# 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.

```
begin
    using CSV    , DataFrames    , StatsPlots    , PlutoUI    , Random    , Statistics
    using LinearAlgebra    : dot, norm, norm1, norm2, I
    using Distributions    : Distributions, Uniform
    using MultivariateStats    : MultivariateStats, PCA
    using StatsBase    : StatsBase

end
```

```
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)
```

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

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

```

```
#### BEGIN SOLUTION


#### END SOLUTION
```

# Preamble

In this assignment, we will implement:

- Q1 **Distributional Regression** ✅
- Q2(a) **Polynomial Features** ✅
- Q2(b) **Mini-batch Gradient Descent** ✅
- Q2(c,d) **Loss functions** MSE ✅ , and gradeint of MSE ✅
- Q2(e-g) **Optimizers**: Constant LR ✅ , Heuristic Stepsize ✅ , and AdaGrad ✅

```
let
    q1_check= _check_complete(__check_dist_reg)
    q2_a_check = _check_complete(_check_Poly2)
    q2_b_check = _check_complete(__check_MBGD)
    q2_c_check = _check_complete(__check_mseloss)
    q2_d_check = _check_complete(__check_msegrad)
    q2_e_check = _check_complete(_check_ConstantLR)
    q2_f_check = _check_complete(_check_HeuristicLR)
    q2_g_check = _check_complete(__check_AdaGrad)

md"""
# Preamble

In this assignment, we will implement:
- Q1 [Distributional Regression](#dist) $(q1_check)
- Q2(a) [Polynomial Features](#graddescent) $(q2_a_check)
- Q2(b) [Mini-batch Gradient Descent](#lossfunc) $(q2_b_check)
- Q2(c,d) [Loss functions](#lossfunc) MSE $(q2_c_check), and gradeint of MSE $(q2_d_check)
- Q2(e-g) [Optimizers](#opt): Constant LR $(q2_e_check), Heuristic Stepsize $(q2_f_check), and AdaGrad $(q2_g_check)
"""
end
```

# Q1: Distribution Regression ✅

```
begin
    __check_dist_reg = let
        m = DistributionRegressor()
        epoch!(m, 1.0, 1.0)
        epoch!(m, 1.0, 0.1)
        m.b == 0.24683544303797472 && m.a == -0.26582278481012656
    end
    HTML("<h1 id=dist> Q1: Distribution Regression $(_check_complete(__check_dist_reg))")
end
```

This is a followup question on Question 3 from Assignment 2. As before, you will implement an algorithm to estimate $p(y|x)$, for a data batch of pairs of $(x, y)$: $\mathcal{D} = \{ (x_i,y_i)\}_{i=1}^n$.

DistributionRegressor is a multivariate regressor with two variables $\mu$ and $\sigma^2$ for a gaussian distribution. Assume that $\mu$ is a linear function of $b$ with $\mu = xb$, and $\sigma^2 = exp(xa)$ depends on prameter $a$. Then for a randomly sampled $x_i, y_i$, you first need to derive the updates for both parameters $a$ and $b$ by calculating the following partial derivatives of the normal distribution

$$\frac{\partial c(w_t)}{\partial b} \qquad \& \qquad \frac{\partial c(w_t)}{\partial a}$$

in Q1(a), where $c(w)$ is the loss function, proportional to the negative log likelihood for this problem.

Q1(b): Implement the updates you have derived in Q1(a) by iterating over the entire dataset in a random order in each epoch:

$$\begin{aligned}
b_{t+1} &= b_t - \eta_t \frac{\partial c(w_t)}{\partial b} \\
a_{t+1} &= a_t - \eta_t \frac{\partial c(w_t)}{\partial a}
\end{aligned}$$

The heuristic for implementing an adaptive stepsize would be:

$$\eta_t = \left(1 + \sqrt{\left(\frac{\partial c(w_t)}{\partial b}\right)^2 + \left(\frac{\partial c(w_t)}{\partial a}\right)^2}\right)^{-1}$$

```
md"""

This is a followup question on Question 3 from Assignment 2. As before, you will implement an algorithm to estimate $p(y | x)$, for a data batch of pairs of $(x,y)$: $\mathcal{D} = \{ (x_i,y_i)\}_{i=1}^n$.

DistributionRegressor is a multivariate regressor with two variables $\mu$ and $\sigma^2$ for a gaussian distribution. Assume that $\mu$ is a linear function of $b$ with $\mu = xb$, and $\sigma^2 = exp(xa)$ depends on prameter $a$.
Then for a randomly sampled $x_i, y_i$, you first need to derive the updates for both parameters $a$ and $b$ by calculating the following partial derivatives of the normal distribution
```math
\begin{align*}
\frac{\partial c(w_t)}{\partial b} \qquad \& \qquad \frac{\partial c(w_t)}{\partial a}
\end{align*}
``` in Q1(a), where $c(w)$ is the loss function, proportional to the negative log likelihood for this problem.


Q1(b): Implement the updates you have derived in Q1(a) by iterating over the entire dataset in a random order in each epoch:

```math
\begin{align*}
b_{t+1} &= b_{t} - \eta_{t}\frac{\partial c(w_t)}{\partial b}
\\
a_{t+1} &= a_{t} - \eta_{t} \frac{\partial c(w_t)}{\partial a}
\end{align*}
```
```

```
```
The heuristic for implementing an adaptive stepsize would be:
```math
\begin{align*}
\eta_{t} &=\left(1+\sqrt{\left(\frac{\partial c(w_t)}{\partial b}\right)^{2} +
\left(\frac{\partial c(w_t)}{\partial a}\right)^{2}}\right)^{-1} \
\end{align*}
```
"""
```

predict (generic function with 1 method)

```
begin
    """

    """
    mutable struct DistributionRegressor
        b::Float64
        a::Float64
    end
    DistributionRegressor() = DistributionRegressor(0.0, 0.0)
    predict(reg::DistributionRegressor, x::Float64) = reg.b * x, exp(reg.a * x)
end
```

epoch! (generic function with 1 method)

```
function epoch!(reg::DistributionRegressor, x::Float64, y::Float64)
    # Based on the above
    #### BEGIN SOLUTION
    p=x*(1-(y-x*reg.b)^(2)*exp(-x*reg.a))*(1/2)
    q=(1/exp(reg.a*x))*(x*reg.b-y)*x
    n=(1/(1+sqrt(p^2+q^2)))
    reg.b=reg.b-n*q
    reg.a=reg.a-n*p


    #### END SOLUTION
end
```

train! (generic function with 7 methods)

```
# Stochastic Gradent Descent for DistributionModel
function train!(reg::DistributionRegressor, X, Y, num_epochs)
    for i in 1:num_epochs
        for j in randperm(length(X))
            epoch!(reg, X[j], Y[j])
        end
    end
end
```

# Q1(c) Testing the Distribution Regressor
```

In the cell below you will compare your implementation of the distribution regressor against the following:

- Mean Regressor
- Range Regressor

```
begin
    md"""
    In the cell below you will compare your implementation of the distribution regressor against the following:
    - Mean Regressor
    - Range Regressor
    """
end
```

```
(RangeRegressor, MeanRegressor)
```

```
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
        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 train!(reg::RangeRegressor, X::AbstractVector, Y::AbstractVector, args...)
        reg.min_value = minimum(Y)
        reg.max_value = maximum(Y)
    end

    """
        MeanRegressor()

