Homework 7

Due by 9PM November 5th

Please submit R code, explanations, and / or plots for any section marked with the **[To submit]** heading by emailing jakeporway+itp@gmail.com.

REFERENCES

These aren't mandatory but might help if you get stuck:

- Go back and read through the notes in Lectures 7 and 8
- Sections 4 and 5 of http://geography.uoregon.edu/geogr/topics/maps.htm (you can read the other sections if you like but they pertain to *another* maps library in R called maptools)

For next time:

- Section 4.1 in http://brainimaging.waisman.wisc.edu/~perlman/R/Chapter%20four%20Descriptive%20statistics%20and%20R%20graphics.pdf (much of this is review but it never hurts to read stuff you already know). The data in the examples already exist by default in R, so if you're following along you can just start using them.
- An introduction to linear regression. There may be some terms you don't know yet in here, but we'll go over them in class. This is optional reading: http://scienceblogs.com/goodmath/2008/03/27/introduction-to-linear-regress/

ASSIGNMENT

Crime Mapping

We've done some basic analysis of our Stop and Frisk data this semester, looking at basic statistics (e.g. number of stops by race or most common crimes by race) and timeseries information. We haven't yet touched geography though. Let's spend this week's assignment answering the question: Where are Stop n Frisks happening in New York?

Where Do Stops Happen?

There are two new datasets available for our analysis this week. You should download them and then load them locally because they're pretty big. The first dataset is a fuller version of the Stop and Frisk data and it lives at http://jakeporway.com/teaching/data/snf_3.csv

You'll see that it has columns "x" and "y", which indicate where the stops occurred. Hmm, after inspecting those variables you should see that they don't look like a geographic format that we're familiar with (i.e. it doesn't look like GIS lat/lon format). It turns out that the city of New York uses a special coordinate system for its stops so that all locations are "x feet east/west" and "y feet north/south" from a point in Long Island. We need to convert this format to a lat/lon format we can use.

The second dataset that we have available to us is a list of conversions from NYC's x/y format to standard lat/lon. You can download that file from:

http://jakeporway.com/teaching/data/geo.csv

Inspect geo.csv. You'll find that it has "xcoord", "ycoord", "lat", and "lon" columns, which map the xcoord and ycoord locations to latitudes and longitudes.

Now, each row in geo.csv maps an x/y location to lat/lon, but it's not aligned with our data, i.e. it's not like row 1 in snf_3.csv corresponds to row 1 in geo.csv. geo.csv is like a lookup table. What we'd like to do is *merge* the two tables, so that, for every x/y we have in snf_3.csv, we can look up the corresponding lat/lon in geo.csv. Let's do that:

[To submit]

1. To merge the two datasets, we're going to have to create a common column for our x/y values instead of two separate columns. We can do that with the *paste()* function, which pastes comma-separated bits of data together. Assuming your data is loaded from snf_3.csv as "snf", we can join the two columns with *paste()* like so:

   ```
   snf$xy <- paste(snf$x, ",", snf$y, sep="")
   ```

   What this line does is take the "x" column of snf, add a "," after every value (R is vectorizing here), then adds every element of the "y" column. The *sep* argument tells R not to put any separating characters in between the things we're merging (the default is a space). You should now have an "xy" column that consists of all the values slapped together.
2. Create an "xy" column in the geo.csv data frame using the same method.
3. Merge the two datasets using the *merge()* function we learned about in Lecture 7.

If you did this right, your final dataset should have the same number of rows as the original snf_3.csv dataset, but you should now have lat/lon columns as well.

Mapping

Now that we have our data merged, let's map it. Let's use the maps library to we learned about in Lecture 7 to map New York City using the *map()* function. Remember that *map()* uses a set of built-in polygons for nations, states, and regions, but we can also zoom in further than that by setting the xlim= and ylim= arguments:

```
map('county', 'new york',
   xlim=c(-74.25026, -73.70196),
   ylim=c(40.50553, 40.91289),
   mar=c(0,0,0,0))
```

Woof. I forgot that *maps* uses low-resolution maps. This is pretty rough looking but, heck, let's use it for now.

