CS/INFO 3300; INFO 5100
Homework 4
Due 11:59pm Wednesday, September 29

Goals: Practice using d3 to create some simple charts. Get more experience importing data and working with loops.

Your work should be in the form of an HTML file called index.html with one `<p>` element per problem. If you must add any SVG canvases programmatically, we suggest that you add a `<div>` element to the HTML to hold each SVG to make positioning a bit easier. For this homework we will be using d3.js. **In the <head> section of your file, please import d3 using this tag: <script src="https://d3js.org/d3.v7.min.js"></script>**

Create a zip archive containing your **HTML file and all associated data files** (such as diamonds.json) and upload it to CMS before the deadline. Submissions that do not include data files may be penalized. Your submission will be graded using a Python web server run in a parent directory containing your zip file contents (e.g. server started in `~/student_hw`, with your homework at `~/student_hw/your_netid/hw3/index.html`) – be sure that it works.

1. In this problem we're going to plot some data about diamond sales. The file `diamonds.json` contains a JSON block that defines an array of objects. Each object represents a particular diamond sold in a marketplace. These have been randomly sampled from a much larger dataset. In addition to numeric columns for price and size (carats), the dataset contains a color rating value where 1 is the best value.

A. Load the data file using an asynchronous request with `d3.json`. You can use `.then()` to handle the promise. **Make sure all of your code for this problem is within the `.then()` function!**

B. Create a 400x400px SVG element using d3 functions. Create two linear scale functions: an x scale for "`carat`" and a y scale for the "`price`". Make the domain start at `0` and end at the maximum value for each respective attribute. Choose the "range" attributes to be appropriate for the size of your plot. The plot can use the entire SVG canvas, but reserve 5 pixels at the top and right as **padding** using your range to make space for the maximum points so they do not clip at the edges. Remember to consider for the way SVG coordinates handle Y when using your y scale; we don't want any upside-down charts. Using any kind of loop you prefer, use d3 and the scales you build to construct `<line>`

element gridlines, **including 0**, for **every integer** for the x axis and **every $1,000** for the y axis on the SVG canvas. Do not create any text labels. Style lines however you want.

C. Using a `forEach` loop in Javascript, create a **3px radius circle** for each point in the dataset, located at its proper place on the chart. Use your scales to place the points and employ `Math.floor()` to make sure you place each circle on an **integer pixel location**. Fill each point in a **dark blue** color of your choice. Set the `opacity` SVG attribute of each circle to 0.4 to make them translucent and show point density.

D. Now let's look at the color rating of each gem. Create a **second** 400x400px SVG element using d3 functions. Instead of charting "`carat`" on the x axis, chart the "`color_rating`" variable using an appropriate scale (hint: modify a copy of step C rather than starting from scratch). Recall that `color_rating=1` is the *best* value. While `color_rating` ranges from 1 to 7, set the domain of your new x scale to [0,8] so that you have more space for columns of points. Create vertical gridlines for integers in range [1,7]. If you feel that including a gridline at 0 would make the chart look better, feel free to do so. You will not need any padding on this axis since you reserved space in the domain. As in part B, create horizontal gridlines for every $1000 on the y axis. Finally, create a **3px radius circle** for each point in the dataset. To locate the points, use your scales and **add a random 5px jitter to each point location** (i.e. `location + (Math.random()*10)-5`). Make sure to `Math.floor()` that result so that you place each circle on an **integer pixel location**. Fill each point in a **dark green** color of your choice. Set the **opacity** SVG attribute of each circle to 0.3 to make them translucent.
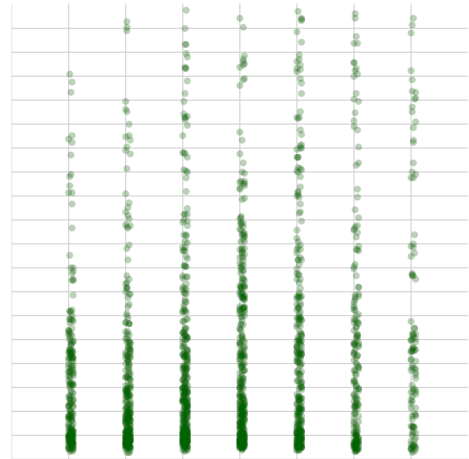
E. There's not much of a trend to see in this chart? This is because the new plot combines points of all sorts of different carats together. A low quality gem might be priced the same as a rank 1 gem if it is absolutely gigantic. Create a **third** 400x400px SVG element using d3 functions. Instead of charting "`price`" on the y axis, create **a new measure that shows the price per carat** of each point (hint: modify a copy of step D rather than starting from scratch). You do not need to record this new measure into the data; rather, just build it on the fly as you construct scales, gridlines, and circles. Create a new y scale to account for this new measure, and draw gridlines every $1000 per carat, starting at 0. Create a **3px radius circle** for each point in the dataset. To locate the points, use your scales and **add a random 5px jitter to each point location** (i.e. `location + (Math.random()*10)-5`). Make sure to `Math.floor()` that result so that you
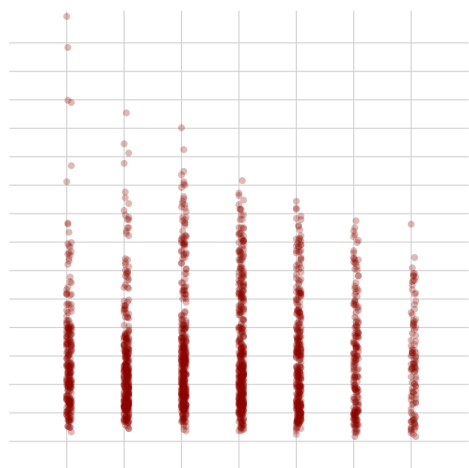
place each circle on an **integer pixel location**. Fill each point in a **dark red** color of your choice. Set the **opacity** SVG attribute of each circle to 0.3 to make them translucent. This view ought to **suggest a relationship** between price per carat and quality. Unfortunately, statistical testing shows no such relation. In the <p> tag of your submission, briefly discuss **why this chart might mislead a viewer into seeing a pattern** that isn't present.

