Ctrl + S

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, TDist, cdf, Normal
    using MultivariateStats    : MultivariateStats, PCA
    using StatsBase    : StatsBase

end
```

A4 Outline


```
PlutoUI.TableOfContents(title="A4 Outline")
```

```
PlotlyBackend()
```

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

!!!IMPORTANT!!!

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

Welcome Nasif Hossain! ✅

```
student =
  (name = "Nasif Hossain", email = "nhossain@ualberta.ca", ccid = "nhossain", idnumber = 15
```

```
student = (name="Nasif Hossain", email="nhossain@ualberta.ca", ccid="nhossain",
idnumber=1545143)
```

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

```
#### BEGIN SOLUTION


#### END SOLUTION
```

Preamble

In this assignment, we will implement:

- Q1(a) **Logistic Regression: sigmoid function** ✅
- Q1(b) **Polynomial Logistic Regression** ✅
- Q1(c) **Mini-batch Gradient Descent** use the old code in Assignment 3 ✅
- Q1(d) **Loss Function** cross entropy ✅
- Q1(e) **Gradient of Loss Function** gradient of cross entropy ✅
- Q1(f) **Optimizer**: adapive stepsize RMSprop ✅
- Q2(a) Hypothesis-testing: Define Null hypothesis and alternative hypothesis
- Q2(b) **Checking for assumptions**: before running the t-test true
- Q2(c) **Running the t-test**: get the pvalue and run the t-test true

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

```
function check_elements(Φ_true::Matrix{Float64}, Φ::Matrix{Float64})
    all_exist = true
    for n in eachindex(Φ_true), m in eachindex(Φ_true[n])
        if Φ_true[n][m] ∉ Φ || Φ[n][m] ∉ Φ_true
            has_sim = false
            for i in eachindex(Φ_true), j in eachindex(Φ_true[i])
                if Φ_true[i][j] ≈ Φ[n][m]
                    has_sim = true
                end
            end
            true_has_sim = false
            for i in eachindex(Φ), j in eachindex(Φ[i])
                if Φ[i][j] ≈ Φ_true[n][m]
                    true_has_sim = true
                end
            end
            if has_sim == false || true_has_sim == false
                all_exist = false
            end
        end
    end
    all_exist
end
```

Q1: Multi-variate Binary Classification

So far, we have only considered algorithms for regression where the target is continuous. We will now explore implementations of algorithms for multi-variate binary classification.

As before, we have broken our ML 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

The only baseline we will use in this assignment is a random classifier.

RandomModel

```
begin
    """
        RandomModel

    Predicts `w*x` where `w` is sampled from a normal distribution.
    """
    struct RandomModel <: AbstractModel # random weights
        W::Matrix{Float64}
        γ::Float64 # Threshold on binary classification confidence
    end
    RandomModel(in, out) = RandomModel(randn(in, out), 0.5)
    # predict(logit::RandomModel, X::AbstractMatrix) = sigmoid(X*logit.W) .>= Array(logit.γ, length(X*logit.W), 1) ? 1.0 : 0.0
    Base.copy(logit::RandomModel) = RandomModel(randn(size(logit.W)...), logit.γ)
    train!(::MiniBatchGD, model::RandomModel, lossfunc, opt, X, Y, num_epochs) =
        nothing
end;
```

```
function predict(logit::RandomModel, X::AbstractMatrix)
    Ŷ = sigmoid(X*logit.W)
    pred = zeros(size(Ŷ))
    for i in 1:length(Ŷ)
        if Ŷ[i] >= logit.γ
            pred[i] = 1.0
        else
            pred[i] = 0.0
        end
    end
    pred
end;
```

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.

```
Main.workspace2.AbstractModel
```

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

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

Q1: Logistic Regression ✅

```
begin
    __check_logit_reg = let
        rng = Random.MersenneTwister(1)
        _X = rand(rng, 3, 3)
        X = sigmoid(_X)
        X_true = [0.5587358993498943 0.5019773105398053 0.7215004060928302;
0.5857727098994119 0.6197795961579493 0.7310398330188039; 0.5775458635048137
0.5525472988567002 0.562585578409889]
        all(X .≈ X_true)
        end
    HTML("<h2 id=dist> Q1: Logistic Regression
$(_check_complete(__check_logit_reg))")
end
```

Linear Model

As before, we define a linear model that inputs a vector $\mathbf{x}$ and outputs prediction $\mathbf{x}^\top \mathbf{w}$. We can also input the whole data matrix $\mathbf{X}$ and output a prediction for each row using $\mathbf{X}\mathbf{w}$. 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 logistic regression, the default is to return the inputs themselves. In polynomial logistic regression, we will replace this function with one that returns polynomial features.

