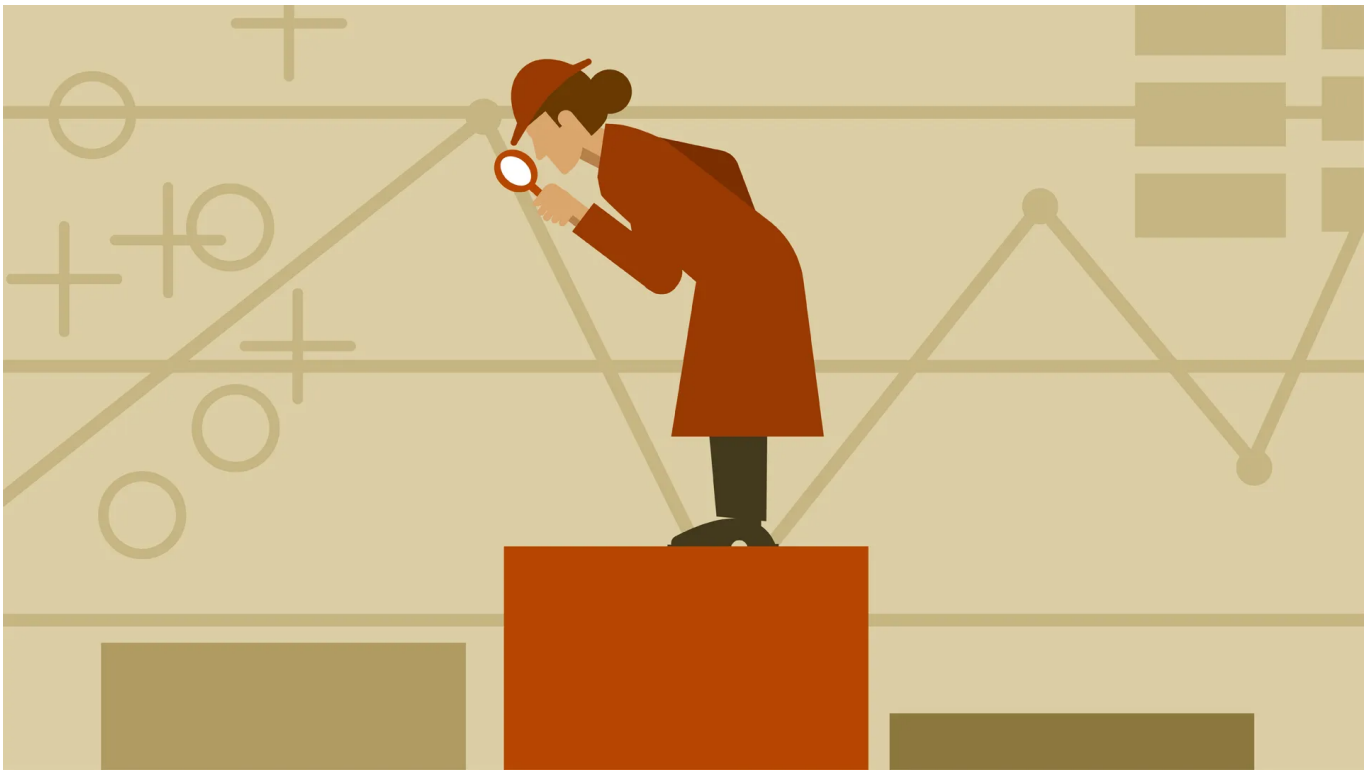
Understanding Logistic Regression

Tanya Gupta · Follow
Published in The Startup · 6 min read · Jan 24, 2021

Today we will be learning about a probabilistic classification algorithm known as logistic regression and its implementation.



