Homework 1 - *Linear Regression, Cross Validation, and Nonlinear Regression*

EE/CS5841, Spring 2024

```
# For this Homework I have used the following refernces :
# CS229 : https://www.youtube.com/watch?v=jGwO_UgTS7I&list=PLoROMvodv4rMiGQp3WXShtMGgzqpfVfbU
# LLM : I have used chatgpt, perplexity ai, to understand the code and how to do programming in julia from what I have known by doing mathematics on paper.
# Other resources :
# 1. https://web.mit.edu/zoya/www/linearRegression.pdf
# 2. https://discourse.julialang.org/t/simple-tool-for-train-test-split/473
```

Part 1: Linear Regression by least-squares (10 points)

```julia
begin
    using LinearAlgebra
    using Random
    using DelimitedFiles
end
```

Function definitions:

linear_regression_train

Creates weights for a least-squares solution for linear regression.

Inputs:

x - n samples x m features

y - n samples x 1 (for now we can assume that we have only a single output)

Outputs:

w - m features x 1 (again, assuming single output variable)

linear_regression_infer

Predicts an output based on weights and inputs

Inputs:

x - n samples x m features

w - m features x 1

Outputs y_predicted - n samples x 1

Hints:

- The LinearAlgebra library includes useful things like pseudoinverse
- We can transpose with transpose() or using '
- We can check dimensions of matrices with size()
- Some useful matrices can be made with ones(), zeros(), and eye()
- Julia expects us to specify type for the above matrices, just use Float64 for most things unless you want an Int
- We can force arrays to have singleton dimensions with a ;; - see the simple test below for an example

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

```
function linear_regression_train(x,y)
    # your code here!
    x= hcat(ones(size(x, 1)), x)
    weights = inv(transpose(x) * x) * transpose(x) * y
    return weights
end
```

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

```
function linear_regression_infer(x,w)
    # your code here
    x= hcat(ones(size(x, 1)), x)
    predicted_y = x * w
    return predicted_y
end
```

Here's a simple test

```
begin #we have to use these for Pluto notebooks when we execute multiple things in the same cell
    x_demo = [1.0; 2.0; 3.0 ;;]
    y_demo = [3.0; 5.0; 7.0 ;;] # the ";;" makes it a column vector

    try
        w = linear_regression_train(x_demo,y_demo)
        y_demo_pred = linear_regression_infer(x_demo,w)

        if norm(y_demo - y_demo_pred) < 1e-10
            print("It works!")
        else
            print("Not working yet")
        end

    catch
        print("Something's wrong, perhaps dimensions aren't matching in your function?")
    end

end
```

```
It works!
```

Part 2, test train splitting and cross validation (10 points)

Function definitions:

mse

Mean squared error

Inputs:

y - a vector of true values y_est - a vector of estimated values

Outputs:

error - the mean squared error between the two vectors

test_train_split

Generates some indices for randomly splitting data. Assume an 80/20 split can just be hardcoded in. It is also acceptable to have this function take in the data instead of the number of points and the first version of the assignment did this.

Inputs:

n - number of data points

Outputs:

test_train_indices - a vector of 1's and 0's where 1's represent the entries to be used for test and the 0's represent entries to be used for training

cross_validate_split

Generate some indices for a cross validation

Inputs:

n - number of data points

k - number of splits for cross validation

Outputs:

split_indices - a vector of values representing which validation split that entry belongs to

eval_model

Function that allows you to pass in data, test/train split indices, model and loss functions, and runs everything for you

Inputs:

x - n samples x m features

y - n samples x 1 (for now we can assume that we have only a single output)

test_train_indices - a vector of 1's and 0's where 1's represent the entries to be used for test and the 0's represent entries to be used for training

model_train - name of the function that you are using to create the model, assumes it uses the same model representation as model_infer

model_infer - name of the function you are using to run inference on the model

loss - function used to compute loss

Outputs:

error - the error for the

run_cross_validation

Function that allows you to pass in data, cross validation split indices, model and loss functions, and runs everything for you

