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
```

```
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(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

Q1: Multi-variate Binary Classification

So far, we have only considered regressor algorithms. In the following section we will explore implementations of algorithms for multi-variate binary classification.

Similar 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 would be using in this assignment is a random classifier.

RandomModel

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

```julia
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
```

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

```julia
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

```
"""
    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 1 method)

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

update_transform! (generic function with 1 method)

```
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)
        println(X)
        true
        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 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.

```
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 the Y is a continuous variable, logistic regression needs to have the predicted Y to lie between 0 and 1. As a result, the predicted value of Y is nothing but the probability of Y equals 1, that is, $P(Y=1)$. So, to limit the predicted value within $[0,1]$ range, we applied a *sigmoid* transformation in *predict*.

$$P(Y=1) = \sigma(w_0 + w_1x_1 + w_2x_2 + .. + w_nx_n)$$

where w_0 represents the bias term. To take the bias term into account, we need to add a column of 1 to regenerate the input matrix X as of size `(samples, features+1)`.

```
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;
```

```julia
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;
```

```julia
function sigmoid(z)
    z
    #### BEGIN SOLUTION
    1.0./(1.0.+exp.(-z))

    #### END SOLUTION
end;
```

(a) Polynomial Features ✅

```julia
begin

    __check_Poly2_logit_reg = let
        pm = Polynomial3Model(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 0.013149828447265864 0.00044063994193260575 1.4765482242222028e-5 4.947791725125075e-7; 1.0 0.34651701419196046 0.4886128300795012 0.12007404112451132 0.16931265897503248 0.2387424977182995 0.04160769821242834 0.05866971705292988 6 0.08272833747007607 0.11665264747038717; 1.0 0.3127069683360675 0.21096820215853596 0.09778564804593431 0.06597122691230639 0.04450758232200489 0.03057825354722182 0.020629662365158116 0.013917831135882103 0.00938968462489641]
        check_1 = all(Φ .≈ Φ_true)
        pm = Polynomial3Model(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_3 = (size(Φ)==size(Φ_true))
        check_1 && check_2 && check_3
    end

    HTML("<h4 id=poly> (a) Polynomial Features $(_check_complete(__check_Poly2_logit_reg))</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 Φ with $p = 3$ degrees in the function `get_features`.

```
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+1)/2.0)), 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(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+1)/2))  # combinations (i.e. x_i*x_j*x_k)

    Φ = zeros(N, num_features)

    # Construct Φ
    #### BEGIN SOLUTION
    for i in 1:N
        index = []
        append!(index, 1)
        for j in 1:m
            append!(index, X[i,j])
        end
        for j in 1:m
            for k in j:m
                append!(index, X[i,j]*X[i,k])
            end
        end
        for j in 1:m
            for k in j:m
                for l in k:m
                    append!(index, X[i,j]*X[i,l]*X[i,k])
                end
            end
        end
        for j in 1:size(index,1)
            Φ[i,j] = index[j]
        end
    end
    #### END SOLUTION

    Φ
end;
```

(b) Mini-batch Gradient Descent ✅

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

In this notebook, we will be focusing on minibatch gradient descent and using a new learning rate adaptation rule called `RMSprop`.

Below you need to (re)implement the function `epoch!`. You can just use your code for Assignment 3 on `MBGD`. **There is no penalty for this section if you got it wrong. Yet, if a bug in this section causes any wrong results in the other sections, you will still get penalized for them.** This function should go through the data set in mini-batches 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

```
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**.

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

    #### BEGIN SOLUTION
    b=1
    match=randperm(length(Y))
    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;
```

```
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 ✅

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

For this notebook we will only be using 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.

```
begin
    _X = [4 3 4 1; 1 0 5 1; 1 5 6 1; 4 4 7 1; 2 4 8 1]
    __check_cegrad = all(gradient(LinearModel(4, 1), CrossEntropy(), _X, [1.0,0.0,0.0,1.0,1.0]) .== [-0.8; -0.6000000000000001; -0.8; -0.1])
    __check_celoss = loss(LinearModel(4, 1), CrossEntropy(), _X, [1.0,0.0,0.0,1.0,1.0]) == 0.6931471805599454

    __check_CrossEntropy = __check_celoss && __check_cegrad

md"""
For this notebook we will only be using 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` $(_check_complete(__check_celoss)) function and the `gradient` $(_check_complete(__check_cegrad)) function for Cross_Entropy.
"""
end
```

```
abstract type LossFunction end
```

(c) Cross-entropy

We will be implementing the loss function of `Cross_Entropy`.

$$c(w) = -\frac{1}{n}\sum_{i}^{n}(y_i ln\sigma(x_i w^T) + (1-y_i)ln(1-\sigma(x_i w^T)))$$

where $f(x)$ is the prediction from the passed model. You should be using the sigmoid function defined in linear model.

