Setup

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

```
begin
    using CSV    , DataFrames    , Random
    using PlutoUI
    using PlotlyJS
    import Colors    : Colors, @colorant_str
    using LinearAlgebra    : dot, norm, norm1, norm2, I
    using Distributions    : Distributions, Uniform
    using Statistics
    using MultivariateStats    : MultivariateStats, PCA
    using StatsBase    : StatsBase
```

!!!IMPORTANT!!!

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

Welcome Sanad Masannat! ✅

```
let
    def_student = (name="NAME as in eclass", email="UofA Email", ccid="CCID",
idnumber=0)
    if length(keys(def_student) ∩ keys(student)) != length(keys(def_student))
        md"You don't have all the right entries! Make sure you have `name`, `emai
`ccid`, `idnumber`. ✖"
    elseif any(getfield(def_student, k) == getfield(student, k) for k in
keys(def_student))
        md"You haven't filled in all your details! ✖"
    elseif !all(typeof(getfield(def_student, k)) === typeof(getfield(student, k))
in keys(def_student))
        md"Your types seem to be off: `name::String`, `email::String`, `ccid::Str
`idnumber::Int`"
    else
        md"Welcome $(student.name)! ✓"
    end
```

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

```
student = (
    name="Sanad Masannat",
    email="sanad@ualberta.ca",
    ccid="sanad",
    idnumber=1626221
```

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

```
#### BEGIN SOLUTION


```

Preamble

In this assignment, we will implement:

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

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

Q2: Multi-variate Regression

In the last assignment you learned the weight for a simplistic univariate setting, to predict y from
Now we get to move to the multivariate setting! This means more than one input, which is a muc
more realistic problem setting.

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 alread
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

Mean Model

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

```
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
```

RandomModel

```
train! (generic function with 2 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
```

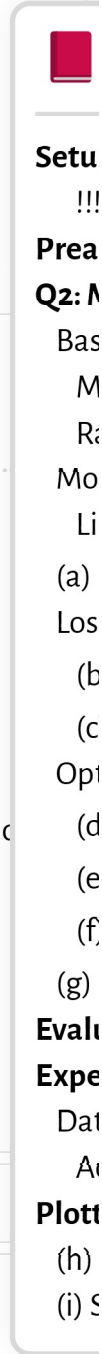
Models

- `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
```

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

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



Linear Model

A linear model is the linear function

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

giving us a prediction $\hat{y}$.

Note that to query this function on more than one sample, we can use the fact that $\mathbf{X}\mathbf{w}$ corresp to a vector where the first element is the dot product between the first row of $\mathbf{X}$ and $\mathbf{w}$, the sec element is the dot product between the second row of $\mathbf{X}$ and $\mathbf{w}$ and so on. We exploit this in `predict`, to return predictions for the data matrix $\mathbf{X}$ of size `(samples, features)`.

We define `get_features`, which we will need for polynomial regression. For linear regression, th default is to return the inputs themselves. In polynomial regression, we will replace this function with one that returns polynomial features.

```
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)) # feature 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

```

(a) ✅ Mini-batch Gradient Descent

```
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> (a) $(_check_complete(__check_MBGD)) Mini-batch Gr
Descent </h2>"
    HTML(str)
```

For this notebook we use minibatch gradient descent, and three stepsize approaches: `ConstantLR`, `HeuristicLR`, and `AdaGrad`. We provide a default update function below, that does gradient descent with a stepsize of 1.0. This update function will be defined for each of these three stepsize approaches later.

Notice that we use `.-=` which is the same as `lm.w = lm.w .- Δw`. The dot-minus means elementwise subtraction, for the vectors `lm.w` and `Δw`. In general, prefacing with a dot means elementwise operations: `a.*b` would mean elementwise product between vectors `a` and `b`, and `a./b` would mean elementwise division.

