2-4, Lesson 2 - Linked Retrieve Interactions

2.1 Highlighting

The final required components of your D3 laboratory assignment are linked highlighting and dynamic labels, both components of the *retrieve* interaction operator. **Highlighting** is visual feeback provided across data views when selecting elements of the visualization (**brushing**). **Linking** is the coupling of any interaction operator performed on one data view to feedback given in all views. Linking is a distinguising feature of coordinated visualizations that allows the user to easily compare data across different types of visualizations.

Dynamic labels (or **tooltips**), as discussed in previous module, are brief labels with critical information about the selected feature that follow the cursor. We will tackle adding these in the third section of this lesson. You may choose to implement other forms of the *retrieve* operator as well.

In order to highlight the enumeration units on our map and the bars in our bar chart, we will actually need two separate functions: a `highlight()` function that styles each to give appropriate visual feedback on brushing, and a `dehighlight()` function that returns the element to its original style. In the examples below, the highlighting strategy we implement is to add a blue stroke to each feature. You may choose a different highlighting strategy that seems more appropriate to you. Again, it should be stressed that the overall style implemented here is almost certainly sub-optimal for your application, and you *must* alter it significantly to receive a passing grade on the assignment.

First, let's write the `highlight()` function, which will restyle the stroke of each enumeration unit and bar:

Example 2.1: Adding a `highlight()` function in *main.js*

JavaScript

```
 //function to highlight enumeration units and bars
function highlight(props){
    //change stroke
    var selected = d3.selectAll("." + props.adm1_code)
        .style("stroke", "blue")
        .style("stroke-width", "2");
};
```

In Example 2.1, `props` is the properties object of the selected element from the GeoJSON data or the attributes object from the CSV data, depending on whether the selected element is an enumeration unit on the map or a bar on the chart (line 2). Since the `adm1_code` attribute should be the same for the matching region and bar, our class selector in the `.selectAll()` method should select both matching elements (line 4). apply a wider blue stroke to both elements with two `.style()` operators, one for the stroke color and one for the stroke width (lines 5-6).

In order to make this function work, we need to call it from `"mouseover"` event listeners attached to our `regions` block and our `bars` block (Example 2.2):

Example 2.2: Adding mouseover event listeners in *main.js*

JavaScript

```
//in setEnumerationUnits()...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 choropleth(d.properties, colorScale);
       })
       .on("mouseover", function(d){
           highlight(d.properties);
       });

   //...

   //in setChart()...set bars for each province
   var bars = chart.selectAll(".bar")
       .data(csvData)
       .enter()
       .append("rect")
       .sort(function(a, b){
           return b[expressed]-a[expressed]
       })
       .attr("class", function(d){
           return "bar " + d.adm1_code;
       })
       .attr("width", chartInnerWidth / csvData.length - 1)
       .on("mouseover", highlight);
```

In Example 2.2, the event listener added to the `regions` block uses an anonymous function to call the `highlight()` function so that the `properties` object can be passed to it without passing the entire GeoJSON feature (lines 13-15). The listener on the `bars` block, on the other hand, simply passes the name of the function as a parameter, since this block uses the `csvData` and thus the datum is already equivalent to the `properties` object within the GeoJSON feature.

If we now test our highlighting, we can see it working (Figure 2.1). But the features retain their blue borders even after the mouse is removed, quickly making a mess of the visualization! This is why we need a `dehighlight()` function as well as a `highlight()` function.

