CSCI 4210 U – Information Visualization
Lab 7. Choropleth Maps and Color

PURPOSE

Learn how to plot geographic data, and explore the design space of color in maps.

TASKS

Choropleth maps are used to depict data that was collected for each subdivision of a geographic area. They are widely used in journalism, especially in election times. In choropleths, the only visual channel available is color, so the choice of scale and palette is critical.

We will plot US census data (unemployment rates) on a choropleth map and experiment with color scales and palettes. We chose the US map because its largest census units (counties) are nicely distributed across the map and their size is not too small relative to the map size; thus we do not need to worry about implementing zoom, which is not the focus of this lab.

Open index.html with your editor of choice. Then use python to serve the page:

```
cd lab7folder

python3 -m http.server 8080
```

Now access the page on http://localhost:8080.

1. Load the shape file and draw map boundaries

Geographic structures (locations, streets, provinces, etc.) are described in shape files. Shape files are available in varying levels of detail and in several formats. We will work with a TopoJSON file that describes the US counties. TopoJSON is a concise format because it encodes topological information. For instance, the border of California and Nevada is represented as a single path shared between the states. Other formats would encode the state boundaries independently and redundantly (borders would be described twice).

Load the file *'us.json'* with *d3.json*. In the callback, load the data file unemployment-us.csv. Then define a function *ready* and call it from the second callback. This will guarantee that when *ready* is called, both files are in memory:

```
d3.json("us.json").then(function(us){

  d3.tsv("us-unemployment.tsv").then(function(data){

      ready(us, data);

  });

});
```

Then we use the library **topojson** to extract the objects of interest from the topology (us). We are interested in counties and states:

```
function ready(us, data){

  var counties = topojson.feature(us, us.objects.counties).features;
```

```
    var states   = topojson.mesh(us, us.objects.states, function(a, b) {
return a !== b; });
  }
```

Note that for counties we called *topojson.feature*, which returns an array of instances of MultiPolygon, one for each county. We will bind these polygons to the unemployment records. These geometries would be enough for drawing the entire map and visualizing the data, but our map would not have distinct state contours, which help us visually aggregate information. We call *topojson.mesh*, which returns an object that encapsulates a single path that covers all state boundaries. Let's draw these structures.

First, create an SVG element:

```
  var svg = d3.select('body')
      .append('svg')
      .attr('width', 960)
      .attr("height", 600);
```

The variables counties and states are GeoJSON objects. The d3 function that knows how to transform these objects in SVG paths is *d3.geoPath*:

```
  var geopath = d3.geoPath();
```

Create an SVG group classed "counties", then select all paths inside it and bind counties to them. Then declare that geopath will be responsible for setting the values of path's *d* attribute. Save this selection in a variable named *countyPaths*.

```
  var countyPaths = svg.append("g")
      .attr("class", "counties")
      .selectAll("path")
      .data(counties)
      .enter().append("path")
      .attr("d", geopath);
```

If now you see a black-filled map of the United States, it is because paths by default are set to be filled. Change this setting **in your CSS (inside the style tag):**

```
.counties {stroke: gray; fill: none; stroke-width: 0.5}
```

Now draw the state mesh. In SVG elements that are added later appear on top.

```
svg.append("path")
      .datum(states)
      .attr("class", "states")
      .attr("d", geopath);
```

Note how we called *datum* to bind states to our selection. Datum is the singular of data in latin. In D3, we call datum when our data is a single element, not an array.

If now you see weird artifacts, it's again because fill is not set to none. Earlier, we set this attribute only for paths that are under the *.counties* group. Let's also change the color, so the state boundaries stand out:

```
.states {stroke: gold; fill: none}
```

2. Create a quantize color scale and fill the counties

As said before, our goal is to depict unemployment rates by county using color. The unemployment rates were loaded in the variable *data* from the file *'us-unemployment.tsv'*. **Open this file to get a notion of the data structure.** Note that counties are represented by IDs. These IDs match IDs that are stored in the county geometries. So, we can get the ID of a county object and lookup the unemployment records for the county's unemployment rate. Then input the rate on a color scale to get the corresponding color.

For the lookup, we need a structure like a hash table, where the key is the id and the value is the rate:

```
var id2rate = {};
```

```
data.forEach(function(record){
  id2rate[record.id] = +record.rate;
});
```

Choose an appropriate palette from Color Picker for Data. With a discrete array of colors and a continuous data domain, we need a quantize scale, which splits the domain in equal parts and maps them to the available colors:

```
  var color = d3.scaleQuantize()
    .domain(d3.extent(data, function(d){return +d.rate}))
    .range(arrayOfColors);  // <— needs to be defined

  countyPaths.style("fill", function(d){
    return color(id2rate[d.id]);
  });
```

3. Use a threshold color scale

The color scale above is too sensitive to outliers. Most of the color range is wasted because the outlier causes most of the values to be concentrated in one end of the domain. We can reduce this problem by allocating colors to sensible intervals of the domain. We know that unemployment values are usually between 0 and 10%. Anything higher than that is an anomaly. We can divide the domain in five intervals, covering 0-2%, 2-4%, 4-6%, 6-8%, 8-100%. The last interval includes all outliers.

D3 allows us to realize this mapping with the threshold scale. For example:

```
var color = d3.scaleThreshold()
    .domain([0, 1])
    .range(["red", "white", "green"]);
```

```
color(-1);   // "red"
color(0);    // "white"
color(0.5);  // "white"
color(1);    // "green"
color(1000); // "green"
```

Comment out your previous quantize scale and implement a threshold scale.

4. Switch to a linear scale

Comment the previous scale and switch to a linear scale. In a linear scale a **color interval** should be defined instead of a set of colors. D3 will linearly interpolate the values according to the chosen color space (RGB, by default). For example, using the interpolator for the perceptually uniform HCL color space:

```
var color = d3.scaleLinear()
    .domain([2,4,6,8])
    .range(["brown", "steelblue"])
    .interpolate(d3.interpolateHcl);


color(20); // "#9a3439"
color(50); // "#7b5167"
```

5. Try a well-tested color schemes

D3 includes many great, well-test color schemes in the form of d3 interpolators. One of them is Viridis. These schemes should be used with a sequential scale, which does not accept a range, as the range is defined by the interpolator. For instance, the following line creates a Rainbow scale:

```
var rainbow = d3.scaleSequential(d3.interpolateRainbow)
                .domain(. . .); // <- domain here as usual
```

Try the many variations of the Viridis color scheme and select your favourite to apply to the unemployment map:

1. d3.interpolateViridis
2. d3.interpolateInferno
3. d3.interpolateMagma
4. d3.interpolatePlasma

Submit index.html with all your changes, including the intermediary steps (in comments).