SI649 – Lab 3

January 23, 2015

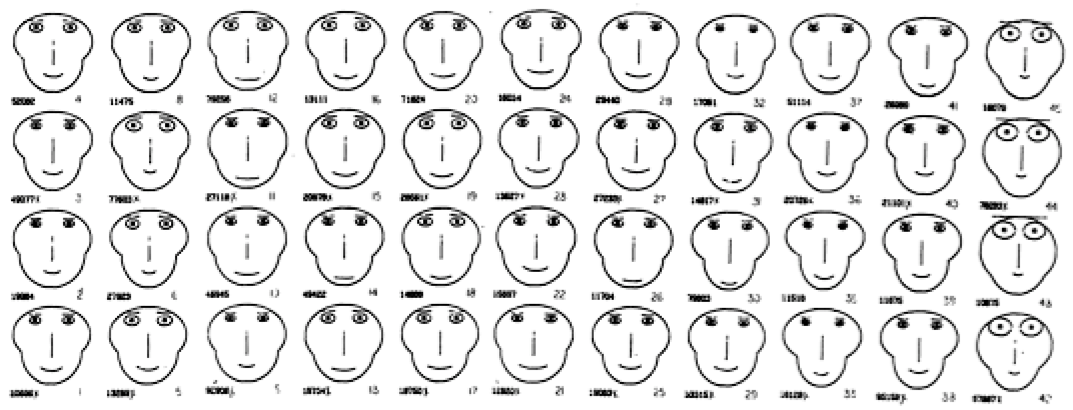
Today we will begin our exploration of d3. As before, **though some of you may get through this, I do not expect you to finish this entire lab in class.** Work through as much as you can today and then finish the rest at home to turn in for next week (upload through ctools by midnight next Thursday… January 29).

Please work with your lab partner in class. You can continue to work with them on the assignment outside of class. You may not share code with other groups/individuals (though you can talk through solutions). You may either edit the code together with your lab partner in one file, on one computer (just don't forget to share it before you leave class today) or you can work on your own laptops. Whatever is more effective for you…

What to turn in: You may hand in different HTML/JS files depending on the step. Please name your file with your name and step (e.g., eadar_step5.html and eadar_step5.js). Your name and your partner's name should be at the top of the file as a comment. Make sure that you are pointing at the right javascript file (relative or absolute path). We will deduct points if we need to modify your assignment so that it runs.

Intro

One idea for visualizing multivariate data is something called a Chernoff Face. The idea is that we can easily recognize faces so why not encode different variables in different facial features (e.g., the more MPG the car gets, the rounder the face, the more horsepower, the longer the nose, and so on).



We're going to do something similar in lab today using bear faces.

Step 0)

Grab the html files from ctools and d3 javascript file from the d3 website. Note that the d3 library name changes occasionally as they introduce new versions, you may need to change the line:

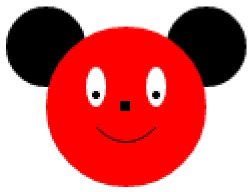
```
<script src='http://d3js.org/d3.v2.min.js'></script>
```

To make things work (I think the most recent release is: http://d3js.org/d3.v3.min.js)

Step 1)

Modify the file bear.html create the SVG elements to make something **approximating** the bear you see below. There are two ears, a face, two eye backgrounds, two eyeballs, a square nose, and a smile (you have some creative license here). There are lots of resources online, but here's a cheat sheet from SVG shapes:

http://www.math.wsu.edu/kcooper/M300/svgcheat.php



Most people find the smile the hardest part. Take a look at this if you're stuck:

http://www.w3schools.com/svg/svg_path.asp

If you get stuck during the class lab, just make the smile a straight line and move on to the next step (and fix the smile at home).

You will turn the code in after this step. Please call your file userid_step1.html (replacing userid with your name) and turn it in. If you have an associated .js file name it userid_step1.js (make sure the html links to the right js file)

Step 2)

Grab bear-program.html file. Modify the code so that you are creating the bear *programmatically*. When you're done, your bear should look exactly the same as above.