```
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
```

```
struct MiniBatchGD
    n::Int
```

First, you will set up the basic minibatch gradient descent code. You need to implement the function `epoch!` which goes through the data set in minibatches of size `mbgd.n`. Remember to shuffle the data for each epoch. In your code, you can call the function

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

to update your model in the epoch. Again, we will use different updates depending on the stepsi
rules, defined in the section below on **optimizers**.

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

```
function epoch!(mbgd::MiniBatchGD, model::LinearModel, lossfunc, opt, X, Y)
    perm=randperm(length(Y))
    X=X[perm,:]
    Y=Y[perm]

    batch_num=(length(Y))/Int(mbgd.n)
    for i in 1:batch_num
        low_range=Int((i-1)*mbgd.n +1)
        high_range=Int(i*mbgd.n)
        if high_range>length(Y)
            break
        end
        update!(model::LinearModel, lossfunc, opt,
X[low_range:high_range,:],Y[low_range:high_range])
    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
```

```
train! (generic function with 3 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),
num_epochs)
```

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

```
function train!(mbgd::MiniBatchGD, model::LinearModel, lossfunc, opt, X, Y, num_e
    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

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
    lm1 = LinearModel(3, 1)
    lm2 = LinearModel(3, 1)
    lm2.w .+= 1
    __check_mseloss = loss(lm1, MSE(), ones(4, 3), [1,2,3,4]) == 3.75 && loss(lm2
MSE(), ones(4, 3), [1,2,3,4]) == 0.75 && loss(lm2, MSE(), ones(4, 3), [7,8,9,0])
10.75
    __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
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.
"""
```

(b) ✅ Mean Squared Error

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

$$c(\mathbf{w}) = \frac{1}{2n}\sum_{i}^{n}(f(\mathbf{x}_i) - y_i)^2$$

where $f(\mathbf{x})$ is the prediction from the passed model.

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

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


    #### END SOLUTION
```

(c) ✅ 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`.

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

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

    #### BEGIN SOLUTION
    n=length(Y)
    XT=X'
    sum=zero(lm.w)
    for i in 1:n
        sum.+=((X[i,:].* lm.w) .- Y[i]) .* X[i,:]
    end
    sum./=n
    ∇w=sum
    #### END SOLUTION
    @assert size(∇w) == size(lm.w)
    ∇w
```

Optimizers

Below you will need to implement three optimizers

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

(d) ✅ Constant Learning Rate

To update the weights for mini-batch gradient descent, we can use `ConstantLR` optimizer which updates the weights using a constant stepsize `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"""
    ### (d) $(_check_complete(_check_ConstantLR)) Constant Learning Rate

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

    ```math
    w = w - η*g
    ```

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

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

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

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

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

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

    #### BEGIN SOLUTION
    lm.w.-=g.*opt.η
    #### END SOLUTION
```

(e) ✅ Heuristic Learning Rate

To update the weights for mini-batch gradient descent, we can use `HeuristicLR` optimizer which updates the weights using a stepsize `η` that is a function of the gradient. We define the stepsize 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 $\bar{g}_t$

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

Then, we use the update

$$w_t = w_{t-1} - \eta_t g_t$$

Implement the `HeuristicLR`.

```
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)
        all(lm.w .≈ 0.11111111111111)
    end
    md"""
    ### (e) $(_check_complete(_check_HeuristicLR)) Heuristic Learning Rate

    To update the weights for mini-batch gradient descent, we can use `HeuristicL
optimizer which updates the weights using a stepsize `η` that is a function of th
gradient. We define the stepsize 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 gradi
```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} \sum_{j=1}^d |g_{t, j}|
    ```

    Then, we use the update

    ```math
    w_{t} = w_{t-1} - \eta_t g_t
    ```
    Implement the `HeuristicLR`.
    """
```
```

```
HeuristicLR
```

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

```
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
    d= length(g)
    sum=0.0
    for i in 1:d
        sum+=abs(g[i])
    end
    sum/=d
    opt.g_bar+=sum
    η=(1+opt.g_bar)^(-1)
    update=g.*η
    (lm.w.-=update)
    #### END SOLUTION
```

(f) ✅ AdaGrad

AdaGrad is another technique for adapting the stepsize where we use a different stepsize for eve element j in the weight vector.

To implement the AdaGrad optimizer, we use the following equations for each j from 1 to the len of the weight vector:

$$\bar{g}_{t,j} = \bar{g}_{t-1,j} + g_j^2$$
$$w_{t,j} = w_{t-1,j} - \frac{\eta}{\sqrt{\bar{g}_{t,j} + \epsilon}} g_j$$

where g is the gradient and g_j is the jth element of the gradient. These equations can be implemented without using a for loop, by using elementwise multiplication and division. If you are stuck on the syntax, feel free to use a for loop. Note that to get the elementwise squaring of gradient g, you would use `g.^2` and to get elementwise sqrt you would use `sqrt.(g)`.