```
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

end;
```

Logistic Regression Model

Logistic regression is very similar to linear regression. But, unlike linear regression where y is a continuous variable, the targets are binary $y \in \{0, 1\}$ for logistic regression. Instead, we directly learn $p(y|\mathbf{x})$ with logistic regression, and then return the prediction $\hat{y}$ that has higher probability. To limit the range of the learned probabilities within $[0, 1]$, we use a *sigmoid* transformation.

$$p(y = 1|\mathbf{x}) = \sigma(w_0 + w_1 x_1 + w_2 x_2 + .. + w_n x_n)$$

where w_0 represents the bias term. To take the bias term into account, we need to add a column of 1s to the input matrix $\mathbf{X}$, resulting in the new dimension `(samples, features+1)`.

In this part you need to implement the sigmoid function. The sigmoid function takes in a scalar z and is equal to $\text{sigmoid}(z) = 1/(1 + \exp(-z))$. To implement it, we can more generally assume it inputs a vector $\mathbf{z} = (z_1, z_2, \ldots, z_n)$ and outputs the sigmoid function applied to each element of this $\mathbf{z}$ (elementwise). Namely,
$\text{sigmoid}(\mathbf{z}) = (1/(1 + \exp(-z_1)), 1/(1 + \exp(-z_2)), \ldots, 1/(1 + \exp(-z_n)))$. You can either implement this with a for loop, or use the fact that we can do elementwise operations on vectors. The `exp` function can be applied elementwise using `exp.(-z)`. And as before, elementwise addition is .+ and elementwise division is ./. If you find bugs using elementwise operations, then start with the straightforward for loop approach, get that working, and then experiment with elementwise operations.

```
begin
    struct LogisticRegressor <: AbstractModel
        model::LinearModel
        γ::Float64 # the probabilty threshold on the output class confidence
        is_poly::Bool
    end

    LogisticRegressor(in, out=1; γ=0.5, is_poly=false) = if is_poly
        in = in - 1
        LogisticRegressor(LinearModel(in+1, out), γ, is_poly) # (feture size + 1 for bias term)  × output size
    else
        LogisticRegressor(LinearModel(in+1, out), γ, is_poly) # (feture size + 1 for bias term)  × output size
    end
    Base.copy(lr::LogisticRegressor) = LogisticRegressor(copy(lr.model),lr.γ,lr.is_poly)
    get_linear_model(lr::LogisticRegressor) = lr.model
end;
```

```
# Add a column of 1 to X to count for the bias term. Start with an "else" statement.
function get_features(m::LogisticRegressor, X::AbstractMatrix)
    d = size(X, 2)
    _X = ones(size(X,1), d+1)
    _X[:, 1:d] = X
    X = _X
end;
```

```
function predict(lr::LogisticRegressor, X::AbstractMatrix)
    if lr.is_poly
        Ŷ = sigmoid(predict(lr.model, X))
    else
        Ŷ = sigmoid(predict(lr.model, get_features(lr, X)))
    end
    pred = zeros(size(Ŷ))
    for i in 1:length(Ŷ)
        if Ŷ[i] >= lr.γ
            pred[i] = 1.0
        else
            pred[i] = 0.0
        end
    end
    pred
end;
```

```
function sigmoid(z)
    z
    #### BEGIN SOLUTION
    for i in 1:size(z,1)
        for j in 1:size(z,2)
            z[i,j] = 1/(1 + exp(-z[i,j]))
        end
    end
    return z
    #### END SOLUTION
end;
```

(a) Polynomial Features ✅

Now, we will implement Polynomial Model which uses the linear model for learning on top of non-linear features. We apply a polynomial transformation to our data, like the previous assignment, but now do a higher degree of $p = 3$.

Recall from the last assignment to obtain 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 function

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

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 by constructing Φ with $p = 3$ degrees in the function `get_features`.

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

```
begin
    struct Polynomial3Model <: AbstractModel
        model::LogisticRegressor
        ignore_first::Bool
    end

