Table of Contents

Lab 02: Gradient Descent

1. Loss landscape
2. The "Gradient" in Gradient Descent

3. Forward & Backward

3.1. Forward
3.2. Backward
4. Implementation
4.1. Import library
4.2. Create data
4.3. Training

Lab 02: Gradient Descent

- Student name: Châu Tấn Kiệt
- ID: 21127329

How to do your homework

- You will work directly on this notebook; the word **TODO** indicates the parts you need to do.
- You can discuss the ideas as well as refer to the documents, but *the code and work must be yours*.

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.

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

Content of the assignment:

- Gradient Descent

1. Loss landscape

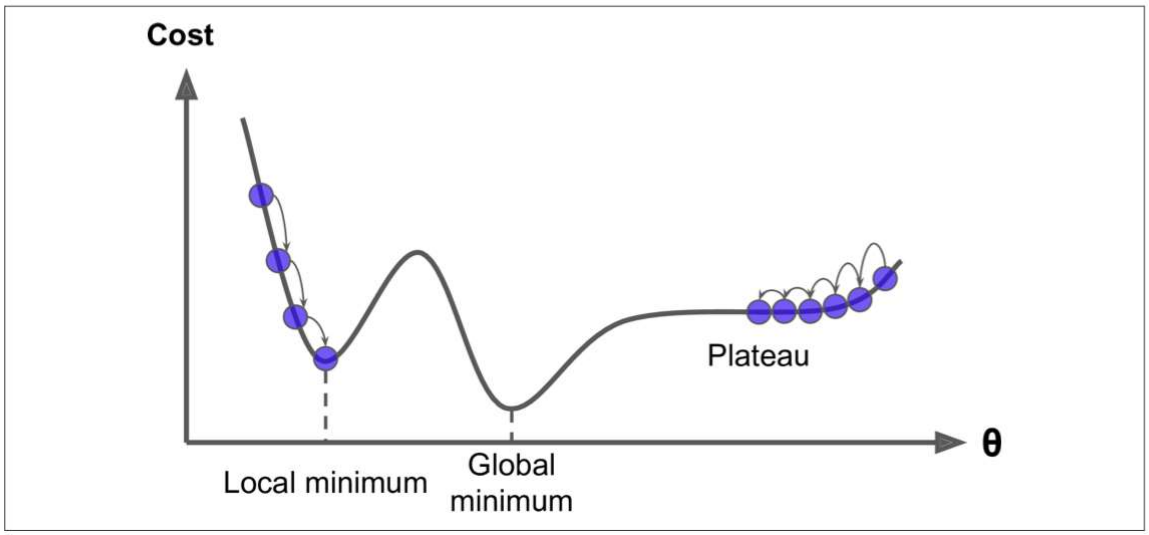


Figure 1. Loss landscape visualized as a 2D plot. Source: codecamp.vn

The gradient descent method is an iterative optimization algorithm that operates over a loss landscape (also called an optimization surface).As we can see, our loss landscape has many peaks and valleys based on which values our parameters take on. Each peak is a local maximum that represents very high regions of loss – the local maximum with the largest loss across the entire loss landscape is the global maximum. Similarly, we also have local minimum which represents many small regions of loss. The local minimum with the smallest loss across the loss landscape is our global minimum. In an ideal world, we would like to find this global minimum, ensuring our parameters take on the most optimal possible values.

Each position along the surface of the corresponds to a particular loss value given a set of parameters $\mathbf{W}$ (weight matrix) and $\mathbf{b}$ (bias vector). Our goal is to try different values of $\mathbf{W}$ and $\mathbf{b}$, evaluate their loss, and then take a step towards more optimal values that (ideally) have lower loss.

2. The "Gradient" in Gradient Descent

We can use $\mathbf{W}$ and $\mathbf{b}$ and to compute a loss function L or we are able to find our relative position on the loss landscape, but **which direction** we should take a step to move closer to the minimum.

- All We need to do is follow the slope of the gradient $\nabla_{\mathbf{W}}$. We can compute the gradient $\nabla_{\mathbf{W}}$ across all dimensions using the following equation:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- But, this equation has 2 problems:
    - 1. It's an **approximation** to the gradient.
    - 2. It's painfully slow.

In practice, we use the **analytic gradient** instead.

3. Forward & Backward

In this section, you will be asked to fill in the black to form the forward process and backward process with the data defined as follows:

- Feature: X (shape: $n \times d$, be already used bias trick)
- Label: y (shape: $n \times 1$)
- Weight: W (shape: $d \times 1$)

3.1. Forward

TODO: Consider one sample $\mathbf{x}_i$. Fill in the blank

$$h_i = \mathbf{x}_i^T W \Rightarrow \frac{\partial h_i}{\partial W} = \mathbf{x}_i^T$$

$$\hat{y}_i = \sigma(h_i) \Rightarrow \frac{\partial \hat{y}_i}{\partial h_i} = \sigma(h_i)(1 - \sigma(h_i))$$

$$loss_i = (\hat{y}_i - y_i)^2 \Rightarrow \frac{\partial loss_i}{\partial \hat{y}_i} = 2(\hat{y}_i - y_i)$$

3.2. Backward

Our loss function is MSE:

$$Loss = \frac{1}{n}\sum_{i=1}^{n} loss_i = \frac{1}{n}\sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

Goal: Compute $\nabla Loss = \frac{\partial Loss(W)}{\partial W}$

How to compute $\nabla Loss$?: Use Chain-rule. Your work is to fill in the blank

TODO: Fill in the blank

$$\nabla Loss = \frac{\partial Loss(W)}{\partial W} = \frac{1}{n}\sum_{i=1}^{n} \frac{\partial loss_i}{\partial W_i}$$

$$= \frac{1}{n}\sum_{i=1}^{n} \frac{\partial loss_i}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial h_i} \frac{\partial h_i}{\partial W}$$

$$= \frac{2}{n}\sum_{i=1}^{n} (\hat{y}_i - y_i)\sigma(h_i)(1 - \sigma(h_i))\mathbf{x}_i^T$$

