Lab 3: Visualization with d3.js

SDS235: Visual Analytics
22 February 2017

Introduction & Setup

In this lab, we will cover the basics of creating visualizations for the web using the d3.js library, which can be found at: http://www.d3js.org. First we'll build a bare-bones bar chart using only HTML elements. Next, we'll build a more flexible version using Scalable Vector Graphics (SVG). Finally, we'll read in some actual data and add axes, labels and interaction.¹ You have several options available for completing this lab:

JSFiddle
If you want to focus on the javascript, you might want to try https://jsfiddle.net, an online resource for testing and running JavaScript code right from your browser:

Under Frameworks & Extensions, you’ll want to select d3 3.x to include the d3.js library. To test it out, simply enter some JS into the JavaScript window, and click the Run button! Get stuck? Try the JSHint button.

¹This lab is based on the “Let’s Make a Bar Chart” tutorial series by Mike Bostock, creator of d3.js. The original tutorial series can be accessed at http://bost.ocks.org/mike/bar/.
On Your Own Machine
If you’d like to run the code from your local machine, you can download a template for this lab at:

http://www.science.smith.edu/~jcrouser/SDS235/labs/d3.zip

Upzip the archive, and navigate to the d3 directory. To be able to load external data, you’ll need to set up a web server, which you can do using the following command:

$ python -m SimpleHTTPServer 8888 &

This starts up a simple web server running on port 8888 on your machine. To access the page, point your browser to http://localhost:8888/lab-03.html (which will be blank to start with). Whenever you edit lab-03.html, just refresh the page to see your changes; you don’t need to restart the server every time.

1 Creating a simple Bar Chart

Let’s start off by creating a small array of numeric data:

```javascript
var data = [4, 8, 15, 16, 23, 42];
```

If we wanted to create a basic bar chart by hand, we could do something like this:

```html
< div class= "chart">
  < div style= "width: 40 px;">4</ div>
  < div style= "width: 80 px;">8</ div>
  < div style= "width: 150 px;">15</ div>
  < div style= "width: 160 px;">16</ div>
  < div style= "width: 230 px;">23</ div>
  < div style= "width: 420 px;">42</ div>
</ div>
```

And then style the chart using CSS like this:

```css
chart div {
  font: 10 px sans-serif;
  background-color: steelblue;
  text-align: right;
  padding: 3 px;
  margin: 1 px;
  color: white;
}
```

Together, that produces:

![Bar Chart Example](image-url)
Because we only have a handful of data points, we could feasibly create 6 individual `<div>` elements one at a time. For most datasets, this would be impractical; we’d like to instead be able to create charts from data automatically.

### 1.1 Getting started with Selections

So now let’s try to create an identical chart using d3, starting with an empty page that contains only a `<div>` of class `.chart`:

```javascript
1 | <div class="chart"></div>
```

And a JavaScript array containing the data:

```javascript
1 | var data = [4, 8, 15, 16, 23, 42];
```

Rather than dealing with elements one at a time, d3 (along with jQuery and other libraries) instead handles groups of related elements called selections. We can use a class selector to grab the chart container like this:

```javascript
1 | var chart = d3.select (".chart");
```

### 1.2 Joining Data to a Selection

Next, we’ll want to initiate the data join by defining the selection to which we will join data:

```javascript
1 | var bar = chart.selectAll("div");
```

Notice that the chart doesn’t currently contain any `<div>` elements. That’s okay! We’re about to create them. You can think of the initial selection as a declaration of the elements you want to exist. Let’s join the data array to the selection using the `.data()` function:

```javascript
1 | var barUpdate = bar.data(data);
```

### 1.3 Creating missing elements on `.enter()`

Since there original selection was empty, the returned update and exit selections are also empty. We need only handle the enter selection, which represents new data for which there was no existing element. We can create these missing elements by appending to the enter selection:

```javascript
1 | var barEnter = barUpdate.enter().append("div");
```

Because these elements were created using a data join, each bar is already bound to data. This means that if we want to set the width of each new bar proportional to the associated data value, we can reference the data for each element:

```javascript
1 | barEnter.style("width", function(d) { return d * 10 + "px"; });
```

---

2For more detail, see “Thinking with Joins” by M. Bostock, available at [http://bost.ocks.org/mike/join/](http://bost.ocks.org/mike/join/)
We can use a similar function to set the text content of each bar and make a label:

```javascript
1 barEnter.text(function(d) { return d; });
```

