Lab03: Logistic Regression.

- Student ID: 21127453
- Student name: Hoang Anh Tra

How to do your homework

You will work directly on this notebook; the word `TODO` indicate the parts you need to do.

You can discuss ideas with classmates as well as finding information from the internet, book, etc...; but *this homework must be your*.

How to submit your homework

- Before submitting, save this file as `<ID>.jl`. For example, if your ID is 123456, then your file will be `123456.jl`. And export to PDF with name `123456.pdf` then submit zipped source code and pdf into `123456.zip` onto Moodle.

> Danger
>
> **Note that you will get 0 point for the wrong submit**.

Contents:

- Logistic Regression.

1. Feature Extraction

Import Library

```julia
begin
    using  Distributions, Plots, Images, LinearAlgebra, Random
end
```

```
TaskLocalRNG()
```

```julia
Random.seed!(2024)
```

Load data

download_dataset (generic function with 2 methods)

```
# I DON'T KNOW WHAT YOUR OS, SO I ATTACHED DATA FOR YOU
# YOU DON'T NEED TO DOWNLOAD AND EXTRACT BY YOURSELF

# IF YOU WANT TO USE JULIA TO DOWNLOAD DATA, LET'S USE THESE CODE
# FOR EXTRACTING MNIST .gz FILE, PLEASE USE gzip

function download_dataset(save_path::String="data")
    # setup directory
    mkpath(joinpath(dirname(@__FILE__), save_path))
    data_dir = joinpath(dirname(@__FILE__), save_path)
    mkpath(joinpath(data_dir, "train"))
    train_dir =  joinpath(data_dir, "train")
    mkpath(joinpath(data_dir, "test"))
    test_dir = joinpath(data_dir, "test")

    # download dataset
    mkpath(joinpath(train_dir, "images"))
    download("http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz",joinpath(train_dir, "images/train-images-idx3-ubyte.gz"))
    train_images_file = joinpath(train_dir, "images/train-images-idx3-ubyte.gz")

    mkpath(joinpath(train_dir, "labels"))
    download("http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz",joinpath(train_dir, "labels/train-labels-idx1-ubyte.gz"))
    train_labels_file = joinpath(train_dir, "labels/train-labels-idx1-ubyte.gz")
Selection deleted
    mkpath(joinpath(test_dir, "images"))
    download("http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz",joinpath(test_dir, "images/t10k-images-idx3-ubyte.gz"))
    test_images_file = joinpath(test_dir, "images/t10k-images-idx3-ubyte.gz")

    mkpath(joinpath(test_dir, "labels"))
    download("http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz",joinpath(test_dir, "labels/t10k-labels-idx1-ubyte.gz"))
    test_labels_file = joinpath(test_dir, "labels/t10k-labels-idx1-ubyte.gz")
end
```

```
# If you have downloaded the dataset yet, please uncomment this line below and run this cell. Otherwise, keep it in uncomment state.
# download_dataset()
```

10000

```
begin
    data_dir = joinpath(dirname(@__FILE__), "data")
    train_x_dir = joinpath(data_dir, "train/images/train-images.idx3-ubyte")
    train_y_dir = joinpath(data_dir, "train/labels/train-labels.idx1-ubyte")

    test_x_dir = joinpath(data_dir, "test/images/t10k-images.idx3-ubyte")
    test_y_dir = joinpath(data_dir, "test/labels/t10k-labels.idx1-ubyte")

    NUMBER_TRAIN_SAMPLES = 60000
    NUMBER_TEST_SAMPLES = 10000
end
```

```
60000×784 adjoint(::Matrix{Float64}) with eltype Float64:
 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     0.0  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  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  0.0  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  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  0.0  0.0  0.0  0.0  0.0  0.0
```

```julia
begin
    # Init arrays
    train_x = Array{Float64}(undef, 28^2, NUMBER_TRAIN_SAMPLES)
    train_y = Array{Int64}(undef, NUMBER_TRAIN_SAMPLES)

    # Init io streams
    io_images = open(train_x_dir)
    io_labels = open(train_y_dir)

    # Iterating through sample length
    for i ∈ 1:NUMBER_TRAIN_SAMPLES
        seek(io_images, (i-1)*28^2 + 16) # offset 16 to skip header
        seek(io_labels, (i-1)*1 + 8) # offset 8 to skip header
        train_x[:,i] = convert(Array{Float64}, read(io_images, 28^2))
        train_y[i] = convert(Int, read(io_labels, UInt8))
    end

    # Close io streams
    close(io_images)
    close(io_labels)

    # Transpose features
    train_x = train_x'
end
```

