Homework 1 Solutions

BEE 4850/5850

> Due Date
>
> Friday, 2/6/25, 9:00pm

Load Environment

The following code loads the environment and makes sure all needed packages are installed. This should be at the start of most Julia scripts.

```
import Pkg
Pkg.activate(@__DIR__)
Pkg.instantiate()
```

The following packages are included in the environment (to help you find other similar packages in other languages). The code below loads these packages for use in the subsequent notebook (the desired functionality for each package is commented next to the package).

```
using Random # random number generation and seed-setting
using DataFrames # tabular data structure
using CSVFiles # reads/writes .csv files
using Distributions # interface to work with probability distributions
using Plots # plotting library
using StatsBase # statistical quantities like mean, median, etc
using StatsPlots # some additional statistical plotting tools
using Dates # DateTime API
```

Problems

Problem 1

Problem 1.1

First, let's load the data.

```
# load the data
data = DataFrame(load("data/bites.csv")) # load data into DataFrame

# split data into vectors of bites for each group
beer = data[data.group .== "beer", :bites]
water = data[data.group .== "water", :bites]

observed_difference = mean(beer) - mean(water)
```

```
4.37777777777778
```

Now, we write a function (`simulate_differences()`) to generate a new data set and compute the group differences under the skeptic's hypothesis by shuffling the data across the two groups (this is called *the non-parametric bootstrap*, which we will talk about more later):

```
# simulate_differences: function which simulates a new group difference based
 ↪  on the skeptic's hypothesis of no "real" difference between groups by
 ↪  shuffling the input data across groups
# inputs:
#   y : vector of bite counts for the beer-drinking group
#   y : vector of bite counts for the water-drinking group
# output:
#   a simulated difference between shuffled group averages
function simulate_differences(y , y )
    # concatenate both vectors into a single vector
    y = vcat(y , y )

    # create new experimental groups consistent with skeptic's hypothesis
    y_shuffle = shuffle(y)   # shuffle the combined data
    n  = length(y )
    x  = y_shuffle[1:n ]
    x  = y_shuffle[(n +1):end]

    # compute difference between new group means
    diff = mean(x ) - mean(x )
```

```
    return diff
end
```

```
simulate_differences (generic function with 1 method)
```

Next, we evaluate this function 10,000 times and plot the resulting histogram of differences.

In Julia (and in Python), it is convenient to use a *comprehension* to automatically allocate the output of the `for` loop to a vector. The syntax for a comprehension is `[some_function(input) for input in some_range]`. In this case, the index `input` doesn't appear in the comprehension as we're just repeating the exact same calculation every time:

```
shuffled_differences = [simulate_differences(beer, water) for i in 1:50_000]
```

```
50000-element Vector{Float64}:
 -1.5466666666666669
 -0.5911111111111111
 -0.3044444444444423
 -0.7822222222222202
  0.9377777777777787
  2.944444444444443
 -1.9288888888888884
  0.7466666666666661
 -2.024444444444441
 -1.3555555555555578

 -0.3999999999999986
 -2.6933333333333316
  0.36444444444444457
 -0.4955555555555584
 -2.2155555555555573
 -0.3044444444444423
  0.46000000000000085
 -2.2155555555555573
  0.5555555555555571
```

Without a comprehension, this loop would look something like:

```
shuffled_diffs = zeros(10_000)
for i in 1:length(shuffled_diffs)
    shuffled_differences[i] = simulate_differences(beer, water)
end
```

Now let's plot the histogram.

```
hist = histogram(shuffled_differences, label="Simulated Differences") # plot
 ↪  the basic histogram
# MAKE SURE TO ADD AXIS LABELS!
xlabel!(hist, "Increased Number of Average Bites for Beer Group")
ylabel!(hist, "Count")

# now add a vertical line for the experimentally-observed difference
vline!(hist, [observed_difference], color=:red, linestyle=:dash,
 ↪  label="Observed Difference")                                          (1)
```

(1) `vline!()` (which creates a vertical line) is an example of a *mutating* function, which modifies an existing object *in-place* (meaning that it does not create a new object). In this case, looking at the plot `hist` before the `vline!()` call would not show the line, while looking at it afterwards will. This is in contrast with a *non-mutating* function, which would return a new object which would need to be saved in a variable. Julia convention is to name mutating functions with exclamation marks (*e.g.* `vline!`) to distinguish them from non-mutating functions; not all languages are this strict, so be aware of whether you're using a mutating or a non-mutating function. Also, specifically passing `hist` to `vline!()` is not strictly necessary; often mutating functions will assume that the object to be modified is the last object which has been referenced. But passing the object removes ambiguity and avoids accidentally overwriting the wrong one.

