Introduction

This is part 3 of a 4-part series on how to build maps using R.

1. How to load geospatial data into your workspace and prepare it for visualization
2. How to make static maps using ggplot2
3. How to make interactive maps (pan, zoom, click) using leaflet
4. How to add interactive maps to a Shiny dashboard

In this post, we will learn how to make interactive maps using the `leaflet` package. `leaflet` is an R package that makes it easy for R coders to create Leaflet JavaScript maps. The **benefit** of creating a JavaScript map over a .jpg map as we did in our last post is that the map is "slippy," that is, it slips around inside its container. You can drag to pan, scroll to zoom, click to show popups, etc. The **downside,** however, is that, since `leaflet` creates a JavaScript map, the map can only be shared in an interactive environment like a web browser. As such, `leaflet` is not a good choice for pasting images in papers and presentations, or for setting a snazzy new desktop background. Go back to Maps, Part 2 for that.

Why Leaflet over other options? To quote from Leaflet's website, "Leaflet is the leading open-source JavaScript library for mobile-friendly interactive maps…Leaflet is designed with simplicity, performance and usability in mind. It works efficiently across all major desktop and mobile platforms, can be extended with lots of plugins, has a beautiful, easy to use and well-documented API and a simple, readable source code that is a joy to contribute to."

I can't speak to how joyous contributing to their source code is (especially since I know no JavaScript), but I can attest that Leaflet's documentation–including their documentation of their R package–is clear and comprehensive. More importantly for the R user, though, the implementation of the `leaflet` package is clean and highly reminiscent of `ggplot2`. These three reasons–the power of the underlying JS library, the comprehensive R documentation, and the familiar R framework–make `leaflet` an obvious choice for the R data analyst.

One important note: *This post does not embed actual Leaflet maps. Getting R and Squarespace to work together is not easy (might actually be impossible?). Instead, you will see video demos of the maps in action. Head over to the Leaflet website if you would like to experiment with Leaflet maps yourself. Or, download the R code used in this post and run it yourself!*

Review: Load data

Below is code to load two datasets for visual analysis in the rest of the post. The first dataset is a .geojson file containing geospatial descriptions of Philadelphia's neighborhoods, courtesy of OpenDataPhilly. This dataset is polygon data and will form our basemap for layering on additional, more interesting, features. (We used this data in our last post, too.)

The second dataset is geospatial point data, also provided by OpenDataPhilly, that contains information from the police department on shooting victims. This is a sobering dataset that allows city residents to see location information for shootings, basic demographic information about shooting victims, and trend the city's gun violence over time.

This dataset is accessed through an call to an API. For those unfamiliar with this type of API, here is a brief introduction.

- To use the API, we need to ping to a particular URL. The API in response, will send us a file to download. This is similar to you clicking a website link that opens a tab which downloads a file
- The API knows what information to put in the file based on the URL we decided to ping