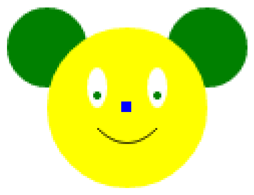
Again, most people get stuck with the smile. **As in step1, If you get stuck during the class lab, just make the smile a straight line and move on to the next step (and fix the smile at home).** Anyway, here's a hint:

```
svg.append("path").attr("d",…)
// make a path, and set its d attribute,
// you'll need to look up d
```

You will turn the code in after this step. Please call your file userid_step2.html (replacing userid with your name) and turn it in. If you have an associated .js file name it userid_step2.js (make sure the html links to the right js file)

Step 3)

Grab bear-program2.html. You should now modify the file so that you've declared a bear variable that will describe all the colors of your bear. I've started the definition for you at the top, but the program will not run until you've fixed everything. If you do this correctly, you should be able to redefine the bear *variable* so that all the colors change:



You will turn the code in after this step. Please call your file userid_step3.html (replacing userid with your name) and turn it in. If you have an associated .js file name it userid_step3.js (make sure the html links to the right js file)

Step 4)

Ok, time for something a bit more tricky. Grab bear-program3.html . We now want to make many bears using the same code.

Take a look at the code, you'll see a "dataset" of 3 bears: bobby, alex, and sam:

```
dataset = ["bear_bobby","bear_alex","bear_sam"];
```

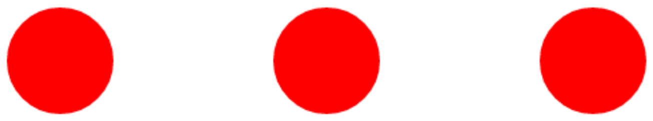
right below that you will see a loop to create a new SVG group for each bear (that's what the "enter" command will do):

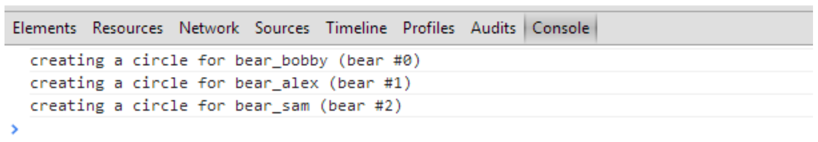
```
// create a group for each bear
         var groups = svg.selectAll("g")
            .data(dataset)
            .enter()
            .append("g");
```

After we do that, we can start piecing together the bears, but notice that we have to do it piece by piece. First we create all the faces, then we create all the left ears, then all the right ears, and so on.

```
// create a circle in each bear to represent the face
         var faces = svg.selectAll("g")
            .data(dataset)
            .append("circle");
// for each face set it to the right position
         faces.attr("cx", function(d,i) {
                 console.log("creating a circle for "+d+" (bear #"+i+")");
                 return ((i*200) + 100);
               })
         .attr("cy",50)
         .attr("r",40)
         .attr("fill","red");
```

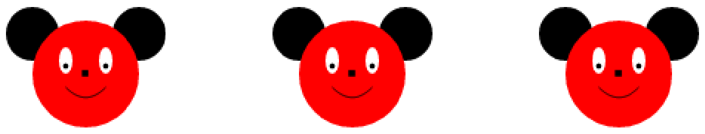
The trickiest part here is the "function" piece for the center of x. This function is used to compute the position of the face dynamically. There are two things being passed into this function "d" and "i"   d is the actual piece of data (e.g., "bear_bobby" or "bear_alex" or "bear_sam"), i is the index of that data in the overall dataset (0,1, or 2). If you open up the console you'll be able to see some debugging info:




Elements Resources Network Sources Timeline Profiles Audits Console

```
creating a circle for bear_bobby (bear #0)
creating a circle for bear_alex (bear #1)
creating a circle for bear_sam (bear #2)
```

Keep adding code to make the rest of the bear pieces, just keep repeating the pattern for face drawing. It should be clearly marked in the file. If you do this correctly, you'll end up with:



You do not need to turn in code for this step, just get it working and keep going

Step 5)

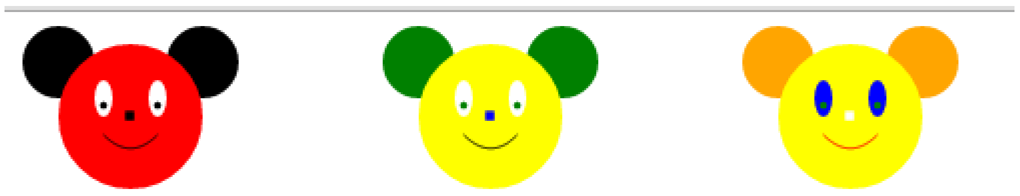
Ok, last piece… let's combine what we did in step 4 and step 5. Instead of having the bear dataset be just the names of the bears, let's actually define what they should look like. Change dataset so that it is 3 different objects (just like in step 4). Each object should define the properties for the look of the particular bear.

