Introduction to Logistic Regression: Fundamentals and Applications

Analysis of the basic principles, mathematical formulation and its application in probability prediction and decision making.

Hernández Pérez Erick Fernando

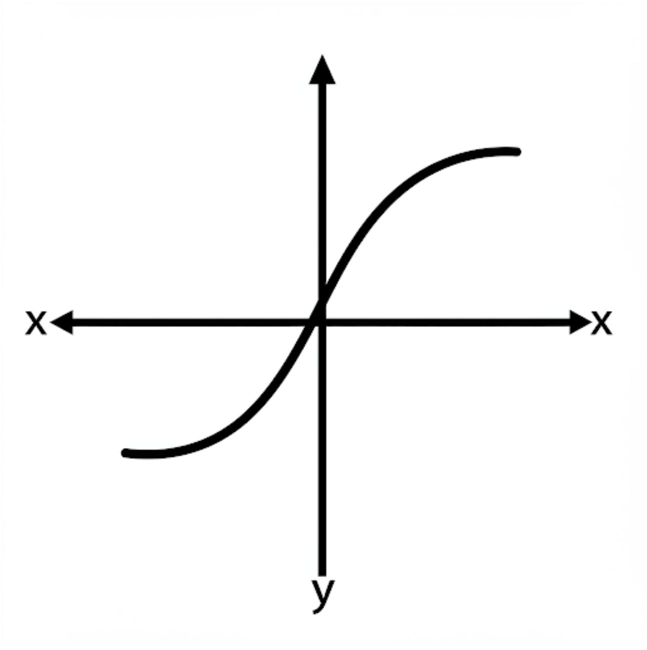


Figure 1

In the world of machine learning, classification is one of the most powerful tools for automated decision-making. From filtering emails as spam or legitimate to identifying objects in images, classification models are constantly used to turn data into concrete answers.

When faced with a classification problem, the first step is to define how many categories are needed. Some scenarios are binary: a medical model determining whether an X-ray shows signs of a disease (yes or no). Others instead require multiple classes, such as an application that recognizes different species of flowers from a photo.

Beyond the number of classes, the question also arises of how confident the model is about its predictions. It is not enough to say "this image is a cat" but it is useful to know the probability assigned to each possible answer. This is where cross-entropy comes into play, a key measure for training models capable of making accurate and well-calibrated predictions.

Thus, behind every decision made by a classification model there is a series of calculations designed to minimize error and maximize confidence in the results.

Raschka et al. (2022)[2] explain the No Free Lunch (NFL) theorem in the context of classifiers, which states that there is no universally superior classification model for all problems. In other words, the performance of a classifier depends on the type of data and the specific context in which it is applied. A model that performs well on one dataset may be inefficient on another, so the choice of classifier should be based on the nature of the problem, the distribution of the data, and the specific goals of the analysis. This underscores the necessity of evaluating multiple models and fine-tuning their parameters according to their performance on a given problem.

For this, metrics derived from the confusion matrix are used, which summarizes the classification results in terms of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). See the seventh entry: Analysis of the Confusion Matrix and Performance Metrics in Classification Models.

But let's start by clarifying the most important thing: What is a classifier?

Classifiers

“In mathematical notation, a classifier is a function $y = f(x)$, where x is the input data item and y is the output category” (Mattmann, 2020) [1]. This highlights the main difference between regression and classification: while regression predicts continuous values, classification assigns discrete labels to data points, typically corresponding to predefined categories. See the fifth entry: Applying Regression in TensorFlow 2: Methods and Techniques.

Just as in linear regression, the learning algorithm explores the space of functions defined by the underlying model, denoted as M. In the case of linear regression, this model is parameterized by w, allowing us to evaluate the function $y = M(w)$ by measuring its associated cost. Ultimately, the objective is to select the value of w that minimizes this cost.