    - The URL for all API calls to the Philly cartographic API start with the same base form (https://phl.carto.com/api/v2/sql) and then will append a brief bit of sql. You do not need vast sql knowledge here. Our query for this project is as simple as “select * from shootings where year > 2018.” The table of data we wish to query is called “shootings,” we want to filter based on a column in that table called “year,” and we want all data (the asterisk is shorthand for “all columns”)
    - We also need to append information to our base URL that tells the API we would like a .geojson file
- The end result is the following URL: https://phl.carto.com/api/v2/sql?q=%0A%20%20select%20%2A%0A%20%20from%20shootings%0A%20%20where%20year%20%3E%202018%0A&format=GeoJSON. Try clicking it here, and you will see your browser download a file for you
- In R, we use `httr::modify_url()` to create our URL and `sf::read_sf()` to download the .geojson file and load it into R as a simple features object

```
# SETUP         #### #### #### #### #### ####
library(dplyr)
library(httr)
library(sf)

# LOAD DATA     #### #### #### #### #### ####
## ## Simple features data: Philadelphia neighborhoods
# Source: OpenDataPhilly. https://www.opendataphilly.org/dataset/philadelphia-neighborhoods
neighborhoods_geojson <- "https://raw.githubusercontent.com/azavea/geo-data/master
/Neighborhoods_Philadelphia/Neighborhoods_Philadelphia.geojson"
neighborhoods_raw <- sf::read_sf(neighborhoods_geojson)

head(neighborhoods_raw)
#> Simple feature collection with 6 features and 8 fields
#> geometry type:  MULTIPOLYGON
#> dimension:      XY
#> bbox:           xmin: -75.28027 ymin: 39.96271 xmax: -75.01684 ymax:
40.09464
#> geographic CRS: WGS 84
#> # A tibble: 6 x 9
#>   name  listname mapname shape_leng shape_area cartodb_id created_at
#>
#> 1 PENN~ Pennypa~ Pennyp~     87084.  60140756.          9 2013-03-19
13:41:50
#> 2 OVER~ Overbro~ Overbr~     57005.  76924995.        138 2013-03-19
13:41:50
#> 3 GERM~ Germant~ Southw~     14881.  14418666.         59 2013-03-19
13:41:50
#> 4 EAST~ East Pa~ East P~     10886.   4231000.        129 2013-03-19
13:41:50
#> 5 GERM~ Germany~ German~     13042.   6949968.         49 2013-03-19
13:41:50
#> 6 MOUN~ Mount A~ East M~     28846.  43152470.          6 2013-03-19
13:41:50
#> # ... with 2 more variables: updated_at , geometry

## ## Simple features data: Philadelphia shootings
# Source: OpenDataPhilly. https://www.opendataphilly.org/dataset/shooting-victims
base_url <- "https://phl.carto.com/api/v2/sql"
```

```
q <- "
  select *
  from shootings
  where year > 2018
"
shootings_geoJSON <-
  httr::modify_url(
    url = base_url,
    query = list(q = q, format = "GeoJSON")
  )
shootings_raw <- sf::read_sf(shootings_geoJSON)

head(shootings_raw)
#> Simple feature collection with 6 features and 22 fields
#> geometry type:  POINT
#> dimension:      XY
#> bbox:           xmin: -75.19185 ymin: 39.90638 xmax: -75.05941 ymax:
40.0475
#> geographic CRS: WGS 84
#> # A tibble: 6 x 23
#>   cartodb_id objectid  year dc_key code  date_               time  race
sex
#>
#> 1       3164     5114  2019 20190~ 0411  2019-02-19 19:00:00 17:4~ B     M
#> 2       3168     5118  2019 20190~ 0111  2019-03-29 20:00:00 21:5~ W     M
#> 3       3169     5119  2019 20190~ 0111  2019-04-09 20:00:00 11:5~ B     M
#> 4       3183     5133  2019 20190~ 0111  2019-09-27 20:00:00 16:5~ B     M
#> 5       3185     5135  2019 20190~ 0411  2019-05-31 20:00:00 12:1~ B     M
#> 6       3188     5138  2019 20190~ 0411  2019-02-10 19:00:00 20:3~ W     M
#> # ... with 14 more variables: age , wound , officer_involved ,
#> #   offender_injured , offender_deceased , location ,
#> #   latino , point_x , point_y , dist , inside ,
#> #   outside , fatal , geometry
```

Review: Clean data

Next, we will do some basic data cleaning. For the `neighborhoods` dataset, we will drop and rename columns. In the `shootings` dataset, we will remove points that have latitude and longitude in Florida. More cleanup to this dataset will come later once we have started making our maps.

Note from above that both of the datasets are already in the WGS 84 CRS. `leaflet` requires that data be in WGS 84, so we would need to convert to WGS 84 (EPSG code: 4326) using `sf::st_transform(shootings, crs = 4326)` if it weren't provided to us with that CRS.

If we want to run non-geospatial analysis on our shootings data, such as plotting shootings over time, calculating totals by demographic, and so on, we can drop the geospatial information and work with a standard tibble using `sf::st_drop_geometry(shootings)`.

```
# CLEAN DATA   #### #### #### #### #### ####
neighborhoods <- neighborhoods_raw %>%
  dplyr::select(label = mapname)

shootings <- shootings_raw %>%
  dplyr::filter(point_x > -80 & point_y > 25) # points in FL
```