    Polynomial3Model(in, out=1; ignore_first=false) =
        Polynomial3Model(LogisticRegressor(1 + in + Int(in*(in+1)/2) + Int(floor((in*(in+1)/2)*(in+2)/3.0)), out, is_poly=true), ignore_first)

    Base.copy(lm::Polynomial3Model) = Polynomial3Model(copy(lm.model), lm.ignore_first)
    get_linear_model(lm::Polynomial3Model) = lm.model.model

end
```

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

```
predict(lr::Polynomial3Model, X) = predict(lr.model, get_features(lr, X))
```

```
function get_features(pm::Polynomial3Model, _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)
                   Int(floor(Int(m*(m+1)/2) * (m+2)/3))  # combinations (i.e. x_i*x_j*x_k)

    Φ = zeros(N, num_features)
    # Construct Φ
    #### BEGIN SOLUTION
    rows, cols = size(X)
    for i in  1:rows
        count = 1
        Φ[i, count] = 1.0
        count += 1
        for j in 1:cols
            Φ[i, count] = X[i,j]
            count += 1
            for k in j+1:cols
                Φ[i, count] = X[i,j] * X[i,k]
                count += 1
            end
            Φ[i, count] = X[i,j] * X[i,j]
            count = count + 1
        end


        for j in 1:cols
            for k in j+1:cols
                Φ[i, count] = X[i,j] * X[i,j] * X[i,k]
                count = count + 1
                Φ[i, count] = X[i,j] * X[i,k]* X[i,k]
                count = count + 1
            end
            Φ[i, count] = X[i,j] * X[i,j] * X[i,j]
            count = count + 1
        end

    end

    #### END SOLUTION

    Φ
end;
```

(b) Mini-batch Gradient Descent ✅

```
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;
```

```
struct MiniBatchGD
    b::Int
end
```

In this notebook, we will be focusing on minibatch gradient descent. Below you need to (re)implement the function `epoch!`. You can just use your code from Assignment 3 on `MBGD`. **We are not grading this section again, since it is the same code as before. But, if it is incorrect, it will give wrong results in later sections, resulting in deductions there. Please ensure this code is correct.** This function should go through the data set in mini-batches of size `mbgd.b`. 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

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

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

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

    #### BEGIN SOLUTION
    start = 1
    batch_size = mbgd.b
    randindex = randperm(length(Y))
    for i in 1:floor(Int, size(X,1)/ mbgd.b)
        X_b = X[randindex[start : batch_size], :]
        Y_b = Y[randindex[start : batch_size]]
        update!(model, lossfunc, opt, X_b, Y_b)
    end
    #### END SOLUTION
end;
```

```
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;
```

```
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;
```

```
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 ✅

For this notebook we only use the cross-entropy, but we still use the abstract type LossFunction as a standard abstract type for all losses. Below you will need to implement the `loss` ✅ function and the `gradient` ✅ function for Cross_Entropy.

```
abstract type LossFunction end
```

(c) Cross-entropy

The cross-entropy loss function for the whole dataset is

$$c(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} c_i(\mathbf{w})$$

for cross-entropy loss c_i on datapoint i

$$c_i(\mathbf{w}) = -y_i \ln \sigma(\mathbf{x}_i^T\mathbf{w}) - (1-y_i)\ln(1-\sigma(\mathbf{x}_i^T\mathbf{w}))$$

You should be using the sigmoid function σ that you implemented above. When computing the loss for a minibatch $\{i_1, i_2, \ldots, i_b\}$, we sum losses c_i for those points

$$\frac{1}{b}\sum_{j=1}^{b} c_{i_j}(\mathbf{w})$$

You will compute the loss for a given batch of data, inputted into `loss` as `X` and `Y`. This data could be the entire dataset, or a minibatch.

