2-3, Lesson 1 - Dynamic Map Styling

In this module, we will apply what you have learned about D3 selections, scales, and geography to dynamically creating a choropleth map and bar chart of your chosen multivariate attribute dataset. The first lesson of this module will walk you through dynamically styling the choropleth map using a color scale. The second lesson will describe how to draw a complementary bar chart, with bars that automatically sort themselves from smallest to largest.

In completing the previous module, you should have loaded your spatial and attribute data into the browser and used projection and path generators to draw a basemap from your spatial data.

When you have finished this module, you should be able to:

- Create a choropleth map based on attribute values for a single attribute within your multivariate dataset.
- Draw a bar chart representing the same attribute values visualized on the map, with the bars automatically sorted from smallest to largest.

> **Note!** There are two reference pieces that you may find useful while completing this module:
> - Murray 2013, ***Interactive Data Visualization for the Web*** ***(http://chimera.labs.oreilly.com/books/1230000000345/index.html)***, Chapter 6 ("Drawing with Data"—skip to the "Making a Bar Chart" section).
> - Bostock 2013, ***Let's Make a Bar Chart*** ***(http://bost.ocks.org/mike/bar/)***.

1.1 Checking Your Data

Before dynamically styling your choropleth map, it is very important that your data is well-ordered and accessible within the DOM. The previous module instructed you to create separate spatial and attribute datasets, with the former stored in TopoJSON format and the latter in CSV format. We have structured the assignment this way in order to demonstrate the differences in the ways that D3 AJAX methods handle CSV and JSON data and to give you practice manipulating data in JavaScript. It is possible to store your attribute data along with the spatial data as you convert from Shapefile to GeoJSON to TopoJSON formats. But we encourage you to give yourself practice writing and understanding JavaScript loop structures by working through the process of joining the attribute data from a standalone CSV dataset to the GeoJSON data for your enumeration units in *main.js*. There are multiple ways you can write this script. Below we will cover one way to do it.

First, let's look at our example data. Figure 1.1 shows our CSV data on the left and GeoJSON data on the right if we `console.log()` the variables that contains them:

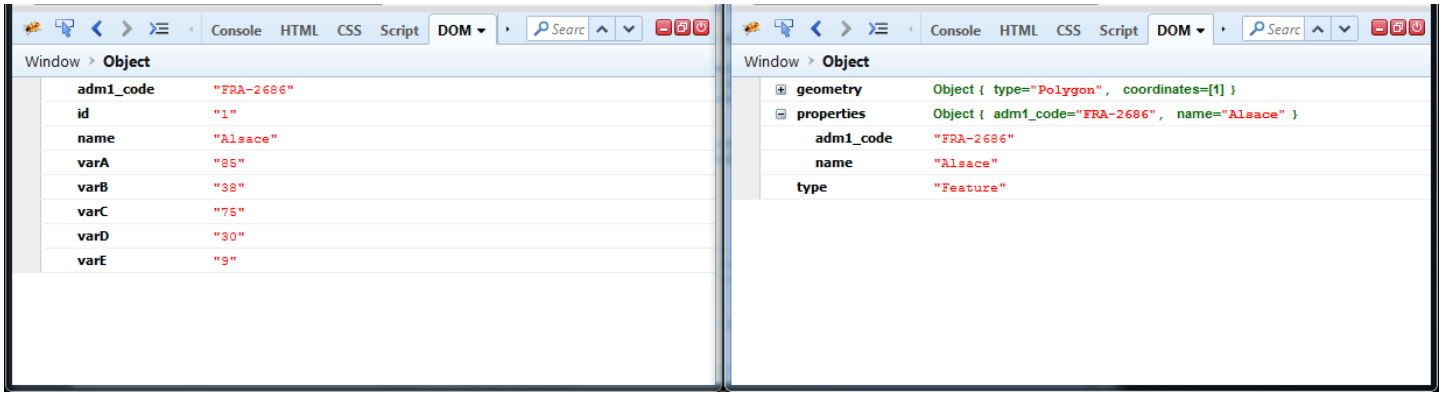

Figure 1.1: Data from a `csvData` array object (left window) and the corresponding `franceRegions` GeoJSON object (right window) prior to joining the data in main.js