Geospatial layers in leaflet

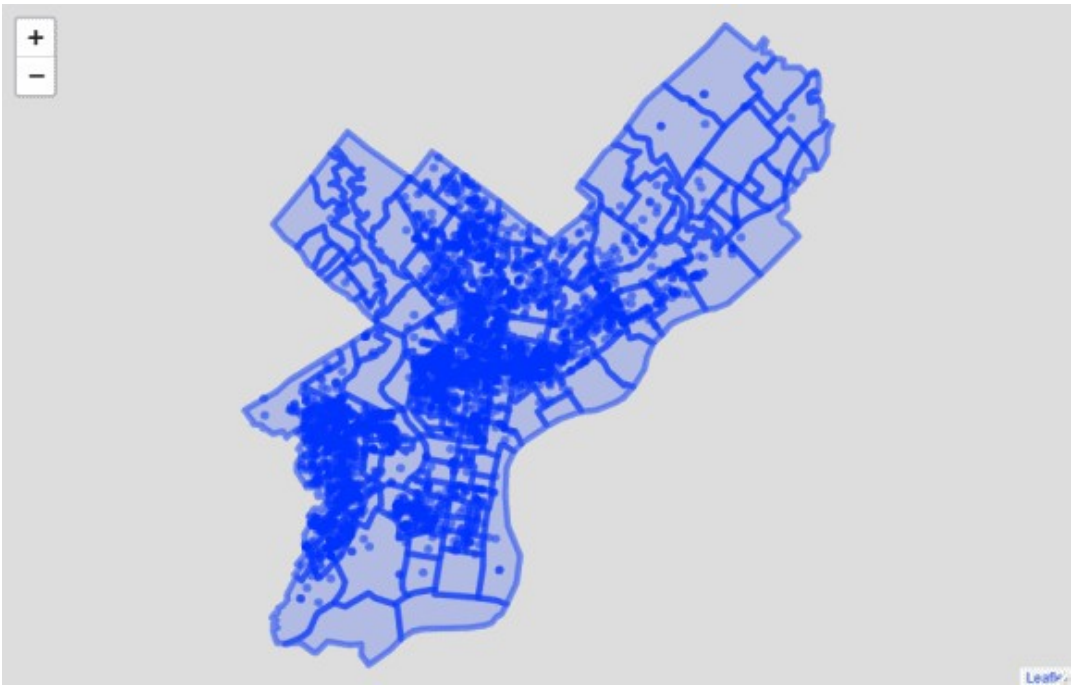
The concepts of loading and mapping various layers of data in `leaflet` are similar to what we had seen in `ggplot2`. You have the option of loading data either as the `data = ...` argument in `leaflet::leaflet()` or waiting until subsequent layers to provide the data. As in our last post, we will add the data in each layer, since we are working with two distinct datasets.

The examples below will walk you through making maps in `leaflet`, starting with the most basic map and building the complexity from there. Below each piece of code you will find a static image of the map. To interact with the map (as it was intended!), run the code chunks or download the R code in its entirety.

Your first map

```
# ANALYZE      #### #### #### #### #### ####
library(leaflet)

leaflet::leaflet() %>%
  leaflet::addPolygons(data = neighborhoods) %>%
  leaflet::addCircles(data = shootings)
```

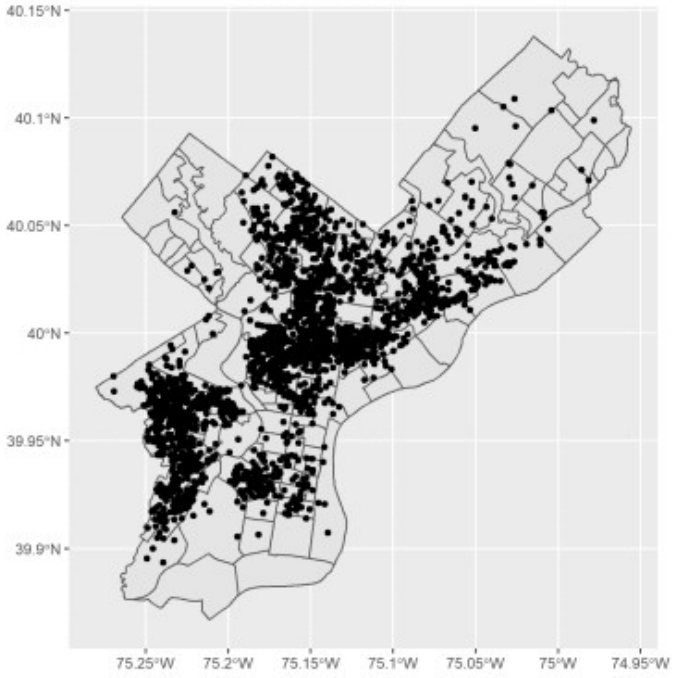


Basic map of Philadelphia gun violence (leaflet; this is a static screenshot of an interactive map). Source: OpenDataPhilly

This should look very similar to what we would have written for `ggplot2`! The only difference in code is that we have to specify if we want to add Polygons or Circles, and we replace the `+` with `%>%`.

```
library(ggplot2)

ggplot2::ggplot() +
  ggplot2::geom_sf(data = neighborhoods) +
  ggplot2::geom_sf(data = shootings)
```

