Assignment 1 - Q4: Simulating Gaussian Distributions

Welcome to Pluto Notebooks! In this course you will be spending quite a bit of time working within these notebooks. These notebooks are an HTML/CSS/Javascript built interface for interacting with and working with Julia. They were inspiried by Jupyter, and you can learn more about the notebooks through their github. The notebooks are lightweight (with files that can be used by the normal julia interpretor), and track how notebook cells depend on each other to re-run cells when their dependencies change.

The only code in this notebook which is not found in julia's base is the plotting code provided by StatsPlots and Plots. Another package we use is PlutoUI which contains utilities for building interfaces in pluto notebooks. To complete this assignment you will not need to import any other packages.

Selection deleted

For all the cells, some are hidden by default. To see hidden cells click on the eye to the top-left of the cell of interest.

In this assignment you will:

1. Learn about using Pluto Notebooks and Stats Plots for visualizing and analysing data.
2. Get experience with calculating the mean, variance, and other metrics of sample data.
3. Build intuition about how different Gaussian parameters impact our estimators.

```julia
# Import the packages and export their exported functions to the main namespace.
using StatsPlots, PlutoUI, Random
```

!!!IMPORTANT!!!

Insert your details below. You should see a green checkmark.

```
student =
  (name = "Mohammad Shahriar Hossain", email = "mhossai6@ualberta.ca", ccid = "mhossai6",
```

```julia
student = (name="Mohammad Shahriar Hossain", email="mhossai6@ualberta.ca",
ccid="mhossai6", idnumber=1724709)
```

Welcome Mohammad Shahriar Hossain! ✅

Gaussian Distribution

A Gaussian distribution has mean μ and standard deviation σ. We will want to sample from Gaussian distribution. We provide an implementation below. We also discuss that implementation, to help you better understand Julia syntax that will be useful for your own implementation.

`GaussianDistribution`

GaussianDistribution(μ::Float64, σ::Float64)

A Gaussian distribution with mean μ, standard deviation σ. You can sample data from this distribution using `sample(gd, n)` to get `n` samples. You can get the mean using `mean(gd)`, the standard deviation using `stddev(gd)`, and the variance using `var(gd)`.

```julia
# The block of text below add documentation to julia struct or function.
# Check out the live docs to the right when your cursor
# is in GaussianDistribution.
"""
    GaussianDistribution(μ::Float64, σ::Float64)

    A Gaussian distribution with mean μ, standard deviation σ. You can sample data
    from this distribution using `sample(gd, n)` to get `n` samples. You can get
    the mean using `mean(gd)`, the standard deviation using `stddev(gd)`, and the
    variance using `var(gd)`.
    """
struct GaussianDistribution
    μ::Float64 # mean
    σ::Float64 # standard deviation
end
```

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

```julia
mean(gd::GaussianDistribution) = gd.μ
```

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

```julia
stddev(gd::GaussianDistribution) = gd.σ
```

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

```julia
var(gd::GaussianDistribution) = gd.σ^2
```

```
sample (generic function with 2 methods)
```

```julia
function sample(gd::GaussianDistribution, n = 1)
    gd.σ*randn(n) .+ gd.μ
end
```

Understanding the Julia code for the Gaussian distribution

Note that in Julia we use `+` for scalar addition and `.+` to add two vectors. If we have two vectors `u` and `v`, both `d > 1` dimensional, then we would write `u .+ v` to add these elementwise. If we have a scalar `s`, then `u .+ s` adds `s` to every element of `u`.

Let us look more carefully at the `sample` function. First note that to generate a Gaussian sample with mean μ and variance σ^2, we 1) call `randn(1)` to generate a sample from a zero-mean, unit variance Gaussian (a normal distribution) and 2) scale it by σ and shift it by the mean μ. We can either call this function function `n` times, to get `n` samples. Or, we can leverage the fact that `randn(n)` returns `n` samples from a normal distribution. `randn(n)` is a vector of size `n` of samples from a normal distribution. Multiplying this vector by the scalar `gd.σ` rescales every element in the vector and then we `.+` the scalar `gd.μ` to shift every element in the vector.

