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.

!!!IMPORTANT!!!

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

Welcome Sanad Masannat! ✅

```
student =
  (name = "Sanad Masannat", email = "sanad@ualberta.ca", ccid = "sanad", idnumber = 162622
```

```
student = (name="Sanad Masannat", email="sanad@ualberta.ca", ccid="sanad",
```

Important Note: You should only write code in the cells that has:

```
#### BEGIN SOLUTION


```

Preamble

In this assignment, we will implement:

- Q3(a,b) **Stochastic Regressor** stochastic regressor ✅
- Q3(c) **Batch Regressor** batch regressor ✅, and minibatch regressor ✅
- Q3(d) **Stochastic Heuristic Regressor** ✅
- Q3(e,f) **Batch Heuristic Regressor** full batch ✅, and minibatch ✅

You can test your algorithms in **experiment section**.

Distance Metrics

Here we define some functions for commonly used distance metrics, to later allows us to measure errors.

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

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

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. This is a common naming convention in Julia to indicate that this is a function that modifies its arguments.

```
Main.workspace2.Regressor
```

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

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

```
epoch! (generic function with 1 method)
```

Baselines

In this section we will define the:

- `MeanRegressor`: Predict the mean of the training set.
- `RandomRegressor`: Predict `w*x` where `w` is sampled from a random normal distribution.

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)
```

RandomRegressor

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

```
begin
    """
        RandomRegressor

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

Q3 a,b,c: Gradient Descent Regressors

In this section you will be implementing two gradient descent regressors, assuming $p(y|x)$ is Gaussian with the update given in the assignment pdf. 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:

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

```
predict (generic function with 5 methods)
```

```
predict (generic function with 6 methods)
```

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

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

Stochastic Regressor ✅

```
begin
    __check_stoch_reg = let
        sgr = StochasticRegressor(0.0, 0.1)
        epoch!(sgr, [1.0, 1.0], [1.0, 1.0])
        predict(sgr, 1.0) ≈ 0.19
    end

    HTML("<h3 id=q3ab> Stochastic Regressor $(_check_complete(__check_stoch_reg))</h3>")
```

The stochastic regressor will be implemented via the stochastic gradient rule

$$w_{t+1} = w_t - \eta(x_i w_t - y_i)x_i$$

by starting from an initial $w_0 = 0$ and doing multiple epochs over the dataset. Each epoch corresponds to iterating once over the entire dataset, in a random order. Here you only have to implement the function that does one epoch. It will be called multiple times in `train!`, which is a function provided for you described in the section on Training the Models. After you implement your algorithm, you should test it in the experiments section to answer Q3 b. Note: to obtain the length of an array `arr`, you can use `length(arr)`. If you see an error that says "MethodError: no method matching...", then this means you have an error in your implementation.

`StochasticRegressor`

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

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

```
# Hint: Checkout the function randperm
function epoch!(reg::StochasticRegressor,
                X::AbstractVector{Float64},
                Y::AbstractVector{Float64})
    # Your code here is for question 3(a)
   indices = randperm(length(X))
    for i in indices
        temp= reg.w*X[i]
        temp -=Y[i]
        temp *= X[i]
        reg.w -= (reg.η*temp)
    end
```

Batch Regressor

- batch ✅
- minibatch ✅

```
begin
    __check_batch_reg = let
        X, Y = [1.0, 2.0, 3.0], [1.0, 0.2, 0.1]
        bgr = BatchRegressor(0.1*3)
        epoch!(bgr, X, Y)
        predict(bgr, 1.0) ≈ 0.17
    end

    __check_mb = let
        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 = BatchRegressor(0.1*2, 2)
        epoch!(mbgr, X, Y)
        predict(mbgr, 1.0) ≈ 0.15616
    end

    HTML("<h3 id=q3c> Batch Regressor</h3>
    <ul><li> batch $(_check_complete(__check_batch_reg))</li> <li>minibatch
$(_check_complete(__check_mb)) </li></ul>")
```

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

$$g_t^j = \sum_{i \in \mathcal{I}_j} (x_i w_t - y_i) x_i$$

$$w_{t+1} = w_t - \eta g_t^j.$$

If no batch size `b` is specified, then your implementation should run Batch Gradient Descent. Once again you need to implement the `epoch!` function, which will be called elsewhere by `train!`.