Selection deleted

```
10000×784 adjoint(::Matrix{Float64}) with eltype Float64:
 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     0.0  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  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  0.0  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  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  0.0  0.0  0.0  0.0  0.0  0.0
```

```julia
begin
    # Init arrays
    test_x = Array{Float64}(undef, 28^2, NUMBER_TEST_SAMPLES)
    test_y = Array{Int64}(undef, NUMBER_TEST_SAMPLES)

    # Init io streams
    io_images_test = open(test_x_dir)
    io_labels_test = open(test_y_dir)

    # Iterating through sample length
    for i ∈ 1:NUMBER_TEST_SAMPLES
        seek(io_images_test, (i-1)*28^2 + 16) # offset 16 to skip header
        seek(io_labels_test, (i-1)*1 + 8) # offset 8 to skip header
        test_x[:,i] = convert(Array{Float64}, read(io_images_test, 28^2))
        test_y[i] = convert(Int, read(io_labels_test, UInt8))
    end

    # Close io streams
    close(io_images)
    close(io_labels)

    # Transpose features
    test_x = test_x'
end
```

```
((60000, 784), (60000), (10000, 784), (10000))
```

```julia
size(train_x), size(train_y), size(test_x), size(test_y)
```

Extract Features

So we basically have 70000 samples with each sample having 784 features - pixels in this case and a label - the digit the image represent.

Let's play around and see if we can extract any features from the pixels that can be more informative. First I'd like to know more about average intensity - that is the average value of a pixel in an image for the different digits

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

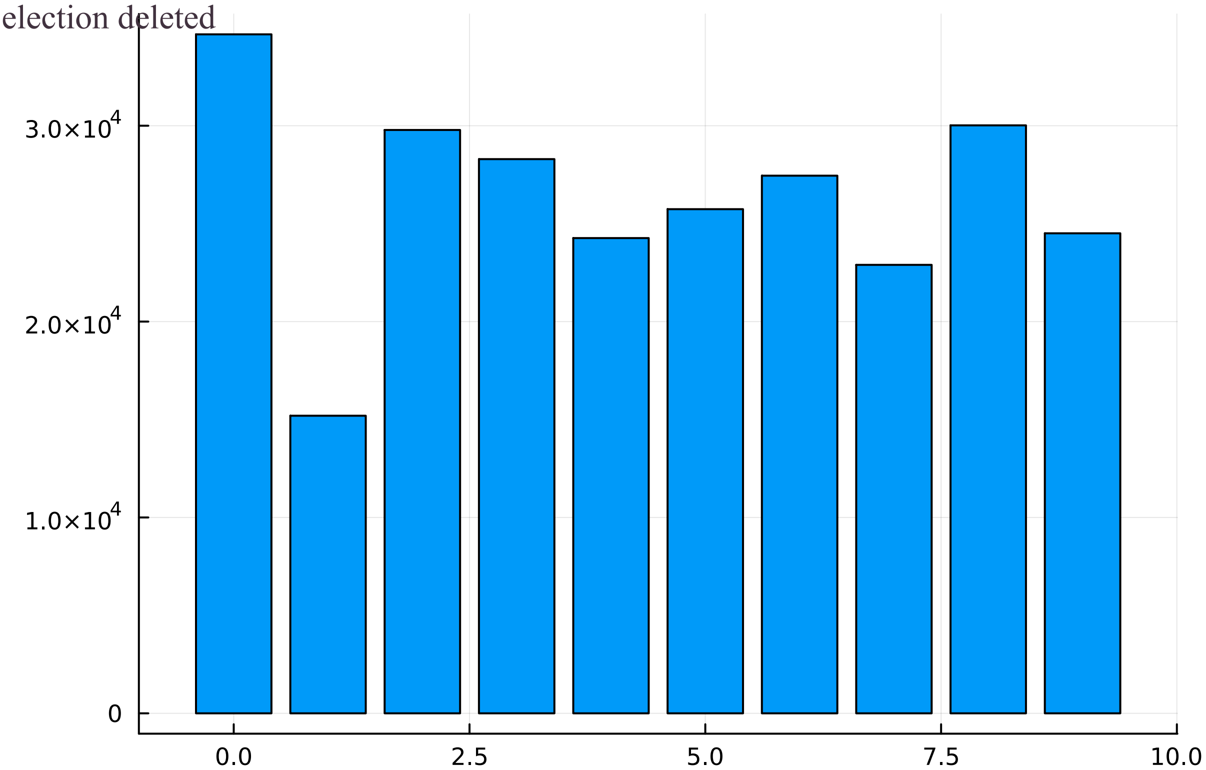
```julia
#TODO compute average intensity for each label
function compute_average_intensity(x, y)
    mean_ = zeros(10) # 10 is number of labels
    #TODO compute average intensity for each label
    num_labels = 10  # Assuming there are 10 different digits (0 to 9)
    sum_pixel_values = zeros(Float64, num_labels)
    sample_counts = zeros(Int, num_labels)
    for i in 1:size(x, 1)
        label = y[i] + 1
        sum_pixel_values[label] += sum(x[i, :])
        sample_counts[label] += 1
    end
    mean_ = sum_pixel_values ./ sample_counts

    return mean_
end
```

```
l_mean =
 [34666.0, 15193.6, 29783.1, 28294.7, 24263.4, 25739.6, 27449.4, 22896.4, 30019.2, 24508.2
```

```julia
l_mean = compute_average_intensity(train_x, train_y)
```