Equivalently, we could have used a for loop and written

```
dataset = zeros(n)
for i in 1:n
   dataset[i] = gd.σ*randn(1)[1] + gd.μ
end
return dataset
```

where randn(1) returns a vector of dimension 1, so we have to further index this vector to return this single scalar. You may wonder why in our vector-based implementation, we did not explicitly have a `return`. In Julia the last value computed in the function is returned when there is no explicit `return`.

Note that Julia is 1-indexed, rather than 0-indexed. This means indexing an array or vector starts at 1, rather than 0. This contrasts Python and C, where indexing starts at 0, and matches Matlab, where indexing starts at 1. It's possible 1-indexing was chosen for Julia to help make it a suitable replacement for Matlab, which is (or was) a popular numerical computing language.

One other note. We have `n=1` as an argument to `sample`. This means that `n` defaults to 1 if it is not provided.

Implementing basic statistics from data

Below you will be implementing the sample mean, variance, and standard deviation of a dataset D. This dataset is guaranteed to be a vector of floating point numbers, where the ith entry corresponds to X_i. You need to fill in the code between `#### BEGIN SOLUTION` and `#### END SOLUTION`.

The sample mean is

$$\text{sample-mean}(D) = \frac{1}{n}\sum_{i=1}^{n} X_i$$

To implement the sample variance, we want you to use the unbiased sample variance formula

$$\text{sample-variance}(D) = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \text{sample-mean}(D))^2$$

Finally, the sample standard deviation is the square root of the sample variance.

A few useful functions for this section include the following, where `v` is a vector and `s` is a scalar. `sum(v)` takes the sum of the elements in `v` and `length(v)` returns the length of the vector `v`. `sqrt(s)` returns the square root of the scalar `s` and `s^2` squares the scalar `s`. You can also call sqrt and squaring on a vector, by calling `sqrt(v)` and `v.^2`. As mentioned above, most basic operations on scalars like `+, -, ^2` can be turned into elementwise operations on a vector by adding a ., namely using `.+, .-, .^2`.

Finally, you might want to use a for loop, which was explained above when discussing the `sample` function. Note that you can get away with simply using the above vector operations, but it can be mentally simpler and just as correct to use a for loop, so it is up to you.

Great job! ✅ 🎉

```
mean (generic function with 2 methods)
```

```julia
function mean(D::Vector{Float64})
    #### BEGIN SOLUTION
    n = length(D)
    sum = 0
    for i in D
        sum = sum + i
    end
    return sum / n

    #### END SOLUTION
end
```

```
var (generic function with 2 methods)
```

```julia
function var(D::Vector{Float64})
    #### BEGIN SOLUTION
    n = length(D)
    sum = 0
    for i in D
        sum = sum + ((i - mean(D))^2)
    end
    return sum/(n-1)
    #### END SOLUTION
end
```

```
stddev (generic function with 2 methods)
```

```julia
function stddev(D::Vector{Float64})
    #### BEGIN SOLUTION
    return sqrt(var(D))
    #### END SOLUTION
end
```

Simulating sample variance

Use the below let blocks to complete question 4(bcde). You will be graded on your written work, not on the code in these cells. Here we have given you one example of how you might call the above functions, to avoid getting hung up on Julia syntax.

```julia
let # example
    n = 3
    gd = GaussianDistribution(6.2, 0.1)
    dataset = sample(gd, n)
    for i in 1:n
        println(dataset[i])
    end
end
```

```
6.158013725046534
6.049248364935387
6.257839151998914
```