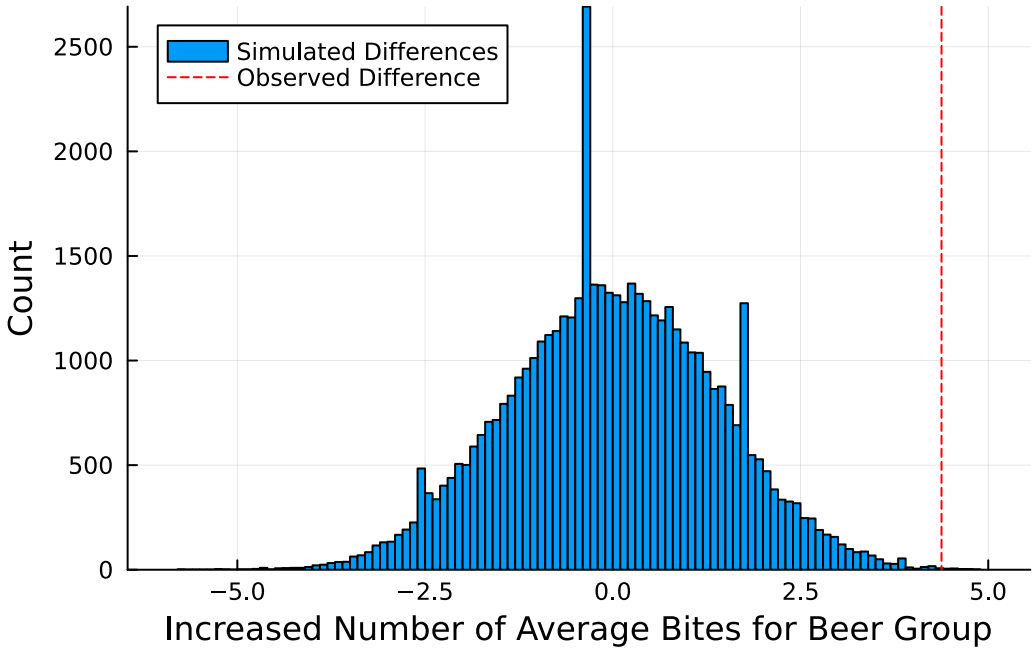

Figure 1: Histogram of simulated differences between the average bites for the beer group and the average bites for the water group under the assumption that differences are due only to random chance. A positive value indicates that the beer group is bitten more often. The red line is the group difference from the experimental data.

Problem 1.2

What do we see from Figure 1?

- The simulated differences follow roughly a normal distribution centered at a value of zero. This is not surprising given the hypothesis that there is no "true" difference in bite frequency between the two groups.
- The observed data are extremely unlikely if the skeptic's hypothesis is true. We can calculate the probability of seeing data that extreme given this hypothesis (also called the *p-value*) by finding the empirical cumulative density function of the simulated data vector:

```
empirical_cdf = ecdf(shuffled_differences)
1 - empirical_cdf(observed_difference)
```

  This shows that we would only expect, *given the skeptic's hypothesis*, to see data at least this extreme by chance in 0.04% of experiments. If we don't think that our experiment is likely to be an outlier, this suggests that the skeptic's hypothesis is quite unlikely.

However, this does not mean our mechanistic theory for the group difference is correct: this would require more work and maybe a more targeted experiment!

Problem 2

Problem 2.1

Let's load the dataset and do some basic EDA. First, the quantiles:

```
# load the data
chicago_dat = DataFrame(load("data/chicago.csv")) # load data into DataFrame

# get the summary quantiles
mapcols(col -> quantile(skipmissing(col), [0, 0.25, 0.5, 0.75, 1.0]),
 ↪  chicago_dat[:, 2:end])                                                    (1)
```

(1) There are two syntax features here that might be of interest to students using Julia. First, `mapcols` applies a function to every column of a `DataFrame` ("map" means a function is applied to subsets of a collection, such as an array). In this case, we define an anonymous function which takes each row, skips the missing values (`'skipmissings`), and computes the quantiles we're interested in (note that we skip the first column; this just the row indices). In other languages, there may be built in functions which do this more directly. Second, `end` is "syntactical sugar" for the final index of an array; it internally calls `nrow`, `ncol`, or `length`, depending on the application (meaning it is not "optimal" in terms of runtime, but the penalty is pretty minor), but is a bit simpler to read. You can also modify it to get indices that aren't from the end, *e.g.* `end-1` gives you the penultimate index.

	death	pm10median	pm25median	o3median	so2median	time	tmpd
	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	69.0	-37.3761	-16.4264	-24.7794	-8.2061	-2556.5	-16.0
2	105.0	-13.1082	-6.58841	-10.2319	-2.68935	-1278.25	35.0
3	114.0	-3.53906	-1.32584	-3.32586	-1.21826	0.0	51.0
4	124.0	8.30292	5.34377	4.46821	0.831593	1278.25	67.0
5	411.0	320.725	38.1504	43.6878	28.9034	2556.5	92.0

To compute the mean, we can take a similar `mapcols` approach:

```
# compute the mean of each column
mapcols(col -> mean(skipmissing(col)), chicago_dat[:, 2:end])
```

	death	pm10median	pm25median	o3median	so2median	time	tmpd
	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	115.419	-0.14639	0.243053	-2.17938	-0.636071	0.0	50.1933