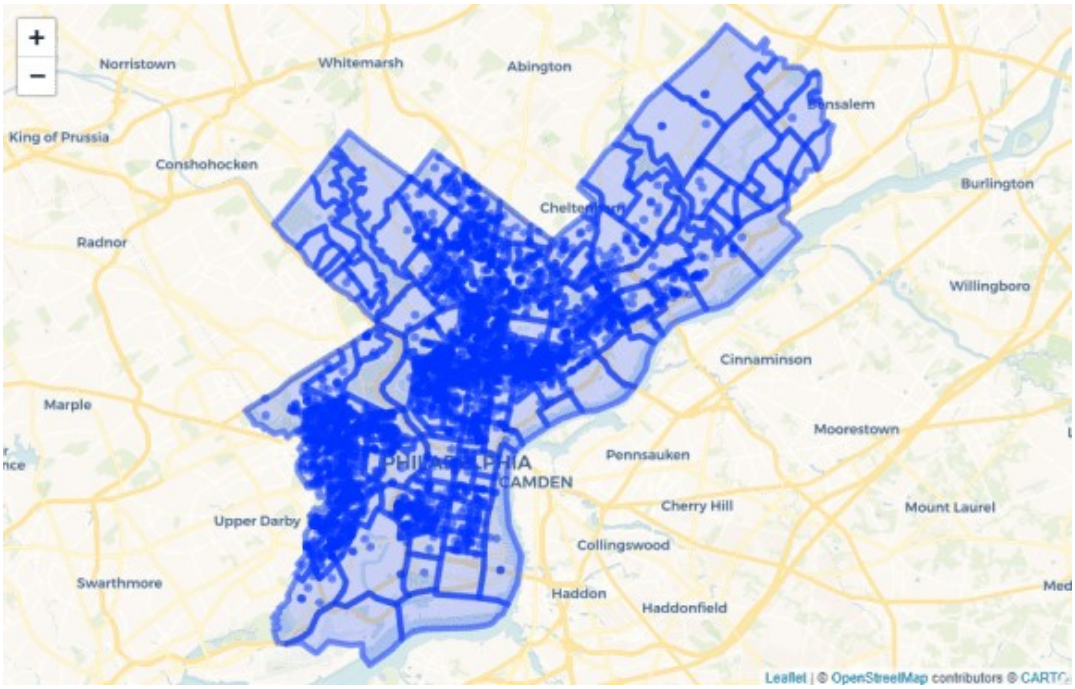


Basic map of Philadelphia gun violence (ggplot2). Source: OpenDataPhilly

Add a basemap

The `leaflet` package makes it easy to add map tiles, or "basemaps" to the layperson. You can either choose to call `addTiles()` with no arguments to get the default basemap from OpenStreetMap or choose to call `addProviderTiles()` to get one of the various third-party options. Our favorite is CartoDB.Voyager, but you can explore the entire set of options and pick your favorite.

```
leaflet::leaflet() %>%
  leaflet::addProviderTiles(providers$CartoDB.Voyager) %>%
  leaflet::addPolygons(data = neighborhoods) %>%
  leaflet::addCircles(data = shootings)
```



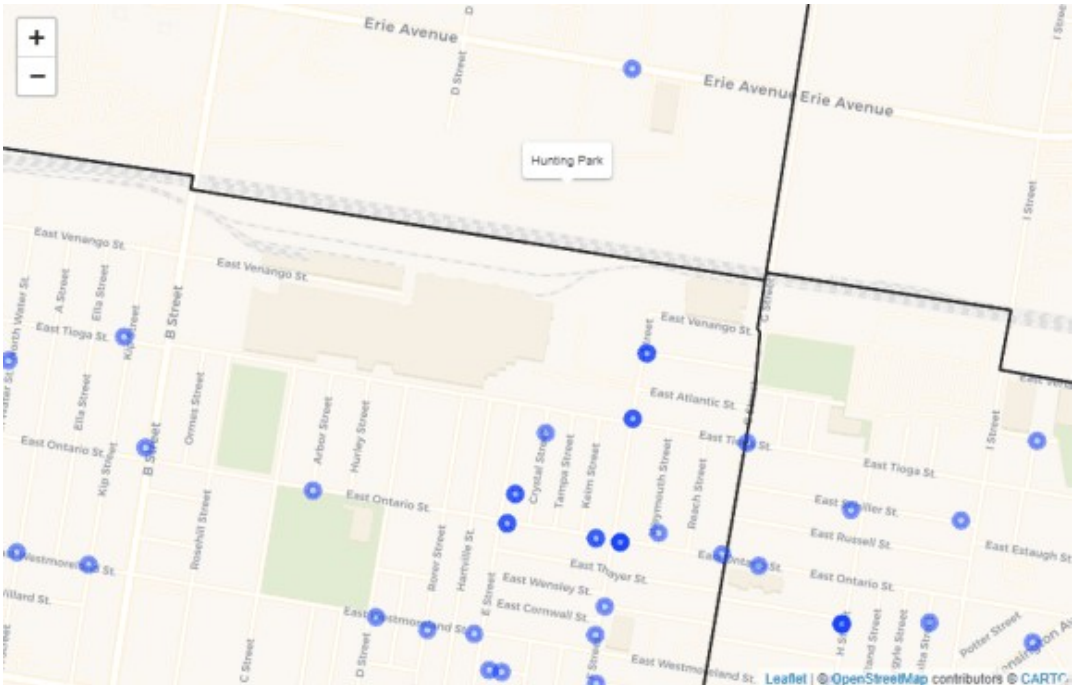
Leaflet map with provider tiles (this is a static screenshot of an interactive map). Source: OpenDataPhilly