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

```
function loss(lm::AbstractModel, ce::CrossEntropy, X , Y)
    θ = predict(lm, X) # θ = XW'
    loss = 0.0
    #### BEGIN SOLUTION

    s=0

    Z=sigmoid(θ)

    for i in 1:size(Z,1)
        s=s + (Y[i] * log.(Z[i]) + (1.0 -Y[i]) .* log.(1.0 - Z[i]))
    end

    ce=-1/size(X,1)*s

    #### END SOLUTION
end;
```

(d) Gradient of Cross_Entropy

```
md"""
#### (d) Gradient of Cross_Entropy
"""
```

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

$$\nabla W = \frac{1}{n}\sum_{i=1}^{n}(\sigma(x_i w^T) - y_i)x_i$$

```
function gradient(lm::AbstractModel, ce::CrossEntropy, X::Matrix, Y::Vector)
    ∇W = zero(lm.W) # gradients should be the size of the weights
    θ = predict(lm, X)
    Z= sigmoid(θ)
    #### BEGIN SOLUTION
    for i in 1:size(∇W,1)
        for j in 1:size(Z,1)
            ∇W[i] = ∇W[i]+(Z[j] - Y[j])*X[j,i]
        end
    end
    ∇W=1/size(Z,1)*∇W


    #### 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 ✅

```
begin
     __check_RMSprop_v, __check_RMSprop_W = let
        lm = LinearModel(2, 1)
        opt = RMSprop(0.1, lm)
        X = [0.1 0.5;
             0.5 0.0;
             1.0 0.2]
        Y = [1, 0, 1]
        update!(lm, CrossEntropy(), opt, X, Y)
        true_G = [0.0009999999999999996; 0.0013611111111111105]
        true_W = [0.31465838776377636; 0.31507247500483543]
        all(opt.G .≈ true_G), all(lm.W .≈ true_W)
    end

    __check_RMSprop = __check_RMSprop_v && __check_RMSprop_W

md"""
#### (f) RMSprop $(_check_complete(__check_RMSprop))


"""
end
```

Root mean square prop or RMSprop is another adaptive learning rate that uses a different learning rate for every parameter W_i and tries to improve AdaGrad.

Instead of taking cumulative sum of squared gradients as like in AdaGrad, we take the exponential moving average of these gradients. To implement RMSprop optimizer, we use the following equations:

$$G_i = \beta G_{i-1} + (1-\beta) g_i^2$$
$$W_i = W_{i-1} - \frac{\eta}{\sqrt{G_i + \epsilon}} * g_i$$

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

```
begin
    mutable struct RMSprop <: Optimizer
        η::Float64 # step size
        β::Float64 # The significance coefficient on the most recent data points
        G::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.G),
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.G) # need to make sure this is of the right shape.
        opt.G = zero(g)
    end

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

    #### BEGIN SOLUTION
    opt.G=β*G+(1-β)*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.

```
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)
```

```
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

The Accuracy of a model is the total number of classes predicted correctly by the model.

```
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_acc_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,
                        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] = get_acc_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.

Data

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

```
"""
    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;
```

```
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;
```

Physics Dataset

```
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;
```

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.