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

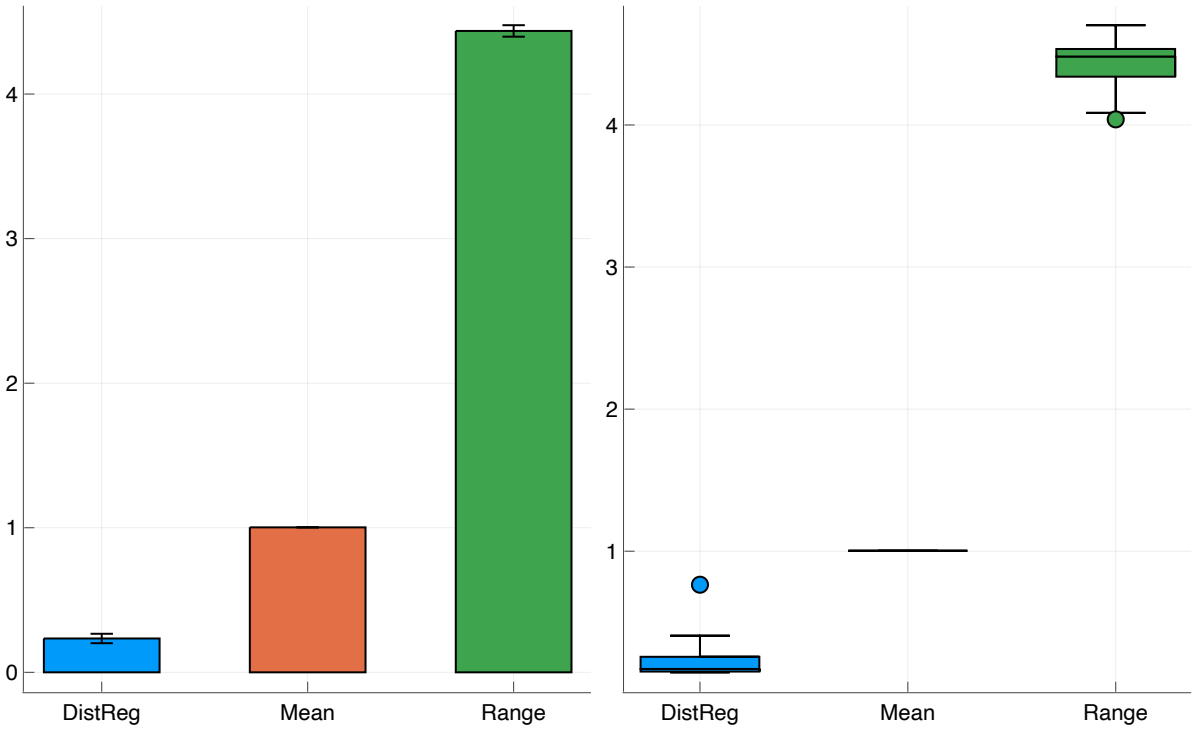
    RangeRegressor, MeanRegressor
end
```

We test the distribution regression against the mean and range baselines implemented above over 20 runs with standard error bars. We use the height and weight dataset from Assignment 2.

```
md"""
We test the distribution regression against the mean and range baselines implemented above over 20 runs with standard error bars. We use the height and weight dataset from Assignment 2.
"""
```

Compare the performance of DistributionRegressor against the baseline algorithms by reporting the average error and standard error after 1 epoch and after 20 epochs over 20 runs. You should only change num_epochs variable and report the resultant output. **You can get the average error and standard error to report from the plot or from the terminal where you run this notebook**.

```
begin
    md"""
    Compare the performance of DistributionRegressor against the baseline algorithms by reporting the average error and standard error after 1 epoch and after 20 epochs over 20 runs. You should only change num_epochs variable and report the resultant output. **You can get the average error and standard error to report from the plot or from the terminal where you run this notebook**.
    """
end
```




```
let
    trainset, testset = splitdataframe(df_hw_norm, 0.1; shuffle=true) do df
        (X=df[!, :height], Y=df[!, :weight]) # create named tuple from DF
    end
    num_runs = 20
    num_epochs = 20

    err_d = zeros(num_runs)
    for r in 1:num_runs
        m = DistributionRegressor()
        train!(m, trainset.X, trainset.Y, num_epochs)
        err_d[r] = mean(abs2, getindex.(predict.((m,), testset.X), 1) - testset.Y)
    end
    mean_error_d = mean(err_d)
    std_error_d = sqrt(var(err_d)/num_runs)

    println("Test error finished after $num_epochs epochs.")

    println("For DistributionRegressor: The mean is $mean_error_d and the standard
error is $std_error_d")

    err_m = zeros(num_runs)
    for r in 1:num_runs
        m = MeanRegressor()
        train!(m, trainset.X, trainset.Y, num_epochs)
        err_m[r] = mean(abs2, getindex.(predict.((m,), testset.X), 1) - testset.Y)
    end
    mean_error_m = mean(err_m)
    std_error_m = sqrt(var(err_m)/num_runs)

    println("For MeanRegressor: the mean is $mean_error_m and the standard error is
$std_error_m")

    err_r = zeros(num_runs)
    for r in 1:num_runs
        m = RangeRegressor()
        train!(m, trainset.X, trainset.Y, num_epochs)
        err_r[r] = mean(abs2, getindex.(predict.((m,), testset.X), 1) - testset.Y)
    end
    mean_error_r = mean(err_r)
    std_error_r = sqrt(var(err_r)/num_runs)

```

```
    println("For RangeRegressor: the mean is $mean_error_r and the standard error is $std_error_r")

    plt = bar(["DistReg" "Mean" "Range"], [mean_error_d mean_error_m mean_error_r], yerr=[std_error_d std_error_m std_error_r], legend=nothing)

    plt2 = boxplot(["DistReg" "Mean" "Range"], [err_d err_m err_r], legend=nothing)

    plot(plt, plt2)
end
```

# Q2: Multi-variate Regression

So far, we have only considered our algorithms when the features are drawn from a single dimension. But this is a considerable limitation. In the following section we will explore implementations of algorithms for multi-variate regression.

Unlike before, instead of having a struct be all the properties of an ML systems we will break our systems into smaller pieces. This will allow us to more easily take advantage of code we've already written, and will be more useful as we expand the number of algorithms we consider. We make several assumptions to simplify the code, but the general type hierarchy can be used much more broadly.

We split each system into:

- Model
- Gradient descent procedure
- Loss Function
- Optimization Strategy

```
md"""
# Q2: Multi-variate Regression

So far, we have only considered our algorithms when the features are drawn from a single dimension. But this is a considerable limitation. In the following section we will explore implementations of algorithms for multi-variate regression.

Unlike before, instead of having a struct be all the properties of an ML systems we will break our systems into smaller pieces. This will allow us to more easily take advantage of code we've already written, and will be more useful as we expand the number of algorithms we consider. We make several assumptions to simplify the code, but the general type hierarchy can be used much more broadly.

We split each system into:
- Model
- Gradient descent procedure
- Loss Function
- Optimization Strategy
"""
```

# Baselines

```
md"""
## Baselines
"""
```

# Mean Model

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

```
begin
    """
        MeanModel()

    Predicts the mean value of the regression targets passed in through `epoch!`.
    """
    mutable struct MeanModel <: AbstractModel
        μ::Float64
    end
    MeanModel() = MeanModel(0.0)
    predict(reg::MeanModel, X::AbstractVector) = reg.μ
    predict(reg::MeanModel, X::AbstractMatrix) = fill(reg.μ, size(X,1))
    Base.copy(reg::MeanModel) = MeanModel(reg.μ)
    function train!(::MiniBatchGD, model::MeanModel, lossfunc, opt, X, Y,
num_epochs)
        model.μ = mean(Y)
    end
end
```

# RandomModel

```
md"""
### RandomModel
"""
```

```
train! (generic function with 4 methods)
```

```
begin
    """
        RandomModel

    Predicts `b*x` where `b` is sambled from a normal distribution.
    """
    struct RandomModel <: AbstractModel # random weights
        W::Matrix{Float64}
    end
    RandomModel(in, out) = RandomModel(randn(in, out))
    predict(reg::RandomModel, X::AbstractMatrix) = X*reg.W
    Base.copy(reg::RandomModel) = RandomModel(randn(size(reg.W)...))
    train!(::MiniBatchGD, model::RandomModel, lossfunc, opt, X, Y, num_epochs) =
        nothing
end
```

# RangeModel

```
md"""
### RangeModel
"""
```

```
train! (generic function with 5 methods)
```

```
begin
    """
        RangeModel

    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 RangeModel <: AbstractModel
        min_value::Float64
        max_value::Float64
    end
    RangeModel() = RangeModel(0.0, 1.0)

    predict(reg::RangeModel, x::AbstractMatrix) =
        rand(Uniform(reg.min_value, reg.max_value), size(x, 1))
    Base.copy(reg::RangeModel) = RangeModel(reg.min_value, reg.max_value)
    function train!(::MiniBatchGD, model::RangeModel, lossfunc, opt, X, Y, num_epochs)
        model.min_value = minimum(Y)
        model.max_value = maximum(Y)
    end
end
```

# Models

```
md"""
# Models

"""
```

# The model interface

- `AbstractModel` : This is an abstract type which is used to derive all the model types in this assignment
- `predict` : This takes a matrix of samples and returns the prediction doing the proper data transforms.
- `get_features` : This transforms the features according to the non-linear transform of the model (which is the identity for linear).
- `get_linear_model` : All models are based on a linear model with transformed features, and thus have a linear model.
- `copy` : This returns a new copy of the model.

