Leaflet in R

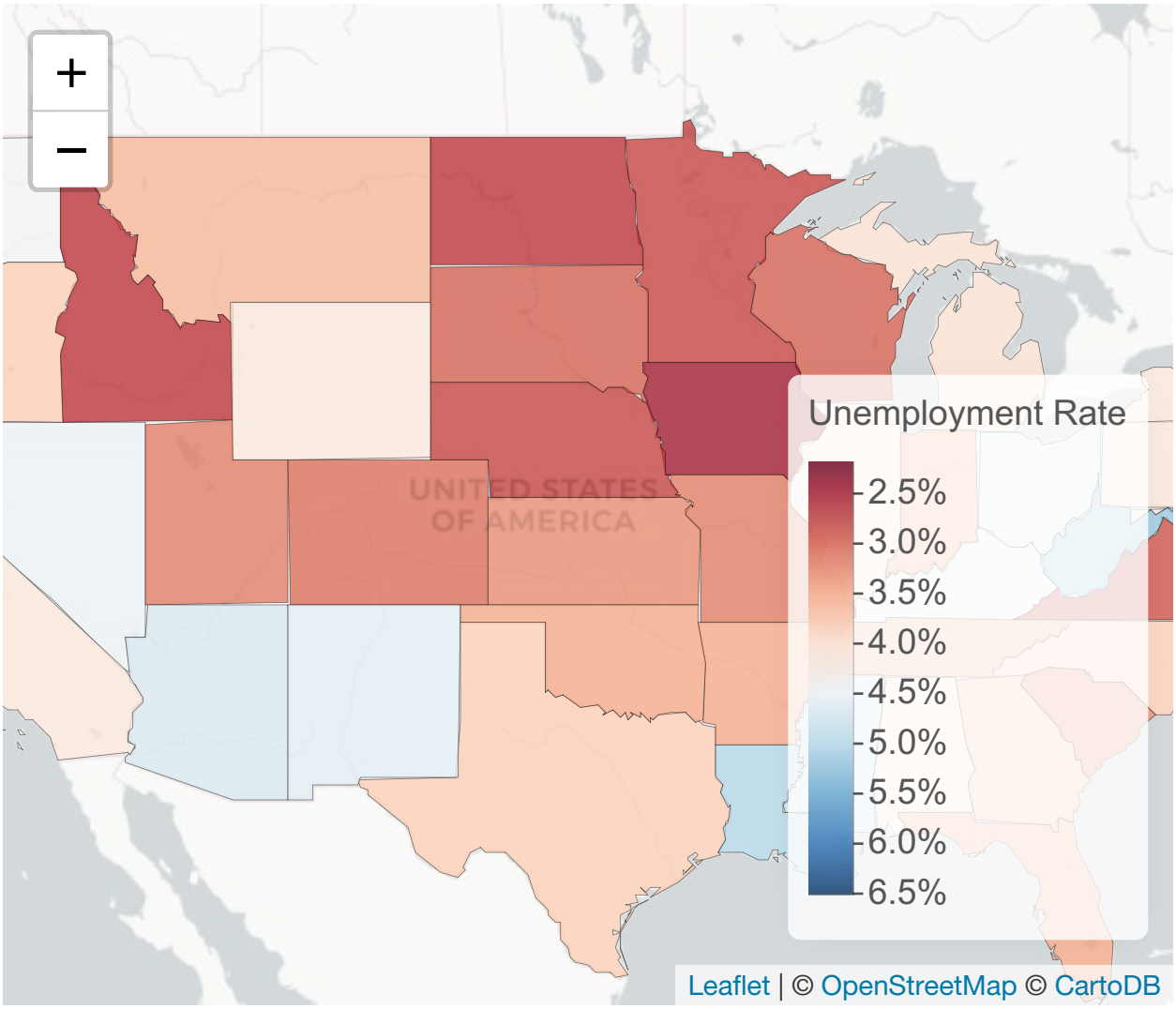
Table of Contents

Introduction

Leaflet is a popular JavaScript library that allows developers to create interactive maps (think homemade Google Maps). The maps allow users to zoom and pan around an area. The maps also support popups and various forms of data visualization, from points to lines to polygons.

The `leaflet` package for R allows you to create Leaflet maps without knowing JavaScript. The syntax is a bit different, but fairly simple given the extremely powerful and versatile maps it produces.

Below is an example of the type of map that we will be producing in this tutorial. This one displays state level unemployment data from September 2018.



Pros and Cons of Leaflet

Pros

- Zoom functionality that is especially handy when you have point data or small polygons such as block groups.
- Interactivity (Click & Hover Messages) so users don't have to rely on color alone to communicate the data.
- Great for applications like R Shiny and for webpages.

Cons

- You must be connected to the internet in order to receive the map tiles.
- There is no way to easily output the Leaflet map as an image like you can with a ggplot map.
- Not great for documents. Maps don't always scale well though this can be adjusted.
- It is difficult to include Alaska and Hawaii in a nice looking state level map. Users will often have to pan around on the map to see data displayed for these states.

A Basic Leaflet Map

I will start by showing you the basic Leaflet map syntax. Don't worry if you don't get all of the terminology now, it will be explained later.

Call Your Libraries

The `leaflet` package is all that's needed to build the map, but in order to read in geographic data (at least polygons and lines) you will need to also import the `rgdal` package.

```
library(leaflet)
library(rgdal)
```

Import Your Data

Here I am importing a locally stored .geojson file that contains state level data. How I prepared this dataset is outside the scope of this tutorial, but you can get the geographic shapefiles from the Census Bureau and the data files used for the map below from BLS's OES Program. The `readOGR()` function is part of the `rgdal` package.

```
dat <- readOGR("./data/states_geojson.geojson", "OGRGeoJSON", verbose = F)
```

Define Your Color Palette and Popup

We need to define how we want colors displayed in our choropleth map. Leaflet uses color palettes from `RColorBrewer`. You can see these palettes by typing `display.brewer.all()` into the R console after loading `RColorBrewer`. You also have several options in whether the color variable is displayed as continuous or a discrete series of bins. The `colorFactor()` function is used for displaying nominal data in the color palette.

