Machine Learning Introduction

Part I: Logistic Regression – A Single Neuron Model

Matthieu Gurrisi

February 2026

In this document, we detail the intuition and mathematics behind this program, which trains a single neuron to perform binary classification.

1 Introduction

We consider a binary classification problem. Our dataset consists of m samples $\{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$, where each label $y^{(i)} \in \{0, 1\}$. Each sample is described by two features, $x^{(i)} = (x_1^{(i)}, x_2^{(i)}) \in \mathbb{R}^2$. As shown in Figure 1, the two classes are linearly separable, meaning that there exists a linear decision boundary that separates the two classes.

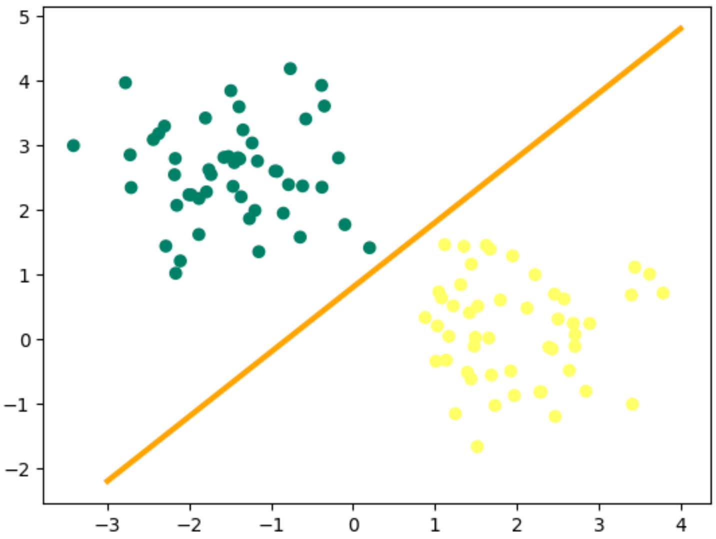

Figure 1: Two-dimensional training set for a binary classification task. Each point is a sample $x^{(i)} = (x_1^{(i)}, x_2^{(i)})$ colored by its label $y^{(i)} \in \{0, 1\}$. The orange line illustrates a linear separation.

Thus, we can predict the label $\hat{y}^{(i)}$ of any input $x^{(i)}$ by checking on which side of the orange decision boundary the point lies. Therefore, we need to determine the equation of the decision boundary.

To determine this boundary, we start with a simple linear model. We feed the neuron with the two input features x_1 and x_2 and assign a weight to each feature. Together with a bias term b, this defines the affine score

$$z(x_1, x_2) := w_1 x_1 + w_2 x_2 + b,$$

as illustrated in Figure 2. The *decision boundary* is the set of points for which the model is undecided, i.e., the line defined by $z = 0$. Our goal is therefore to learn the parameters w_1, w_2, and b so that the prediction y_{pred} matches the true class of each input x as well as possible.

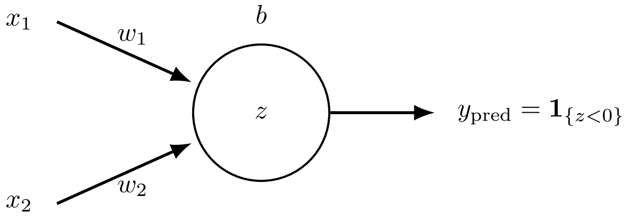

Figure 2: Perceptron model used for binary classification

In this perceptron model, the prediction is obtained by thresholding the score:

$$y_{\text{pred}} = \begin{cases} 1, & \text{if } z < 0, \\ 0, & \text{otherwise.} \end{cases}$$

While this produces a hard class decision, it does not provide any sense of probability.

To solve this, we replace the threshold with a smooth activation function: the sigmoid.

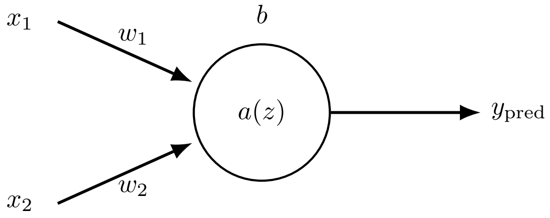

Figure 3: Sigmoid neuron model used for binary classification.