PRO TIP: If you ever want to zoom in on a section of this map, you can use *locator()* to identify the top-left and bottom-right corners of the rectangle you want to zoom in on, then change the xlim= and ylim= arguments to that rectangle. Totally optional, but I thought some of you might want to do that.

[To submit]
Cool, we've got our map, let's add the points to it. Use the *points()* function to add the lat/lon points of every stop onto the map. Use the *rgb()* function from Lecture 7 to set the color of the points so that they have some transparency. What do you see?

Colors
So we've got our points on a map, but that doesn't tell us much about the nature of the stops, merely where lots of them are. Let's plot the points using colors to explore where each race is being stopped. You've got your choice of one of three ways of doing this, all of which revolve around creating a vector of colors for the points. Go ahead with whichever you like:

[To submit] Plot of stops colored by race

1. **Method 1 – The Simplest, Built-in Colors:** Let's just use R's default colors to plot the races of each stop. To do that, we need to map our race values to integers (e.g "B" = 1, "W" = 2). Note that our races are back to being characters (e.g. "B", "W", "Q", etc.). Check this trick: to convert them, we can use as.factor() to treat the race characters as factors (review Lecture 8 on factors if that doesn't quite make sense) and then use as.numeric() to compress those factor values down into integers. Plot those points using the resulting vector of integers as the color vector.
2. **Method 2 – The Next Simplest, Custom Colors:** R's basic colors are super boring and, moreover, Method 1 doesn't let us assign colors explicitly to each race (we don't know which race is red, for example). However, we saw how we can *merge()* datasets together so let's create a little lookup table we can use to merge into our Stop and Frisk data:
    a. Create a data frame that looks like this:

Race	colors
"B"	rgb(r1, g1, b1, t1)
"W"	rgb(r2, g2, b2, t2)
…	…
"Q"	rgb(r8, g8, b8, t8)

    This table basically specifies which colors to use for each race, using 8 colors of your choosing.

    b. *merge()* this table with snf on the race column. Make sure to convert the resulting "colors" column to a string using as.character() (R will try to automatically convert it to a factor). The result will be that every row will now have the color for that race attached to it.

    c. Add the points to the map using the colors in the newly merged "colors" column.
3. **Method 3 – The Craziest, Color Brewer's Palettes:** If you dared to look at Method 2, you would find that defining the 8 colors by hand could be a bit of a pain. How do we even know if we've picked "good" colors? Well it turns out that R has a library called Color Brewer for generating colors from pre-built palettes. We can use Color Brewer to select 8 distinct colors very easily.

a. install.packages(“RColorBrewer”) and then load the library(RColorBrewer)

b. Color Brewer’s main function is “*brewer.pal(n, palette)*”, which selects *n* colors from a given *palette*. You can see all the palette names by typing ?brewer.pal, but try this one on for size:

```
colors <- brewer.pal(8, “Set3”)
```

You can use that column for the “colors” column of the look up table in Method 2.

No matter which plot you made, what do you see?

Let’s Make a Movie

OK, OK, we’ve got our plot, but static images never tell the whole story. Let’s make a movie! Let’s plot the stops by day.

1. Load up the *animation* library that we learned about in Lecture 7.
2. Create a day column in our snf data. You can use one of two methods:
    a. Strip out the days from the *time* column using *substr()* as we did in a previous homework.
    b. Use *POSIXct* and *lubridate* package as we did in Lecture 6 to convert the time strings to time objects and pull out the day using the *mday()* function from *lubridate*.
3. Create a for loop that ranges from 1:30 (one for each day in November) and plot the points that occur on each day in each iteration of the loop. Wrap this all in the saveHTML() function so that it makes an animation. You can pretty much just lift the code from Lecture 7 and modify it to follow this rough structure:

```
saveHTML( {
        for (i in 1:30) {
                # plot map
                # get subset of data where the day == i
                # plot the points from that subset
                ani.pause()
}})
```

Last time: What do you see?

NOTE: I know this seems like a lot but the longest route through this homework is about 30 lines of code.