You can also define your popup message. You don't have to define a popup in order for the map to display. Note here that I use HTML tags. These popups can be as elaborate as you want them to be if you know HTML. You can include images, tables, graphs, etc. In this popup, I just display the data that will be shown in the map.

```
# DEFINE COLOR VARIABLE (colorNumeric(), colorQuantile(), colorBin(), colorFactor())
pal <- colorNumeric("RdYlGn", domain = dat@data$tot_employment, na.color = "gray50")

# DEFINE POPUP FOR HOVER OR CLICK FUNCTIONALITY
popup <-  with(dat@data,paste(sep = "",
                                "<h3> ",NAME,"</h3>",
                                "<b>Total Employment:</b>",format(tot_employment, big.mark = ",")))
```

Build Your Map

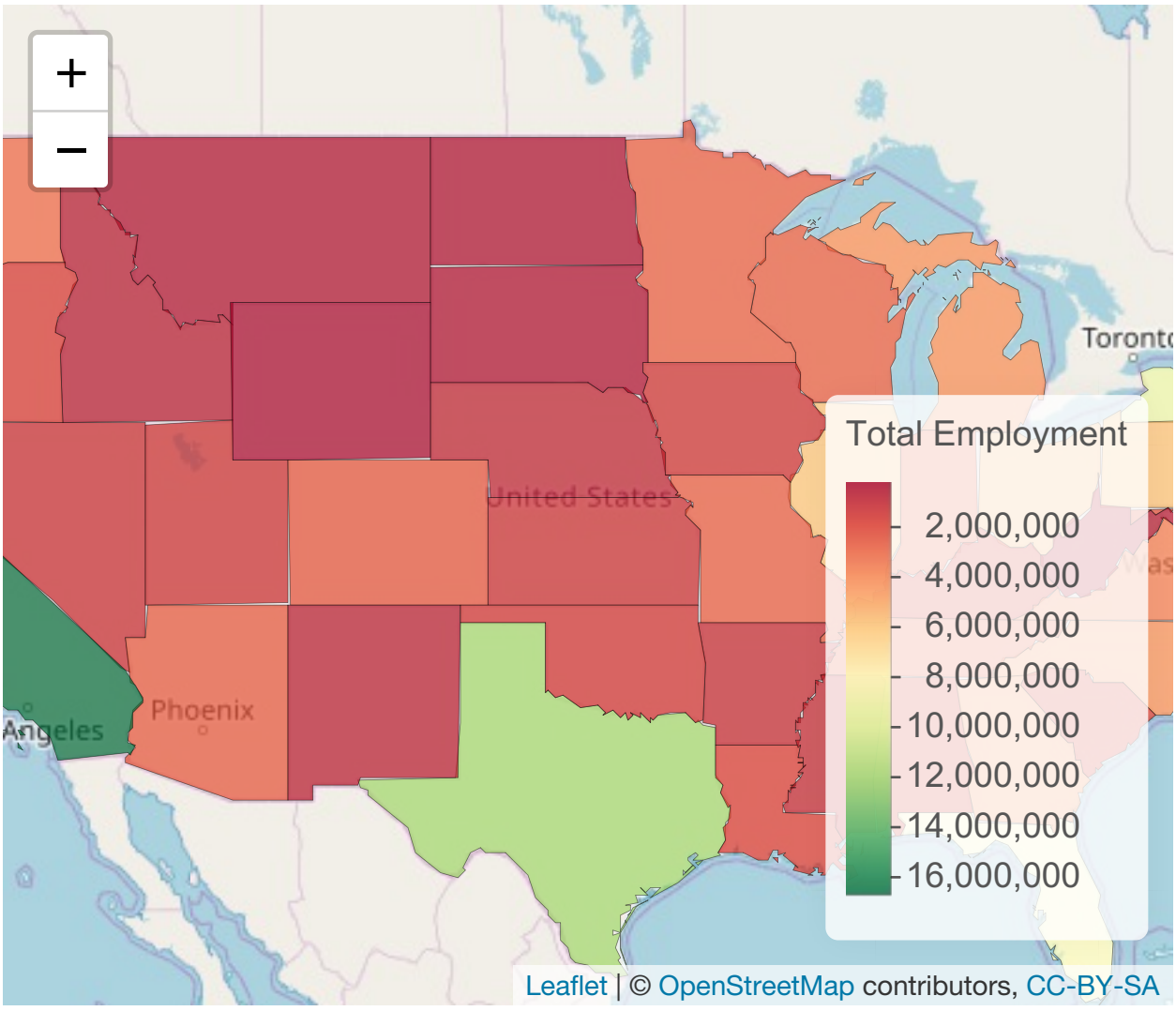
Finally, we build our map. The code below actually renders the Leaflet map. Notice the pipe `%>%` syntax similar to that used in tidyverse packages. The pipe functions the same way here feeding the results of a previous line of code to the next line. The code can be written without it, but you have to store the value of the map into an object and call it again on each subsequent line.

You start by defining your leaflet object and loading the data. You can also set your display settings. You then set the geographic center of the map and the zoom level using `setView()`. If you do not set the view, it will be set to the center of the data you are displaying with a wide enough zoom level to fit it all in. This can be problematic for state data because of Alaska and Hawaii.

You then add your basemap tiles. Below, I load the basic, default tiles, but there is a huge selection. See the Basemaps section below.

