Homework 2 - *KNN and Logistic Regression*

EE/CS5841 , Spring 2024

```
# References:
# I have used many references for this assisgment
# 1.
https://web.stanford.edu/class/archive/cs/cs109/cs109.1166/pdfs/40%20LogisticRegression.pdf
# 2. Cs229 lectures
# 3. https://cseweb.ucsd.edu/~elkan/250B/logreg.pdf
# 4.https://zlatankr.github.io/posts/2017/03/06/mle-gradient-descent
# 5. https://zstevenwu.com/courses/s20/csci5525/resources/slides/lecture05.pdf
# 6. https://fluxml.ai/Flux.jl/stable/tutorials/logistic_regression/
# 7. https://machinelearninggeek.com/mnist-with-julia/
# 8. https://int8.io/category/classification/
# 9. https://int8.io/logstic-regression-with-gradient-descent-in-julia/
# 10. https://int8.io/logistic-regression-part-ii-evaluation/
# 11. I have various Large language models for code understanding and generation julia and of debugging too
```

Template to load MNIST

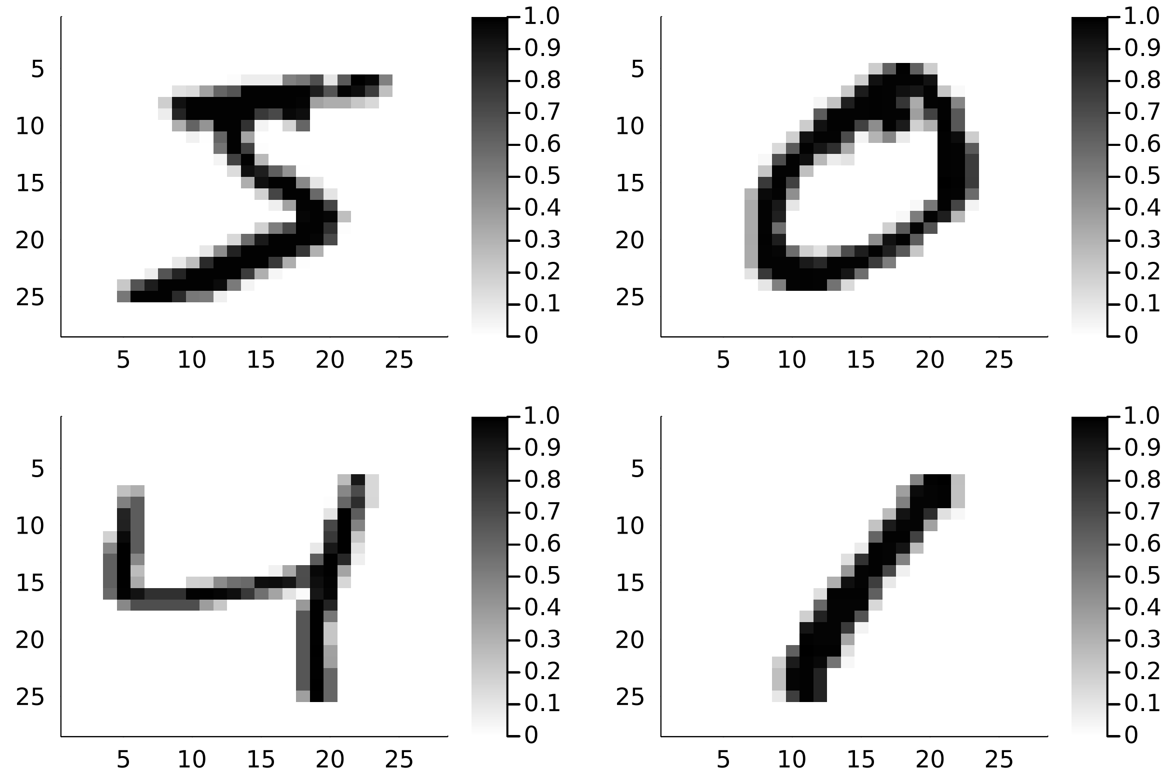
Below is an example template in Julia for loading the MNIST dataset and plotting a couple of images. (Make sure that you have the necessary packages installed, and adjust the paths accordingly)

```
begin
    using Plots
    using MLDatasets
    using LinearAlgebra
end
```

Selection deleted

```
(28, 28, 60000)
```

```
begin
    # load training set
    train_x, train_y = MNIST(split=:train)[:]
    train_x = permutedims(train_x,(2,1,3))

    # load test set
    test_x,  test_y  = MNIST(split=:test)[:]
    test_x = permutedims(test_x,(2,1,3))
    size(train_x)
end
```




```
begin
    # Plot a couple of training images. Note that the image y axis is flipped, this is common for images because of the way pixel position is notated.
    plot(heatmap(train_x[:,:,1], c=:binary, yflip=true),heatmap(train_x[:,:,2], c=:binary, yflip=true),heatmap(train_x[:,:,3], c=:binary, yflip=true),heatmap(train_x[:,:,4], c=:binary, yflip=true))

end
```

Selection deleted

```
[5, 0, 4, 1]
```

```
train_y[1:4]
```

(10 pts) Part 1: 1-nearest neighbour classifier

Use the MNIST dataset of handwritten digits with corresponding labels. It may be helpful to convert the 28x28 pixel image into a 784-dimensional feature vector. Implement a 1-nearest neighbor classifier without scaling or normalizing pixel values. The goal is to identify the number of images in the testing set that have been correctly labelled. Discuss the results and any insights gained from the analysis.

