Logistic Regression — Simple Python Implementation

Follow

· 6 min read

> *Obs: I always wanted to post something on Medium however my urge for procrastination has been always stronger than me. So, one day I woke up, watched some rocky balboa movies, hit the gym and decided that I'd change my procrastination behavior while listening to Eye of Tiger by Survivors because everything done while listening to Eye of Tiger looks awesome. That pretty much sums up the history behind this post.*

First of all, I believe that in order to make sure that you understood something then you have to be able to explain it to someone else using your own words. That's what I am going to try to do here. After studying logistic regression I decided to make this post so I can be sure that I understood the magic behind the curtains.

Therefore, I am going to solve a simple binary classification probably without using any kind of ML framework.

> ***First of fall, what is Logistic Regression? It is a function that can be used to solve classification problems. Differently from its sister, Linear Regression, the Logistic one fits well when the target variable is categorical.***

For a better understanding, let's use some data. Below, I generated a data frame randomly. It contains three columns: *X1*, *X2*, and *Y*. We are going to use the first two columns (*X1* and *X2*) to predict the third one (*Y*) which contains only ones and zeros.

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

x1 = np.random.rand(500)*10
x2 = np.random.rand(500)*2
df= pd.DataFrame(data={'x1': x1, 'x2': x2})
df['y'] = np.where(df['x2']*df['x1']>5, 1, 0)
df.head()
```

Out[188]:

	x1	x2	y
0	9.885780	1.747193	1
1	6.783038	0.181040	0
2	8.131378	1.945022	1
3	9.893597	1.310851	1
4	6.078718	1.574567	1

Dataframe generated randomly

Out[250]:

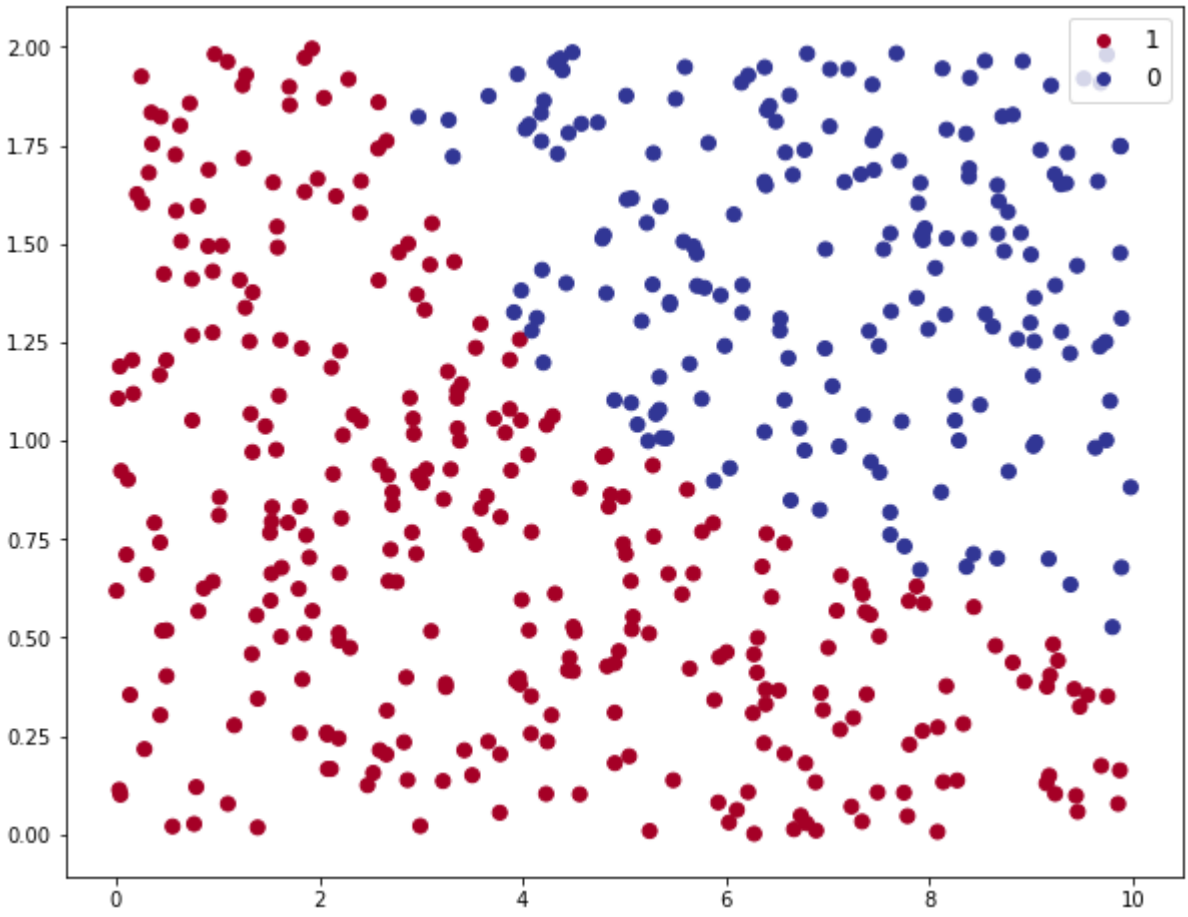
	y	Amount
0	0	311