Implement `AdaGrad`.

```
begin
    mutable struct AdaGrad <: Optimizer
        η::Float64 # step size
        gbar::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.gbar), adagrad.
```

```
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.gbar) # need to make sure this is of the right shape.
        opt.gbar = zero(g)
    end

    # update opt.gbar and lm.w
    #### BEGIN SOLUTION
    n=length(lm.w)
    opt.gbar.+=(g.^2)
    lm.w.-= (opt.η./ sqrt.((opt.gbar .+ opt.ϵ) )) .* g
    #### END SOLUTION

end
```

(g) ✅ 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 = check_elements(Φ_true, Φ) #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 = check_elements(Φ_true, Φ)#all(Φ .≈ Φ_true)
        check_1 && check_2
    end

    HTML("<h2 id=poly> (g) $(_check_complete(_check_Poly2)) Polynomial Features <
end
```

Now, we will implement Polynomial Model which uses the linear model on non-linear features. T
do so, we apply a polynomial transformation to our data to create new polynomial features. For d
inputs with a polynomial of size p, the number of features is $m = \binom{d+p}{p}$, giving polynomial func

$$f(\mathbf{x}) = \sum_{j=1}^{m} w_j \phi_j(\mathbf{x}) = \boldsymbol{\phi}(\mathbf{x})^\top \mathbf{w}$$

We simply apply this transformation to every data point $\mathbf{x}_i$ to get the new dataset $\{(\boldsymbol{\phi}(\mathbf{x}_i), y_i)\}$

Implement the polynomial feature transformation for $p = 2$ degrees in the function `get_featur`

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

```
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_firs
    else
        Polynomial2Model(LinearModel(1 + in + Int(in*(in+1)/2), out), ignore_firs
    end
    Base.copy(lm::Polynomial2Model) = Polynomial2Model(copy(lm.model), lm.ignore_
    get_linear_model(lm::Polynomial2Model) = lm.model

end
```

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

```
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

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

    Φ = zeros(N, num_features)

    # Construct Φ
    #### BEGIN SOLUTION
    i1=zeros(N)
    i2=zeros(N)
    i3=zeros(N)
    i4=zeros(N)
    i5=zeros(N)
    i6=zeros(N)
    for i in 1:N
        i1[i]=1
        i2[i]=X[i,1]
        i3[i]=X[i,2]
        i4[i]=i2[i]*i3[i]
        i5[i]=i2[i]^2
        i6[i]=i3[i]^2
    end
    Φ[:,1]=i1
    Φ[:,2]=i2
    Φ[:,3]=i3
    Φ[:,4]=i4
    Φ[:,5]=i5
    Φ[:,6]=i6

    #### END SOLUTION

    Φ
```

Setu
!!!
Prea
Q2: M
Bas
M
R
Mo
Li
(a)
Los
(b
(c
Opt
(d
(e
(f
(g)
Evalu
Expe
Dat
A
Plott
(h)
(i) S

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. 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.

```
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
```

```
Base.copy(lp::GDLearningProblem) =
```

```
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)
```

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

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

Run Experiment

Below are the helper functions for running an experiment.

```
Main.workspace2.run_experiment
```

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.

```
Main.workspace2.splitdataframe
```

```
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
```

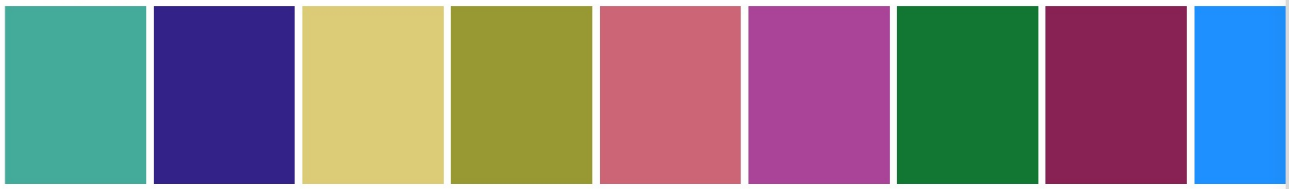
Admissions Dataset

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

Plotting Utilities

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