Finally, we want to count the number of missing values. In Julia, we can do this by summing the `ismissing()` function, which returns a true value if the value is missing and is false otherwise. There are similar functions in other languages; but this is a reason to avoid imputing values such as 0 where data are missing.

```
# count the number of missing values
mapcols(col -> sum(ismissing.(col)), chicago_dat[:, 2:end])
```
①

① We need to broadcast `ismissing` as it is defined on individual values, not vectors (and the entire column is not missing, so it will return `false`). Just calling `sum(ismissing(col))` would give us a `0`. Make sure you look at the documentation of functions to ensure that they behave as you expect!

	death	pm10median	pm25median	o3median	so2median	time	tmpd
	Int64	Int64	Int64	Int64	Int64	Int64	Int64
1	0	251	4387	0	27	0	0

The temperature units are in degrees Fahrenheit, not Celsius, which we can see by the scale. That the pollution variables are below zero approximately half the time (based on the median) suggests that they are anomalies, in this case from the mean.

Problem 2.2

We first want to convert `time` to the calendar date. To do that, we need to identify the zero date and recognize that the dates are also shifted by a value of `0.5`. We can see from the first and last values of the `time` column that the values are symmetric, which means they are likely days from a central value. If we eliminate the decimal by adding `0.5`, the first value (which should correspond to Jan. 1, 1987) is -2556. We can find the zero date by finding the date which is 2556 days from 01-01-1987. This will look different in different languages, but in Julia (using the features and syntax from `Dates.jl`:

```
Date(1987-01-01) + Day(2556)
```
①

① This uses the `DateTime` formatting from `Dates.jl`. The nice thing about this syntax is it lets us treat units as time arithmetically.

```
1992-01-01
```

So, since we added 0.5 to each row, the zero date is Dec. 31, 1993 (it would be Jan. 1, 1994 if we subtracted). As a result, we can convert the `time` column to a date:

```
day_zero = Date("1993-12-31")
chicago_dat.Date = day_zero .+ Day.(chicago_dat.time .+ 0.5)                    (1)
```

(1) `df.colname` lets us reference and extract only the named column.

```
5114-element Vector{Date}:
 1987-01-01
 1987-01-02
 1987-01-03
 1987-01-04
 1987-01-05
 1987-01-06
 1987-01-07
 1987-01-08
 1987-01-09
 1987-01-10

 2000-12-23
 2000-12-24
 2000-12-25
 2000-12-26
 2000-12-27
 2000-12-28
 2000-12-29
 2000-12-30
 2000-12-31
```

Now, let's plot the deaths versus the date. We'll use a line plot to make the time trends easier to see.

```
# make a line plot of the death time series.
plot(chicago_dat.Date, chicago_dat.death, xlabel="Date",
 ↪  ylabel="Non-Accidental Deaths", legend=false)                               (1)
```

(1) I set the legend to false since we only have one data series in this plot. And make sure you include axis labels and units!

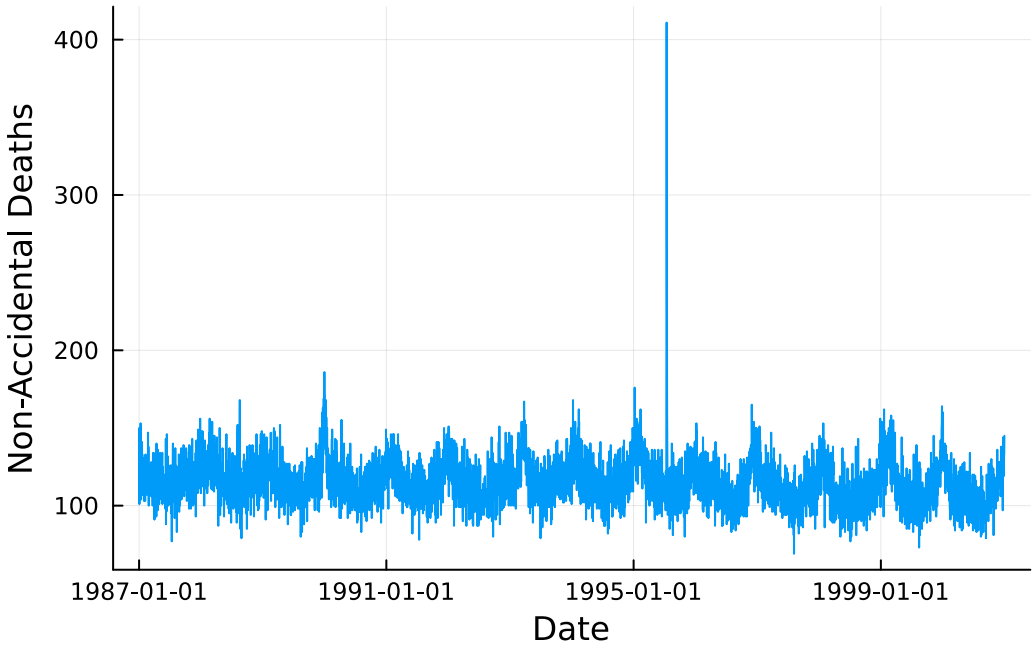

Figure 2: Number of non-accidental deaths in Chicago from 1987—2000.

