Logistic Regression

Mads Møller

LinkedIn: https://www.linkedin.com/in/madsmoeller1/

Mail: mads.moeller@outlook.com

This paper is the second in the series about machine learning algorithms. Logistic Regression is our first **classification algorithm**. Even though *regression* appears in the name, the Logistic Regression algorithm is a classification algorithm, not a regression algorithm.

Logistic regression is highly connected with linear regression. Actually, what you will see is that linear regression is happening within logistic regression. First of all, logistic regression is used for classification problems, so let us just define what we mean when we say "classification". The formal definition of classification is

"Classification is a process of categorizing a given set of data into classes"

An example of classification could be determining whether a person has cancer or not based on an x-ray photo. This is actually a great example of what *deep learning* is used for in real life. Another example could be classifying whether an email is spam or not. Logistic regression can be applied on such problems. In order to understand how the logistic regression algorithm works, let us inspect the simple case of **binary logistic regression**.

1 Binary Logistic Regression

Even though logistic regression is a classification algorithm the output is continuous. We inspect a simple example where we have a binary outcome. We define the linear regression as z, which we are going to use in order to understand logistic regression:

$$z = w_0 + w_1 x = \underline{w}^T \underline{x} \tag{1}$$

In binary logistic regression we can either classify our data as 0 or 1. The way logistic regression works is by squeezing the output of linear regression to be between 0 and 1. Logistic regression is able to transform the linear output with the help of the *sigmoid function*. The sigmoid function (or sometimes called the logistic function) is defined mathematically as:

$$\sigma(z) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-(\underline{w}^T \underline{x})}} \tag{2}$$

The output of the sigmoid function is probabilistic meaning that its output reflect the probability of the data belonging to each class. If we inspect the sigmoid function visually it is clear that the output will be between 0 and 1:

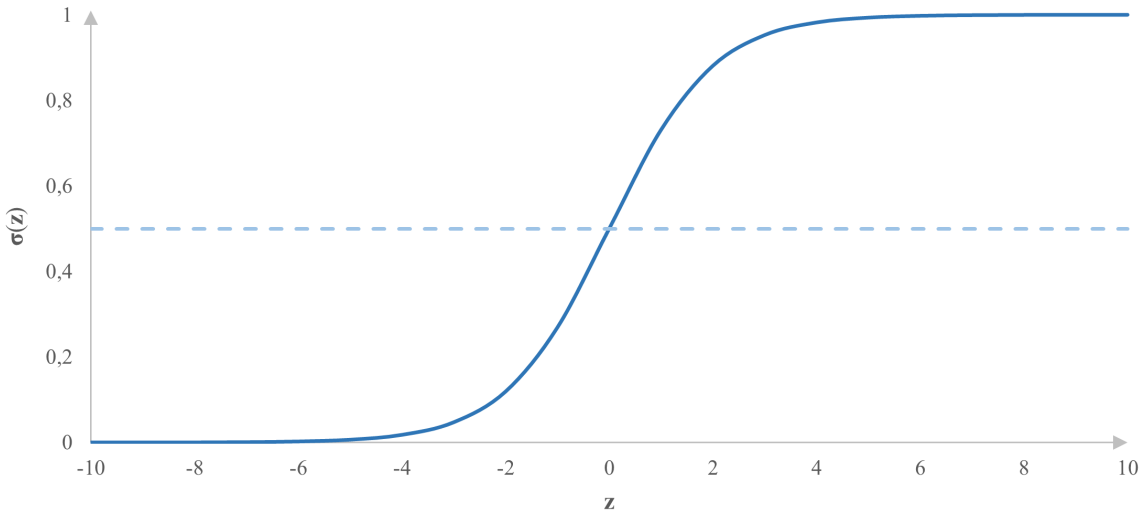


Figure 1: Sigmoid Function

In order to decide when we would classify an output from the sigmoid function as either 1 or 0 we need to set up a **decision boundary**. The decision boundary maps our probability score to a discrete class (0 or 1). In figure 1 the decision boundary is illustrated as the dotted line. The decision boundary is the line that separates the area where y = 0 and where y = 1. It is created by our hypothesis function. In general, the decision boundary is set with a threshold of 0.5, such that:

$$\sigma(z) \geq 0.5 \rightarrow \hat{y} = 1$$

$$\sigma(z) < 0.5 \rightarrow \hat{y} = 0$$

Our **hypothesis function**, which is denoted by h_w, tells us what the probability is of that point belonging to the class "1":

$$h_w(\underline{x}) = p(y = 1|\underline{x}, \underline{w}) \tag{3}$$

Because we have a binary classification problem it must also apply that:

$$p(y = 0|\underline{x}, \underline{w}) = 1 - p(y = 1|\underline{x}, \underline{w}) \tag{4}$$

It is worth mentioning that the input to the sigmoid function doesn't need to be linear. We could have $\underline{w}^T\underline{x} = w_0 + w_1x_1^2 + w_2x_2^2$, such that we could describe split our classes with a circle.

1.1 Cost Function

As in linear regression and all other machine learning algorithms we also has a cost function for logistic regression. The cost function is used to update the weights, which are going to change the decision boundary. In linear regression we used the *Mean Squared Error (MSE)* as our cost function. In logistic

regression we use a cost function called **Cross-Entropy**. This cost function can be divided into two seperate cost functions. One for $y = 0$ and one for $y = 1$:

$$J(\underline{w}) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_w(x^{(i)}, y^{(i)}) \tag{5}$$

Like for the linear regression model we also need to be able to compute the cost of a function for logistic regression:

$$Cost(h_w(x), y) = \begin{cases} -log(h_w(x)) & \text{if } y = 1 \\ -log(1 - h_w(x)) & \text{if } y = 0 \end{cases} \tag{6}$$

Were $h_w(x)$ is some value predicted and y is the actual value. But why does this make sense? Let us illustrate the cost function:

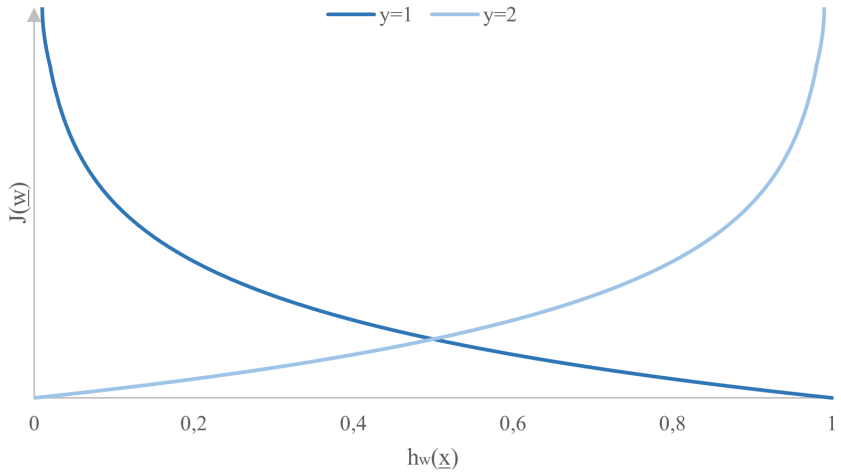


Figure 2: Cost function

Note that writing the cost function in this way guarantees that $J(\underline{w})$ is convex for logistic regression. When the actual value of a prediction is 1 and our hypothesis function returns 0.7 our cost will be very low. However, if we observe a hypothesis of 0.1 we will have a very high cost. This can be seen from figure 2. If we observe an actual value of 0 an we see a hypothesis of 0.1 it means that we have a very low cost (the light blue graph in figure 2). However, if we observe a hypothesis of 0.9 it will have a high cost. We can rewrite the cost from equation 6:

$$Cost(h_w(x), y) = -y \cdot log(h_w(x)) - (1 - y)log(1 - h_w(x)) \tag{7}$$

Therefore, we can write our cost function from 5 as:

$$J(\underline{w}) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_w(x^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \cdot log(h_w(x^{(i)})) + (1 - y^{(i)}) \cdot log(1 - h_w(x^{(i)})) \tag{8}$$