We then create the data layer(s). There can be as many data layers as you would like. They will be layered in the order you define them. You may find it useful to have points on top of a polygon map for example. Here is where we make use of the color palette and popup we defined earlier. We also define the legend using the color palette.

```
#DEFINE LEAFLET OBJECT AND DATA
leaflet(dat, width="100%", height="400px") %>%
      #SET VIEW AND ZOOM LEVEL (if necessary)
      setView(lng = -98.55, lat = 39.80, zoom = 4) %>%
      #ADD BASEMAP
      addTiles() %>%
      #ADD DATA TYPE (Points, Lines, Polygons, Irregular Shapes)
      addPolygons(weight = .3, color = "black", popup=popup,
                  fillColor = ~pal(tot_employment), fillOpacity = 0.7) %>%
      #ADD LEGEND
      addLegend("bottomright", pal = pal, values = ~tot_employment, na.label = "",
                title = "Total Employment", opacity = 0.8)
```



Leaflet Map Components

There are only four basic components you need to think about when creating your Leaflet map.

1. the basemap
2. the type of data you want to display and the color palette to use
3. the legend
4. the popup

Once you have these defined, everything else is just a variation on these core concepts.

Basemaps

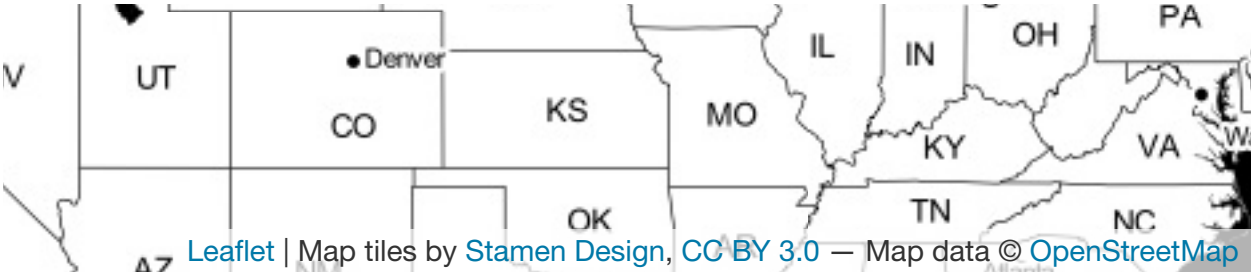
In the basic map built above, I used the default tile set where you simply call the function `addTiles()`. You can also use open-source map tiles provided by various companies and organizations by using the `addProviderTiles()` function instead and specifying which of these map types you would like.

There are many, many options for basemaps in Leaflet. Below is a small subset of what is available. These map tiles vary in their general color patterns and their level of geographic detail when you zoom in. Sometimes these colors and details can be distracting. You may find it useful to experiment a bit with this. I often use `addProviderTiles("CartoDB.Positron")` because it has muted colors and limited road and building clutter when zoomed in to the state and city level.

addTiles()

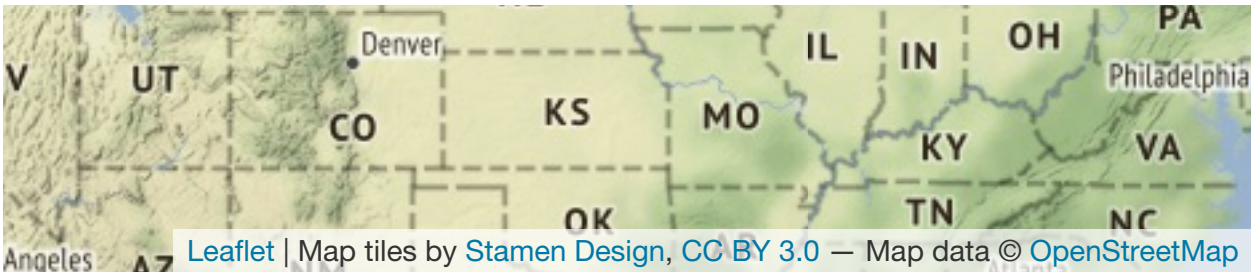
addProviderTiles("CartoDB.Positron")

addProviderTiles("Stamen.Toner")

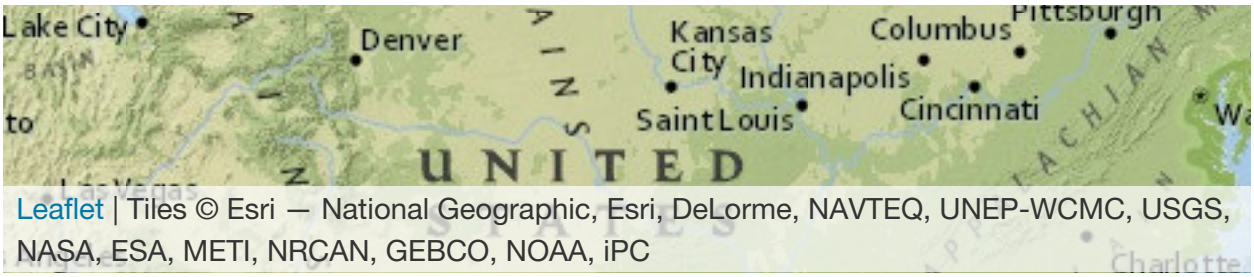


addProviderTiles("OpenStreetMap.BlackAndWhite")

addProviderTiles("Stamen.Terrain")



addProviderTiles("Esri.NatGeoWorldMap")



addProviderTiles("OpenTopoMap")

Data Layers and Palettes

So far we have only seen polygon maps (i.e., state shapes) but leaflet can handle many different types of geographic data. I will only cover the basic types of points, lines, polygons, and raster (i.e., irregular shapes).

