Lab 03

Optimization and maximum likelihood

2023-09-08

Table of contents

# 1. Problem statement

In the wake of a severe flood, an insurance company has comissioned you to study flood damage in one of the most-affected neighborhoods. This neighborhood is perfectly flat, so we can assume that all houses experienced the same flood depth. However, they are elevated to different heights, use different materials, and are built to different standards, and as a result they experienced different amounts of damage. Specifically, the insurance company has provided you with a dataset of the fraction of the value of each house that was lost in the flood and asked you to model the distribution of losses.

## 1.1 Mathematical model

Our model for the loss fraction for the th house is

where and are the shape parameters of the [Beta Distribution](https://en.wikipedia.org/wiki/Beta_distribution). Because we are not including any explanatory information in our model, we are assuming that the distribution of loss fractions is the same for all houses. This is reasonable for our neighborhood, but would not be applicable to another neighborhood or a different flood event.

The above notation is shorthand for

where and is the Gamma function. We will work directly with the pdf and logpdf functions in the Distributions package, so you don’t need to memorize this formula..

## 1.2 Setup

As for labs 1 and 2, make sure you follow the three standard initial setup steps:

1. Open the lab 3 folder in VS Code. Do NOT open a “parent” directory containing lab 3. If you’re not sure what folder you’re in, open the Juila REPL and type pwd(). It should say something like /.../lab03-username....
2. Activate the project environment: ] to enter package mode then activate .. Don’t forget the . at the end, it’s very important.
3. Install all required packages: ] to enter package mode, then instantiate.

At this point, check to make sure you can render the document. In VS Code, open the command palette (Windows: Ctrl+Shift+P, Mac: Cmd+Shift+P) and type Render: HTML. If this gives you trouble, try the following:

1. ] in the Julia REPL to enter Package mode
2. build IJulia

If this still gives you trouble, ask for help on Canvas or in-person.

## 1.3 Package imports

As usual we start by specifying the packages we are using

using Distributions # probability distributions  
using DelimitedFiles # read data  
using LaTeXStrings # LaTeX plot labels  
using Optim # optimization  
using Plots # plotting  
using StatsPlots # plot distributions

## 1.4 About the Beta distribution

We can learn something about the Beta distribution defined above by plotting it for some different values:

p = plot(; xlabel=L"y", ylabel=L"p(y | \alpha, \beta)", legend=:top)  
for (α, color) in zip([1.0, 5.0, 25.0], [:orange, :purple, :black])  
 for (β, linestyle) in zip([1.0, 5.0, 25.0], [:solid, :dash, :dot])  
 plot!(  
 p,  
 Beta(α, β);  
 label="α = $α, β = $β",  
 color=color,  
 linestyle=linestyle,  
 linewidth=2  
 )  
 end  
end  
p

Line 1

This defines a blank plot for us to add to

Line 2

When we loop through our values of α, we zip it with a vector of colors so that we can plot each value of α in a different color.

Line 3

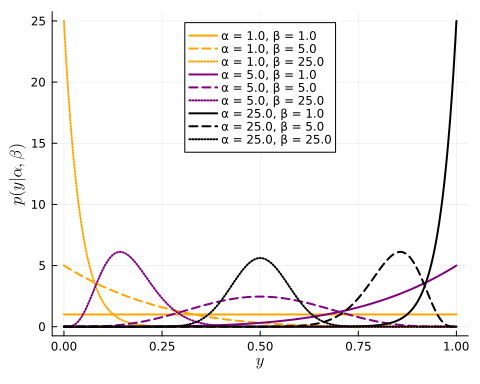
Similarly, we can attach each value of β to a different linestyle.

Line 4

plot!(p, ...) will add more elements to the plot p that we defined above.

Line 6

Using the StatsPlots package, we can plot a distribution by passing it to plot! with no additional arguments.



# 2. First steps

While the insurance company is aggregating and collecting all the data, they ask us to go ahead and get started developing a workflow. They fax (!!!) over the first three data points, rounded to two decimals, which we can read in as:

fax\_fname = joinpath("data", "fax.txt")  
fax = DelimitedFiles.readdlm(fax\_fname)[:, 1]

Line 1

We use the joinpath function to join together the path to the data directory and the filename. This is a good practice to make sure that your code works on different operating systems.

Line 2

We use the readdlm function from DelimitedFiles.jl to read the data. It is a Matrix bby default so we use [:, 1] to select all rows (:) and the first column (there is only 1).

3-element Vector{Float64}:  
 0.51  
 0.36  
 0.55

Recall that we can do things like

length(fax)

3

and

typeof(fax)

Vector{Float64} (alias for Array{Float64, 1})

to learn more about our data.