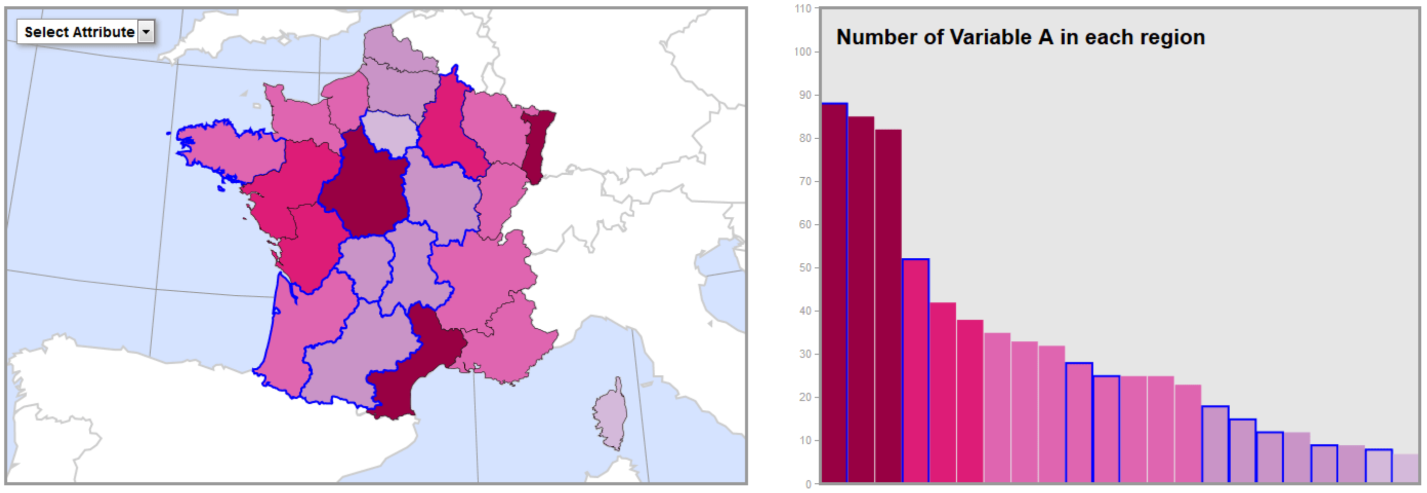


Figure 2.1: Linked highlighting

> **Practice!** Implement highlighting on your choropleth map and linked visualization.

2.3 Dehighlighting

The `dehighlight()` function is more challenging to implement than the `highlight()` function because you have to figure out how to tell each enumeration unit and bar to go *back* to whatever style it had before. This is trickier than it seems. How will the interpreter know what style to assign? One solution is to use the exact same style for both map and chart; this is not recommended and may not be possible depending on the type of visualization you choose to implement. Another choice is to hard-code each style as a global variable and create separate selections for map elements and chart elements to restyle each.

The implementation shown here uses a third method: it takes advantage of the SVG `<desc>` (https://developer.mozilla.org/en-US/docs/Web/SVG/Element/desc) element, a simple element that only holds text content, remains invisible to the user, and can be appended to any other kind of SVG element. We can add a `<desc>` element containing a text description of the original style to each of our map's `<path>` elements and our chart's `<rect>` elements (Example 2.3):

Example 2.3: Adding `<desc>` elements with style descriptors in *main.js*

JavaScript

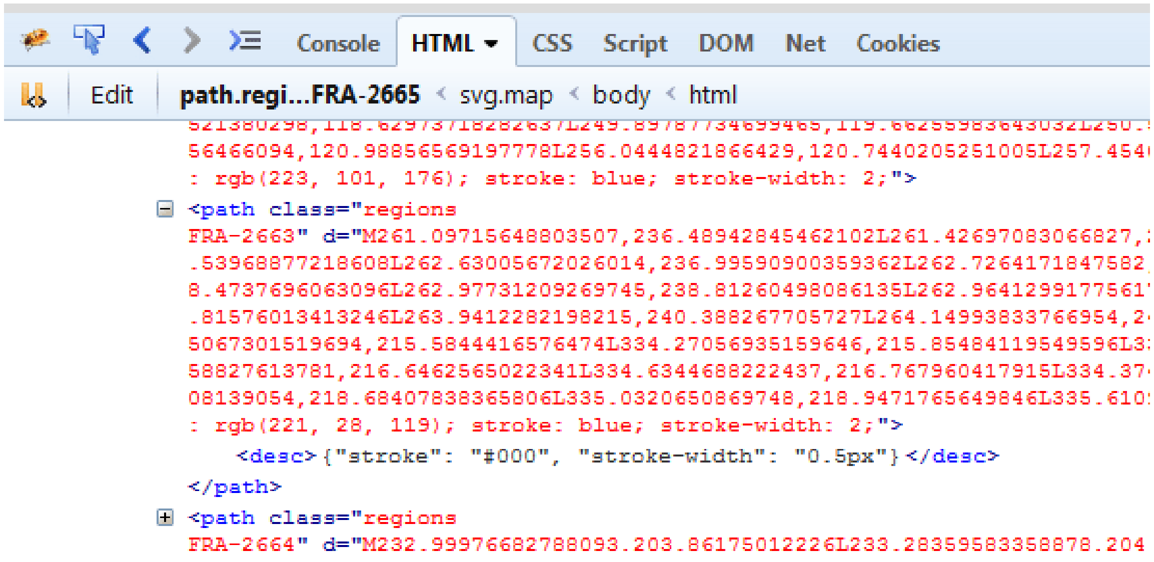
```
    //below Example 2.2 line 16...add style descriptor to each path
    var desc = regions.append("desc")
        .text('{"stroke": "#000", "stroke-width": "0.5px"}');

    //...

    //below Example 2.2 line 31...add style descriptor to each rect
    var desc = bars.append("desc")
        .text('{"stroke": "none", "stroke-width": "0px"}');
```

