📚 Table of Contents

Lab 02: Gradient Descent

Selection deleted

- Student name: Phan Huy Đức Tài
- ID: 21127687

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

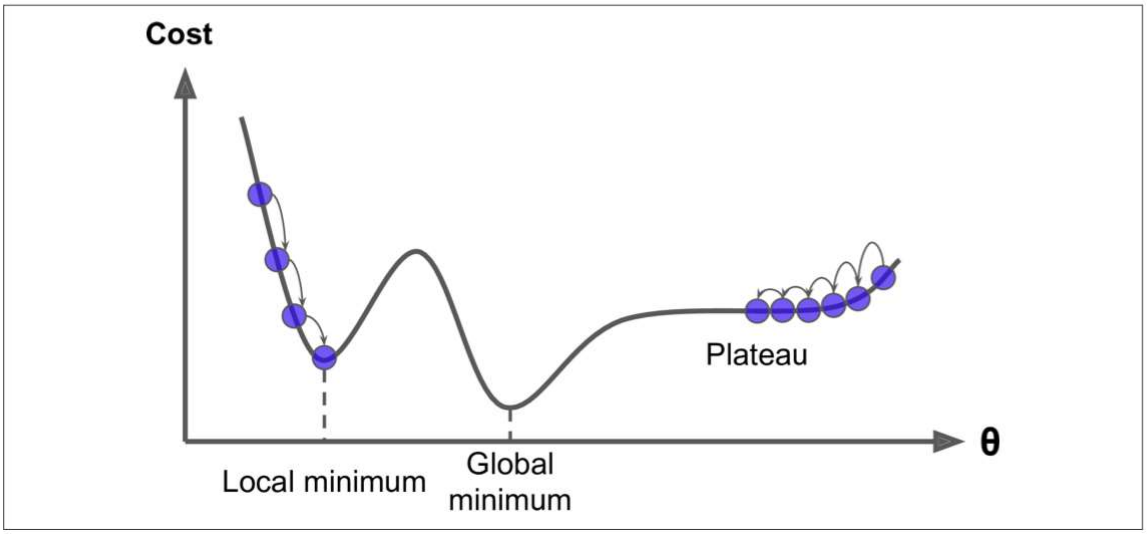


Figure 1. Loss landscape visualized as a 2D plot. Source: codecamp.vn

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.

Selection deleted

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

Selection deleted

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

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

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

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

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

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

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

$$= \frac{2}{n}\sum_{i=1}^{n}\hat{y}_i^2 x_i^T - \hat{y}_i^3 x_i^T - \hat{y}_i y_i x_i^T + \hat{y}_i^2 y_i x_i^T$$