```
1  1      189
```

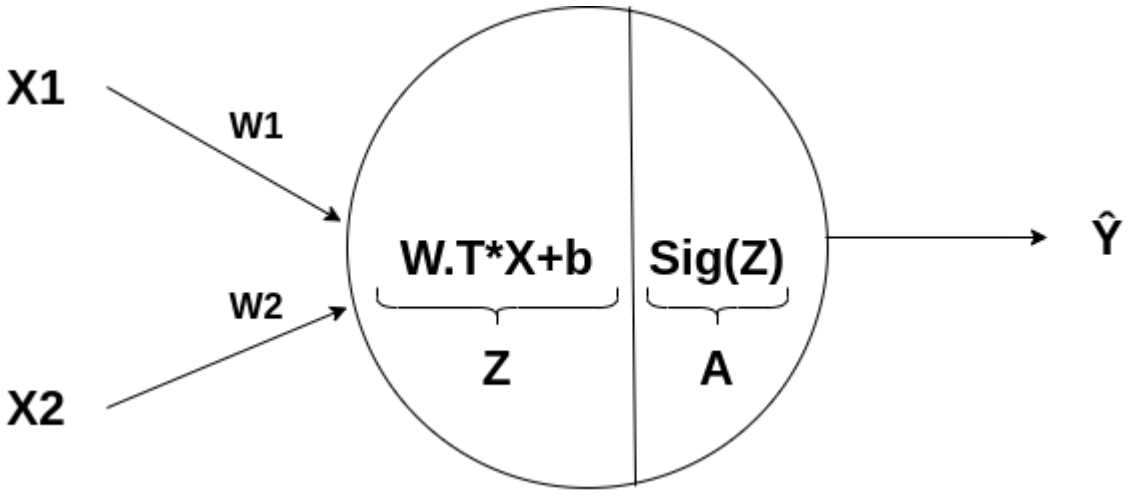
Amount of data in each category

We can plot our data and color it using our target variable. The blue and red dots represent the 1 and 0 values from our *Y* variable, respectively.