Plot the average intensity using matplotlib

Selection deleted




```julia
bar(0:9, l_mean, legend=false)
```

```
(60000, 1)
```

```julia
begin
    #TODO compute average intensity for each data sample
    intensity = mean(train_x, dims=2)
    size(intensity)
end
```

Some digits are symmetric (1, 3, 8, 0) some are not (2, 4, 5, 6, 9). Creating a new feature capturing this could be useful. Specifically, we calculate $s = -\frac{s_1+s_2}{2}$ for each image:

- s_1

  : flip the image along y-axis and compute the mean value of result

- s_2

  : flip the image along x-axis and compute the mean value of result

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

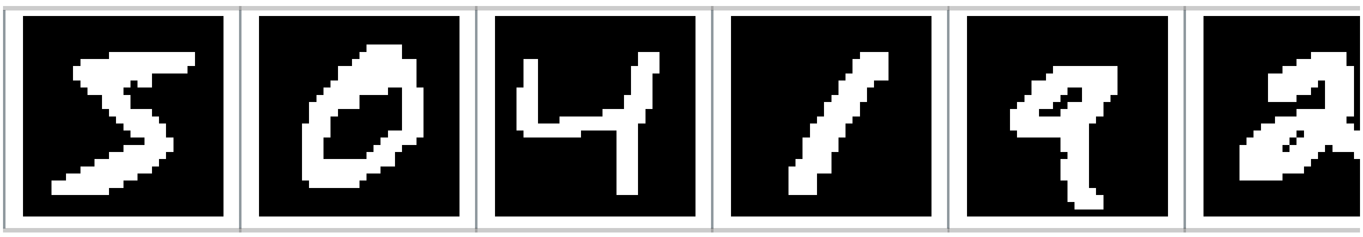
```
function compute_symmetry(train_x)
    symmetry = []
    for i in 1:size(train_x)[1]
        img = reshape(train_x[i,:], (28,28))
        s1 = mean(abs.(img - reverse(img, dims=1)))
        s2 = mean(abs.(img - reverse(img, dims=2)))
        s = -0.5 .* (s1 + s2)
        append!(symmetry, s)
    end
    return symmetry
end
```

Selection deleted

```
(60000)
```

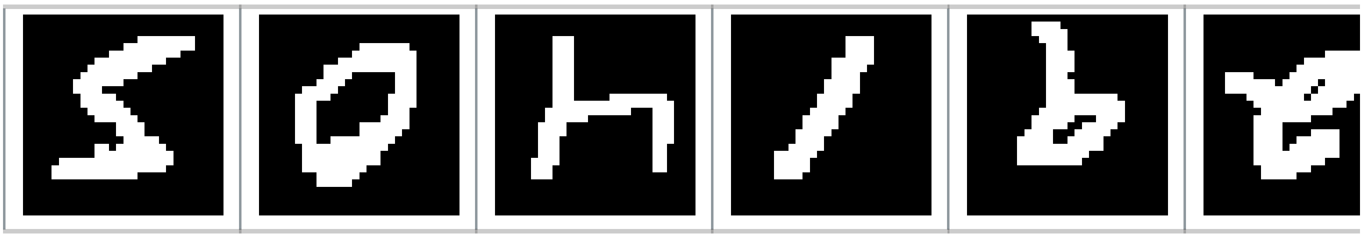
```
begin
    symmetry = compute_symmetry(train_x)
    size(symmetry)
end
```