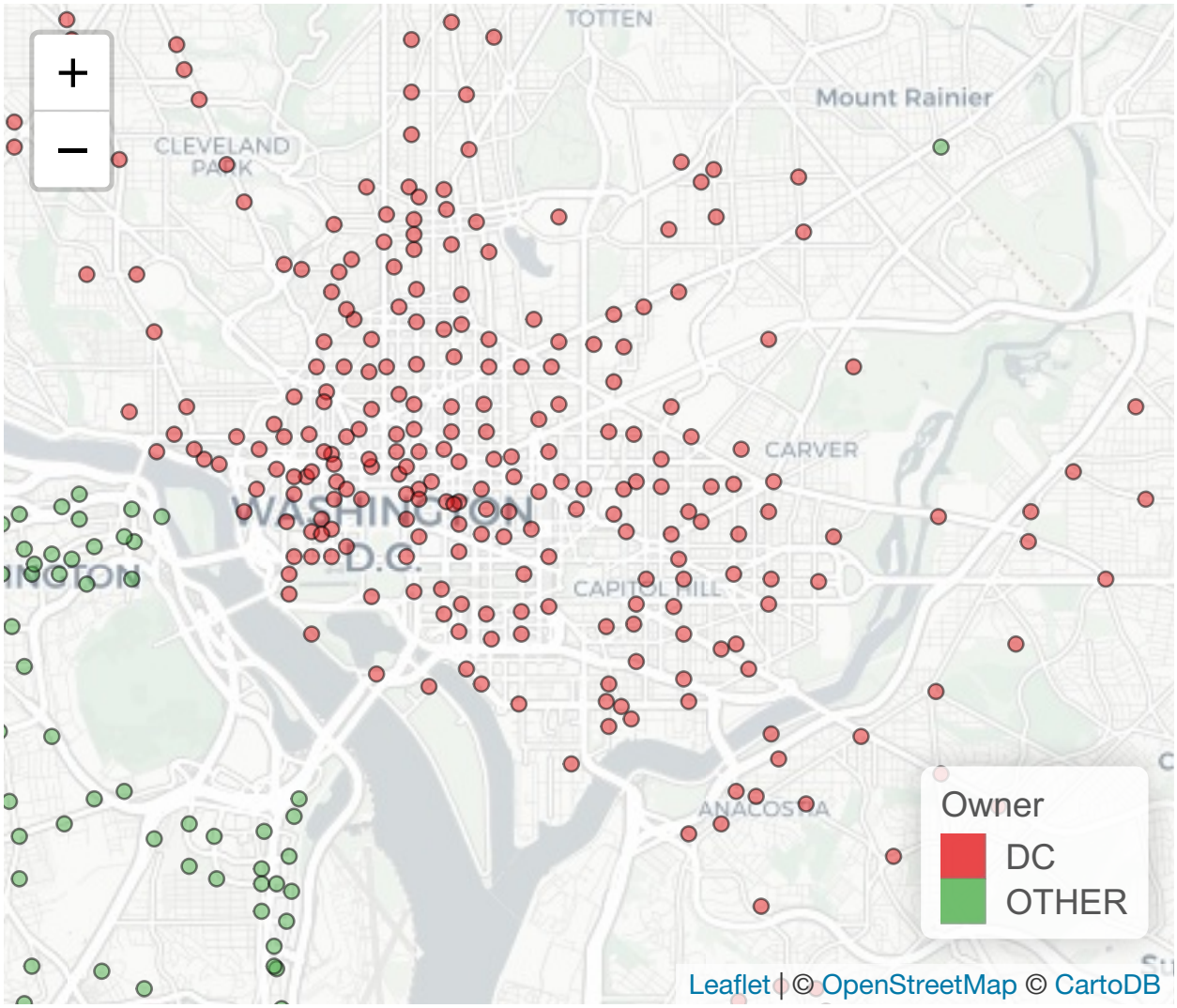
Points

If your dataset contains columns for latitude and longitude, you can map those as points in a leaflet map. You may use point data to label respondent locations (though not in a public map obviously) or locations of transportation options. For these types of data, you can read in the dataset as your normally would; as a CSV, Excel, etc.

For point data you can use one of the following functions; `addCircles()`, `addCircleMarkers()`, `addMarkers()`, or `addAwesomeMarkers()`.

The map below displays Capital Bikeshare locations around Washington, DC using `addCircleMarkers()`. Circle Markers are different from Circles in that they resize when the map is zoomed in and out. The map also uses a `colorFactor()` color palette. The only useful nominal variable in the dataset is the ownership of the location.

```
bikeshare <- read.csv("./data/Capital_Bike_Share_Locations.csv", stringsAsFactors = F)
bikeshare$OWNER[bikeshare$OWNER == ""] <- "OTHER"

pal <- colorFactor("Set1", domain=bikeshare$OWNER)

popup <-  with(bikeshare,paste(sep = "",
                               "<b><h4>",ADDRESS,"</h4></b>",
                               "<b>OWNER: </b> ",OWNER,"<br/>",
                               "<b># of Bikes: </b> ",NUMBER_OF_BIKES,"<br/>",
                               "<b># of Empty Docks: </b> ",NUMBER_OF_EMPTY_DOCKS,"<br/>"))

leaflet(bikeshare, width="100%", height="400px") %>%
      setView(-77.008332,38.898035, zoom = 12) %>%
      addProviderTiles("CartoDB.Positron") %>%
      addCircleMarkers(lng = ~LONGITUDE, lat = ~LATITUDE, radius = 3, popup= popup,
                       stroke=TRUE, weight=1, color="black", fillColor = ~pal(OWNER), fillOpacity = 0.5
      addLegend("bottomright", pal = pal, values = ~OWNER, title = "Owner", opacity = 0.8)
```



Lines

Line data are sequences of points. These you will have to load using `rgdal` into a SpatialLinesDataFrame. You may want to highlight roadways, public transportation lines, or some other line data using `addPolyLines()`. Even though some of these will appear on the basemap, it may still be useful to give them extra emphasis.