## 2.1 Likelihood model

function log\_lik(y::Vector{T}, α::T, β::T) where {T<:Real}  
 # fill in here  
 # refer to lecture slides for syntax and function names  
 # don't forget your `return` statement!  
end

|  |
| --- |
| Important |
| Fill in the log\_lik function where y is a vector of data, and α and β are the parameters of the Beta distribution. Then, convert it to live code by adding curly brackets, like you did in lab 02. *Hint: define a function function log\_lik(y::T, α::T, β::T) where {T<:Real}... that takes in a single point. Then this function that takes in a vector y can calculate the log-likelihood for each data point individually and combine them. This is not the only way to solve this problem* |

## 2.2 Check if it’s right

We can check your log\_lik function by comparing what you calculate using it to a known, correct value.

your\_val = log\_lik([0.2, 0.4, 0.6, 0.8], 5., 5.) # calls your implementation  
true\_value = -0.2947032775653282 # I calculated this  
@assert isapprox(your\_val, true\_value) # checks if they're close  
println("😁")

Convert this to a live code block by adding curly brackets, like you did in lab 02, and run. You should see a 😄.

## 2.3 Plot

Now that we have the log\_likelihood, we’re going to plot it for many values of and .

First, let’s define a grid of values for and to plot:

α\_plot = range(0.01, 50; length=500)  
β\_plot = range(0.01, 50; length=501)

1. Define them to be different lengths so that if we make an indexing mistake, we should get an error.

Next, we can calculate the log likelihood at each point on the grid of α\_plot and β\_plot as:

log\_lik\_fax = [log\_lik(fax, αi, βi) for αi in α\_plot, βi in β\_plot]  
  
# ensure that the dimensions are correct  
@assert size(log\_lik\_fax) == (length(α\_plot), length(β\_plot))  
display(size(log\_lik\_fax))

Now, we’re ready to plot. You can use the following code, which provides some helpful keyword arguments to make your plot look nice. Feel free to play around with them.

p1 = plot(  
 α\_plot,  
 β\_plot,  
 log\_lik\_fax';  
 st=:heatmap,  
 xlabel=L"$\alpha$",  
 ylabel=L"$\beta$",  
 legend=:topright,  
 colorbar\_title=L"$\log p(y | \alpha, \beta)$"  
)  
p1

|  |
| --- |
| Size and plotting |
| Note that we plot log\_lik\_fax' instead of log\_lik\_fax. That is equivalent to transpose(log\_lik\_fax). This is a quirk of plots.jl syntax (see [here](https://jverzani.github.io/CalculusWithJuliaNotes.jl/differentiable_vector_calculus/plots_plotting.html#visualizing-functions-from-r2-rightarrow-r)). We avoid confusion by checking the size of log\_lik\_fax above. |

|  |
| --- |
| Tip |
| Note that we have assigned our plot a variable name, p1. This will let us add elements to it later. |

|  |
| --- |
| Important |
| Add curly brackets to the code blocks in this section once your log\_lik function is working. |

## 2.4 Best estimates

What values of α\_plot and β\_plot that maximize the log likelihood? We can find out in several different ways.

### 2.4.1 Optimization on a grid

The easiest thing to do is to find the maximum value of the log likelihood on our grid. We can do that as follows:

idx\_fax = argmax(log\_lik\_fax) # the index that maximizes the log likelihood  
α\_fax\_best = α\_plot[idx\_fax[1]]  
β\_fax\_best = β\_plot[idx\_fax[2]]

|  |
| --- |
| Important |
| Add curly brackets to the code blocks in this section once your log\_lik function is working. |

### 2.4.2 Optim.jl

We can also use the optimize function from Optim.jl to find the maximum value of the log likelihood.

# define the function to be optimized  
loss\_fax(θ) = # define the negative log-likelihood here, using your `log\_lik` function and passing in `fax` as the `y` argument.  
lower = [0.001, 0.001] # lower bound -- don't need to change  
upper = [Inf, Inf] # upper bound -- don't need to edit  
guess = [1.0, 1.0] # initial guess -- leave as-is  
  
res = optimize(loss\_fax, lower, upper, guess) # run the optimization  
θ\_fax = Optim.minimizer(res) # get the best parameters

|  |
| --- |
| Important |
| Add curly brackets to the code blocks in this section once you have implemented everything. Follow the commented instructions. |

### 2.4.3 Distributions.jl

As an alternative to Optim.jl, we can use the fit\_mle function from Distributions.jl for most distributions. We can use this to check our work:

Distributions.fit\_mle(Beta, fax)

|  |
| --- |
| Important |
| Add curly brackets to the code blocks in this section once you have implemented everything above. Compare the fit\_mle result to your result using optimize. |

## 2.5 Update the plot

These best estimates should show up as points on our plot. We can add them as follows:

scatter!(p1, [α\_fax\_best], [β\_fax\_best]; label="Grid Search")  
scatter!(p1, [θ\_fax[1]], [θ\_fax[2]]; label="MLE")  
p1

When we add a single point, we have to wrap it in brackets [] to make it a vector. The x and y inputs to plot (or scatter!) need to be vectors.

|  |
| --- |
| Important |
| Add curly brackets to the code blocks in this section once you have implemented everything above. |

# 3. Email data

Pleased with our preliminary analysis, the insurance company emails us another batch of data. This includes 20 observations (our original 3 plus 17 more), still rounded to two decimals, which we can read in as:

email = DelimitedFiles.readdlm(joinpath("data", "email.txt"))[:, 1];  
display(length(email))  
display(typeof(email))

Line 3

display does the same thing as just typing the variable name, but it’s useful when you want to display multiple things at once.

20

Vector{Float64} (alias for Array{Float64, 1})

They ask us to regenerate the plot with this new data. Because we know that the full dataset will be arriving soon, and we don’t want to copy and paste everything three times, we decide to write a function that takes in the data and returns:

1. The plot
2. The best estimates from the grid search
3. The maximum likelihood estimates

We can define such a function as follows:

function insurance\_analysis(y::Vector{T}) where {T<:Real}  
  
 # fill in here  
 # refer to lecture slides for syntax and function names  
 # don't forget your `return` statement!  
  
 # DEFINE THE GRID TO SEARCH OVER  
 α\_plot = exp.(-4:0.05:4.05) # good to define this inside the function   
 β\_plot = exp.(-4:0.05:4) # the exp ensures > 0  
  
 # CALCULATE THE LIKELIHOOD ON THE GRID USING THE log\_like function defined above  
 # hint: use the list comprehension syntax  
 insurance\_log\_lik = #...  
  
 # CALCULATE THE BEST PARAMETER FROM THE GRID SEARCH  
 θ\_gridsearch = # ...  
  
 # CALCULATE THE MAXIMUM LIKELIHOOD ESTIMATE USING OPTIMIZATION  
 θ\_MLE = # ...  
  
 # build the plot (definitely multiple lines --  
 # put each argument on a new line as above for clarity  
 plt = plot(  
 α\_plot,  
 β\_plot,  
 insurance\_log\_lik;  
 ... # add additional keyword arguments here  
 )  
  
 # CONCLUDE WITH A RETURN STATEMENT  
 # we are allowed to return more than one thing (technically a `Tuple`)  
 return plt, θ\_MLE, θ\_gridsearch  
end;

We can check this analysis on our fax dataset and make sure it’s the same as what we did above:

plt\_fax, θ\_MLE\_fax, θ\_gridsearch\_fax = insurance\_analysis(fax)

Finally, we can run this analysis on our email data

plt\_email, θ\_MLE\_email, θ\_gridsearch\_email = insurance\_analysis(email)

|  |
| --- |
| Important |
| Implement the insurance\_analysis function above. You should copy and paste the code from above, but replace fax with y and update other variable names as appropriate. The plot should show the log-likelihood you implemented, as we did above, and should also include the MLE and grid-search estimate as clearly labeled points. |

# 4. All data

Eventually, the insurance company sends us all the data they have collected. This includes 1000 observations with no rounding, which we can read in as:

database = DelimitedFiles.readdlm(joinpath("data", "database.txt"))[:, 1]  
display(length(database))  
first(database, 3)

Line 3

We can call first on a vector, not just a DataFrame.

1000

3-element Vector{Float64}:  
 0.5114586443855297  
 0.35804353645348974  
 0.5538518630958222

|  |
| --- |
| Important |
| Using the function defined above, run the analysis on the full dataset. Store your results as plt\_database, θ\_MLE\_database, and θ\_gridsearch\_database. |

# 5. Compare and reflect

We can compare the plots from each of the three analysis steps as follows:

plot(plt\_fax, plt\_email, plt\_database; layout=(1, 3), size=(1\_350, 300))

|  |
| --- |
| Important |
| Look at the three plots and compare them. What is happening to the function as we collect more data? What is happening to our point estimates? |

# 6. Submission instructions

As you did for lab 2, you will submit your lab by pushing it to GitHub. In addition, submit the DOCX file (as for lab 2) to Canvas.