Note that both datasets contain the `adm1_code` and `name` attributes. Either of these attributes could act as a primary key on which to join the data, but the `adm1_code` is an internationally-assigned code and more reliably identical between datasets, so it is better to use that attribute as the primary key. As we loop through each row of our CSV data, we can use this primary key to find the matching GeoJSON feature and transfer the other attributes to it (Example 1.1):

Example 1.1: Joining CSV data to GeoJSON enumeration units in *main.js*

JavaScript

```
//translate europe and France TopoJSONs
var europeCountries = topojson.feature(europe, europe.objects.EuropeCountries),
    franceRegions = topojson.feature(france, france.objects.FranceRegions).features;

//variables for data join
var attrArray = ["varA", "varB", "varC", "varD", "varE"];

//loop through csv to assign each set of csv attribute values to geojson region
for (var i=0; i<csvData.length; i++){
    var csvRegion = csvData[i]; //the current region
    var csvKey = csvRegion.adm1_code; //the CSV primary key

    //loop through geojson regions to find correct region
    for (var a=0; a<franceRegions.length; a++){

        var geojsonProps = franceRegions[a].properties; //the current region geojson properties
        var geojsonKey = geojsonProps.adm1_code; //the geojson primary key

        //where primary keys match, transfer csv data to geojson properties object
        if (geojsonKey == csvKey){

            //assign all attributes and values
            attrArray.forEach(function(attr){
                var val = parseFloat(csvRegion[attr]); //get csv attribute value
                geojsonProps[attr] = val; //assign attribute and value to geojson properties
            });
        };
    };
};
```

This is one of many possible ways to accomplish the data join. If you choose to experiment with other implementations, it is important that the outcome be similar to what is shown on the right side of Figure 1.2, which is the same GeoJSON feature as in Figure 1.1 after completing the join:

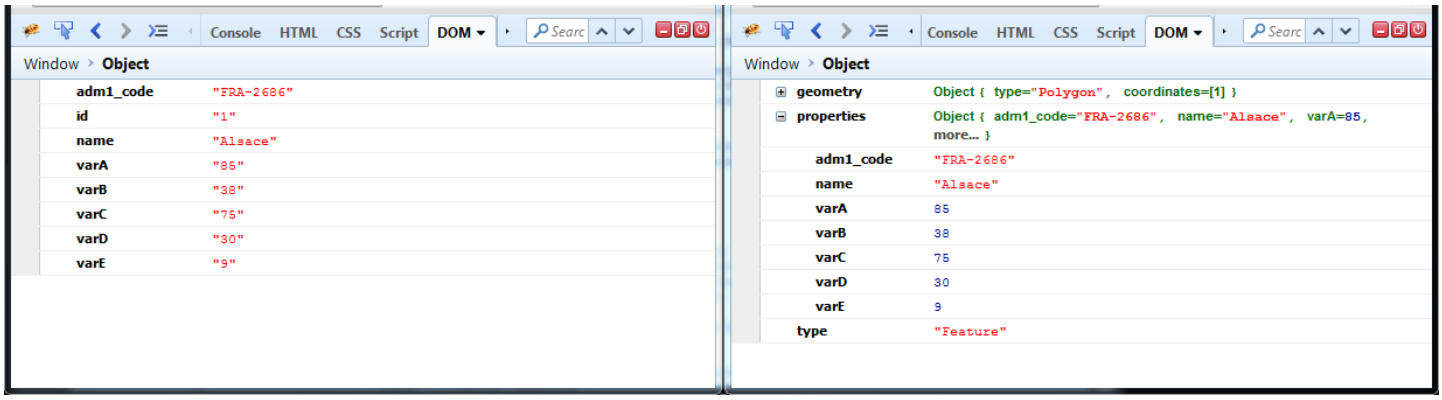

Figure 1.2: Data from a `csvData` array object (left window) and the corresponding `franceRegions` GeoJSON object (right window) after joining the data in main.js

Compare the other attributes that have appeared in the GeoJSON feature properties in Figure 1.2 to the data in the CSV feature. The numbers are identical, but note that all CSV attribute values are strings, whereas the numerical attributes in the GeoJSON feature are numbers. To work with a D3 linear scale, your attribute data *must* be typed as numbers—hence the use of the `parseFloat()` JavaScript method to change the CSV strings into numbers as they are transferred (Example 1.1 line 24).