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

```
begin
        function one_nearest_neighbor_classifier(train_x, train_y, test_x)
            # Your code for 1-nearest neighbor classifier here
            # Loop over testing_data, find nearest neighbor in training_data, and
        check correctness
            infer_labels=[]
            for i in 1:size(test_x, 3)

                # Flatten the current test image into a 784-dimensional vector
                test_image = reshape(test_x[:, :, i], 784)

                # Find the index of the training image with the minimum Euclidean
        distance
                closest_index = argmin([LinearAlgebra.norm(test_image -
        reshape(train_x[:, :, j], 784)) for j in 1:size(train_x, 3)])

                # Infer the label from the training set and store it
                push!(infer_labels, train_y[closest_index])
            end
            return infer_labels
        end
    end
```

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

Selection deleted

```
begin
    function accuracy(infer_labels, true_labels)
    # Your code here
    accuracy = sum(infer_labels .== true_labels) / length(true_labels)
    println(accuracy)
    return accuracy
end
end
```

0.9691

```
accuracy(one_nearest_neighbor_classifier(train_x, train_y, test_x), test_y)
```

0.9691

Selection deleted

(10 pts) Part 2: KNN leave-one-out

Implement a KNN leave-one-out approach and test values of K from 1 to 20. Plot the leave-one-out error vs. K. Add a comment discussing the results. (If you are running into time problems using all 60,000 data points for leave-one-out, feel free to randomly sample the training set to estimate the best K.)

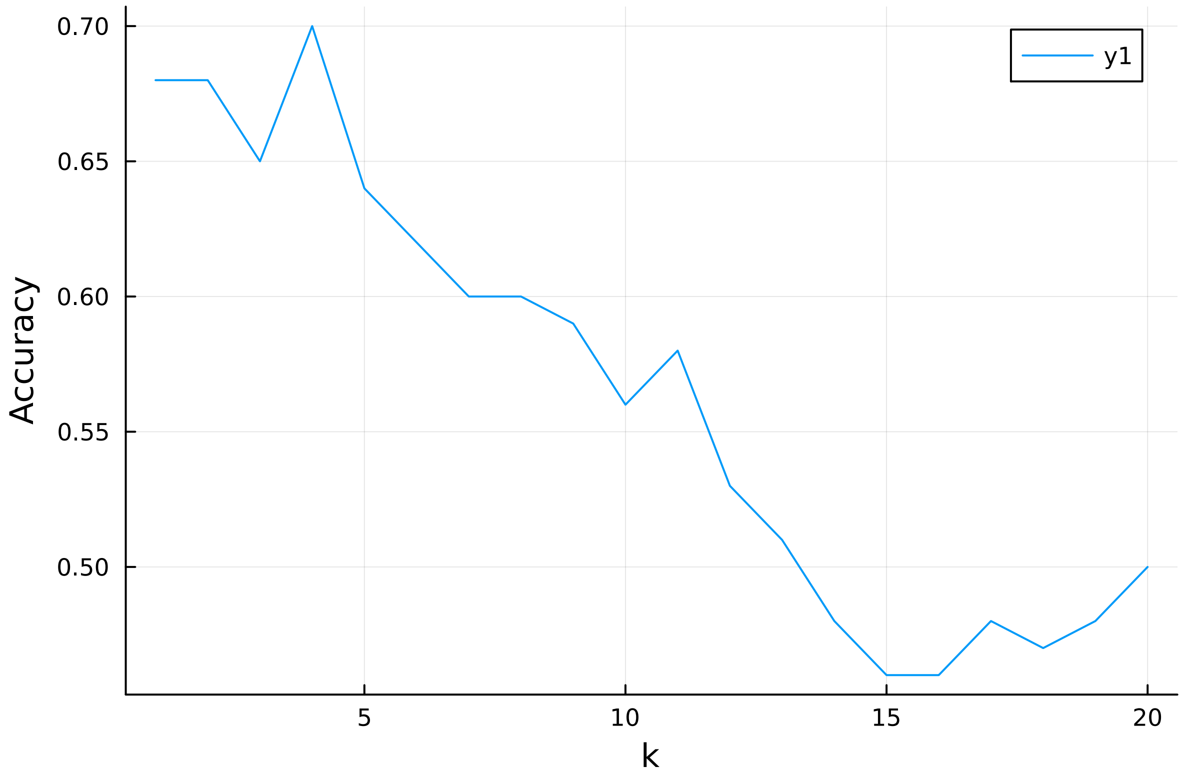
Selection deleted

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

```julia
begin
using Random
using Statistics
using StatsBase
    # Set a fixed seed for reproducibility
    seed_value = 42
    Random.seed!(seed_value)

    # Define your sample size
    sample_size = 100

    # Randomly sample indices for both training and test sets
    train_indices = randperm(size(train_x, 3))[1:sample_size]
    test_indices = randperm(size(test_x, 3))[1:sample_size]

    # Use the sampled indices to extract the corresponding data
    sampled_train_x = train_x[:, :, train_indices]
    sampled_train_y = train_y[train_indices]

    sampled_test_x = test_x[:, :, test_indices]
    sampled_test_y = test_y[test_indices]

    function knn_leave_one_out(train_x,train_y, k_values)
        # Your code for KNN leave-one-out approach here
        infer_labels = []

        for k in k_values
            inferred_labels = []
            correct_predictions = 0

            for i in 1:sample_size
                # Flatten the current test image into a 784-dimensional vector
                test_image = reshape(sampled_test_x[:, :, i], 784)

                # Find the indices and labels of the k-nearest neighbors
                distances = [LinearAlgebra.norm(test_image - reshape(train_x[:, :, j], 784)) for j in 1:sample_size]
                nearest_indices = sortperm(distances)[1:k]
                nearest_labels = train_y[nearest_indices]

                # Infer the label by majority voting
                inferred_label = argmax(countmap((nearest_labels)))
                push!(inferred_labels, inferred_label)
            end

            push!(infer_labels, inferred_labels)
        end
        return infer_labels

    end
end
```

Selection deleted

```
WARNING: using StatsBase.crossentropy in module workspace#4 conflicts with a
n existing identifier.
```




```
begin
    #Plot your accuracy vs k here
    k = 1:20
    accur=[]
    for i in k
    predicted_y=knn_leave_one_out(sampled_train_x,sampled_train_y,k)
    acc = sum(predicted_y[i].==sampled_test_y)/length(sampled_test_y)
        push!(accur, acc)
    end
        #replace this with your real accuracy!!
    plot(k,accur,xlabel="k",ylabel="Accuracy",)

end


```

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

Selection deleted

(30 pts) Part 3: Downsampling and KNN Experiment

(10 pts) Section a: Downsampling

Implement a downsampling function for an image dataset and conducting a K-nearest neighbors (KNN) leave-one-out experiment. The goal is to observe the impact of downsampling on the testing results and subjectively evaluate the query time of the classifier.