Other than a large spike on a particular day in 1995, Figure 2 reveals that there is some seasonal variation in the number of deaths. We would need to do more analysis to understand these variations, but they visually appear regular, with no real trends over time.

Problem 2.3

Now let's plot the deaths versus the temperature. Here we might want to use a scatterplot since there is no clear ordering between the temperatures on the x-axis, as there was in Problem 2.2 with time.

```
# make a scatterplot of the deaths vs. temperatures.
scatter(chicago_dat.tmpd, chicago_dat.death, xlabel="Temperature (°F)",
 ↪  ylabel="Non-Accidental Deaths", markersize=2, alpha=0.8, legend=false)  ①
```

① We use a smaller markersize and lower alpha to make it easier to distinguish individual points which overlap. You may not need to use both.

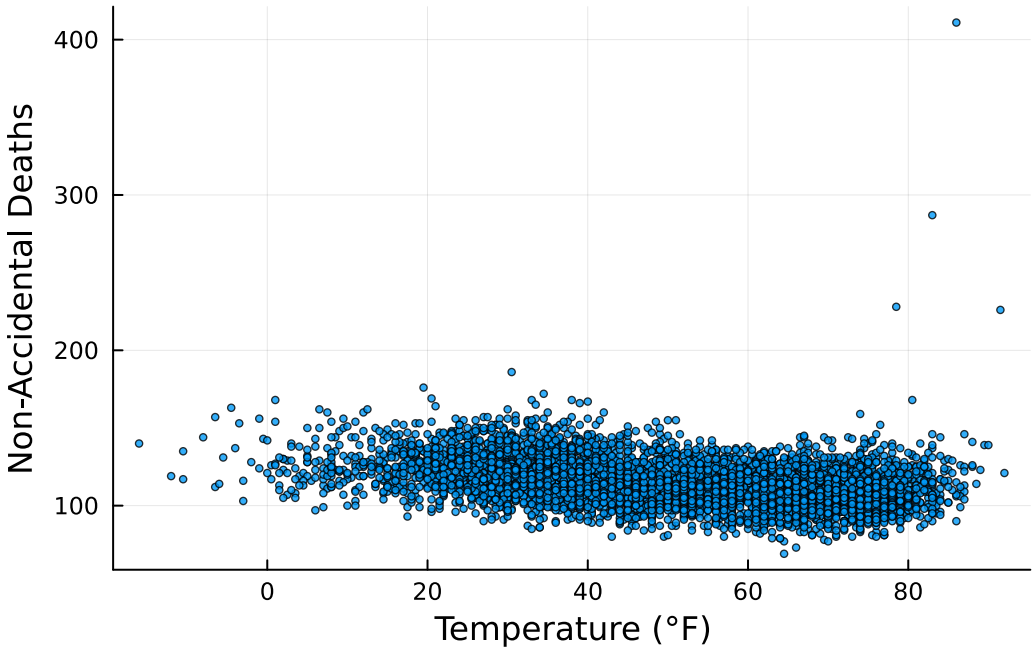

Figure 3: Number of non-accidental deaths in Chicago from 1987—2000 by daily average temperature.

Figure 3 shows that there are some large outliers at higher temperatures, but it looks like there could be a slight negative trend in the deaths as the temperatures increase. **Could** is the key word: there is a lot of variation so it's unclear if that is signal or noise.

Problem 2.4

Neither is particularly clear, but there does not appear visually to be a trend in time, whereas there does appear to be a slight trend in temperature.

Problem 3

Problem 3.1

This looks the same as in Problem 2, except we only care about summarizing the `Flowering.DDY` column. We'll save the values in a variable since we want to use them in our future plotting.

```
# load the data
cherry_dat = DataFrame(load("data/kyoto.csv")) # load data into DataFrame
```

```
# get the summary quantiles
flower_quantiles = quantile(skipmissing(cherry_dat[:, :"Flowering.DOY"]), [0,
↪  0.25, 0.5, 0.75, 1.0])                                                    ①
```

① The `df[:, :colname]` syntax can be used instead of `df[:, colindex]` (the `:` indicates that this is a `Symbol`, which is how Julia encodes things like `DataFrame` column names). When `colname` is a simple string, this syntax is straightforward; in this case, there is a decimal in the column name, which gets more complicated as `df.colname` is also a way to reference a specific column, as seen in the solution to Problem 2.1. Using quotes around the name to indicate that the entire thing is a `String` solves that problem.

```
5-element Vector{Float64}:
  86.0
 100.0
 105.0
 109.0
 124.0
```

Now, let's compute the mean. We'll save it in a variable since we want to use it in our future plotting.

```
# calculate the mean
flower_mean = mean(skipmissing(cherry_dat[:, :"Flowering.DOY"]))
```

```
104.54050785973398
```