```
md"""
## The model interface

- `AbstractModel`: This is an abstract type which is used to derive all the model types in this assignment
- `predict`: This takes a matrix of samples and returns the prediction doing the proper data transforms.
- `get_features`: This transforms the features according to the non-linear transform of the model (which is the identity for linear).
- `get_linear_model`: All models are based on a linear model with transformed features, and thus have a linear model.
- `copy`: This returns a new copy of the model.
"""
```

Main.workspace2.AbstractModel

```
"""
    AbstractModel

Used as the root for all models in this notebook. We provide a helper `predict` function for `AbstractVectors` which transposes the features to a row vector. We also provide a default `update_transform!` which does nothing.
"""
abstract type AbstractModel end
```

predict (generic function with 5 methods)

```
predict(alm::AbstractModel, x::AbstractVector) = predict(alm, x')[1]
```

update_transform! (generic function with 1 method)

```
update_transform!(AbstractModel, args...) = nothing
```

# Linear Model

As before, we define a linear model as a linear map

$$f(x) = \hat{y} = \mathbf{w}^\top x$$

or with a data matrix $X$ of size `(samples, features)`

$$f(X) = \hat{Y} = X\mathbf{w}$$

To make the predict function simpler we provide a convenience predict function for all abstract models which transforms a `Vector` (which in julia is always a column vector), to a row vector (or a 1xn matrix). So you can call `predict(model, rand(10))` without worrying about whether `x` is a column or row vector. You will still need to pay attention to this when implementing future code.

```
md"
#### Linear Model

As before, we define a linear model as a linear map
```math
f(x) = \hat{y} = \mathbf{w}^\top x
```

or with a data matrix $$X$$ of size `(samples, features)`

```math
f(X) = \hat{Y} = X \mathbf{w}
```

To make the predict function simpler we provide a convenience predict function for all abstract models which transforms a `Vector` (which in julia is always a column vector), to a row vector (or a 1xn matrix). So you can call `predict(model, rand(10))` without worrying about whether `x` is a column or row vector. You will still need to pay attention to this when implementing future code.

"
```

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

```
begin
    struct LinearModel <: AbstractModel
        W::Matrix{Float64} # Aliased to Array{Float64, 2}
    end

    LinearModel(in, out=1) =
        LinearModel(zeros(in, out)) # feture size × output size

    Base.copy(lm::LinearModel) = LinearModel(copy(lm.W))
    predict(lm::LinearModel, X::AbstractMatrix) = X * lm.W
    get_features(m::LinearModel, x) = x

end
```

# (a) Polynomial Features ✅

```
begin

    _check_Poly2 = let
        pm = Polynomial2Model(2, 1)
        rng = Random.MersenneTwister(1)
        X = rand(rng, 3, 2)
        Φ = get_features(pm, X)
        Φ_true = [
            1.0 0.23603334566204692 0.00790928339056074 0.05571174026441932 0.0018668546204633095 6.25567637522e-5;
            1.0 0.34651701419196046 0.4886128300795012 0.12007404112451132 0.16931265897503248 0.2387424977182995;
            1.0 0.3127069683360675 0.21096820215853596 0.09778564804593431 0.06597122691230639 0.04450758232200489]
        check_1 = all(Φ .≈ Φ_true)
        pm = Polynomial2Model(2, 1; ignore_first=true)
        X_bias = ones(size(X, 1), size(X, 2) + 1)
        X_bias[:, 2:end] .= X
        Φ = get_features(pm, X_bias)
        check_2 = all(Φ .≈ Φ_true)
        check_1 && check_2
    end

    HTML("<h4 id=poly> (a) Polynomial Features $(_check_complete(_check_Poly2)) </h4>")
end
```

Now, we will implement Polynomial Model which basically uses the linear model with non-linear features. To transform features, we apply polynomial transformation to our data.

To acheive polynomial fit of degree $p$, we will have a non-linear map of features

$$f(x) = \sum_{j=0}^{p} w_j x^j$$

which we can write as a basis function:

$$f(x) = \sum_{j=0}^{p} w_j \phi_j(x) = \mathbf{w}^\top \Phi$$

where $\phi_j(x) = x^j$ so we simply apply this transformation to every data point $x_i$ to get the new dataset $\{(\phi(x_i), y_i)\}$.

Implement polynomial features transformation by constructing $\Phi$ with $p = 2$ degrees in the function `get_features`.

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

```julia
begin
    struct Polynomial2Model <: AbstractModel
        model::LinearModel
        ignore_first::Bool
    end
    Polynomial2Model(in, out=1; ignore_first=false) = if ignore_first
        in = in - 1
        Polynomial2Model(LinearModel(1 + in + Int(in*(in+1)/2), out), ignore_first)
    else
        Polynomial2Model(LinearModel(1 + in + Int(in*(in+1)/2), out), ignore_first)
    end
    Base.copy(lm::Polynomial2Model) = Polynomial2Model(copy(lm.model),
lm.ignore_first)
    get_linear_model(lm::Polynomial2Model) = lm.model

end
```

```
predict (generic function with 7 methods)
```

```julia
predict(lm::Polynomial2Model, X) = predict(lm.model, get_features(lm, X))
```

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

```julia
function get_features(pm::Polynomial2Model, _X::AbstractMatrix)

    # If _X already has a bias remove it.
    X = if pm.ignore_first
        _X[:, 2:end]
    else
        _X
    end

    m = size(X, 2)
    N = size(X, 1)
    num_features = 1 + # Bias bit
                   m + # p = 1
                   Int(m*(m+1)/2) # combinations (i.e. x_i*x_j)

    Φ = zeros(N, num_features)

    # Construct Φ
    #### BEGIN SOLUTION
    for i in 1:N
        for j in 1:1
            Φ[i,j] = 1
        end
        for j in 1:m
            Φ[i,j+1] = X[i,j]
        end
        permutations = []
        for k in 1:m+1
            append!(permutations,0)
        end
        for a in 1:m
            for b in a:m
                append!(permutations,X[i,a]*X[i,b])
            end
        end

        for j in m+2:num_features
            Φ[i,j] = permutations[j]
        end
    end

    #### END SOLUTION

    Φ
end
```

# (b) Mini-batch Gradient Descent ✅

```julia
begin
    __check_MBGD = let

        lm = LinearModel(3, 1)
        opt = _LR()
        lf = _LF()
        X = ones(10, 3)
        Y = collect(0.0:0.1:0.9)
        mbgd = MiniBatchGD(5)
        epoch!(mbgd, lm, lf, opt, X, Y)
        all(lm.W .== -10.0)
    end
    str = "<h2 id=graddescent> (b) Mini-batch Gradient Descent
$(_check_complete(__check_MBGD)) </h2>"
    HTML(str)
end
```

```julia
begin
    struct _LR <: Optimizer end
    struct _LF <: LossFunction end
    function gradient(lm::LinearModel, lf::_LF, X::Matrix, Y::Vector)
        sum(X, dims=1)
    end
    function update!(lm::LinearModel,
                     lf::_LF,
                     opt::_LR,
                     x::Matrix,
                     y::Vector)