Simple formatting adjustments

The basemap helps with the visual aesthetic, but we still have a long way to go. Let's make some basic formatting adjustments to the polygons layer: line color, line weight, line opacity, and fill opacity (0 = no fill). We'll also add a label, which will appear upon hover. Notice that we need to add `htmltools::htmlEscape()`. Leaflet recommends escaping HTML text for security reasons in situations where labels and popups might contain unwanted HTML content. We zoomed in on the map before taking a screenshot so that you can see the points in more detail.

```
library(htmltools)

leaflet::leaflet() %>%
  leaflet::addProviderTiles(providers$CartoDB.Voyager) %>%
  leaflet::addPolygons(
    color = "#222", weight = 2, opacity = 1, fillOpacity = 0,
    label = ~lapply(label, htmltools::htmlEscape),
    labelOptions = leaflet::labelOptions(direction = "top"),
    data = neighborhoods
  ) %>%
  leaflet::addCircles(data = shootings)
```



Zoomed region of Leaflet map, showing hover label (this is a static screenshot of an interactive map). Source: OpenDataPhilly

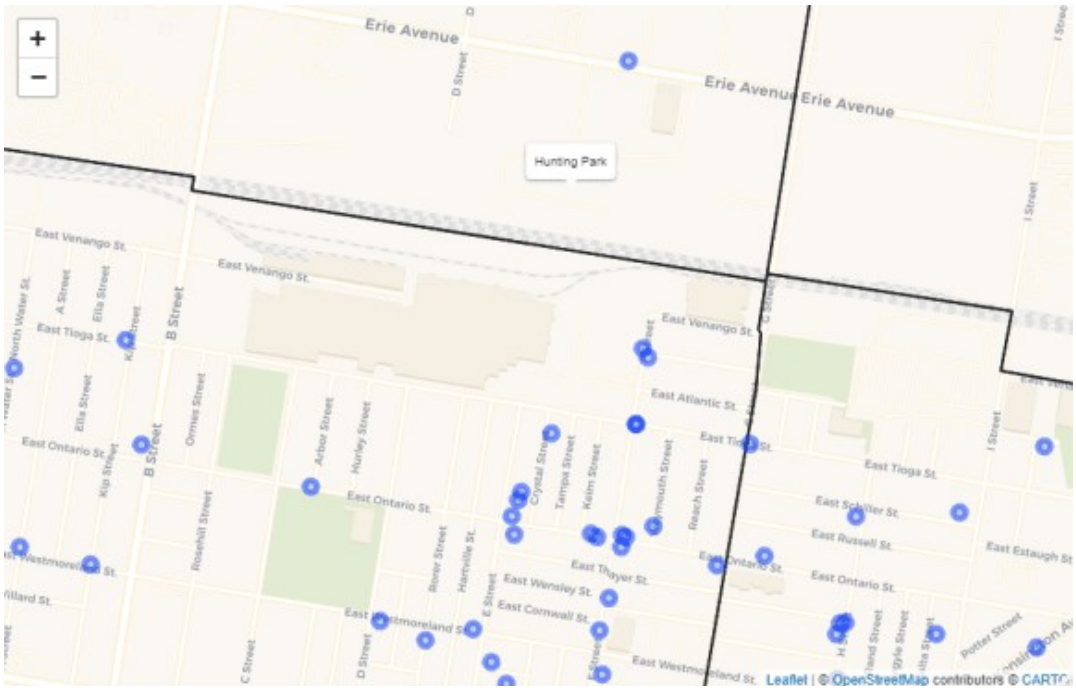
Jitter points so that we can see them more clearly