In Example 2.3, note that each style descriptor string adheres to JSON format (lines 3 and 9). This will make the information easier to parse in the `dehighlight()` function. Be aware that JSON formatting uses even stricter syntax than regular JavaScript: each property and value *must* be encased by *double-quotes*. The JSON parser will fail if single quotes are used, if necessary quotes are left our, or if there are excess or missing punctuation marks.

Using the Inspector, we can see that each `<path>` element and each `<rect>` element now have child `<desc>` elements with our pseudo-object string (Example 2.2):



```
521380298,118.6297371828263 7L249.8978773469946 5,119.662559836430 32L250.
56466094,120.9885656919777 8L256.0444821866429,120.7440205251005L257.454
: rgb(223, 101, 176); stroke: blue; stroke-width: 2;">
<path class="regions
FRA-2663" d="M261.09715648803507,236.48942845462102L261.42697083066827,
.5396887721860 8L262.6300567202601 4,236.99590900359362L262.7264171847582
8.4737696063096L262.9773120926974 5,238.8126049808613 5L262.964129917756 1
.815760134132 46L263.941228219821 5,240.38826770572 7L264.1499383376695 4,2
5067301519694,215.58444165764 74L334.27056935159646,215.85484119549596L3
58827613781,216.64625650223 41L334.6344688222437,216.76796041791 5L334.37
08139054,218.68407838365806L335.0320650869748,218.947176564984 6L335.610
: rgb(221, 28, 119); stroke: blue; stroke-width: 2;">
    <desc>{"stroke": "#000", "stroke-width": "0.5px"}</desc>
</path>
<path class="regions
FRA-2664" d="M232.99976682788093.203.86175012226L233.28359583358878.204
```

Figure 2.2: Inspecting the <desc> elements

We can now make use of the contents of these `<desc>` elements in our `dehighlight()` function. It still takes a bit of doing to retrieve the information stored in the `<desc>` elements, so we will step carefully through the `dehighlight()` example below (Example 2.4):

Example 2.4: Adding a `dehighlight()` function in *main.js*

JavaScript

```
 //function to reset the element style on mouseout
function dehighlight(props){
    var selected = d3.selectAll("." + props.adm1_code)
        .style("stroke", function(){
            return getStyle(this, "stroke")
        })
        .style("stroke-width", function(){
            return getStyle(this, "stroke-width")
        });

    function getStyle(element, styleName){
        var styleText = d3.select(element)
            .select("desc")
            .text();

        var styleObject = JSON.parse(styleText);

        return styleObject[styleName];
    };
};
```

In Example 2.4, the `dehighlight()` function begins much the same as the `highlight()`function, creating a `selected` block that restyles the `stroke` and `stroke-width` styles (lines 3-9). However, we can't pass one value for each style, since we are resetting both enumeration units and bars, which have different styles. Instead, each style calls an anonymous function, which in turn calls a separate `getStyle()` function to retrieve the information stored in the `<desc>` element for that style. The `getStyle()`function takes as parameters the current element in the DOM—represented by the keyword `this`—and the style property being manipulated (lines 5 and 8). The results returned by `getStyle()` are passed along to the style object and in turn to the `.style()` operator, which applies them to each element.

Within the `getStyle()` function, we retrieve the `<desc>` content by creating a selection of the current DOM element, selecting its `<desc>` element, and returning the text content using the `.text()` operator with no parameters (lines 14-16). We then parse the JSON string to create a JSON object (line 18) and return the correct style property's value (line 20).