> **Practice!** Implement script to join your CSV data to your GeoJSON features. Check the results of your data join script against the GeoJSON data structure on the right side of Figure 1.2. If your script does not produce similar results, use Example 1.1 to determine where the problem may lie.

1.2 Advanced JavaScript: From Global to Local

We will now take a brief but important diversion into computer programming best practice. Starting with our color scale, we will be building a number of functions that make use of the array of attribute names (`attrArray`) and the `expressed` attribute. Passing these variables between functions as parameters will quickly become overly complicated. For convenience, we will want to move these variables to the top of the script to make them globally accessible. This seems like a no-brainer, but it actually brings up a hidden, generally not-well-understood aspect of JavaScript. To become a skilled web developer and avoid problems when building more complicated web apps down the road, it is important to grasp this next part.

Advanced web programmers consider it bad practice to use global variables and functions. The reason has to do with the concept of **scope** (https://en.wikipedia.org/wiki/Scope_(computer_science)) in JavaScript. So far, we have succumbed to this less-than-ideal practice by defining most of our functions in the **global scope**, the segment of code execution where any entity in it is visible to the entire program. Every variable and function defined within a function is automatically moved to the **local scope** (also

called the **function scope**), in which it is only visible to other functions and variables within the parent function. There are times when you may want to keep variables in the global scope—as when you want them to be accessible from multiple *.js* files all linked to *index.html*. But doing this can also prevent these variables from being "cleaned up" when they are no longer needed, resulting in a build-up of junk in your computer memory and slowing down your application.

If you want a more thorough understanding, there are lots of online resources that explain the difference between global and local in JavaScript and why defining things in the global scope is generally a not a good idea. **This W3C wiki page (http://www.w3.org/wiki/JavaScript_best_practices#Avoid_globals)** makes the case concisely and lays out a few alternatives for when you need variables to be globally available. In Example 1.2, we implement the last alternative listed, wrapping all of our script in a self-executing anonymous function to move our script from the **global scope** into the **local scope**. Our "global" variables—which will really be operating in the local scope—can then be defined immediately within the wrapper function:

Example 1.2: Defining `attrArray` and `expressed` as pseudo-global variables in *main.js*

JavaScript

```
//First line of main.js...wrap everything in a self-executing anonymous function to move to local scope
(function(){

//pseudo-global variables
var attrArray = ["varA", "varB", "varC", "varD", "varE"]; //list of attributes
var expressed = attrArray[0]; //initial attribute

//begin script when window loads
window.onload = setMap();

... //the rest of the script

})(); //last line of main.js
```

Now is also a good time to neaten up our script by moving some of our code that performs specific tasks out of the callback function and into separate functions (Example 1.3):

Example 1.3: Subdividing the callback script into multiple functions in *main.js*

JavaScript

```
//set up choropleth map
function setMap(){

    //...MAP, PROJECTION, PATH, AND QUEUE BLOCKS FROM PREVIOUS MODULE

    function callback(error, csvData, europe, france){

        //place graticule on the map
        setGraticule(map, path);

        //translate europe and France TopoJSONs
        var europeCountries = topojson.feature(europe, europe.objects.EuropeCountries),
```

```
            franceRegions = topojson.feature(france, france.objects.FranceRegions).features;

        //add Europe countries to map
        var countries = map.append("path")
            .datum(europeCountries)
            .attr("class", "countries")
            .attr("d", path);

        //join csv data to GeoJSON enumeration units
        franceRegions = joinData(franceRegions, csvData);

        //add enumeration units to the map
        setEnumerationUnits(franceRegions, map, path);
    };
}; //end of setMap()

function setGraticule(map, path){
    //...GRATICULE BLOCKS FROM PREVIOUS MODULE
};

function joinData(franceRegions, csvData){
    //...DATA JOIN LOOPS FROM EXAMPLE 1.1

    return franceRegions;
};

function setEnumerationUnits(franceRegions, map, path){
    //...REGIONS BLOCK FROM PREVIOUS MODULE
};
```

In Example 1.3, we have moved three tasks into their own functions. The three blocks to create the background graticule have been moved to `setGraticule()` (lines 8-9 and 29-31). The loops used to accomplish the CSV to GeoJSON attribute data transfer have been moved to `joinData()` (lines 21-22 and 33-37), which returns the updated `franceRegions` GeoJSON features array. Finally, the `regions` block that adds our enumeration units to the map has been moved to its own `setEnumerationUnits()` function (lines 24-25 and 39-41). For each of these functions, the variables needed by the script within the function are passed to it as function parameters.