        ϕ = get_features(lm, x)

        ΔW = gradient(lm, lf, ϕ, y)[1, :]
        lm.W .-= ΔW
    end
end;
```

```julia
struct MiniBatchGD
    n::Int
end
```

Gradient descent is another strategy for learning weights of a model. Instead of creating a closed form solution (like OLS) we learn iteratively following the gradient of the loss/cost function. When our data needs to be represented in more complex forms, we often will use some variant of gradient descent to learn complex parameterizations. Gradient Descent also doesn't require the $X^TX$ to be invertable to find a solution.

In this notebook we will be focusing on minibatch gradient descent, and using 3 learning rate adaptation rules `ConstantLR`, `HeuristicLR`, and `AdaGrad`. All of these have their use in various parts of the literature and in various settings.

Below you need to implement the function `epoch!` which goes through the data set in minibatches of size `mbgd.n`. Remember to randomize how you go through the data **and** that you are using the correct targets for the data passed to the learning update. In this implementation you will use

```julia
update!(model, lossfunc, opt, X_batch, Y_batch)
```

to update your model. So you will basically randomize and divide the dataset into batches and call the update function for each batch. These functions are defined in the section on **optimizers**.

```
md"""

Gradient descent is another strategy for learning weights of a model. Instead of creating a closed form solution (like OLS) we learn iteratively following the gradient of the loss/cost function. When our data needs to be represented in more complex forms, we often will use some variant of gradient descent to learn complex parameterizations. Gradient Descent also doesn't require the `X^TX` to be invertable to find a solution.

In this notebook we will be focusing on minibatch gradient descent, and using 3 learning rate adaptation rules `ConstantLR`, `HeuristicLR`, and `AdaGrad`. All of these have their use in various parts of the literature and in various settings.

Below you need to implement the function `epoch!` which goes through the data set in minibatches of size `mbgd.n`. Remember to randomize how you go through the data **and** that you are using the correct targets for the data passed to the learning update. In this implementation you will use

```julia
update!(model, lossfunc, opt, X_batch, Y_batch)
```

to update your model. So you will basically randomize and divide the dataset into batches and call the update function for each batch. These functions are defined in the section on [optimizers](#opt).

"""
```

epoch! (generic function with 3 methods)

```
function epoch!(mbgd::MiniBatchGD, model::LinearModel, lossfunc, opt, X, Y)

    #### BEGIN SOLUTION
#   A=[X Y]
#   A = A[shuffle(1:end), :]

#   for i in 1:mbgd.n:size(A, 1)
#       first_row = i
#       last_row = i + mbgd.n - 1
#       if last_row>size(A,1)
#           batch_i_matrix = A[first_row:last_row-1,:]
#       else
#           batch_i_matrix = A[first_row:last_row,:]
#       end
#       columns(batch_i_matrix) = [ batch_i_matrix[:,i] for i in 1:size(batch_i_matrix, 2)]
#       X_batch = columns(batch_i_matrix)[1]
#       if length(columns(batch_i_matrix))>1
#           for i=2:length(columns(batch_i_matrix))-1
#               X_batch=[X_batch columns(batch_i_matrix)[i]]
#           end
#       end
#       Y_batch=columns(batch_i_matrix)[size(batch_i_matrix,2)]
#       update!(model, lossfunc, opt, X_batch, Y_batch)

    match=randperm(length(Y))
    b=1
    minibatch=length(Y)/mbgd.n
    for i in 1:minibatch
        X_batch=zeros(mbgd.n,size(X,2))
        Y_batch=zeros(mbgd.n)
        k=1
        for j in match[b:b+mbgd.n-1]
            X_batch[k, :]=X[j, :]
            Y_batch[k]=Y[j]
            k+=1
        end
        update!(model, lossfunc, opt, X_batch, Y_batch)
        b+=mbgd.n

    end
    #### END SOLUTION
end
```

epoch! (generic function with 2 methods)

```
function epoch!(mbgd::MiniBatchGD, model::AbstractModel, lossfunc, opt, X, Y)
    epoch!(mbgd, get_linear_model(model), lossfunc, opt, get_features(lp.model, X), Y)
end
```

train! (generic function with 6 methods)

```
function train!(mbgd::MiniBatchGD, model::AbstractModel, lossfunc, opt, X, Y, num_epochs)
    train!(mbgd, get_linear_model(model), lossfunc, opt, get_features(model, X), Y, num_epochs)
end
```

train! (generic function with 8 methods)

```
function train!(mbgd::MiniBatchGD, model::LinearModel, lossfunc, opt, X, Y, num_epochs)
    ℒ = zeros(num_epochs + 1)
    ℒ[1] = loss(model, lossfunc, X, Y)
    for i in 1:num_epochs
        epoch!(mbgd, model, lossfunc, opt, X, Y)
        ℒ[i+1] = loss(model, lossfunc, X, Y)
    end
    ℒ
end
```

# Loss Functions ✅

```
HTML("<h3 id=lossfunc> Loss Functions $(_check_complete(__check_MSE)) </h3>")
```

For this notebook we will only be using MSE, but we still introduce the abstract type LossFunction for the future. Below you will need to implement the `loss` ✅ function and the `gradient` ✅ function for MSE.

```
begin
    # __check_mseGrad
    __check_mseloss = loss(LinearModel(3, 1), MSE(), ones(4, 3), [1,2,3,4]) == 3.75
    __check_msegrad = all(gradient(LinearModel(3, 1), MSE(), ones(4, 3), [1,2,3,4]) .== -2.5)

    __check_MSE = __check_mseloss && __check_msegrad

md"""
For this notebook we will only be using MSE, but we still introduce the abstract type LossFunction for the future. Below you will need to implement the `loss` $(_check_complete(__check_mseloss)) function and the `gradient` $(_check_complete(__check_msegrad)) function for MSE.
"""
end
```

```
abstract type LossFunction end
```

## (c) Mean Squared Error

```
md"""
#### (c) Mean Squared Error
"""
```

We will be implementing 1/2 MSE in the loss function.

$$c(w) = \frac{1}{2n} \sum_i^n (f(x_i) - y_i)^2$$

where $f(x)$ is the prediction from the passed model.

```
md"""
We will be implementing 1/2 MSE in the loss function.

```math
c(w) = \frac{1}{2n} \sum_i^n (f(x_i) - y_i)^2
```

where $f(x)$ is the prediction from the passed model.
"""
```

```
struct MSE <: LossFunction end
```

loss (generic function with 1 method)

```
function loss(lm::AbstractModel, mse::MSE, X, Y)
    0.0
    #### BEGIN SOLUTION
    s=0
    for i in 1:size(X,1)
        s+=(predict(lm,X[i,:])-Y[i])^2
    end
    mse=s/(2*size(X,1))

    #### END SOLUTION
end
```

## (d) Gradient of Mean Squared Error

You will implement the gradient of the MSE loss function `c(w)` in the `gradient` function with respect to `w`, returning a matrix of the same size of `lm.W`.

```
md"""
You will implement the gradient of the MSE loss function `c(w)` in the `gradient`
function with respect to `w`, returning a matrix of the same size of `lm.W`.
"""
```

gradient (generic function with 2 methods)

```
function gradient(lm::AbstractModel, mse::MSE, X::Matrix, Y::Vector)
    ∇W = zero(lm.W) # gradients should be the size of the weights

    #### BEGIN SOLUTION
    for i in 1:length(∇W)
        for j in 1:length(predict(lm,X))
            ∇W[i] +=(predict(lm,X[j,:])- Y[j])*X[j,i]
        end
    end
    ∇W=∇W/size(X,1)

    #### END SOLUTION

    @assert size(∇W) == size(lm.W)
    ∇W
end
```

# Optimizers ✅

```
HTML("<h3 id=opt> Optimizers $(_check_complete(_check_ConstantLR &&
__check_AdaGrad)) </h3>")
```

Below you will need to implement three optimizers

- Constant learning rate ✅
- Heuristic learning rate ✅
- AdaGrad ✅

```
md"""
Below you will need to implement three optimizers

- Constant learning rate $(_check_complete(_check_ConstantLR))
- Heuristic learning rate $(_check_complete(_check_HeuristicLR))
- AdaGrad $(_check_complete(__check_AdaGrad))
"""
```

```
abstract type Optimizer end
```

# (e) Constant Learning Rate ✅

To update the weights for mini-batch gradient descent, we can use `ConstantLR` optimizer which updates the weights using a constant learning rate `η`

$$W = W - \eta * g$$

where `g` is the gradient defined by the loss function.

Implement the `ConstantLR` optimizer.