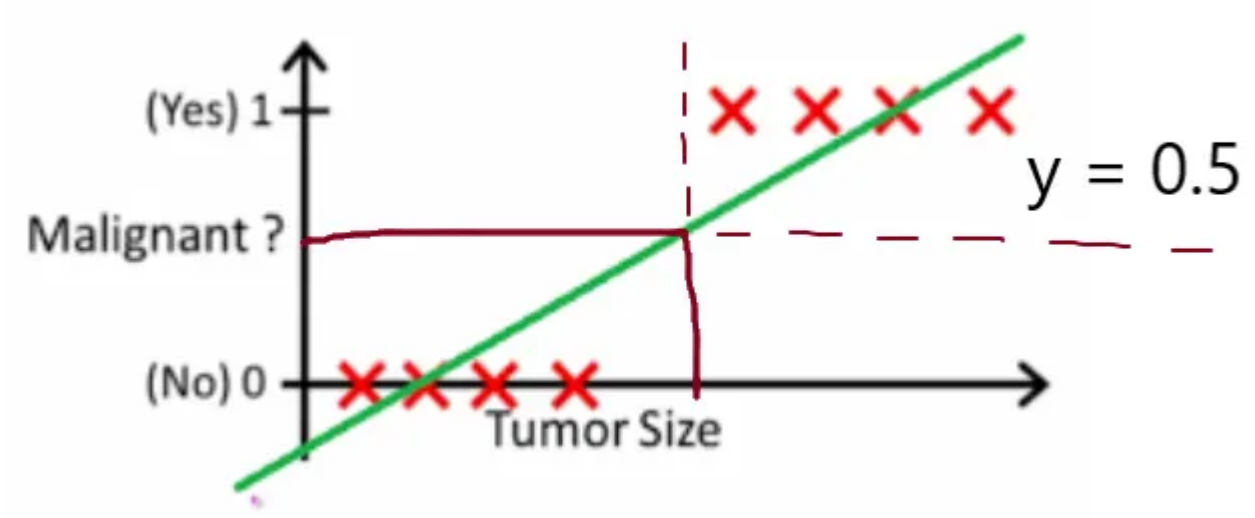
Source: dimensionless.in

Introduction

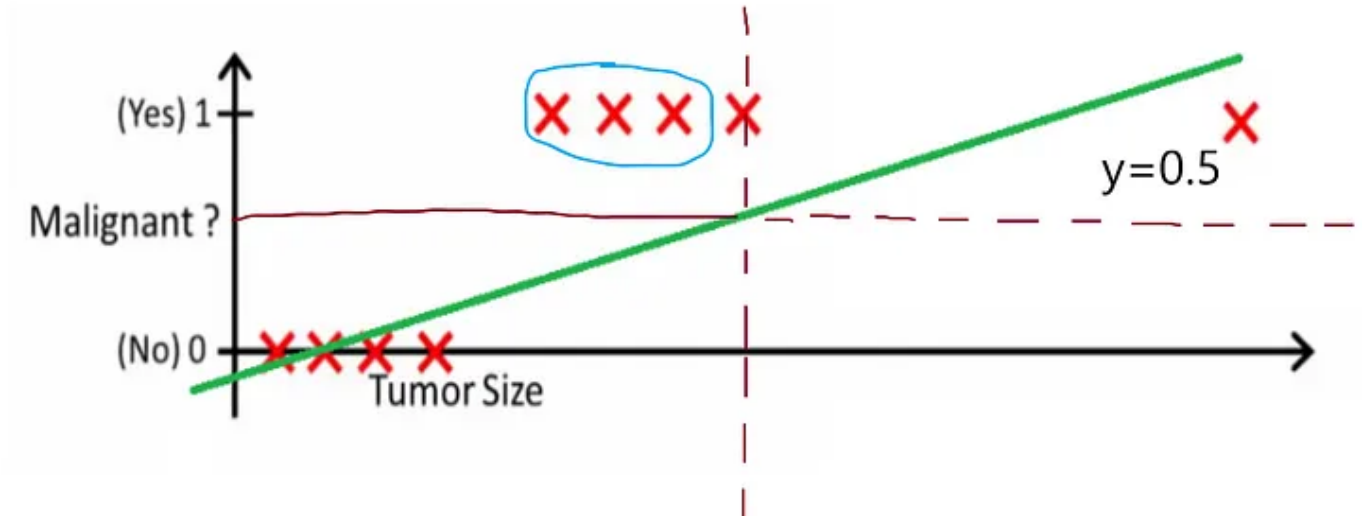
Logistic regression comes under the supervised machine learning algorithms. Its name is a bit misleading as it is used for *binary classification* however, the regression part stems from its similarity to linear regression. As we have learnt, linear regression involves around finding out a regression line which "fits" our dataset. For this we find out the parameters for our line which gives the minimum value of loss function. Logistic regression is pretty similar to that. However, it includes just one more thing, namely sigmoid or logit function which takes a value and results in a number lying between 0 and 1.

But then why use logistic and not linear regression?

This is because the regression line can be affected a lot due to outliers. Let's take an example. In the image given below, without outliers the regression line is pretty acceptable. If we take a threshold value like y=0.5 and if we get a y value ≥0.5 then it is classified in the "Malignant" group else, it would be classified as "Benign".



But with introduction of an outlier, we see that in order to decrease the loss function and "fit" with dataset better, the regression line must adjust itself. This causes some points to be misclassified as shown below.



