Mapping with R

INFO 640 | Pratt Institute | McSweeney

Spatial Analysis & Mapping

Mapping in R is dependent upon a lot of packages. In general, spatial analysis and any GIS type work is package or program heavy for reasons we will discuss in class.

```
install.packages("ggmap")
install.packages("maps")
install.packages("mapdata")
install.packages("sf")
install.packages("sp")
install.packages("ggthemes")
install.packages("tigris")
install.packages("tmap")
install.packages("leaflet")
install.packages("rgdal")
```

```
library(tidyverse)
library(lubridate)
library(leaflet)
library(ggmap)
library(tmap)
library(tigris)
library(sp)
library(ggthemes)
library(maps)
library(mapdata)
library(sf)
library(stringr)
library(rgdal)
```

Data Sources http://data.beta.nyc https://nycopendata.socrata.com/Social-Services/311-Service-Requests-from-2010-to-Present/erm2-nwe9 https://data.

cityofnewyork.us

Getting Started

In this tutorial, we will cover layering datasets on a basemap. We will focus on NYC, and use 311 data that is publicly available on NYC Open Data. We want to know what areas have the most noise violations.

The idea of maps in R or any GIS software is very similar to the idea behind the ggplot library: layering a reference system and data with some geometry. While there are many mapping packages for R, this tutorial will introduce ggmap (best for making visualizations with already transformed data) and tm_map (best for doing spatial analysis, joins, and tabulations.

R is overall a statistical language, it is not GIS software. If your goal is to do advanced spatial statistics, draw buffers or any more complicated transformations, you may find it easier to work with a different program (i.e., QGIS). If your goal is simply to display spatial information on a map, go with ggmap. If your goal is to do basic analysis, and display a map with layers of data, R is perfect. Finally, if your goal is to visualize many small maps quickly, R is the best tool, in my opinion.

Ggmaps: Just like ggplot

We will get started by loading a map of NYC. We will use the "get_map" function that comes with the ggmap package. This function works by making an API call to google maps, so we will first need to register for an API key so we can make the call.

To obtain an API key and enable services, go to https://cloud.google.com/maps-platform/.

For the purposes of this tutorial only, you can borrow my key (in the LMS). As of mid-2018, Google requires a billing account to get an API key. It is highly unlikely that you will use it enough to be charged, but it is worth noting that you have to have to give them a credit card to get a key. If you are uncomfortable with this, contact me and I will share my key.

Once you register your key with Google, come back and tell R about it.

```
register_google(key = "YOUR KEY", write=TRUE)
```

Now let's load a map of NYC. We will get this map by using the latitude and longitude of NYC, and passing that as a location to the function, get_map. If you go out to a browser and search for NYC on google maps, the address contains all of this information:

https://www.google.com/maps/@40.6971494,-74.2598655,10z

```
nyc <- c(lon = -73.950, lat = 40.6971)
nyc_map <- get_map(location = nyc, zoom = 10, scale=1)
ggmap(nyc_map)
```

The get_map function takes 3 arguments: the location, the zoom, and the scale. **Location** is the latitude and longitude at the center point of your desired map. **Zoom** controls the level of detail and how much fits on a screen. Zoom level 10 shows the NYC area, zoom 18 would be appropriate for a neighborhood, 1 is the whole world. **Scale** controls how fine the download resolution is. It's 2 by default.

If you want a prettier map, try get_stamen in place of get_map. More on the stamen maps here: http://maps.stamen.com/#terrain/12/40.70/-74.26

Exercise

Try calling your map with different zoom levels to see what changes.

```
nyc_map <- get_map(location = nyc, zoom = ____, scale=1)
ggmap(nyc_map)
```

ggmap() follows the same logic as ggplot(). Every ggmap call requires a dataset and a geometry, and uses the same geometries (though ggmap() has a smaller set). If you just want to plot points, you can call geom_point().

The syntax is slightly different from the syntax we've been using. Instead of just calling the geometry with no parameters, we will pass the data into the geometry call. You can do this with ggplot() as well, but it's essential for ggmap() since you may want to plot multiple datasets on the same basemap.