```
begin
    _check_ConstantLR = let
        lm = LinearModel(3, 1)
        opt = ConstantLR(0.1)
        lf = MSE()
        X = ones(4, 3)
        Y = [0.1, 0.2, 0.3, 0.4]
        update!(lm, lf, opt, X, Y)
        all(lm.W .== 0.025)
    end
    md"""
    #### (e) Constant Learning Rate $(_check_complete(_check_ConstantLR))

    To update the weights for mini-batch gradient descent, we can use `ConstantLR` optimizer which updates the weights using a constant learning rate `η`

    ```math
    W = W - η*g
    ```

    where `g` is the gradient defined by the loss function.

    Implement the `ConstantLR` optimizer.
    """
end
```

```
struct ConstantLR <: Optimizer
    η::Float64
end
```

```
Base.copy(clr::ConstantLR) = ConstantLR(clr.η)
```

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

```julia
function update!(lm::LinearModel,
                 lf::LossFunction,
                 opt::ConstantLR,
                 x::Matrix,
                 y::Vector)

    g = gradient(lm, lf, x, y)

    #### BEGIN SOLUTION
    for i in 1:size(lm.W,1)
        lm.W[i]-=opt.η * g[i]
    end

    #### END SOLUTION
end
```

# (f) Heuristic Learning Rate ✅

To update the weights for mini-batch gradient descent, we can use `HeuristicLR` optimizer which updates the weights using a learning rate `η` that is a function of the gradient. We define the learning rate at time $t$ as:

$$\eta_t = (1 + \bar{g}_t)^{-1}$$

where $\bar{g}_t$ is an accumulating gradient over time that uses the gradient `g` defined by the loss function. We use the following to compute $\bar{g}_t$

$$\bar{g}_t = \bar{g}_{t-1} + \frac{1}{d+1} \sum_{j=0}^{d} |g_{t,j}|$$

Then, we use the update

$$W_t = W_t - \eta_t g_t$$

Implement the `HeuristicLR` by implementing the adaptive learning rate and update rule.

```
begin
    _check_HeuristicLR = let
        lm = LinearModel(3, 1)
        opt = HeuristicLR()
        lf = MSE()
        X = ones(4, 3)
        Y = [0.1, 0.2, 0.3, 0.4]
        update!(lm, lf, opt, X, Y)
        println(lm.W)
        all(lm.W .≈ 0.11111111111111)
    end
    md"""
    #### (f) Heuristic Learning Rate $(_check_complete(_check_HeuristicLR))

    To update the weights for mini-batch gradient descent, we can use `HeuristicLR` optimizer which updates the weights using a learning rate `η` that is a function of the gradient. We define the learning rate at time $t$ as:

    ```math
    \eta_t = (1 + \bar{g}_{t})^{-1}
    ```
    where $\bar{g}_{t}$ is an accumulating gradient over time that uses the gradient ```g``` defined by the loss function. We use the following to compute $\bar{g}_{t}$

    ```math
    \bar{g}_{t} = \bar{g}_{t-1} + \frac{1}{d+1} \sum_{j=0}^d |g_{t, j}|
    ```

    Then, we use the update

    ```math
    W_t = W_t - \eta_t g_t
    ```
    Implement the `HeuristicLR` by implementing the adaptive learning rate and update rule.
    """

end
```

HeuristicLR

```
begin
    mutable struct HeuristicLR <: Optimizer
        g_bar::Float64
    end
    HeuristicLR() = HeuristicLR(1.0)
end
```

```
Base.copy(hlr::HeuristicLR) = HeuristicLR(hlr.g_bar)
```

update! (generic function with 3 methods)

```
function update!(lm::LinearModel,
                 lf::LossFunction,
                 opt::HeuristicLR,
                 x::Matrix,
                 y::Vector)

    g = gradient(lm, lf, x, y)

    #### BEGIN SOLUTION

    s=0
    for i=1:length(g)
        s+=abs(g[i])
    end
    opt.g_bar+=s/(length(g))
    for i in 1:size(lm.W,1)
        lm.W[i]-=g[i]/(1+opt.g_bar)
    end
    #### END SOLUTION
end
```

# (g) AdaGrad ✅

```
begin
     __check_AdaGrad_v, __check_AdaGrad_W = let
        lm = LinearModel(2, 1)
        opt = AdaGrad(0.1, lm)
        X = [0.1 0.5;
             0.5 0.0;
             1.0 0.2]
        Y = [1, 2, 3]
        update!(lm, MSE(), opt, X, Y)
        true_G = [1.8677777777777768, 0.13444444444444445]
        true_W = [0.09999973230327601, 0.099996281199188]
        all(opt.G .≈ true_G), all(lm.W .≈ true_W)
    end

    __check_AdaGrad = __check_AdaGrad_v && __check_AdaGrad_W

md"""
#### (g) AdaGrad $(_check_complete(__check_AdaGrad))


"""
end
```

AdaGrad is another technique for adapting the learning rate where we use a different learning rate for every parameter $W_i$

To implement AdaGrad optimizer, we use the following equations:

$$\begin{aligned} G_i &= G_i + g_i^2 \\ W_i &= W_i - \frac{\eta}{\sqrt{G_i + \epsilon}} * g_i \end{aligned}$$

where $g$ is the gradient, and $W$ are the weights.

Implement `AdaGrad`.

```
md"""
AdaGrad is another technique for adapting the learning rate where we use a different
learning rate for every parameter $W_i$

To implement AdaGrad optimizer, we use the following equations:

```math
\begin{align}
G_i &= G_i + g_i^2 \\
W_i &= W_i - \frac{\eta}{\sqrt{G_i + \epsilon}} * g_i
\end{align}
```
where $g$ is the gradient, and $W$ are the weights.

Implement ```AdaGrad```.
"""
```

```
begin
    mutable struct AdaGrad <: Optimizer
        η::Float64 # step size
        G::Matrix{Float64} # exponential decaying average
        ϵ::Float64 #
    end

    AdaGrad(η) = AdaGrad(η, zeros(1, 1), 1e-5)
    AdaGrad(η, lm::LinearModel) = AdaGrad(η, zero(lm.W), 1e-5)
    AdaGrad(η, lm::AbstractModel) = AdaGrad(η, get_linear_model(model))
    Base.copy(adagrad::AdaGrad) = AdaGrad(adagrad.η, zero(adagrad.G), adagrad.ϵ)
end
```

update! (generic function with 4 methods)

```
function update!(lm::LinearModel,
                 lf::LossFunction,
                 opt::AdaGrad,
                 x::Matrix,
                 y::Vector)

    g = gradient(lm, lf, x, y)
    if size(g) !== size(opt.G) # need to make sure this is of the right shape.
        opt.G = zero(g)
    end

    # update opt.v and lm.W
    η, G, ϵ = opt.η, opt.G, opt.ϵ

    #### BEGIN SOLUTION
    opt.G+=g.^2
    for i in 1:size(lm.W,1)
        lm.W[i]-=g[i]*opt.η/sqrt(opt.G[i]+opt.ϵ)
    end

    #### END SOLUTION

end
```

# Evaluating models

In the following section, we provide a few helper functions and structs to make evaluating methods straightforward. The abstract type `LearningProblem` with children `GDLearningProblem` and `OLSLearningProblem` are used to construct a learning problem. You will notice these structs contain all the information needed to `train!` a model for both gradient descent and for OLS. We also provide the `run` and `run!` functions. These will update the transform according to the provided data and train the model. `run` does this with a copy of the learning problem, while `run!` does this inplace.