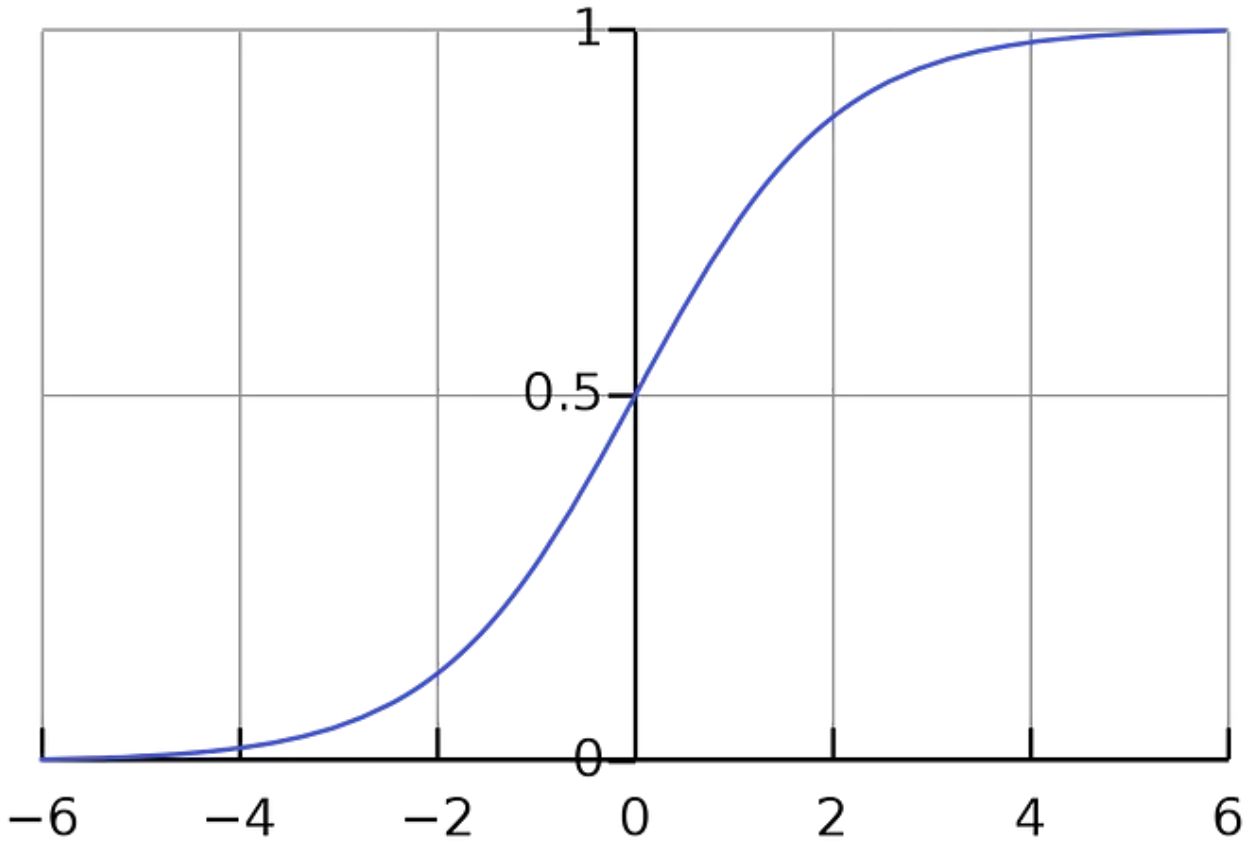
We can see that the points circled in blue are now, classified incorrectly. Another flaw here is, that linear regression might result in y value >1 or <0. In both cases we won't be able to determine exactly in which class should our data point lie in.

How logistic regression works?

Apart from the procedure followed in linear regression, we have a small addition in case of logistic regression known as sigmoid function which helps in normalizing the dot product of θ (the line parameters) and x (data points) . This is important because we have seen above that outliers might drastically change the line and it may lead to mis-classifications.

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

Sigmoid function (Source: oreilly.com)



Graphical representation of the sigmoid function (Source : en.wikipedia.org)

where, $\mathbf{z = \theta^T x}$.

But the above function σ(z) lies between 0 and 1 thus, nullifying any effect that the outliers might have.

So, the hypothesis of logistic regression (hθ(x))is σ(z) which gives the probability of our data point belonging in a certain class.

After calculating the probability from σ(z), we use a threshold value (= 0.5, generally) to ultimately predict where our data point lies. So for σ(z)≥ 0.5, the data point belongs to the class y =1. Else, it belongs to y=0.

This also implies that, σ(z)≥ 0.5 => z (= $\theta^T x$)≥ 0 and similarly, σ(z)< 0.5 if z < 0.

