Setup

This section loads and installs all the packages. You should be setup already from assignment 1, but if not please read and follow the `instructions.md` for further details.

```
using CSV    , DataFrames    , StatsPlots    , PlutoUI    , Random    , Statistics
```

```
using LinearAlgebra    : dot, norm, norm1, norm2
```

```
using Distributions    : Uniform
```

```
PlotlyBackend()
```

```
plotly() # In this notebook we use the plotly backend for Plots.
```

!!!IMPORTANT!!!

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

Welcome Joshua George! ✅

```
student =
  (name = "Joshua George", email = "jjgeorge@ualberta.ca", ccid = "jjgeorge", idnumber = 1
```

```
student = (name="Joshua George", email="jjgeorge@ualberta.ca", ccid="jjgeorge",
idnumber=1665548)
```

Distance Metrics

Here we are defining some convenience functions for commonly used distance metrics.

```
l1_error (generic function with 1 method)
```

```
begin
    RMSE(x̂, x) = sqrt(mean(abs2.(x̂ .- x))) # abs2 is equivalent to squaring, but
faster and better numerically.
    l2_error(x̂, x) = norm2(x̂ .- x)
    l1_error(x̂, x) = norm1(x̂ .- x)
end
```

Abstract type Regressor

This is the basic Regressor interface. For the methods below we will be specializing the `predict(reg::Regressor, x::Number)`, and `epoch!(reg::Regressor, args...)` functions. Notice the `!` character at the end of epoch, as discussed earlier this this is a commonly used naming practice throughout the Julia language to indicate a function which modifies its arguments.

Main.workspace2.Regressor

```
"""
    Regressor

Abstract Type for regression algorithms. Interface includes `predict` and an
`epoch!`. In this notebook, we will only be using single variate regression.
- `predict(reg::Regressor, X::Number)`: return a prediction of the target given the
feature `x`.
- `epoch!(reg::Regressor, X::AbstractVector, Y::AbstractVector)`: trains using the
features `X` and regression targets `Y`.
"""
abstract type Regressor end # assume linear regression
```

predict (generic function with 1 method)

```
predict(reg::Regressor, x::Number) = Nothing
```

predict (generic function with 2 methods)

```
predict(reg::Regressor, X::AbstractVector) = [predict(reg, x) for x in X]
```

epoch! (generic function with 1 method)

```
epoch!(reg::Regressor, X::AbstractVector, Y::AbstractVector) = nothing
```

Baselines

In this section we will define the:

- `MeanRegressor` : Predict the mean of the training set.
- `RandomRegressor` : Predict `b*x` where `b` is sampled from a random normal distribution.
- `RangeRegressor` : Predict randomly in the range defined by the training set.

All the following baselines assume one dimension

MeanRegressor

```
epoch! (generic function with 2 methods)
```

```
begin
    """
        MeanRegressor()

    Predicts the mean value of the regression targets passed in through `epoch!`.
    """
    mutable struct MeanRegressor <: Regressor
        μ::Float64
    end
    MeanRegressor() = MeanRegressor(0.0)
    predict(reg::MeanRegressor, x::Number) = reg.μ
    epoch!(reg::MeanRegressor, X::AbstractVector, Y::AbstractVector) = reg.μ =
mean(Y)
end
```

RandomRegressor

```
predict (generic function with 4 methods)
```

```
begin
    """
        RandomRegressor

    Predicts `b*x` where `b` is sambled from a normal distribution.
    """
    struct RandomRegressor <: Regressor # random weights
        b::Float64
    end
    RandomRegressor() = RandomRegressor(randn())
    predict(reg::RandomRegressor, x::Number) = reg.b*x
end
```