First, we need to read in our data. We are going to explore the different types of noise complaints in NYC. This dataset came from the 311 dataset, which tracks all the 311 calls the city recieves. This dataset has been filtered by noise complaints, explore the dataset to find out more.

We exported our file as csv and will load it in here. We'll glimpse it to be sure it has the data we want.

```
noise_311 <- read.csv("Desktop/INFO640-Labs/Datasets/311_Service_Requests_from_2010_to_prese
glimpse(noise_311)
dim(noise_311)
```

Notice a few things. There are Latitude and Longitude columns, and the variable we are intersted in is in the Descriptor column. Take note that there is a Unique Key column. Do not remove or transorm this column. You Unique Key is essential for maintaining the integrity of your data. We are most interested in visualizing this dataset gepgraphically, but notice that you could also do text

analysis on the Resoultion Description to answer questions about how noise complaints get resolved.

We notice that this dataset is for 2017, but the date-time column is a factor, so we can't extract things like month or time of day. We need to change that if we do time analysis (not in this tutorial - see the Data Visualization tutorial for more on lubridate()). We may want to visualize the boroughs separately, but having those as a factor is no problem. Finally, the location type could be interesting, again, a factor is ok. Many of these varibles are irrelevant to our project. We can remove those. Let's do some cleaning.

First remove any column that is all NA's. We will do this with a function. This is a more simple function than the one we defined previously.

```
not_all_na <- function(x) any(!is.na(x))

noise_311_clean <- noise_311 %>%
select_if(not_all_na)

glimpse(noise_311_clean)
```

That took us from 53 to 38 variables. You could clean more if you need, but we will stop here. Now let's change the dates using lubridate() - the format is in month day year hour minute second. Just giving lubridate this much information, it can usually sort out what the date is.

```
noise_311_clean$Created.Date <- mdy_hms(noise_311_clean$Created.Date)
noise_311_clean$Closed.Date <- mdy_hms(noise_311_clean$Closed.Date)
glimpse(noise_311_clean)
```

Now let's plot the noise complaints on our map using a point geometry and specifying the aesthetic with Longitude (x) and Latitude (y)

```
ggmap(nyc_map) +
  geom_point(aes(Longitude, Latitude), data = noise_311_clean)
```

That is a lot of points and not terribly informative. We may want to filter by just one type of noise complaint. We will look at complaints about Ice Cream Trucks. We may want to know where complaints about ice cream trucks are the most prominent. We could further analyze the location types that complaints come from or the time of day that complaints are made.

First, let's filter our dataset using the str_detect function from the stringr package. This function returns 'TRUE' for anything that contains the substring you are interested in. Since "Ice Cream Truck" appears in different forms, combined with contruction complaints, etc., this will let us find anyone who is complaining about the Ice Cream truck even if the complaint is joined with another.

```
unique(noice_311_clean311$Descriptor)
as.character(noise_311_clean$Descriptor)
```

```
icecream_311 <- noise_311_clean %>%
filter(str_detect(Descriptor, "Ice Cream"))

dim(icecream_311)
unique(icecream_311$Descriptor)
glimpse(icecream_311)
```

Let's see what our map looks like now.

```
ggmap(nyc_map) +
  geom_point(aes(Longitude, Latitude), data = icecream_311)
```

This is much better, though we may want to increase the zoom level and the scale to improve the resolution.

```
nyc_map <- get_map(location = nyc, zoom = 11, scale=2)
ggmap(nyc_map) +
  geom_point(aes(Longitude, Latitude), data = icecream_311)
```

What if we wanted to know when people were complaining from different location types: address, intersection, or a blockface - which means that the Ice Cream truck is mid-block. We could use color to represent the three types. Just like with ggplot, we can use color to represent categorical variables, and size to represent ordinal variables.

```
ggmap(nyc_map) +
  geom_point(aes(Longitude, Latitude, color=Address.Type), data = icecream_311)
```