BatchRegressor

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

epoch! (generic function with 4 methods)

```
function epoch!(reg::BatchRegressor,
                X::AbstractVector{Float64},
                Y::AbstractVector{Float64})
    # Your code here is for question 3(c)
    # You will need to move to the experiment section to test your implementation
    #### BEGIN SOLUTION
    if isnothing(reg.b)
        sum=0.0
        for i in 1:length(X)
            temp= reg.w*X[i]
            temp -=Y[i]
            temp *= X[i]
            sum+=temp

        end
        sum*= 1/length(X)
        reg.w -= (reg.η*sum)
    else
        batch_num=Int(length(X)/reg.b)
        for i in 1:(batch_num)
            sum=0.0
            for j in 1:reg.b
                index=Int(j+(i-1)*reg.b)
                temp= reg.w*X[index]
                temp -=Y[index]
                temp *= X[index]
                sum+=temp

            end
            sum *= 1/reg.b
            reg.w-= sum*reg.η
        end
    end

    #### END SOLUTION
```

Stepsize Heuristic

Q3 d: Stochastic Regressor with heuristic ✅

```
begin
    __check_shr = let
        sgr = StochasticHeuristicRegressor(0.0)
        epoch!(sgr, [1.0, 1.0], [1.0, 1.0])
        predict(sgr, 1.0) * 6 ≈ 5.0
    end
    HTML("<h3 id=q3d> Q3 d: Stochastic Regressor with heuristic
$(_check_complete(__check_shr))")
```

StochasticHeuristicRegressor

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

epoch! (generic function with 6 methods)

```
# Hint: Checkout the function randperm
function epoch!(reg::StochasticHeuristicRegressor,
                X::AbstractVector{Float64},
                Y::AbstractVector{Float64})
    # Your code here is for 3(d)
   indices = randperm(length(X))
    η=0
    for i in indices
        temp= reg.w*X[i]
        temp -=Y[i]
        temp *= X[i]
        gt=abs(temp)
        gt+=1
        η=1/gt
        reg.w-=η*temp
    end
```

Q3 e: Batch Regressor with heuristic

- Full Batch: ✅
- Minibatch: ✅

BatchHeuristicRegressor

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

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

```
function epoch!(reg::BatchHeuristicRegressor,
                X::AbstractVector{Float64},
                Y::AbstractVector{Float64})
        if isnothing(reg.b)
        sum=0.0
        for i in 1:length(X)
            temp= reg.w*X[i]
            temp -=Y[i]
            temp *= X[i]
            sum+=temp

        end
        sum*= 1/length(X)
        η=1/(abs(sum)+1)
        reg.w -= (η*sum)
    else
        batch_num=Int(length(X)/reg.b)
        for i in 1:(batch_num)
            sum=0.0
            for j in 1:reg.b
                index=Int(j+(i-1)*reg.b)
                temp= reg.w*X[index]
                temp -=Y[index]
                temp *= X[index]
                sum+=temp
            end
            sum*= 1/reg.b
            η=1/(abs(sum)+1)
            reg.w -= (η*sum)
        end
    end
end
```

Plotting Utilities

Below we define two plotting helper functions for using PlotlyJS. You can ignore these if you want. We use them below to compare the algorithms.

```
color_scheme =
```

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

```
function boxplot(
        names::Vector{String},
        data::Vector{<:AbstractVector};
        col=color_scheme,
        kwargs...)
    colors =
    traces = GenericTrace{Dict{Symbol, Any}}[]
    for (idx, (name, datum)) in enumerate(zip(names, data))
        tr_bx = box(
            name=name,
            y=datum,
            jitter=0.3,
            marker_color=col[idx])
        push!(traces, tr_bx)
    end
    layout = Layout(; showlegend=false, kwargs...)
    plt = Plot(traces, layout)
```

