packages for developing interactive visualizations (whether plots or maps) use the same general concepts as `ggplot2`, but with their own preferred syntax for specifying plot options and customizations. As you choose among these (and other) packages for making visualizations,

consider the style of code you prefer to use, the trade-off of customizability versus ease of use, and the visual design choices of each package. There are dozens (if not hundreds) of other packages available and more created every day; exploring and learning these packages is an excellent way to expand your programming and data science skills.

That said, when you are exploring new packages, be careful about using code that is poorly documented or not widely used—such packages may have internal errors, memory leaks, or even security flaws that haven't been noticed or addressed yet. It's a good idea to view the package code *on GitHub*, where you can check the popularity by looking at the number of *stars* (similar to "likes") and *forks* for the project, as well as how actively and recently new commits have been made to the code. Such research and consideration are vital when choosing one of the many packages for building interactive visualizations—or doing any other kind of work—with R.

For practice building interactive visualizations, see the set of accompanying book exercises.[16]

[16] **Interactive visualization exercises**: https://github.com/programming-for-data-science/chapter-17-exercises