The function downsample_image takes an image represented as a 784-dimensional feature vector (1D array of UInt8) and a downsampling factor n. The function should downsample the image by selecting every nth pixel (feature) and return the downsampled feature vector.

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

```
function downsample_image(image, n)
    # Your code for downsampling here
    # Downsample by selecting every nth pixel along both dimensions
    downsampled_image =image[1:n:end, 1:n:end]

    # Reshape the downsampled image back into a 1D array
    return vec(downsampled_image)
end

```

Selection deleted

knn_leave_one_out_experiment (generic function with 1 method)

```
begin
    function knn_leave_one_out_experiment(train_x, train_y, test_x, test_y, k_values)
    # Your code for the KNN leave-one-out experiment here
    downsample_factor= 10
    accuracy_vector = []

    for k in k_values
        inferred_labels = []

        for i in 1:size(test_x, 3)
            # Flatten and downsample the current test image
            test_image = downsample_image(reshape(sampled_test_x[:, :, i], 784),
downsample_factor)

            # Downsample training images as well
            sampled_train_x_downsampled =
[downsample_image(reshape(sampled_train_x[:, :, j], 784), downsample_factor) for j
in 1:size(train_x, 3)]

            # Find the indices and labels of the k-nearest neighbors
            distances = [LinearAlgebra.norm(test_image -
sampled_train_x_downsampled[j]) for j in 1:size(train_x, 3)]
            nearest_indices = sortperm(distances)[1:k]
            nearest_labels = sampled_train_y[nearest_indices]

            # Infer the label by majority voting
            inferred_label = argmax(countmap((nearest_labels)))
            push!(inferred_labels, inferred_label)
        end

        # Calculate accuracy and store in accuracy_vector
        acc = accuracy(inferred_labels, sampled_test_y)
        push!(accuracy_vector, acc)
    end

    return accuracy_vector
end
end
```

Selection deleted

```
[0.68, 0.63, 0.62, 0.62, 0.59, 0.59, 0.58, 0.58, 0.58, 0.58, 0.56, 0.56, 0.52, 0.49, 0.5, 0
```

```
knn_leave_one_out_experiment(sampled_train_x, sampled_train_y, sampled_test_x,
sampled_test_y, k)
```

```
0.68
0.63
0.62
0.62
0.59
0.59
0.58
0.58
0.58
0.58
0.56
0.56
0.52
0.49
0.5
0.51
0.48
0.47
0.47
0.47
```

(20 pts) section b: Bin-based Downsampling

Implement a function that downsamples the image using cluster-based downsampling. For example, if n is 4, bin the pixels into groups of 4 and replace each pixel in a cluster with the average value of the pixels in that cluster. Create a second downsampling function that takes the the max value of the pixels in the bin. Comment on the testing results and the query time of the classifier. Start from the 28x28 images, bin them (maybe try displaying what they look like downsampled if you're curious), and then reshape them into a 1xm vector before using your classifier.

Try a bin size of 2x2 and 4x4 for full credit. If you're really curious, feel free to try other sizes.

Selection deleted

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

```
begin
    function average_pooled_downsampling(image, n)
    # Your code for clustering and downsampling here
    height, width, num_samples = size(image)

    # Calculate the number of clusters in each dimension
    clusters_per_row = div(width, n)
    clusters_per_col = div(height, n)

    # Initialize the downsampled image
    downsampled_image = zeros(Float64, clusters_per_col, clusters_per_row,
num_samples)

    for k in 1:num_samples
        for i in 1:clusters_per_col
            for j in 1:clusters_per_row
                # Define the boundaries of the current cluster
                start_row, end_row = (i - 1) * n + 1, i * n
                start_col, end_col = (j - 1) * n + 1, j * n

                # Extract the pixels within the current cluster
                cluster_pixels = image[start_row:end_row, start_col:end_col, k]

                # Calculate the average value for the cluster
                average_value = mean(cluster_pixels)

                # Assign the average value to the downsampled image
                downsampled_image[i, j, k] = average_value
            end
        end
    end

    return downsampled_image
end
end
```

Selection deleted

```
 0.0242647  0.101716  0.0         0.257353  0.823284  0.0         0.0
 0.0        0.0       0.0         0.457108  0.497059  0.0         0.0
 0.0        0.0       0.0         0.678922  0.456373  0.0         0.0
 0.0        0.0       0.0         0.339706  0.104902  0.0         0.0

;;; …

[:, :, 98] =
 0.0  0.0  0.0        0.0        0.0       0.0  0.0
 0.0  0.0  0.0419118  0.715686   0.376961  0.0  0.0
 0.0  0.0  0.41299    0.118382   0.575735  0.0  0.0
 0.0  0.0  0.328676   0.707598   0.443627  0.0  0.0
 0.0  0.0  0.527451   0.0404412  0.407108  0.0  0.0
 0.0  0.0  0.520343   0.400245   0.334804  0.0  0.0
 0.0  0.0  0.0        0.0        0.0       0.0  0.0

[:, :, 99] =
 0.0  0.0        0.0669118  0.201471  0.0        0.0         0.0
 0.0  0.0232843  0.693382   0.669608  0.121324   0.0         0.0
 0.0  0.0        0.0193627  0.409559  0.255637   0.0         0.0
 0.0  0.0        0.0        0.66299   0.0335784  0.00784314  0.0
 0.0  0.0        0.36348    0.931128  0.899755   0.478186    0.0
 0.0  0.0        0.520343   0.393873  0.0958333  0.106373    0.0
 0.0  0.0        0.0        0.0       0.0        0.0         0.0

[:, :, 100] =
 0.0  0.0         0.0       0.0       0.0       0.0         0.0
 0.0  0.00833333  0.332843  0.542157  0.332353  0.00833333  0.0
 0.0  0.640931    0.555882  0.25098   0.779902  0.707353    0.0
 0.0  0.902206    0.0       0.0       0.0       0.902941    0.0
 0.0  0.680637    0.557353  0.361765  0.626226  0.683333    0.0
 0.0  0.0720588   0.476961  0.540196  0.353676  0.00808824  0.0
 0.0  0.0         0.0       0.0       0.0       0.0         0.0
```

```
average_pooled_downsampling(sampled_train_x, 4)
```