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

```julia
# Generates samples, useful for 4b, 4c, 4d, 4e
function genSample(n::Int64, μ::Float64, σ::Float64)
    gd = GaussianDistribution(μ, σ)
    dataset = sample(gd, n)
end
```

```julia
let # 4b
    n = 10
    μ = 0.0
    σ² = 1.0
    σ = 1.0
    M = Float64[]
    for i in 1:5
        D = genSample(n, μ, σ)
        push!(M, mean(D))
    end
    print("Array of Sample average: ")
    println(M)
    print("\n")
    print("Sample Variance: ",var(M))
end
```

```
Array of Sample average: [0.49211934859673956, 0.09869666421366792, 0.1222
2665509696688, 0.03207402982251641, -0.050702985899367926]

Sample Variance: 0.043504823884724
```

```julia
let # 4c
    n = 100
    μ = 0.0
    σ² = 1.0
    σ = 1.0
    M = Float64[]
    for i in 1:5
        D = genSample(n, μ, σ)
        push!(M, mean(D))
    end
    print("Array of Sample average: ")
    println(M)
    print("\n")
    print("Sample Variance: ",var(M))
end
```

```
Array of Sample average: [-0.0062631596615402765, -0.16671664049267332, -
0.019557383494495456, -0.0883543440078518, 0.08989795314506584]

Sample Variance: 0.009202144904763666
```

```
let # 4d, 4e
    n = 30
    μ = 0.0
    σ² = 10.0
    σ = sqrt(10.0)
    D = genSample(n, μ, σ)
    M = mean(D)
    V = var(D)
    print("Sample Average: ", M, "\n\n")
    #4d starts
    # 95% confidence interval: (mean(D)- 1.96*(var(D)/n), mean(D)+ 1.96*(var(D)/n))
    print("95% Confidence Interval assuming the samples are Gaussian: ")
    LBound = M - (1.96 * sqrt(V/n))
    UBound = M + (1.96 * sqrt(V/n))
    print("(",LBound,", ",UBound,")\n")
    #4d ends
    print("\n")
    #4e starts
    # 95% confidence interval: (mean(D)- sqrt(var(D)/(delta*n)), mean(D)+
    sqrt(var(D)/(delta*n)))
    print("95% Confidence Interval without assuming the samples are Gaussian: ")
    δ = 0.05
    ε = sqrt(V/(δ*n))
    LBound = M - ε
    UBound = M + ε
    print("(",LBound,", ",UBound,")")
    #4e ends
end
```

```
Sample Average: 0.6380038170109376

95% Confidence Interval assuming the samples are Gaussian: (-0.30176156093700
17, 1.5777691949588768)

95% Confidence Interval without assuming the samples are Gaussian: (-1.506260
7421649545, 2.7822683761868294)
```

Plotting

In this section we help you plot samples from the Gaussian, to get a better intuition for the impact of the underlying mean, variance and the number of samples. There are no explicit questions related to this section, but it might help you better understand your answers to question 4.

Below you will see some example plotting code `plot_density` and `plot_box_and_violin`.

`plot_density` plots a histogram of the data `D` and a density estimated through a kernel density algorithm (see implementation on github for more details).

`plot_box_and_violin` plots a box plot over a violin plot. A violin plot shows the density of the sampled data (same as the `density` function), while the overlayed box plot shows the first quartile, median, and third quartile. More information can be found on github about these plotting utilites.

plot_density (generic function with 1 method)

```
function plot_density(D)
    histogram(
        # data/transform paramters
        D, norm=true,
        # make plot pretty parameters
        grid=false, # removes background grid
        tickdir=:out, # changes tick direction to be out
        lw=1, # makes line width thicker
        color=RGB(87/255, 123/255, 181/255), # Changes fill color of histogram
        legend=nothing, # removes legend
        fillalpha=0.6) # makes the histogram transparent

    density!(D, color=:black, lw=2)
end
```

plot_box_and_violin (generic function with 1 method)

```
function plot_box_and_violin(D)
    plt = violin(
        ["data"], #The label for the data on the x-axis
        D, # the data
        grid=false, # remove the background grid
        tickdir=:out, # set the ticks to be out
        lw=0, # set the line width to be zero
        color=RGB(87/255, 123/255, 181/255), # set color
        legend=nothing)
    boxplot!(
        plt, # explicitly pass in plt object
        ["data"], #The label for the data on the x-axis
        D, # the data
        fillalpha=0.5, # make transparent
        lw=3) # emphasize the lines

end
```

Visualizing the distribution from samples

Below are some sliders you can use to visualize different normal distribtuions interactively. The data is then plotted using the above plotting functions.