```
struct CrossEntropy <: LossFunction end
```

```
function loss(lm::AbstractModel, ce::CrossEntropy, X , Y)
    θ = predict(lm, X) # θ = Xw
    loss = 0.0
    #### BEGIN SOLUTION
    for i in 1:length(Y)
        sigma = sigmoid(transpose(X[i,:]) * lm.w)
        loss += (-Y[i] * log(sigma[1])) - ((1 - Y[i]) * log(1 - sigma[1]))
    end
    return loss/ length(Y)
    #### END SOLUTION
end;
```

(d) Gradient of Cross-Entropy

You will implement the gradient of the cross-entropy loss function in the `gradient` function, returning a matrix of the same size of `lm.w` using the following formula:

$$\nabla c_i(\mathbf{w}) = (\sigma(\mathbf{x}_i^T\mathbf{w}) - y_i)\mathbf{x}_i$$

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

    #### BEGIN SOLUTION
    for i in 1: length(Y)
        sigma = sigmoid(transpose(X[i,:]) * lm.w)
        ∇w += (sigma[1] - Y[i])*X[i, :]
    end
    return ∇w/length(Y)
    #### END SOLUTION

    @assert size(∇w) == size(lm.w)
    ∇w
end;
```

Optimizers ✅

Below you will need to implement an optimizer:

- RMSprop ✅

```
abstract type Optimizer end
```

(f) RMSprop ✅

RMSprop is another adaptive learning rate that uses a different learning rate for every parameter w_j. Instead of taking cumulative sum of squared gradients as in AdaGrad, we take the exponential moving average of these gradients. The motivation for doing so is to allow the gradient to stay larger for longer, if needed. The stepsize for Adagrad can decay quite quickly, due to the accumulation of squared gradients.

To implement RMSprop, we use the following equations:

$$\bar{\mathbf{g}}_{t+1} = \beta \bar{\mathbf{g}}_t + (1-\beta)\mathbf{g}_t^2$$
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta}{\sqrt{\bar{\mathbf{g}}_{t+1} + \epsilon}}\mathbf{g}_t$$

where $\mathbf{g}_t$ is the gradient at time step t and the addition, squaring, multiplication and division are all elementwise. The coefficient β represents the degree of weighting decrease, a constant smoothing factor between 0 and 1. A higher β discounts older observations faster.

Implement `RMSprop` below.

```
begin
    mutable struct RMSprop <: Optimizer
        η::Float64 # step size
        β::Float64 # The significance coefficient on the most recent data points
        gbar::Matrix{Float64} # exponential decaying average
        ϵ::Float64 #
    end

    RMSprop(η) = RMSprop(η, 0.9, zeros(1, 1), 1e-5)
    RMSprop(η, lm::LinearModel) = RMSprop(η, 0.9, zero(lm.w), 1e-5)
    RMSprop(η, model::AbstractModel) = RMSprop(η, get_linear_model(model))
    Base.copy(rmsprop::RMSprop) = RMSprop(rmsprop.η, rmsprop.β, zero(rmsprop.gbar), rmsprop.ϵ)
end
```

```
function update!(lm::LinearModel,
                 lf::LossFunction,
                 opt::RMSprop,
                 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
    for i in 1: length(lm.w)
        opt.gbar[i] = opt.β * opt.gbar[i] + (1- opt.β) * g[i]^2
        lm.w[i] = lm.w[i] - (opt.η * 1/sqrt(opt.gbar[i] + opt.ϵ)) * g[i]
    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 child `GDLearningProblem` is used to construct a learning problem. Once again, we introduce an abstract type even though we only have one child, but provide a design that does not constrain the types of learners that are possible. This struct contain all the information needed to `train!` a model for gradient descent. We also provide the `run` and `run!` functions. These will apply the feature transform (if there is one) and train the model. `run` does this with a copy of the learning problem, while `run!` does this inplace.

```
abstract type LearningProblem end
```

```
"""
    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)
```

```
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;
```

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

Accuracy and Misclassification Error

The Accuracy of a model is the percent correct on the given (testing) data, namely the percent of points where we predicted the correct class. The misclassification error is 100 minus this number, shows the percent incorrect. The misclassification error on the test set is an estimate of the expected 0-1 cost of the classifier, given as a percent rather than a percentage between 0 and 1.

```
function get_accuracy(Y, Ŷ)
    correct = 0
    # count number of correct predictions
    correct = sum(Y .== Ŷ)
    # return percent correct
    return (correct / Float64(length(Y))) * 100.0
end;
```

```
function get_misclassification_error(Y, Ŷ)
    return (100 - get_accuracy(Y, Ŷ))
end;
```

Run Experiment

Below are the helper functions for running an 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,
                        data,
                        num_epochs,
                        runs)
    err = zeros(runs)

    X, Y = data.X, data.Y
    train_size=20000
    test_size = 1000

    for i in 1:runs

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

        # 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] = get_misclassification_error(test_data[2], Ŷ)
    end

    err
end;
```

Experiments

In this section, we will run an experiment on the algorithms we implemented above. We provide the data in the `Data` section, and then follow the experiment and its description. You will need to analyze and understand the experiment for the written portion of this assignment.