Selection deleted

```
14×14×100 Array{Float64, 3}:
[:, :, 1] =
 0.0  0.0         0.0        0.0       …  0.0        0.0       0.0       0.0  0.0
 0.0  0.0         0.0        0.0          0.0284314  0.0       0.0       0.0  0.0
 0.0  0.0         0.0        0.0          0.896078   0.223529  0.0       0.0  0.0
 0.0  0.0         0.0        0.0          0.992157   0.859804  0.027451  0.0  0.0
 0.0  0.0         0.0        0.0          0.626471   0.988235  0.272549  0.0  0.0
 0.0  0.0         0.0        0.0       …  0.497059   0.990196  0.434314  0.0  0.0
 0.0  0.0         0.0656863  0.558824     0.361765   0.988235  0.494118  0.0  0.0
 0.0  0.00392157  0.654902   0.992157     0.42451    0.990196  0.495098  0.0  0.0
 0.0  0.227451    0.988235   0.67451      0.591177   0.988235  0.407843  0.0  0.0
 0.0  0.498039    0.990196   0.965686     0.992157   0.857843  0.027451  0.0  0.0
 0.0  0.095098    0.605882   0.907843  …  0.714706   0.222549  0.0       0.0  0.0
 0.0  0.0         0.0        0.027451     0.0        0.0       0.0       0.0  0.0
 0.0  0.0         0.0        0.0          0.0        0.0       0.0       0.0  0.0
 0.0  0.0         0.0        0.0          0.0        0.0       0.0       0.0  0.0

[:, :, 2] =
 0.0  0.0  0.0  0.0         0.0        …  0.0        0.0       0.0       0.0
 0.0  0.0  0.0  0.0         0.0           0.0        0.0       0.0       0.0
 0.0  0.0  0.0  0.0         0.0           0.0        0.0       0.0       0.0
 0.0  0.0  0.0  0.0         0.0           0.0        0.0       0.0       0.0
 0.0  0.0  0.0  0.0         0.287255      0.0156863  0.0       0.0       0.0
 0.0  0.0  0.0  0.132353    0.914706   …  0.221569   0.0       0.0       0.0
 0.0  0.0  0.0  0.00588235  0.0490196     0.288235   0.0       0.0       0.0
 0.0  0.0  0.0  0.177451    0.669608      0.0892157  0.0       0.110784  0.0
 0.0  0.0  0.0  0.876471    0.293137      0.798039   0.737255  0.660784  0.0
 0.0  0.0  0.0  0.671569    0.611765      0.0147059  0.0       0.0       0.0
 0.0  0.0  0.0  0.0117647   0.147059   …  0.0        0.0       0.0       0.0
 0.0  0.0  0.0  0.0         0.0           0.0        0.0       0.0       0.0
 0.0  0.0  0.0  0.0         0.0           0.0        0.0       0.0       0.0
 0.0  0.0  0.0  0.0         0.0           0.0        0.0       0.0       0.0
```

```julia
average_pooled_downsampling(sampled_train_x, 2)
```

Selection deleted

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

```julia
begin
function max_pooled_downsampling(image, n)
    # Your code for clustering and downsampling here
    # Get the dimensions of the original image
     height, width, num_samples = size(image)

    # Calculate the number of clusters in each dimension
    clusters_per_row = div(width, n)
    clusters_per_col = div(height, n)

    # Initialize the downsampled image
   downsampled_image = zeros(Float64, clusters_per_col, clusters_per_row, num_samples)

    # Iterate over clusters and calculate max value for each
    for k in 1:num_samples
    for i in 1:clusters_per_col
        for j in 1:clusters_per_row
        # Define the boundaries of the current cluster
        start_row, end_row = (i - 1) * n + 1, i * n
        start_col, end_col = (j - 1) * n + 1, j * n

        # Extract the pixels within the current cluster
        cluster_pixels = image[start_row:end_row, start_col:end_col,k]

        # Calculate the max value for the cluster
        max_value = maximum(cluster_pixels)

        # Assign the max value to the downsampled image
        downsampled_image[i, j, k] = max_value
    end
   end
    end

    return downsampled_image
end
end
```