Inputs:

x - n samples x m features

y - n samples x 1 (for now we can assume that we have only a single output)

split_indices - a vector of 1's and 0's where 1's represent the entries to be used for test and the 0's represent entries to be used for training

model_train - name of the function that you are using to create the model, assumes it uses the same model representation as model_infer

model_infer - name of the function you are using to run inference on the model

loss - function used to compute loss

Outputs:

loss_per_run - a vector that is the same length as the number of cross validation folds that has the validation loss for that fold. if you're really motivated, you could make it output a training and validation loss seprately

Hints:

- use vectorized dot operations, for example if we wanted to find everything equal to 1 in an array, we could use '.== 1' and it would return a 'true' for every element that had a 1

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

```
function mse(y,y_est)
    # Your code here VV
    n = length(y)
    error = sum((y .- y_est).^2) / n
    return error
end

```

Let's write a simple test for this

```
begin
    y1 = [1 0 0]
    y2 = [1 2 0]
    try
        error = mse(y1,y2)
        if error - 1.33333333 < 1e-3
            print("Yay, it works!")
        else
            print("Something isn't right")
        end
    catch
        print("Something isn't working")
    end
end
```

```
Yay, it works!
```

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

```
function test_train_split(n)
    train_size = Int(0.8 * n)
    test_size = n - train_size
    test_train_indices = vcat(zeros(train_size), ones(test_size))
    test_train_indices = shuffle(test_train_indices)
    return test_train_indices
end
```

```
2×1 Matrix{Int64}:
 5
 8
```

```
#lazy visual test, needs asserts/condition checking for a proper test
#we can select from data using this one-hot by making it into a boolean
begin
    try

        example_split = test_train_split(10)
        selection_test_x = [1:10 ;;] #force it to be an nx1 if we only have 1
        feature
        example_train_indices = iszero.(example_split)
        test_indices = .!example_train_indices
        selection_test_x[test_indices,:]
    catch
    end
end
```

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

```
function cross_validate_split(n,k)
    # Your code here
    split_size = div(n, k)
    split_indices = zeros(Int, n)
    for i in 1:k
        start_idx = (i - 1) * split_size + 1
        end_idx = min(i * split_size, n)
        split_indices[start_idx:end_idx] .= i
    end
    split_indices = shuffle(split_indices)
    return split_indices
end
```

```
[4, 3, 5, 3, 1, 4, 2, 5, 2, 1]
```

```
#little visual test - it will update once the above code works
try
    cross_validate_split(10,5)
catch
end

```

```
#some demo code to show you how you could use the split indices to separate some
data
#this may be useful a little bit further down...
begin
    try

        example_split = cross_validate_split(10,5)
        selection_test_x = [1:10 ;;] #force it to be an nx1 if we only have 1
        feature
        splits = zeros(5,2)
        for split = 1:5
            val_set_indices = findall(==(split),example_split)
            splits[split,:] = selection_test_x[val_set_indices,:]
        end
        print(splits)
    catch
    end
end
```

```
[2.0 5.0; 6.0 7.0; 4.0 10.0; 3.0 8.0; 1.0 9.0]
```

This next part is just combining stuff from Part 1 with the loss function (MSE) from Part 2. It's just to see how we can pass functions into functions to do things for us.

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

```
function eval_model(x,y,test_train_indices,model_train,model_infer,loss)
    x = [x ;;] # force it to be an nX1 instead of just an n-element array if x is
    1D
    x_train = x[test_train_indices .== 0,:]
    y_train = y[test_train_indices .== 0,:]
    x_test = x[test_train_indices .== 1,:]
    y_test = y[test_train_indices .==1,:]
    trained_model = model_train(x_train, y_train)
    y_pred = model_infer(x_test, trained_model)
    error = loss(y_test, y_pred)
    return error
end
```

Let's reuse our test code from Part 1, modified slightly. This isn't a thorough test, as it's a zero-error case with almost no data, but we could write a second test with a higher known error if we wanted to include that as well.

```
begin
    try
        test_train_indices = [0,1,0]
        loss =
        eval_model(x_demo,y_demo,test_train_indices,linear_regression_train,linea
        r_regression_infer,mse)

        if loss < 1e-3
            print("It works!")
        else
            print("Not working yet")
        end

    catch
        print("Something's wrong, perhaps dimensions aren't matching in your
        function?")
    end

end
```

```
It works!
```