Classification through Linear Regression

I know what you might be thinking right now: “Didn’t we say these were two separate tasks? What a rip-off!” But before you stop reading, hear me out: this is actually a natural step. Let’s try to work with what we have at hand; after all, we never said it was impossible.

If we consider a binary classification problem, we could try to model it using linear regression, where the predicted values are interpreted as scores that can be thresholded to determine the assigned class.

For this experiment, two sets of data will be generated following normal distributions:

- A group of points with mean 5 and standard deviation 1, labeled as class 0.
- A second group with mean 2 and standard deviation 1, labeled as class 1.

These points will be represented in a graph where the X axis corresponds to the input values and the Y axis to the binary labels (0 or 1). This visualization will allow us to observe how the data is distributed in the input space.

The model used will follow the classic equation of a straight line: $\hat{y} = w_1 x + w_0$ Where represents w_1 the slope and w_0 the intersection with the Y axis. Initially, these parameters are set to zero and updated using a stochastic gradient descent (SGD) algorithm to minimize the loss function. Training will be carried out by minimizing the mean square error (MSE).

After training, a straight line is obtained that attempts to separate both classes. This straight line is superimposed on the original graph, allowing us to analyze how well the model has managed to capture the separation between the classes.

Linear Regression for Binary Classification

```python
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

x_label0 = np.random.normal(5, 1, 10)
x_label1 = np.random.normal(2, 1, 10)
xs = np.append(x_label0, x_label1)
labels = [0.] * len(x_label0) + [1.] * len(x_label1)

lr = 0.01
training_epochs = 1000

def model(X, w):
    return w[1] * X + w[0]

w = tf.Variable([0., 0.], dtype=tf.float32, name='parameters')
optimizer = tf.keras.optimizers.SGD(learning_rate=lr)

for epoch in range(training_epochs):
    with tf.GradientTape() as tape:
        y_pred = model(xs, w)
        loss = tf.reduce_mean(tf.square(y_pred - labels))
    gradients = tape.gradient(loss, [w])
    optimizer.apply_gradients(zip(gradients, [w]))

    if epoch % 100 == 0:
        print(f"Epoch {epoch}: Loss = {loss.numpy()}")

print(f"Trained parameters: w0 = {w.numpy()[0]}, w1 = {w.numpy()[1]}")
w_val = w.numpy()

fig, ax = plt.subplots(figsize=(8, 5))

equation_text = f"y = {w_val[1]:.4f}x + {w_val[0]:.4f}"
ax.text(5.5, 0.5, equation_text, fontsize=12, color='green', ha='center')

ax.scatter(x_label0, [0] * len(x_label0), color='blue', label='Class 0',
↪   edgecolors='black', s=100, alpha=0.7)
ax.scatter(x_label1, [1] * len(x_label1), color='red', label='Class 1',
↪   edgecolors='black', s=100, alpha=0.7)

all_xs = np.linspace(0, 10, 100)
ax.plot(all_xs, all_xs * w_val[1] + w_val[0], color='green', linestyle='--', linewidth=2,
↪   label='Decision Boundary')

ax.axvline(x=3.5, color='black', linestyle=':', linewidth=2, label='Threshold')

ax.set_title('Linear Regression for Binary Classification', fontsize=14,
↪   fontweight='bold')
ax.set_xlabel('Feature Value (X)', fontsize=12)
ax.set_ylabel('Predicted Class', fontsize=12)
ax.set_yticks([0, 1])
ax.set_yticklabels(['Class 0', 'Class 1'])
ax.legend()
ax.grid(True, linestyle='--', alpha=0.6)

plt.show()
```

In this case, we can say that all points that fall to the right of 3.5 (our threshold) in our regression are more likely to belong to class 0 than to class 1. That is:

$$\text{Class} = \begin{cases} 0, & \text{if } w_1x + w_0 \geq 3.5 \\ 1, & \text{if } w_1x + w_0 < 3.5 \end{cases}$$

Logistic Regression

“Logistic regression is a classification model that is very easy to implement and performs very well on linearly separable classes. It is one of the most widely used algorithms for classification in industry” (Raschka, 2022)[2]. We will study logistic regression for the case of a binary classifier.