> **Practice!** Move your attribute array and `expressed` variables to the top of *main.js*, encapsulate your script within a self-executing anonymous wrapper function, and group tasks within the callback into their own defined functions

---

1.3 Creating a Color Scale

The next step toward creating our choropleth map is to build a color scale that we will use to visualize our attribute data on the map. You worked with a linear color scale in Previous Module that created an unclassed color scheme. For your D3 lab assignment, you should use a classed color scheme. Recall from your Introduction to Cartography course that there are multiple classification methods for classed

choropleth maps; the three most common are quantile, equal interval, and Natural Breaks. Your choropleth map should be classed, but which classification method you choose should depend on the structure of your data. To review:

- **Quantile** classification places an equal number of data values in each class, and works best when you want to create a map with the same number of enumeration units in each class but don't care about how wide the class ranges are.
- **Equal interval** classification breaks the data into classes with equal ranges (e.g., 0-10, 10-20, 20-30, etc.) and works best for data that are spread evenly across the entire data range.
- **Natural Breaks** classification uses an algorithm (typically Jenks) based on minimizing the statistical distances between data points within each class.

It is also possible to implement a piecewise scale wherein you manually manipulate the breakpoints of the data. Because this introduces an arbitrary element into the classification scheme, it is generally not advised. The examples below will demonstrate how to create each of the three main classification schemes. **Choose only one of these classification methods to implement for your choropleth map based on your dataset**.

We will start by building a quantile color scale. To keep our code neat, we can create the color scale generator in a new function, which will make use of our attribute data from the `callback()` function (Example 1.4):

Example 1.4: Creating the quantile color scale generator in *main.js*

JavaScript

```
        //create the color scale
        var colorScale = makeColorScale(csvData);

        //Example 1.3 line 24...add enumeration units to the map
        setEnumerationUnits(franceRegions, map, path, colorScale);
    };
}; //end of setMap()

//...EXAMPLE 1.3 LINES 29-41

//function to create color scale generator
function makeColorScale(data){
    var colorClasses = [
        "#D4B9DA",
        "#C994C7",
        "#DF65B0",
        "#DD1C77",
        "#980043"
    ];

    //create color scale generator
    var colorScale = d3.scaleQuantile()
        .range(colorClasses);

    //build array of all values of the expressed attribute
```

```
    var domainArray = [];
    for (var i=0; i<data.length; i++){
        var val = parseFloat(data[i][expressed]);
        domainArray.push(val);
    };

    //assign array of expressed values as scale domain
    colorScale.domain(domainArray);

    return colorScale;
};
```

In Example 1.4, we implement the color scale using `d3.scale.quantile()` (https://github.com/d3/d3-scale/blob/master/README.md#quantile-scales) to create a quantile scale generator (line 22). The generator takes an input domain that is either continuous or a discreet set of values and maps it to an output range of discreet values. When the domain is continuous, the output will be an equal interval scale; when the domain is discreet, a true quantile scale will be generated. For the range, rather than letting D3 interpolate between two colors as we did in Previous Module, we pass an array of five color values derived from **ColorBrewer** (http://colorbrewer2.org/) to the `.range()` operator (lines 13-19 and 23). These will be our five class colors in our classification scheme. (**Note**: You can also reference ColorBrewer scales using **ColorBrewer.js** (https://github.com/axismaps/colorbrewer/) or the **d3-scale-chromatic** (https://github.com/d3/d3-scale-chromatic) plugin).

To build a quantile scale, we need to assign all of the attribute values for the currently expressed attribute in our multivariate dataset as the scale's domain (line 33). This requires us to build an array of these values using a loop to access the value for each feature in the dataset (lines 26-30). The function then returns the scale generator. Within the callback, we create a `colorScale` variable to accept the scale generator from the `makeColorScale()` function, passing the `csvData` into the function (line 2). We also add the `colorScale` as a parameter sent to `setEnumerationUnits()` (line 5).

