INFO/CS 3300
Final Exam
Due Sunday, May 22 at 11:59pm (no extensions, no late work)

You will find four HTML files in the zip file along with this file, which correspond to the four sections of this exam. You will answer all questions by editing these files. Some questions involve coding, some involve short written responses. When you edit Javascript code, the graders will appreciate if you add a comment stating what you changed or added. For short answer questions, there will be a `div` element in the HTML file for the section containing the problem. Clearly state the question number when writing your answers. When you are done, save your edits and package all files in a new zip file. Turn this zip file in through CMS.

This exam is due Sunday, May 22 at 11:59pm. There is no specific time limit. You do not need to finish it in one session. You are, in fact, strongly encouraged to work on it over the course of the week.

You may use online references such as APIs and all class notes from GitHub. Reading previous posts from question forums like StackOverflow is permitted, but "official" resources like the D3 docs are preferred, and more likely to be useful. **You may not communicate with any other person in any form.**

Some problems contain an optional extra challenge question. This is not graded, but may be interesting if you want to explore a problem in greater depth.

If you have any questions, please write to mimno@cornell.edu. Any corrections or clarifications resulting from such questions will be posted in a pinned note on Piazza. Start early. Upload to CMS frequently. If for some reason CMS is not accepting your submission, mail it as an attachment. Make sure that you have submitted the correct file. Timestamps from your computer are not admissible.

1. Debug code

The file `broken.html` contains a script that should display a scatterplot of red dots, but is not working. There are 19 errors in this script. Edit this file to fix the errors. Include comments to describe your changes. Note that in some cases the same error occurs on two lines of code (that is, the same error is made twice, in consecutive lines). These count as two errors. All edits will be in the code, the data file `broken.txt` has no errors (but it may be worth looking at). If you're stuck, try adding console.log() statements, and use the Element inspector tool in your browser to see what you're actually creating in the page. (19 pts)

2. Manufacturing jobs, time-series plots

We've heard a lot about manufacturing jobs during the current US election campaign, mostly with regard to international competition. But what about domestic competition? In this problem we'll look at data from the US Bureau of Labor Statistics for all manufacturing employment *by state* over the period from 1990 to 2016. You will modify the file `manufacturing.html`. Data (heavily processed by me) is in the file `manufacturing.txt`.

a. Add code to turn the raw input from the `d3.tsv()` call into usable data. You will need to convert the Jobs field into a number and the Date field into a javascript Date object. Next use `d3.nest()` to construct an array that has one element for each state, an object which contains the name of the state and an array containing the Date/Jobs objects (Hint: see the documentation for d3's nest function, which will do all of this for you). (4 pts)

b. Fill in the body of the `plotRawData()` function. This should create an `<svg>` element in the `#rawPlot` div. You will need to define appropriate scales and axes, and add one `<path>` element for every state, showing the number of manufacturing jobs in that state during each month over 26 years. Use appropriate css styles (we will not clarify that, it's up to you) to highlight four states: NY, TN, NV, and OH. Place a small `<text>` element at the end of each path to show which state the line represents. (2 pts)

c. Now create another view of the same data, showing *relative* gains and losses. Fill in the body of the `plotNormalizedData()` function. This should create an `<svg>` element in the `#normalizedPlot` div. You will need to define appropriate scales and axes, and add one `<path>` element for every state, showing the number of manufacturing jobs in that state during each month over 26 years **divided by** the value for Jan 1990. As before, highlight NY, TN, NV, and OH and label each state's line in the right margin. (2 pts)

d. Answer the following questions in the provided spaces. (4 pts)

- How has NY manufacturing done relative to OH? Was there a difference between how these states fared in the 1990s and the 2000s?
- What question can you answer from the raw plot that would be difficult, or impossible, to answer from the normalized plot?
- Going the other way, what question can you answer from the normalized plot that would be difficult, or impossible, to answer from the raw plot?
- Many state boundaries are based on long-forgotten disputes between English settlers in the 1600s. Does it make sense to report statistics for RI and CA? Assume you know the location of every individual job. What other way could the BLS report aggregate statistics about employment, and how would your method affect the appearance of the plots?

3. Driving fatalities.

The number of traffic deaths in the US for 2015 was released recently, and it's too shocking not to share with you. This problem is similar to the previous one, but with more of a focus on data collection and curation. You will modify the file `fatalities.html`.

a. Compile a full dataset of traffic fatality counts. Combine data from wikipedia with the new 2015 value from the National Safety Council (see the HTML file for links). You may construct this as a separate file or as a JSON element in the HTML file. (3 pts)

b. Plot the time series using a `<path>` element in the `#path` div. Use appropriate scales and axes. (3 pts)

c. In the provided field: What factor do you think might have caused the observed increase in fatalities? What additional information would you collect in order to test that hypothesis? (You do **not** actually have to collect any data for this problem, so be creative!) (3 pts)

4. Poisson distributions, Histograms, Tom Brady

Recently there was a controversy in American Football. The New England Patriots were accused of deliberately deflating footballs to make them easier to grip. Some sports statisticians noticed that the Patriots fumble (drop the ball) less often than other teams. Can we test this assertion?

The Poisson distribution is a distribution over discrete, non-negative random variables, like the number of times a team fumbles the ball in a game. We often use the Poisson as a "boring" distribution: it arises when you count rare events that occur at random times during a specified interval. If count data looks like a Poisson, there's no reason to think that the data isn't basically random. If the data does NOT look like a Poisson, there is likely to be some factor that you need to explain.