4. Implementation

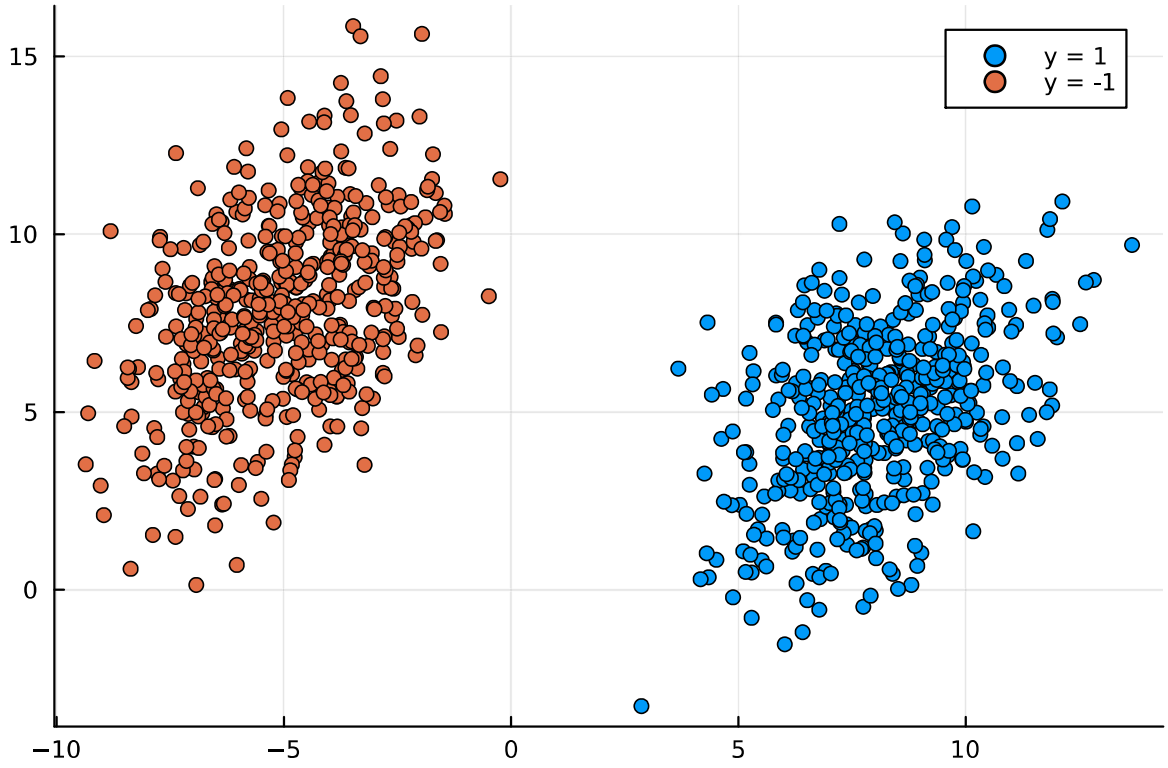
4.1. Import library

```
using Distributions, Plots, LinearAlgebra, Random
```

```
MersenneTwister(2024)
```

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

4.2. Create data




```
begin
    # DOT NOT MODIFY THIS CODE
    # generate a 2-class classification problem with 1,000 data points, each data point is a 2D feature vector
    # number of data points
    n = 1000

    # dimensionality of data
    d = 2

    # mean
    μ = 5

    # variance
    Σ = 8

    # Generate two class for synthesized data
    positive = rand(MvNormal([Σ, μ], 3 .* [1 (μ - d)/μ; (μ - d)/μ d]), n ÷ 2)
    negative = rand(MvNormal([-μ, Σ], 3 .* [1 (μ - d)/μ; (μ - d)/μ d]), n ÷ 2)

    # Combine two class of generated data.
    # X = features
    # y = label
    X = hcat(positive, negative)
    y = vcat(ones(n ÷ 2) .- 1, ones(n ÷ 2))'

    # Visualization
    plt = scatter(positive[1, :], positive[2, :], label="y = 1")
    scatter!(plt, negative[1, :], negative[2, :], label="y = -1")
    # DOT NOT MODIFY THIS CODE
end
```

```
4×1000 Matrix{Float64}:
 7.00921  5.40614  7.77031  10.2313   …  -0.235459  -2.19833  -4.99492  -5.55589
 4.07511  1.63066  9.29032   8.68727     11.5486    10.9082    7.40612   8.02935
 1.0      1.0      1.0       1.0          1.0        1.0       1.0       1.0
 0.0      0.0      0.0       0.0          1.0        1.0       1.0       1.0
```

```
begin
    # insert a column of 1's as the last entry in the feature matrix
    # -- allows us to treat the bias as a trainable parameter

    X_aug = vcat(X, ones(n)')
    data = vcat(X_aug, y)
end
```

```
(700×3 Matrix{Float64}:  , [0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 1.0, 1.0,    more ,0.0], 3
   7.05549   4.70516  1.0
  -6.87358   3.98823  1.0
  -2.05821   6.87055  1.0
  -5.71526   7.10931  1.0
  -7.14349   3.62296  1.0
   8.85531   6.33639  1.0
  10.045     5.23048  1.0
   ⋮
  -5.93771   6.71143  1.0
  -5.38006   8.4446   1.0
  -4.582    10.4695   1.0
  -4.57232   7.8681   1.0
   9.01379   5.11398  1.0
   7.12285   3.71975  1.0
```

```
begin
    # DOT NOT MODIFY THIS CODE
    # Split data, use 50% of the data for training and the remaining 50% for testing
    # Prepare data
    D = data'[shuffle(1:end), :]

    # Calculate the number of samples for each split
    n_train = Int(n * 0.7)

    # Split the samples into train, and test sets
    train_data = D[begin:n_train, :]
    test_data = D[n_train + 1: end, :]
    println(size(train_data), size(test_data))

    # Move samples to train-test features and labels
    X_train, y_train, X_test, y_test = train_data[:,1:3], train_data[:,4],
    test_data[:,1:3], test_data[:,4]
    # DOT NOT MODIFY THIS CODE
end
```

```
(700, 4)(300, 4)
```

4.3. Training

Sigmoid function and derivative of the sigmoid function

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

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

    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?
        return x * (1.0 - x)
    end
end
```

Compute output

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

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

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

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

        return preds
    end
end
```

Compute gradient

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

```
begin
    function compute_gradient(error, y_hat, trainX)
        #TODO
        """
        the gradient descent update is the dot product between our
        features and the error of the sigmoid derivative of
        our predictions
        """
        # return?
        return trainX' * error
    end
end
```