Finally, the number of missing values:

```
# count the number of missing values
sum(ismissing.(cherry_dat[:, :"Flowering.DOY"]))
```

```
388
```

Problem 3.2

We'll make a similar time series plot as in Problem 2.2, with a line plot for the observations during the year, to help make trends stand out.

```
# plot the flowering series
p = plot(cherry_dat."Year.AD", cherry_dat."Flowering.DOY", xlabel="Year
↪  (AD)", ylabel="Flowering Day of Year", label="Observed Flowering Day")
# add horizontal lines for the mean
```

```
hline!(p, [flower_mean], color=:red, label="Flowering DOY Mean")                ①
# add horizontal lines for the 0.25 and 0.75 quantiles
hline!(p, flower_quantiles[[2,4]], color=:purple, label="Flowering DOY
 ↪  Quantiles (0.25/0.75)")
xticks!(p, 800:100:2100)                                                        ②
```

① `hline!()` (and the vertical equivalent, `vline!()`) wants a vector of values. For the mean, this means that we had to package the single, scalar value as a 1-value vector by wrapping it in square brackets. For the quantiles, we are passing a vector of two values, so we don't have to do this.

② `xticks!()` and `yticks!()` let us change the position and labels of axis tick marks from the default ones. Here, we don't need custom labels, but might benefit from having ticks every century to help our subsequent analysis. This is typical `start:step:stop` syntax for specifying a numerical range with set end points and constant steps.

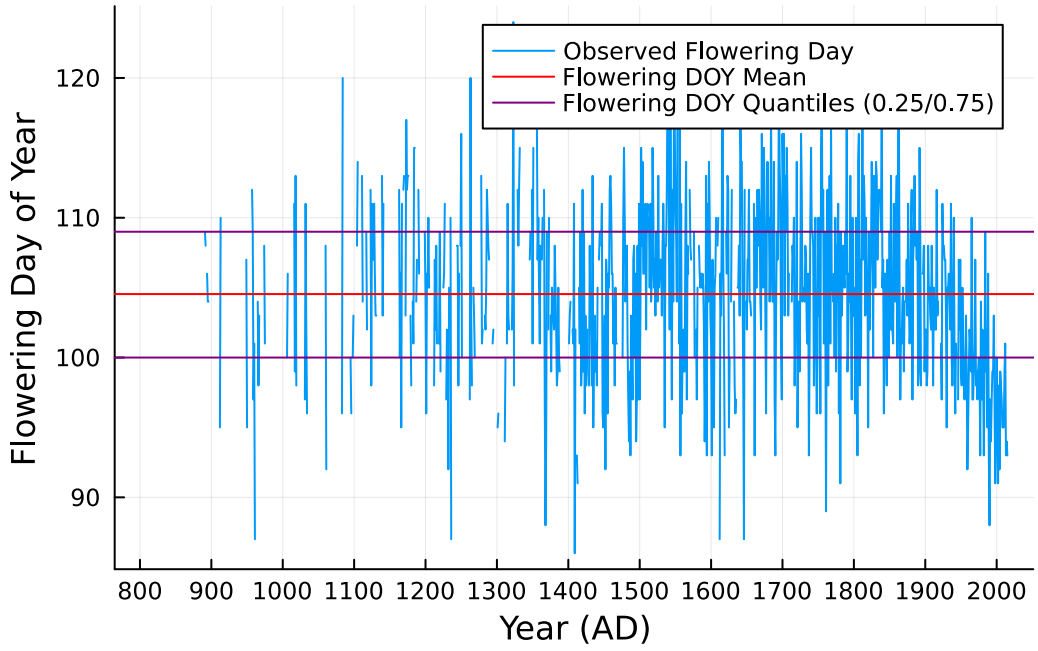

Figure 4: Day of years of cherry blossom flowering. The red line is the mean and the purple are the 0.25 and 0.75 quantile throughout the entire historical period.

It looks from Figure 4 as though, until the late 19th/early 20th century, there is no real trend in the data. However, the "typical" day of flowering appears to be getting earlier after 1900.

Problem 4

Problem 4.1

Under the hypothesis that there is no correlation between rigor and impact and that each marginal property follows a standard normal, we can sample 200 realizations of each and find the indices of the ones which have been selected. Since we want to do this with multiple replications, we'll create a function.

```
# simulate the described grant selection process.
function grant_selection(n)
    grants = rand(Normal(0, 1), (n, 2))                                    ①
    total_scores = sum(grants; dims=2)                                     ②
    select_threshold = sort(total_scores; dims=1, rev=true)[Int64(0.1 * n)] #
 ↪  get the threshold score for the top 10%
    select_idx = total_scores .>= select_threshold
    grants = hcat(grants, select_idx) # append selection status to grant
 ↪  matrix
    grants_df = DataFrame(grants, [:rigor, :impact, :selected])            ③
    return grants_df
end
```

① This syntax samples a 200x2 array of values from `Normal(0, 1)`. We could also have sampled two 200x1 arrays separately; this is just a bit more concise.