Selection deleted

```
7×7×100 Array{Float64, 3}:
[:, :, 1] =
 0.0        0.0       0.0       0.0       0.113725  0.0       0.0
 0.0        0.0       0.0       0.996078  0.996078  0.992157  0.0
 0.0        0.0       0.996078  0.992157  0.996078  0.992157  0.0
 0.0156863  0.996078  0.992157  0.941176  0.94902   0.992157  0.0
 1.0        0.996078  0.996078  0.996078  0.996078  0.992157  0.0
 0.380392   0.992157  0.992157  0.992157  0.992157  0.741176  0.0
 0.0        0.0       0.0       0.0       0.0       0.0       0.0

[:, :, 2] =
 0.0  0.0        0.0       0.0       0.0       0.0       0.0
 0.0  0.0        0.0       0.298039  0.0       0.0       0.0
 0.0  0.423529   0.996078  0.996078  0.996078  0.443137  0.0
 0.0  0.670588   0.996078  0.917647  1.0       0.619608  0.329412
 0.0  1.0        0.996078  0.996078  0.996078  0.941176  0.898039
 0.0  0.0470588  0.294118  0.0       0.0       0.0       0.0
 0.0  0.0        0.0       0.0       0.0       0.0       0.0

[:, :, 3] =
 0.0       0.0       0.0       0.0       0.0       0.0       0.0
 0.0       0.0       0.121569  0.992157  1.0       0.972549  0.0
 0.913725  0.996078  0.996078  0.996078  0.996078  0.972549  0.0
 0.388235  0.662745  0.0       0.996078  1.0       0.0       0.0
 0.0       0.0       0.0       0.996078  0.996078  0.0       0.0
 0.0       0.0       0.0       1.0       0.996078  0.0       0.0
 0.0       0.0       0.0       1.0       0.662745  0.0       0.0

;;; …

[:, :, 98] =
 0.0  0.0  0.0       0.0       0.0       0.0  0.0
 0.0  0.0  0.576471  1.0       0.992157  0.0  0.0
```

```julia
max_pooled_downsampling(sampled_train_x, 4)
```

Selection deleted

(40 pts) Part 4: Regularized Logistic regression classifier

Build a regularized logistic regression classifier, where you use ridge (`2) regularization. Test this classifier on the MNIST data set by developing 10 classifiers: 0 versus all, 1 versus all, 2 versus all, ... , 9 versus all. Provide a confusion matrix, accuracy for each digit, and overall accuracy. Plot the overall test accuracy versus the regularization value where a log-scale is used for regularization value.

Essentially, the '1 versus all' classifier is trained to give you a probability of the digit 1 versus all other digits. Hence, digit 1 is class +1 and all other digits are class 0. Hence, to classify a test image, you take the maximum probability from all 10 classifiers, giving the predicted class of the input image. `2 regularized logistic regression uses the following log-likelihood,

$$L(w) = \sum_{i=1}^{N} \log(1+\exp(-y_i w^T x_i)) + \lambda \|w\|_2$$

Selection deleted

$$\mathcal{L}(\mathbf{w}) = \sum_{i=1}^{N} \log(1 + \exp(-y_i \mathbf{w}^T \mathbf{x}_i)) + \frac{\lambda}{2}\|\mathbf{w}\|_2.$$

```

md"""
# (40 pts) Part 4: Regularized Logistic regression classifier
Build a regularized logistic regression classifier, where you use ridge (‘2) regularization. Test this classifier
on the MNIST data set by developing 10 classifiers: 0 versus all, 1 versus all, 2 versus all, ... , 9 versus all. Provide a confusion
matrix, accuracy for each digit, and overall accuracy. Plot the overall test accuracy versus the regularization value where a
log-scale is used for regularization value.

Essentially, the ‘1 versus all’ classifier is trained to give you a probability of the digit 1 versus all other digits. Hence, digit
1 is class +1 and all other digits are class 0. Hence, to classify a test image, you take the maximum probability from all 10
classifiers, giving the predicted class of the input image.
‘2 regularized logistic regression uses the following log-likelihood,

>### L(w) = ᵢ₌₁∑ᴺlog(1+exp(-yᵢwᵀxᵢ)) + λ ‖w‖₂
#
![eq](https://miro.medium.com/v2/resize:fit:640/format:webp/1*lmbn5fQTXB4gxqFylmhGqQ.png)


"""