This is great, but if we build our maps this way, we will have to specify our data every time, which is just redundant, and we won't be able to do faceting. Both of these problems are easily solved with specifying the basemap within the ggmap() call. We want to use the basemap for things that will be common to every plot we want to make, in our case, we will keep longitude & latitude and the dataset the same, but we may want to visualize different categorical variables in different plots. Therefore, we will specify the dataset in the base_layer and the Address Type in the geometry.

```
ggmap(nyc_map,
    base_layer = ggplot(icecream_311, aes(Longitude, Latitude))) +
  geom_point(aes(color=Address.Type)
```

Now we can facet by address type.

```
ggmap(nyc_map, base_layer = ggplot(icecream_311, aes(Longitude, Latitude))) +
  geom_point(aes(color=Address.Type)) +
  facet_wrap(~Address.Type)
```

We are getting somewhere, but there is more room being taken up with our scales than with our maps. So much so that it's hard to see our maps. We need

to remove all the scales (theme_void()), add back a legend and a title, and move the title to the center (element_text(hjust=.5)). Add labels and a caption (labs).

```
ggmap(nyc_map,
    base_layer = ggplot(icecream_311, aes(Longitude, Latitude))) +
  geom_point(aes(color=Borough)) +
  theme_void()+
  theme(legend.position="bottom", plot.title = element_text(hjust = 0.5)) +
  labs(title="Ice Cream Noise Complaints by Address Type", caption="Source: NYC Open Data")
  facet_wrap(~Address.Type)
```

Congratulations! You've made your first map!

Exercise

Trade out the address type for borough - for both color and faceting. Does it make sense to do that? Why or why not?

What combination of visualizations (color and faceting) makes sense to display Address Type and Borough and to be able to compare across versions of the same map? If you don't know immediately, try it both ways.

Thematic Maps

What if we wanted to join spatial datasets together or calculate the relationship between them. We will need to use the tm_map package.

Let's start by visualizing census data for median income in NYC. We will need both a shape file (the spatial container) and a datafile (with some sort of key to line them up). You loaded the tigris package above. This allows you to pull shape files with any of the census-defined boundaries directly into your project. We will pull a shapefile of census tracts for all of the NYC counties.

```
nyc_counties <- c("New York", "Kings", "Queens", "Bronx", "Richmond")
nyc_tracts <- tracts(state = "NY", nyc_counties, cb = TRUE)

summary(nyc_tracts)

plot(nyc_tracts)
```

You'll notice that the class is SpatialPolygonsDataFrame. This is a special type of Dataframe that will allow you to draw polygons. It basically contains directions to connect the dots and draw the shapes.

Let's get some more information. The best way to start is to inspect an NYC tracts object. We're only going to call 1 entry because they are long and hard

to read.

```
head(nyc_tracts, n=1)
```

Spatial objects are almost always the S4 type, defined by a series of points and the order in which to connect those points. For census shapes, there will be State, County, tract, and 2 types of GeoId'. For all S4 object, there will be a CRS specified. You can actually call it directly with:

```
proj4string(nyc_tracts)
```

If you want to use data from 2 shapefiles on the same map, they MUST be in the same CRS. If they are not, they will not line up. This is similar to how you might try to measure in miles and kilometers EXCEPT that the conversion is not linear, let R do the conversion for you.

You'll notice that the Census tracts that we can load from the Tigris API are not clipped to the shore. There are 2 ways to deal with this. 1. Layer water over the top. 2. Use a shapefile that has been "clipped to the shoreline" such as the census tracts map from NYC Open Data. We will load a shapeful from NYC Open Data later.