When the quantile scale generator is provided all values in the dataset (the `domainArray`) as its domain, it divides the values into bins that have an equal number of values each and assigns each bin one of the color classes. But `d3.scaleQuantile()` can also be used to create an equal interval scale. The way to do this is to generate a continuous domain by passing `.domain()` an array with only two values: the minimum and maximum value of the dataset (Example 1.5):

Example 1.5: Creating an equal interval color scale generator in *main.js*

JavaScript

```
 //Example 1.4 line 11...function to create color scale generator
function makeColorScale(data){
    var colorClasses = [
        "#D4B9DA",
        "#C994C7",
        "#DF65B0",
        "#DD1C77",
        "#980043"
    ];
```

```
    //create color scale generator
    var colorScale = d3.scaleQuantile()
        .range(colorClasses);

    //build two-value array of minimum and maximum expressed attribute values
    var minmax = [
        d3.min(data, function(d) { return parseFloat(d[expressed]); }),
        d3.max(data, function(d) { return parseFloat(d[expressed]); })
    ];
    //assign two-value array as scale domain
    colorScale.domain(minmax);

    return colorScale;
};
```

Given a two-value input domain and a range array with five output values, the generator will create five bins with a equal ranges of values between the minimum and maximum. For either the quantile or equal interval scale generator, you can use the Console to discover the class breaks that the scale creates by adding the statement `console.log(colorScale.quantiles())` at the bottom of the function.

The third major classification scheme, Natural Breaks, tries for a happy medium between quantile and equal interval classification, avoiding the disadvantages of each by finding "natural" clusterings of the data. If the distributions of your attribute values have long tails or several outliers, you should consider implementing a Natural Breaks classification.

To create a Natural Breaks color scale generator, we need to use a D3 **threshold scale (https://github.com/d3/d3-scale/blob/master/README.md#threshold-scales)** instead of a quantile scale. The threshold scale generator takes the same discreet array of color strings for its range, but it requires a set of specified class breaks for the domain (this is also how you would create a scale with arbitrary class breaks, which is not recommended). The number of class breaks in the domain array should be one less than the number of range output values. Any data values that are the same as a class break value are included in the class *above* the break.

To create the breaks, you will need a clustering algorithm. The Jenks algorithm commonly used by cartographers used to be included in the **Simple Statistics (http://simplestatistics.org/)** code library; however, it was replaced by the more recent **Cartesian k-means (http://www.cs.toronto.edu/~norouzi/research/papers/ckmeans.pdf)** (Ckmeans) algorithm. Ckmeans does an excellent job for our purposes. If you wish to implement a Natural Breaks classification, download *simple-statistics.js* from the link above, place it in your *lib* folder, and add a script link to it in your *index.html*.

The script in Example 1.6 is loosely based on cartographer Tom MacWright's **Natural Breaks choropleth example (http://bl.ocks.org/tmcw/4969184)**, which uses the Jenks algorithm from the older version of Simple Statistics (included in his example). We have implemented our example below using Ckmeans (Example 1.6):

Example 1.6: Creating a Natural Breaks color scale generator in *main.js*

JavaScript

```
//function to create color scale generator
function makeColorScale(data){
    var colorClasses = [
        "#D4B9DA",
        "#C994C7",
        "#DF65B0",
        "#DD1C77",
        "#980043"
    ];

    //create color scale generator
    var colorScale = d3.scaleThreshold()
        .range(colorClasses);

    //build array of all values of the expressed attribute
    var domainArray = [];
    for (var i=0; i<data.length; i++){
        var val = parseFloat(data[i][expressed]);
        domainArray.push(val);
    };

    //cluster data using ckmeans clustering algorithm to create natural breaks
    var clusters = ss.ckmeans(domainArray, 5);
    //reset domain array to cluster minimums
    domainArray = clusters.map(function(d){
        return d3.min(d);
    });
    //remove first value from domain array to create class breakpoints
    domainArray.shift();

    //assign array of last 4 cluster minimums as domain
    colorScale.domain(domainArray);

    return colorScale;
};
```

In Example 1.6, we start with a call to `d3.scaleThreshold()` rather than `d3.scaleQuantile()` (line 12). The range remains the same (line 13), and we build a `domainArray` from all expressed attribute values as if we were implementing a quantile scale (lines 16-20). The extra step not present in the other classification schemes is to use the Simple Statistics `ckmeans()` (http://simplestatistics.org/docs/#ckmeans) method to generate five clusters from our attribute values (line 23). These clusters are returned in the form of a nested array, which you can see in the Console if you pass `clusters` to a `console.log()` statement. We then reset the `domainArray` to a new array of break points, using JavaScript's `.map()` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map) method to build a new array out of each cluster's minimum value (lines 25-27). Since the threshold scale includes each break point in the class above it, we want our array of break points to be class minimums, which we select using `d3.min()` (line 26). The final step in formatting the `domainArray` is to remove the first value of the array using the JavaScript `.shift()` (http://www.w3schools.com/jsref/jsref_shift.asp) method,

leaving the correct number of break points (4)—each of which is included by the class above it—in the `domainArray`.