```

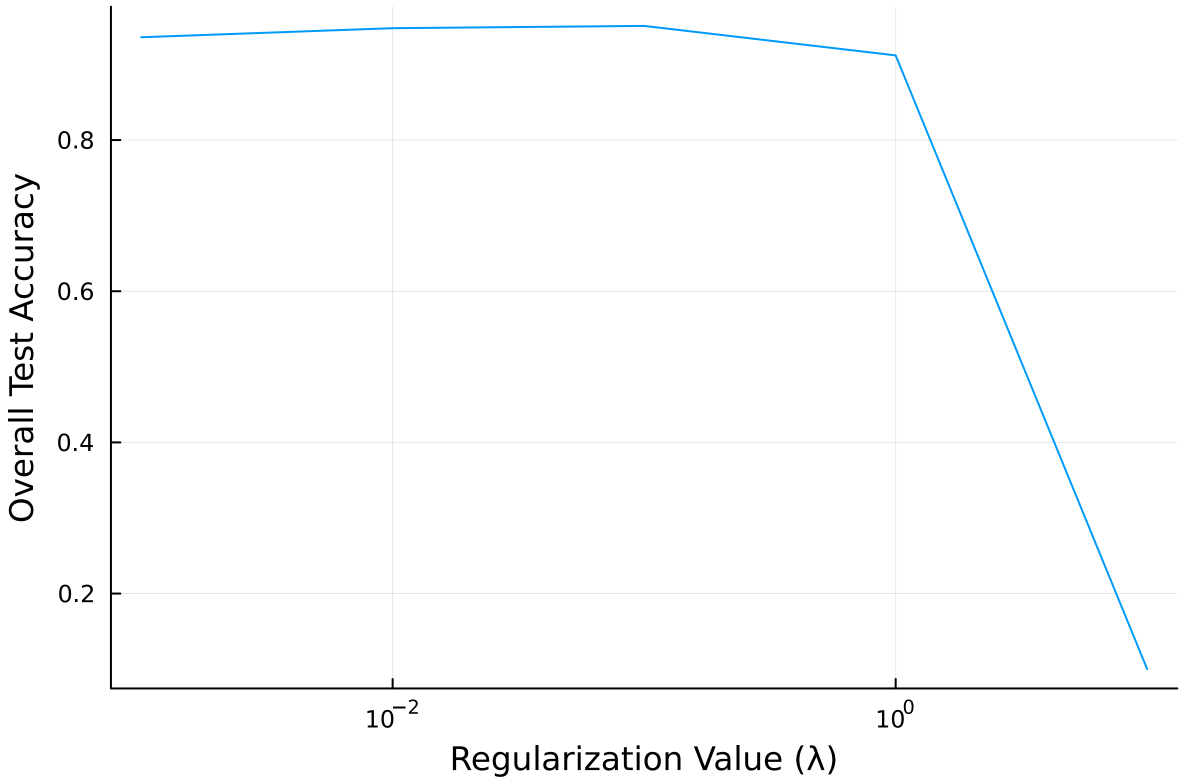
```
begin
   using Flux: Chain, Dense, σ, binarycrossentropy, gradient, Optimise, update!, params, onehotbatch, onecold, crossentropy, normalise
    using Flux
    using Printf
    using LinearAlgebra: norm
end
```

```
begin
# Preprocess data (normalize each image individually)
function preprocess_data(images, labels, digit)
   X = Float32.(images)  # Convert to Float32
   X = reshape(X, :, size(X, 3))
   X = [(x .- mean(x)) / std(x) for x in eachcol(X)]  # Normalize each image individually
   X = hcat(X...)
   y = (labels .== digit)
   return X, y
end

# Function to calculate confusion matrix
function calculate_confusion_matrix(predictions, actual)
   num_classes = 2
   confusion_matrix = zeros(Int, num_classes, num_classes)

   for i in 1:length(actual)
       true_class = Int(actual[i]) + 1
       pred_class = Int(predictions[i]) + 1
       confusion_matrix[true_class, pred_class] += 1
   end

   return confusion_matrix
end

# Define logistic regression model
function logistic_regression_model(input_size)
   return Chain(Dense(input_size, 1, σ))
end

# Define loss function (logistic loss + ridge regularization)
function loss_function(model, X, y, λ)
   ŷ = model(X)
   ŷ = vec(ŷ)  # Reshape ŷ to match the shape of y
   loss = Flux.binarycrossentropy(ŷ, y)
   reg = sum(norm(param, 2)^2 for param in params(model))
   return loss + λ * reg
end

# Define training function using gradient descent
function train_model!(model, X, y, λ, lr, epochs)
   opt = Flux.Optimise.Descent(lr)
   ps = params(model)
   dataset = Iterators.repeated((X, y), epochs)
   for (X_batch, y_batch) in dataset
       grads = Flux.gradient(() -> loss_function(model, X_batch, y_batch, λ), ps)
       Flux.Optimise.update!(opt, ps, grads)
   end
end

# Train classifiers for each digit
function train_classifiers(train_x, train_y, test_x, test_y, λ, lr, epochs)
   accuracies = []
```

```
    confusion_matrices = []
    for digit in 0:9  # Binary classification
        println("Training classifier for digit ", digit)
        X_train, y_train = preprocess_data(train_x, train_y, digit)
        X_test, y_test = preprocess_data(test_x, test_y, digit)
        model = logistic_regression_model(size(X_train, 1))
        train_model!(model, X_train, y_train, λ, lr, epochs)

        # Check if there are any weight parameters
        weights = params(model)
        if isempty(weights)
            println("No weight parameters found for classifier $digit")
            push!(accuracies, NaN)
            continue
        end

        ŷ = model(X_test)
        predictions = Flux.round.(Int, vec(ŷ))
        accuracy = mean(predictions .== y_test)
        push!(accuracies, accuracy)

        # Calculate and store confusion matrix
        confusion_matrix = calculate_confusion_matrix(predictions, y_test)
        push!(confusion_matrices, confusion_matrix)

        println("Accuracy for digit ", digit, ": ", accuracy)
        println("Confusion Matrix for digit ", digit, ":")
        println(confusion_matrix)
    end
    return accuracies, confusion_matrices
end

# Define parameters
samp_size = 60000
λ_vals = [0.001, 0.01, 0.1, 1.0, 10]
learning_rate = 0.1
epochs = 10

# Train classifiers
accuracies_per_lambda = []
confusion_matrices_per_lambda = []

for λ in λ_vals
    println("Lambda = ", λ)
    accuracies, confusion_matrices = train_classifiers(sampled_train_x,
sampled_train_y, sampled_test_x, sampled_test_y, λ, learning_rate, epochs)
    push!(accuracies_per_lambda, accuracies)
    push!(confusion_matrices_per_lambda, confusion_matrices)
end

# Display overall accuracy for all lambdas
println("Overall Accuracies for All Lambdas:")
for (λ, accs) in zip(λ_vals, accuracies_per_lambda)
    println("Lambda = ", λ, ": ", mean(accs))
```

```
end
```

```
Lambda = 0.001
Training classifier for digit 0
Accuracy for digit 0: 0.96
Confusion Matrix for digit 0:
[90 4; 0 6]
Training classifier for digit 1
Accuracy for digit 1: 0.98
Confusion Matrix for digit 1:
[87 1; 1 11]
Training classifier for digit 2
Accuracy for digit 2: 0.98
Confusion Matrix for digit 2:
[87 0; 2 11]
Training classifier for digit 3
Accuracy for digit 3: 0.88
Confusion Matrix for digit 3:
[85 2; 10 3]
Training classifier for digit 4
Accuracy for digit 4: 0.91
Confusion Matrix for digit 4:
[89 0; 9 2]
Training classifier for digit 5
Accuracy for digit 5: 0.9
Confusion Matrix for digit 5:
[85 3; 7 5]
Training classifier for digit 6
Accuracy for digit 6: 0.96
Confusion Matrix for digit 6:
[91 0; 4 5]
Training classifier for digit 7
Accuracy for digit 7: 0.95
Confusion Matrix for digit 7:
```




```
begin
# Plot overall test accuracy versus regularization value
plot(λ_vals, mean.(accuracies_per_lambda), xscale=:log10, xlabel="Regularization Value (λ)", ylabel="Overall Test Accuracy", label="")

end
```