We use the sigmoid activation

$$a(z) = \frac{1}{1+e^{-z}}, \qquad \text{with } z = w_1x_1 + w_2x_2 + b.$$

Let us explain why this resolves the limitations of the perceptron:

- **Probabilistic interpretation.** Since $a(z) \in (0,1)$, we can interpret it as the model's estimated probability that the label equals 1. In particular,

  $$P(Y=1 \mid x) = a(z), \qquad P(Y=0 \mid x) = 1 - a(z),$$

  thus, we have

  $$Y \mid x \sim \text{Bernoulli}(a(z))*$$

  * In Part 3, we will see that this is not entirely true and explore an alternative: *the softmax.*

- **Smoothness for backpropagation.** Unlike a hard threshold, the sigmoid is differentiable. During training, we will update the parameters w_i and b so that the decision boundary $z = 0$ becomes optimal. For gradient-based optimization to make sense, small parameter changes should lead to small changes in the output. Using a first-order Taylor expansion, we obtain

  $$\delta a \approx \sum_i \frac{\partial a}{\partial w_i}\,\delta w_i \;+\; \frac{\partial a}{\partial b}\,\delta b,$$

  which is exactly the kind of behavior we need for gradient descent.

2 The Loss Function

To quantify the errors made by the model, a loss function is used. There is a wide range of loss functions to choose from. We will use the *logarithmic loss* (log-loss), defined as

$$L = -\frac{1}{m}\sum_{i=1}^{m}\Big(y_i \log(a_i) + (1-y_i)\log(1-a_i)\Big),$$

where m is the number of data points, y_i is the label of the i-th example, and a_i is the model output for the i-th example.

Let us now explain why this choice is natural from a probabilistic point of view.

2.1 Maximum Likelihood Perspective

We consider the dataset labels

$$Y = \{y^{(1)}, y^{(2)}, \ldots, y^{(m)}\}$$

as realizations of random variables defined on a probabilistic space. Given an input $x^{(i)}$, the model outputs a value $a_i = a(z^{(i)}) \in (0,1)$, which we interpret as the probability that the corresponding label equals 1. In other words, we assume that

$$Y^{(i)} \mid x^{(i)} \sim \text{Bernoulli}(a_i).$$

Under this assumption, the probability of observing a particular label $y^{(i)}$ is given by

$$P(Y^{(i)} = y^{(i)} \mid x^{(i)}) = a_i^{y^{(i)}}(1-a_i)^{1-y^{(i)}}.$$

We further assume that the observations are independent. The likelihood of the entire dataset is therefore

$$\mathcal{L}(a_1,\ldots,a_m) = \prod_{i=1}^{m} a_i^{y^{(i)}}(1-a_i)^{1-y^{(i)}}.$$

Intuitively, maximizing this likelihood means choosing the model parameters such that the observed labels are as probable as possible under the model.

2.2 Log-Likelihood and Numerical Stability

In practice, the likelihood $\mathcal{L}$ is a product of many terms smaller than 1. As the number of samples m increases, this product rapidly tends toward zero, which makes direct optimization numerically unstable.

To solve this issue, we maximize the *log-likelihood* instead. Since the logarithm is a strictly increasing function, maximizing the likelihood is equivalent to maximizing its logarithm:

$$\log\mathcal{L} = \sum_{i=1}^{m}\Big(y^{(i)}\log(a_i) + (1-y^{(i)})\log(1-a_i)\Big).$$

Rather than maximizing the log-likelihood, it is usual to minimize its opposite. Dividing by m gives the average loss per data point:

$$-\frac{1}{m}\log\mathcal{L} = -\frac{1}{m}\sum_{i=1}^{m}\Big(y^{(i)}\log(a_i) + (1-y^{(i)})\log(1-a_i)\Big).$$

We recover exactly the log-loss function introduced earlier. This shows that minimizing the log-loss is equivalent to performing maximum likelihood estimation under a Bernoulli model for the labels.