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

```
function learning_curve(names, data; err, col = color_scheme, kwargs...)
    trcs = GenericTrace{Dict{Symbol, Any}}[]
    for (idx, (n, d)) in enumerate(zip(names, data))
        terr = scatter(;
            x = vcat(1:length(d), length(d):-1:1),
            y = vcat(d.-err[idx], d.+err[idx]),
            fill="tozerox",
            fillcolor="rgba($(col[idx].r), $(col[idx].g), $(col[idx].b), 0.2)",
            line_color="transparent")
        t = scatter(;x = 1:length(d),
                y=d,
                line_color=col[idx],
                name=n)
        push!(trcs, t)
    end
    layout = Layout(;kwargs...)
    Plot(trcs, layout)
```

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"),
```

Successfully loaded dataset ✅

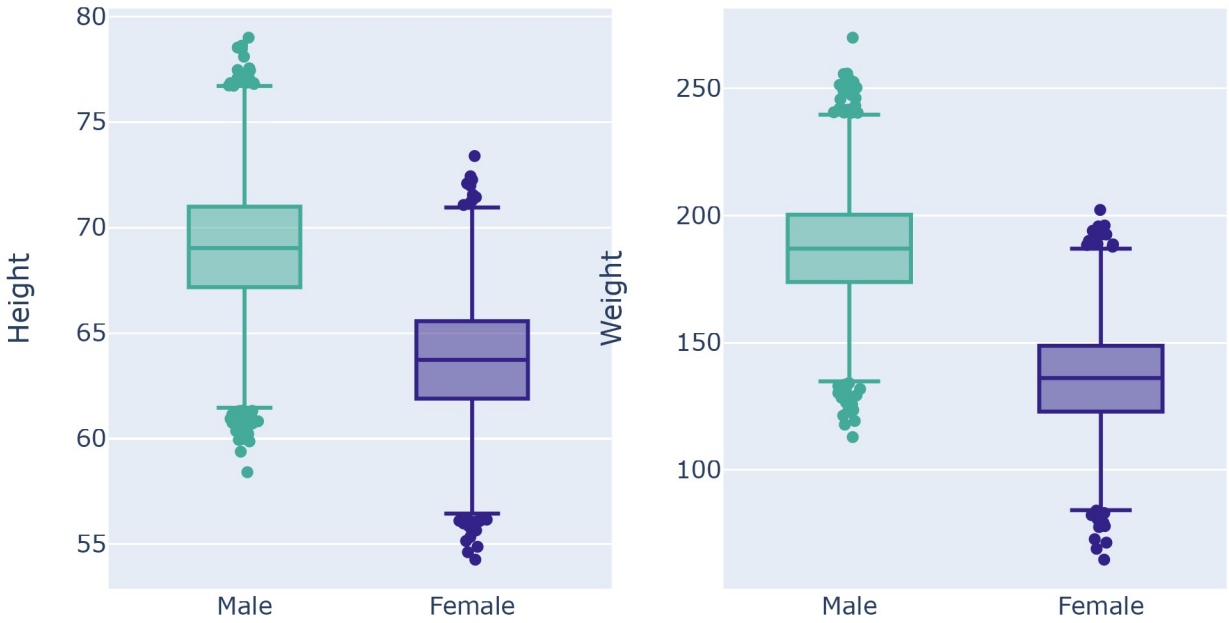
`df_hw_norm =`

	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)
    sex_names = ["Male", "Female"]
    get_attr = (sex, attr) -> df[df.sex .== sex, :][!, attr]
    p1 = boxplot(sex_names,
            get_attr.(sex_names, :height),
            yaxis=attr(title="Height"))

    p2 = boxplot(sex_names,
                get_attr.(sex_names, :weight),
                yaxis=attr(title="Weight"))

    p = [p1 p2]
    PlotlyJS.relayout(p, height=400, showlegend=false)

```

```
Main.workspace2.splitdataframe
```

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

```
((X = [1.25346, -0.0339669, 0.661534, -1.79859, 0.609185,    more ,2.87953], Y = [0.6977
```

```
let
    #=
        A do block creates an anonymous function and passes this to the first paramete
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
```

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.