This completes the `dehighlight()` function, which we can now add event listeners to call (Example 2.5):

Example 2.5: Adding `mouseout` event listeners in *main.js*

JavaScript

```
        //Example 2.2 line 12...regions event listeners
        .on("mouseover", function(d){
            highlight(d.properties);
        });
        .on("mouseout", function(d){
            dehighlight(d.properties);
        });

        //...

        //Example 2.2 line 30...bars event listeners
        .on("mouseover", highlight);
        .on("mouseout", dehighlight);
```

We now have working linked highlighting and dehighlighting, allowing only one feature to be selected at a time (Figure 2.3):

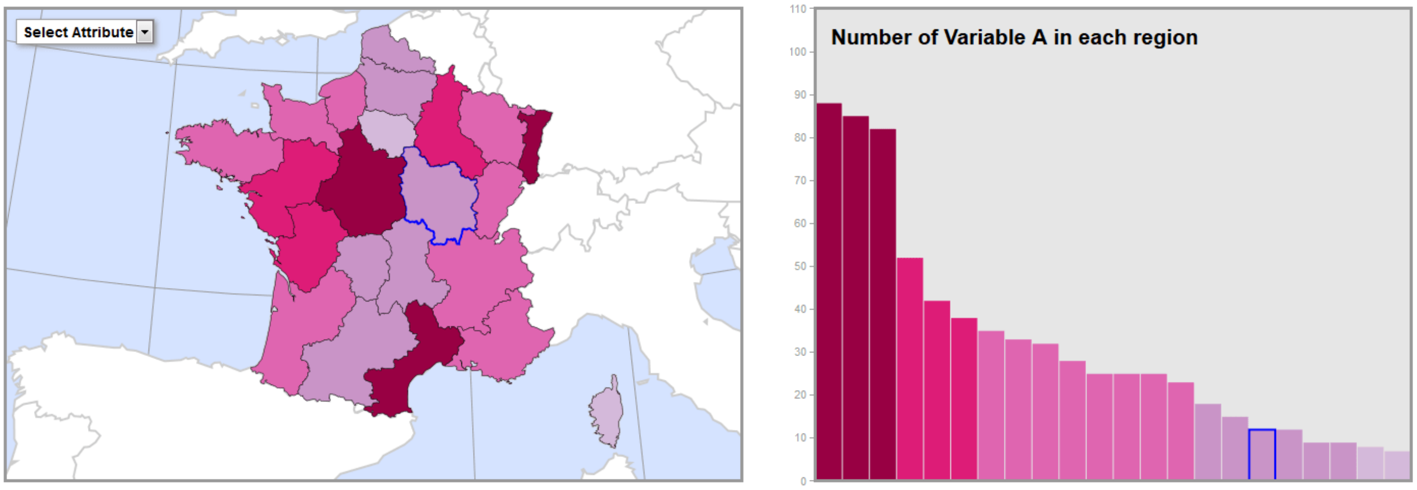


Figure 2.3: Linked highlighting and dehighlighting

> **Practice!** Implement dehighlighting on your choropleth map and linked visualization.

2.3 Dynamic Label

The final assigned task in support of the *retrieve* interaction operator is to implement a dynamic label (or tooltip) showing the attribute values for each region of France. For this tutorial, we will implement a simple label that moves with the cursor. First, to create the label, we can write a new `setLabel()` function that makes use of the feature properties (Example 2.6):

Example 2.6: Creating the dynamic label in *main.js*

JavaScript

```
//function to create dynamic label
function setLabel(props){
    //label content
    var labelAttribute = "<h1>" + props[expressed] +
        "</h1><b>" + expressed + "</b>";

    //create info label div
    var infolabel = d3.select("body")
        .append("div")
        .attr("class", "infolabel")
        .attr("id", props.adm1_code + "_label")
        .html(labelAttribute);

    var regionName = infolabel.append("div")
        .attr("class", "labelname")
        .html(props.name);
};
```

In Example 2.6, within the `setLabel()` function, we first create an HTML string containing an `<h1>` element with the selected attribute value and a `<b>` element with the attribute name (lines 4-5).

If these elements needed attributes, it would quickly become unweildy to include them in an HTML string, but since they don't, writing an HTML string is a handy shortcut. Next, we create the actual label `<div>` element, giving it `class` and `id` attributes and assigning our HTML string with the `.html()` operator (lines 8-12). Finally, we add a child `<div>` to the label to contain the name of the selected region.