The odds in favor of a particular event can be written as $\frac{p}{1-p}$ where p is the probability of our positive class (the class that we want to predict, it could be fraud, ill, etc.; our class label $y = 1$).

We can define our probability p as a conditional probability that a particular example belongs to a certain class 1 given its features, $\mathbf{x}$:

$$p := p(y = 1|\mathbf{x})$$

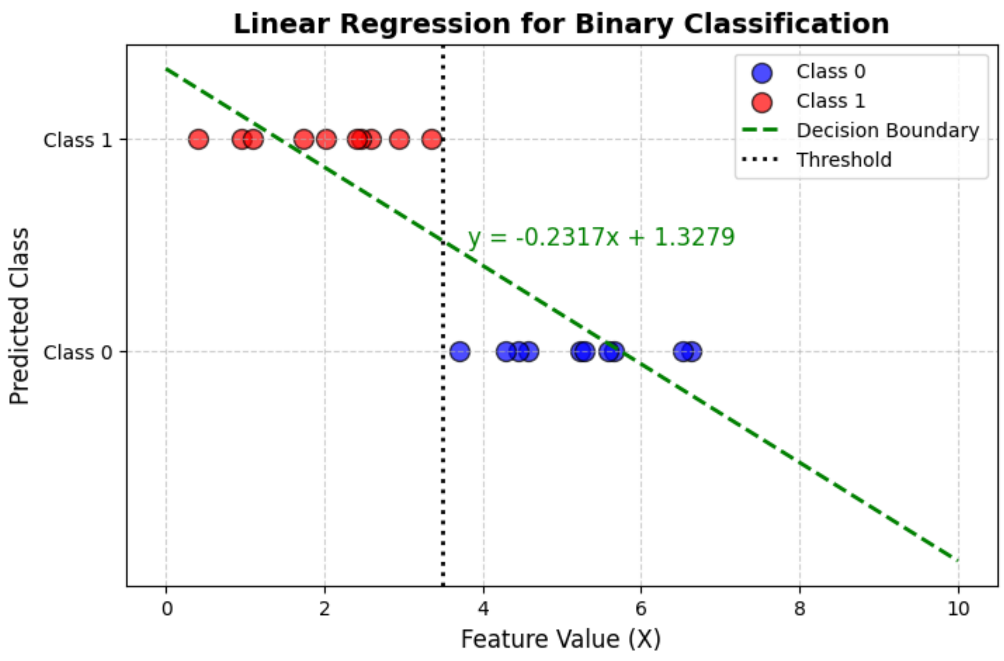


Figure 2: Linear Regression for Binary Classification, our poor result

Now, we can define de logit function (log-odds), that takes input values in the range 0 to 1 and transforms them into values over the entire real-number range (log refers to the natural logaritm in this context).

$$\operatorname{logit}(p) = \log\left(\frac{p}{1-p}\right)$$

We assume, under the logistic model, that there is a linear relationship between the weighted inputs and the log-odds. This is:

$$\operatorname{logit}(p) = w_1 x_1 + ... + w_n x_n + b = \sum_{i=j} w_j x_j + b = \mathbf{w}^T \mathbf{x} + b$$

But what we are actually interested in is the probability p, the class-membership probability of an example given its features. So we need the inverse of this function to map the real-number range back to a $[0, 1]$ range for the probability p.

The inverse of the logit function is the logistic sigmoid function, commonly known as the sigmoid function (Figure 3a).