The `shootings` datast is only as precise as the block on which the event happened. Multiple shootings on the same block result in overlapping points. You may have seen that some of the points in the plot above were darker than others. This is because we had overlapped multiple translucent circles. To avoid this issue, we will "jitter" our points, adding a small amount of random displacement in the x- and y-directions. To make this jitter consistent each time you render the plot, remember to set the seed value for the random jitter using `set.seed()`.

```
set.seed(1776)

shootings <- sf::st_jitter(shootings, factor = 0.0004)
```

```
leaflet::leaflet() %>%
  leaflet::addProviderTiles(providers$CartoDB.Voyager) %>%
  leaflet::addPolygons(
    color = "#222", weight = 2, opacity = 1, fillOpacity = 0,
    label = ~lapply(label, htmltools::htmlEscape),
    labelOptions = leaflet::labelOptions(direction = "top"),
    data = neighborhoods
  ) %>%
  leaflet::addCircles(data = shootings)
```



Zoomed region of Leaflet map, with jittered points (this is a static screenshot of an interactive map). Source: OpenDataPhilly

Add labels for clearer communication

Our final set of aesthetic changes will be to our point layer. We add two new variables to our `shootings` dataset: a “color” variable that encodes encodes the “fatal” variable into red and grey, and a “popup” variable that summarizes key information about each shooting. This popup variable will appear in our map when we click on a point. In `leaflet`, labels appear upon hover, and popups appear upon click. Our popup variable contains html. Here is a quick translation for those unfamiliar with html: and mean to start and end a section of bold text, and mean to start and end a section of italics, and
means to add a line break. With some creative combinations of these html tools, we can create a simple and effective popup box.

Adding the color and popup variables to our plot is as simple as specifing `color = ~color` and `popup = ~popup` inside our `addCircles()` function call.

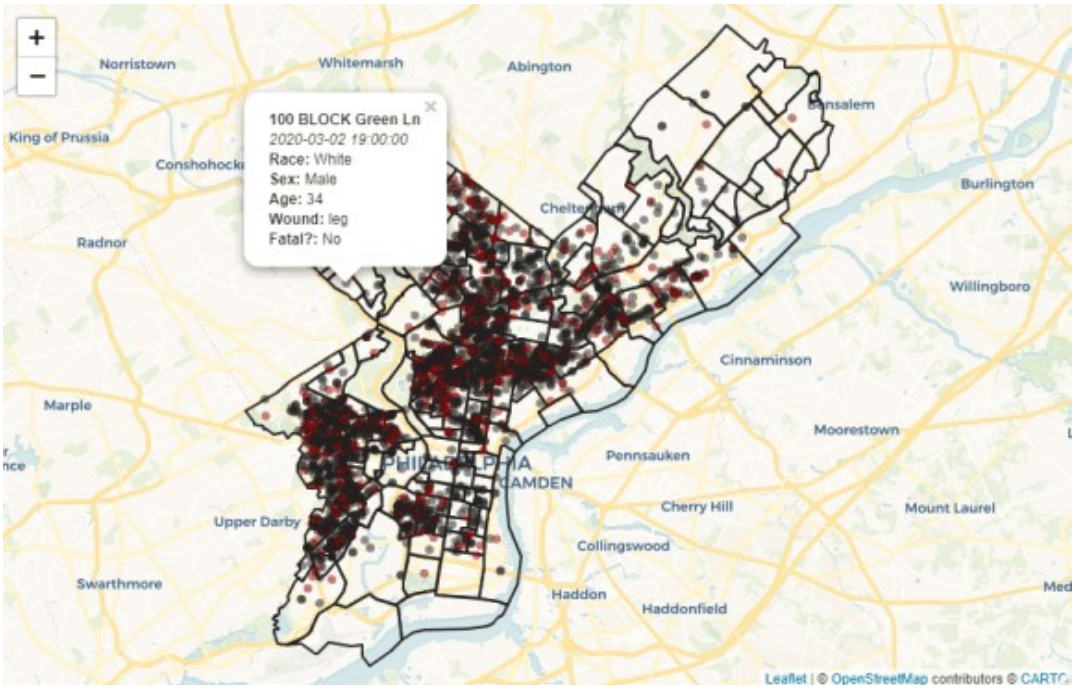
```
shootings <- shootings %>%
  dplyr::mutate(
    color = dplyr::if_else(fatal == 1, "#900", "#222"),
    popup = paste0(
      "", location, "",
      "
", date_, "",
      "
Race: ", dplyr::case_when(
        race == "B" ~ "Black",
```

```
        race == "W" ~ "White",
        TRUE ~ "NA"
      ),
      "
<b>Sex:</b> ", dplyr::case_when(
        sex == "M" ~ "Male",
        sex == "F" ~ "Female",
        TRUE ~ "NA"
      ),
      "
<b>Age:</b> ", age,
      "
<b>Wound:</b> ", wound,
      "
<b>Fatal?:</b> ", dplyr::case_when(
        fatal == 1 ~ "Yes",
        fatal == 0 ~ "No",
        TRUE ~ "NA"
      )
    )
  ) %>%
  dplyr::select(color, popup)

leaflet::leaflet() %>%
  leaflet::addProviderTiles(providers$CartoDB.Voyager) %>%
  leaflet::addPolygons(
    color = "#222", weight = 2, opacity = 1, fillOpacity = 0,
    label = ~lapply(label, htmltools::htmlEscape),
    labelOptions = leaflet::labelOptions(direction = "top"),
    data = neighborhoods
  ) %>%
  leaflet::addCircles(
    color = ~color, popup = ~popup,
    data = shootings
  )
```