② The `dims` argument for `sum` is equivalent to *mapping* sum across a dimension of an array; there is similar syntax in NumPy and this could also be done with a loop or a comprehension.

③ This isn't at all necessary, but converting all of the sampled scores to a single DataFrame including selection status will let me look at both the overall correlations and create plots of the full sample compared to the selected subsample. There are many other solutions that will work!

```
grant_selection (generic function with 1 method)
```

Just to explore the impact of the selection process, let's look at the outcomes from a single realization.

```
grant_sim = grant_selection(200)
grant_selected = subset(grant_sim, :selected => x -> x .== 1)              ①
p_grants = scatter(grant_sim[!, :rigor], grant_sim[!, :impact], xlabel="Rigor
 ↪  Scores", ylabel="Impact Scores", label="All Grants")                   ②
scatter!(p_grants, grant_selected[!, :rigor], grant_selected[!, :impact],
 ↪  color=:red, label="Selected Grants")                                   ③
```

① `subset` selects a set of rows based on the condition in the second argument; in this case, evaluating the function `x -> x == 1` over each element of the `selected` column. Since `grant_sim[!, :selected]` takes 0 and 1 values, we could also have used `x -> Bool.(x)` to transform the values of x to `true` and `false`.
② This command creates the basic scatterplot using the `Plots.jl` syntax.
③ The `plot!` command adds new artifacts to an existing plot objective (in this case, `p_grants`). If you don't specify a plot object, it will automatically add to the last plot.

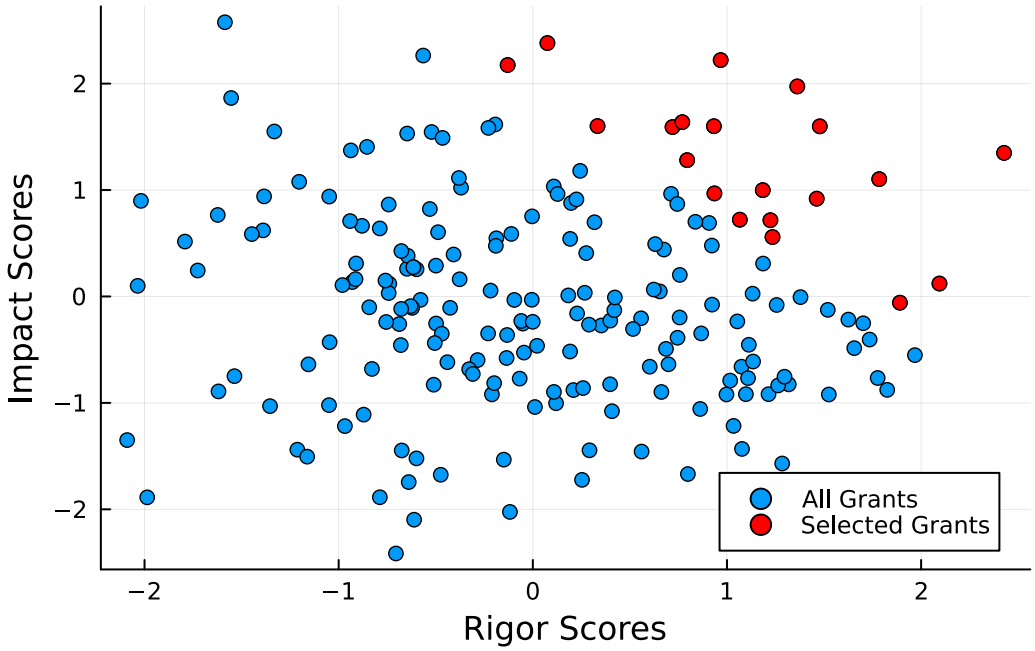

Figure 5: Outcomes of a single realization of the simulated grant selection process under the assumption of no correlation between rigor and impact. The grants selected for funding are colored in red.

We can see from Figure 5 that while the overall scores form an uncorrelated blob (correlation between grant and impact is -0.03), the selected grants have a strong negative correlation induced by the selection process (correlation -0.62).

That was just one sample, though: now let's look across 1,000 simulations and plot the distribution of the correlation coefficients. Again, we'll create a function to do the grant simulation and extract the correlation coefficients to simplify the replication process.

```
function grant_correlation(n_grants, n_replicates)
    grant_correlations = zeros(n_replicates, 2)                    ①
    for i = 1:n_replicates                                         ②
```

```
        grant_sim = grant_selection(n_grants)
        grant_selected = subset(grant_sim, :selected => x -> x .== 1)
        grant_correlations[i, 1] = cor(Matrix(grant_sim[!, 1:2]))[1, 2]      ③
        grant_correlations[i, 2] = cor(Matrix(grant_selected[!, 1:2]))[1, 2]
    end
    return grant_correlations
end

grant_cor = grant_correlation(200, 1_000)
p_grantcor = boxplot(["Full Pool" "Selected Grants"], grant_cor,
 ↪  legend=false, ylabel="Correlation Between Rigor and Impact")
```

① The `zeros` function initializes an array of zero values with the desired dimensions; in this case, each replication should have two values, one for the total pool correlation and one for the selected grant correlation. It's always a good idea to pre-allocate memory for arrays that you intend to fill sequentially when you can; constantly re-allocating memory can slow down runtimes dramatically.