RangeRegressor

```
epoch! (generic function with 3 methods)
```

```
begin
    """
        RangeRegressor

    Predicts a value randomly from the range defined by `[minimum(Y), maximum(Y)]`
as set in `epoch!`. Defaults to a unit normal distribution.
    """
    mutable struct RangeRegressor <: Regressor
        min_value::Float64
        max_value::Float64
    end
    RangeRegressor() = RangeRegressor(0.0, 1.0)

    predict(reg::RangeRegressor, x::Number) =
        rand(Uniform(reg.min_value, reg.max_value))
    predict(reg::RangeRegressor, x::AbstractVector) =
        rand(Uniform(reg.min_value, reg.max_value), length(x))
    function epoch!(reg::RangeRegressor, X::AbstractVector, Y::AbstractVector)
        reg.min_value = minimum(Y)
        reg.max_value = maximum(Y)
    end
end
```

Gradient Descent Regressors: Q3 a,b,c

In this section you will be implementing two gradient descent regressors, assuming a gaussian hypothesis class. First we will create a gaussian regressor, and then use this to build our two new GD regressors. You can test your algorithms in the **experiment section**

All the Gaussian Regressors will have data:

- `b::Float64` which is the parameter we are learning.

```
abstract type GaussianRegressor <: Regressor end
```

predict (generic function with 7 methods)

```
predict(reg::GaussianRegressor, x::Float64) = reg.b * x
```

predict (generic function with 8 methods)

```
predict(reg::GaussianRegressor, X::Vector{Float64}) = reg.b .* X
```

probability (generic function with 1 method)

```
function probability(reg::GaussianRegressor, x, y)
end
```

Stochastic Regressor ✅

The stochastic regressor will be implemented via the stochastic gradient rule

$$b_{i+1}^t = b_i^t - \eta(x_i b_i^t - y_i)x_i.$$

Where $b_{N+1}^t = b^{t+1}$, and each epoch iterates over the entire dataset in a random order.

StochasticRegressor

```
begin
    mutable struct StochasticRegressor <: GaussianRegressor
        b::Float64
        η::Float64
    end
    StochasticRegressor(η::Float64) = StochasticRegressor(0.0, 0.01)
end
```

```
epoch! (generic function with 7 methods)
```

```
# Hint: Checkout the function randperm
function epoch!(reg::StochasticRegressor,
                X::AbstractVector{Float64},
                Y::AbstractVector{Float64})
    for i=1:length(X)
        for j=1:length(Y)
            if i==j
                reg.b=reg.b-reg.η*(X[i]*reg.b-Y[j])*X[i]
            end
        end
    end
    reg.b
end


```

Batch Regressor ✅

The Minibatch regressor will be implemented via the gradient rule for a minibatch `j` with indicies for a batch defined by the set $\mathcal{I}$

$$g_t^j = \sum_{i \in \mathcal{I}_j} (x_i b_t - y_i) x_i$$
$$b_{t+1} = b_t - \eta g_t.$$

Your implementation should handle Batch Gradient Descent when the batch size is not specified. The minibatch regressor ✅ can also be implemented through this interface using the same struct and `epoch!` function.

```
BatchRegressor
```

```
begin
    mutable struct BatchRegressor <: GaussianRegressor
        b::Float64
        η::Float64
        n::Union{Int, Nothing}
    end
    BatchRegressor(η, n=nothing) = BatchRegressor(0.0, η, n)
end
```

```
epoch! (generic function with 7 methods)
```

```julia
function epoch!(reg::BatchRegressor,
                X::AbstractVector{Float64},
                Y::AbstractVector{Float64})

if reg.n==nothing
    reg.n=length(X)
    end

for j in 1:(length(X)÷reg.n)
    size=j*reg.n
    sum1=0
        for i in (size-reg.n)+1:size
            sum1=sum1+(X[i]*reg.b-Y[i])*X[i]
        end
    reg.b=reg.b-(reg.η)*sum1/reg.n
    end
    print(reg.b)
end