```
plt.figure(figsize=(10,8))
scatter=plt.scatter(df.x1, df.x2, s=50, c=df.y, cmap='RdYlBu')
plt.legend(handles=scatter.legend_elements()[0],labels=['1','0'],
           loc='upper right', fontsize=12
          )
plt.show()
```



Looking at the scatter plot above we see that our data is very well divided into two groups. Our goal here is to build a binary logistic regression that learns to make such division. This model will return the probability of a specific data point to be categorized as 1 ($\mathbf{P(y=1|x)}$). The image below represents the structure of our model.

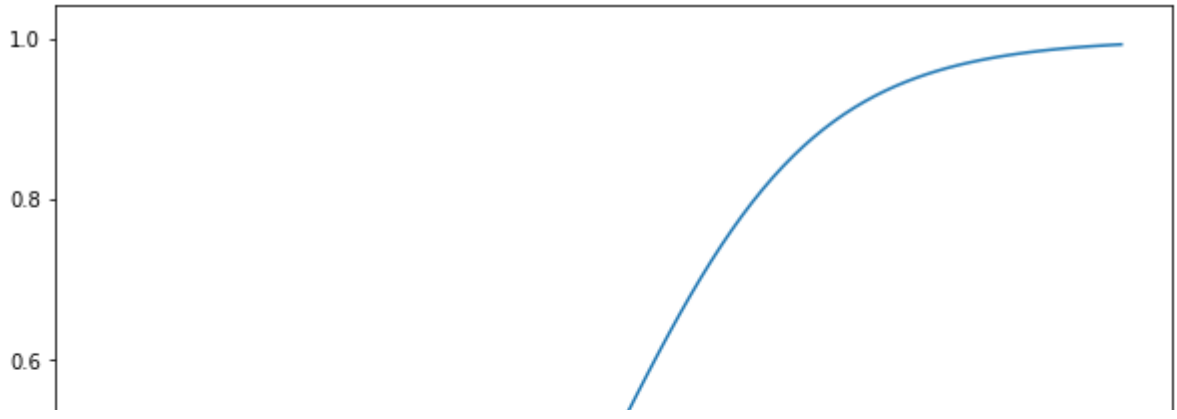


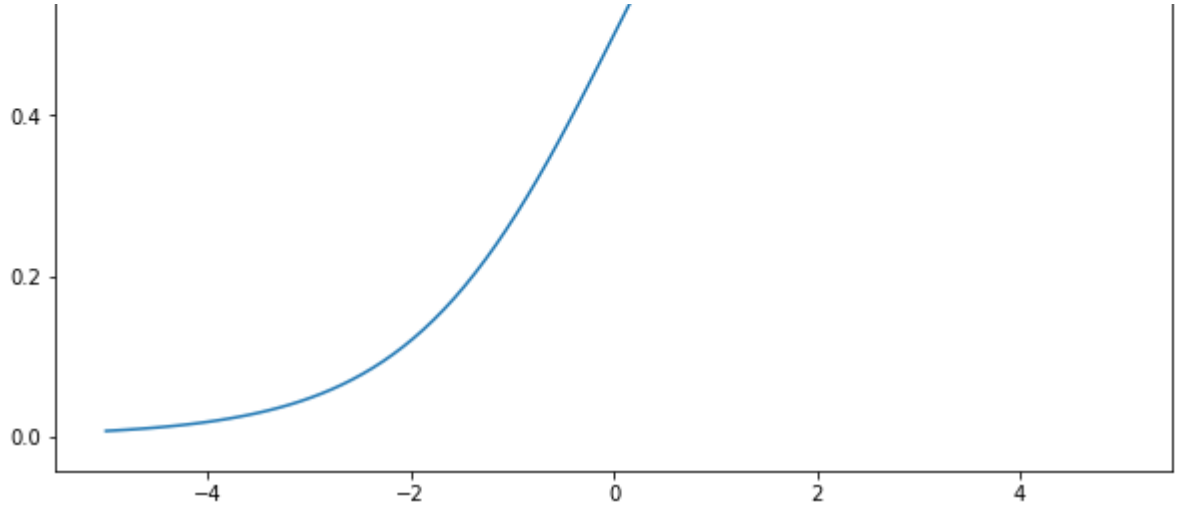
- *X1* and *X2* are the data points as vectors
- *w1* and *w2* are the weight of our model
- *Z* is our linear model.
- *Sig*(*Z*) is the sigmoid function.
- $\hat{Y}$ is the output of our model

Why use the Sigmoid function?

> *Because we cannot use Z directly as it is defined {-Inf, +Inf}. Therefore, we need to map the Z to [0,1] so we can perform our classification. The Sigmoid function does that. Below, we see its formula.*

$$Sig(Z) = \frac{1}{1 + e^{-Z}}$$





Now, we need to define our cost function. This function is going to be used to check how well our model is performing. Because we are facing a binary classification problem we can use the **binary-cross entropy** algorithm as our cost function**.**

$$LL(y, pred) = -y \ln(pred) - (1 - y) \ln(1 - pred)$$

The loss function for one instance

$$J = \sum_{i=1}^{m} LL(y^i, pred^i)$$

Cost Function

The cost function is simply the mean of the loss function applied to all the instances. The idea behind using a cost function is finding the values of *W* and *b* by minimizing it. The way to minimizing is by applying the **Gradient Descent** algorithm which I will explain in another post. For now, we just need to know that we use the derivatives of our cost function along with a **learning rate** in order to update our *W* and *b*. The learning rate parameter is simply the step size that will update our model. If we choose a huge step, then our model might not learn. On another hand, if we choose a step too small then the learning will take too much time. Below, we see the final formulas:

$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)})$$

$$\frac{\partial J}{\partial} = \frac{1}{}$$

$$\frac{\partial}{\partial w} = \frac{1}{m}X(A-Y)^T$$

The update will be computed inside a predetermined number of iterations. In each iteration, we do:

```
Repeat{
    - make prediction
    - computes the cost of our prediction
    - computes the gradients
    - updates the W and b using the previous gradient
}
```

Now, we have everything to start implementing our logistic regression. For that, we'll be using the *NumPy* package as it makes our life much easier.

```
class LogisticRegression:

    def __init__(self, df, X_cols, y_col, iterations=10000,
learning_rate=0.001):
        self.df = df
        self.X_cols = X_cols
        self.y_col = y_col
        self.iterations = iterations
        self.learning_rate = learning_rate

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

    def train(self):
        # We initialize our W and b as zeros
        w = np.zeros(len(self.X_cols)).reshape(len(self.X_cols),1)
        b = 0

        X = self.df[self.X_cols].to_numpy().T
        y = self.df[self.y_col].to_numpy().T
        m = self.df.shape[0] #Number of samples

        loss = [] #Keeping track of the cost function values

        for i in range(self.iterations):
            #Computes our predictions
            z = np.dot(w.T, X)+b
            pred = self.sigmoid(z)

            #Computes our cost function
```

```
            cost = (-1/m)*np.sum(np.dot(y,np.log(pred).T) + np.dot(1-
y,np.log(1-pred).T ))
            loss.append(cost)

            #Computes the gradient
            dw = (1/m)*np.dot(X,(pred-y).T)
            db = (1/m)*np.sum(pred-y, axis=1)

            #Updates the W and b
            w = w - self.learning_rate*dw
            b = b - self.learning_rate*db

        return {"W":w, "b": b, "loss": loss}
```

Now, we enjoy the feeling of accomplishment after the implementation.

A representation of my accomplishment feeling

Let's build our model using the data frame we created at the beginning of this post.

```
lg = LogisticRegression(df, ['x1', 'x2'], ['y'])
model = lg.train()
```

```
In [241]: model['W']

Out[241]: array([[0.14388723],
                 [0.33349879]])

In [242]: model['b']
```

```
Out[242]: array([-1.16393234])
```

It seems that the training performed smoothly. Let's evaluate the model by computing its accuracy. To do that, all we need to do is using the *W* and *b* found in the training step and make a prediction for each data point. Just remember that our model outputs a probability which we will be using to make the classification. Below, we see a function that wraps up this idea.

```
def predict(X, W, b):
    pred = sigmoid(np.dot(W.T,X)+b)
    pred = [1 if prob >= 0.5 else 0 for prob in pred[0]]
    return pred

X = df[['x1', 'x2']].to_numpy().T
pred = predict(X, model['W'], model['b'])
print("Acuracy: ", np.array(pred == df.y).sum()/500)
```

```
In [245]: X = df[['x1', 'x2']].to_numpy().T
          pred = predict(X, model['W'], model['b'])
          print("Acuracy: ", np.array(pred == df.y).sum()/500)

          Acuracy:  0.872
```

That's not bad. We were capable of reaching an accuracy of 87%.

Conclusion

I hope this post was useful for someone out there. My goal here was to make a quick tutorial of the idea behind the logistic regression algorithm implementation. As I said before, in real life no one will implement it from zero as there is already a lot of ML packages that do this for you and using the most efficient way. However, we should be curious about knowing how things work behind the curtains if we really want to understand the world of machine learning.

Source

https://www.coursera.org/learn/neural-networks-deep-learning/home/welcome