To clean this up, we can chain together all the methods from Sections 1.1 to 1.3, which would look something like this:

```javascript
1 d3.select(".chart")   // Select the container
2 .selectAll("div")     // Select the elements we want to bind to
3   .data(data)          // Bind the data to the selection
4   .enter().append("div") // Append/style new elements
5   .style("width", function(d) { return d * 10 + "px"; })
6   .text(function(d) { return d; });
```

### 1.4 Using Scales

One weakness of the code we’ve written so far is its dependence on the magic number 10, which we used to scale each bar to the appropriate pixel width. But what if we wanted to make our chart a bit larger, say 430px? That would be a pain to calculate manually. We can eliminate the magic number problem by using scales, which allow us to specify a mapping from data space (domain) to display space (range):

```javascript
1 var x = d3.scaleLinear()
2   .domain([0, d3.max(data)])
3   .range([0, 430]);
```

We can now use `x` as a function, which will return the appropriate value in the range for any given data value in the domain. For example, an input value of 4 returns 40, and an input value of 16 returns 160. To use the new scale, just replace the hard-coded multiplication by calling the `scale()` function:

```javascript
1 d3.select(".chart")
2   .selectAll("div")
3     .data(data)
4   .enter().append("div")
5     .style("width", function(d) { return x(d) + "px"; }) // Ta-da!
6     .text(function(d) { return d; });
```

Great! Now you should (again) have something that looks like this:

[Image of a bar chart with bars of different heights]
2 Working with SVGs

One problem of working with pure HTML is that we’re pretty much restricted to rectangular shapes. While this is sufficient for building a bar chart, there are other visualizations that will need more complex primitives like Bézier curves, gradients, clipping and masks. For that, we’ll need to use Scalable Vector Graphics, or SVGs.

2.1 Getting started

We can start with the same basic setup we had in pt. 1, replacing the empty `<div>` with an `<svg>`:

```html
<svg class="chart"></svg>
```

And of course we’ll still need our data array:

```javascript
var data = [4, 8, 15, 16, 23, 42];
```

In the style section, we’ll want to break out the styling for the `rect` and `text` elements, as they will be defined separately when using SVG:

```css
.chart rect {
  fill: steelblue;
}

.chart text {
  fill: white;
  font: 10px sans-serif;
  text-anchor: end;
}
```

Now let’s set up the dimensions of the `<svg>`, and define our scale:

```javascript
var width = 420,
    barHeight = 20;

var x = d3.scaleLinear()
  .domain([0, d3.max(data)])
  .range([0, width]);

var chart = d3.select(".chart")
  .attr("width", width)
  .attr("height", barHeight * data.length);
```

2.2 Joining data to SVG elements

Now we’d like to bind the data to SVG group or `<g>` objects, which will eventually contain the `rect` and `text` for each bar. While we’re at it, let’s put each bar in the right place as well:
```javascript
1 var bar = chart.selectAll("g")
2 .data(data)
3 .enter().append("g")
4 .attr("transform",
5     function(d, i) {
6         return "translate(0," + i * barHeight + ")";
7     });

Remember that all the new <g> objects we just created are now accessible from bar. For each bar,
we want to append an appropriately-sized rect and an appropriately-positioned text label:

```javascript
1 bar.append("rect")
2 .attr("width", x)
3 .attr("height", barHeight - 1);
4
5 bar.append("text")
6 .attr("x", function(d) { return x(d) - 3; })
7 .attr("y", barHeight / 2)
8 .attr("dy", ".35em")
9 .text(function(d) { return d; });

If all goes well, we should be... right back to where we started:

![Bar chart example]

3 Creating charts from real data

Now that you’ve built a basic bar chart in HTML and then in SVG, let’s polish up the display by
rotating the chart into columns and adding axes. We’ll also switch to using a real dataset, showing
the relative frequency of letters appearing in the English language.

