CSE 564 (Visualization) – Lab 1

Name : Viraj Kamat
SBU ID : 112818603

Dataset and cleaning

The dataset was obtained from kaggle.com. It mainly consisted all the cars manufactured from 1990 onwards from different car manufacturers in the USA. Since the dataset was very large it was reduced to only account for cars manufactured 2013 and later.

The main challenge was finding a dataset that contained a good mix of numerical and categorical variables. The cars dataset was a good way to go as it contained good mix of those along with good number of datapoints.

I used basic python programming to filter the data, those cars that had major fields empty were discarded. In some cases an average was taken to assign the appropriate value to an empty field. Another way we could fill missing values was by looking at the same car model manufactured in a different year and fill up details that were missing, for eg. Mazda RX6 whose car category field was missing for the year 2017 could be filled by looking at the same car model in the other years, thus many empty categorical fields were identified this way.

Some information such as car transmission, car fuel, etc was not present in the initial dataset. I searched for other car datasets and made a unique combination of the car manufacturer, the model and searched for this same key in other car datasets to obtain more fields. This way the number of dimensions was expanded to 15.

External libraries used

The chunk of the implementation was done using d3.js. However, to enable better rendering of the html document bootstrap library was used for html alignment/display. jQuery for better DOM manipulation and lastly to have a good color palette for the barchart and histogram bars I used palette.js.
To enable dynamically changing the bins for the histogram, d3-slider.js was used that would create a slider that could be slid to the left or right decreasing or increasing the number of bins, this would then update the histogram accordingly.

Building the main layout

The main layout consisted of a dropdown menu, that contained the list of the variables/dimensions for whom the barchart or histogram had to be rendered. At the center of the html document lay the main svg container, the svg container was then worked upon with the help of d3.js functionality to create the charts. For each variable or dimension that was selected from the dropdown there was a separate logic that would handle the creation of the chart.

Loading the data

The whole data present in a csv file, this csv file was loaded just once using the the d3.csv function and stored in a global variable. Every time a column was selected from the dropdown, the appropriate field was selected from the global variable containing the csv data, this was done for each field.

Once the field was picked from the csv file it was then processed to create a variable that could be sent to the two main functions in my code – renderChart() and renderHistogram().

The dataset passed to these two function were as follows :

1. For a barchart, a json object of key value pair. Key being the category and the value being the count
2. For the histogram, all the numerical values were grouped in a json object.
3. Both functions took the labels for the x and y axis as parameters. This would be applied when creating the axis.

Building the Barchart

The renderChart function took the dataset, the x and y axis labels. To begin with I first created the x axis scale and the y axis scale, because given the with and height of the **svg** container within which we had to create the plot we had to know the width, height the x value and the y value of the bar that was to rendered. This following code snippet elaborates the usage :

```
//Create a variable used for scaling the values across the x axis.
var xscale = d3.scaleBand()
.range([0,width])
//Similarly create a variable for scaling across the y axis.
var yscale = d3.scaleLinear()
.range([height,0])
xscale.domain(dataset.map(function(d){
    return d.type.replace('_',' ').toUpperCase()
}))
yscale.domain(
    [0,d3.max(dataset,function(d){ return d.value })*1.5]
)
 var y_axis = d3.axisLeft()
                .scale(yscale);
```

The domain here for the x axis scale is simply the unique list of categorical variables, the range being the width, this would generate the appropriate parameters used to render the bar.

Similarly, for the y axis the domain would be 0 to the maximum numerical value of a category to identify the range in which to render the bar.

Note that in the yscale scaling function for the y axis we have set the range in the opposite direction, this is because the svg container starts from the top left – the other way round compared to a regular graph.

Once the **svg** container was selected from the html document using d3.select() the next step was appending the bar. This was done by appending a **rect** element sequentially to the svg container.

The following code snippet elaborates on the same :

```
    //Append each rectangular element denoting the bars
    rect_margin = margin + 10
    groups
    .append('rect')
    .attr('class','bar')
    .attr('x',function(d){  return xscale(d.type.replace('_',' ').toUpperCase()); })
    .attr('width',xscale.bandwidth()*.90)
    .attr('y',function(d){ return yscale(d.value); })
    .attr('height',function(d){ return height - yscale(d.value) })
    .attr('transform','translate('+rect_margin+',0)')
    .attr('fill',function(d,i){
        //Depending on the value fill with appropriate color
        return("#"+color_scale(d.value))

    })
    .on("mouseover", function(d) {
```

As we see above the xscale and yscale scaling functions are being called with the respective category variables and its numeric value; this then helps compute the width, height, x axis value and the y axis value on the svg container.

For each of the bar chart elements I then appended a text element that contained the numeric value for that bar, though it was hidden. I also added mouseover and mouseout event handlers for each of the bar **rect** elements on the bar chart. Whenever the user would hover over the bar, the mouseover event handler of d3 along with my code would make the bar a bit higher, a bit wider and set the text element appended to each to be visible, this would display a text of the numerical value of that bar just above the bar.

