Nathan Mitchell

Geog:3540 Assignment 2 – Cheat Sheet

March 1, 2021

**Observable Notebook:** https://observablehq.com/@njmitchell20/third-exercise-on-observable-work-in-progress

**Transforming Spatial Data**

Shapefile to topoJson

1. Open a browser and navigate to <https://mapshaper.org/>.

2. Click the “select” button in the center of the page and select the location of the desired shapefile.

4. Leave “detect line intersections” checked and hit “import”.

5. A display of the shapefile should appear. If the file is too large, choosing “simplify” can help reduce the size.

7. Select “export” at the top right of the screen and select “topoJSON” and click the export button.

8. The topoJSON will download to your computer, which can be later accessed and manipulated

DBF to CSV

1. Using mapshaper once again, click the “select” button and navigate to the location of the shapefile.

2. Select the file ending in .dbf and click “import”, the screen should display some boxes.

3. Find the “export” button again.

4. This time, select the “CSV” option and hit export.

**Upload and Read JSON on Observable**

1. Navigate to <https://observablehq.com> and edit/create a notebook.

2. Using the three dots in the upper right, select “file attachments”.

3. Select the json that was downloaded earlier and upload the file.

4. To access these files through the program, they must be plugged into the kernels. In their own kernels, name the notebook and set “d3” equal to: “require("d3@5")”.

5. Create another kernel and enter “topojson = require(“topojson-client@3”)” to give value to the topojson variable.

6. To insert the topojson file, enter “Cities = FileAttachment("eu\_cities.json").json()” into a new kernel. Replace “Cities” with corresponding variable name and “eu\_cities” with the name of the json file.

**Upload and Read CSV on Observable**

1. Once the “Upload and Read JSON on Observable” steps are complete, the CSV can be added to illuminate the features of the dataset.

2. In a new kernel, beneath the ones created in the previous steps, enter “csv\_data = d3.csvParse(await FileAttachment("eu\_cities.csv").text())”. Replace “csv\_data” with corresponding variable name and “eu\_cities” with the name of the csv file.

**Creating a new variable**

1. Using the previous kernel’s code and created variable, it is possible to use existing variables from the CSV to alter the new variable. In this case Population is taken over Population Rank to normalize the data.

2. In the CSV import kernel, append “({FIPS, POP, POP\_RANK}) => [FIPS, [+POP,+POP/+POP\_RANK]]” as a second argument. The kernel should now look like: “csv\_data = d3.csvParse(await FileAttachment("eu\_cities.csv").text(), ({FIPS, POP, POP\_RANK}) => [FIPS, [+POP,+POP/+POP\_RANK]])”.

3. Make sure the variable names are correct, the code runs, and the notebook is published.

**Create an unclassed map on Observable**

1. The function scaleLinear() is used to create the linear unclassed map by using the minimum and maximum value of the data that is scaled linearly.

2. This single line will create the map:

linear = d3.scaleLinear().domain([d3.min(childpct),d3.max(childpct)]).range(["white", "blue"])

3. To display the map on Observable, use the chart function: chart(numericSort(childpct), linear)

**How to use classification on Observable**

Quantile – each class contains an equal number of features.

Note: Classes often have varying ranges in order to contain an equal number of values in each class. This example does not require much separation, 3-5 classes should be sufficient, I chose 5.

1. Similar to an unclassed mapping, we can use scaleQuantile() to create the classification for quantile.

2. The domain for quantile classification should be the normalized variable we created earlier.

3. This single line needed to create the classification is largely unchanged. The colors have been changed but they can be any color you desire (more on that later).

quantile = d3.scaleQuantile().domain(childpct).range(["#edf8fb", "#b2e2e2", "#66c2a4", "#2ca25f", "#006d2c"])

4. The chart function will display the chart just as before: chart(numericSort(childpct), quantile)

Equal Interval – range of values is divided into equally sized intervals.

Note: works best on data wherein values are spread across the whole range. In this example, my Equal Interval created a class equal in size to the remaining classes combined. The method is somewhat unpredictable in how the classification is broken down and isn’t a good fit for this example.

1. Same as linear scale, equal interval takes your data’s minimum and maximum, but this time the function scaleQuantize() will create our classes.

2. quantize = d3.scaleQuantize().domain([d3.min(childpct),d3.max(childpct)]).range(["white","green","blue"])

Natural Breaks – classes are based off natural groupings; similar values are grouped together.

Note: is considered the “best” method with fewest drawbacks, 3-7 classes work best.

1. scaleThreshold() is the function required for Jenks classification.

2. First we create a new variable that we set equal to simple.ckmeans(childpct, 3).map(v => v.pop())

This enters the data and desired classes (3) which the function will use to create the classification just like the others.

3. The code should be:

jenks = d3.scaleThreshold().domain(naturalbreaks).range(["#fee8c8", "#fdbb84", "#e34a33"])

4. chart(numericSort(childpct), jenks) will display the chart.

**How to create and use colors on Observable**

1. Using Colorbrewer, you can select the number of data classes, nature of data, color scheme, and other options to create HEX color codes. They will look like: "#fee8c8".

2. Depending on the nature of your data, Colorbrewer will recommend different schemes. For sequential single or multi-hued options are available with equal increments between each hue. Diverging and qualitative color schemes are not bound to hue differences, but diverging’s middle color will always be white or neutral.

3. Observable will recognize these codes and they can be substituted for natural colors such as “white”. Using sequential Colorbrewer, it is possible to create a hued scheme that has equal differences from one class to the next sequentially.

.range(["white", "green", "blue", "red"]) can be changed to something similar to

.range(["white", "#b2e2e2", "#66c2a4", “#2ca25f”]) which will better display sequential data types

4. Simply adding a new color to a range within a classification function will add a new class of that color. Substituting one color for another will recolor the old class to a new color but will not edit/create classes.