Visualize 10 samples in order to illustrate symmetry



(a vector displayed as a row to save space)

```
begin
    num_img = 10
    img_flat = train_x[1:num_img,:]
    img = [reshape(img_flat[i,:], (28,28))' for i in 1:num_img]
    [colorview(Gray, Float32.(img[i])) for i in 1:num_img]
end
```



(a vector displayed as a row to save space)

```
begin
    img_reverse_flat = reverse(img_flat, dims=2)
    img_reverse = [reshape(img_reverse_flat[i,:], (28,28))' for i in 1:num_img]
    [colorview(Gray, Float32.(img_reverse[i])) for i in 1:num_img]
end
```

Our new data will have 70000 samples and 2 features: intensity, symmetry.

```
(60000, 2)
```

```
begin
    #TODO create X_new by horizontal stack intensity and symmetry
    train_x_new = hcat(intensity, symmetry)
    size(train_x_new)
end
```

Selection deleted

2. Training

Usually logistic regression is a good first choice for classification. In this homework we use logistic regression for classifying digit 1 images and not digit 1's images.

Normalize data

First normalize data using Z-score normalization

- **TODO: Study about Z-score normalization**
- **TODO: Why should we normalize data?**

Z-score normalization, also known as standardization, is a technique used in statistics to transform data into a standard normal distribution. In the context of machine learning and data analysis, it is commonly employed to normalize features, making them more comparable and aiding in the convergence of certain optimization algorithms.

The Z-score of a data point measures how many standard deviations it is from the mean. The formula for Z-score normalization is: $z = \frac{x-\mu}{\sigma}$

where:

- $$z$$
  is the standardized value (Z-score),
- $$x$$
  is the original value,
- $$\mu$$
  is the mean of the feature,
- $$\sigma$$
  is the standard deviation of the feature.

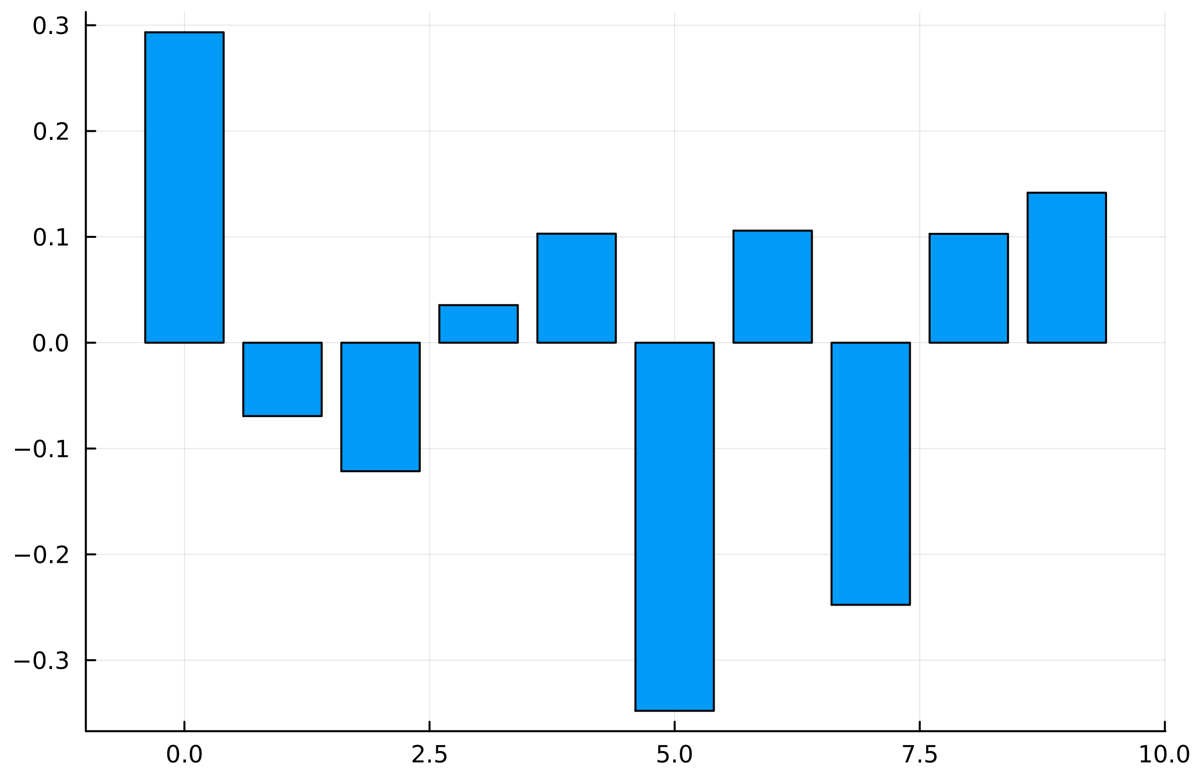
Here are some reasons why normalization is often performed on data:

1. **Scale Independence:** Machine learning algorithms often rely on measures of distance between data points. If the features have different scales, the algorithm might give more weight to features with larger magnitudes. Normalizing the data ensures that all features have the same scale.
2. **Convergence:** For algorithms that use optimization techniques (e.g., gradient descent), normalization helps the algorithm converge faster because it prevents one feature from dominating the learning process.
3. **Interpretability:** Normalization can make the interpretation of coefficients or feature importance more straightforward. It ensures that coefficients represent the relative importance of features rather than being influenced by their scales.
4. **Numerical Stability:** Some algorithms, like SVMs or neural networks, may be numerically unstable if the input features have large differences in scale. Normalization can help mitigate

this issue.

normalize (generic function with 3 methods)

```
function normalize(x, mean_=nothing, std_=nothing)
   if mean_ == nothing && std_ == nothing
        #TODO normalize x_train
        #return 'normalized_train_x', 'mean_', 'std_'
        #mean_ and std_ will be re-used to pre-process test set
        mean_ = mean(x, dims=1)
        std_ = std(x, dims=1)
    end
    # Z-score normalization
    normalized_x = (x .- mean_) ./ std_

    return normalized_x, mean_, std_ #return 'normalized_train_x' calculated by using mean_ and std_ (both of them are passed and not null)
end


```




```
begin
    normalized_train_x, mean_, std_ = normalize(train_x_new)

    s_mean = compute_average_intensity(normalized_train_x, train_y)
    bar(0:9, s_mean, legend=false)
end
```

Construct data

(60000, 1)

```
begin
    train_y_new = reshape(deepcopy(train_y), (size(train_y)[1], 1))
    train_y_new[train_y_new .!= 1] .= 0
    size(train_y_new)
end
```

(60000, 3)

```
begin
    # contruct data by adding ones
    add_one_train_x = hcat(ones(size(normalized_train_x)[1],), normalized_train_x)
    size(add_one_train_x)
end
```

Sigmoid function and derivative of the sigmoid function

sigmoid_activation (generic function with 1 method)

```
function sigmoid_activation(x)
    #TODO
    """compute the sigmoid activation value for a given input"""
    #return?
    activation_value = 1.0 ./ (1.0 .+ exp.(-x))
    return activation_value
end
```

sigmoid_deriv (generic function with 1 method)

```
function sigmoid_deriv(x)
    #TODO
    """
    Compute the derivative of the sigmoid function ASSUMING
    that the input 'x' has already been passed through the sigmoid
    activation function
    """
    #return?
    deriv = x .* (1.0 .- x)

    return deriv
end
```

Compute output

compute_h (generic function with 1 method)

```
function compute_h(W, X)
    #TODO
    """
    Compute output: Take the inner product between our features 'X' and the weight
    matrix 'W'
    """
    return X * W
end
```

predict (generic function with 1 method)

```
function predict(W, X)
    #TODO
    """
    Take the inner product between our features and weight matrix,
    then pass this value through our sigmoid activation
    """
    preds = sigmoid_activation(compute_h(W, X))

    # apply a step function to threshold the outputs to binary
    # class labels
    preds[preds .<= 0.5] .= 0
    preds[preds .> 0] .= 1

    return preds
end
```