```
md"""
# Evaluating models

In the following section, we provide a few helper functions and structs to make
evaluating methods straightforward. The abstract type `LearningProblem` with
children `GDLearningProblem` and `OLSLearningProblem` are used to construct a
learning problem. You will notice these structs contain all the information needed
to `train!` a model for both gradient descent and for OLS. We also provide the `run`
and `run!` functions. These will update the transform according to the provided data
and train the model. `run` does this with a copy of the learning problem, while
`run!` does this inplace.

"""
```

```
abstract type LearningProblem end
```

Main.workspace2.GDLearningProblem

```
"""
    GDLearningProblem

This is a struct for keeping a the necessary gradient descent learning setting components together.
"""
struct GDLearningProblem{M<:AbstractModel, O<:Optimizer, LF<:LossFunction} <: LearningProblem
    gd::MiniBatchGD
    model::M
    opt::O
    loss::LF
end
```

```
Base.copy(lp::GDLearningProblem) =
    GDLearningProblem(lp.gd, copy(lp.model), copy(lp.opt), lp.loss)
```

run! (generic function with 1 method)

```
function run!(lp::GDLearningProblem, X, Y, num_epochs)
    update_transform!(lp.model, X, Y)
    train!(lp.gd, lp.model, lp.loss, lp.opt, X, Y, num_epochs)
end
```

run (generic function with 1 method)

```
function run(lp::LearningProblem, args...)
    cp_lp = copy(lp)
    ℒ = run!(cp_lp, args...)
    return cp_lp, ℒ
end
```

# Run Experiment

```
HTML("<h4 id=cv> Run Experiment </h2>")
```

Below are the helper functions for running an experiment.

Main.workspace7.run_experiment

```
"""
    run_experiment(lp, X, Y, num_epochs, runs; train_size)

Using `train!` do `runs` experiments with the same train and test split (which is made by `random_dataset_split`). This will create a copy of the learning problem and use this new copy to train. It will return the estimate of the error.
"""
function run_experiment(lp::LearningProblem,
                        train_data,
                        test_data,
                        num_epochs,
                        runs)

    err = zeros(runs)

    for i in 1:runs
        # train
        cp_lp, train_loss = run(lp, train_data[1], train_data[2], num_epochs)

        # test
        Ŷ = predict(cp_lp.model, test_data[1])
        err[i] = sqrt(mean(abs2, test_data[2] - Ŷ))
    end

    err
end
```

# Experiments

In this section, we will run three experiments on the different algorithms we implemented above. We provide the data in the `Data` section, and then follow with the three experiments and their descriptions. You will need to analyze and understand the three experiments for the written portion of this assignment.

```
md"
# Experiments

In this section, we will run three experiments on the different algorithms we implemented above. We provide the data in the `Data` section, and then follow with the three experiments and their descriptions. You will need to analyze and understand the three experiments for the written portion of this assignment.
"
```

## Data

This section creates the datasets we will use in our comparisons. Feel free to play with them in `let` blocks.

```
md"""
## Data

This section creates the datasets we will use in our comparisons. Feel free to play with them in `let` blocks.
"""
```

Main.workspace2.splitdataframe

```
"""
    splitdataframe(split_to_X_Y::Function, df::DataFrame, test_perc; shuffle = false)
    splitdataframe(df::DataFrame, test_perc; shuffle = false)

Splits a dataframe into test and train sets. Optionally takes a function as the first parameter to split the dataframe into X and Y components for training. This defaults to the `identity` function.
"""
function splitdataframe(split_to_X_Y::Function, df::DataFrame, test_perc;
                        shuffle = false)
    #= shuffle dataframe.
    This is innefficient as it makes an entire new dataframe,
    but fine for the small dataset we have in this notebook.
    Consider shuffling inplace before calling this function.
    =#

    df_shuffle = if shuffle == true
        df[randperm(nrow(df)), :]
    else
        df
    end

    # Get train size with percentage of test data.
    train_size = Int(round(size(df,1) * (1 - test_perc)))

    dftrain = df_shuffle[1:train_size, :]
    dftest = df_shuffle[(train_size+1):end, :]

    split_to_X_Y(dftrain), split_to_X_Y(dftest)
end
```

unit_normalize_columns! (generic function with 1 method)

```
function unit_normalize_columns!(df::DataFrame)
    for name in names(df)
        mn, mx = minimum(df[!, name]), maximum(df[!, name])
        df[!, name] .= (df[!, name] .- mn) ./ (mx - mn)
    end
    df
end
```

# Admissions Dataset

```
md"""
### **Admissions Dataset**
"""
```

```
admissions_data = let
    data = CSV.read("data/admission.csv", DataFrame, delim=',', ignorerepeated=true)[:, 2:end]
    data[!, 1:end-1] = unit_normalize_columns!(data[:, 1:end-1])
    data
end;
```

# Plotting our data

The `plot_data` function produces two plots that can be displayed horizontally or vertically. The left or top plot is a box plot over the cv errors, the right or bottom plot is a bar graph displaying average cv errors with standard error bars. This function will be used for all the experiments, and you should use this to finish your written experiments.

```
md"""
## Plotting our data

The `plot_data` function produces two plots that can be displayed horizontally or vertically. The left or top plot is a box plot over the cv errors, the right or bottom plot is a bar graph displaying average cv errors with standard error bars. This function will be used for all the experiments, and you should use this to finish your written experiments.

"""

```

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

```
function plot_data(algs, errs; vert=false)
    stderr(x) = sqrt(var(x)/length(x))

    plt1 = boxplot(reshape(algs, 1, :),
                   errs,
                   legend=false, ylabel="MSE",
                   pallette=:seaborn_colorblind)

    plt2 = bar(reshape(algs, 1, :),
               reshape(mean.(errs), 1, :),
               yerr=reshape(stderr.(errs), 1, :),
               legend=false,
               pallette=:seaborn_colorblind,
               ylabel=vert ? "MSE" : "")

    if vert
        plot(plt1, plt2, layout=(2, 1), size=(600, 600))
    else
        plot(plt1, plt2)
    end
end
```

# (h) Non-linear feature transforms

We will compare the linear to non-linear models using the a simulated data set and the admissions dataset.

To run these experiments use ☑

```
md"""
## (h) Non-linear feature transforms

We will compare the linear to non-linear models using the a simulated data set and the admissions dataset.

