Lab 1: Getting Started with Julia

Due Date

Friday, 2/2/24, 9:00pm

::: {.content-visible when-format="html"}

Caution

If you are enrolled in the course, make sure that you use the GitHub Classroom link provided in Ed Discussion, or you may not be able to get help if you run into problems. Otherwise, you can find the Github repository here.

Introduction

Overview

The goal of this lab is to get you up and running with Julia. You'll start to be introduced to some of the basic syntax and workflow for data analysis in Julia, which we will build on over the course of the semester. We'll do some stats without making this apparent, and later in the semester, we'll formalize the type of analysis that you'll complete in this lab.

This lab repeats the analysis from Statistics Without The Agonizing Pain by John Rauser (which is a neat watch!).

Learning Objectives

After completing this lab, students wil be able to:

- write functions in Julia;
- simulate alternative datasets to test a hypothesis;
- make plots to compare summaries of observed datasets to alternatives.

Setup

Loading Packages

The first step in any Julia script or program is to load the environment, which contains any needed packages. To keep things efficient, "base" Julia contains relatively minimal functionality, and additional packages can be installed and loaded to add new functions and tools. In Julia, package management is handled through the `Pkg.jl` module. Julia stores information about packages in two files: `Project.toml` and `Manifest.toml`, which is why these are included in all of your assignment repositories. You won't need to touch these directly.

The following lines load `Pkg.jl` and activate the desired environment.

```
using Pkg # load Pkg.jl
Pkg.activate(@__DIR__) # load the environment based on the *.toml files in the same direct
Pkg.instantiate() # install any needed packages which are missing from the local install
Pkg.status() # print the packages available in the environment.
```

```
  Activating project at `~/work/simulation-data-analysis/simulation-data-analysis/labs/lab01
```

```
Status `~/work/simulation-data-analysis/simulation-data-analysis/labs/lab01/Project.toml`
  [7073ff75] IJulia v1.24.0
  [91a5bcdd] Plots v1.38.8
  [2913bbd2] StatsBase v0.33.21
  [9a3f8284] Random
```

I will start all of your assignments and scripts with at least the first three of these lines (`Pkg.status()` can be useful when you don't know what's available, but won't be necessary for our purposes going forward).

If you want to add additional packages for later use, you can do so with `Pkg.add()`. I've commented out these lines, as they are not needed, but feel free to uncomment and test: trying to `add` a package which is already in the environment won't do anything harmful.

```
# Pkg.add("Plots") # add Plots.jl, the base plotting library, to the environment
```

Now, we load the packages that we will need with the `using` keyword.

```
using Random # this loads functionality for random number generation
using StatsBase # this includes a bunch of statistical functions, including mean and quant
using Plots # this loads Plots.jl
```

And we're all set! Because we provided the environment files, the rest of this notebook should work smoothly (assuming you're using Julia 1.8.x), but if not, please ask or post about it!

> **Caution**
>
> Sometimes difficulties can emerge if the version of Julia is different. The first thing to try is to delete `Manifest.toml` and re-run `Pkg.instantiate()`, as this might resolve some issues related to package versions, though it runs the risk of a update changing some of the syntax. If you run into trouble, please bring it up on Ed Discussion, and we'll work through it.

Load Data

The underlying question we would like to address is: what is the influence of drinking beer on the likelihood of being bitten by mosquitoes?

First, we need to load the data. We will look at ways to work with structured data files later. For now, let's just enter the data manually. We will do this using *vectors*, which are data structures which correspond to one-dimensional lists. To define a vector, enclose the values between two square brackets, `[` and `]`[1] Line breaks don't matter: Julia is smart enough to recognize that the procedure isn't complete until it sees the closing bracket.

Let's load data for the number of bites reported by the participants who drank beer.

```
beer = [27, 20, 21, 26, 27, 31, 24, 21, 20, 19, 23, 24, 28, 19, 24, 29, 18, 20, 17, 31, 20
```

```
25-element Vector{Int64}:
 27
 20
 21
 26
 27
 31
 24
 21
 20
 19
 23
 24
```

[1] There is some nuance about whether you separate values with commas or spaces, which determines if the resulting vector is a row vector or a column vector. This isn't particularly relevant for this assignment, but can matter later.

```
28
19
24
29
18
20
17
31
20
25
28
21
27
```

We can see what type of variable `beer` is by using the `typeof` function:

```
typeof(beer)
```

```
Vector{Int64} (alias for Array{Int64, 1})
```

This tells us that `beer` is a vector consisting of integers (which means they don't include any decimals). Next, let's load the data for the number of times participants who were drinking water were bitten.

```
water = [21, 22, 15, 12, 21, 16, 19, 15, 22, 24, 19, 23, 13, 22, 20, 24, 18, 20];
```

Notice that, unlike when we created `beer`, we didn't get any output from creating `water`. This is because we ended the statement with a semi-colon `;`, which suppresses the output that would otherwise be displayed. This can be a convenient way to clean up the output of your code.