Our goal is now to train the neuron, i.e., to determine the parameters w_i and b that minimize the loss function. To do so, we use *gradient descent.*

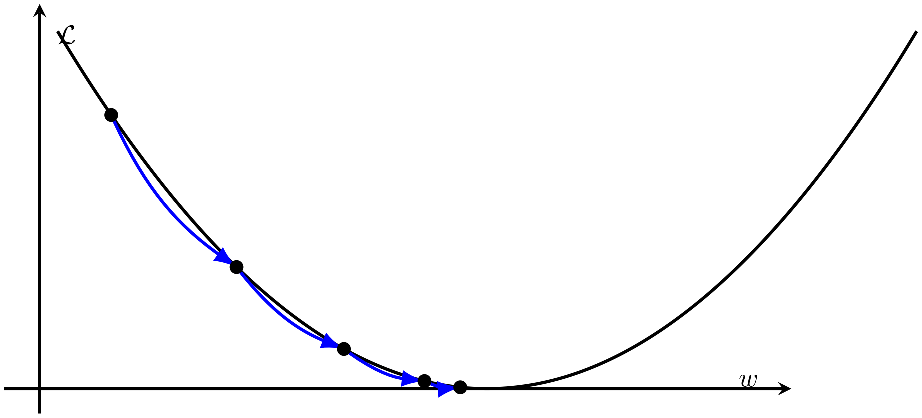

Figure 4: Illustration of gradient descent on a one-dimensional loss function $\mathcal{L}(w)$.

3 Gradient Descent

Gradient descent is an iterative optimization algorithm that updates the parameters in the direction of the negative gradient in order to reduce the loss, as illustrated in Figure 4.

This iterative process runs as follows:

$$\begin{cases} W_{i+1} = W_i - \alpha \, \dfrac{\partial \mathcal{L}}{\partial W_i} \\ b_{i+1} = b_i - \alpha \, \dfrac{\partial \mathcal{L}}{\partial b_i} \end{cases}$$

where i denotes the i-th iteration and

$$W = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

Moreover, the gradient with respect to W is the vector of partial derivatives:

$$\frac{\partial \mathcal{L}}{\partial W} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_1} \\ \frac{\partial \mathcal{L}}{\partial w_2} \end{bmatrix}$$

Therefore, we need to determine

$$\frac{\partial \mathcal{L}}{\partial w_i} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial b}$$

3.1 Notation

Before deriving these gradients, let us clarify the notation used throughout the rest of this document.

We denote the average loss over the dataset by

$$\mathcal{L} = \frac{1}{m} \sum_{i=1}^{m} L^{(i)},$$

where $L^{(i)}$ is the loss associated with the i-th example :

$$L^{(i)} = -\Big(y^{(i)} \log\left(a^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - a^{(i)}\right) \Big).$$

For each training example, we define the scalar score

$$z^{(i)} = W x^{(i)} + b,$$

where W is a row vector and $x^{(i)}$ is a column vector:

$$W = \begin{bmatrix} w_1 & w_2 \end{bmatrix}, \qquad x^{(i)} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \end{bmatrix}.$$

The model output is obtained by applying the activation function:

$$a^{(i)} = a\left(z^{(i)}\right).$$

We also introduce the following column vectors collecting all examples:

$$A = \begin{bmatrix} a^{(1)} \\ a^{(2)} \\ \vdots \\ a^{(m)} \end{bmatrix}, \qquad Y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}.$$

3.2 Detailed Gradient Computation

We begin by calculating the partial derivative of the loss with regard to a weight w_j. Recall that

$$\mathcal{L} = \frac{1}{m} \sum_{i=1}^{m} L^{(i)}.$$

By linearity of differentiation, we obtain

$$\frac{\partial \mathcal{L}}{\partial w_j} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial L^{(i)}}{\partial w_j}.$$

Using the chain rule, we decompose this derivative as

$$\frac{\partial L^{(i)}}{\partial w_j} = \frac{\partial L^{(i)}}{\partial a^{(i)}} \cdot \frac{\partial a^{(i)}}{\partial z^{(i)}} \cdot \frac{\partial z^{(i)}}{\partial w_j}.$$

Each term can be computed explicitly:

$$\frac{\partial L^{(i)}}{\partial a^{(i)}} = -\left(\frac{y^{(i)}}{a^{(i)}} - \frac{1 - y^{(i)}}{1 - a^{(i)}}\right), \qquad \frac{\partial a^{(i)}}{\partial z^{(i)}} = a^{(i)}(1 - a^{(i)}), \qquad \frac{\partial z^{(i)}}{\partial w_j} = x_j^{(i)}.$$