```
color_scheme =
```



```
Main.workspace2.plot_results
```

(h) Comparing Linear Regression and Polynomial Regression

We will compare the linear regression and polynomial regression with $p = 2$ using the a simulat data set and the admissions dataset.

To run these experiments use ☑

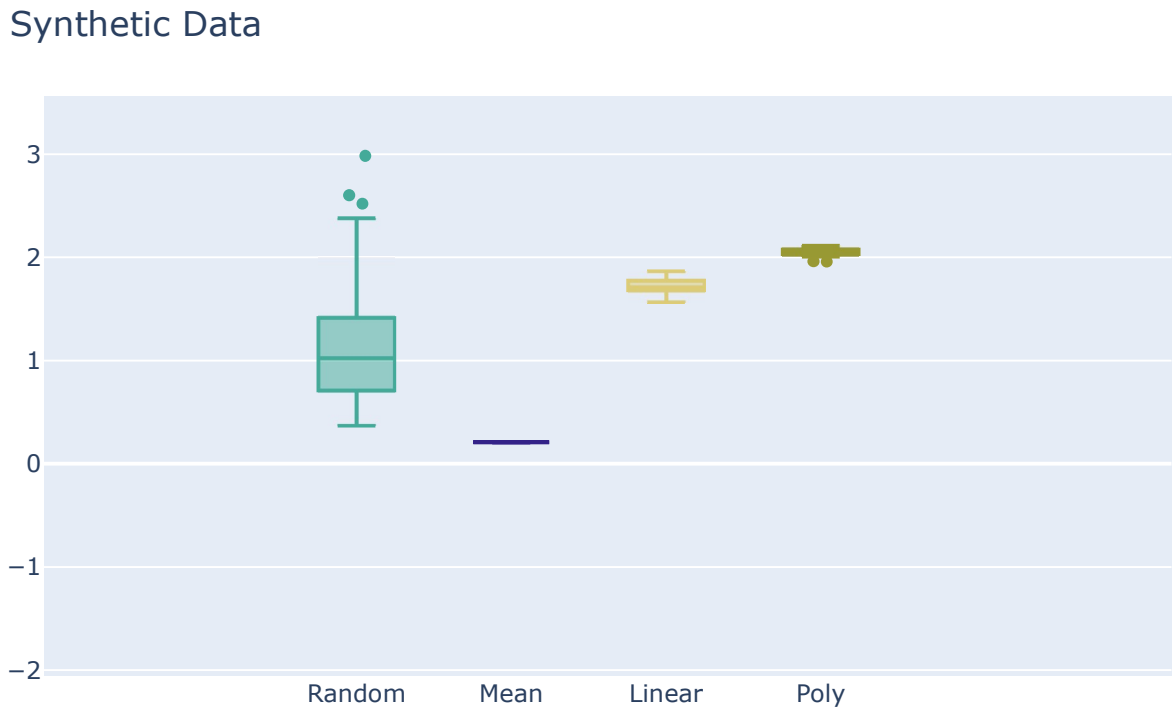
This first experiment uses a simulated training set. For a given input $\mathbf{x} \in [0.0, 1.0]^5$, the nonline function that defines $\mathbb{E}[Y|\mathbf{x}]$ is

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

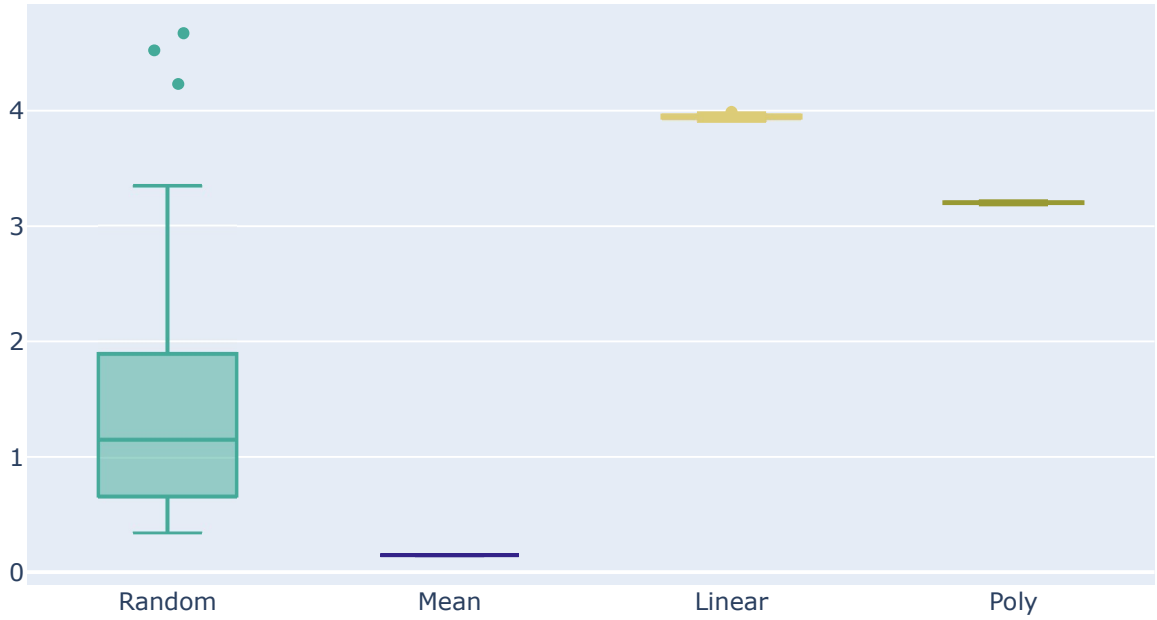
To get the target, we use