② Julia's loops are quite fast, unlike some languages, so we won't shy away from using them!

③ `cor` returns a correlation matrix, and we're interested in the off-diagonal elements (the diagonal elements are just one).

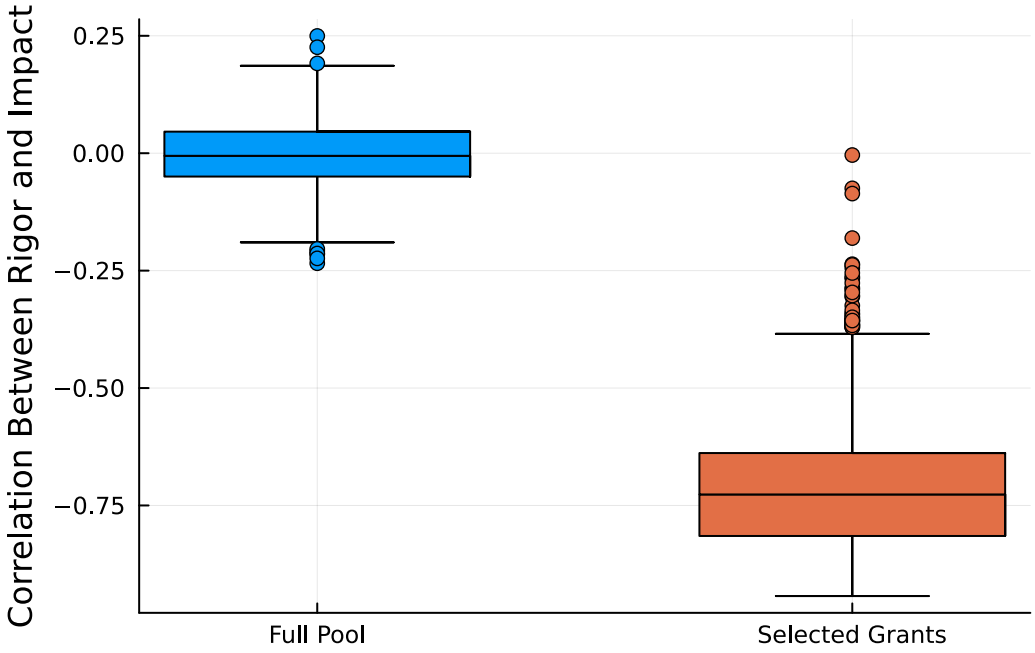

Figure 6: Distributions of correlations across the simulated grant pools and the selected grants.

We can see from Figure 6 that while the overall pool has its correlation around zero (as designed), as in our individual example, the selected grants typically have strong negative correlations between rigor and impact (though there are some simulations with a far weaker correlation).

Problem 4.2

Why does this correlation occur? The selection process filters for the total scores which are above a certain threshold. This means that a grant can get above this threshold by compensating for having a lower rigor score by having a higher impact score, or vice versa, which creates a negative correlation when one only looks at the selected grants. In other words, this is **purely an artifact of the selection process, not anything intrinsic to the grants themselves** (hence the selection-distortion effect). In fact, the selected grants likely can't be too weak at either category; the correlation only looks at the linear relationship between the two measures, not their values.

Problem 5

Problem 5.1

Let's start by sketching out the simulation model. For every student, we first need to simulate the outcome of the first coin flip, which has a 50% probability of heads. If this coin flip comes up as heads, then the student answers honestly, and admits to cheating with probability p. If the coin flip comes up tails, the student flips another coin and answers that they cheated with probability 50%. After looping over this procedure for each student in the class, we add up the "true" values.

We might code this model as follows.

```
# cheating_model: function which simulates the outcome of the interview
 ↪  procedure described in this problem and returns the number of confessions
 ↪  obtained.
# inputs:
#   p: base probability of cheating under a given hypothesis
#   n: vector of bite counts for the water-drinking group
# output:
#   a simulated number of confessions for one realization of the process.
function cheating_model(p, n)
    # initialize the storage vector for whether students admit to cheating
    # we do this with a boolean vector, which is a little faster, but storing
     ↪  integers is basically the same thing
    cheat = zeros(Bool, n)
    # loop over every student to simulate the interview process
```

```
    for i in 1:n
        # initial coin flip
        # rand() simulates a uniform random number between 0 and 1
        if rand() >= 0.5
            # if this came up heads, simulate whether the student cheated
             ↪  based on the cheating probability
            if rand() < p
                cheat[i] = true
            else
                cheat[i] = false
            end
        else
            # otherwise, simulate another coin flip
            if rand() >= 0.5
                cheat[i] = true
            else
                cheat[i] = false
            end
        end
    end
    # return the total number of cheating admissions
    return sum(cheat)
end
```

```
cheating_model (generic function with 1 method)
```