InterruptException:

```
md"""
The general weight update equation for gradient ascent is the following:

>### wₜ₊₁ = wₜ + ηₜ∇L(f(x;w),y)
#

Refer to the uploaded file '40 Logistic Regression.pdf' for a derivation of the gradient of unregularized logistic regression




### (30 pts) section a: Regularized logistic regression
Derive the update equation for the regularized logistic regression by taking the gradient of the maximum log likelihood and present your derivation in a few sentences. (10 points)

Apply that classifier to the
MNIST data set. Experiment with the learning rate (ηₜ) to train your model. Don’t forget to add a column of 1s to your 784-length feature vector (the image values) so that you get the
bias term with your classifier. (20 points)

```

```
Use the given test-train split with no cross validation.



"""
```

InterruptException:

```
# I am uploading the derivation in another file please check it out
```

```julia
begin
    using Flux: ADAM
    using Random: seed!

    # Preprocess data (normalize each image individually)
    function new_preprocess_data(images, labels, digit)
        X = Float32.(images)  # Convert to Float32
        X = reshape(X, :, size(X, 3))
        X = [(x .- mean(x)) / std(x) for x in eachcol(X)]  # Normalize each image individually
        X = hcat(X...)
        X = vcat(X, ones(1, size(X, 2)))  # Add a column of 1s for bias
        y = (labels .== digit)
        return X, y
    end

    # Define logistic regression model with bias
    function new_logistic_regression_model(input_size)
        return Chain(Dense(input_size, 1, σ))
    end

    # Define loss function (logistic loss + ridge regularization)
    function new_loss_function(model, X, y, λ)
        ŷ = model(X)
        ŷ = vec(ŷ)
        loss = Flux.binarycrossentropy(ŷ, y)
        reg = sum(norm(param, 2)^2 for param in params(model))
        return loss + λ * reg  # New equation for regularization
    end

    # Define training function using gradient descent
    function new_train_model!(model, X, y, λ, lr, epochs)
        opt = ADAM(lr)
        ps = params(model)
        dataset = Iterators.repeated((X, y), epochs)
        for (X_batch, y_batch) in dataset
            grads = Flux.gradient(() -> new_loss_function(model, X_batch, y_batch, λ), ps)
            Flux.Optimise.update!(opt, ps, grads)
        end
    end


    # Train classifiers for each digit with bias
    function train_classifiers_with_bias(train_x, train_y, test_x, test_y, λ, lr, epochs)
        accuracies = []
        confusion_matrices = []
        for digit in 0:9  # Binary classification
            println("Training classifier for digit ", digit)
            X_train, y_train = new_preprocess_data(train_x, train_y, digit)
            X_test, y_test = new_preprocess_data(test_x, test_y, digit)
            model = new_logistic_regression_model(size(X_train, 1))
            new_train_model!(model, X_train, y_train, λ, lr, epochs)
```

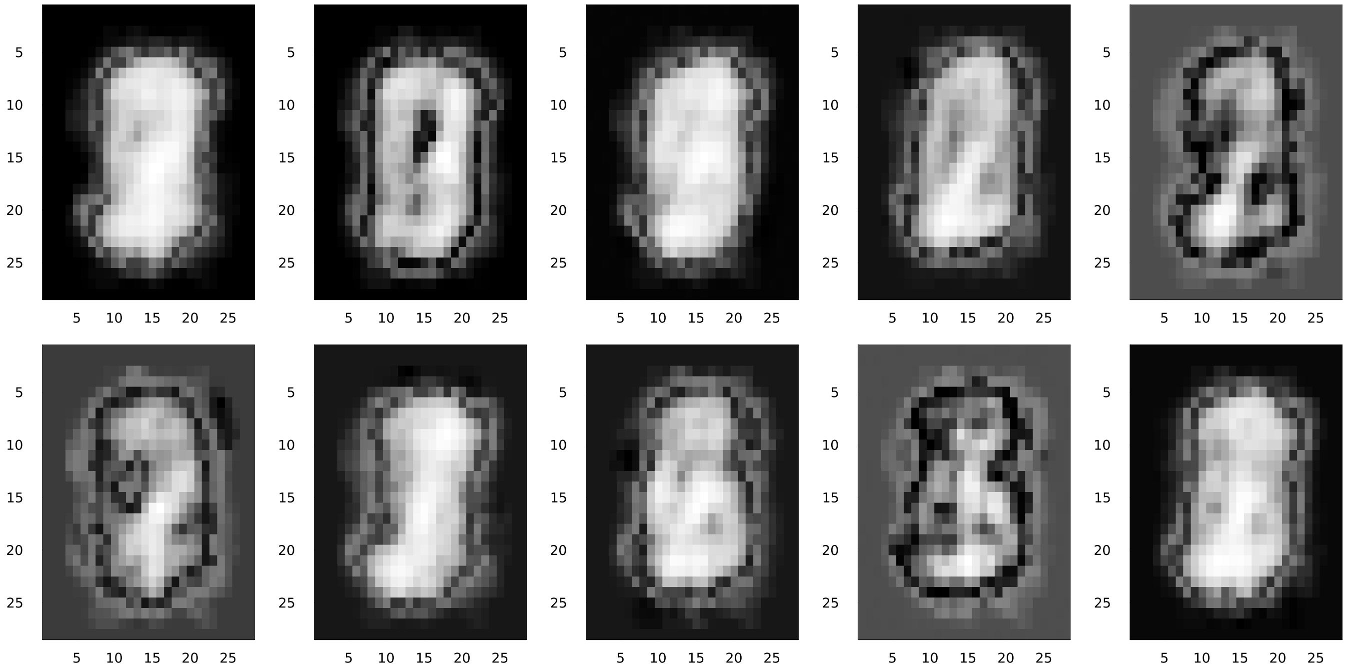
```julia

            # Check if there are any weight parameters
            weights = params(model)
            if isempty(weights)
                println("No weight parameters found for classifier $digit")
                push!(accuracies, NaN)
                continue
            end


            ŷ = model(X_test)
            predictions = Flux.round.(Int, vec(ŷ))
            accuracy = mean(predictions .== y_test)
            push!(accuracies, accuracy)

            # Calculate and store confusion matrix
            confusion_matrix = calculate_confusion_matrix(predictions, y_test)
            push!(confusion_matrices, confusion_matrix)

            println("Accuracy for digit ", digit, ": ", accuracy)
            println("Confusion Matrix for digit ", digit, ":")
            println(confusion_matrix)
        end
        return accuracies, confusion_matrices
    end

    # Define parameters

    λ_val =  0.001
    learning_rates = [0.001, 0.01, 0.1, 1.0, 10]


    # Train classifiers with bias
    new_accuracies_per_learning_rate = []
    new_weights_per_learning_rate = []
    for learning_rate in learning_rates
        println("learning_rate = ", learning_rate)
        accuracies, weights = train_classifiers_with_bias(sampled_train_x, sampled_train_y, sampled_test_x, sampled_test_y, λ_val, learning_rate, epochs)
        push!(new_accuracies_per_learning_rate, accuracies)
        push!(new_weights_per_learning_rate, weights)
    end

    # Display overall accuracy for all lambdas
    println("Overall Accuracies for All learning_rates:")
    for (learning_rates, accs) in zip(learning_rates, new_accuracies_per_learning_rate)
        println("learning_rates = ", learning_rates, ": ", mean(accs))
    end