Lastly and most importantly I added the x and yaxis. The code below elaborates how:

```
    //Append the y axis
     var y_axis = d3.axisLeft()
               .scale(yscale);
    svg.append("g")
       .attr("transform", "translate("+margin+", 0)")
       .call(y_axis)

    //Add x-axis
    svg.append("text")
         .attr("x", width/2 )
         .attr("y",  height + margin + 12 )
         .style("text-anchor", "middle")
         .text(xlabel)
         .attr('fill','#f00')
         .style('font-weight','bold')
```

The x and y axis also had their respective labels. This was as simple as appending a text to each of them, the difference for the y axis being that its label was rotated by 90 degrees.

Building the Histogram

Building the histogram consisted of two functions, the first function called the preHistogramRender() took the dataset, the x/y axis labels and simply created the slider required for updating the bins of the histogram. The event handler when sliding the bins was then modified to compute the histogram with the renderHistogram() function with the new bin size.

From this function we call the renderHistogram() function, however when we move the slider to update the bin values of the histogram the mouse handler of the slider would take the required action.

The renderHistogram function takes in the dataset containing list of numerical values along with the x and y axis labels.

As with the bar chart the first step was to create the scaling function for the x axis which took the minimum and maximum values of the dataset that would give us the appropriate x axis values for each of the bars.

We then use the d3.histogram() function to create the bins as we pass our dataset into it. The bins returned from this function contains and array or arrays, with each element containing the values in each bin. These bins were then passed into our y scaling function which would help us compute the height of the histogram bar, the following code snippet elaborates its working :

```
//Create a scaling function that will enable appending values to the x axis
var xscale = d3.scaleLinear()
.domain([minimum,maximum])
.range([0,width])

//The main histogram function, when this is called with the dataset we create the bins
var myHistogram = d3.histogram()
.domain(xscale.domain())
.value(function(d){
    return d.value;
})
.thresholds(xscale.ticks(ticks));

//Create a scaling function for the yaxis, we pass the bins (array of array)
var yscale = d3.scaleLinear()
.range([available_height,0])
.domain([0,d3.max(bins,function(d){
    return d.length
```

Then in the **svg** container I appended unique **rect** elements for each of the bins, its width/height and position computed with the help of the scaling functions built earlier. The code snippet below highlights its working:

```
 //For each group append the rect required for the histogram bar
 groups
 .append("rect")
 .attr("x", 1)
 .attr("transform", function(d) {
   var xstart = xscale(d.x0)+margin
   return "translate(" + xstart +"," + yscale(d.length) + ")";
 })
 .attr("width", function(d) {      //x0 and x1 are the end and starting x values for a bin.
   return (xscale(d.x1) - xscale(d.x0));
 })
 .attr("height", function(d) {     // Compute the required height from the scaling function we built
   return available_height - yscale(d.length);
 })
 .attr('class','bar')
 .attr('fill',function(d){
     return ('#'+color_scale(d.length))
 })
.on("mouseover", function(d) {
   //On mouse over redice the opacity of other bars.
   d3.selectAll('.bar')
   .style('opacity','0.5')
```

For each of the **rects** I appended a text element containing the its numeric value, that is number of elements in that bin, this however was hidden. I added mouseover and mouse out even handlers to each of the **rect** elements that renders the bar. When we do hover over each of the bar the event handler makes the bar a bit higher and a bit wider and makes the text element containing the value of the bar visible. I also added a mouse out event handler which would return the bars to their original position.

I lastly added the x and y axis to the svg container, this is shown in the code snippet below :

```
    //Append the y axis
    svg.append("g")
    .attr("transform", "translate("+margin+", 0)")
    .call(d3.axisLeft().scale(yscale));

    //Append the y axis label, note that it is rotated by 90 degrees
    y_margin = 0 - margin
    svg
    .append('g')
    .append("text")
    .attr( "transform", "rotate(-90)" )
    .attr("y", y_margin )
    .attr("x",0 - (height / 2))
    .attr("dy", "1em")
    .style("text-anchor", "middle")
    .text(ylabel)
    .style("fill", "red")
    .style('font-weight','bold')

    //Similarly append the x axis
    svg
    .append("g")
    .attr("transform", "translate("+margin+"," + available_height  + ")")
    .call(d3.axisBottom(xscale).ticks(ticks))
    .selectAll('text')
    .attr("transform", "translate(-10,10)rotate(-45)")
```

Use of color palettes

The library palette.js was used to create a range of colors for the bars in the barchart and histogram. All we had to do is specify the minimum and maximum value of each of the bars to the palette function and it would output an array containing the range of colors.

For each of the bars in the chart we simply modify the **fill** attribute based on the color array that we pass. For higher values the color applied is darker and the lower values it is lighter.

The following code snippets show the creating of the color array and its application to the bars :

```
//Create a color palette, will be used to color the divs of the histogram
var color_pal = palette('cb-PuBu',9)
var color_scale =  d3.scaleQuantize().domain([0,d3.max(dataset,function(d){
    return d.value
})]).range(color_pal)
```

```
.attr('fill',function(d,i){
    //Depending on the value fill with appropriate color
    return("#"+color_scale(d.value))

})
```

Note that when creating a scaling function we use scaleQuantize() that will create a range of discrete color values for the bar.