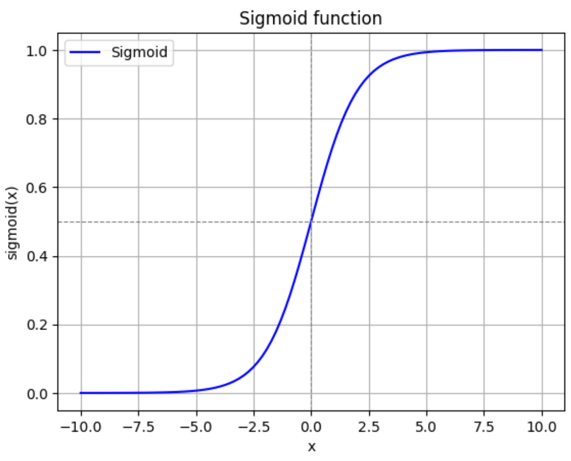
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Here, z represents the net input, which is the linear combination of weights and inputs (i.e., the features associated with the training examples).

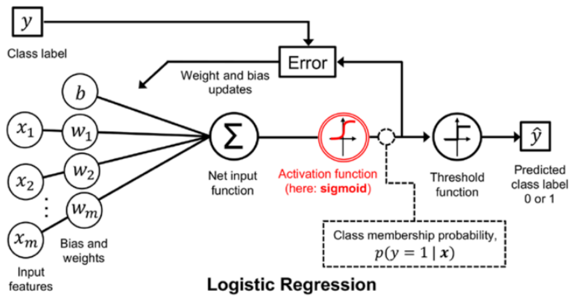
$$z = \mathbf{w}^T \mathbf{x} + b$$

The sigmoid function becomes our activation function. To understand this visually we can refer to the great illustration provided by Raschka et al. (2022) (Figure 3b). For a deeper dive into the background of weighted sum, see Entry 4: Analysis of the Rosenblatt Perceptron: Convergence and Practical Application.

The output of the sigmoid function is then interpreted as the probability of a particular example belonging to class 1, $\sigma(z) = p(y = 1|\mathbf{x}; \mathbf{w}, b)$, given its features, $\mathbf{x}$, and parameterized by the weights and bias, $\mathbf{w}$ and b. The predicted probability can then simply be converted into a binary outcome via a threshold function.



(a) Sigmoid function



(b) Logistic Regression (In red the activation function)[2]

Figure 3

The loss function

We want to maximize our likelihood, $\mathcal{L}$ (It is a measure of how well a model explains the observed data. Likelihood tells us how likely it is that the data we observe occurred under a certain model), when we build a logistic regression model. Let's assume that the individual examples in our dataset are independent of one another (Keep in mind a Bernoulli variable).

$$\mathcal{L}(\mathbf{w},b|\mathbf{x}) = p(y|\mathbf{x};\mathbf{w},b) = \prod_{i=1}^{n} p(y^{(i)}|\mathbf{x}^{(i)};\mathbf{w},b) = \prod_{i=1}^{n}(\sigma(z^{(i)}))^{y^{(i)}}(\sigma(1-z^{(i)}))^{1-y^{(i)}}$$

It is easier to maximize the (natural) log of this equation, the log-likelihood:

$$l(\mathbf{w},b|\mathbf{x}) = \log\mathcal{L}(\mathbf{w},b|\mathbf{x}) = \sum_{i=1}[y^{(i)}\log(\sigma(z^{(i)})) + (1-y^{(i)})\log(1-\sigma(z^{(i)}))]$$

But, if we want to minimize because we don't want to use Gradient ascent and because this is no a loss function, we need to change a little bit our log-likelihood.

$$L(\mathbf{w},b) = \sum_{i=1}^{n}[-y^{(i)}\log(\sigma(z^{(i)})) - (1-y^{(i)})\log(1-\sigma(z^{(i)}))] = -\sum_{i} y_i\cdot\log(\hat{y}_i)$$

Where we can recognize the cross entropy at the end.