Combining these expressions yields the simplification

$$\frac{\partial L^{(i)}}{\partial w_j} = \left(a^{(i)} - y^{(i)}\right) x_j^{(i)}.$$

Finally, we obtain

$$\boxed{\frac{\partial \mathcal{L}}{\partial w_j} = \frac{1}{m} \sum_{i=1}^{m} \left(a^{(i)} - y^{(i)}\right) x_j^{(i)}}$$

We proceed similarly for the bias term b. Using again

$$\mathcal{L} = \frac{1}{m} \sum_{i=1}^{m} L^{(i)},$$

we have, by linearity of differentiation,

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial L^{(i)}}{\partial b}.$$

Since b only influences $L^{(i)}$ through $z^{(i)}$, the chain rule gives

$$\frac{\partial L^{(i)}}{\partial b} = \frac{\partial L^{(i)}}{\partial z^{(i)}} \cdot \frac{\partial z^{(i)}}{\partial b}.$$

From the previous derivation, we already know that

$$\frac{\partial L^{(i)}}{\partial z^{(i)}} = a^{(i)} - y^{(i)},$$

and since $z^{(i)} = Wx^{(i)} + b$, we have

$$\frac{\partial z^{(i)}}{\partial b} = 1.$$

Therefore,

$$\frac{\partial L^{(i)}}{\partial b} = a^{(i)} - y^{(i)}.$$

Finally, we obtain

$$\boxed{\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} \left(a^{(i)} - y^{(i)}\right)}$$

Remark. All the derivations presented above remain valid if the dimensionality of the input space increases. In particular, if each data point $x^{(i)}$ is described by d features instead of two, the weight vector simply becomes $W = (w_1, \dots, w_d)$ and the score is given by

$$z^{(i)} = \sum_{j=1}^{d} w_j x_j^{(i)} + b.$$

The expressions of the gradients remain unchanged in form: each partial derivative $\partial \mathcal{L}/\partial w_j$ is obtained by summing the contributions of all samples weighted by the corresponding feature $x_j^{(i)}$. This scalability property is one of the key strengths of the linear model and naturally motivates the vectorized formulation introduced in the next section.

4 Toward a Vectorized Implementation

So far, the gradients have been derived using scalar expressions. While this formulation is useful for understanding the underlying mathematics, it is not optimal for efficient computation. We now rewrite all expressions in a vectorized (matrix-based) form.

4.1 Vectorized Notation

We stack all input samples into a single matrix

$$X = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ \vdots & \vdots \\ x_1^{(m)} & x_2^{(m)} \end{bmatrix} \in \mathbb{R}^{m \times 2}.$$

Similarly, the labels are grouped into a column vector

$$Y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} \in \mathbb{R}^m.$$

The parameter vector is written as

$$W = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \in \mathbb{R}^2.$$

4.2 Model in Vector Form

Using this notation, the linear scores for all samples can be computed at once:

$$Z = XW + b,$$

where the bias b is broadcasted to all components of Z.

Applying the sigmoid activation elementwise yields

$$A = \sigma(Z),$$

where

$$A = \begin{bmatrix} a^{(1)} \\ a^{(2)} \\ \vdots \\ a^{(m)} \end{bmatrix}.$$

4.3 Loss Function

The log-loss over the dataset can then be written compactly as

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^{m} \Big(y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)}) \Big).$$

4.4 Vectorized Gradients

From the scalar derivations obtained earlier, we immediately recover the vectorized gradients:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{1}{m} X^\top (A - Y), \qquad \frac{\partial \mathcal{L}}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)}).$$

These expressions are mathematically equivalent to the scalar gradients, but allow all computations to be carried out efficiently using matrix operations.

This vectorized formulation directly corresponds to the implementation used in this notebook, where all computations are performed using matrix operations for efficiency and clarity.

In the next part, we extend this framework by combining multiple neurons into a *multilayer perceptron*. This allows us to move beyond linear decision boundaries and model non-linearly separable data.