To run these experiments and draw plots use ☑

Experiment ran for:

- ✅ `Mean`
- ✅ `Random`
- ✅ `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" ⇒ #11, "Stochastic" ⇒ #9, "StochasticHeuristic" ⇒ #12, "Rar
```

```
regressor_init = Dict(
    # use the keys "Batch", "Stochastic", and "Minibatch", "StochasticHeuristic"
"BatchHeuristic", and "MinibatchHeuristic"
    # This is the actual experiment section
    "Mean"=>()->MeanRegressor(),
    "Random"=>()->RandomRegressor(),
    "Batch"=>()->BatchRegressor(0.01, nothing),
    "Minibatch"=>()->BatchRegressor(0.01, 100),
    "Stochastic"=>()->StochasticRegressor(0.01),
    "BatchHeuristic"=>()->BatchHeuristicRegressor(),
    "MinibatchHeuristic"=>()->BatchHeuristicRegressor(100),
    "StochasticHeuristic"=>()->StochasticHeuristicRegressor()
```

```
results =
  Dict("MinibatchHeuristic" ⇒ [(regressor = BatchHeuristicRegressor(0.92423, 100), train_
```

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)
```

```

let
    if __run_q3_experiments
        # Play with data here! You can explore how to get different values.
        println("Stochastic")
        println(getindex.(results["Stochastic"], :test_error))
        println("Mean")
        println(getindex.(results["Mean"], :test_error))
        println("Random")
        println(getindex.(results["Random"], :test_error))
        println("Adaptive Mini Batch")
        println(getindex.(results["MinibatchHeuristic"], :test_error))
        println("Adaptive Stochastic")
        println(getindex.(results["StochasticHeuristic"], :test_error))
        println("mini Batch")
        println(getindex.(results["Minibatch"], :test_error))
    end
```