The Cost Function

Cost = -y*log(hθ(x))-(1-y)*log(1-hθ(x))

The above function gives a convex shape which makes it easier for us to use gradient descent to find the convergence (point where cost function is minimum). Notice if y=1, then the second term vanishes while for y=0 the first term vanishes.

For every right classification cost = 0 while for wrong ones it penalizes with cost tends to ∞. This happens as for y = 1 and hθ(x) is very close to 0 then, log(0)->∞. Similarly, for y = 0 and hθ(x) very close to 1 then, log(1-hθ(x))≈log(0)->∞.

Applying gradient descent

- Partial differentiating the cost function w.r.t θ will result in,

∂ (cost)/∂θ = [hθ(x)-y]*x

Let, $\sigma(z) = \frac{1}{1+e^{-\theta x}}$ $\rightsquigarrow h_\theta(x)$ then

PAGE NO. DATE / /

$$\frac{\partial}{\partial\theta}(\text{cost}) = \frac{\partial}{\partial\theta}\left[-y\log(h_\theta(x)) - (1-y)\log(1-h_\theta(x))\right]$$

$$\Rightarrow \frac{-y}{h_\theta(x)} \cdot \frac{\partial h_\theta(x)}{\partial\theta} - \frac{(1-y)}{(1-h_\theta)} \frac{\partial(1-h_\theta(x))}{\partial\theta}$$

$$\Rightarrow \frac{-y}{h_\theta(x)} + \frac{(1-y)}{1-h_\theta(x)} \cdot \frac{\partial(h_\theta(x))}{\partial\theta}$$

$$\Rightarrow \frac{(1-h_\theta(x))(-y) + (1-y)h_\theta(x)}{h_\theta(x)\cdot(1-h_\theta(x))} \cdot \frac{\partial h_\theta(x)}{\partial\theta}$$

$$\Rightarrow \boxed{\left[\frac{h_\theta(x)-y}{h_\theta(x)\cdot(1-h_\theta(x))}\right] \cdot \frac{\partial h_\theta(x)}{\partial\theta}} \quad \text{— (1)}$$

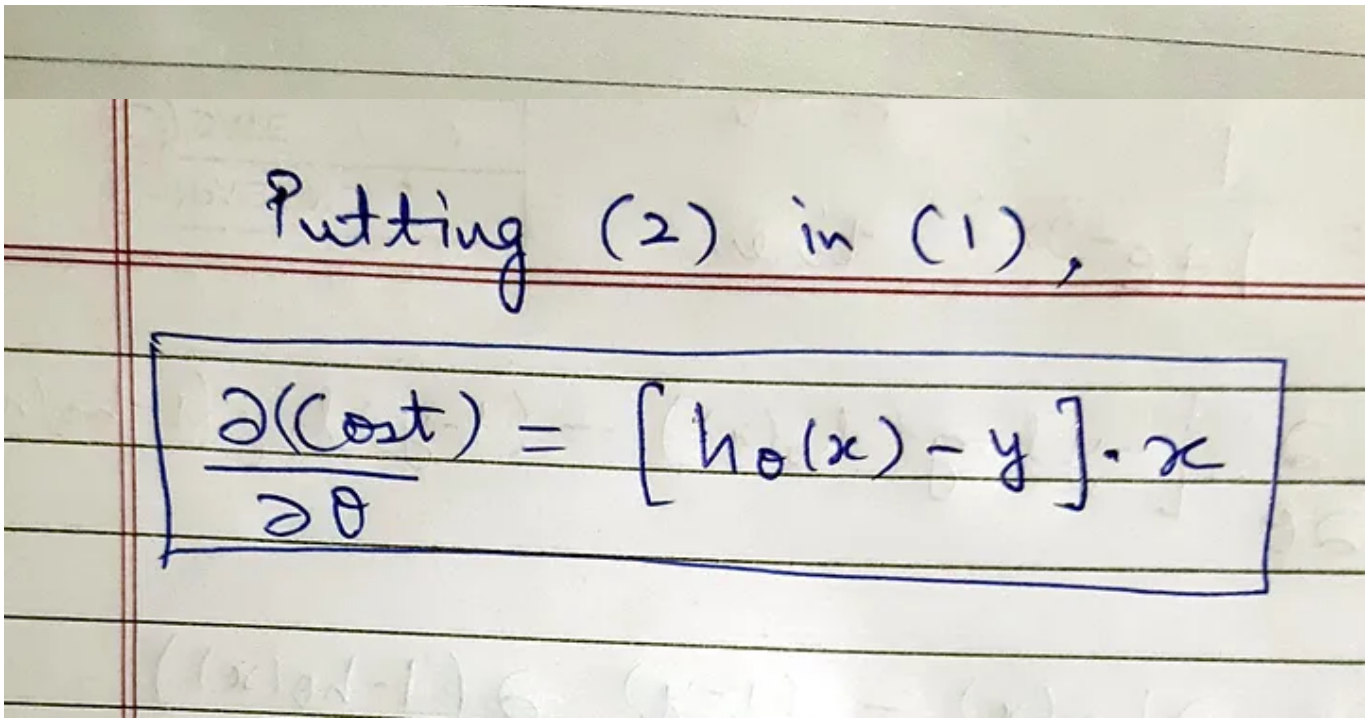
Finding $\frac{\partial h_\theta(x)}{\partial\theta}$,