Since we want our label to show up whenever the user highlights a region or bar, we can now call `setLabel()` from within `highlight()`, passing it the `props` variable as a parameter. To make sure our labels don't stack up in the DOM, we need to remove each new label on dehighlight as well (Example 2.7):

Example 2.7: Removing the info label on dehighlight in *main.js*

JavaScript

```
    //below Example 2.4 line 21...remove info label
    d3.select(".infolabel")
        .remove();
```

Without any styles applied to it, the label will look pretty messy. Let's style it in *style.css* (Example 2.8):

Example 2.8: Label styles in *style.css*

CSS

```
.infolabel {
    position: absolute;
    height: 50px;
    min-width: 100px;
    color: #fff;
    background-color: #000;
    border: solid thin #fff;
    padding: 5px 10px;
}

.infolabel h1 {
    margin: 0 20px 0 0;
    padding: 0;
    display: inline-block;
    line-height: 1em;
}
```

These styles create a simple black label with white text (Figure 2.4):

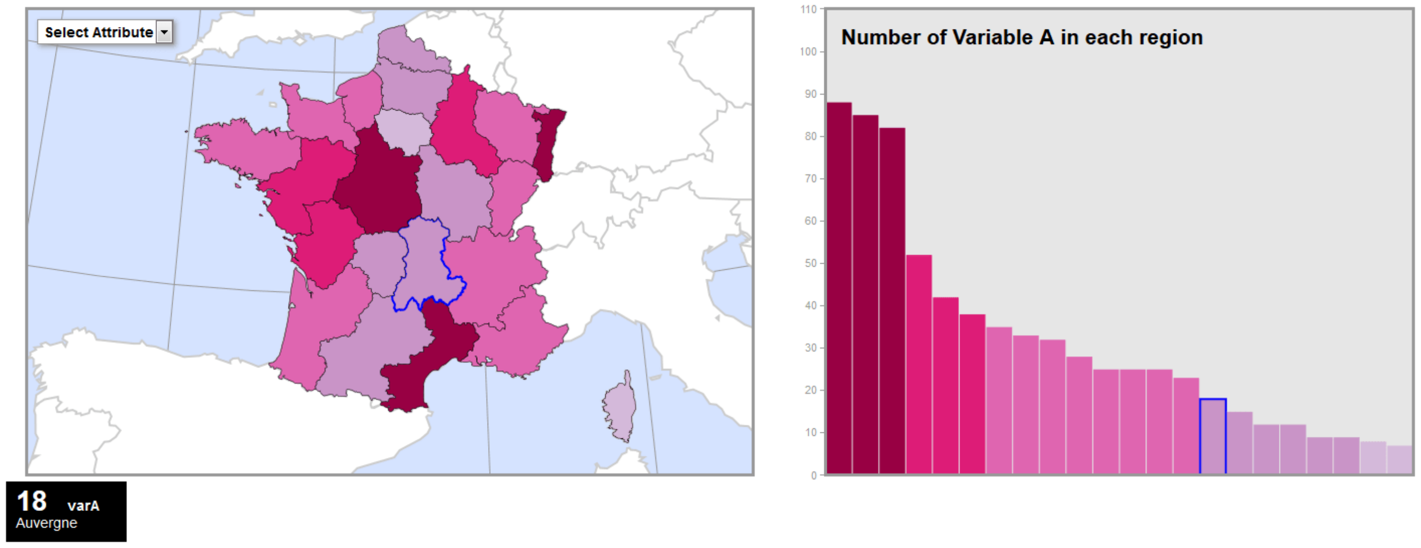


Figure 2.4: Styled info label

The next step, of course, is to remove the our label from its moorings at the bottom of the page and teach it to follow the user's cursor like a good little tooltip. This may seem tricky, but luckily for us, D3 provides a handy object, `d3.event` (https://github.com/d3/d3-selection/blob/master/README.md#event), that holds the position of the mouse whenever an event is fired on the page. We can use this to set the position of our info label in a function that is called on any `mousemove` event (Example 2.8):

Example 2.8: Adding movement to the info label in *main.js*

CSS

```
//function to move info label with mouse
function moveLabel(){
    //use coordinates of mousemove event to set label coordinates
    var x = d3.event.clientX + 10,
        y = d3.event.clientY - 75;

    d3.select(".infolabel")
        .style("left", x + "px")
        .style("top", y + "px");
};
```