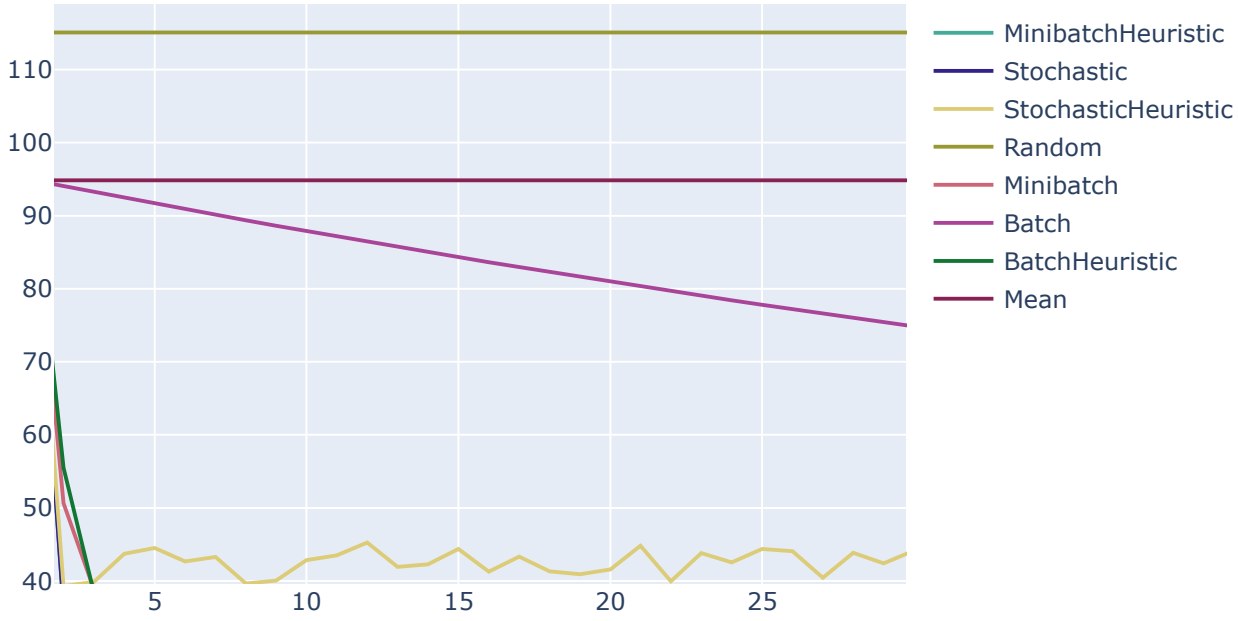
Learning Curves

Plot the average learning curve with the standard error calculated as

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

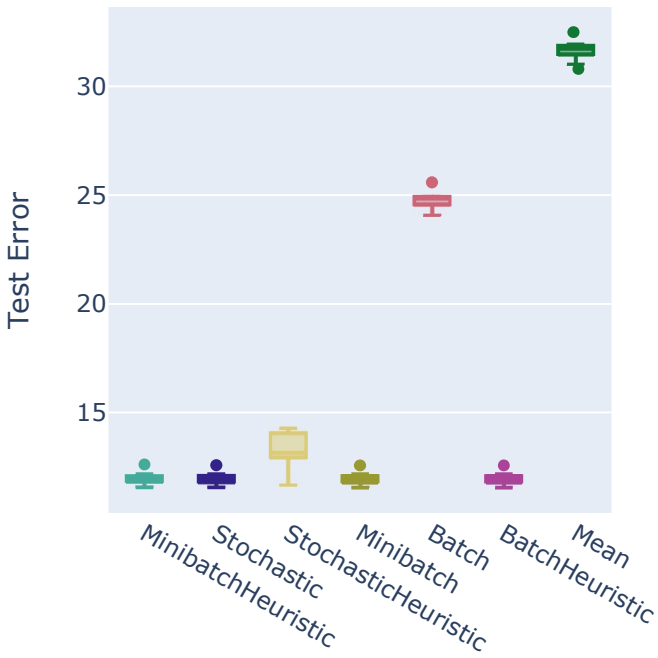
Note that $\mathbf{x}$ is a vector over runs, not over epochs.

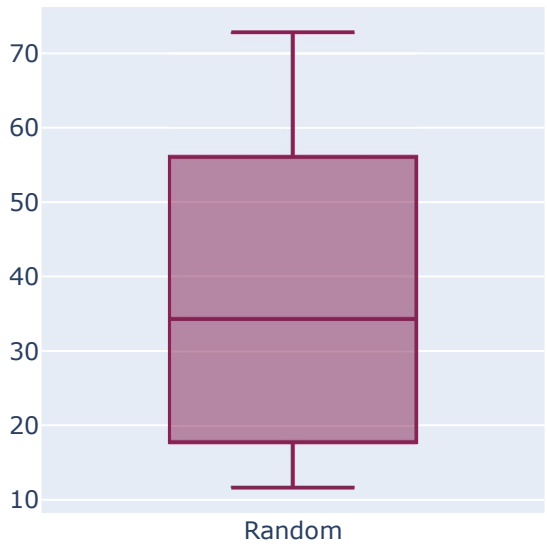
Note: if you notice one method is dominating the plot, change the axis limits to make sure the methods we are most concerned with (i.e. Stochastic, Batch, and Minibatch) are visible.



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 `Random` baseline.


Test Error
30
25
20
15
MinibatchHeuristic
Stochastic
StochasticHeuristic
Minibatch
Batch
BatchHeuristic
Mean


70
60
50
40
30
20
10
Random