The map below uses `addPolyLines()` to plot WMATA Metro lines within the DC city limits. Some of the lines appear to be overlapping, but if you zoom in you can see that they are offset. Also different here is that I didn't use a formal color palette, but fed the line color to the color argument of the `addPolyLines()` function.

```
metro <- readOGR("./data/Metro_Lines.geojson", stringsAsFactors = F, verbose = F)
metro@data$color <- gsub(" .*","", metro@data$NAME)

leaflet(metro, width="100%", height="400px") %>%
 addProviderTiles("CartoDB.Positron") %>%
 addPolylines(color=~color, opacity=0.6)
```

Polygons

Polygons are similar to lines in that they are a sequence of points. The difference is that, for polygons, the last point in the sequence is the same as the first, leading to a closed loop. Packages like `leaflet` and `rgdal` then know to treat these sequence of points and the space within it as a single unit.

We have dealt with polygons before, so I will only reference them here to say that using `rgdal`, you can create a SpatialPolygonsDataFrame that contains the geospatial data for the polygons, but also a regular, rectangular dataset containing data about those polygons (e.g., the name of the area or any numeric data associated with it). Note that when a dataframe related to US States is read in below it says that there are 52 features (i.e., states and territories) and 10 fields (i.e., data elements about those features). Those data fields in the object `dat` can be accessed through `dat@data`.

```
dat <- readOGR("./data/states_geojson.geojson", "OGRGeoJSON")
```

```
## OGR data source with driver: GeoJSON
## Source: "[redacted]/R/leaflet-app/data/states_geojson.geojson", layer: "OGRGeoJS
## with 52 features
## It has 10 fields
```

```
printdat <- xtable::xtable(head(dat@data))
print(printdat, type="latex")
```

% latex table generated in R 3.5.0 by xtable 1.8-2 package % Mon Oct 22 14:39:13 2018

	GEO_ID	STATE	NAME	LSAD	CENSUSAREA	unemp_rate	unemp_rank	tot_employment
0	0400000US23	23	Maine		30842.92	3.30	15.00	599180.00
1	0400000US25	25	Massachusetts		7800.06	3.60	22.00	3528070.00
2	0400000US26	26	Michigan		56538.90	4.00	30.00	4276040.00
3	0400000US30	30	Montana		145545.80	3.60	22.00	460740.00
4	0400000US32	32	Nevada		109781.18	4.50	42.00	1310220.00
5	0400000US34	34	New Jersey		7354.22	4.20	38.00	4007470.00