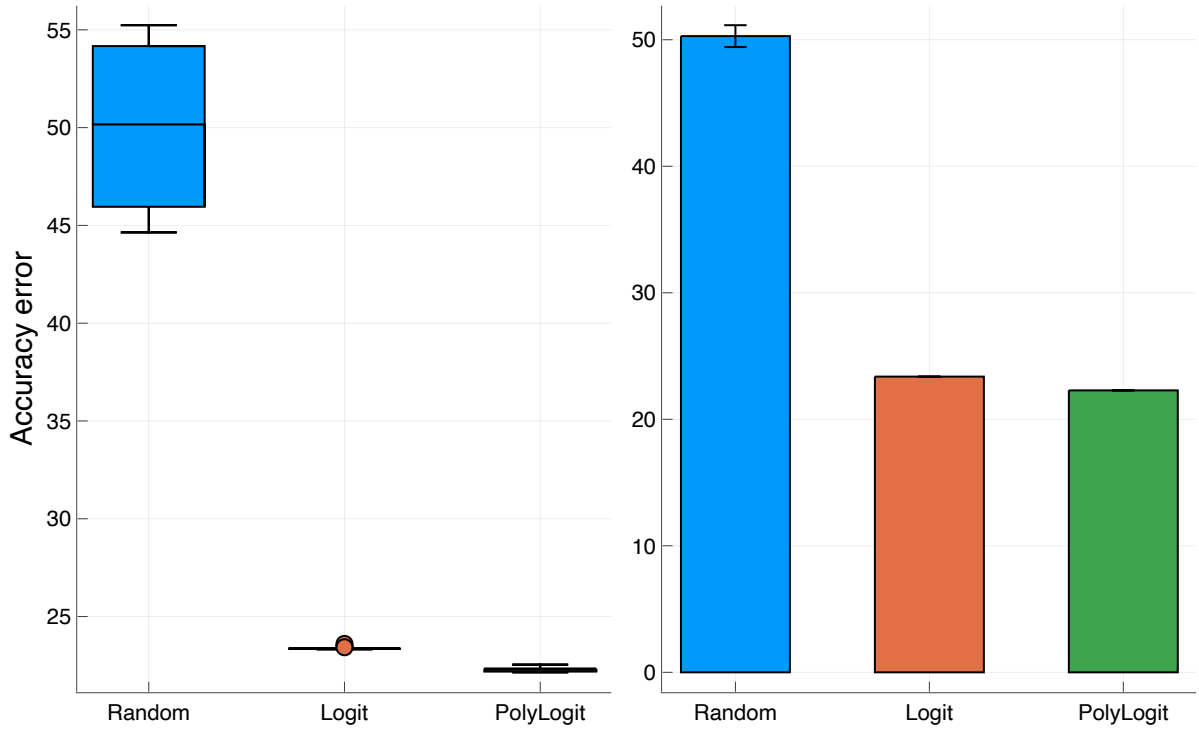
```
function plot_data(algs, errs; vert=false)
    stderr(x) = sqrt(var(x)/length(x))

    plt1 = boxplot(reshape(algs, 1, :),
                   errs,
                   legend=false, ylabel="Accuracy 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 ? "Accuracy error" : "")

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

(g) Evaluating Classifiers

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

To run this experiment click ☑

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




```
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())
            ];

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

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

            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(classification_problems)

                err = run_experiment(prblms, train_data, test_data, 100, 20)
                push!(errs, err)
            end
```

```
            errs
        end

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

        println("Average accuracy error on test set for Random model is $mean_error_Random.")

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

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

        plot_data(algs, acc_errs)
    end

end
```

And 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 t
wo leptons (electrons or muons). In high energy physics experiments, such as the AT
LAS and CMS detectors at the CERN LHC, one major hope is the discovery of new parti
cles. To accomplish this task, physicists attempt to sift through data events and c
lassify 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 informatio
n to which we have access are the final state particles). However, we can attempt t
o define parts of phase space that will have a high percentage of signal events. Ty
pically this is done by using a series of simple requirements on the kinematic quan
tities of the final state particles, for example having one or more leptons with la
rge 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 probabili
ty that an event is from a signal or a background event and rather than using the k
inematic 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 even
ts. The dataset we are using has the value of 18 kinematic variables ("features") o
f 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 lept
ons 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 t
he two classes. You can think of them as physicists attempt to use non-linear funct
ions to classify signal and background events and they have been developed with a l
ot of deep thinking on the part of physicist. There is however, an interest in usin
g deep learning methods to obviate the need for physicists to manually develop such
features. Benchmark results using Bayesian Decision Trees from a standard physics p
ackage and 5-layer neural networks and the dropout algorithm are presented in the o
riginal paper to compare the ability of deep-learning to bypass the need of using s
uch high level features. We will also explore this topic in later notebooks. The da
taset consists of 5 million events, the first 4,500,000 of which we will use for tr
aining the model and the last 500,000 examples will be used as a test set.
```

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. The hypothesis is that polynomial logistic regression is better than logistic regression and you want to run a one-tailed test to see if this is true.

(a) Defining Null hypothesis

Define the null hypothesis and the alternative hypothesis. Assume μ_1 to be the true expected accuracy error for LogisticRegressor and μ_2 to be the true expected accuracy error for "PolynomialLogisticRegressor".