The data in tabular form looks something like this:

<table>
<thead>
<tr>
<th>letter</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>.08167</td>
</tr>
<tr>
<td>B</td>
<td>.01492</td>
</tr>
<tr>
<td>C</td>
<td>.02782</td>
</tr>
<tr>
<td>D</td>
<td>.04253</td>
</tr>
</tbody>
</table>

³If you’re using JSFiddle, you’ll need to copy/paste the the data into a JavaScript variable called data. The
dataset can be found at: [http://www.science.smith.edu/~jcrouser/SDS235/demos/lab-02/data.json](http://www.science.smith.edu/~jcrouser/SDS235/demos/lab-02/data.json)
3.1 Getting started

Let’s start again with just an empty `<svg>`. To give us room to add some axes later, we can add margins and move the container over a bit:

```javascript
var margin = { top: 20, right: 20, bottom: 30, left: 40 },
width = 960 - margin.left - margin.right,
height = 500 - margin.top - margin.bottom;

var svg = d3.select("body").append("svg")
  .attr("width", width + margin.left + margin.right)
  .attr("height", height + margin.top + margin.bottom)
  .append("g")
  .attr("transform", "translate(" + margin.left + "," + margin.top + ")");
```

We’d like our chart to be oriented vertically this time, so we’ll need a scale for our y values:

```javascript
var y = d3.scaleLinear()
  .range([height, 0]);
```

The x axis is a little trickier, since we are now working with `ordinal` values. Luckily, d3 has ordinal scales as well, and we can use the `.scaleBand()` function to split the range up into bins:

```javascript
var x = d3.scaleBand()
  .rangeRound([0, width], .1)
  .paddingInner(0.1);
```

3.2 Loading external data

You’ll notice that we haven’t included a `domain` on either or our scales. This is because we haven’t loaded the data yet. Let’s do that now using the `.tsv()` function:

```javascript
d3.tsv("letters.tsv", function(error, data) {
  // We'll fill this in shortly
});
```

Every operation that depends on that data will go inside this function. Otherwise, because JavaScript is asynchronous, the operation might try to execute before the data is done loading.

Once the data loads, we can add our `domains`:

```javascript
d3.tsv("letters.tsv", function(error, data) {
  x.domain(data.map(function(d) { return d.letter; }));
  y.domain([0, d3.max(data, function(d) { return d.frequency; })]);
});
```
3.3 Adding axes

Now we can build our axes:

```javascript
var xAxis = d3.axisBottom()
    .scale(x);
var yAxis = d3.axisLeft()
    .scale(y)
    .ticks(10, "%")
```

We can add the axes to the chart using the `.append()` function. Let’s start with the x axis, using the `.translate()` transform to move it to the bottom of the SVG:

```javascript
// This needs to happen inside the call to .tsv(...)
svg.append("g")
    .attr("class", "x axis")
    .attr("transform", "translate(0," + height + ")")
    .call(xAxis);
```

For good measure, let’s add in our CSS styling too:

```css
.bar { fill: steelblue; }
.axis text { font: 10px sans-serif; }
.axis path, .axis line {
    fill: none;
    stroke: #000;
    shape-rendering: crispEdges;
}
.x.axis path { display: none; }
```

Great! That gave us:

And now the y axis, which by default is positioned on the lefthand side of the SVG:

```javascript
# Still inside .tsv(...)  
svg.append("g")
    .attr("class", "y axis")
    .call(yAxis)
```
Now we’ve got:

```javascript
3.4 Finishing touches

Finally, we’ll want to add in the bars for each letter:

```javascript
1. # And STILL inside .tsv(...)
2. svg.selectAll(".bar")
3. .data(data)
4. .enter().append("rect")
5. .attr("class", "bar")
6. .attr("x", function(d) { return x(d.letter); })
7. .attr("width", x.bandwidth())
8. .attr("y", function(d) { return y(d.frequency); })
9. .attr("height", function(d) { return height - y(d.frequency); })
```

And perhaps add a title, centered at the top of the chart:

```javascript
1. # This doesn’t depend on the data, so it can happen outside .tsv(...)
2. svg.append("text")
3. .attr("x", (width / 2))
4. .attr("y", 0 + (margin.top / 2))
5. .attr("text-anchor", "middle")
6. .text("Relative Frequency of Letters in the English Alphabet");
```
All together, this should result in something that looks like this:

![Relative Frequency of Letters in the English Alphabet](image)

And there you have it! You’ve built your first data visualization using d3.

**Getting Credit**

To get credit for this lab, respond to the prompt on the #lab3 slack channel.

**Want to learn more?**

Check out:


for documentation and tutorials.

Looking for examples of cool stuff other people have done with d3? Try the galleries at:

- [http://christopheviau.com/d3list/gallery.html](http://christopheviau.com/d3list/gallery.html)
- [http://c3js.org/examples.html](http://c3js.org/examples.html)