To run these experiments use $(@bind __run_nonlinear PlutoUI.CheckBox())
"""
```

This first expereiment uses a simulated training set which aims to predict this function

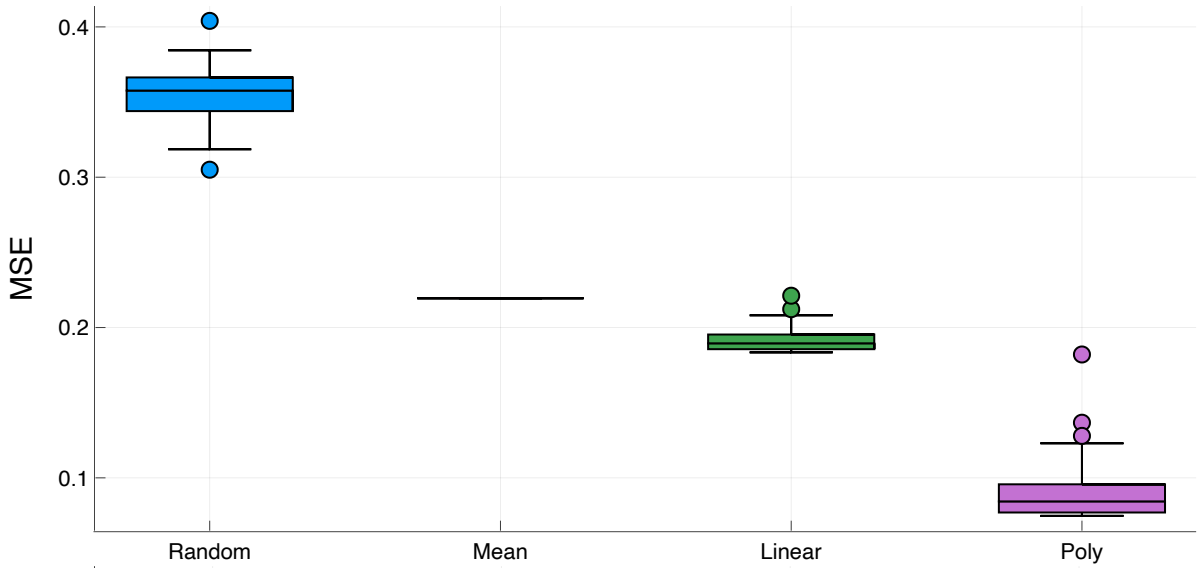
```julia
f(x) = sin(π*x[1]*x[2]^2) + cos(π*x[3]^3) + x[5]*sin(π*x[4]^4) + 0.001*randn()
```

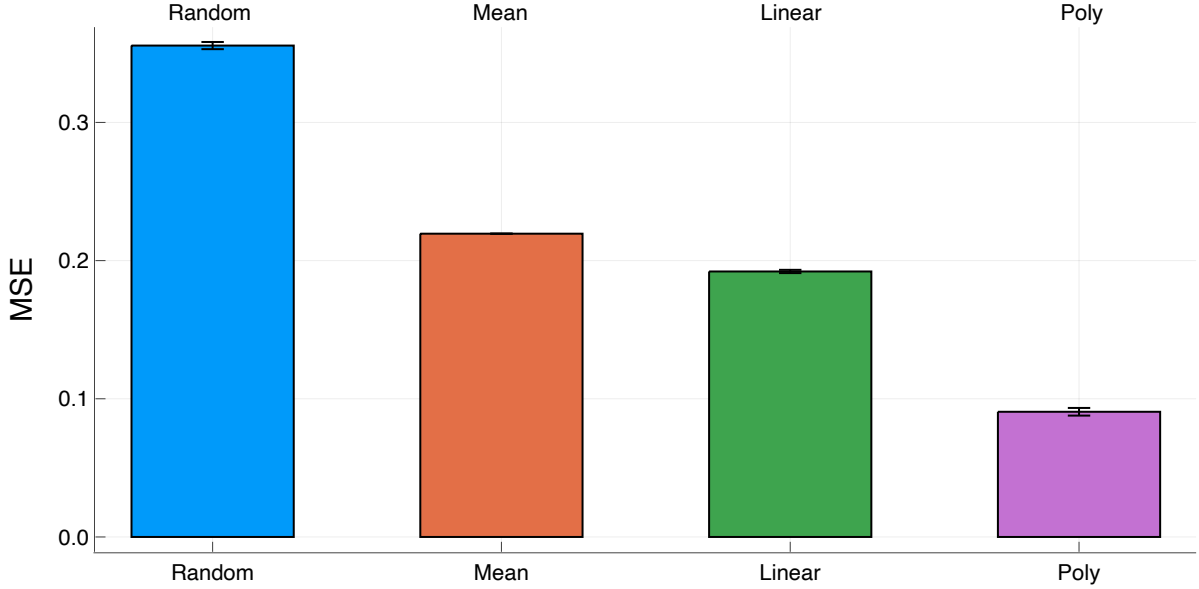
from inputs $\mathbf{x} \in [0.0, 1.0]^5$. We compare a linear representation and a Polynomial (p=2) representation with two baselines.

```
md"""
This first expereiment uses a simulated training set which aims to predict this
function

```julia
f(x) = sin(π*x[1]*x[2]^2) + cos(π*x[3]^3) + x[5]*sin(π*x[4]^4) + 0.001*randn()
```

from inputs $\mathbf{x} \in [0.0, 1.0]^5$. We compare a linear representation and a
Polynomial (p=2) representation with two baselines.
"""
```




```
let
    if __run_nonlinear
        algs = ["Random", "Mean", "Linear", "Poly"]
        non_linear_problems_sin = [
            GDLearningProblem(
                MiniBatchGD(30),
                RangeModel(),
                ConstantLR(0.0),
                MSE()),
            GDLearningProblem(
                MiniBatchGD(30),
                MeanModel(),
                ConstantLR(0.0),
                MSE()),
            GDLearningProblem(
                MiniBatchGD(30),
                LinearModel(5, 1),
                ConstantLR(1.0),
                MSE()),
            GDLearningProblem(
                MiniBatchGD(30),
                Polynomial2Model(5, 1),
                ConstantLR(0.5),
                MSE())
            ];
        nonlinear_errs_sin = let
            Random.seed!(2)
            X = rand(500, 5)
```

```
            f(x) = sin(π*x[1]*x[2]^2) + cos(π*x[3]^3) + x[5]*sin(π*x[4]^4) +
0.001*randn()
            Y = [f(x) for x in eachrow(X)]
            Y .= (Y.-minimum(Y))/(maximum(Y) - minimum(Y))
            plot(Y)
            errs = Vector{Float64}[]

            train_size=400

            rp = randperm(length(Y))
            train_idx = rp[1:train_size]
            test_idx = rp[train_size+1:end]
            train_data = (X[train_idx, :], Y[train_idx])
            test_data = (X[test_idx, :], Y[test_idx])

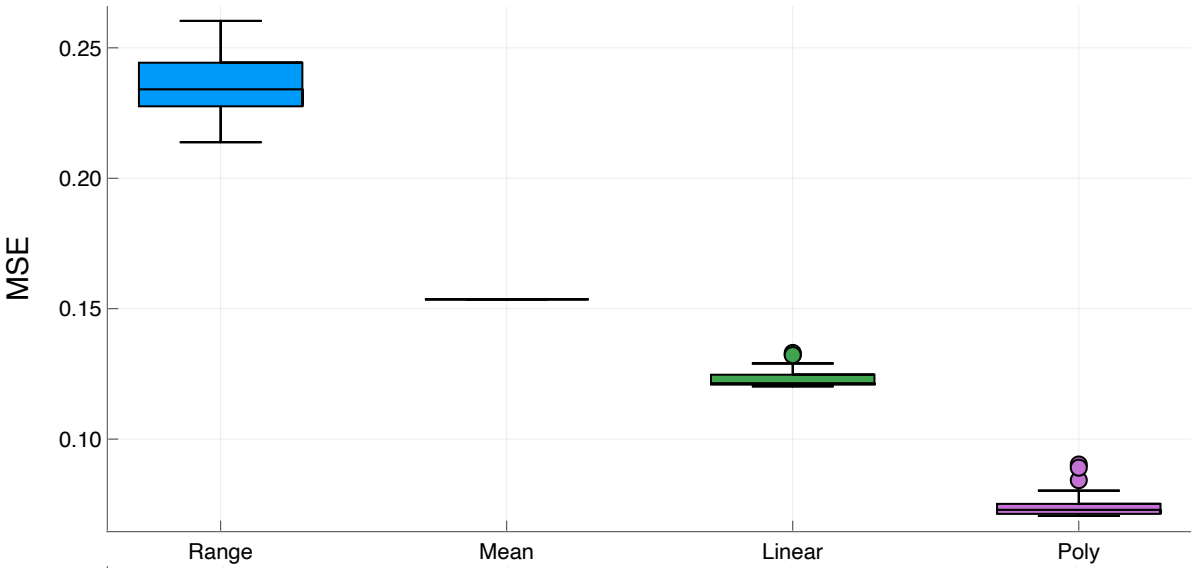
            for (idx, prblms) in enumerate(non_linear_problems_sin)
                cv_err = run_experiment(prblms, train_data, test_data, 10, 50)
                push!(errs, cv_err)
            end
            errs
        end