syntax: extra token "hypothesis" after end of expression

1. **top-level scope** @ *none:1*

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

Null hypothesis: mu_1 = mu_2 or mu_1-mu_2 = 0
Alternate hypothesis: mu_1 > mu_2

#### END SOLUTION
```

(b) Checking for Assumptions ✅

Before running the tailed 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`.

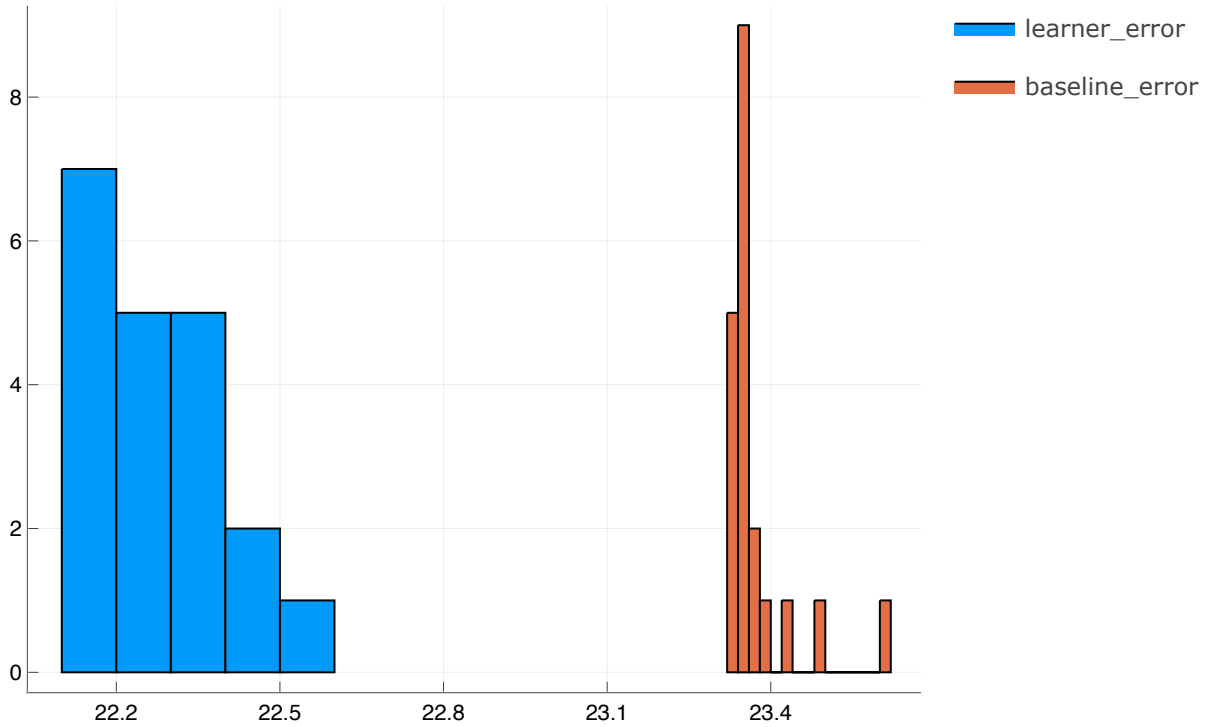
syntax: extra token "worker" after end of expression

1. **top-level scope** @ *none:1*

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

From worker 2:  BaseLine Error: mean = 23.374104676308455 and Standard deviation =
0.06485828985542887
From worker 2:  Learner Error: mean = 22.28459105738822 and Standard deviation =
0.10664609122575452
In the class notes its given
"This test can be used
if both errors appear to be distributed normally and if they have similar variance."
Since the models have dissimilar variance and its not normally distributed we cant
use
a paired t-test.


#### 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, mu_2, std_1, std_2 =
mean(baseline_error),mean(learner_error),std(baseline_error),std(learner_error)
    plot_histogram(baseline_error, learner_error)
    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
    e1 = acc_errs[2]
    e2 = acc_errs[3]

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

end
```

(c) Running the t-test ✅

Regardless of the outcome of (b), let's run the paired 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. 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?

syntax: extra token "worker" after end of expression

1. **top-level scope** @ *none:1*

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

From worker 2:  With pvalue = 0.0, the null hypothesis is rejected under a pvalueThreshold of 0.05
pvalue = 0.0 or close to 0
Therefore we can reject the Null hypothesis for a significant threshold of 0.05 and 0.01.

#### 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 paired t-test
    @assert size(learner_error) == size(baseline_error)
    m = size(learner_error, 1) # the number of features
    dof = m - 1
    t=0.0
    #### BEGIN SOLUTION
    q = baseline_error - learner_error
    u = sum(q)/m
    sd = sqrt(sum((q .- u).^2)/dof)
    t = u/(sd/sqrt(m))

    #### 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
    baseline_error = acc_errs[2]
    learner_error =  acc_errs[3]

    baseline_name = "LogisticRegression"
    learner_name = "PolynomialLogisticRegression"

    pvalueThreshold = 0.05

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

end
```