However, as in the case of linear regression it is easier to implement these algorithms in vectorized format. Therefore we can now rewrite the cost function from equation 8 in vectorized format:

$$\underbrace{\underline{h_w}}_{m\times 1} = \sigma(\underbrace{\underline{\underline{X}}}_{m\times(n+1)} \cdot \underbrace{\underline{w}}_{(n+1)\times 1})$$

$$\underbrace{J(\underline{w})}_{1\times 1} = \frac{1}{m}\left(-\underbrace{\underline{y}^T}_{1\times m} log(\underbrace{\underline{h}_w}_{m\times 1}) - \underbrace{(1-\underline{y})^T}_{1\times m} log(1-\underbrace{\underline{h}_w}_{1\times m}) \right) \tag{9}$$

1.2 Gradient Descent

For the logistic regression algorithm we will use gradient descent again as our optimization algorithm. In practice other optimization algorithms are often used. Specifically an algorithm called **Adam**, which we will cover later in this series. For now we will just stick with the gradient descent algorithm as our optimizer. Our problem is the same as in the case of linear regression:

$$\min_{\underline{w}} J(\underline{w})$$

We will minimize our cost by simultaneously updating the weights by using gradient descent:

$$w_j = w_j - \alpha\frac{\partial J}{\partial w_j} = w_j - \frac{\alpha}{m}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})x_j^{(i)} \tag{10}$$

A vectorized implementation of this algorithm can be implemented as:

$$\underline{w} = \underline{w} - \frac{a}{m}\underline{\underline{X}}^T(\underline{h}_w - \underline{y}) \tag{11}$$

As we see the gradient descent algorithm can now easily be implemented in Python. Before implementation let us inspect the case of **multi-class classification**.

2 Multi-Class Logistic Regression

In the last section we saw how one could use the logistic regression algorithm for binary classification. However, it is not always the case that we only have two different classes. One might as well have many different classes. But how does the algorithm work, if we are having more than two classes?

It is actually straightforward how the logistic regression algorithm works on multi-class problems. It uses a method called **one-vs-all**. The *one-vs-all* is an algorithm for multi-class classification problems. It the multi-class classification problem into binary classification problems. By doing so the algorithm is looking at "one" class vs. all others as a binary problem. The concept of one-vs-all is illustrated below in figure 3:

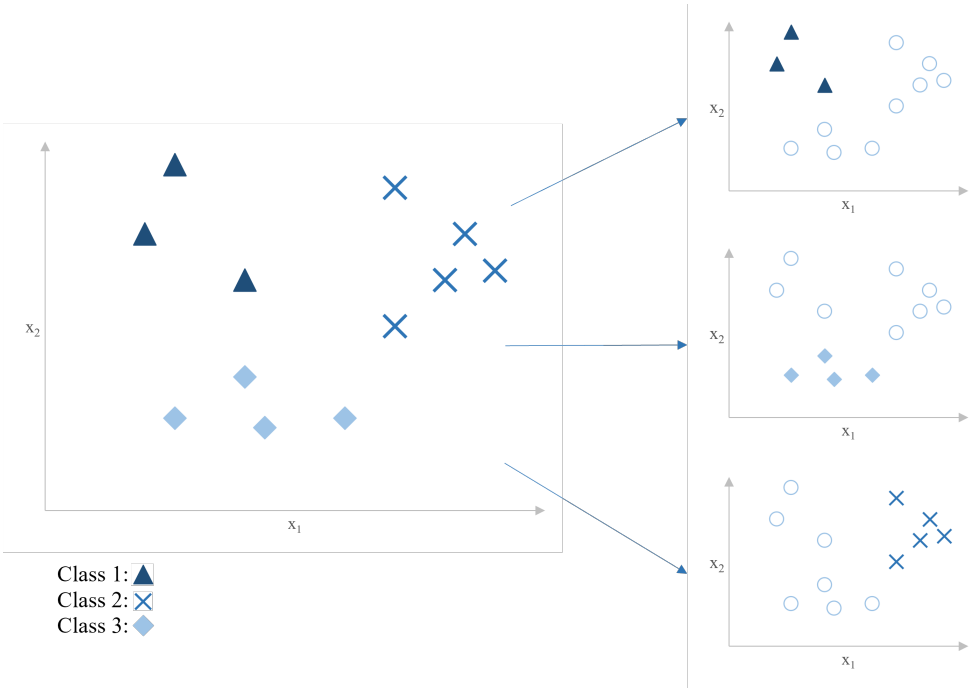


Figure 3: One-Vs-All

Since $y = \{0, 1, \ldots, n\}$, we divide our problem into $n + 1$ binary classification problems. In each binary problem we predict the probability that y is a member of one of our classes:

$$h_w(\underline{\underline{X}}) = \begin{pmatrix} h_w^{(0)}(\underline{\underline{X}}) = P(y = 0 | \underline{\underline{X}}, \underline{w}) \\ h_w^{(1)}(\underline{\underline{X}}) = P(y = 1 | \underline{\underline{X}}, \underline{w}) \\ \vdots \\ h_w^{(n)}(\underline{\underline{X}}) = P(n = 0 | \underline{\underline{X}}, \underline{w}) \end{pmatrix} \tag{12}$$

We find our prediction as:

$$\hat{y} = \max_i h_w^{(i)}(\underline{\underline{X}}) \tag{13}$$

We are choosing one class and then lumping all the others into a single second class. We do this repeatedly, applying binary logistic regression to each case, and then use the hypothesis that returned the highest value as our prediction.

3 Implementation

For our implementation of logistic regression we will firstly implement the algorithm from scratch. I like to use **Object Oriented Programming** when coding these algorithms from scratch. If you are not familiar with the concepts of object oriented programming i will recommend you to read this article before trying to understand the code implementation. Afterwards we will see how logistic regression can be implemented using the Python machine learning library *Scikitlearn*.

3.1 Implementation - From Scratch

For implementing linear regression without fancy machine learning packages we are going to use the following libraries:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Let us import the dataset. We are looking at a binary classification problem.

```
data = pd.read_csv('ex2data1.txt', header = None)
X = np.array(data.iloc[:, [0,1]])
y = np.array(data.iloc[:, -1])
print('dim of X:', X.shape)
print('dim of y:', y.shape)
```

We are now ready to implement our Logistic Regression class:

```
class LogisticRegression:
    def __init__(self, alpha = 0.01, iterations = 100000):
        self.alpha = alpha
        self.iterations = iterations
```

Within our **LogisticRegression** class we will now create functions in order to perform logistic regression on a binary dataset:

```
    def add_intercept(self, X):
        ones = np.ones((X.shape[0], 1))
        return np.concatenate((ones, X), axis = 1)

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def cost(self, hw, y):
        m = len(y)
        return 1/m*(-np.dot(y.T,np.log(hw))-np.dot((1-y).T,np.log(1-hw)))

    def fit(self, X, y):
        X = self.add_intercept(X)

        #initialize weights
        self.w = np.zeros(X.shape[1])
```