Compute gradient

Loss Function: Average negative log likelihood

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N} -\left(y^i \ln h_{\mathbf{w}}\left(\mathbf{x}^i\right) + \left(1-y^i\right)\ln\left(1-h_{\mathbf{w}}\left(x^i\right)\right)\right)$$

$$\text{Sigmoid Activation: } z = \sigma(h) = \frac{1}{1+e^{-h}}$$

$$\text{Cross-entropy: } J(w) = -\left(ylog(z) + (1-y)log(1-z)\right)$$

$$\text{Chain rule: } \frac{\partial J(w)}{\partial w} = \frac{\partial J(w)}{\partial z}\frac{\partial z}{\partial h}\frac{\partial h}{\partial w}$$

$$\frac{\partial J(w)}{\partial z} = -\left(\frac{y}{z} - \frac{1-y}{1-z}\right) = \frac{z-y}{z(1-z)}$$

$$\frac{\partial z}{\partial h} = z(1-z)$$

$$\frac{\partial h}{\partial w} = X$$

$$\frac{\partial J(w)}{\partial w} = X^T(z-y)$$

compute_gradient (generic function with 1 method)

```
function compute_gradient(error, train_x)
    #TODO
    """
    This is the gradient descent update of "average negative loglikelihood" loss function.
    In lab02 our loss function is "sum squared error".
    """
    return transpose(train_x) * error / size(train_x, 1)
end
```

train (generic function with 1 method)

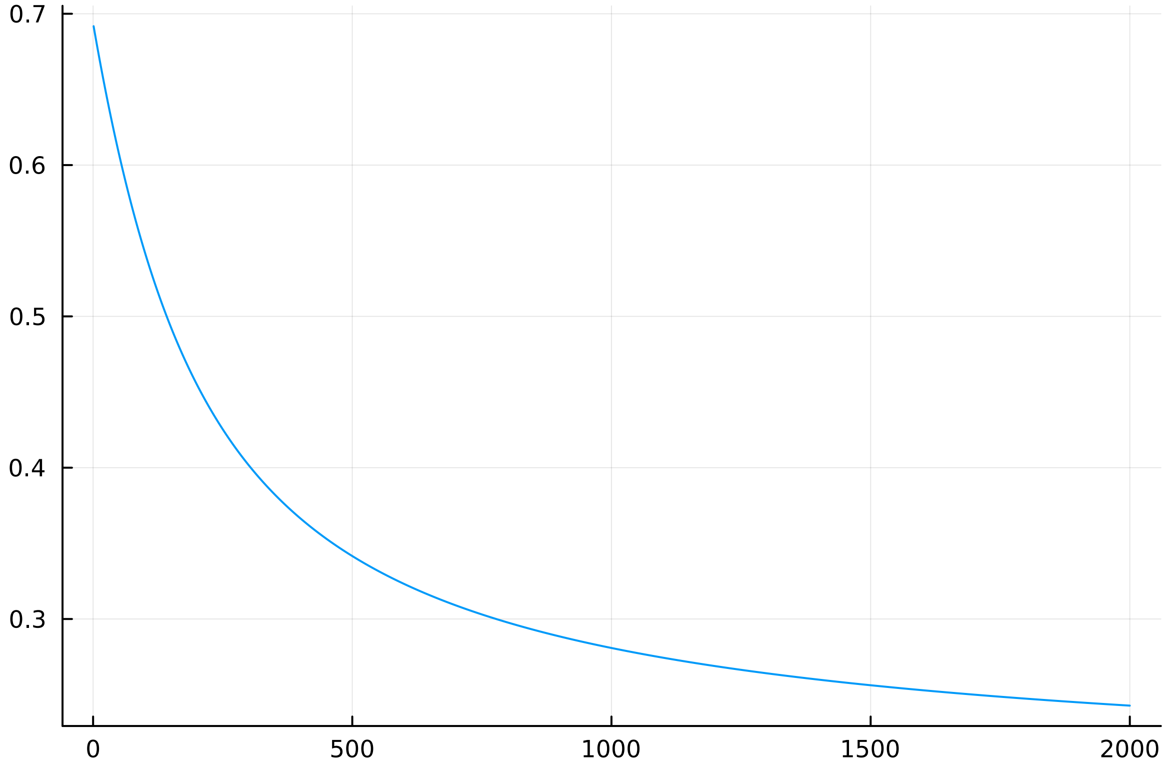
```
function train(W, train_x, train_y, learning_rate, num_epochs)
    losses = []
    for epoch in 1:num_epochs
        y_hat = sigmoid_activation(compute_h(W, train_x))
        error = y_hat - train_y
        append!(losses, mean(-1 .* train_y .* log.(y_hat) .- (1 .- train_y) .* log.(1 .- y_hat)))
        grad = compute_gradient(error, train_x)
        W -= learning_rate * grad

        if epoch == 1 || epoch % 50 == 0
            print("Epoch=$epoch; Loss=$(losses[end])\n")
        end
    end
    return W, losses
end
```