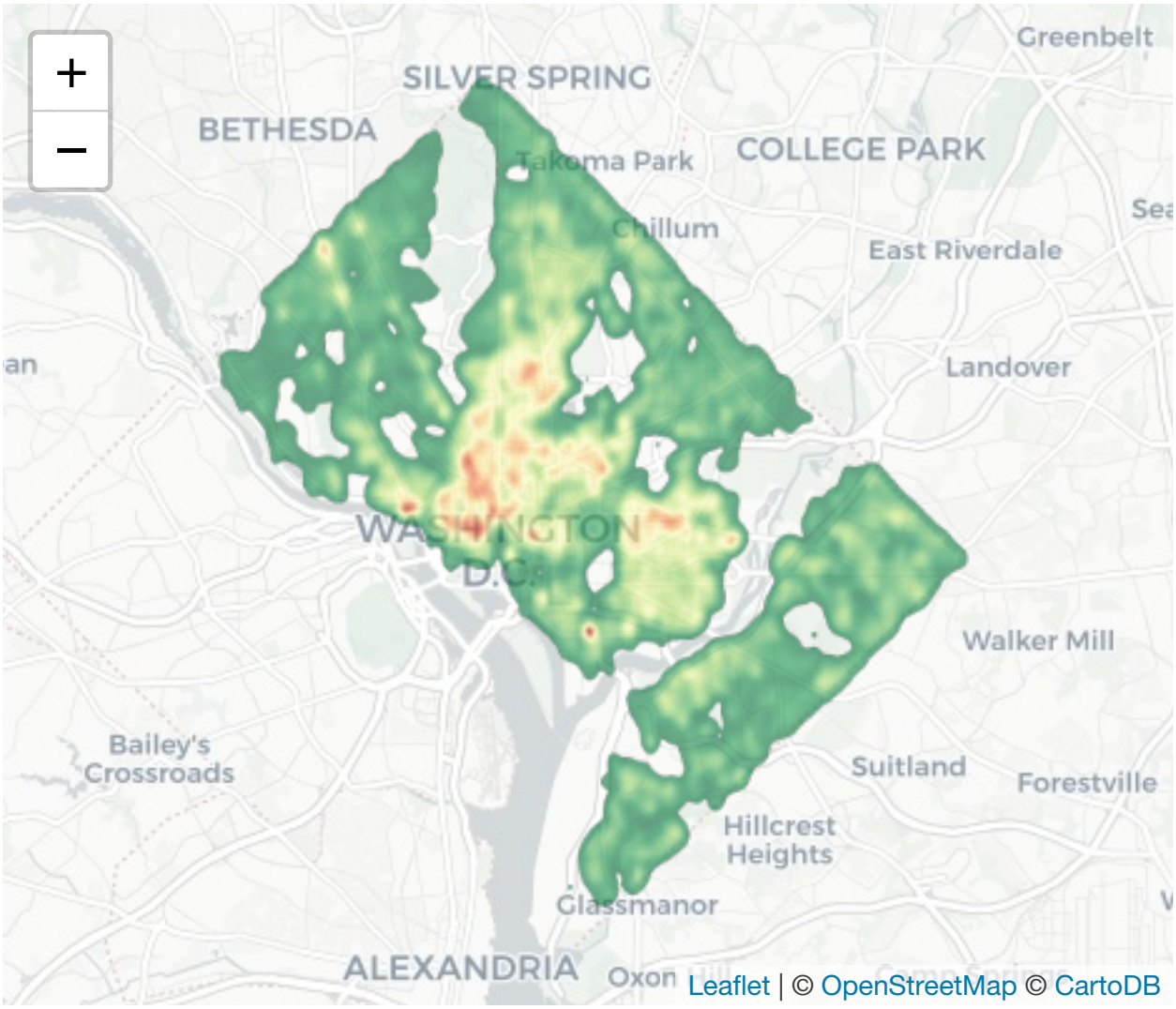
Raster

Raster images are irregular shapes that correpond to specific locations, but don't necessarily follow geographic boundaries. Think of how weather data are plotted on maps. The area where rainfall will be < 0.5 inches may have irregular boundaries and those boundaries will shift as iterations of that map are made. The most common use case I have found for the use of `addRasterImage()` is aggregating individual points into a heatmap.

The map below aggregates over 310,000 service requests received by the District of Columbia through the 311 phone number and through their online form. The locations are where the services (e.g., leaf collection, sidewalk shoveling, or noise complaints) were requested so the heatmap below could give city planners an idea of where to stage resources. How the service call points were aggregated into the raster layer below is not within the scope of this tutorial, but you can look into kernel density estimation using the `KernSmooth` package.

Also, the map below uses a Green to Yellow to Red color palette. The default order, with red where there were few points and green where there were many points didn't seem to fit the idea of a heatmap, thus using the `reverse=TRUE` argument to flip the color scheme so that red indicates more service calls in a narrowly defined area.

```
library(raster)
load("./data/DC311.RData")

pal <- colorNumeric(palette = "RdYlGn", domain = values(loc_density_raster), na.color = "transparent",r

leaflet(width="100%", height="400px") %>%
  addProviderTiles("CartoDB.Positron") %>%
  addRasterImage(x = loc_density_raster, colors=pal, opacity = 0.6, project = FALSE)
```



Legends

Displaying data with a color variable doesn't make sense without some means to differentiate between those color categories. You can add one more more legends to your Leaflet map using the `addLegend()` function. You then need to feed your color palette and the values that correspond to the color palette. Leaflet then maps the color scheme to the values and displays them in the corner of the map that you specify.

```
# EXAMPLE OF Color Palette
pal <- colorFactor("Set1", domain=bikeshare$OWNER)

# EXAMPLE OF ADD LEGEND FUNCTION
addLegend("bottomright", pal = pal, values = ~OWNER, title = "Owner", opacity = 0.8)
```

Popups and Hovers

One of the greatest advantages of Leaflet maps over static maps that you can create using ggplot or other packages is the hover and popup functionality. Popups allow the users to click on a point, line, or polygon and get information. You can provide whatever information you want and you have extensive options for visually formatting what you display. Hovers (referred to in Leaflet as labels) are essentially the same thing

except they display when the user hovers over the geographic element. You can display both popups and hovers in the same map or just one.

The information displayed in the hover or popup needs to somehow be linked to the geographic element you want to display data for. Typically this means that the data appear in the same row of a data table as the latitude and longitude of a point (as shown in the table below) or that the data are in the corresponding row of the embedded dataframe within a SpatialPolygonsDataFrame or SpatialLinesDataFrame. You access the elements in the same way as you would any column within that dataset.

% latex table generated in R 3.5.0 by xtable 1.8-2 package % Mon Oct 22 14:39:19 2018

	LATITUDE	LONGITUDE	ADDRESS	OWNER	NUMBER_OF_BIKES	NUMBE
1	38.87	-77.11	Pershing & N George Mason Dr	OTHER	4	
2	38.99	-77.02	Fenton St & New York Ave	OTHER	5	
3	38.98	-77.10	Bethesda Ave & Arlington Rd	OTHER	6	
4	38.98	-77.09	Montgomery Ave & Waverly St	OTHER	6	
5	39.10	-77.19	Fallsgrove Blvd & Fallsgrove Dr	OTHER	14	
6	39.09	-77.20	Traville Gateway Dr & Gudelsky Dr	OTHER	7	

You can format the data displayed in the hover or popup using HTML and CSS. Here you are only limited by your knowledge of HTML & CSS syntax. You can create data tables, you can display images or plots, and you can use visual formatting to draw users attention where you want it.

