📚 Table of Contents

Lab 02: Gradient Descent

- Student name: Âu Dương Khang
- ID: 21127621

Cell deleted (UNDO)

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

Cell deleted (UNDO)

1. Loss landscape

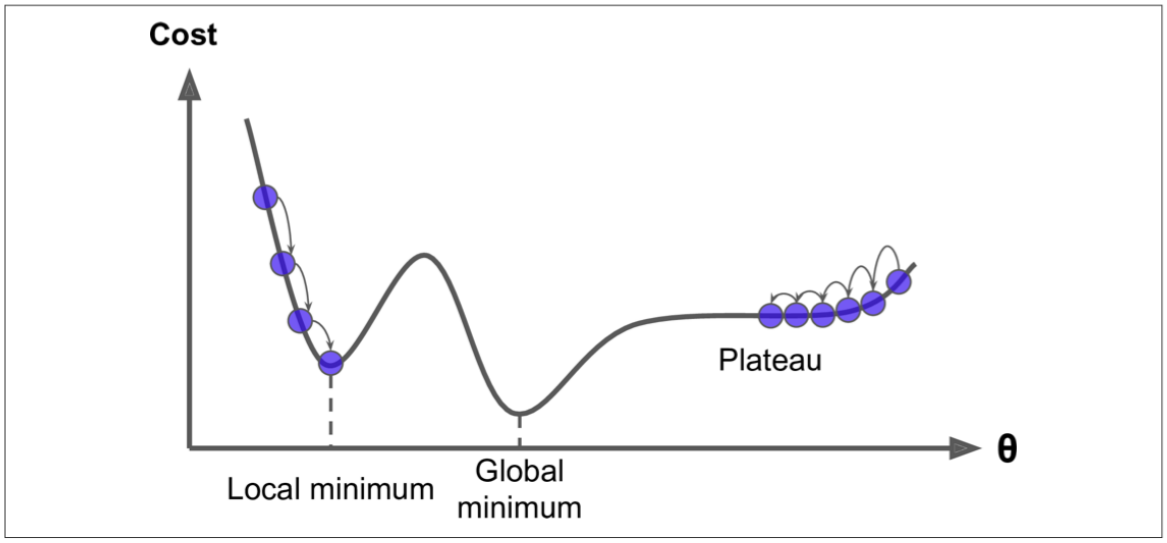


Figure 1. Loss landscape visualized as a 2D plot. Source: codecamp.vn

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.

Cell deleted (UNDO)

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

Cell deleted (UNDO)

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

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

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

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

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

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

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

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

"""
```

3.2. Backward

Cell deleted (UNDO)

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

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

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

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

```
md"""
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

$$\nabla Loss = \frac{\partial Loss(W)}{\partial W} $$
$$= \frac{1}{n} \sum_{i=1}^n\frac{\partial Loss_{i}}{\partial \hat{y}_i} \times
\frac{\partial \hat{y}_i}{\partial W}$$