The Dataset

We use a the physics dataset, from the UCI repository. We normalize the columns using min-max scaling. A description of the dataset is given below, given by the group that released the dataset.

```
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;
```

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

Here is a description of the physics dataset in case you are interested.

```
The data has been produced using Monte Carlo simulations and contains events with two leptons (electrons or muons). In high energy physics experiments, such as the ATLAS and CMS detectors at the CERN LHC, one major hope is the discovery of new particles. To accomplish this task, physicists attempt to sift through data events and classify them as either a signal of some new physics process or particle, or instead a background event from understood Standard Model processes. Unfortunately we will never know for sure what underlying physical process happened (the only information to which we have access are the final state particles). However, we can attempt to define parts of phase space that will have a high percentage of signal events. Typically this is done by using a series of simple requirements on the kinematic quantities of the final state particles, for example having one or more leptons with large amounts of momentum that is transverse to the beam line ( pT ). Here instead we will use logistic regression in order to attempt to find out the relative probability that an event is from a signal or a background event and rather than using the kinematic quantities of final state particles directly we will use the output of our logistic regression to define a part of phase space that is enriched in signal events. The dataset we are using has the value of 18 kinematic variables ("features") of the event. The first 8 features are direct measurements of final state particles, in this case the  pT , pseudo-rapidity ( η ), and azimuthal angle ( ϕ ) of two leptons in the event and the amount of missing transverse momentum (MET) together with its azimuthal angle. The last ten features are functions of the first 8 features; these are high-level features derived by physicists to help discriminate between the two classes. You can think of them as physicists attempt to use non-linear functions to classify signal and background events and they have been developed with a lot of deep thinking on the part of physicist. There is however, an interest in using deep learning methods to obviate the need for physicists to manually develop such features. Benchmark results using Bayesian Decision Trees from a standard physics package and 5-layer neural networks and the dropout algorithm are presented in the original paper to compare the ability of deep-learning to bypass the need of using such high level features. We will also explore this topic in later notebooks. The dataset consists of 5 million events, the first 4,500,000 of which we will use for training the model and the last 500,000 examples will be used as a test set.
```

Plotting our results

The `plot_results` function produces two plots. The left plot is a box plot over the errors, the right plot is a bar graph displaying average errors with standard error bars. This function will be used for all the experiments, and you should use this to finish your written experiments.