Training function

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

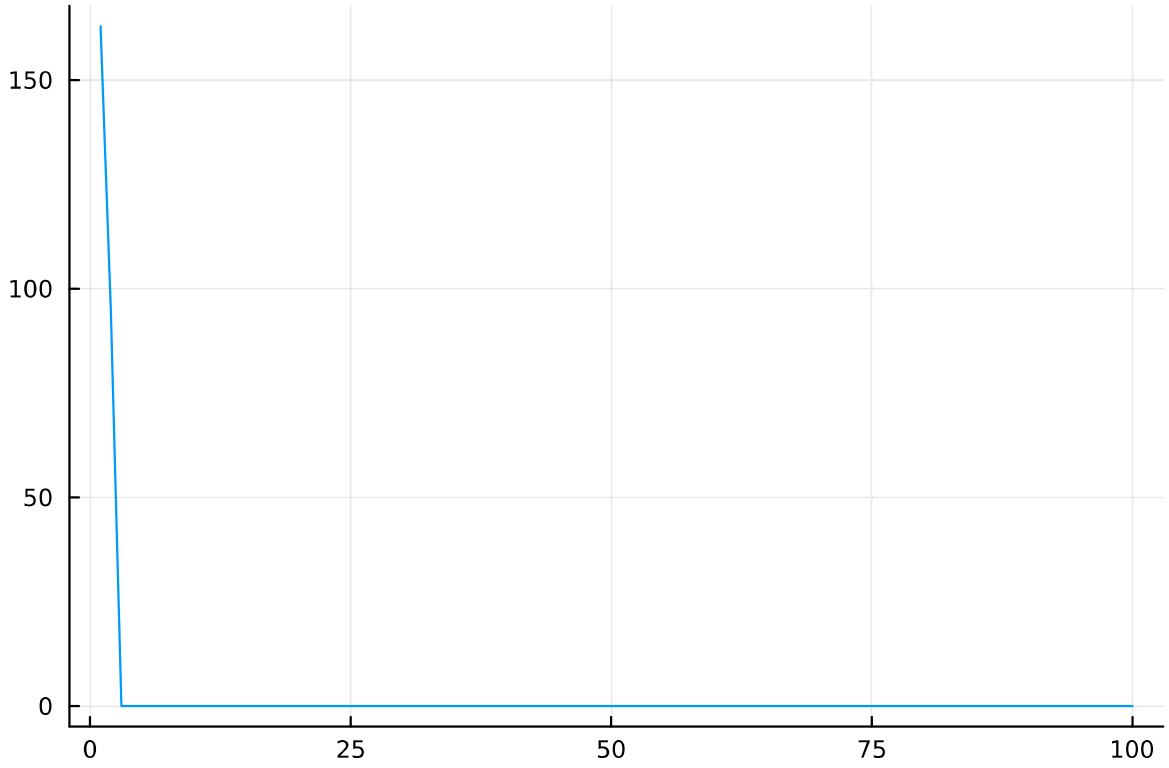
```
begin
    function train(W, trainX, trainY, learning_rate, num_epochs)
        losses = []
        for epoch in 1:num_epochs
            y_hat = sigmoid_activation(compute_h(W, trainX))
            # now that we have our predictions, we need to determine the
            # 'error', which is the difference between our predictions and
            # the true values
            error = y_hat - trainY
            append!(losses, 0.5 * sum(error .^ 2))
            grad = compute_gradient(error, y_hat, trainX)
            W -= learning_rate * grad

            if epoch == 1 || epoch % 5 == 0
                println("Epoch=$epoch; Loss=$(losses[end])")
            end
        end
        return W, losses
    end
end
```

Initialize our weight matrix and list of losses

```
0.1
```

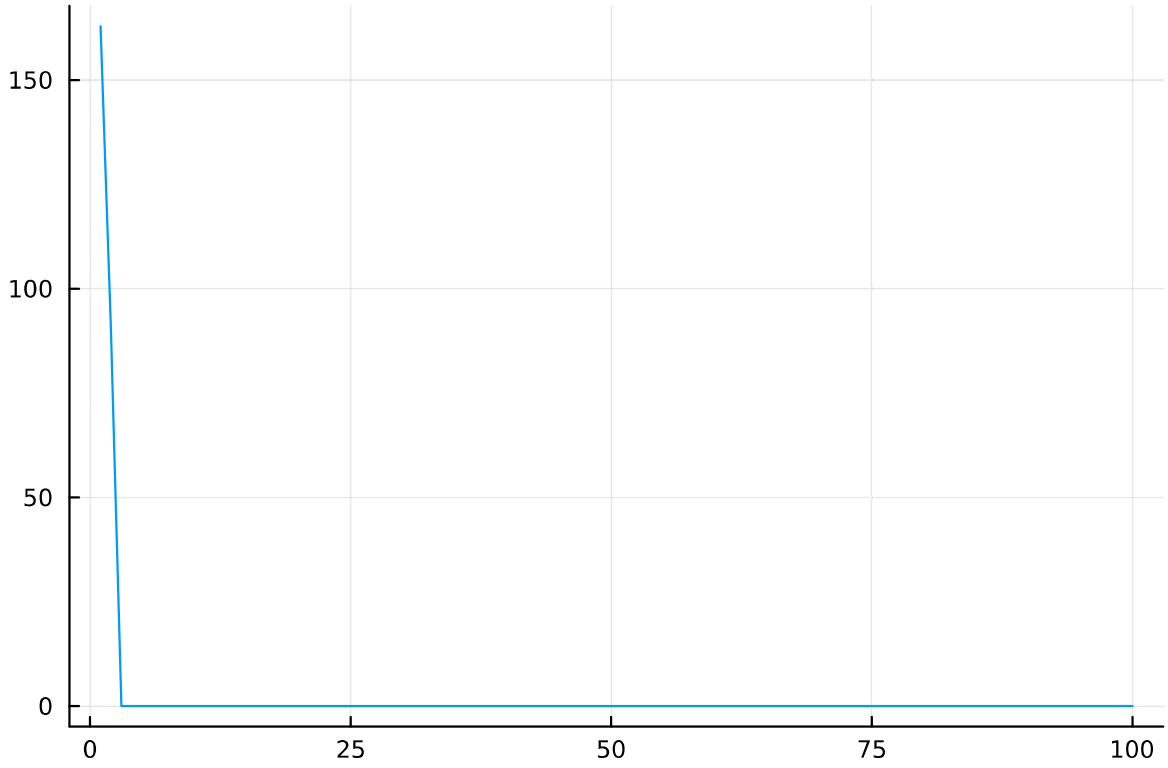
```
begin
    #initialize our weight matrix and necessary hyperparameters
    W = rand(Normal(), (size(X_train)[2], 1))
    num_epochs=100
    learning_rate=0.1
end
```