Of the three classification schemes, which should we use? It depends on the distribution of our data. Figure 1.3 demonstrates the different bins created by the three classification schemes and shows where each enumeration unit's varA attribute value fits:

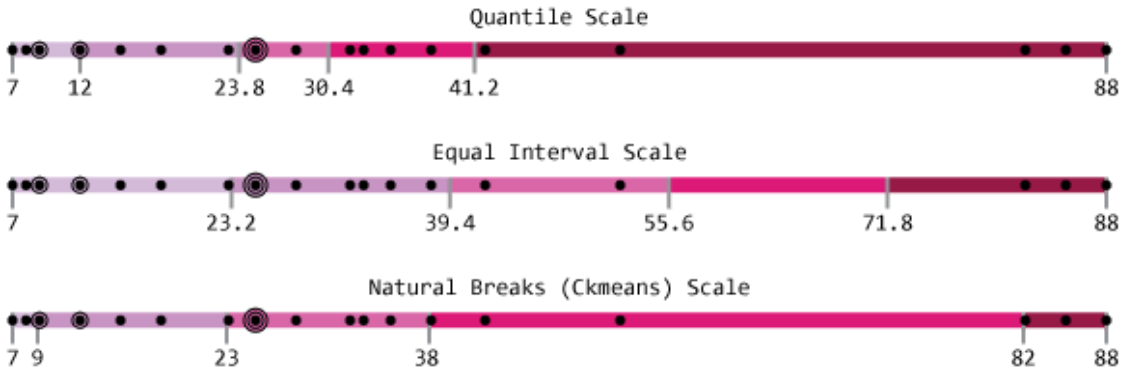

Figure 1.3: Difference between quantile and equal interval classification of the varA attribute

Notice in Figure 1.3 that mapping our example datset with an equal interval classification scheme would result in many of our enumeration units falling into one of the first two classes, a few units in each of the third and fifth classes, and none of the enumeration units falling into the fourth class for the varA attribute. The quantile scale results in every color class appearing on the map a similar number of times, but as a result groups the three highest values with the next two lowest despite a very large gap in between. Natural Breaks ensures that each class is represented but clusters the data in such a way as to minimize the gaps between data values within a single class.

> **Practice!** Choose a choropleth classification scheme based on your dataset and create a color scale generator that implements that scheme in *main.js*.

1.4 Coloring the Enumeration Units

Once we have constructed our color scale generator, the final step in coloring our choropleth is to apply it to our `regions` selection. We can do this by adding a `.style()` operator at the end of the `regions` block with an anonymous function that applies the `colorScale` to each datum's currently expressed attribute value to return the fill (Example 1.7 lines 13-15):

Example 1.7: Coloring enumeration units in *main.js*

JavaScript

```
//Example 1.3 line 38
function setEnumerationUnits(franceRegions, map, path, colorScale){

    //add France regions to map
    var regions = map.selectAll(".regions")
```

```
        .data(franceRegions)
        .enter()
        .append("path")
        .attr("class", function(d){
            return "regions " + d.properties.adm1_code;
        })
        .attr("d", path)
        .style("fill", function(d){
            return colorScale(d.properties[expressed]);
        });
};
```

We now have a choropleth map (Figure 1.4):