```
y = f(x) + 0.001*randn()
```

namely we add a small amount of Gaussian noise. We compare a linear regression, polynomial regression and two baselines.



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 below the plot and experiment code**.

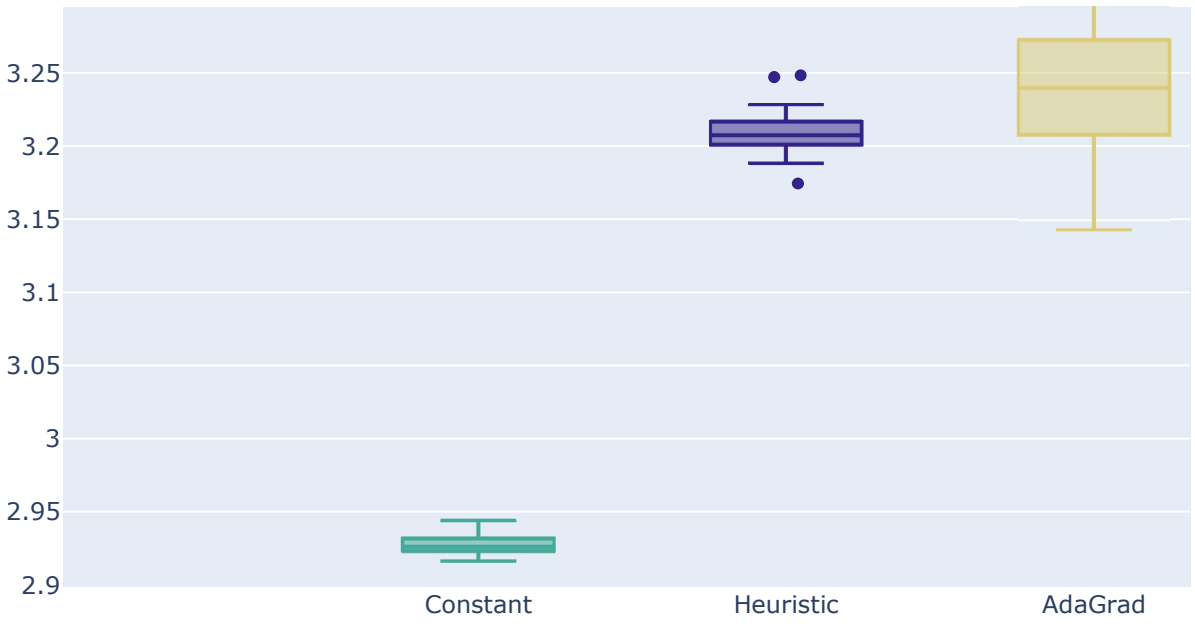


(i) Stepsize Adaptation

We will compare the different stepsize algorithms on a subset of the **Admissions dataset**. From t 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 or from below the plot and experiment code.



Setu
!!!
Prea
Q2: M
Bas
M
R
Mo
Li
(a)
Los
(b
(c
Opt
(d
(e
(f
(g)
Evalu
Expe
Dat
A
Plott
(h)
(i) S