Train our model



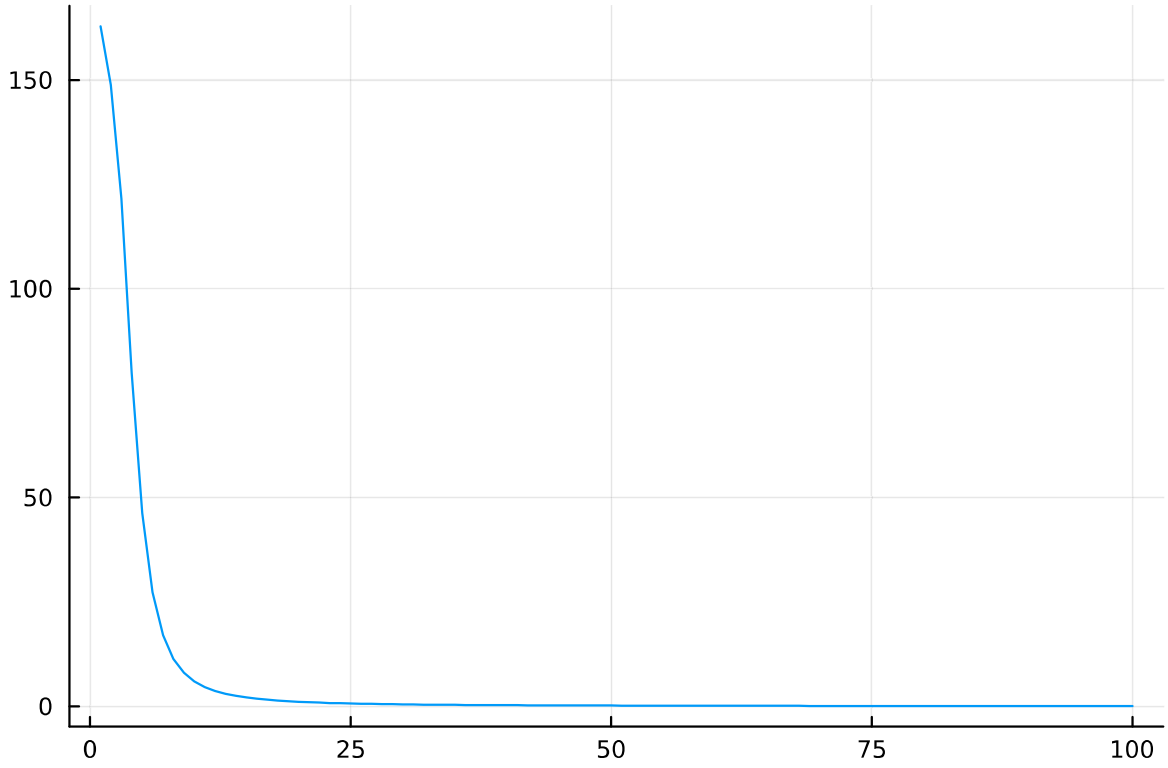

```
begin
    #training model
    θ, losses = train(W, X_train, y_train, learning_rate, num_epochs)
    #visualiza training process
    plot(1:num_epochs, losses, legend=false)
end
```

```
Epoch=1; Loss=162.89330008883746
Epoch=5; Loss=1.402995161955437e-297
Epoch=10; Loss=1.402995161955437e-297
Epoch=15; Loss=1.402995161955437e-297
Epoch=20; Loss=1.402995161955437e-297
Epoch=25; Loss=1.402995161955437e-297
Epoch=30; Loss=1.402995161955437e-297
Epoch=35; Loss=1.402995161955437e-297
Epoch=40; Loss=1.402995161955437e-297
Epoch=45; Loss=1.402995161955437e-297
Epoch=50; Loss=1.402995161955437e-297
Epoch=55; Loss=1.402995161955437e-297
Epoch=60; Loss=1.402995161955437e-297
Epoch=65; Loss=1.402995161955437e-297
Epoch=70; Loss=1.402995161955437e-297
Epoch=75; Loss=1.402995161955437e-297
Epoch=80; Loss=1.402995161955437e-297
Epoch=85; Loss=1.402995161955437e-297
Epoch=90; Loss=1.402995161955437e-297
Epoch=95; Loss=1.402995161955437e-297
Epoch=100; Loss=1.402995161955437e-297
```



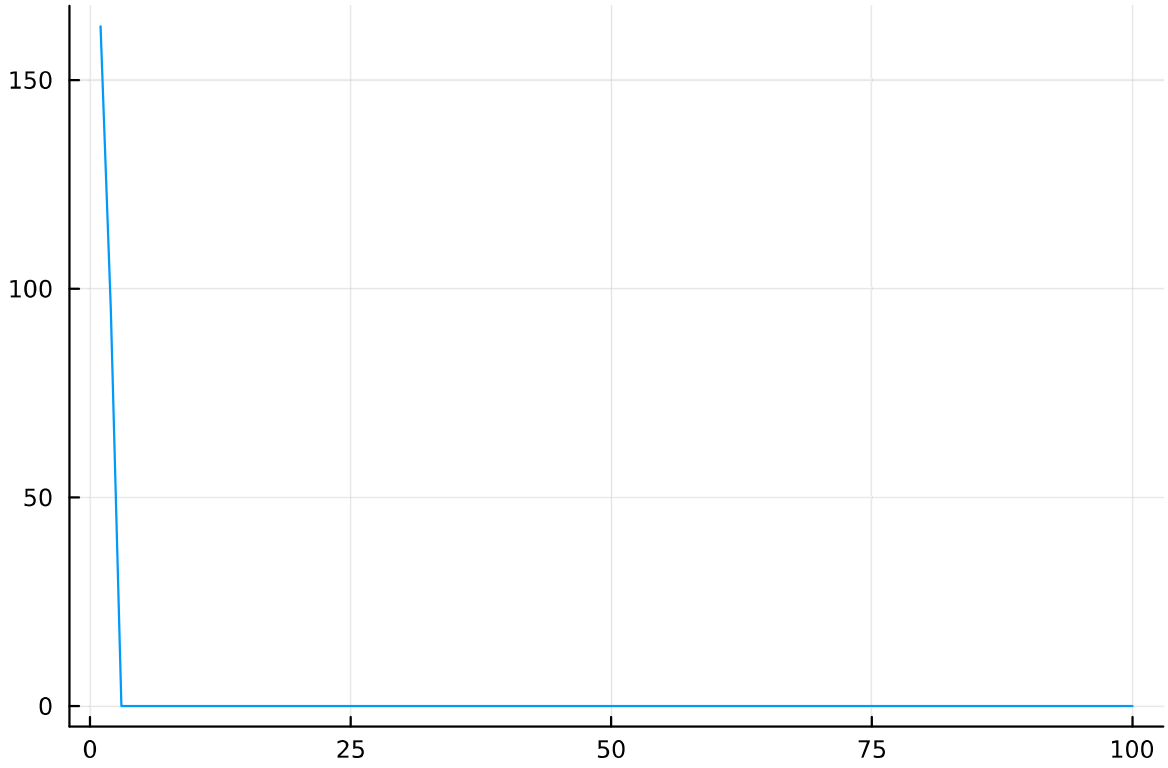

```
begin
    learning_rate_1=0.05
    θ_one, losses_one = train(W, X_train, y_train, learning_rate_1, num_epochs)
    #visualiza training process
    plot(1:num_epochs, losses_one, legend=false)
end
```

```
Epoch=1; Loss=162.89330008883746
Epoch=5; Loss=1.567340875966009e-136
Epoch=10; Loss=1.567340875966009e-136
Epoch=15; Loss=1.567340875966009e-136
Epoch=20; Loss=1.567340875966009e-136
Epoch=25; Loss=1.567340875966009e-136
Epoch=30; Loss=1.567340875966009e-136
Epoch=35; Loss=1.567340875966009e-136
Epoch=40; Loss=1.567340875966009e-136
Epoch=45; Loss=1.567340875966009e-136
Epoch=50; Loss=1.567340875966009e-136
Epoch=55; Loss=1.567340875966009e-136
Epoch=60; Loss=1.567340875966009e-136
Epoch=65; Loss=1.567340875966009e-136
Epoch=70; Loss=1.567340875966009e-136
Epoch=75; Loss=1.567340875966009e-136
Epoch=80; Loss=1.567340875966009e-136
Epoch=85; Loss=1.567340875966009e-136
Epoch=90; Loss=1.567340875966009e-136
Epoch=95; Loss=1.567340875966009e-136
Epoch=100; Loss=1.567340875966009e-136
```