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

```
function run_cross_validation(x,y,split_indices,model_train,model_infer,loss)
    # Your code here
    x = [x ;;] #force it to be an nx1 if we only have 1 feature
    n_folds = maximum(split_indices)
    loss_per_run = zeros(n_folds)
    for fold in 1:n_folds
        test_train_indices = split_indices .== fold
        loss_per_run[fold] = eval_model(x, y, test_train_indices, model_train,
model_infer, loss)
    end

    return loss_per_run
end
```

Test it out on some synthetic data. Create a line with a slope of 2 and an intercept of 1 and add in some Gaussian noise

Run 5-fold cross validation on it and don't worry about a test/train split for now.

1.072133458475335

```
begin
    try
        x = collect(1:100)
        y = x*2+ones(Float64,100)+randn(Float64,100)
        split_indices = cross_validate_split(100,5)
        loss_per_run = run_cross_validation(x,y,split_indices,linear_regression_train,linear_regression_infer,mse)
        mean_error = sum(loss_per_run)/5
    catch
    end
end
```

Part 3 Regularized Linear Regression (10 points)

For now we will just use L2 regularization, so modify your code from the first set of linear regression functions to include an L2 regularization term

regularized_linear_regression_train

Performs L2-regularized linear regresion

Inputs:

x - n samples x m features

y - n samples x 1 (for now we can assume that we have only a single output)

lambda - regularization constant

Outputs:

w - m features x 1 (again, assuming single output variable)

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

```
function regularized_linear_regression_train(x,y,lambda)
    # your code here!
    x = hcat(ones(size(x, 1)), x)
    m = size(x, 2)
    regularization_matrix = lambda * I(m)
    weights = inv(transpose(x) * x + regularization_matrix) * transpose(x) * y
    return weights
end
```

```
Enter cell code...
```

```
3×1 Matrix{Float64}:
 1.357803640704271
 0.674425544613559
 0.6744255446135217
```

```
#let's do a simple test
try
    begin
        x_demo_reg = 1.0*[1 1; 3 3; 4 4]
        y_demo_reg = [3.0; 5.0; 7.0 ;;] # the ";;" makes it a column vector
        w_reg01 = regularized_linear_regression_train(x_demo_reg,y_demo_reg,0.1)
    end
catch
end
```

```
3×1 Matrix{Float64}:
 0.7023809523809517
 0.7619047619047639
 0.7619047619047632
```

```
#for fun we can vary the regularization constant
try
    begin
        x_demo_reg = 1.0*[1 1; 3 3; 4 4]
        y_demo_reg = [3.0; 5.0; 7.0 ;;]
        w_reg1 = regularized_linear_regression_train(x_demo_reg,y_demo_reg,1)
    end
catch
end

```

```
#we can compare with no regularization as well
try
    begin
        x_demo_reg = 1.0*[1 1; 3 3; 4 4]
        y_demo_reg = [3.0; 5.0; 7.0 ;;]
        w_noreg = linear_regression_train(x_demo_reg,y_demo_reg)
    end
catch
end

```

Part 4 Basis functions (10 points)

Modify the function below to create n polynomials for the input features x in increasing power

polynomial_basis

Input:

x - n samples x m features n - highest polynomial to create

Output:

xns - n samples x mXn features. features should be ordered as x1n1 x2n1 ... xmn1 x1n2 x2n2 ... xmn2 ... xmnn for each row of samples

Hints:

- We can concatenate arrays using square brackets
- We can do elementwise operations with the dot operator. ie. x.^2 squares each element

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

```
function polynomial_basis(x,n)
    # Your code here
    xns = []
    for power in 1:n
        xns =  hcat([x .^ power for power in 1:n]...)
    end

    return xns #xns should be {x,x^2,...x^n}
end

```

```
2×6 Matrix{Int64}:
 1  2  1   4   1   8
 3  4  9  16  27  64
```

```
#simple visual test
begin
    try
        x_mat = [1 2; 3 4]
        xmat_n2= polynomial_basis(x_mat,3)
    catch
    end
end
```

Part 5 Put it all together! (10 points)

Download and load the following dataset: Yacht Hydrodynamics: https://archive.ics.uci.edu/ml/datasets/Yacht+Hydrodynamics Load the data and run some experiments with it using regularized linear or polynomial regression.