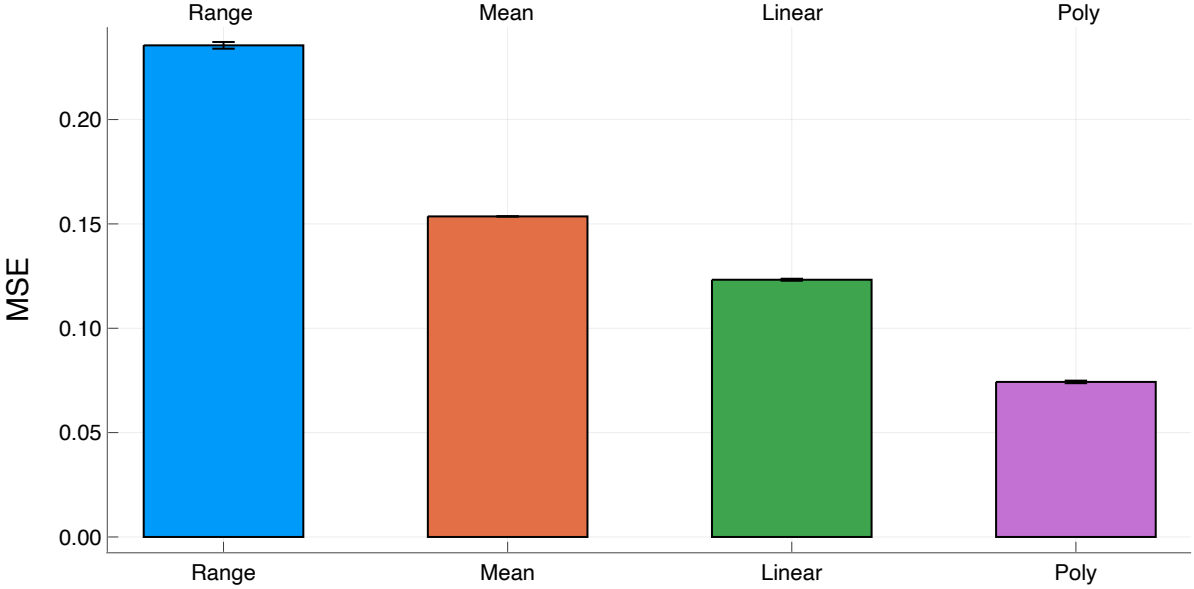
        plot_data(algs, nonlinear_errs_sin, vert=true)

    end
end
```

The following experiment uses the addmistions dataset, which you should report. **You can get the average error and standard error to report from the plot or from the terminal where your ran this notebook**.

```
md"""
The following experiment uses the addmistions dataset, which you should report.
**You can get the average error and standard error to report from the plot or from
the terminal where your ran this notebook**.
"""
```



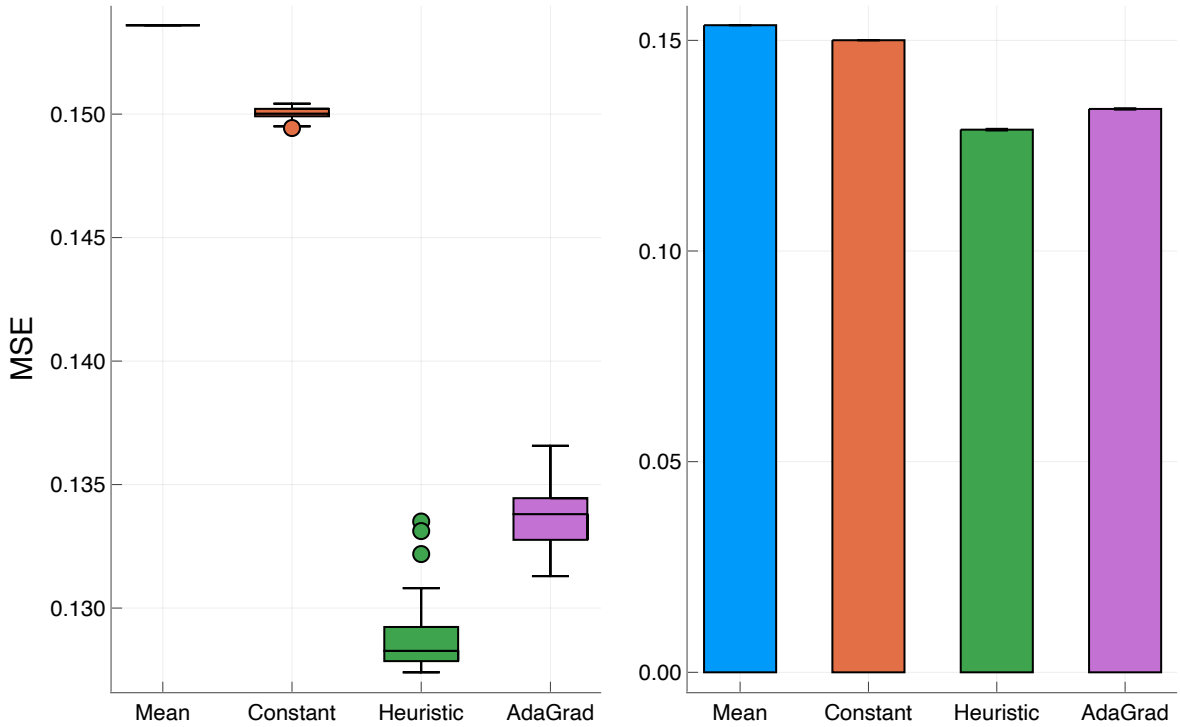




# (i) Learning Rate adapatation

We will compare the different learning rate algorithms on a subset of the **Admissions dataset**. From this dataset we will be predicting the likelihood of admission.

To run this experiment click ☑

**You can get the average error and standard error to report from the plot or from the terminal where your ran this notebook**.




```
let
    if __run_lra
        algs_lr = ["Mean", "Constant", "Heuristic", "AdaGrad"]
        lr_adapt_problems = [
            GDLearningProblem(
                MiniBatchGD(30),
                MeanModel(),
                ConstantLR(0.0),
                MSE()),
            GDLearningProblem(
                MiniBatchGD(30),
                LinearModel(7, 1),
                ConstantLR(0.05),
                MSE()),
            GDLearningProblem(
                MiniBatchGD(30),
                LinearModel(7, 1),
                HeuristicLR(),
                MSE()),
            GDLearningProblem(
                MiniBatchGD(30),
                LinearModel(7, 1),
                AdaGrad(0.1),
                MSE()),
        ];
        lr_errs = let

            Random.seed!(2)
            test_idx = 1
            data = (X=Matrix(admissions_data[:, 1:end-1]), Y=admissions_data[:,
end])
            @show size(data.X)
            errs = Vector{Float64}[]

            X, Y = data.X, data.Y
            train_size=350

            rp = randperm(length(Y))
            train_idx = rp[1:train_size]
            test_idx = rp[train_size+1:end]
            train_data = (X[train_idx, :], Y[train_idx])
            test_data = (X[test_idx, :], Y[test_idx])
```

```
            for (idx, prblms) in enumerate(lr_adapt_problems)

                err = run_experiment(prblms, train_data, test_data, 5, 50)
                push!(errs, err)
            end
            errs
        end
        num_runs = size(lr_errs[4])
        stderr(x) = sqrt(var(x)/length(x))

        mean_error_constantLR = mean(lr_errs[2])
        mean_error_HeuristicLR = mean(lr_errs[3])
        mean_error_AdaGrad = mean(lr_errs[4])

        std_error_constantLR = stderr(lr_errs[2])
        std_error_HeuristicLR = stderr(lr_errs[3])
        std_error_AdaGrad = stderr(lr_errs[4])

        println("Average error on test set for Linear model with ConstantLR is $mean_error_constantLR with standard error $std_error_constantLR")

        println("Average error on test set for Linear model with HeuristicLR is $mean_error_HeuristicLR with standard error $std_error_HeuristicLR")

        println("Average error on test set for Linear model with AdaGrad is $mean_error_AdaGrad with standard error $std_error_AdaGrad")

        plot_data(algs_lr, lr_errs)
    end

end
```