To understand how the weights are updated when using our loss function, we will first differentiate with respect to the j-th weight, w_j. To simplify the notation, we will consider our sigmoid function, $\sigma(z)$, as σ

$$\frac{\partial L}{\partial w_j} = \frac{\partial L}{\partial \sigma}\frac{\partial \sigma}{\partial z}\frac{\partial z}{\partial w_j}$$

Let's calculate the partials separately:

$$\begin{aligned}
\frac{\partial L}{\partial \sigma} &= -\frac{y}{\sigma} + \frac{1-y}{1-\sigma} = \frac{\sigma - y}{\sigma(1-\sigma)}\\
\frac{\partial \sigma}{\partial z} &= -(1+e^{-z})^{-2}(-e^{-z}) = \frac{e^{-z}}{(1+e^{-z})^2} = \sigma(1-\sigma)\\
\frac{\partial z}{\partial w_j} &= x_j
\end{aligned}$$

Therefore

$$\frac{\partial L}{\partial w_j} = \frac{\sigma - y}{\sigma(1-\sigma)}\sigma(1-\sigma)x_j = -(y-\sigma)x_j$$

Thus, and changing the direction of the gradient, we have the updates of the weights and the bias:

$$w_j := w_j + \eta(y-\sigma)x_j$$
$$b := b + \eta(y-\sigma)$$

An example applied in TensorFlow2

Logistic Regression

```python
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

x_label0 = np.random.normal(-4, 2, 1000)
x_label1 = np.random.normal(4, 2, 1000)
xs = np.append(x_label0, x_label1).astype(np.float32)
labels = np.array([0.] * len(x_label0) + [1.] * len(x_label1), dtype=np.float32)

plt.scatter(xs, labels, alpha=0.3)

lr = 0.01
training_epochs = 1000

def model(X, w):
    return w[1] * X + w[0]

def sigmoid(x):
    return 1. / (1. + np.exp(-x))

w = tf.Variable([0., 0.], dtype=tf.float32, name="parameters")

optimizer = tf.keras.optimizers.Adam(learning_rate=lr)
loss_fn = tf.losses.BinaryCrossentropy(from_logits=True)

for epoch in range(training_epochs):
    with tf.GradientTape() as tape:
        y_pred = model(xs, w)
        loss = loss_fn(labels, y_pred)

    gradients = tape.gradient(loss, [w])
    optimizer.apply_gradients(zip(gradients, [w]))

    if epoch % 100 == 0:
        print(f"Epoch {epoch}: Loss = {loss.numpy()}")

print(f"Trained parameters: w0 = {w.numpy()[0]}, w1 = {w.numpy()[1]}")

w_val = w.numpy()
all_xs = np.linspace(-10, 10, 100)
plt.plot(all_xs, sigmoid(all_xs * w_val[1] + w_val[0]), label="Sigmoid Model")
plt.scatter(xs, labels, alpha=0.3, label="Data")
plt.legend()
plt.show()
```