```

Stepsize Heuristic: Q3 d

```julia
md"""
# Stepsize Heuristic: Q3 d
"""
```

Stochastic Regressor with heuristic ✅

```
StochasticHeuristicRegressor
```

```julia
begin
    mutable struct StochasticHeuristicRegressor <: GaussianRegressor
        b::Float64
    end
    StochasticHeuristicRegressor() =
        StochasticHeuristicRegressor(0.0)
end
```

```
epoch! (generic function with 6 methods)
```

```julia
# Hint: Checkout the function randperm
function epoch!(reg::StochasticHeuristicRegressor,
                 X::AbstractVector{Float64},
                 Y::AbstractVector{Float64})
    for i=1:length(X)
        reg.b=reg.b-((X[i]*reg.b-Y[i])*X[i])/(1+abs(((X[i]*reg.b-Y[i])*X[i])))
    end
    reg.b
end

```

Batch Regressor with heuristic

- Full Batch: ✅
- Minibatch: ✅

```
let
    X, Y = [1.0, 2.0, 3.0], [1.0, 0.2, 0.1]
    bgr = BatchHeuristicRegressor()
    epoch!(bgr, X, Y)
    batch_test = predict(bgr, 1.0) ≈ 0.36170212765 ? "✅" : "❌"

    X, Y = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0], [0.32, 0.32, 0.32, 0.32, 0.32, 0.32]
    mbgr = BatchHeuristicRegressor(2)
    epoch!(mbgr, X, Y)
    mb_test = predict(mbgr, 1.0) ≈ 0.319968983713 ? "✅" : "❌"

    md"""
    ### Batch Regressor with heuristic
    - Full Batch: $(batch_test)
    - Minibatch: $(mb_test)
    """
end
```

BatchHeuristicRegressor

```
begin
    mutable struct BatchHeuristicRegressor <: GaussianRegressor
        b::Float64
        n::Union{Int, Nothing}
    end
    BatchHeuristicRegressor(n=nothing) = BatchHeuristicRegressor(0.0, n)
end
```

epoch! (generic function with 7 methods)

```
function epoch!(reg::BatchHeuristicRegressor,
                X::AbstractVector{Float64},
                Y::AbstractVector{Float64})
if reg.n==nothing
    reg.n=length(X)
    end

for j in 1:(length(X)÷reg.n)
    size=j*reg.n
    sum1=0
    for i in (size-reg.n)+1:size
        sum1=sum1+(X[i]*reg.b-Y[i])*X[i]
    end
    reg.b=reg.b-sum1/((1+abs(sum1/reg.n))*reg.n)
    end
    print(reg.b)
end
```

Data

Next we will be looking at the `height_weight.csv` dataset found in the data directory. This dataset provides three features `[sex, height, weight]`. In the following regression task we will be using `height` to predict `weight`, ignoring the `sex` feature.

The next few cells:

- Loads the dataset
- Plots distributions for the `height` and `weight` features seperated by `sex`
- Standardize the set so both `height` and `weight` conform to a standard normal.
- Defines `splitdataframe` which will be used to split the dataframe into training and testing sets.

```
# Read the data from the file in "data/height_weight.csv". DO NOT CHANGE THIS VALUE!
df_height_weight = DataFrame(CSV.File(joinpath(@__DIR__, "data/height_weight.csv"),
                             header=["sex", "height", "weight"]));
```