Mean: 0.0

Std Dev: 1.0

number of samples: 100

Resample

```
# This is a markdown block.
md"""
## Visualizing the distribution from samples

Below are some sliders you can use to visualize different normal distribtuions interactively. The data is then plotted using the above plotting functions.

Mean: $(@bind mu Slider(-100.0 : 0.1 : 100.0; default=0.0, show_value=true))

Std Dev: $(@bind sigma Slider(0.1 : 0.1: 100.0; default=1.0, show_value=true))

number of samples: $(@bind n Slider(10 : 10 : 10000; default=100, show_value=true))

$(@bind resample_btn Button("Resample"))
"""
```

gd = GaussianDistribution(0.0, 1.0)

```
gd = GaussianDistribution(mu, sigma)
```

0.0

```
gd.:μ
```

```
begin
    resample_btn
    D = sample(gd, n)
end;
```

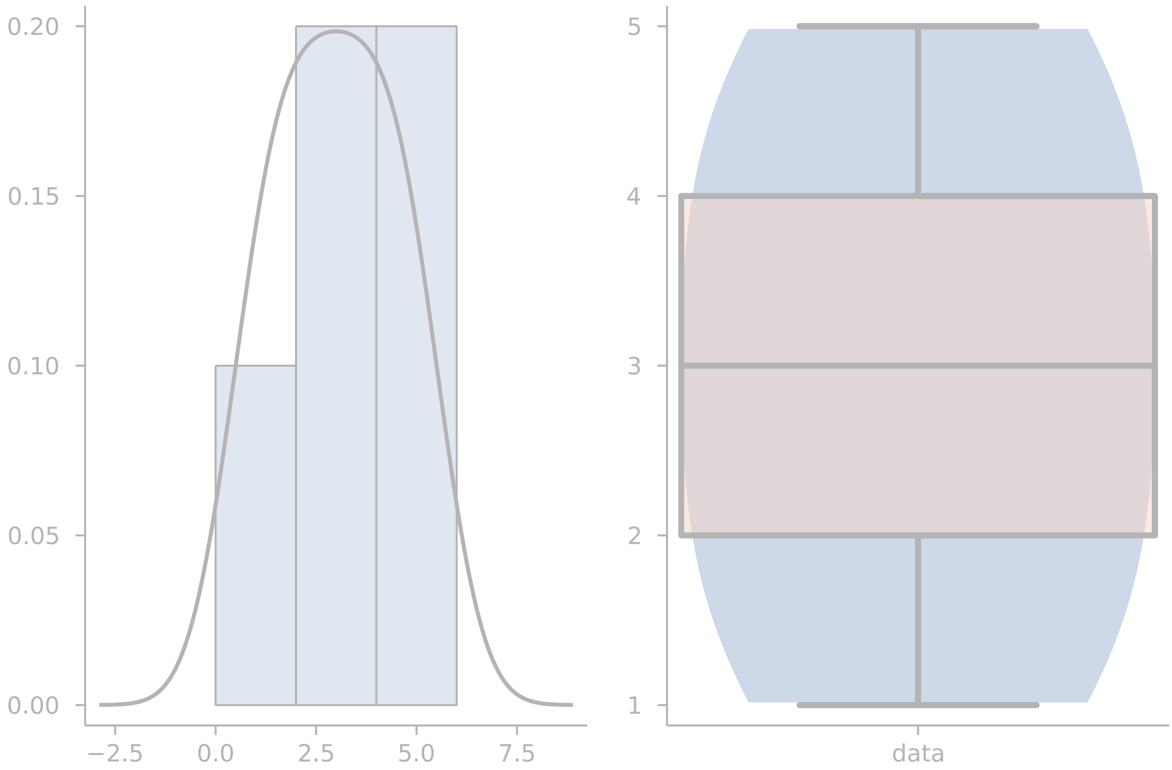
```
MethodError: no method matching mean(::Vector{Int64})
Closest candidates are:
mean(!Matched::Main.var"workspace#3".GaussianDistribution) at C:\Users\shahr\Downloads\
mean(!Matched::Vector{Float64}) at C:\Users\shahr\Downloads\A1.jl#==#6fcf8a25-07a9-4d5a
```

1. **top-level scope** @ *Local: 2*

```julia
let
    x̄ = mean(D)
    σ̄ = stddev(D)

    md"""
    **True mean:** $(gd.μ), **True Standard Deviation:** $(gd.σ)

    **Sample mean:** $(x̄), **Sample Standard Deviation:** $(σ̄)"""
end
```




```julia
let
    plt_1 = plot_density(D)
    plt_2 = plot_box_and_violin(D)
    plot(plt_1, plt_2)
end
```