Zoomed region of final Leaflet map, showing popup (this is a static screenshot of an

interactive map). Source: OpenDataPhilly



Final Leaflet map, showing popup (this is a static screenshot of an interactive map). Source: OpenDataPhilly

Choropleths in leaflet

Choropleths–maps in which each region is colored according to a summary statistic–are a powerful way to visualize data. In this example, let us suppost that we would like to show the total number of shootings in each neighborhood. This is a six-step process:

1. Join polygon data (`neighborhoods`) with point data (`shootings`) using `sf::st_join()`
2. Group by the ID of the polygon data (`neighborhoods$label`) and summarize our metric of interest (`n()`, but other situations could use count per square mile, maximum, minimum, average, etc.)
3. (Optional) Filter based on the summary statistics
4. (Optional) Create labels / popups
5. Save as a new simple features object
6. Create a palette function to instruct `leaflet` on how to color the regions

The final step–the creation of a palette function–is somewhat unique to `leaflet`. Pick a color palette from a `RColorBrewer` or `viridis`, or build your own. Then use one of `leaflet`'s options such as `colorBin()`, `colorFactor()`, `colorNumeric()` or `colorQuantile()` to appropriately assignin the color palette to your range of data. In the example below, we picked a palette ranging from Yellow to Red and assigned it to the "total_shootings" variable.

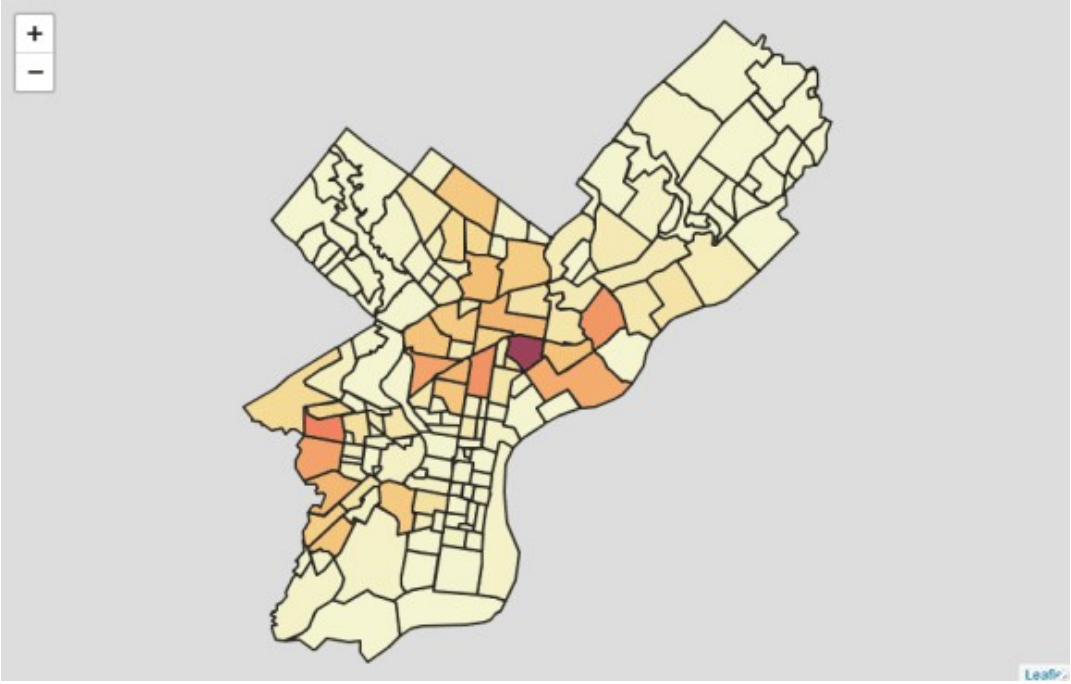
Basic map

```
shootings_count <- sf::st_join(neighborhoods, shootings) %>%
  dplyr::group_by(label) %>%
  dplyr::summarise(total_shootings = n(), .groups = "drop") %>%
  dplyr::mutate(
    label = paste0("", label, ": ", total_shootings)
  ) %>%
  dplyr::select(label, total_shootings)

pal <- leaflet::colorNumeric(
```

```
  "YlOrRd",
  domain = shootings_count$total_shootings
)

leaflet::leaflet(shootings_count) %>%
  leaflet::addPolygons(
    color = "#222", weight = 2, opacity = 1,
    fillColor = ~pal(total_shootings), fillOpacity = 0.7
  )
```