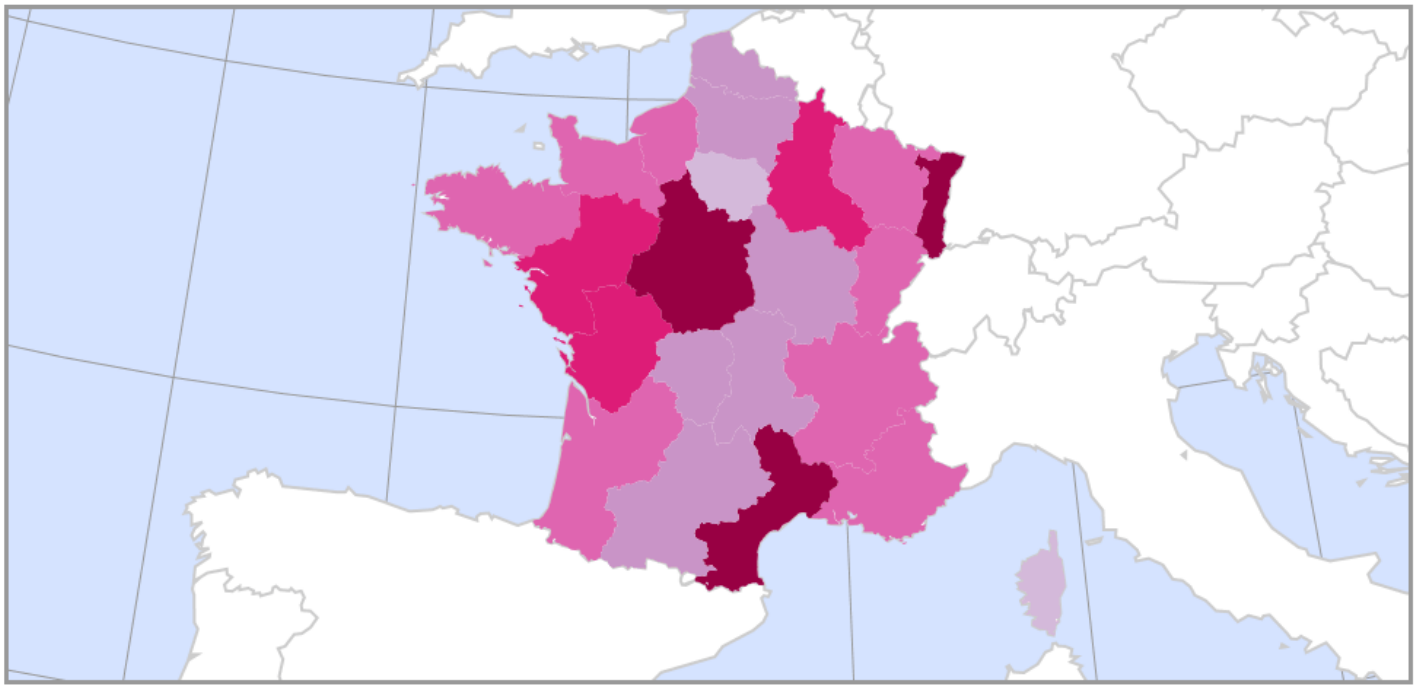

Figure 1.4: Colored enumeration units

This works fine if *every* enumeration unit has a value for the current attribute. But you may have some features in your dataset that do not have values for every attribute. Given the script used in Example 1.3, these may cause an error or result in some enumeration units having a default black fill. We can handle this situation with a helper function that tests for the presence of each attribute value, returns the correct color class if it exists, and returns a neutral gray if it does not (Example 1.8):

Example 1.8: Choropleth helper function in *main.js*

JavaScript

```
//function to test for data value and return color
function choropleth(props, colorScale){
    //make sure attribute value is a number
    var val = parseFloat(props[expressed]);
    //if attribute value exists, assign a color; otherwise assign gray
    if (typeof val == 'number' && !isNaN(val)){
        return colorScale(val);
    } else {
```

```
        return "#CCC";
    };
};
```

In Example 1.8, our `choropleth()` function accepts the feature properties from each enumeration unit and the `colorScale` generator as parameters (line 2). It then accesses the expressed attribute value, typing the value as a number and assigning it to the variable `val` (line 4). The conditional statement tests to see whether the expressed attribute value exists and is a real number using JavaScript's **typeof** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof) operator and **isNaN()** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/isNaN) function, and if so, applies the `colorScale` to it; if not, it returns a neutral gray (lines 6-10). Now, rather than directly applying the `colorScale` to each datum in the `regions` block, we can call the `choropleth()` function, passing in the feature properties and scale generator and returning whatever color string it generates to the `<path>` element's `fill` style (Example 1.9):

Example 1.9: Applying the `choropleth` function within the `regions` block in *main.js*