2. For this problem we have processed a public dataset of NCAA Basketball games from the 2018-2019 and 2020-2021 seasons (aggregated from [data](#) scraped by Luke Benz). Our goal is to recreate a popular heatmap visualization made by [Max Woolf](#). You can see our final version of it to the right. Areas of high activity in the game are colored yellow and areas of low activity are black. Marks are individual <rect> squares and channels are aligned position and color hue+luminosity.
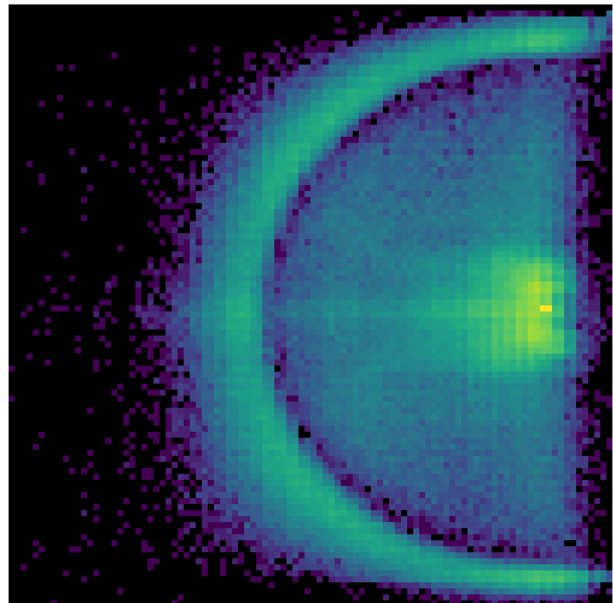
To make this visualization we have added up all the shot attempts made by players at different locations in the court. A shot attempt refers to when a player throws the ball attempting to get it into the opposing team's basket to score points (where and when they can throw is influenced by rules and team strategy). You can see the basketball hoop in the middle bottom and an arc created by the "3 point line" that dictates where players can throw the ball to earn more points.

A. In your HTML, create **a 500x500px SVG element**. Use CSS styles to give it a **black background**. Now load the included data file NCAA_shots.csv by using an **asynchronous request** (i.e. `d3.csv().then()`). Implement the rest of this problem in the promise function. Use console.log() to check out the data you're using for this problem. You will notice that each element contains x, y, `width`, and `height` values for making the colored rectangles. The other keys contain different kinds of count data about what shots happened in a specific area of the court. For this assignment we'll start by examining the total number of all successful and missed shots at a location: `attempt`.

We have one challenge to tackle first. The dataset comes with x and y positions ranging from 0 to 100, but our SVG is 500 pixels in size. We need to adjust the x, y, `width`, and `height` values so that they match. At the top of your promise function, use a `forEach()` loop to **alter the data**. For each point in the dataset, **multiply x, y, `width`, and height by 5** so that they range from 0 to 500. You could also do this with scales, but this is far more convenient.

Now create a **new sequential color scale** for the heatmap. Use `d3.extent()` to **figure out the extent** of `attempt` in the dataset. Then, make a sequential color scale using that as your domain. Use the `d3.interpolateViridis` color scale in your sequential scale (hint: docs).

Finally, use a `for` or `forEach` loop to **create new <rect> elements for each row** of data in your dataset. As the dataset already now has correct `x`, `y`, `width`, and `height` values, configuring the rect elements should be straightforward. **Adjust the fill of the rectangles** using your color scale and the `attempt` value. Please note that at this stage, **your final visualization will not look like the example image**. It will be mostly dark blue.

B.   There is something odd with the visualization you've created. If you've done it properly, you should see **an intensely yellow blob and not much else**. This is because the data have an *exponential distribution*. Close to the basket there are many, many more attempts than far away. This causes the color scale to assign yellow to a large value while all other values are so small that they receive virtually the same color at the bottom of the scale. One common approach for resolving this issue is to **use a logarithmic scale instead of a linear scale** (which scaleSequential uses). While there are ways to do this with d3 scales, they are needlessly complex. Instead, **we have provided for you another data attribute:** `log_attempt`.

Adjust your code so that you use `log_attempt` instead of `attempt` for your rectangle fill color. You should only need to change your `d3.extent()` call and `"fill"` setter.

Compare that result with your previous visualization. **In 2-3 sentences in your <p> tag, please describe one advantage and one disadvantage of the logarithmic color scale as compared to the original, linear scale.**

You do not need to submit both versions of #3. Only submit the version that uses `log_attempt`.