Train our model

```
(3×1 Matrix{Float64}:, [0.691785, 0.689814, 0.687854, 0.685905, 0.683967, 0.68204, 0.680
 -2.30281
```

```julia
begin
    W = rand(Normal(), (size(add_one_train_x)[2], 1))

    num_epochs=2000
    learning_rate=0.01
    W, losses = train(W, add_one_train_x, train_y_new, learning_rate, num_epochs)
end
```

```
Epoch=1; Loss=0.6917852943075712
Epoch=50; Loss=0.6072488211905173
Epoch=100; Loss=0.5422193706368078
Epoch=150; Loss=0.4929810905763288
Epoch=200; Loss=0.45511847467171035
Epoch=250; Loss=0.42544748653743736
Epoch=300; Loss=0.40173896687772676
Epoch=350; Loss=0.3824411618658025
Epoch=400; Loss=0.36646624314799875
Epoch=450; Loss=0.3530412981950098
Epoch=500; Loss=0.34160813925024985
Epoch=550; Loss=0.3317566326660123
Epoch=600; Loss=0.32318020842468376
Epoch=650; Loss=0.31564588897209556
Epoch=700; Loss=0.30897384023686353
Epoch=750; Loss=0.3030232191041429
Epoch=800; Loss=0.29768223213039274
Epoch=850; Loss=0.29286104753078307
Epoch=900; Loss=0.2884866668303988
Epoch=950; Loss=0.28449916115212487
Epoch=1000; Loss=0.28084887097584443
Epoch=1050; Loss=0.2774942954513533
Epoch=1100; Loss=0.2744004818232623
Epoch=1150; Loss=0.27153778225936315
Epoch=1200; Loss=0.26888088392656
Epoch=1250; Loss=0.2664080446720284
Epoch=1300; Loss=0.2641004851227947
Epoch=1350; Loss=0.2619419010189548
Epoch=1400; Loss=0.25991806886462937
Epoch=1450; Loss=0.2580165246644947
Epoch=1500; Loss=0.25622630038663263
Epoch=1550; Loss=0.2545377063825633
Epoch=1600; Loss=0.25294215066682085
Epoch=1650; Loss=0.25143198796519967
```




```
plot(1:num_epochs, losses, legend=false)
```

3. Evaluate our model

In this section, you will evaluate your model on train set and test set and make some comment about the result.

Evaluate model on training set

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

```
function tpfptnfn_cal(y_test, y_pred, positive_class=1)
    true_positives = 0
    false_positives = 0
    true_negatives = 0
    false_negatives = 0

    # Calculate true positives, false positives, false negatives, and true negatives
    for (true_label, predicted_label) in zip(y_test, y_pred)
        if true_label == positive_class && predicted_label == positive_class
            true_positives += 1
        elseif true_label != positive_class && predicted_label == positive_class
            false_positives += 1
        elseif true_label == positive_class && predicted_label != positive_class
            false_negatives += 1
        elseif true_label != positive_class && predicted_label != positive_class
            true_negatives += 1
        end
    end

    return true_positives, false_positives, true_negatives, false_negatives
end
```

```
begin
    preds_train = predict(W, add_one_train_x)
    train_y_n = reshape(train_y_new, length(train_y_new), 1)

    acc = 0
    precision = 0
    recall = 0
    f1 = 0

    for i ∈ 1:10
        # Calculate true positives, false positives, false negatives, and true negatives
        true_positives, false_positives, true_negatives, false_negatives = tpfptnfn_cal(train_y_n, preds_train)

        # Calculate precision, recall, and F1-score
        acc += (true_positives + true_negatives) / (true_positives + false_positives + true_negatives + false_negatives)
        precision += true_positives / (true_positives + false_positives)
        recall += true_positives / (true_positives + false_negatives)
    end

    acc = acc / 10
    precision = precision / 10
    recall = recall / 10
    f1 = 2 * precision * recall / (precision + recall)
    print(" acc: $acc\n precision: $precision\n recall: $recall\n f1_score: $f1\n")
end
```

```
acc: 0.9247833333333333
precision: 0.8906414300736067
recall: 0.37689113022841886
f1_score: 0.5296508598228244
```