Successfully loaded dataset ✅

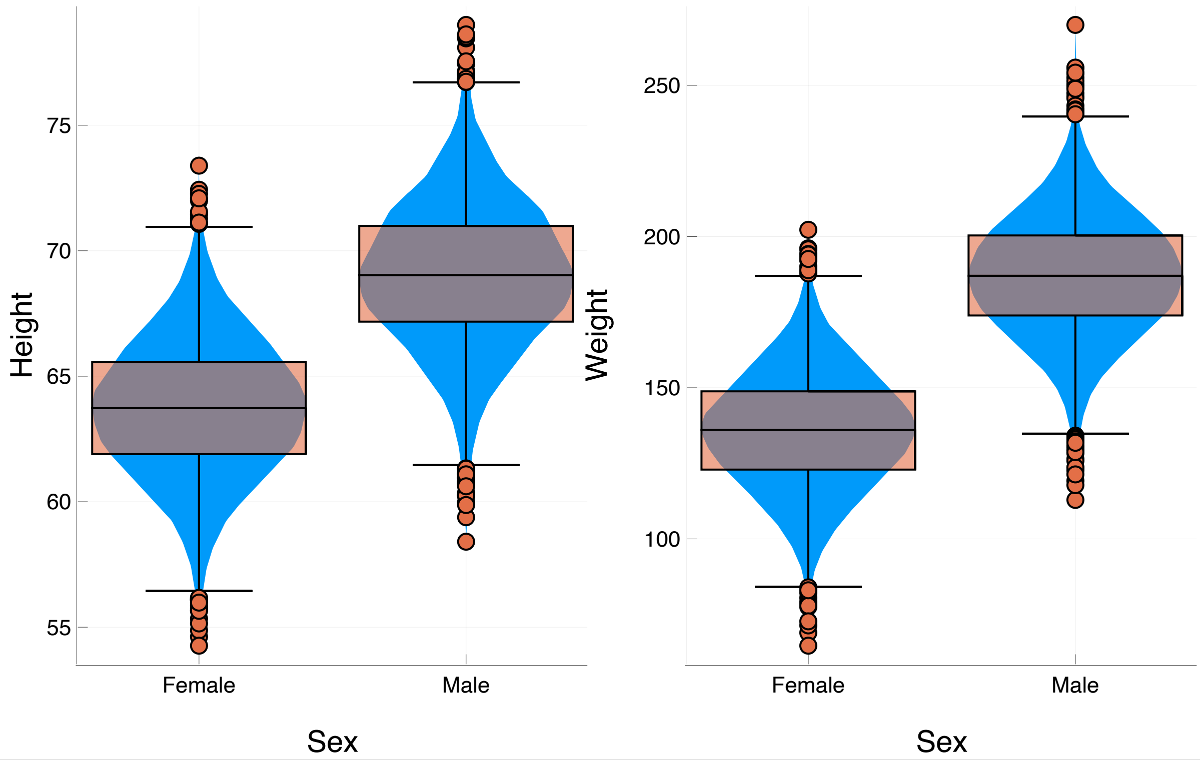
`df_hw_norm =`

	sex	height	weight
1	"Male"	1.94396	2.50567
2	"Male"	0.627505	0.0270993
3	"Male"	2.01234	1.59773
4	"Male"	1.39399	1.82513
5	"Male"	0.913375	1.39868
6	"Male"	0.230136	-0.287407
7	"Male"	0.628331	0.700362
8	"Male"	0.514865	0.203397
9	"Male"	0.169301	0.451255
10	"Male"	-0.756607	-0.156989
more			
10000	"Female"	-1.14965	-1.48843

Plot data ✅

Plot a boxplot and violin plot of the height and weight. This can be with the classes `male` and `female` combined or with them separate.

plt_hw =




```
plt_hw = let
    df = df_height_weight # For convenience in the bellow code
    nothing
    plt1 = plot(xlabel="Sex", ylabel="Height", legend=nothing)
    @df df violin!(:sex, :height, linewidth=0)
    @df df boxplot!(:sex, :height, fillalpha=0.6)

    plt2 = plot(xlabel="Sex", ylabel="Weight", legend=nothing)
    @df df violin!(:sex, :weight, linewidth=0)
    @df df boxplot!(:sex, :weight, fillalpha=0.6)

    plot(plt1, plt2)
end
```

Main.workspace2.splitdataframe

splitdataframe (generic function with 2 methods)

((X = [1.45872, 0.398926, -0.0813984, -0.00416331, 0.0392535,   more ,-0.845657], Y = [(

```
let
    #=
        A do block creates an anonymous function and passes this to the first
parameter of the function the do block is decorating.
    =#
    trainset, testset =
        splitdataframe(df_hw_norm, 0.1; shuffle=true) do df
            (X=df[!, :height], Y=df[!, :weight]) # create namedtuple from dataframes
        end
end
```

Training the Models

The following functions are defined as utilities to train and evaluate our models. While hidden below, you can expand these blocks to uncover what is happening. `run_experiment!` is the main function used below in "**Using and Analyzing you Algorithms**".