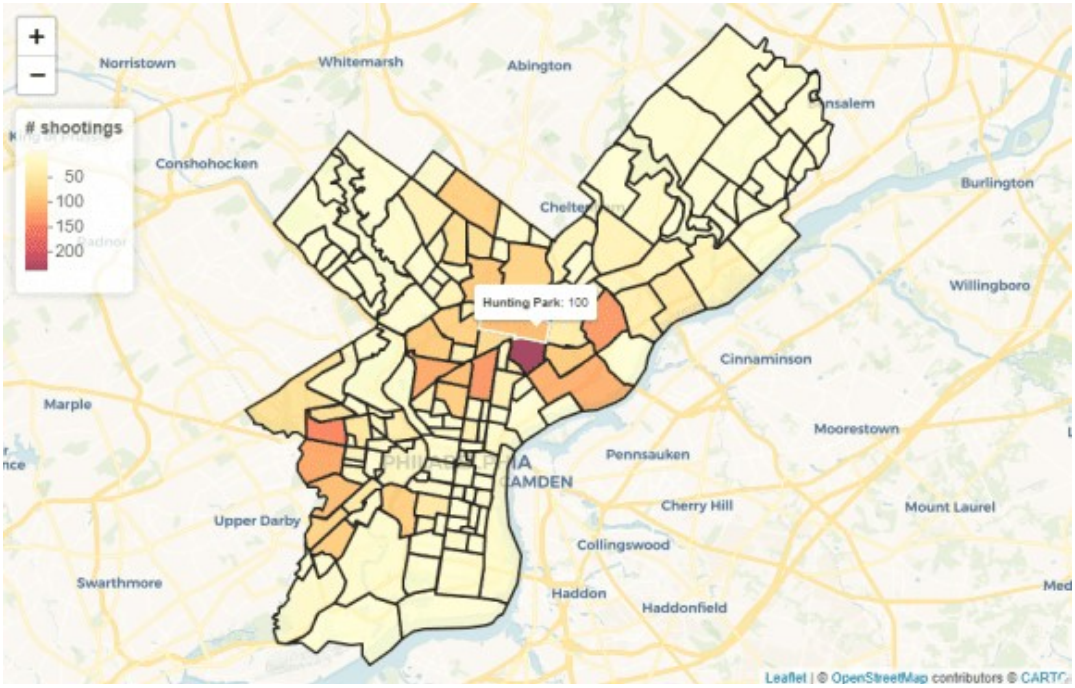


Basic Leaflet choropleth (this is a static screenshot of an interactive map). Source: OpenDataPhilly

Final map

In this final map, we add back our provider tiles, our label, and our highlight options, with no changes here from what had been done earlier in this post. We have also added a legend (and assigned the palette function to it), which describes the color range. Notice that we can change the opacity and location of the legend so that it is as unobtrusive as possible.

```
leaflet::leaflet(shootings_count) %>%
  leaflet::addProviderTiles(providers$CartoDB.Voyager) %>%
  leaflet::addPolygons(
    color = "#222", weight = 2, opacity = 1,
    fillColor = ~pal(total_shootings), fillOpacity = 0.7,
    label = ~lapply(label, htmltools::HTML),
    labelOptions = leaflet::labelOptions(direction = "top"),
    highlight = leaflet::highlightOptions(
      color = "#FFF", bringToFront = TRUE
    )
  ) %>%
  leaflet::addLegend(
    pal = pal, values = ~total_shootings, opacity = 0.7,
    title = "# shootings", position = "topleft"
  )
```



Final Leaflet choropleth, showing hover text and region highlight (this is a static screenshot of an interactive map). Source: OpenDataPhilly

Conclusion

Simple, right? The `leaflet` approach to plotting graphs can be somewhat tricky at first, as there are a lot of parameters that you can choose to adjust. In the end though, it's relatively similar to `ggplot2` in that we create a base object, add layers to it, and adjust each layer's parameters as we add it. The options available to us are a little bit different. We would struggle to recreate and *exact* copy of `ggplot2`'s maps in `leaflet`. But, that is to be expected. These two packages create two different types of maps–static and interactive–for different analytical purposes. What `leaflet` might lose in creating annotations and allowing for extremely precise aesthetic changes, it gains by allowing for paning, zooming, hovers, and popups.