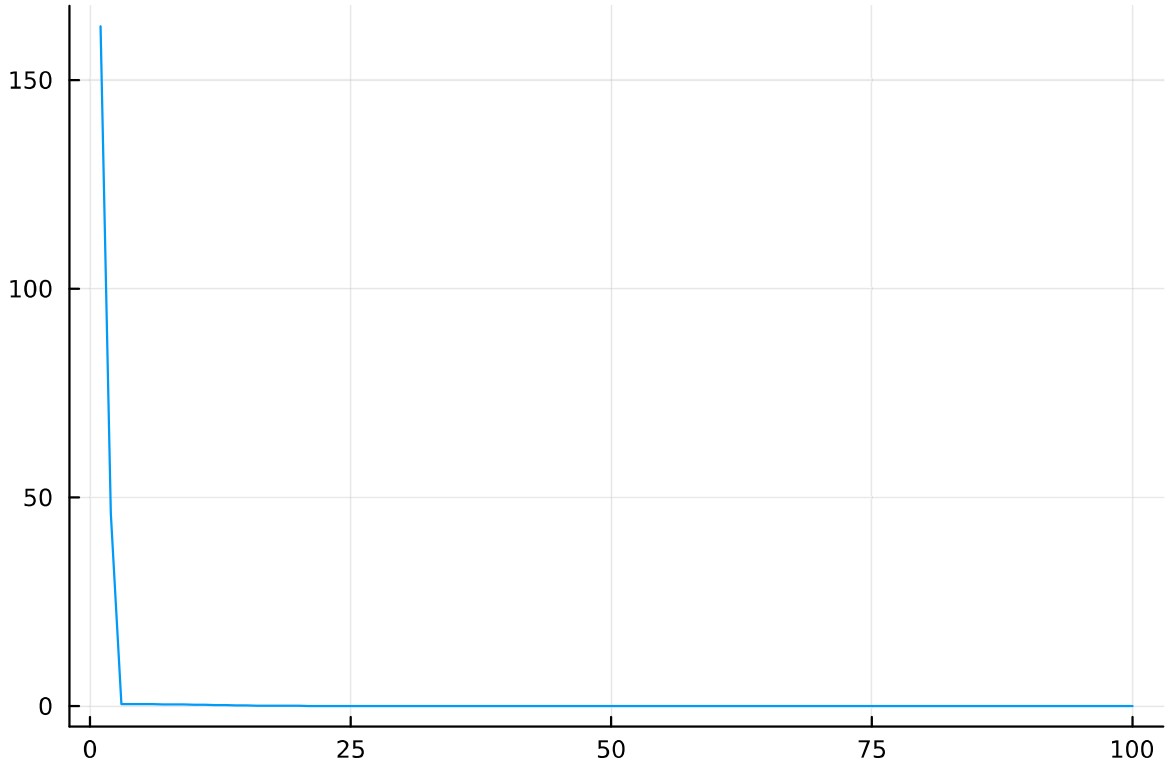

```
begin
    learning_rate_2 = 0.000125
    θ_two, losses_two = train(W, X_train, y_train, learning_rate_2, num_epochs)
    #visualiza training process
    plot(1:num_epochs, losses_two, legend=false)
end
```

```
Epoch=1; Loss=162.89330008883746
Epoch=5; Loss=46.28365337926513
Epoch=10; Loss=5.968990637590512
Epoch=15; Loss=2.104536260106337
Epoch=20; Loss=1.0719137421165839
Epoch=25; Loss=0.6582333417615046
Epoch=30; Loss=0.4549024116783561
Epoch=35; Loss=0.33874612769832246
Epoch=40; Loss=0.2645703136572967
Epoch=45; Loss=0.2133490068358574
Epoch=50; Loss=0.17601250805823204
Epoch=55; Loss=0.14773861603927493
Epoch=60; Loss=0.1257211860923215
Epoch=65; Loss=0.10820611830685481
Epoch=70; Loss=0.09403437119192487
Epoch=75; Loss=0.08240655542138091
Epoch=80; Loss=0.07275259886126535
Epoch=85; Loss=0.06465504030950722
Epoch=90; Loss=0.0578015315637662
Epoch=95; Loss=0.0519541671237952
Epoch=100; Loss=0.04692898469114107
```




```
begin
    learning_rate_3=0.4
    θ_three, losses_three = train(W, X_train, y_train, learning_rate_3, num_epochs)
    #visualiza training process
    plot(1:num_epochs, losses_three, legend=false)
end
```

```
Epoch=1; Loss=162.89330008883746
Epoch=5; Loss=0.0
Epoch=10; Loss=0.0
Epoch=15; Loss=0.0
Epoch=20; Loss=0.0
Epoch=25; Loss=0.0
Epoch=30; Loss=0.0
Epoch=35; Loss=0.0
Epoch=40; Loss=0.0
Epoch=45; Loss=0.0
Epoch=50; Loss=0.0
Epoch=55; Loss=0.0
Epoch=60; Loss=0.0
Epoch=65; Loss=0.0
Epoch=70; Loss=0.0
Epoch=75; Loss=0.0
Epoch=80; Loss=0.0
Epoch=85; Loss=0.0
Epoch=90; Loss=0.0
Epoch=95; Loss=0.0
Epoch=100; Loss=0.0
```