`evaluate (generic function with 1 method)`

`evaluate_l∞ (generic function with 1 method)`

`train! (generic function with 2 methods)`

`train! (generic function with 2 methods)`

`run_experiment! (generic function with 1 method)`

`run_experiment (generic function with 1 method)`

Using and Analyzing your Algorithms

In this section we will be running and analyzing a small experiment. The goal is to get familiar with analyzing data, plotting learning curves, and comparing different methods. Below we've provided a start with the baselines. Add new initilizors for a Batch update `(η = 0.01)`, a Minibatch update `(η = 0.01, n = 100)`, and a Stochastic update `(η = 0.01)`. Also add their heuristic counterparts.

As a point of reference: running

```
results = run_experiment(regressor_init, 10, 30)
```

in the cell below takes roughly `8 seconds` on my machine.

Experiment ran for:

- ✅ `Mean`
- ✅ `Random`
- ✅ `Range`
- ✅ `Stochastic` : with stepsize= `0.01` ✅
- ✅ `Batch` : with stepsize= `0.01` ✅
- ✅ `Minibatch` : with stepsize= `0.01` ✅ and batch size = `100` ✅
- ✅ `StochasticHeuristic`
- ✅ `BatchHeuristic`
- ✅ `MinibatchHeuristic` : with batch size = `100` ✅

```
regressor_init =
  Dict("MinibatchHeuristic" ⇒ #39, "Range" ⇒ #33, "Stochastic" ⇒ #34, "StochasticHeuris
```

```
regressor_init = Dict(
    "Mean"=>()->MeanRegressor(),
    "Random"=>()->RandomRegressor(),
    "Range"=>()->RangeRegressor(),
    # use the keys "Batch", "Stochastic", and "Minibatch".
    "Stochastic"=>()->StochasticRegressor(0.01),
    "Batch"=>()->BatchRegressor(0.01),
    "Minibatch"=>()->BatchRegressor(0.01, 100),
    "StochasticHeuristic"=>()->StochasticHeuristicRegressor(),
    "BatchHeuristic"=>()->BatchHeuristicRegressor(),
    "MinibatchHeuristic"=>()->BatchHeuristicRegressor(100)


)
```

```
results =
  Dict("MinibatchHeuristic" ⇒ [(regressor = BatchHeuristicRegressor(0.924241, 100), trair
```

```
results = run_experiment(regressor_init, 10, 30)
```

The results dictionary is the resulting data from the experiment we run using `regressor_init` as the intializors. You will see the same keys used as in the `regressor_init` dictionary. For each run the experiment returns the final regressor, the training error vector, and the final test error. You can get one of these components for a particular methd using `getindex` and broadcasting:

```
getindex.(results["Mean"], :test_error)
```

```
0.21771749433405885
```

```
let
    # Play with data here! You can explore how to get different values.
    mean(getindex.(results["Mean"], :test_error))
    mean(getindex.(results["Random"], :test_error))
    mean(getindex.(results["Range"], :test_error))
    mean(getindex.(results["Stochastic"], :test_error))
    mean(getindex.(results["Batch"], :test_error))
    mean(getindex.(results["Minibatch"], :test_error))
    mean(getindex.(results["StochasticHeuristic"], :test_error))
    mean(getindex.(results["BatchHeuristic"], :test_error))
    mean(getindex.(results["MinibatchHeuristic"], :test_error))
    std(getindex.(results["Mean"], :test_error))
    std(getindex.(results["Random"], :test_error))
    std(getindex.(results["Range"], :test_error))
    std(getindex.(results["Stochastic"], :test_error))
    std(getindex.(results["Batch"], :test_error))
    std(getindex.(results["Minibatch"], :test_error))
    std(getindex.(results["StochasticHeuristic"], :test_error))
    std(getindex.(results["BatchHeuristic"], :test_error))
    std(getindex.(results["MinibatchHeuristic"], :test_error))

end
```

```
Enter cell code...
```