Something like:

```
dataset = [{“ears”:”black”,…},{“ears”:”green”,…},{“ears”:”orange”}];
```

modify your code from step 5 to use these values.

Hint: Recall that `function(d,i)` passes you the actual data element, so `d[“ears” ]` might get you something helpful. Depending on the colors you choose, you'll get something like:



In this case, you're ONLY using colors and explicitly defining them.

For example, the first bear might be:

```
{"ears":"black",
"face":"red",
"eyeback":"white",
"eyeball":"black",
"nose":"black",
"smile":"black"}
```

You will turn the code in after this step. Please call your file userid_step5.html (replacing userid with your name) and turn it in. If you have an associated .js file name it userid_step5.js (make sure the html links to the right js file)

Step 6)

Write a short paragraph explaining why Chernoff Faces may not be effective or expressive (not necessarily our bear, but Chernoff faces in general). Use what we learned about in class, but you can go beyond that if you have other reasons. Just turn this in a short text file with your name on top (please check grammar/spelling and write a *coherent* argument).

Put your answer to this question in a PDF or txt file. Save that as userid_step6.pdf or userid_step6.txt and turn it in.

Extra Credit

In step 6 the JSON for the bears explicitly defined colors. That's nice, but what we really want is for the colors and sizes of objects to be defined by some value.

So instead of:

```
{"ears":"black",
"face":"red",
"eyeback":"white",
"eyeball":"black",
"nose":"black",
"smile":"black"}
```

We might have

```
{"v1":4,
"v2":2,
"v3":5,
"v4":-1,
"v5":6,
"v6":3}
```

Where v1-v6 are variables (numerical) that correspond to facial features (ears, face, eyeback, eyeball, nose, and smile respectively). Modify the code so that you have a color gradient for each object based on the value of v1 to v6. You can pick the end points on the gradient (you can even make them all the same, but you should find the actual numerical end points dynamically (e.g., the min and max). You can steal code from your other labs to do this).

If you're feeling really adventurous, you can even modify the shape (bigger eyes, different smile, etc.) in response to the number (you can get EXTRA EXTRA credit for that).

Name your file username_ec.html

At the end, you should be submitting 3-4 html/js files (steps 1, 2, 3, 5, and possible EC) and one PDF/txt. You can zip those up if you like or just leave them separate.

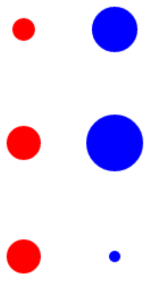
Lab 3 hints

If you're still struggling with the last piece of the assignment, let me suggest you investigate the use of the SVG "g" tag.   This is a grouping operator that doesn't actually do anything (much like a div… it just holds other stuff).

So let's say I have a really dumb dataset:

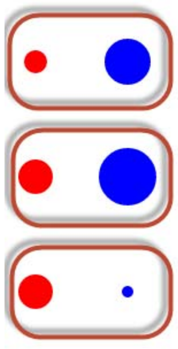
```
dataset = [ {"a":10,"b":20},   //m1
            {"a":15,"b":25},   //m2
            {"a":15,"b":5}];   //m3
```

So basically 3 elements (m1, m2, m3), with two variables a and b. For whatever reason, I want a visualization that looks like this:



The first column (the red elements) are circles, one for each piece of data. Specifically, the red circle encodes the 'a' values in the radius of the circle. So the radius of the upper left circle is 10 (m1's "a"), the next one over is 15 (m2's "a"), and the one after that is also 15 (m3's "a"). The second column—the blue circles—is the same for the "b" values. So what we have in the end is the columns representing first a and then b, and the rows representing each of the data elements.