JavaScript

```
        //Example 1.7 line 13
        .style("fill", function(d){
            return choropleth(d.properties, colorScale);
        });
```

Finally, we can visually highlight the color change between enumeration units by adding a solid border to the `regions` class in *style.css* (Example 1.10):

Example 1.10: Adding a border to enumeration units in *style.css*

CSS

```
.regions {
    stroke: #000;
    stroke-width: 0.5px;
    stroke-linecap: round;
}
```

Here is our final result (Figure 1.5):

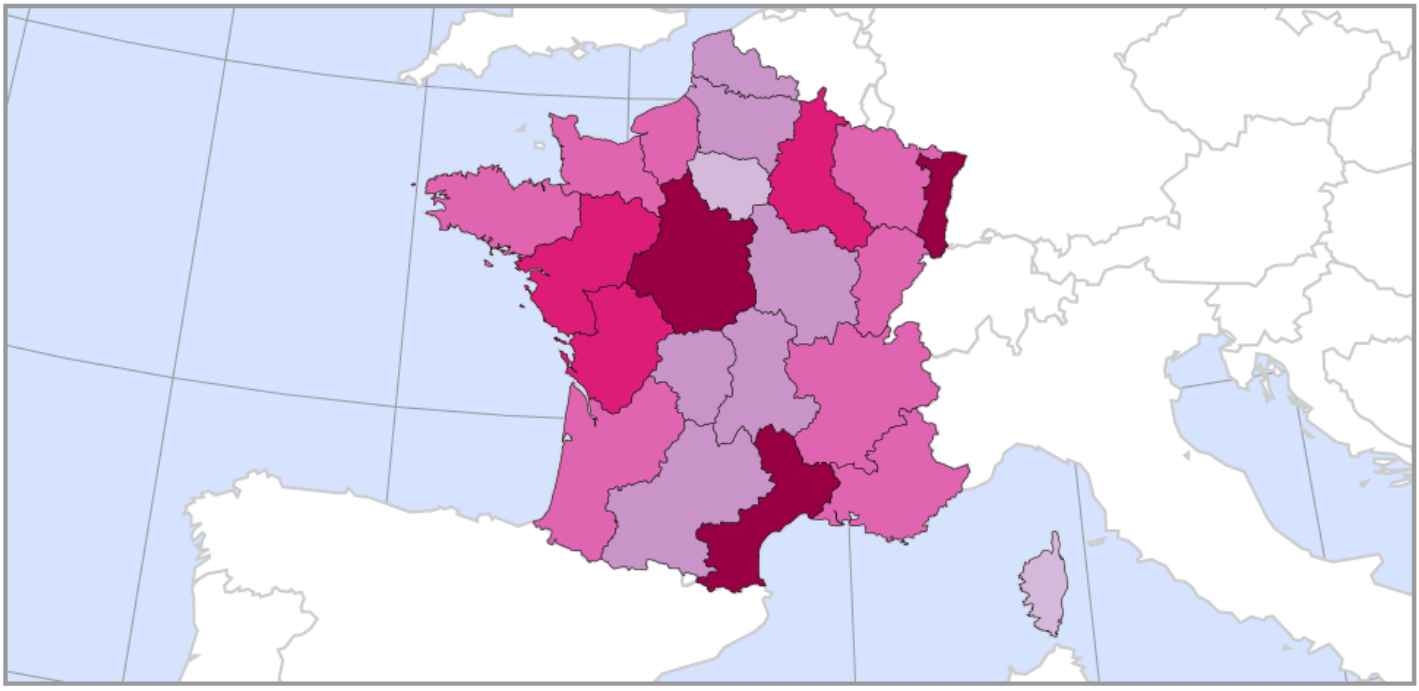

Figure 1.5: The choropleth map with enumeration unit borders

> **Practice!** Apply your color scale generator to your enumeration units. Make sure your script assigns a neutral color to any units with no value for the expressed attribute.

> **Self-Check:**
>
> **1. Which of the following choropleth classification schemes will result in approximately the same number of enumeration units in each class?**
>
> **a.** Quantile
> **b.** Equal Interval
> **c.** Natural Breaks
> **d.** None of the above
>
> **2. True/False:** Within your *main.js* script, it is typically a good idea to define variables and functions in the global scope so they are visible to the entire program.