Learning Curves ✅

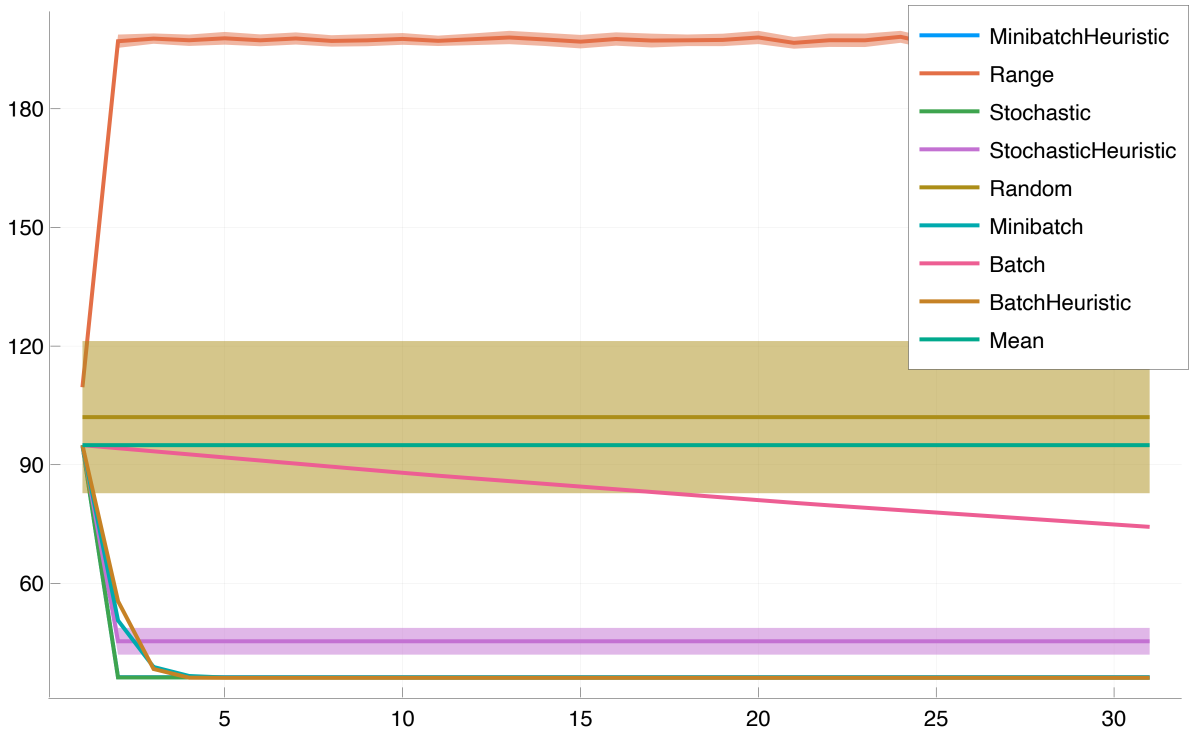
Plot the average learning curve with the standard error calculated as

$$\sigma_{err}(\mathbf{x}) = \sqrt{\frac{\mathrm{Var}(x)}{|x|}}$$

Note that $\mathbf{x}$ is a vector over runs, not over epochs.

Note: if you notice one method is dominating the plot, change the axis limits to make sure the methods we are most concerned with (i.e. Stochastic, Batch, and Minibatch) are visible.