4. Implementation

Section 1 Heading

4.1. Import library

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

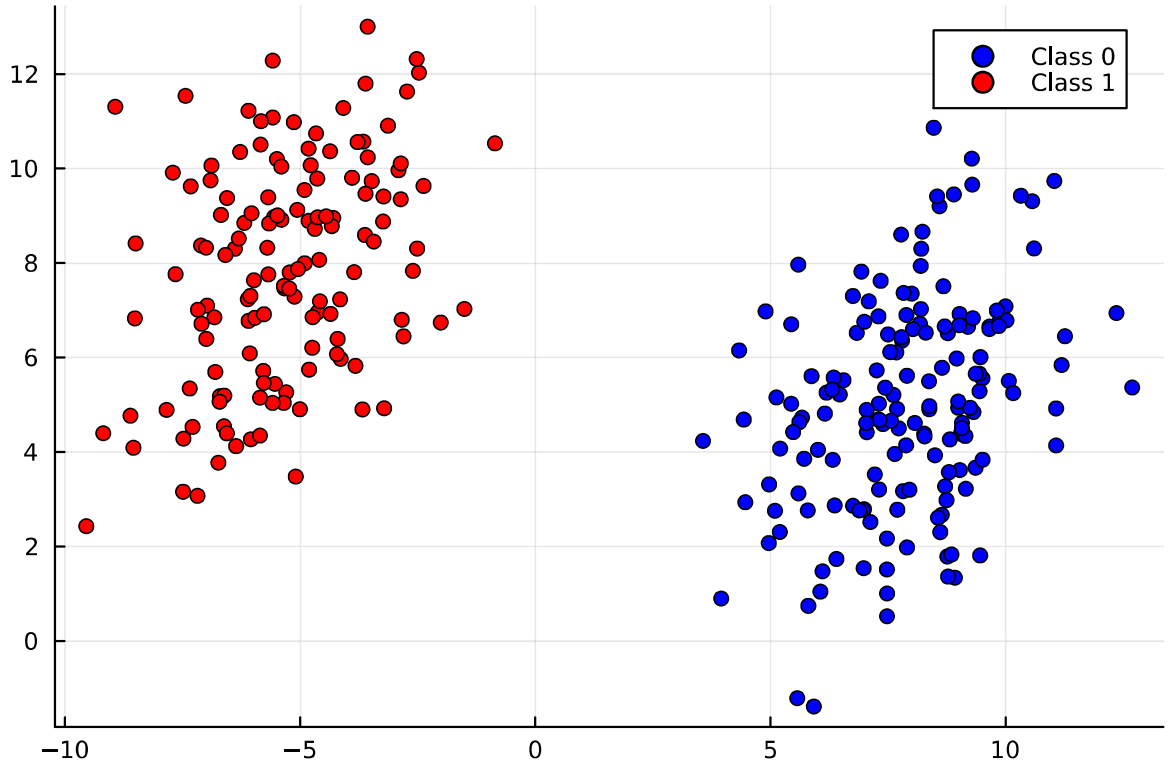
MersenneTwister(2024)

```
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))'
    print(size(X))
    print(size(y))

    # 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
```

```
(2, 1000)(1, 1000)
```




```
begin
    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)
end
```

```
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)
    print(size(X_aug))
end
```

```
(3, 1000)
```

```
(700×3 Matrix{Float64}:   , [1.0, 1.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0,    more ,0.0], 3
 -6.19381    6.61483   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
        #reshape(x, (size(x), 1),))
        """compute the sigmoid activation value for a given input"""
        return 1. ./ (1. .+ 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
        """
        temp = x' * (1. .- x)
        #print("X dimension: ", string(size(x)))
        #print("Dimension: ", string(size(temp)))
        return temp
    end

    #print(sigmoid_activation([3, 4, 5]))
    #print(sigmoid_deriv(sigmoid_activation([3, 4, 5])))
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 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 = 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)
```

```
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
        """

        deriv_error = sigmoid_deriv(y_hat)
        # features: trainX (n x d)

        return trainX' * (error * deriv_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
```

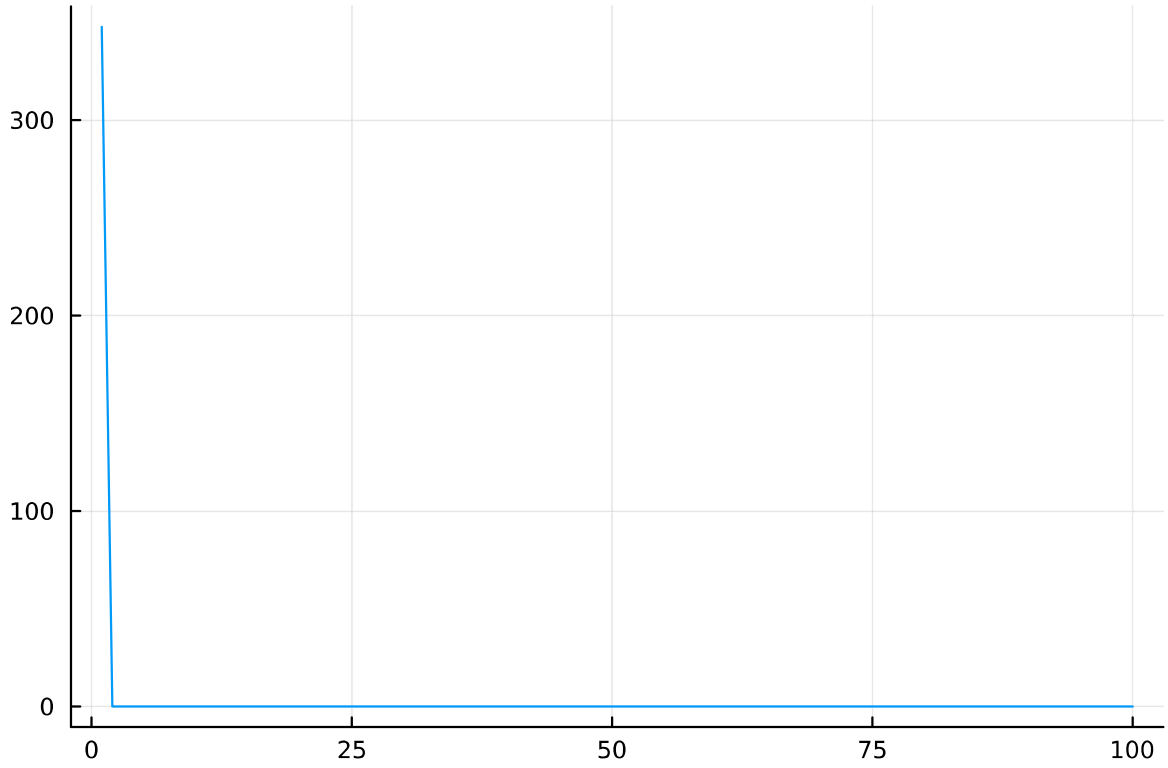
Initialize our weight matrix and list of losses

```
0.01
```

```
begin
    #initialize our weight matrix and necessary hyperparameters
    W = rand(Normal(), (size(X_train)[2], 1))
    print(size(W))
    #print(size(compute_h(W, X_train)),size(X_train), size(W))
    num_epochs=100
    #learning_rate=0.08
    #learning_rate = 0.1
    learning_rate = 0.01
end
```

```
(3, 1)
```