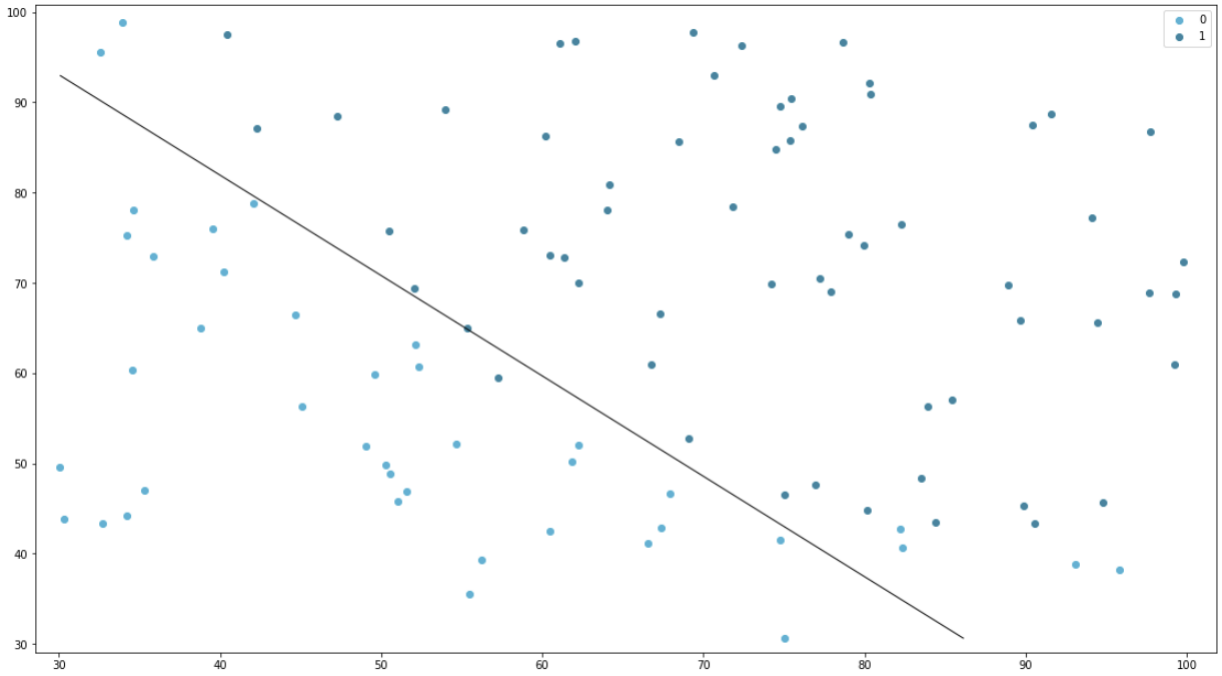
```
        #gradient descent
        cost_iterations = []
        for i in range(self.iterations):
            z = np.dot(X, self.w)
            hw = self.sigmoid(z)
            m = len(y)
            gradient = 1/m*np.dot(X.T,(hw-y))
            self.w -= self.alpha*gradient
            cost_iterations.append(self.cost(hw,y))

        self.final_cost = self.cost(hw, y)

    def predict_prob(self, X):
        X = self.add_intercept(X)
        return self.sigmoid(np.dot(X, self.w))

    def predict(self, X):
        return self.predict_prob(X).round()

    def plot_scatter(self, X, y):
        plt.figure(figsize=(19.20, 10.80))
        plt.scatter(X[y == 0][:, 0], X[y == 0][:, 1], color='#06b2d6', marker = 'o', label='0')
        plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color='#0085A1', marker = 'o', label='1')
        plt.legend()
        x1_min, x1_max = X[:,0].min(), X[:,0].max(),
        x2_min, x2_max = X[:,1].min(), X[:,1].max(),
        xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max), np.linspace(x2_min, x2_max))
        grid = np.c_[xx1.ravel(), xx2.ravel()]
        probs = self.predict_prob(grid).reshape(xx1.shape)
        plt.contour(xx1, xx2, probs, [0.5], linewidths=1, colors='black')
        plt.xlim([x1_min-x1_min*0.05,x1_max+x1_max*0.02])
        plt.ylim([x2_min-x2_min*0.05,x2_max+x2_max*0.02])
```

This is all the code we need in order to implement Logistic Regression from scratch. Let us try to test it for our dataset:

```
LR = LogisticRegression(alpha = 0.0016)
LR.fit(X, y)
print('min cost:', LR.final_cost)
```

min cost: 0.3181404335967254

```
LR.plot_scatter(X, y)
```




```
LR.w
array([-6.87627765,  0.06105823,  0.05445471])

LR.final_cost
0.3181404335967254
```

3.2 mplementation - Scikit-Learn

We will import the following libraries:

```
from sklearn.linear_model import LogisticRegression
```

As usual Scikit-learn implementation is straightforward:

```
model = LogisticRegression()
model.fit(X, y)

print(model.intercept_)
print(model.coef_)

[-25.05219314, 0.20535491 0.2005838 ]
```

What we can see is that we actually get completely different weights for our Scikit-learn implementation. This model is much better than the one we implemented from scratch. This is because Sckit-Learn doesn't use gradient descent as its optimization algorithm. As i mentioned earlier there exist much better optimization algorithms than gradient descent. The reason of why our own implementation doesn't find as great a model is most likely because gradient descent ends up converting at a local minimum on our cost function. We discussed the problem of local minima in the linear regression paper.