Another handy method for displaying variables is the `@show` macro. A *macro* is just a particular type of Julia function which starts with `@` and is "applied" to some other code:

```
@show water;
```

```
water = [21, 22, 15, 12, 21, 16, 19, 15, 22, 24, 19, 23, 13, 22, 20, 24, 18, 20]
```

The nice thing about using (**show?**) is that it explicitly tells us the name of the variable and what it equals. However, we do have to include the semi-colon, or else it will double-print the output.

Initial Analysis

How can we determine if there's a meaningful difference between these two sets of numbers? Naively, we might simply look at the different in means between the two datasets.

```
observed_differece = mean(beer) - mean(water)
@show observed_difference;
```

```
LoadError: UndefVarError: observed_difference not defined
```

This tells us that the participants in the experiment who drank beer were bitten approximately 4.3 more times than the participants who drank water! Does that seem like a meaningful difference, or could it be the result of random chance? In this problem, we will use a *simulation* approach to address this question.

Suppose someone is skeptical of the idea that drinking beer could result in a higher attraction to mosquitoes, and therefore more bites. To this skeptic, the two datasets are really just different samples from the same underlying population of people getting bitten by mosquitoes, rather than two different populations with different propensities for being bitten. If this is the case, then we can "shuffle" all of the measurements between the two datasets and re-compute the differences in the means, repeat this procedure a large number of times, and the difference between the two observed means should be captured by the distribution of possible differences.

Problems

Problem 1: Write a Simulation Function (10 points)

Write a function which:

1. takes in two vectors;
2. combines them (you can use the `vcat()` function to concatenate two column vectors);
3. shuffles the combined vector (using the `shuffle` function);
4. splits this shuffled vector into two vectors with the length of the original data;
5. returns the difference between the means of these two vectors.

We've given you a starting skeleton function below; you just need to fill in the code. This may require some Googling to figure out the exact syntax, and you can consult the course website's Julia Basics tutorial. Make sure to reference any consulted resources in the References section below!

Why Write a Function?

Writing a function for code that you are going to re-use multiple times makes your program more organized and readable. Plus, if you find a bug, you only have to fix it in one place! Julia's compiler also can better optimize code which is organized around functions.

```
# the function name says that it does: try to use meaningful variable and and function nam
# y1 and y2 are the input vectors
function simulate_difference(y1, y2)
# concatenate both vectors into a single vector

# shuffle this new vector

# split the shuffled vector into two vectors with the same lengths as y1 and y2

# compute the difference in means between these two new vectors; call this "difference"

# return the difference
return difference
end
```

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

Apply your function to `beer` and `water`.

```
simulate_differences(beer, water)
```

```
LoadError: UndefVarError: simulate_differences not defined
```

Problem 2: Plot Simulations (5 points)

Generate 10,000 simulations by applying your function to `beer` and `water` repeatedly. The most straightforward way to do this is with a *comprehension*, which will automatically evaluate the function over a vector if inputs and return a vector. The general format for a comprehension is `output = [function(var) for var in some_range]`.

For example, to compute the square of all of the integers between 0 and 10:

```
squares = [x^2 for x in 1:10]
```

```
10-element Vector{Int64}:
   1
   4
   9
  16
  25
  36
  49
  64
  81
 100
```

To reuse a function repeatedly on the same inputs, you can just use the `for var in some_range` as a counter, without `var` changing the inputs to the function.

You could also

```
# compute your differences here
```

Finally, plot a histogram of the differences in means along with a vertical line showing the observed difference. You can adjust the code below based on what your named your vector of differences above. If you have questions about any of the arguments to `histogram()` or 'vline!(), please experiment, ask, and/or look at the website's Making Plots with Julia tutorial.

! Mutating Functions

Functions which change their inputs in-place (without needing the output to be saved to another variable) are called *mutating*. In Julia, the convention is to end functions which mutate their inputs with an exclamation point `!`. For example, `vline(...)` will create a new plot with just a vertical line, but `vline!(p, ...)` will add a vertical line to the existing plot `p`. There are many mutating functions, particularly for plotting, that you will encounter, but it's good practice in general to avoid mutation for your own functions.

i Keyword Arguments

Julia separates arguments which are required and those which are optional (and therefore require keywords to tell the function that they're not just the defaults) with a semi-colon, as you can see in the code below.

```
p = histogram(simulated_differences; # this is the vector for the histogram
xlabel="Differences in Means", # set the x-axis label
ylabel="Proportion of Samples", # set the y-axis label
```

```
label="Simulations Assuming Skeptical Hypothesis", # set legend label for the data in this
legend=:topleft, # legend position; you can turn the legend off with legend=false
normalize=true, # plot proportion of samples instead of raw counts
color=:blue # set the color of histogram bars
)
vline!(p, # plot on the previous plot p
[observed_difference]; # vline wants the value(s) to be in a vector,
label="Observed Difference", # add line label to legend
linewidth=2, # set the width of this line
color=:red # set the color of the line
)
```

```
LoadError: UndefVarError: simulated_differences not defined
```

Problem 3: Interpret Results (5 Points)

Congratulations, you just tested a hypothesis based on analyzing simulations! What do you think about the plausibility of the skeptic's hypothesis that there is no difference? Feel free to use any quantitative or qualitative assessments of your simulations and the observed difference.

References Consulted