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

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

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

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

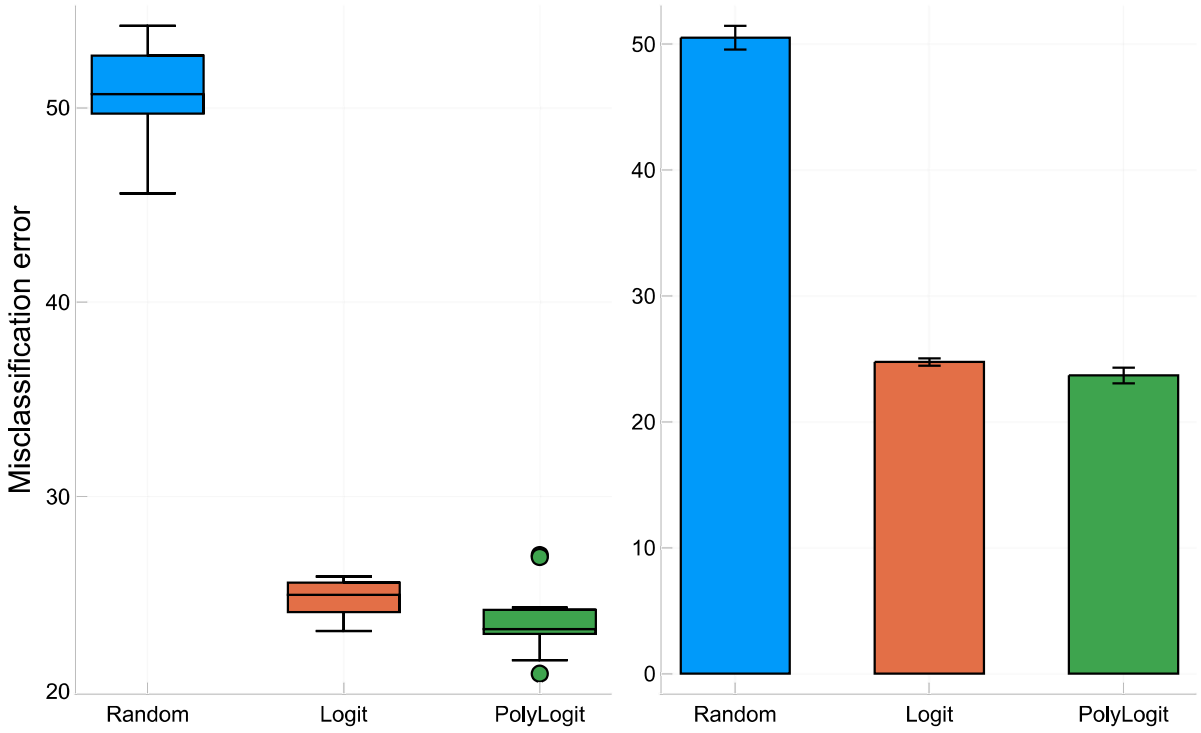
(g) Evaluating the Classifiers

We will compare different classifiers on the **Physics dataset**.

To run this experiment click ☑

You can get the misclassification error to report from the plot or from the terminal where your ran this notebook.

Note that it might take a bit of time to run this experiment (even a few minutes), so be a bit patient waiting for the graph to appear right below this text.




```
begin
    if __run_class
        algs = ["Random", "Logit", "PolyLogit"]
        classification_problems = [
            GDLearningProblem(
                MiniBatchGD(200),
                RandomModel(8, 1),
                RMSprop(0.01),
                CrossEntropy()),
            GDLearningProblem(
                MiniBatchGD(200),
                LogisticRegressor(8, 1),
                RMSprop(0.01),
                CrossEntropy()),
            GDLearningProblem(
                MiniBatchGD(200),
                Polynomial3Model(8, 1),
                RMSprop(0.01),
                CrossEntropy())
            ];

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

            for (idx, prblms) in enumerate(classification_problems)

                err = run_experiment(prblms, data, 100, 10)
                push!(errs, err)
            end
            errs
        end

        mean_error_Random = mean(misclass_errs[1])
        mean_error_Logit = mean(misclass_errs[2])
        mean_error_PolyLogit = mean(misclass_errs[3])
```

```
        println("Misclassification error on the test set for Random model is $mean_error_Random.")

        println("Misclassification error on the test set for Logistic Regression model is $mean_error_Logit.")

        println("Misclassification error on the test set for Polynomial Logistic Regression model is $mean_error_PolyLogit.")

        plot_results(algs, misclass_errs)
    end

end
```

Q2: Hypothesis Testing

In this question, you will use the paired t-test to compare the performance of two models. You will compare the two models from above (logistic regression and polynomial logistic regression) both using RMSprop for optimization. You hypothesize that polynomial logistic regression is better than logistic regression and you want to run a one-tailed t-test to see if this is true.

The paired t-test is designed for continuous errors, rather than 0-1 errors. For 0-1 errors, and one test set, we often use McNemar's test. However, there is another alternative. We can run the algorithm on multiple random training and testing splits (say 10) to get 10 estimates of misclassification error (average 0-1 error on a test set, as a percent). The standard deviation and average of these 10 errors provides a useful insight into the quality of the model we would get for this problem. We therefore run each algorithm 10 times, on random training and test splits. We control the splits so that both algorithms train and test on the same split, to have paired samples for the 10 samples of error. We then conduct a paired t-test on these 10 numbers.

The misclassification error estimate above, in fact, is the average of these 10 numbers. It looks like polynomial logistic regression is better than logistic regression. Now we will examine whether we can say this with confidence (that the difference is statistically significant).