Let's merge our income data. This came from NYC BetaData which is a great source when working with the NYC Metropolitan Statistical Area. Our file contains median household income from the 2010 census. This is obviously a few years before the noise data, but it's the most recent dicentenial census available.

```
income <- read.csv("Desktop/INFO640-Labs/Datasets/medianhouseholdincomecensustract.csv", hea
```

```
glimpse(income)
glimpse(nyc_tracts)
```

Now we'll merge income data to the tract data. We might think that we want to use the tract column, but we should check it first to be sure there aren't any duplicate values: Joins won't work on duplicates.

```
any(duplicated(income$TRACTCE10))
any(duplicated(income$GEOID10))
```

GeoID is a better choice. When merging shapefiles, always start with the shape file and add to it the datatable. You want to add data to the shapefile, NOT add shape to the datafile. Specify the x&y column names.

```
nyc_tracts_merge <- sp::merge(nyc_tracts, income, by.x = "GEOID", by.y = "GEOID10")
```

```
glimpse(nyc_tracts_merge)
```

Now we'll make a chloropleth with this newly merged dataset.

```
tm_shape(nyc_tracts_merge) +
  tm_fill(col = "MHI")
```

```
nyc_water <- area_water("NY", nyc_counties)

tm_shape(nyc_tracts_merge) +
  tm_fill(col = "MHI", title="Median Income NYC") +
  tm_shape(nyc_water)+
  tm_fill(col = "grey90")
```

What do you think is going on with the darkest shapes?

For New Yorkers, this can be less informative than Neighborhood Tabulation Areas (NTAs). Let's layer the NTA data over the top and aggregate to NTA. First we need to read it in. We will use readOGR for this.

```
nta <- readOGR("Desktop/INFO640-Labs/Datasets/Neighborhood Tabulation Areas/geo_export_482da

proj4string(nta)
```

WGS84 +no_defs" means "unprojected", or that it's strictly defined by latitude and longitude points on a grid without an associated method for projecting the points. If it was actually different (or already transformed), we would modify it with:

```
nta <- spTransform(nta,proj4string(nyc_tracts))
```

Now let's plot the NTAs over the census tracts

```
tm_shape(nyc_tracts_merge) +
  tm_fill(col = "MHI") +
  tm_shape(nyc_water)+
  tm_fill(col = "grey90") +
  tm_shape(nta) +
  tm_borders()
```

Layering the NTA's makes it a little easier to visually group the values together. But, if you want to summarize by neighborhood rather than census tract, you would need to join the tracts to the neighborhoods and then groupby neighborhood. This requires a file that can map one to the other. In GIS software (such as ArcGIS or QGIS), you could do this through a spatial join, but in R you must have a file that links the two. This requires a few transformation steps since the equivalency file we have does not have the GEOID, we have to use the Tracts, and since there is duplicate tract names in different counties, it has to be done county-by-county. This is not difficult, and you have all the tools you need to do it in the Data Cleaning and Data Visualization tutorials, but in the interest of time, we are not going to do it in this tutorial.

Let's finish tidying our map. Add a title to the fill col for the tracts, and specify the color palette as 'Red'. Then make the nta borders a little darker and wider (line width = lwd), and add credits to specify where your data come from. Finally, save your map.

```
tm_shape(nyc_tracts_merge) +
  tm_fill(col = "MHI", title="Median Income NYC", palette = "Reds")+
  tm_shape(nyc_water)+
  tm_fill(col = "grey90") +
  tm_shape(nta) +
  tm_borders(col= "grey30", lwd=2)
  tm_credits("Source: 2010 Census, 10 year Estimates")

save_tmap(width=6, heigth=10)
```

Interactive Maps - OPTIONAL

Leaflet has a package for R. Leaflet is an interactive Webmapping platform. You would export this type of map as a webpage and then embed it into a site. A big benefit here is the use of popups.

```
leaflet(nta) %>%
  addTiles() %>%
  addPolygons(popup = ~ntaname) %>%
  addProviderTiles("CartoDB.Positron")
```

You can also add multiple types of data to your leaflet map.

```
leaflet(nta) %>%
  addTiles() %>%
  addPolygons(popup = ~ntaname) %>%
  addMarkers(~Longitude, ~Latitude, data = icecream_311) %>%
  addProviderTiles("CartoDB.Positron") %>%
  setView(-73.98, 40.75, zoom = 13)
```

If you want to customize your Leaflet map, check out the Leaflet documentation https://rstudio.github.io/leaflet/

If you want to build more advanced maps with R or want to specify the breakstype in your Chloropleth, check out the Cartography package: https://github.com/riatelab/cartography