In this problem you will plot a histogram of the number of times each NFL (i.e. American football) team fumbles per game. You will compare those histograms to two Poisson distributions: one estimated from the league as a whole and another from the team by itself. The data files representing 14 years of data (from www.repole.org) contain two rows for each game, one for each team's statistics. You will edit the file `nflpoisson.html`. Recall that a Poisson distribution is parameterized by a single number, the "rate," which is equal to the mean and variance of the distribution. You are allowed to use the d3 `queue` function for this problem, but it is not required.

a. The file contains an array listing the names of CSV files. Get the contents of each CSV file in the yearFilenames array. Concatenate the rows from each year into a single large array in the variable footballData. When each one has completed, the array footballData should

contain one element for each row in the CSV files. Once the code has finished loading all 14 files, run init(). (5 pts)

b. Calculate the mean of the `FumblesOff` attribute over all games. array (HINT: use `d3.mean()`). We will use this mean as the parameter for a Poisson distribution. (4 pts)

c. Divide the array `footballData` into an array containing one element for each team, each of which contains only the rows for that team. You may wish to use `d3.nest()` to group rows based on the `TeamName` attribute. (5 pts)

d. The code contains a loop over each team. Set the value of the variable `sequence` to an array that contains the number of offensive fumbles in each game for this team. Calculate the mean of those numbers, and save it in the variable `teamMean`. (4 pts)

e. Compare the histogram of games with *n* fumbles (the max is five) for each team to an expected number of games with *n* fumbles based on a Poisson distribution with rate equal to the mean number of fumbles from **all teams in all games**. Most of the code is the same as you've seen before, and is provided. Fill in the argument to the `data()` function. (4 pts)

f. Now add green horizontal lines to show the expected number of games with *n* fumbles based on a Poisson distribution with rate equal to the mean number of fumbles from **only this team**. (4 pts)

g. Do you think the Poisson distribution based on the entire league (the black horizontal lines) is a good fit for the number of fumbles per game? Why or why not? Is the Poisson distribution based on each team a good fit? Why or why not? Are the New England Patriots unusual? (Use the free-response at the top of the page.) (4 pts)

OPTIONAL NON-GRADED EXTRA CHALLENGE. Did these statistics change over time? Add a feature to allow a user to select specific year ranges, and comment on any patterns that you observe. (**0 pts**)

5. Linear Regression, the t-test

Simple linear regression takes, as input, an array of pairs of variables. Each pair consists of an input *x* and an output *y*. The model says that there is a linear relationship between the input and output: each time we increase the input by 1.0, we expect the output to change by `slope`. What how can we be sure there really is a relationship? The linear regression function will always give us a slope, but can we trust it?

We talked about p-values in class. You may have seen them before. When you run a linear regression in R or SAS, for each parameter (ie slope, intercept) you get a number that is between 0 and 1. Values less than 0.05 are supposed to be good. But what do these

numbers really mean?

One way to evaluate the significance of a slope parameter is to use a *t*-test. We calculate a function of the absolute value of the slope, and pass that value to a piece of code that calculates the tail probability of a Student *t* distribution. The result is your p-value. I've included code used to calculate that p-value.

P-values often seem mysterious, and the way they are calculated is, as you can see from the `pValue()` code, incredibly confusing. You will compare the p-value you get from this code to the value we get using a *permutation test* like we did in class. You will edit code in the file `permutation.html`. When you open it and hit the "Run" button, the script will generate 10 points from a linear model, plot the points, and plot the linear regression line for those points. It will also show you a p-value for the slope.

Remember that the linear model is testing whether there is a systematic relationship between the input and output variable. A permutation test tells us what would happen if there were *no* relationship between these variables by actually randomizing this association. We then compare the model we actually got from the real data to the possible models we could have gotten from randomly shuffled variations of the same data.

a. Create a function that randomly shuffles the values of the *y* variable. You can start with the provided stub of a function called `permute()`, you will fill in the body of this function. In this function create a new array of *x,y* pairs. The *x* values should appear in the same order as the *x* values in the input array `points`, but the *y* values should be randomly shuffled. In other words, each *y* value in the input array should appear the same number of times in the output array, but it may or may not be paired with its "correct" *x* value. (Class notes may be of interest.) (5 pts)

b. Inside the `run()` function, create 200 random permutations of the real data. For each one, compute a linear model from that permuted data. Call `drawLine()` with a different color or opacity value from the real-data line. Check whether the absolute value of the permuted-data slope is larger than the absolute value of the real-data slope. Set `steeperSlopes` equal to the total number of permutations for which this condition is true. (5 pts)

c. You can now count (half) the proportion of permutations that had a more extreme (ie steeper) slope than the "real" slope. Hit "Run" a few times. How does this permutation test value compare to the *t*-test p-value? (Use the free-response `<div>` at the top of the page.) (5 pts)

d. Create a second SVG element with an x-axis for p-values and a y-axis for permutation test values. Both scales should go from zero to one. Add a diagonal line from (0,0) to (0.5, 0.5). (5 pts)

e. The code prints two values each time you hit "Run", a p-value and a permutation test value. Plot a circle on the second SVG element whose *x* position is the p-value and whose *y* position is the permutation test value. (5 pts)

f. How do these two tests compare? What happens when you change the parameters passed to `generateLinear`? Turn in your work using the default settings, but experiment with using more points, a steeper slope, and greater noise. Describe how the p-value and permutation results change. (Use the free-response at the top of the page.) (5 pts)

OPTIONAL NON-GRADED EXTRA CHALLENGE. Why do we need to divide the ratio of steeper slopes by half? There are 10! possible permutations of *y*-values for 10 data points. Can we really just sample 200 of them? Can you make it possible to set parameters interactively? (**0 pts**)