Now we simulate under our assumption of low cheating and the TA's assumption of widespread cheating and plot the results.

```
# conduct the simulations
no_cheat_data = [cheating_model(0.0, 100) for i in 1:50_000]
low_cheat_data = [cheating_model(0.05, 100) for i in 1:50_000]
high_cheat_data = [cheating_model(0.30, 100) for i in 1:50_000]

# plot the histograms with axis labels and a vertical line for the "real"
 ↪  outcome of the procedure
p_cheat = histogram(no_cheat_data, color=:orange, label="Cheating Rate 0%",
 ↪  alpha=0.4)
histogram!(p_cheat, low_cheat_data, color=:blue, label="Cheating Rate 5%",
 ↪  alpha=0.4)                                                              (1)
histogram!(p_cheat, high_cheat_data, color=:red, label="Cheating Rate 30%",
 ↪  alpha=0.4)
```

```
xlabel!("Number of Students who Confess to Cheating")                    ②
ylabel!("Count")
vline!([31], linestyle=:dash, color=:black, linewidth=3, label="Observed
 ↪  Outcome")
```

① We can use the mutating versions of plotting functions (`histogram!`, `plot!`, `scatter!`, etc) to put multiple plots on the same set of axes (for example, overlaying a probability density function over a histogram).

② Using `xlabel!()` and `ylabel!()` is an alternative to specifying axis labels within the original plotting function call. This may help make code more readable.

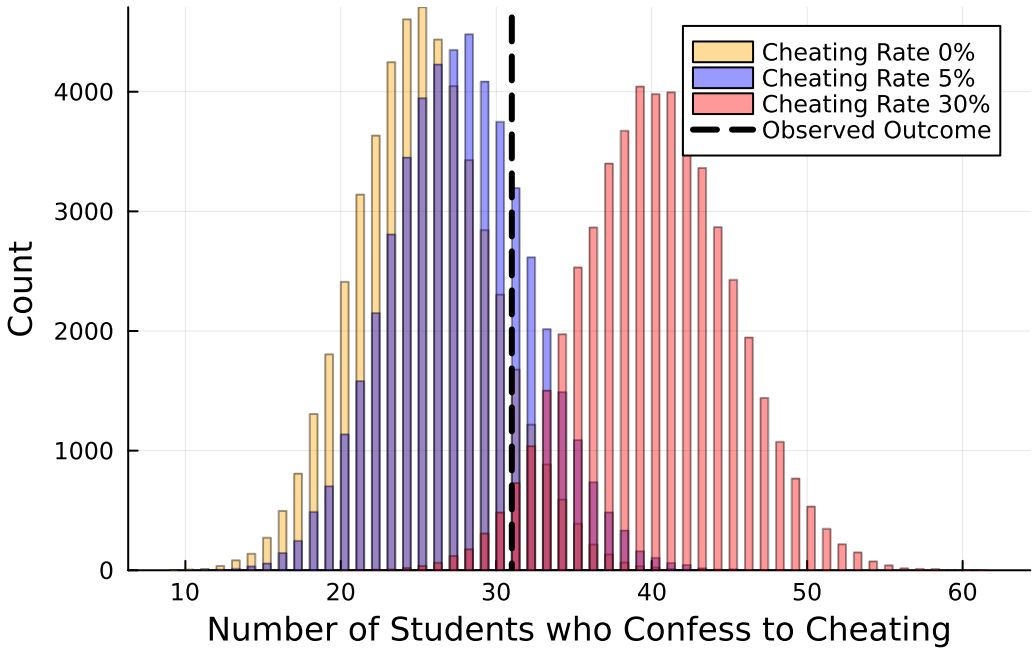

Figure 7: Histograms of the number of simulated confessions obtained under the hypothesis of a 5% cheating rate (blue) and a 30% cheating rate (red). The green line is the observed number of cheating confessions.

Problem 5.2

From Figure 7, we can see:

- There is some overlap between your hypothesis and the TA's around the 25-42 confession count rate. Lower than 25 students confessing would strongly suggest that the TA is overstating the rate of cheating, while more than 40 would strongly suggest that you are underestimating the cheating rate.

- Note that neither of these is "confirmation" of either theory, but evidence about the relative proportion of cheating being more or less consistent with one of our hypotheses.
- This interview process is noisy, but appears to work to separate very large differences in cheating rates. On the other hand, it might not work so well if we cared about the difference between 5% cheating and 10% cheating rates, as the difference in the "true" confessions would be swamped by the coin flips. If we wanted to tease out those differences, we could use a weighted coin (to increase the number of "honest" confessions).

So we might say that around 42 "Yes" answers would let us completely distinguish the two hypotheses. On the other hand, values between 25 and 42 would give us different levels of confidence about the two hypotheses; around 38 (just to make up a number) might provide evidence that the TA's hypothesis is stronger than ours. We'll talk later in the semester about how to try to make these determinations.

References