Practically, one way to build this is to use the same grouping as we have in our data. Something like this:



Basically wrapping each pair of red/blue (a/b) circles into a group (denoted by the rectangle, which we won't actually draw).

In SVG it would look something like this:

```
<svg>
<g> <circle …> <circle …>  </g>
<g> <circle … > <circle …> </g>
<g> <circle …> <circle …> </g>
</svg>
```

The nice thing about this structure is that we can break apart the d3 process into: (1) create a new g element for every piece of data, (2) to each of the g's add a new circle for the "a" values, and (3) add a new circle for each of the "b" values. The appeal is that we only need to call the whole "enter()" stuff once. We are creating groups and then we "append" the circles to those groups.

Our code would look something like this:

```
 // make the svg
var svg = d3.select("body").append("svg")
           .attr("width",1000).attr("height",500);

// dataset
dataset = [ {"a":10,"b":20},  //m1
            {"a":15,"b":25},  //m2
            {"a":15,"b":5}];  //m3

// create a grouping operator, one for each
var groups = svg.selectAll("g")// grab all g's (we have none)
   .data(dataset) // loop over dataset
   .enter()       // this will make us as many "g's" as we have data
   .append("g");

// CREATE THE RED CIRCLE
// into every group, we're going to first draw
 // a circle for the "a" dimension.  Notice that we don't
 // have to "enter" anymore.  There are 3 pieces of data
// and we previously created 3 g's.
groups
.append("circle")
 .data(dataset)   // we don't actually need this line
.attr("cx",20) // always in the same column, but move the cy:
.attr("cy", function(d,i) {return ((i*100) + 50);  })
.attr("r",function (d) {return d['a'];})
.attr("fill","red");

// CREATE THE BLUE CIRCLE
groups
.append("circle")
.data(dataset)   // we don't actually need this line
.attr("cx", 100)  // always in the same column, but move the cy:
.attr("cy", function(d,i) {return ((i*100) + 50);  })
.attr("r",function (d) {return d['b'];})
.attr("fill","blue");
```

The other cool thing about this approach is that D3 will propagate your data to nested objects. Because we've bound "dataset" to our "g" objects, the data will get pushed to the circles that are contained inside those "g" objects. So strictly speaking, we can get rid of the lines in red and it would still work.

Hint for Extra Credit and Chernoff Faces

Here's an example dataset (it's not about bear faces, it's about cars, obviously with made up numbers)

```
dataset = [ {"mpg":30,                  // car 1
               "hp":100,
               "price":1000,
               "cylinders":4,
               "torque":5,
                "zeroTo60":20},
             {"mpg":32,                 // car 2
                "hp":200,
                "price":1500,
                "cylinders":6,
                "torque":7,
                "zeroTo60":100},
             {"mpg":20,                  // car 3
                 "hp":120,
                 "price":2000,
                  "cylinders":8,
                  "torque":10,
                  "zeroTo60":4}];
```

What I want you to do is to map the variables above (mpg, hp, price, cylinders, torque, zeroTo60) to the facial features of the bear (e.g., ears, face, eyeback, eyeball, nose, smile).

So let's say we're mapping mpg to ear colors. Mpg ranges from 20 to 32 and if I'm using colors, maybe I want the minimum mpg to be red and the max to be blue. So 20 should be red, 32 should be blue, and 30 should be somewhere in the middle. Your task for the extra credit is to support this.

- First: map the arbitrary dataset (not too arbitrary, you can define how many variables you have… probably something in line with the number of bear face pieces).
- Second: calculate the range the original variable can take (e.g., 20-32 for mpg, 100-200 for hp, etc.)
- Third: map those values to some color gradient (it can be the same color gradients, e.g., red—blue but note that each variable has it's own min and max… so red for mpg is 30, but is 100 for hp)
- Fourth: render the bear using those mappings variable → face element, and variable value → color

If you're doing the "extra extra credit" which is actually the way Chernoff faces work, you'll be mapping the ranges to *sizes* of the face elements instead of the colors. So mpg may still map to the ears, and hp to the nose, but now we're going to make the size of those face elements vary (so high mpg means big ears, low mpg small ears, and so on…). Instead of mapping the range (e.g., 20-32) to colors (red—blue), you'll be mapping them to circle radius (for example)… so: 10-30 (again, for example).