$$= \frac{1}{n} \sum_{i=1}^n 2(\hat{y}_i - y_i) \times x_i$$
"""
```

4. Implementation

4.1. Import library

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

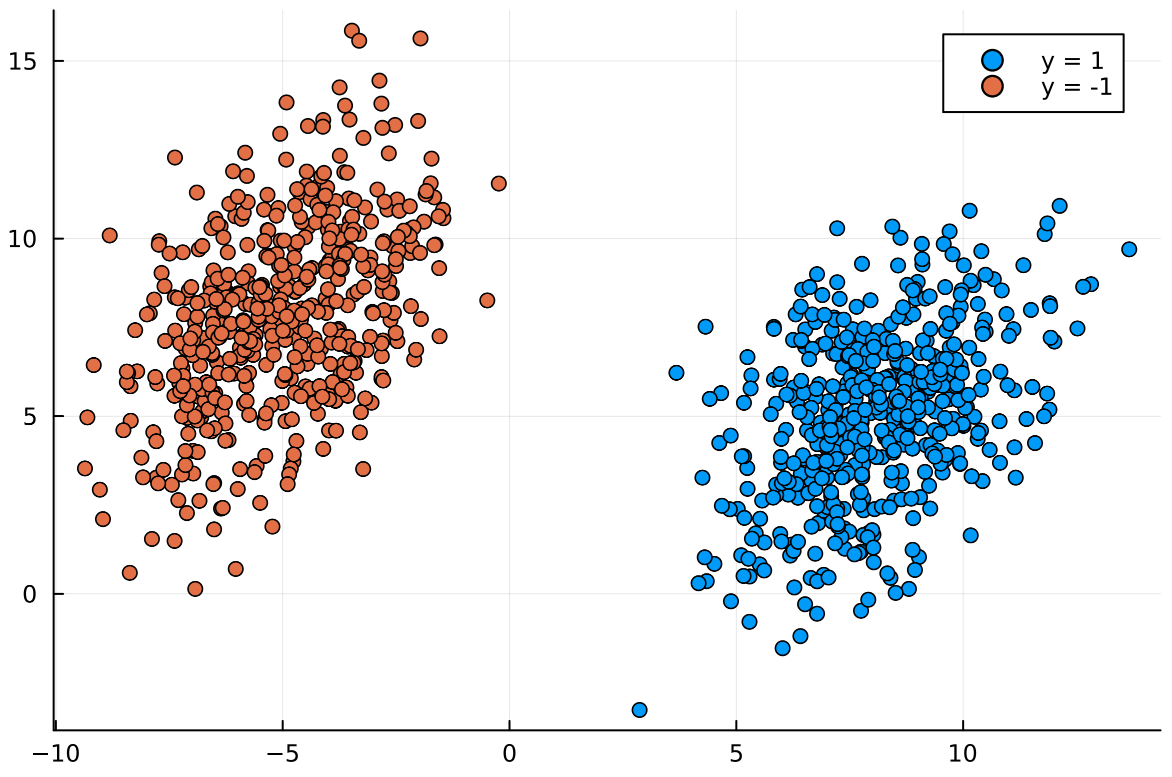
Cell deleted (UNDO)

```
MersenneTwister(2024)
```

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

4.2. Create data

Cell deleted (UNDO)



```
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
```

Cell deleted (UNDO)

```
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
```

```julia
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], 300
   7.05549   4.70516  1.0                                                            9
  -6.87358   3.98823  1.0                                                           -6
  -2.05821   6.87055  1.0                                                            8
  -5.71526   7.10931  1.0                                                           -4
  -7.14349   3.62296  1.0                                                           -1
   8.85531   6.33639  1.0                                                            5
  10.045     5.23048  1.0                                                           -4
   ⋮                                                                                 ⋮
  -5.93771   6.71143  1.0                                                            6
  -5.38006   8.4446   1.0                                                            7
  -4.582    10.4695   1.0                                                           -3
  -4.57232   7.8681   1.0                                                            8
   9.01379   5.11398  1.0                                                           -7
   7.12285   3.71975  1.0                                                           -6
```

```julia
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)
```

Cell deleted (UNDO)

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 sigmoid_activation(x) .*(1.0 .- sigmoid_activation(x))

    end
end
```

Compute output

Cell deleted (UNDO)

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

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

    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 = ...
        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
end
```

Compute gradient

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

```julia
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
        """
        dactivation = sigmoid_deriv(y_hat)
        gradient = trainX' * (error .* dactivation)
        # return?
        return gradient