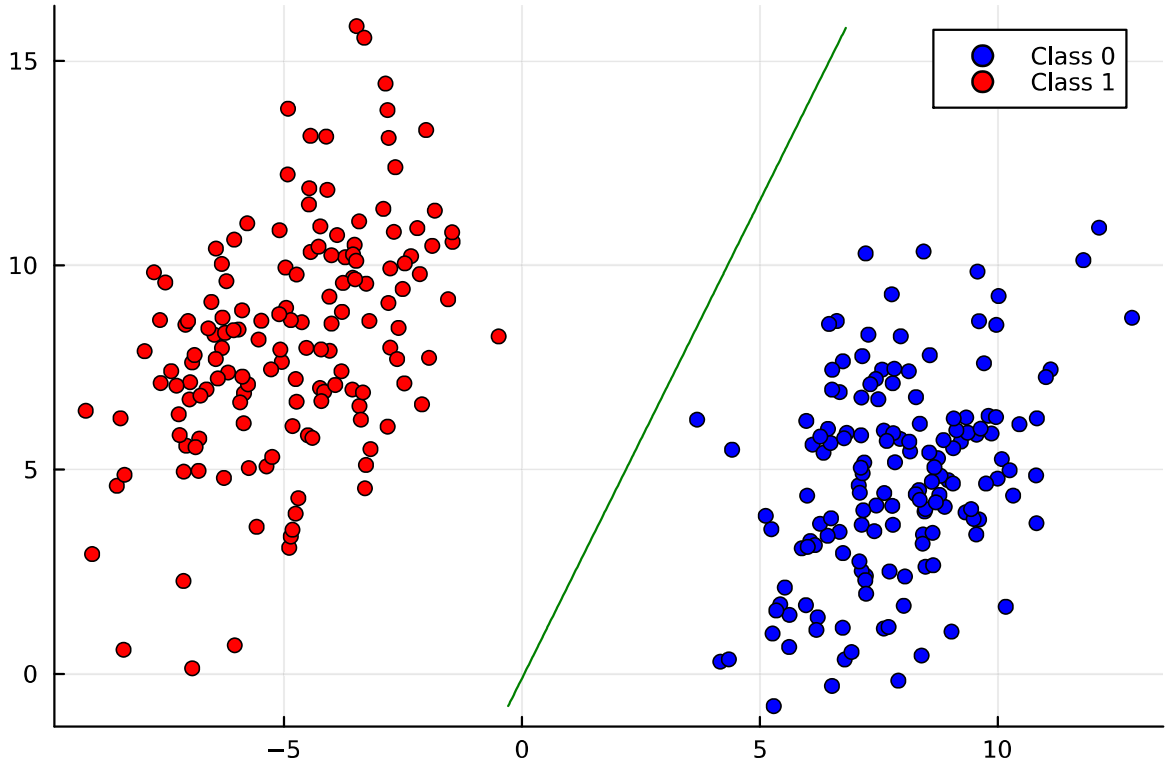

```
begin
    learning_rate_4 = 0.005
    θ_four, losses_four = train(W, X_train, y_train, learning_rate_4, num_epochs)
    #visualiza training process
    plot(1:num_epochs, losses_four, legend=false)
end
```

```
Epoch=1; Loss=162.89330008883746
Epoch=5; Loss=0.4674791065419807
Epoch=10; Loss=0.34713373454415825
Epoch=15; Loss=0.15454571556704924
Epoch=20; Loss=0.05860360503402221
Epoch=25; Loss=0.02644292916226384
Epoch=30; Loss=0.014229279862187614
Epoch=35; Loss=0.008678058176301693
Epoch=40; Loss=0.005775581648998413
Epoch=45; Loss=0.004093928522976015
Epoch=50; Loss=0.003040288665854061
Epoch=55; Loss=0.002341182378235579
Epoch=60; Loss=0.0018550419433261288
Epoch=65; Loss=0.0015041760070381687
Epoch=70; Loss=0.0012431010061284416
Epoch=75; Loss=0.0010438385379911658
Epoch=80; Loss=0.0008884519430210901
Epoch=85; Loss=0.000765032613540704
Epoch=90; Loss=0.00066543337610538
Epoch=95; Loss=0.0005839337375775883
Epoch=100; Loss=0.0005164248615807377
```

Evaluate result

```
begin
    y_pred = predict(θ, X_test)
    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 == 1 && predicted_label == 1
            true_positives += 1
        elseif true_label == 0 && predicted_label == 1
            false_positives += 1
        elseif true_label == 1 && predicted_label == 0
            false_negatives += 1
        elseif true_label == 0 && predicted_label == 0
            true_negatives += 1
        end
    end

    # Calculate precision, recall, and F1-score
    accuracy = (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)
    f1_score = 2 * precision * recall / (precision + recall)

    # Display
    print("acc: $accuracy, precision: $precision, recall: $recall, f1_score:
    $f1_score\n")
end
```

```
acc: 1.0, precision: 1.0, recall: 1.0, f1_score: 1.0
```