(a) Defining the Null Hypothesis

Define the null hypothesis and the alternative hypothesis. Assume μ_1 to be the true expected misclassification error for Logistic Regression and μ_2 to be the true expected misclassification error for Polynomial Logistic Regression. This is not code that is run, rather it is simply your written answer to your question. For marking purposes, it is easier for us if you to write this description here. Note that to write these answers, you have to put them in comments because they are not valid Julia code.

```
# discussion should go here
#### BEGIN SOLUTION

# Null Hypothesis: Mu_1 = Mu_2 or Mu_1 - Mu_2 = 0, that is Polynomial logistic regression is not better than logistic regression as the true expected squared error of pol. logistic regression Mu_2 is equal to Mu_1
# Alternative Hypothesis: Mu_1 > Mu_2 holds. That is, Polynomial Logistic regression is better than Logistic regression

#### END SOLUTION
```

(b) Checking for Assumptions ✅

Before running the paired t-test, you should check that the assumptions are not violated. One way to satisfy the assumption for the paired t-test is to check that the errors are (approximately) normally distributed with (approximately) equal variances. The Student's t-distribution is approximately like a normal distribution, with a degrees-of-freedom parameter $m - 1$ that makes the distribution look more like a normal distribution as m becomes larger.

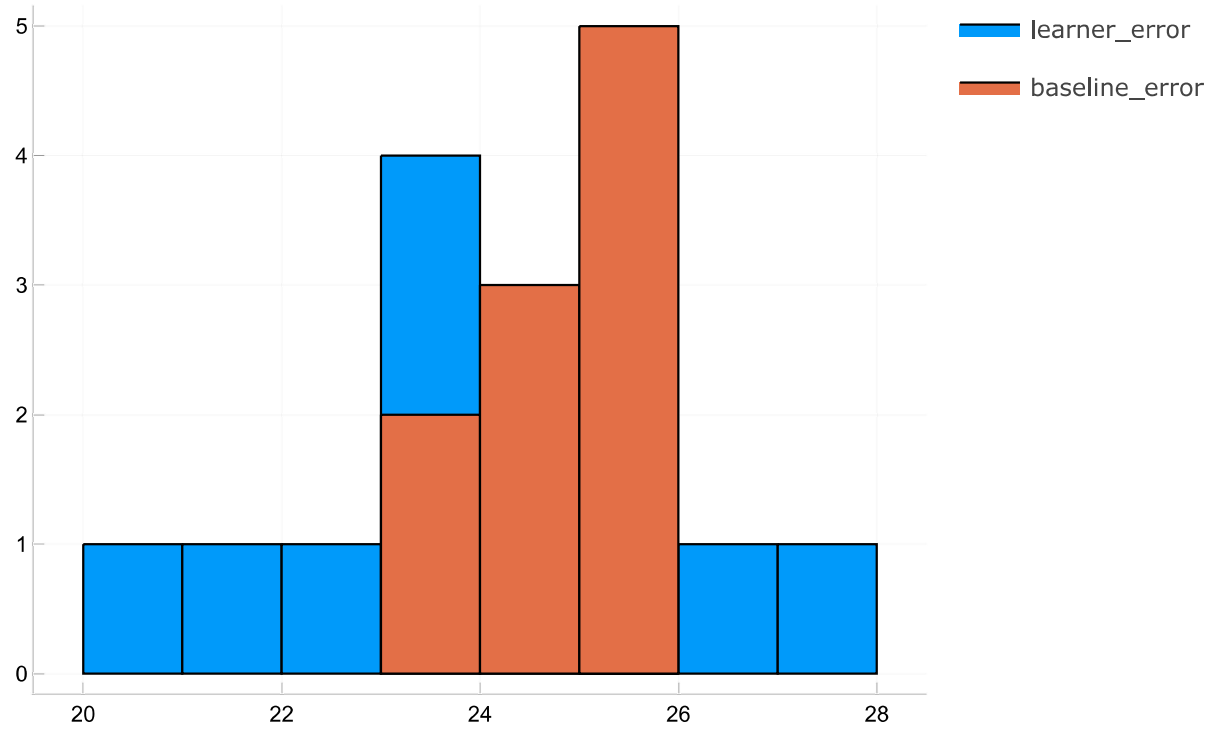
To do this, you need to implement the `checkforPrerequisites` method below. For each model, you can plot a histogram of its errors on the test set. You can do so by using the two vectors of errors and the function `plot_histogram` function to visualize the error distributions simultaneously. Discuss why it is ok or not ok to use the paired t-test to get statistically sound conclusions about these two models. From Q1, you will use the `logisticRegression_error` as the `baseline_error` and `PolynomialLogisticRegression_error` as the `learner_error`.