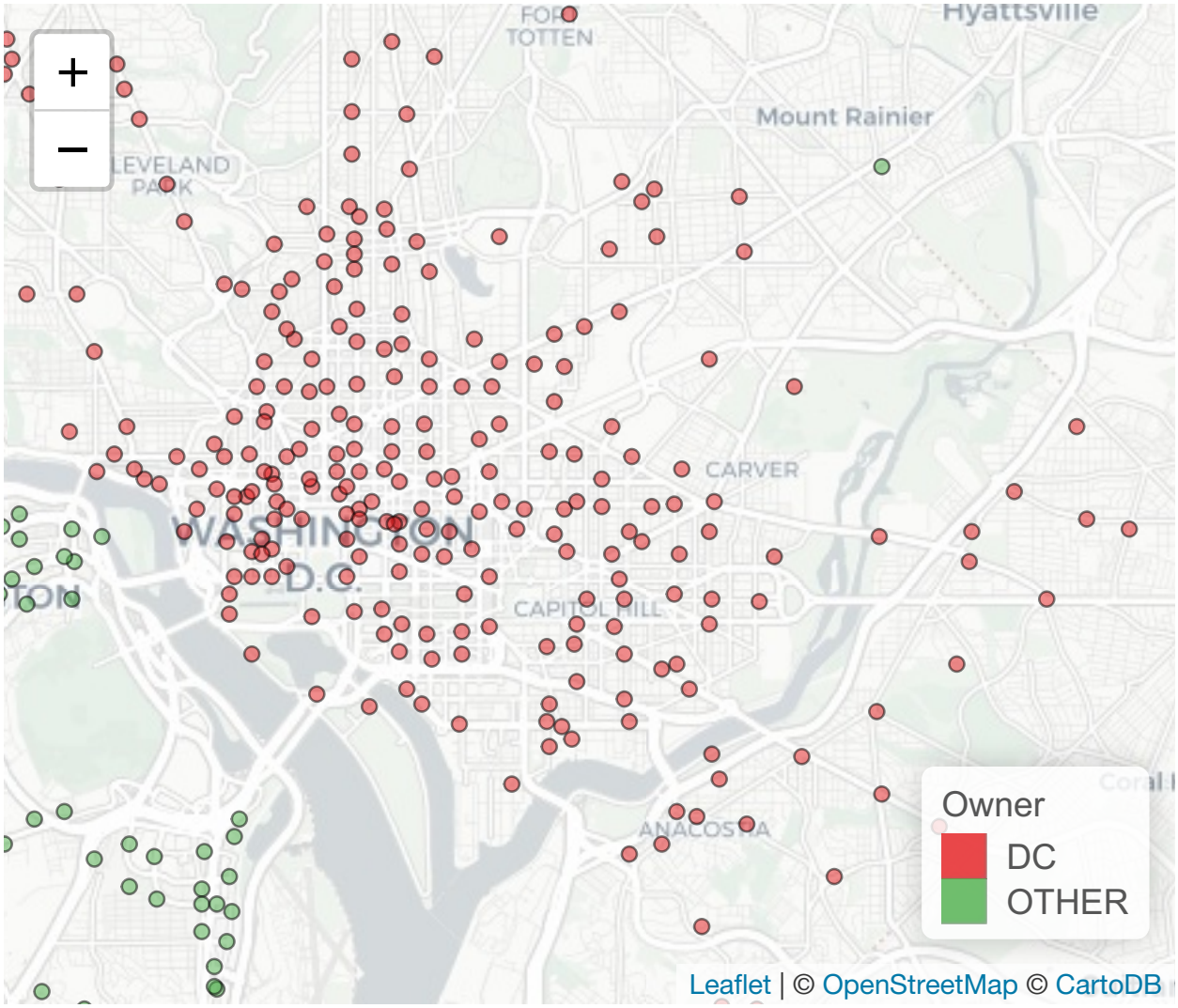
To use popups and hovers you must first define them. Below are simple examples of a hover and a popup. Notice the`<b></b>` tags for bold text. The ADDRESS row also has some additional CSS specifying the font should be one and a half times as large. Hovers are defined similarly. I often used hovers to display an abbreviated set of information from the popup; usually whatever is being displayed by the color variable.

```
#EXAMPLE OF DEFINING A POPUP
popup <-  with(bikeshare,paste(sep = "",
                               "<b style='font-size:1.5em'>",ADDRESS,"</b><br>",
                               "<hr style='margin:0px'>",
                               "<b>Owner: </b> ",OWNER,"<br/>",
                               "<b># of Bikes: </b> ",NUMBER_OF_BIKES,"<br>",
                               "<b># of Empty Docks: </b> ",NUMBER_OF_EMPTY_DOCKS,"<br>"))

#EXAMPLE OF DEFINING A HOVER
hover <- with(bikeshare,paste(sep = "",
                               "<b style='font-size:1.5em'>",ADDRESS,"</b><br>",
                               "<b>Owner:</b> ",OWNER,"<br/>"))
```

In order to use the popups and hovers you must use the `popup` and `label` arguments within the data layer function. Here within `addCircleMarkers()`, we say `popup = popup` in order to feed in the popup object we defined above. The hover is fed in using `label=lapply(hover, HTML)` because, for some reason, the HTML functioning does not work by default for the labels. You must loop through each element of the hover list and apply the `HTML` function. The `HTML()` function is part of the `htmltools` package so that needs to be called as well.

```
library(htmltools)

pal <- colorFactor("Set1", domain=bikeshare$OWNER)

leaflet(bikeshare, width="100%", height="400px") %>%
        setView(-77,38.90, zoom = 12) %>%
        addProviderTiles("CartoDB.Positron") %>%
        addCircleMarkers(lng = ~LONGITUDE, lat = ~LATITUDE, radius = 3, popup= popup, label = lapply(ho
```

```
                        stroke=TRUE, weight=1, color="black", fillColor = ~pal(OWNER), fillOpacity = 0
      addLegend("bottomright", pal = pal, values = ~OWNER, title = "Owner", opacity = 0.8)
```