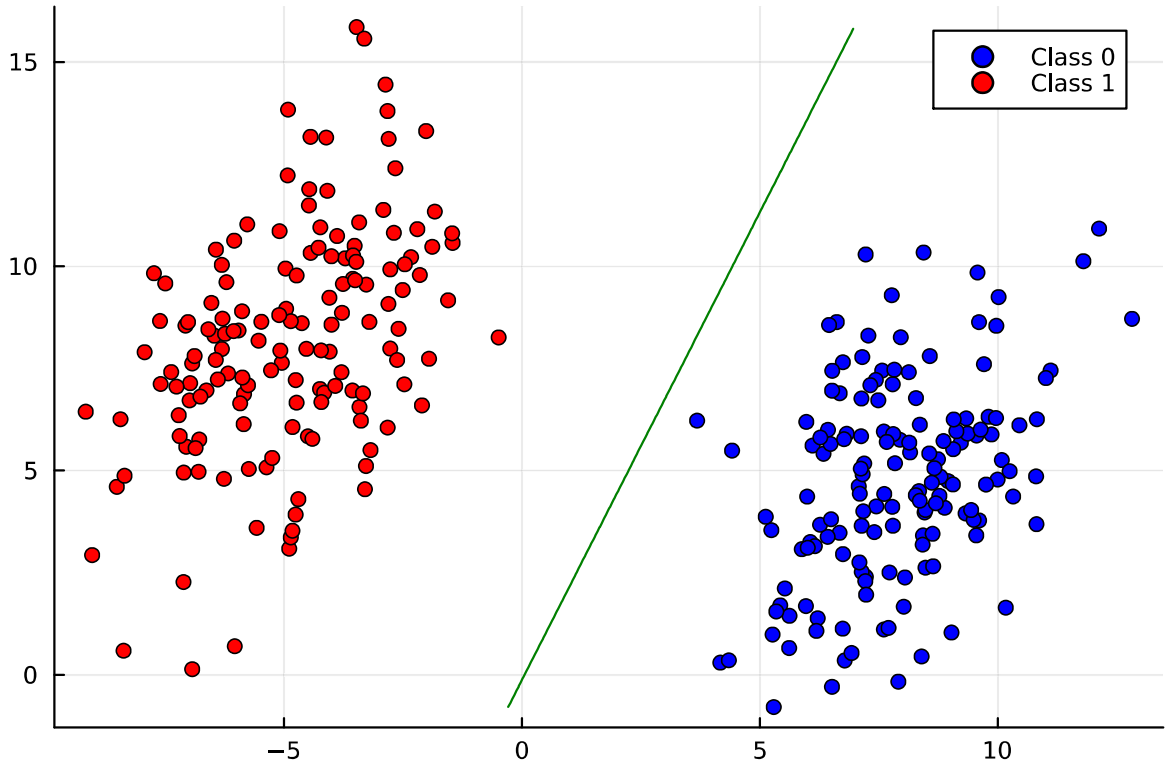



```
begin
    function visualization(X_test, y_test, θ)
    # Create a scatter plot
    plt = scatter(X_test'[1,:][y_test .== 0], X_test'[2,:][y_test .== 0],
    label="Class 0", color=:blue, legend=:topright, markersize=4) # assign to plt
    var
    scatter!(X_test'[1,:][y_test .== 1], X_test'[2,:][y_test .== 1], label="Class
    1", color=:red, markersize=4)

    # Getting decision boundary configuration
    b = θ[3]
    θ_ml = θ[1:2]

    decision(x) = θ_ml' * x + b

    D_test = ([
      tuple.(eachcol(hcat(X_test'[1,:][y_test .== 0], X_test'[2,:][y_test .==
    0])'), 1)
      tuple.(eachcol(hcat(X_test'[1,:][y_test .== 1], X_test'[2,:][y_test .==
    1])'), -1)
    ])

    # Max, mix for visualization decision boundary
    x_min = minimum(map((p) -> p[1][1], D_test))
    y_min = minimum(map((p) -> p[1][2], D_test))
    x_max = maximum(map((p) -> p[1][1], D_test))
    y_max = maximum(map((p) -> p[1][2], D_test))

    # Display decision boundary
    contour!(plt, x_min:0.1:x_max, y_min:0.1:y_max,
            (x, y) -> decision([x, y]),
            levels=[0], linestyles=:solid, label="Decision boundary",
            colorbar_entry=false, color=:green)
    return plt
end
```

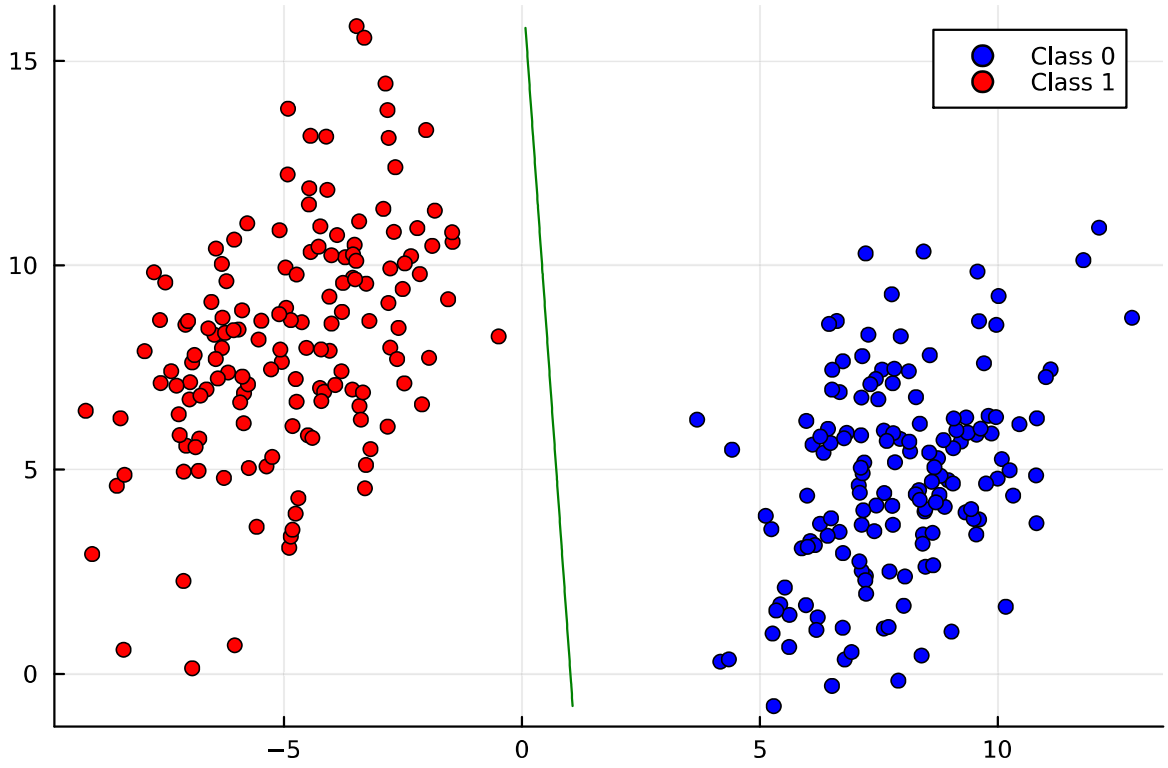
```
visualization(X_test, y_test, θ)
```

TODO: Study about accuracy, recall, precision, f1-score.