```

```
Layer with Float32 parameters got Float64 input.
  The input will be converted, but any earlier layers may be very slow.
layer: Dense(785 => 1, σ)  # 786 parameters
summary(x): "785×100 Matrix{Float64}"
```

```
learning_rate = 0.001
Training classifier for digit 0
Accuracy for digit 0: 0.94
Confusion Matrix for digit 0:
[94 0; 6 0]
Training classifier for digit 1
Accuracy for digit 1: 0.86
Confusion Matrix for digit 1:
[85 3; 11 1]
Training classifier for digit 2
Accuracy for digit 2: 0.87
Confusion Matrix for digit 2:
[86 1; 12 1]
Training classifier for digit 3
Accuracy for digit 3: 0.87
Confusion Matrix for digit 3:
[87 0; 13 0]
Training classifier for digit 4
Accuracy for digit 4: 0.87
Confusion Matrix for digit 4:
[87 2; 11 0]
Training classifier for digit 5
Accuracy for digit 5: 0.86
Confusion Matrix for digit 5:
[86 2; 12 0]
Training classifier for digit 6
```

(10 pts) section b: Model analysis

For each of your 10 trained classifiers, show an image of the 784 weights (don't show the bias weight) as a 28 × 28 image. Do these images provide any insight to how this classifier works?



```
begin
    using Plots: heatmap


    # Preprocess data (normalize each image individually)
function nw_preprocess_data(images, labels, digit)
    X = Float32.(images)  # Convert to Float32
    X = reshape(X, :, size(X, 3))
    X = [(x .- mean(x)) / std(x) for x in eachcol(X)]  # Normalize each image individually
    X = hcat(X...)
    X = vcat(X, ones(1, size(X, 2)))
    y = (labels .== digit)
    return convert(Matrix{Float32}, X), y  # Convert X to Matrix{Float32}
end

    # Train classifiers for each digit with bias
    function weight_train_classifiers_with_bias(train_x, train_y, test_x, test_y, λ, lr, epochs)
        accuracies = []
        confusion_matrices = []
        weights_matrices = []  # Store weights matrices for visualization
        for digit in 0:9  # Binary classification
            X_train, y_train = nw_preprocess_data(train_x, train_y, digit)
            X_test, y_test = nw_preprocess_data(test_x, test_y, digit)
            model = new_logistic_regression_model(size(X_train, 1))
            new_train_model!(model, X_train, y_train, λ, lr, epochs)

            # Check if there are any weight parameters
            weights = params(model)
            if isempty(weights)
                println("No weight parameters found for classifier $digit")
                push!(accuracies, NaN)
                continue
            end
```

```

            ŷ = model(X_test)
            predictions = Flux.round.(Int, vec(ŷ))
            accuracy = mean(predictions .== y_test)
            push!(accuracies, accuracy)

            # Calculate and store confusion matrix
            confusion_matrix = calculate_confusion_matrix(predictions, y_test)
            push!(confusion_matrices, confusion_matrix)

            # Store weights for visualization, excluding bias weights
            push!(weights_matrices, weights[1])  # Exclude bias weights
        end
        return accuracies, confusion_matrices, weights_matrices
    end
    function plot_weights_per_digit(weights_matrices)
    plots = []
    for i in 1:10
        reshaped_data = reshape(weights_matrices[i][1, 1:784], 28, 28)
        heatmap_plot = heatmap(reshaped_data, c=:binary, yflip=true, colorbar=false)
        push!(plots, heatmap_plot)
    end
    plot(plots..., layout=(2, 5), size=(1200, 600))
end
    # Train classifiers and store weights for visualization
    nw_accuracies_per_learning_rate = []
    nw_weights_per_learning_rate = []
    for learning_rate in learning_rates
        accuracies, _, weights = weight_train_classifiers_with_bias(sampled_train_x, sampled_train_y, sampled_test_x, sampled_test_y, λ_val, learning_rate, epochs)
        push!(nw_accuracies_per_learning_rate, accuracies)
        push!(nw_weights_per_learning_rate, weights)
    end
plot_weights_per_digit(nw_weights_per_learning_rate[end])

end

```