    end
end
```

Cell deleted (UNDO)

ction

```
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

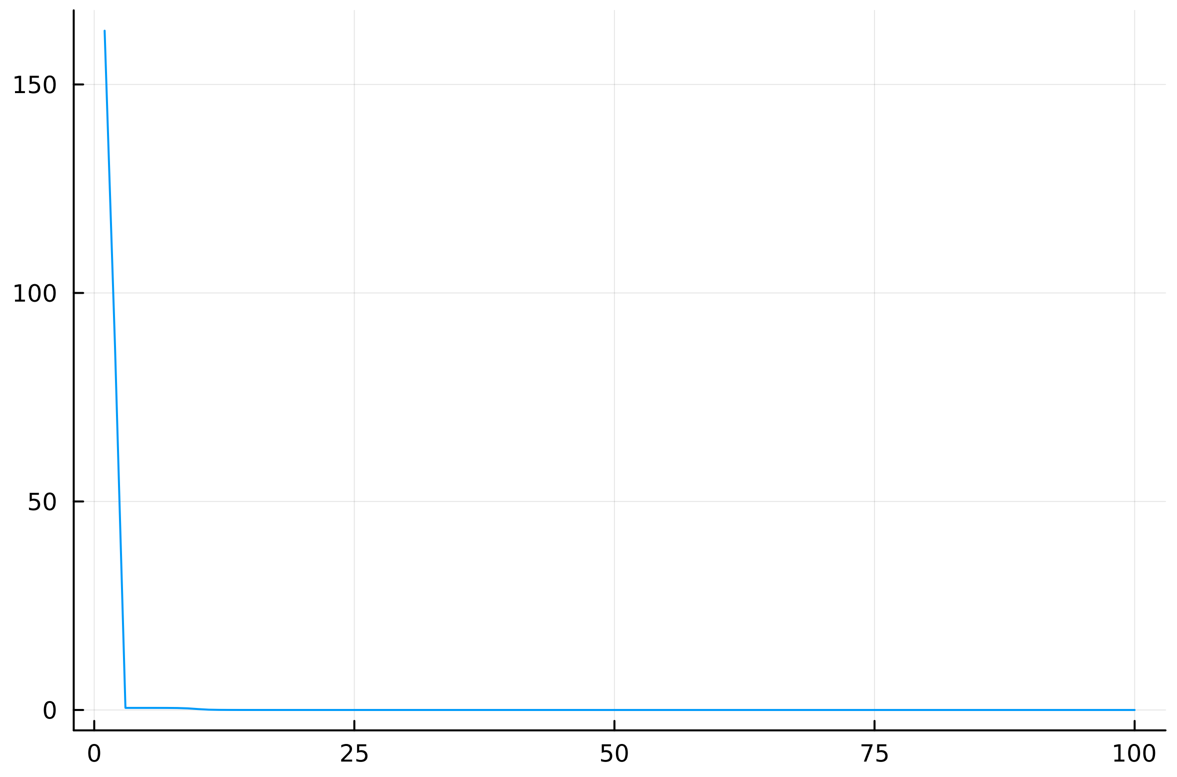
```
md"""
#### Initialize our weight matrix and list of losses
"""
```

```
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

Cell deleted (UNDO)