For full credit, separate off 20% as a test set, explore at least 3 options (regularization and/or polynomial basis functions), use cross validation to select the best one, and report the error on the held out dataset. Most importantly, report your conclusions and justify it with a sentence or two!!

```
begin
    #Download the data and load it using DelimitedFiles function
    #update the code with your file location or use another loading method
    local_data = readdlm("yacht_data.csv", ',', Any, '\n')
    local_data_to_use = convert(Array{Float64,2}, local_data[:,1:7])
    local_n = size(local_data_to_use, 1)
    new_split_indices = cross_validate_split(local_n, 5)
    x_new = local_data_to_use[:, 1:6]
    x_new = [x_new ;;]
    y_new = local_data_to_use[:, 7]
    y_new = [y_new ;;]
    lambda_values = 0.0001:0.5:10.0

    function new_run_cross_validation(x_new, y_new,
    new_split_indices,regularized_linear_regression_train,
    linear_regression_train, linear_regression_infer, mse, lambda=nothing)
        n_folds = maximum(new_split_indices)
        loss_per_run = zeros(n_folds)

        for fold in 1:n_folds
            test_train_indices = new_split_indices .== fold
            x_train = x_new[.!test_train_indices, :]
            y_train = y_new[.!test_train_indices, :]
            x_test = x_new[test_train_indices, :]
            y_test = y_new[test_train_indices, :]
```

```julia

        if lambda === nothing
            w = linear_regression_train(x_train, y_train)
        else

            w = regularized_linear_regression_train(x_train, y_train, lambda)
        end

        y_pred = linear_regression_infer(x_test, w)
        loss_per_run[fold] = mse(y_test, y_pred)
    end

    return loss_per_run
end


# Calculate errors for each lambda value
errors_reg = [new_run_cross_validation(x_new, y_new, new_split_indices, regularized_linear_regression_train, linear_regression_train, linear_regression_infer, mse, lambda_val) for lambda_val in lambda_values]


# Find the index of the minimum error
best_lambda_index_reg = argmin(errors_reg)
# Get the corresponding lambda value
best_lambda_reg = lambda_values[best_lambda_index_reg]

# Print the results
println("Mean Loss for Each Lambda: ", errors_reg)
println("Best Lambda Index: ", best_lambda_index_reg)
println("Best Lambda Value: ", best_lambda_reg)

println("Best Lambda for Regularized Regression: $best_lambda_reg, Mean Loss: $(errors_reg[best_lambda_index_reg])")

# Run cross-validation for polynomial regression with degrees 1 to 5
errors_poly = []
for degree in 1:10
    # Generate polynomial basis
    x_poly = polynomial_basis(x_new, degree)

    # Run cross-validation
    errors = [new_run_cross_validation(x_poly, y_new, new_split_indices, regularized_linear_regression_train, linear_regression_train, linear_regression_infer, mse, lambda_val) for lambda_val in lambda_values]

    # Save the average error for this degree
    push!(errors_poly, sum(errors) / 5)
end
# Find the degree with the minimum average error
best_degree_poly = argmin(errors_poly)

# Report results for polynomial regression
println("Best Degree for Polynomial Regression: ", best_degree_poly)
println("Average Cross-Validation Error for Polynomial Regression: ", errors_poly[best_degree_poly])
```

```

    function new_test_train_split(local_n)
        train_size = round(Int, 0.8 * local_n)
        test_size = local_n - train_size
        train_test_indices = vcat(zeros(train_size), ones(test_size))
        train_test_indices = shuffle(train_test_indices)
        return train_test_indices
    end


    # Now, separate off 20% as a test set and report the error on the held-out
    dataset for the best model
    new_train_test_indices = new_test_train_split(local_n)