Train our model



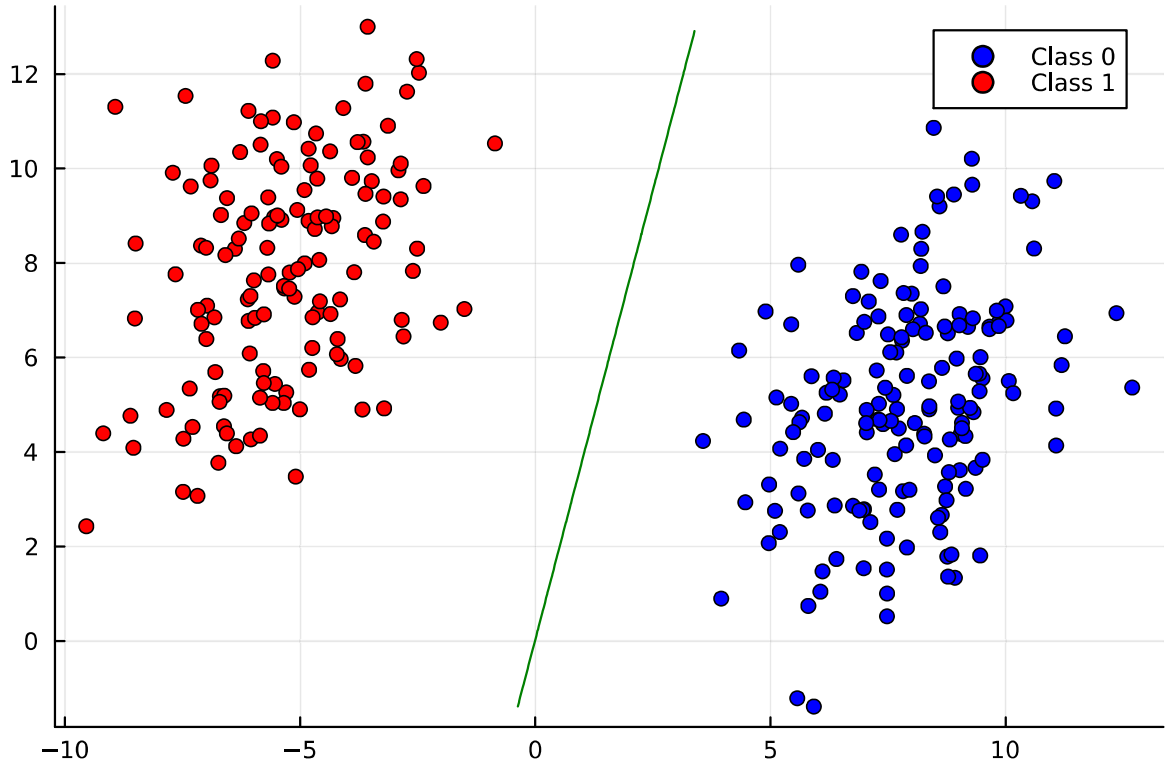

```
begin
    #training model
    #print(sigmoid_activation(compute_h(W, X_train)))
    #print(sigmoid_activation(compute_h(W, X_train)) - y_train)
    θ, losses = train(W, X_train, y_train, learning_rate, num_epochs)
    #visualiza training process
    plot(1:num_epochs, losses, legend=false)
end
```

```
Epoch=1; Loss=347.7942690062178
Epoch=5; Loss=5.1017344600080736e-194
Epoch=10; Loss=5.1017344600080736e-194
Epoch=15; Loss=5.1017344600080736e-194
Epoch=20; Loss=5.1017344600080736e-194
Epoch=25; Loss=5.1017344600080736e-194
Epoch=30; Loss=5.1017344600080736e-194
Epoch=35; Loss=5.1017344600080736e-194
Epoch=40; Loss=5.1017344600080736e-194
Epoch=45; Loss=5.1017344600080736e-194
Epoch=50; Loss=5.1017344600080736e-194
Epoch=55; Loss=5.1017344600080736e-194
Epoch=60; Loss=5.1017344600080736e-194
Epoch=65; Loss=5.1017344600080736e-194
Epoch=70; Loss=5.1017344600080736e-194
Epoch=75; Loss=5.1017344600080736e-194
Epoch=80; Loss=5.1017344600080736e-194
Epoch=85; Loss=5.1017344600080736e-194
Epoch=90; Loss=5.1017344600080736e-194
Epoch=95; Loss=5.1017344600080736e-194
Epoch=100; Loss=5.1017344600080736e-194
```