```
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=0.4996629372262392
Epoch=10; Loss=0.213901725493055
Epoch=15; Loss=0.00519495524271303
Epoch=20; Loss=0.0013469312005308456
Epoch=25; Loss=0.000604331216694869
Epoch=30; Loss=0.0003413439086842821
Epoch=35; Loss=0.00021895307403778155
Epoch=40; Loss=0.00015226302432830107
Epoch=45; Loss=0.00011197159288541358
Epoch=50; Loss=8.578542214448916e-5
Epoch=55; Loss=6.78146276933873e-5
Epoch=60; Loss=5.495070756475468e-5
Epoch=65; Loss=4.5427387866039997e-5
Epoch=70; Loss=3.818080640171032e-5
Epoch=75; Loss=3.253916550707599e-5
Epoch=80; Loss=2.8061322021449194e-5
Epoch=85; Loss=2.4447897715880722e-5
Epoch=90; Loss=2.1489930265636757e-5
Epoch=95; Loss=1.9037981612168296e-5
Epoch=100; Loss=1.6982893206780018e-5
```

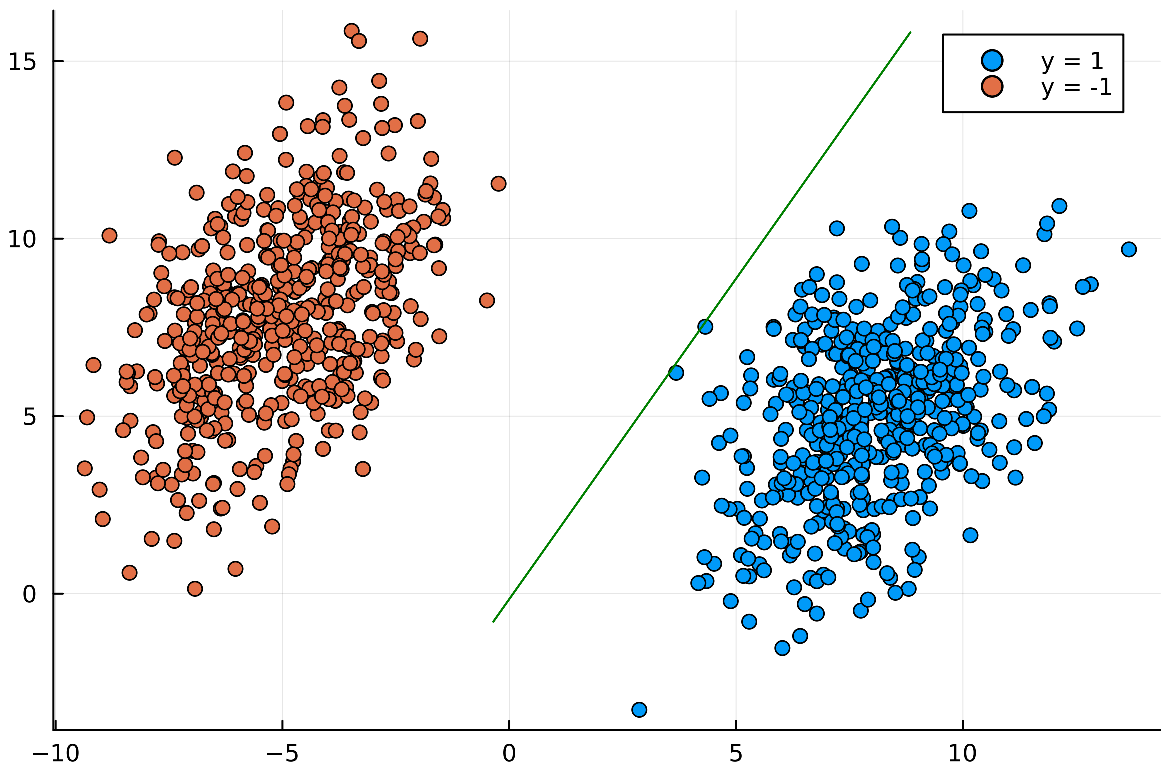
Evaluate result

Cell deleted (UNDO)

```julia
begin
    y_pred = predict(W, 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: 0.52, precision: 1.0, recall: 0.03355704697986577, f1_score: 0.0649
3506493506493
```

Cell deleted (UNDO)



```julia
begin
    # Create a scatter plot
    scatter(X_test'[1,:][y_test .== 0], X_test'[2,:][y_test .== 0], label="Class 0",
    color=:blue, legend=:topright, markersize=4)
    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]
    θₘₗ = θ[1:2]

    decision(x) = θₘₗ' * 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
    xₘᵢₙ = minimum(map((p) -> p[1][1], D_test))
    yₘᵢₙ = minimum(map((p) -> p[1][2], D_test))
    xₘₐₓ = maximum(map((p) -> p[1][1], D_test))
    yₘₐₓ = maximum(map((p) -> p[1][2], D_test))

    # Display decision boundary
    contour!(plt, xₘᵢₙ:0.1:xₘₐₓ, yₘᵢₙ:0.1:yₘₐₓ,
             (x, y) -> decision([x, y]),
             levels=[0], linestyles=:solid, label="Decision boundary",
    Cell deleted (UNDO) lorbar_entry=false, color=:green)
end
```

Cell deleted (UNDO)

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

- Accuracy: This is the ratio of true positives and true negatives to all cases. It's a measure of how good our model is at predicting the correct category (classes or labels).

$$Accuracy = \frac{TP + TN}{TP + TN + NP + NN}$$

- Recall: This is the ratio of true positives to all actual positives. Recall is important when the cost of missing a positive instance is high.

$$Recall = \frac{TP}{TP + FN}$$

- Precision: This is the ratio of true positives to all predicted positives. We focus on precision when we need our predictions to be correct, i.e., ideally, we want to make sure our model is right when it predicts a label.

$$Precision = \frac{TP}{TP + FP}$$

- F1: This is the weighted average of precision and recall and is usually more useful than accuracy, especially if the class distribution is uneven or the cost of false positives and false negatives are different. F1 score is the harmonic mean of precision and recall and becomes 1 only when both are 1. F1 score can be calculated by F1 = 2 * (precision * recall)/ (precision + recall)

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

Looking at the results, it seems like the model has a high precision (1.0), meaning that when it predicts a positive result, it's usually correct. However, its recall is quite low (0.033576497989677), indicating that it's missing a lot of actual positive instances. The F1-score (0.06493506493506493) reflects this imbalance between precision and recall. The accuracy (0.52) isn't very high either, suggesting that the model isn't performing very well overall.