In Example 2.8, we retrieve the coordinates of the mousemove event and manipulate them to set the bottom-left corner of the label above and to the right of the mouse (lines 4-5). We then pass those coordinate values to the `left` and `top` styles of the label—which we use instead of `margin-left` and `margin-top` because the label's position is set to `absolute` instead of `relative` (lines 7-9). We now need to call this function as a listener handler for a `mousemove` event on both the map and chart (Example 2.9):

Example 2.9: Adding `mousemove` event listeners in *main.js*

JavaScript

```
        //Example 2.5 line 1...regions event listeners
        .on("mouseover", function(d){
            highlight(d.properties);
        })
        .on("mouseout", function(d){
            dehighlight(d.properties);
        })
        .on("mousemove", moveLabel);

        //...

        //Example 2.5 line 11...bars event listeners
        .on("mouseover", highlight)
        .on("mouseout", dehighlight)
        .on("mousemove", moveLabel);
```

This should cause our info label to follow our mouse. However, there are two minor issues we should resolve. First, if the mouse gets too high or too far to the right, the label may overflow the page. Second, depending on your browser speed, you may notice that sometimes the label is added to the page before the position styles take affect, causing it to flash briefly in the corner.

To tackle the first problem, we need to test whether the label has moved off the page, and if so, switch which side of the mouse it appears on. For the vertical (`y`) coordinate, we can simply test whether the event Y coordinate is less than our desired distance between mouse and upper-left label corner; if it is, then we need to use a vertical coordinate for the label that switches which side of the mouse it is on vertically (Example 2.10).

For the horizontal (`x`) coordinate, since the label is to the right of the mouse by default, we need to check to see if the label overflows the right side of the page. To do this, we need to access the browser window's `innerWidth` property and subtrack the width of the label and a desired buffer from it. If the event X coordinate is greater than this number, the label will overflow the right side of the page and should therefore be switched to the left side of the mouse (Example 2.10).

Example 2.10: Dynamically switching label position to avoid page overflow in *main.js*

JavaScript

```
//Example 2.8 line 1...function to move info label with mouse
function moveLabel(){
    //get width of label
    var labelWidth = d3.select(".infolabel")
        .node()
        .getBoundingClientRect()
        .width;

    //use coordinates of mousemove event to set label coordinates
    var x1 = d3.event.clientX + 10,
        y1 = d3.event.clientY - 75,
        x2 = d3.event.clientX - labelWidth - 10,
        y2 = d3.event.clientY + 25;

    //horizontal label coordinate, testing for overflow
    var x = d3.event.clientX > window.innerWidth - labelWidth - 20 ? x2 : x1;
    //vertical label coordinate, testing for overflow
    var y = d3.event.clientY < 75 ? y2 : y1;

    d3.select(".infolabel")
        .style("left", x + "px")
        .style("top", y + "px");
};
```

In Example 2.10, to get the width of the label, we select the label then use the `.node()` (https://github.com/d3/d3-selection/blob/master/README.md#selection_node) operator to return its DOM node (lines 4-5). From there, we can use the native JavaScript `.getBoundingClientRect()` (https://developer.mozilla.org/en-US/docs/Web/API/Element/getBoundingClientRect) method to return an object containing the size of the label, from which we access its `width` property (lines 6-7). We use this value to set the backup x coordinate that will shift the label to the left of the mouse when it approaches the right side of the page (line 12). After setting our default coordinates (`x1` and `y1`) and backup coordinates (`x2` and `y2`), we perform each overflow test, assigning the backup coordinates if the defaults would overflow the page, and the default coordinates if not (lines 16 and 18).

Finally, the flicker issue is not really worth solving in the script; instead we can sort of sweep it under the rug by assigning the label a default position that moves it off the page entirely (Example 2.11 line 9):

Example 2.11: Adding a default `top` style to hide the label in *style.css*

JavaScript

```
.infolabel {
    position: absolute;
    height: 50px;
    min-width: 100px;
    color: #fff;
    background-color: #000;
    border: solid thin #fff;
    padding: 5px 10px;
    top: -75px;
}
```

We now have a label that handsomely follows the mouse and switches sides to avoid overflow (Figure 2.5):