`plt_lc =`

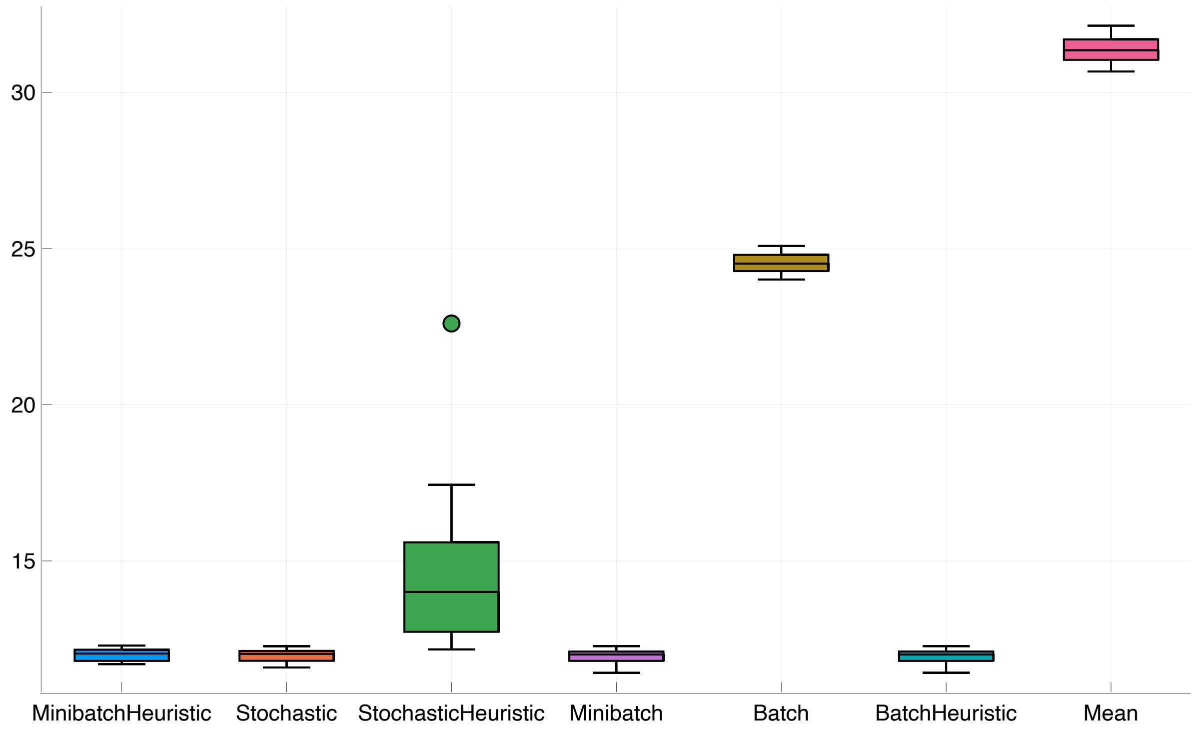



```
plt_lc = let
    plt = plot()
    # plt=plot(lw=2)
    for method ∈ keys(results)
        μ = mean(getindex.(results[method], :train_error))
        σ = sqrt.(var(getindex.(results[method], 2)) / length(results[method]))
        plot!(plt, μ, ribbon=σ, lw=2, label=method)
    end
    plt
end
```

Final Errors ✅

Finally, we want to compare the final test errors of the different methods. One way to do this is through box plots. See **this great resource** to learn how to compare data using a box and whisker plot. In this plot you can ignore the `Range` and `Random` baselines.

plt_fe =




```
plt_fe = let
    plt = plot(legend=nothing)
    for method ∈ keys(results)
        if method ∈ ["Range", "Random"]
            continue
        end
        data = getindex.(results[method], :test_error)
        boxplot!([method], data)
    end
    plt
end
```