```
md"""
**TODO: Study about accuracy, recall, precision, f1-score.**
- Accuracy: This is the ratio of true positives and true negatives to all cases. It'
a measure of how good our model is at predicting the correct category (classes or
labels).
$$Accuracy = \frac{TP + TN}{TP + TN + NP + NN}$$
- Recall: This is the ratio of true positives to all actual positives. Recall is
important when the cost of missing a positive instance is high.
$$Recall = \frac{TP}{TP + FN}$$
- Precision: This is the ratio of true positives to all predicted positives. We focu
on precision when we need our predictions to be correct, i.e., ideally, we want to
make sure our model is right when it predicts a label.
Cell deleted (UNDO) \frac{TP}{TP + FP}$$
```

```
- F1: This is the weighted average of precision and recall and is usually more usefu
than accuracy, especially if the class distribution is uneven or the cost of false
positives and false negatives are different. F1 score is the harmonic mean of
precision and recall and becomes 1 only when both are 1. F1 score can be calculated
by F1 = 2 * (precision * recall)/ (precision + recall)
$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

Looking at the results, it seems like the model has a high precision (1.0), meaning
that when it predicts a positive result, it's usually correct. However, its recall i
quite low (0.033576497989677), indicating that it's missing a lot of actual positive
instances. The F1-score (0.06493506493506493) reflects this imbalance between
precision and recall. The accuracy (0.52) isn't very high either, suggesting that th
model isn't performing very well overall.
```

Cell deleted (UNDO)

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

- Learning rate is used to scale the magnitude of parameter updates during gradient descent. The choice of the value for learning rate can impact two things: 1) how fast the algorithm learns and 2) whether the cost function is minimized or not.
- High learning rate: If the learning rate is too high, the algorithm might overshoot the minimum and diverge, resulting in large oscillations. The cost function may end up bouncing back and forth and take a longer time to converge, or in some cases, it might not converge at all. The value of high learning rate is about higher 0.1.
- Optimal learning rate: An optimal learning rate would lead to a steady and quick convergence to the global minimum. The cost function decreases and eventually stabilizes, leading to the lowest possible loss. The value of optimal learning rate is about 0.01 < lr < 0.1.
- Low learning rate: If the learning rate is too low, the algorithm will eventually reach the minimum but the convergence will be very slow. This means the algorithm will take a very long time to run and will be computationally expensive. The value of optimal learning rate is about lower 0.01.

```
md"""
**TODO: Try out different learning rates. Give me your observations**
- Learning rate is used to scale the magnitude of parameter updates during gradient descent. The choice of the value for learning rate can impact two things: 1) how fast the algorithm learns and 2) whether the cost function is minimized or not.
- High learning rate: If the learning rate is too high, the algorithm might overshoot the minimum and diverge, resulting in large oscillations. The cost function may end up bouncing back and forth and take a longer time to converge, or in some cases, it might not converge at all. The value of high learning rate is about higher 0.1.
- Optimal learning rate: An optimal learning rate would lead to a steady and quick convergence to the global minimum. The cost function decreases and eventually stabilizes, leading to the lowest possible loss. The value of optimal learning rate is about 0.01 < lr < 0.1.
- Low learning rate: If the learning rate is too low, the algorithm will eventually reach the minimum but the convergence will be very slow. This means the algorithm will take a very long time to run and will be computationally expensive. The value of optimal learning rate is about lower 0.01.
"""
```

Cell deleted (UNDO)

Cell deleted (UNDO)