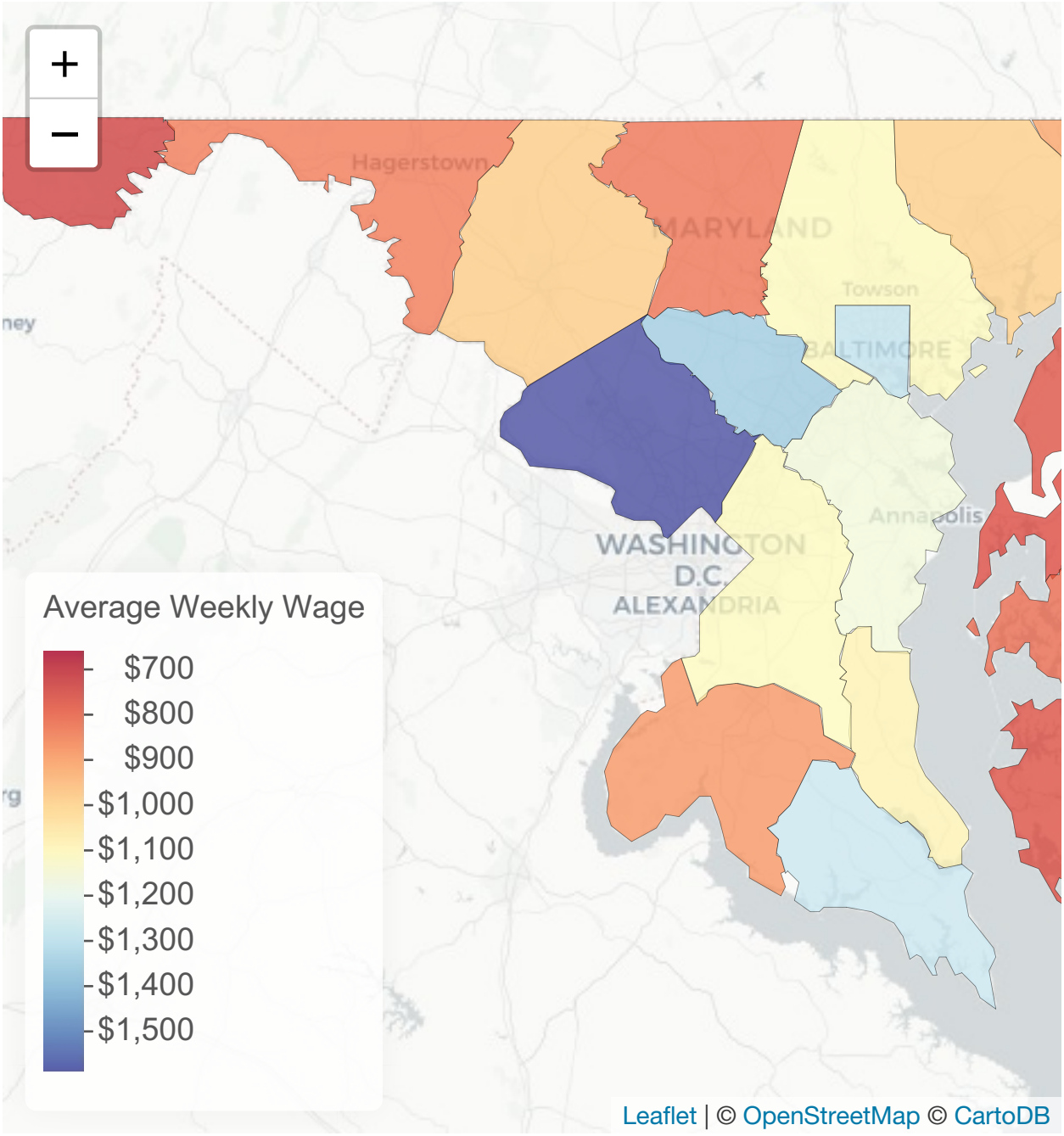


Putting It All Together

I wanted to close by giving you the whole process in one place. I added features to this map like including both a hover and a popup, extensivly formatting the popup, and making use of the highlighting options.

```
dat <- readOGR("./data/md_county_geojson.geojson", "OGRGeoJSON", verbose=F)

pal <- colorNumeric("RdYlBu", domain = dat@data$avg_wkly_wage, na.color = "gray35")

hover <-  with(dat@data,paste(sep = "",
                              "<b style='font-size:2em'> ",NAME,"</b><br>",
                              "<b>Average Weekly Wage:</b>  $",avg_wkly_wage))

popup <-  with(dat@data,paste(sep = "",
                "<img src='https://brandonkopp.com/wp-content/uploads/2017/10/BLS_emblem_2016.png' style
                "<b style='font-size:2em;line-height:45px;margin-left:5px'> ",NAME,"</b>",
```

```
                "<hr style='margin:5px'>",
                "<table style='font-size:1.2em; width:250px'>",
                "<tr><td><b># of Establishments:</b></td><td>",format(qtrly_estabs, big.mark = ","),"</
                "<tr><td><b>Employment Level:</b></td><td>",format(month3_emplvl, big.mark = ","),"</td
                "<tr><td><b>Average Weekly Wage:</b></td><td>$",avg_wkly_wage,"</td><tr>",
                "<tr><td><b>OTY Wage Change:</b></td><td>", oty_avg_wkly_wage_pct_chg,"%</td><tr>",
                "</table>"))

leaflet(dat, width="100%",height="500px") %>%
  setView(lng=-77.45,lat=38.88,zoom=8) %>%
  addProviderTiles("CartoDB.Positron") %>%
  addPolygons(weight = .3 ,color = "black", popup=popup, label=lapply(hover, HTML),
              fillColor = ~pal(avg_wkly_wage),fillOpacity = 0.7,
              highlightOptions = highlightOptions(weight=3,opacity = 1,color="white")) %>%
  addLegend("bottomleft", pal = pal, values = ~avg_wkly_wage, na.label = "",
            title = "Average Weekly Wage", opacity = 0.8, labFormat = labelFormat(prefix = "$"))
```



Example Leaflet Applications

- OES Data Mapping Application (Elizabeth Cross) - Displays state and MSA level employment and wage data from BLS's Occupational Employment Statistics program.
- American Community Survey API Application (Brandon Kopp) - Displays state, county, and congressional district level data on various demographics and housing characteristics accessed through the Census Bureau's ACS API.
- Montgomery Count Crime Explorer (Brandon Kopp) - Displays near real-time data of crime data accessed through Montgomery County, Maryland's open data portal.