Once again, this is not code that is run. Rather, we will read your comments. Note that this answer need not be long; a few sentences suffices.

```
# discussion should go here
#### BEGIN SOLUTION

#### END SOLUTION
```

```
function plot_histogram(baseline_error::AbstractVector{Float64},
    learner_error::AbstractVector{Float64})
    histogram([learner_error baseline_error], label = ["learner_error" "baseline_error"])
end;
```

```
function checkforPrerequisites(baseline_error::AbstractVector{Float64},
        learner_error::AbstractVector{Float64},
        learner_name::AbstractString,
        baseline_name::AbstractString)
    # Compute mean and std of the error distributions and plot their histograms

    mu_1, mu_2, std_1, std_2 = 0.0, 0.0, 0.0, 0.0
    #### BEGIN SOLUTION
    mu_1 = mean( baseline_error)
    std_1 = std(baseline_error)

    mu_2 = mean( learner_error)
    std_2 = std(learner_error)

    # plot_histogram(baseline_error, learner_error)

    #### END SOLUTION

    println("BaseLine Error: mean = $mu_1 and Standard deviation = $std_1")
    println("Learner Error: mean = $mu_2 and Standard deviation = $std_2")

    mu_1, mu_2, std_1, std_2

end;
```




```
begin
    if __run_class
        e1 = misclass_errs[2]
        e2 = misclass_errs[3]

        checkforPrerequisites(baseline_error, learner_error, "LogisticRegression", "PolynomialLogisticRegression")
        plot_histogram(e1, e2)

    end
end
```

(c) Running the t-test ✅

Regardless of the outcome of (b), let's run the one-tailed t-test. (Note, I am not advocating that you check for violated assumptions and then ignore the outcome of that step. The goal of this question is simply to give you experience actually running a statistical significance test. Presumably, in practice, you would pick an appropriate one after verifying assumptions).

To run this test, you need to compute the p-value. To do this implement the `getPValue` method, which returns the p-value for the one-tailed paired t-test. It looks at the positive part of the tail: we consider the difference between the baseline and learner, where if it is a larger positive number that the error for the learner is much lower than the baseline. This test therefore checks if the learner (polynomial logistic regression) is statistically significantly better than the baseline (logistic regression).

Report the p-value. Would you be able to reject the null hypothesis with a significance threshold of 0.05? How about of 0.01? Include both the p-value as well as this discussion in the comments right below this text.

```
# discussion should go here
#### BEGIN SOLUTION


#### END SOLUTION
```

```
# helper function to get the positive tail p-value using t-distribution
function pValueTDistPositiveTail(t::Float64, dof::Int64)
    1 - cdf(TDist(dof), t)
end;
```

```
function tDistPValue(baseline_error::AbstractVector{Float64},
learner_error::AbstractVector{Float64})
    # Computes the p-value using one-tailed t-test
    @assert size(learner_error) == size(baseline_error)
    m = size(learner_error, 1) # the number of test samples
    dof = m - 1
    t = 0.0
    #### BEGIN SOLUTION
    s = 0.0
    d = zeros(size(learner_error))
    # println(d)
    d = baseline_error - learner_error
    d_bar = sum(d) / m
    s_d = sqrt(sum((d .- d_bar).^2)/dof)
    t = d_bar/(s_d/sqrt(m))

    # println(1 - cdf(TDist(dof), t))
    #### END SOLUTION
    pValueTDistPositiveTail(t, dof)
end;
```

Next, you will run the t_test given the functions implemented, then you can tell whether we reject the null hypothesis or not.

```
function t_test(baseline_error::AbstractVector{Float64},
        learner_error::AbstractVector{Float64},
        learner_name::AbstractString,
        baseline_name::AbstractString,
        pvalueThreshold::Float64)

    checkforPrerequisites(baseline_error, learner_error, baseline_name, learner_name)
    pval = tDistPValue(baseline_error, learner_error)

    if pval < pvalueThreshold
        result = "rejected"
    else
        result = "not rejected"
    end
    println("With pvalue = $pval, the null hypothesis is $result under a pvalueThreshold of $pvalueThreshold")
end;
```

```
begin
    if __run_class
        baseline_error = misclass_errs[2]
        learner_error =  misclass_errs[3]

        baseline_name = "LogisticRegression"
        learner_name = "PolynomialLogisticRegression"

        pvalueThreshold = 0.05

        t_test(baseline_error, learner_error, learner_name, baseline_name, pvalueThreshold)
    end
end
```