Evaluate model on test set

In order to predict the result on test set, you have to perform data pre-process first. The pre-process is done exactly what we have done on train set. That means, you have to:

- Change the label in `test_y` to `0` and `1` and store in a new variable named `test_y_new`
- Calculate `test_intensity` and `test_symmetry` to form `test_x_new` (the shape should be `(10000,2))`
- Normalized `test_x_new` by z-score. Note the you will re-use variable `mean_` and `std_` to calculate `test_x_new` instead of compute new ones. You will store the result in `normalized_test_x`
- Add a column that's full of one to `test_x_new` and store in `add_one_test_x` (the shape should be `(10000,3))`

```
(10000, 3)
```

```julia
begin
    #TODO
    # compute test_y_new
    test_y_new = copy(test_y)
    test_y_new[test_y .== 0] .= 0
    test_y_new[test_y .> 0] .= 1


    # compute test_intensity and test_symmetry to form test_x_new
    test_intensity = mean(test_x, dims=2)
    test_symmetry = compute_symmetry(test_x)
    test_x_new = hcat(test_intensity, test_symmetry)

    # normalize test_x_new to form normalized_test_x
    normalized_test_x = normalize(test_x_new, mean_, std_)[1]
    add_one_test_x = hcat(ones(size(test_x_new)[1],), normalized_test_x)
    size(add_one_test_x)
end
```

After doing all these stuffs, you now can predict and evaluate your model

```julia
begin
    preds_test = predict(W, add_one_test_x)

    test_y_n = reshape(test_y_new, length(test_y_new), 1)

    _acc = 0
    _p = 0
    _r = 0
    _f1 = 0

    for i ∈ 1:10
        # Calculate true positives, false positives, false negatives, and true negatives
        tp, fp, tn, fn = tpfptnfn_cal(test_y_n, preds_test)

        # Calculate precision, recall, and F1-score
        _acc += (tp + tn) / (tp + fp + tn + fn)
        _p += tp / (tp + fp)
        _r += tp / (tp + fn)
    end

    _acc = _acc / 10
    _p = _p / 10
    _r = _r / 10
    _f1 = 2 * _p * _r / (_p + _r)

    print(" acc: $_acc\n precision: $_p\n recall: $_r\n f1_score: $_f1\n")
end
```

```
acc: 0.1491
precision: 0.996116504854369
recall: 0.05687361419068736
f1_score: 0.10760356581017304
```

TODO: Comment on the result

Accuracy (0.1491): Accuracy is the proportion of correctly classified instances out of the total instances. An accuracy of 0.1491 indicates that the model is correctly classifying approximately 14.91% of the instances. While this is better than random guessing, it may still suggest that the model is not performing well overall.

Precision (0.9961): Precision is the ratio of correctly predicted positive observations to the total predicted positives. A precision of 0.9961 is very high, indicating that when the model predicts the positive class, it is almost always correct. This suggests that the model is very selective when predicting the positive class.

Recall (0.0569): Recall, also known as sensitivity or true positive rate, is the ratio of correctly predicted positive observations to all observations in the actual class. A recall of 0.0569 indicates that the model is identifying only a small fraction of the actual positive instances. It suggests a high number of false negatives.

F1 Score (0.1076): The F1 score is the weighted average of precision and recall. It is a metric that considers both false positives and false negatives. An F1 score of 0.1076 is relatively low, indicating a trade-off between precision and recall. The model is struggling to balance precision and recall effectively.

Comments: Accuracy Concerns: Overall accuracy is quite low, indicating potential model enhancements.

Precision Strength: The model excels in accurately predicting positive cases.

Recall Challenge: Struggles to identify all positive instances, leading to a low recall.

F1 Score Balance: Shows a balance between precision and recall, emphasizing the need for improvement in both areas.

In summary, the model's precision is commendable, but addressing recall issues could significantly enhance its overall performance.