    x_train = x_new[new_train_test_indices .== 0, :]
    y_train = y_new[new_train_test_indices .== 0, :]
    x_test = x_new[new_train_test_indices .== 1, :]
    y_test = y_new[new_train_test_indices .== 1, :]

    # Train the best model on the training set
    if best_degree_poly > 1
        x_train = polynomial_basis(x_train, best_degree_poly)
        x_test = polynomial_basis(x_test, best_degree_poly)
    end

    if best_lambda_reg > 0
        w_best = regularized_linear_regression_train(x_train, y_train,
        best_lambda_reg)
    else
        w_best = linear_regression_train(x_train, y_train)
    end

    # Make predictions on the test set
    y_pred_test = linear_regression_infer(x_test, w_best)

    # Calculate and report the test error
    test_error = mse(y_test, y_pred_test)
    println("Test Error for the Best Model: ", test_error)
    #Ignore the 8th null column and reference the website for what each column
    means
end
```

```
Mean Loss for Each Lambda: [[73.13597893840252, 84.7932320146545 5, 80.7459135429622, 85.73458126160574, 71.51706649145336], [75.1386231596296 7, 88.01833148487924, 81.39312509227567, 87.72265969220771, 74.96478232708034], [82.625104355779, 96.70165001363551, 86.97338935924103, 96.7443202502033 3, 82.52092483366155], [90.96468316808907, 106.62959979498498, 93.64023636439249, 106.63385734637447, 90.30992088613199], [99.03799704313319, 116.35665958118247, 100.28379896761633, 116.14096666091069, 97.6676907148307], [106.50006097699246, 125.3922519985465, 106.52568354686433, 124.88204827657839, 104.39868931599479], [113.27418921571079, 133.60857185132258, 112.25340635423643, 132.78165441961707, 110.47898968933555], [119.38305979662273, 141.01827327419355, 117.45868664116126, 139.87735202081083, 115.94828036258471], [124.88403958480914, 147.6853638994207, 122.17360973019514, 146.2445118598747 6, 120.86679512768248], [129.84306759379595, 153.6883621888366, 126.44383796512841, 151.96628538590278, 125.29774203852472], [134.32414075232174, 159.105155383460 7, 130.31716209766043, 157.12184541014557, 129.30037996447197], [138.3854191481204, 164.00718863897853, 133.83875265131195, 161.7822848397584, 132.927668738518], [142.07818381599512, 168.45771759665584, 137.0494195519167, 166.00974660504852, 136.22588006286463], [145.44699762170012, 172.51179970000618, 139.9852254141762, 169.85786838183998, 139.23501059999953], [148.5303302783897 5, 176.2169771495621, 142.67768782795318, 173.3726984410848, 141.98947755973023], [151.3613159957836, 179.6141786797402, 145.15421209326806, 176.5937138328199, 144.51886649949225], [153.968499362234 8, 182.73863171731264, 147.4385870286641, 179.5547868942413, 146.84863449427198], [156.3765119649631, 185.62069961699743, 149.5514682797893, 182.2850438070898 3, 149.0007335347996], [158.60666237351506, 188.28661580137188, 151.51081798307814, 184.80960273791345, 150.994147033487], [160.67744002445372, 190.7591124132860 6, 153.33229097695605, 187.15019774195946, 152.84534436035028]]
Best Lambda Index: 1
Best Lambda Value: 0.0001
Best Lambda for Regularized Regression: 0.0001, Mean Loss: [73.1359789384025 2, 84.79323201465455, 80.7459135429622, 85.73458126160574, 71.5170664914533 6]
```

4.4171410930234085

```
test_error
```

```
# Conclusions and Justify
#The test error for the best model was calculated to be 4.417, indicating that the model's performance on new data is relatively low,  it will generalize well to new data.
# The mean loss for regularized regression was minimized at a lambda value of 0.0001,  a very small regularization parameter was most effective in reducing overfitting.  the best degree for polynomial regression is 4. The optimal balance between bias and variance in this model is good ,  it has the lowest average cross-validation error.
```