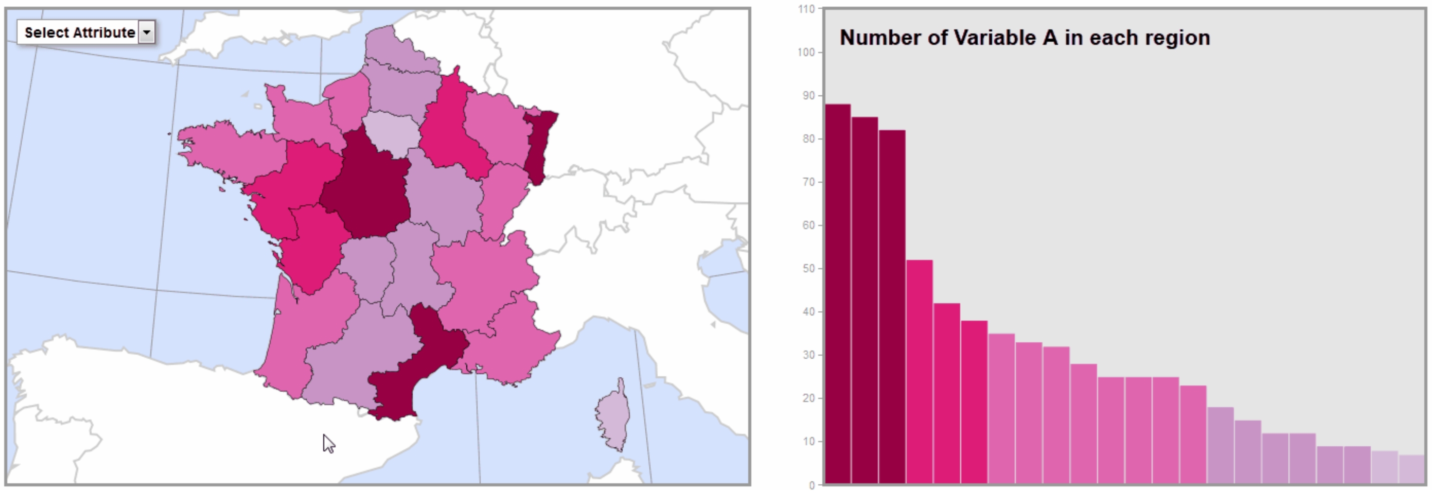


Figure 2.5: Dynamic label

> **Practice!** Implement a dynamic label on your choropleth map and linked visualization.

2.4 Extending Your Coordinated Visualization

Here ends the tutorials related to constructing your multivariate coordinated visualization. But your work is not over. If you chose to begin by following the tutorial examples, it is now time to implement your own custom UI/UX design. But don't stop there. You should use the principles of cartographic design and interaction that you have learned up to this point to push beyond the basic requirements of the D3 lab assignment and make your final product visually stunning and an experience your users will remember.

Consider implementing the following components that have not been covered in these modules:

- A dynamic choropleth legend that updates on attribute sequencing
- Other interaction operators that make sense given your dataset (*zoom, pan, search, filter, reexpress, overlay, resymbolize, reproject, arrange,* or *calculate*)
- Additional coordinated data visualizations
- Metadata and other supplementary information about the topic of your coordinated visualization
- Any other tools or features that add to the utility, usability, and/or aesthetics of the coordinated visualization

There is only so much you can learn from following along with written tutorials. You have probably already grappled with making use of online examples, documentation, and help forums to do something awesome. We have tried to open the door for you; to become a professional-level web mapper, you need to dive in and figure out the rest on your own. If you've made it this far, you should feel highly confident in your ability to do so.

> **Practice!** Add logical additional features to your coordinated visualization and finalize its user interface design.

Self-Check:

1. The SVG `<desc>` element can be used to hold which of the following types of information?

- **a.** objects
- **b.** arrays
- **c.** text
- **d.** other SVG elements

2. True/False: The .getBoundingClientRect() method is a native JavaScript method that can be used on the DOM node of an HTML element.

Module Reminder

1. Implement a visual affordance allowing the user to change the expressed attribute, and script that updates the choropleth map and data visualization in response to this user input.
2. Add animated transitions to attribute changes on the choropleth map and data visualization.
3. Implement coordinated highlighting and dehighlighting on the choropleth map and data visualization.
4. Add a dynamic label to the choropleth map and data visualization.
5. Add additional features and finalize the user interface of your coordinated visualization application.
6. Annotate your coordinated visualization with a title and value labels or one or more axes.
7. With all of the above tasks completed, commit changes to your *d3-coordinated-viz* web directory and sync with GitHub.