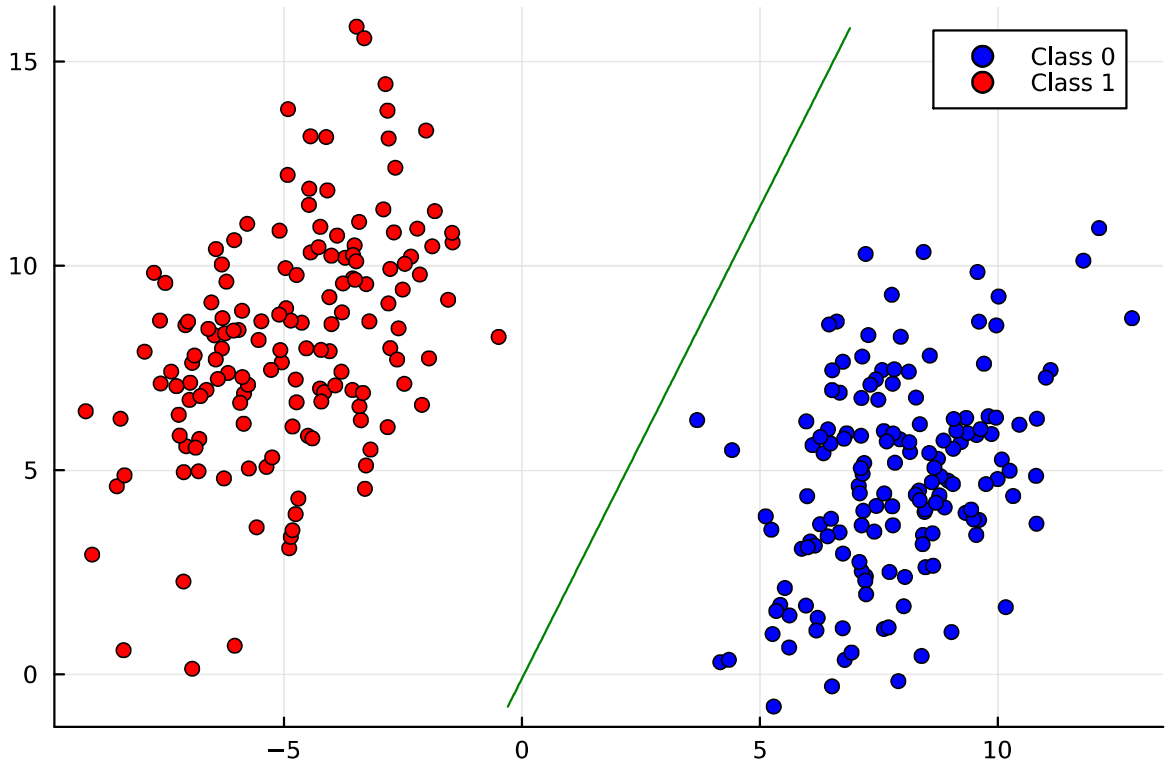
- Accuracy: The accuracy of this model is nearly 100%, meaning the predicted values in this model are almost correct
- Recall: The recall of this model is nearly 100%, indicating almost all true predictions are positive
- Precision: The precision of this model is nearly 100%, meaning the number of correct positive takes nearly almost all of the total positive observations
- F1: The F1-score of this model is nearly 100%, meaning the relationship between recall and precision is almost harmonic



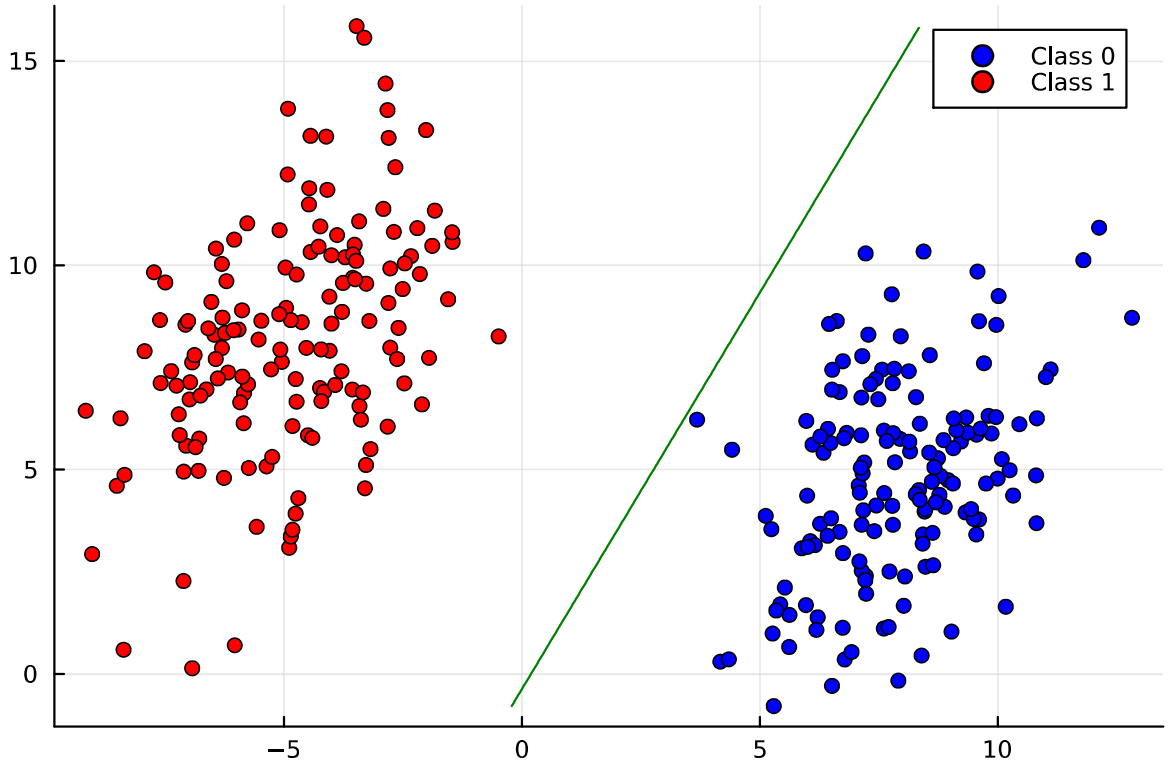

```
begin
    visualization(X_test, y_test, θ_one)
end
```




```
begin
    visualization(X_test, y_test, θ_two)
end
```




```
begin
    visualization(X_test, y_test, θ_three)
end
```




```
begin
    visualization(X_test, y_test, θ_four)
end
```

TODO: Try out different learning rates. Give me your observations

- Learning rate 0.05: All the statistics are nearly the same as learning rate 0.1
- Learning rate 0.000125: The learning curve is clearer, showing the models is working
- Learning rate 0.4: The results dropped down to 0, indicating the rate of change of the loss function with respect to the parameters of the model is zero
- Learning rate 0.005: The loss function gradually reduces for every epochs, similar to the abov