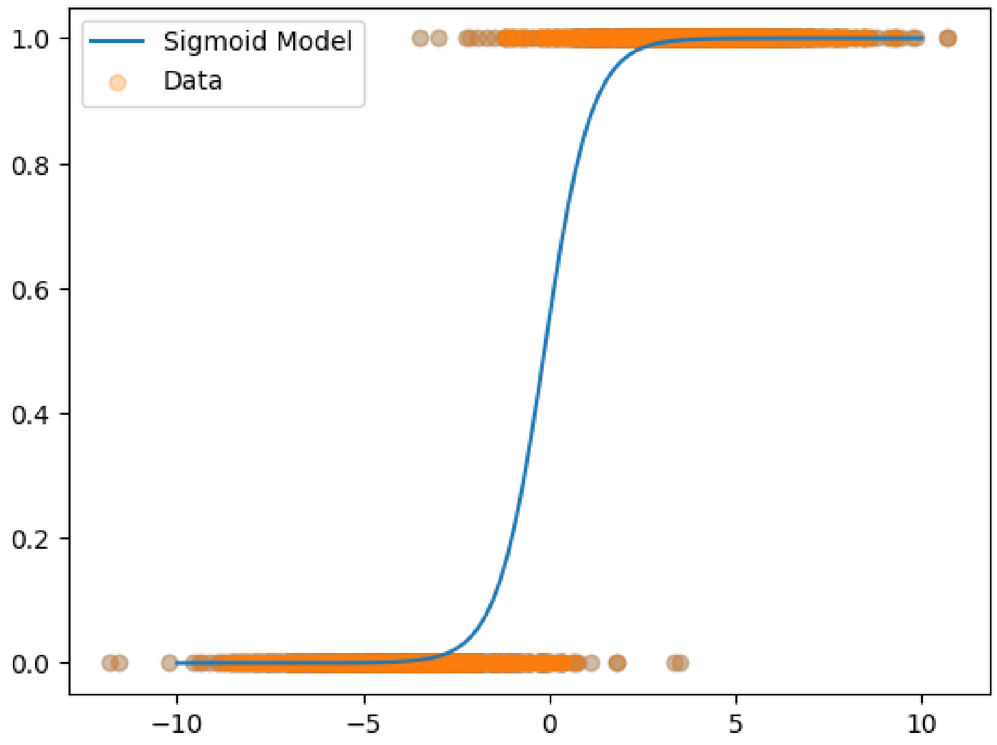


Figure 4: The logistic model

To begin, we import essential libraries for machine learning, numerical operations, and visualization. We create a dataset with two groups of points, each drawn from a normal distribution. One group is centered around a negative value and the other around a positive value, assigning labels accordingly. This dataset helps demonstrate logistic regression in a binary classification task.

We then define the logistic regression model as a linear function with trainable parameters and use the sigmoid function to convert real-valued predictions into probabilities. The sigmoid function ensures that outputs fall within the range of 0 to 1, making it suitable for classification.

Next, we initialize model parameters, which include a bias term and a weight coefficient. We choose the Adam optimizer to update these parameters dynamically during training. Additionally, we define the binary cross-entropy loss function, which measures how well the model's predictions align with actual labels.

Training the model involves iterating through multiple epochs, where we compute predictions, calculate the loss, and update parameters using gradient descent. By leveraging TensorFlow's automatic differentiation, gradients are efficiently computed and applied to optimize the model. Periodically, we print the loss value to monitor training progress.

Once training is complete, we retrieve the learned parameters and visualize the decision boundary by plotting the sigmoid function over the trained model. This step helps interpret how well the model separates the two classes. The visualization includes data points and the corresponding probability curve, demonstrating the effectiveness of logistic regression in classification tasks.

Applications of logistic regression

Logistic regression is a statistical model widely used in various disciplines due to its ability to predict probabilities and facilitate data-driven decision making. Below are some of its main applications:

- **Probability prediction**: One of the most common applications of logistic regression is the estimation of the probability of an event occurring. For example, in the medical field, it can be used to predict the probability of a patient developing a disease based on certain risk factors. Similarly, in the financial sector, it is used to assess the probability of a customer defaulting on a loan.

- **Data-driven decision making**: By providing a quantitative estimate of the risk or probability of an event, logistic regression enables informed decision making. In human resources, for example, it can be used to identify candidates with a higher probability of success in a job. In marketing, it allows customers to be segmented based on the probability that they will respond to an advertising campaign, optimizing commercial strategies.
- **Binary classification and pattern detection**: Logistic regression is widely used in classification problems where the output variable has two categories. A practical case is fraud detection in banking transactions, where the model assigns probabilities to each transaction to determine whether it is legitimate or fraudulent.

References

[1] Mattmann, C. (2020). *Machine Learning with TensorFlow.* Manning.

[2] Raschka, S., Liu, Y. & Mirjalili, V. (2022). *Machine Learning with PyTorch and Scikit-Learn.* Packt.