Evaluate result

```julia
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
    # 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]
    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)
end
```

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

- Accuracy: Assuming the problem's category is that of classification, the accuracy score represents the percentage of the dataset that the model correctly predicted with no regard to any other statistics. For example, if the model correctly predicted 9 of out 10 datapoints, the accuracy will be 90%. This goes true for whatever the class the model predicted, as long as it is correct.
- Recall: With the above assumption, recall represents the percentage of True Positive (TP) among those that belong to the Positive group (True Positive + False Negative). Hence the formula: recall = TP / TP + FN
- Precision: Unlike recall, precision represents the percentage of TP among those that are predicted as Positive, that includes the wrongly predicted Negative datapoints. We have the formula: precision = TP / TP + FP
- F1: This is the harmonic mean of recall and precision. When calculating the recall or precision values, trying to lower one value will result in the other being raised. F1-score is used to prevent this, as the difference in both values will lower the F1-score. Proving that the model is inefficient.

TODO: Try out different learning rates. Give me your observations The data is very sensitive to the learning rate and the initial weights generated. With the learning rate $\alpha = 0.1$, often times the model could not generate a sufficient prediction, reaching 60% to 80% in accuracy but generally leaning towards either the negative or positive samples. There are some cases the model was able to accurately give a decision bounddary that separates two groups from one another, but the succession rate is quite low. This is the same for many other learning rates such as 0.06, 0.05, ... The submitted version contains the visualization of a good run with learning rate $\alpha = 0.01$, as the model converges almost immediately. However, trying to use the same learning rate for another data generation might not produce the same results. This holds true to the concept of gradient descent, as choosing a learning rate that puts the initial x_0 too far from x^* will result in a longer conversion. And the larger the learning rate, the higher the possibility that it could not converge fully after n epochs.

```
md"""
**TODO: Try out different learning rates. Give me your observations**
The data is very sensitive to the learning rate and the initial weights generated. With the learning rate $\alpha = 0.1$, often times the model could not generate a sufficient prediction, reaching 60% to 80% in accuracy but generally leaning towards either the negative or positive samples. There are some cases the model was able to accurately give a decision bounddary that separates two groups from one another, but the succession rate is quite low. This is the same for many other learning rates such as 0.06, 0.05, ...
The submitted version contains the visualization of a good run with learning rate $\alpha = 0.01$, as the model converges almost immediately. However, trying to use the same learning rate for another data generation might not produce the same results.
This holds true to the concept of gradient descent, as choosing a learning rate that puts the initial $x_0$ too far from $x^*$ will result in a longer conversion. And the larger the learning rate, the higher the possibility that it could not converge fully after $n$ epochs.
"""
```