$$\frac{\partial}{\partial\theta}\left[\frac{1}{1+e^{-\theta x}}\right] = -\left(\frac{1}{1+e^{-\theta x}}\right)^2 \cdot \frac{\partial}{\partial\theta}(1+e^{-\theta x})$$

$$= -\left(\frac{1}{1+e^{-\theta x}}\right)^2 \cdot (-x) \cdot e^{-\theta x}$$

$$\frac{\partial h_\theta(x)}{\partial\theta} = x \cdot \frac{e^{-\theta x}}{1+e^{-\theta x}} \cdot \frac{1}{1+e^{-\theta x}}$$

$$\boxed{\frac{\partial h_\theta(x)}{\partial\theta} = x\cdot(1-h_\theta(x))\cdot h_\theta(x)} \quad \text{— (2)}$$



derivation for the above expression for the curious.

- Update θ for all the features from $1 \le j \le n$ (α is the learning rate) until we achieve convergence:

Source: stats.stackexchange.com

Implementation on breast cancer dataset

You can find the dataset here.

- Import numpy, matplotlib.pyplot , pandas libraries.
- Convert the dataframe to numpy array. I did this so that I can shuffle up the rows of the dataset by using np.random.shuffle(dataset).
- I split my dataset in 75: 25 ratio for train and test set. Only after splitting should we normalize our data. This is because, if we find the minimum and maximum values of the entire dataset and use these to normalize the dataset then, we might leak some of the test data to our train data when we split it. Also, remember to use the same minimum and maximum value of training data to normalize the testing data.

```
for i in range(x_train.shape[1]):
    mean = np.mean(x_train[:,i])
    std =  np.std(x_train[:,i])
    x_train[:,i] = (x_train[:,i]-mean)/std
    x_test[:,i] = (x_test[:,i]-mean)/std
```

normalization of dataset

- I concatenate a column of ones (as the first column) to our training and testing set for the intercept value in the line equation $y = \theta_1 + \theta_2*x+...+\theta_n*x$.

```
ones = np.ones((x_train.shape[0],1))
x_train = np.concatenate((ones,x_train),axis=1)

ones = np.ones((x_test.shape[0],1))
x_test = np.concatenate((ones,x_test),axis=1)
```

- The gradient descent function is given below

```
def gradient_descent(X,y,alpha,epoch):
    r = X.shape[0]
    c = X.shape[1]
    theta = np.ones((c,1))
    min_cost=None
    min_theta=[]
    Cost_list=[]
    for i in range(epoch):
        h = hypothesis(theta,X)
        grad = np.dot(X.T,(h-y))
        grad/=r
        theta = theta - alpha*grad
        cost = FindCost(h,y)
        #print(cost)
        Cost_list.append(cost)
        if min_cost is None or min_cost>cost:
            min_cost=cost
            min_theta=list(theta)
    return theta,Cost_list
```

alpha and epoch are the learning rate and number of iterations respectively.

r and c store the number of rows and columns for the dataset X respectively.

theta (or parameter vector) is initialized with ones and it has shape (c,1).

I have initialized min_cost with an empty list and min_theta with None. These variables store the minimum cost possible during the iterations and min_theta stores the parameters corresponding to the minimum cost.

At each iteration, we calculate the hypothesis using the functions below:

```
def hypothesis(b,X):
    z= np.dot(X,b)
    #print(z)
    return sigmoid(z)
```

```
def sigmoid(z):
    return 1/(1+np.exp(-1*z))
```

- gradient or partial derivative is found by finding the dot product between the transpose of dataset(X) and (h-y) where, h is hypothesis and y is the response variable (for our dataset, it is “diagnosis”). The dot product is then divided by the number of rows.
- Then we update each theta by subtracting alpha*gradient.
- I find the value of cost function at each iteration for comparison and to find the theta having minimum cost using the function below:

```
def FindCost(h,y):
    r = y.shape[0]
    cost = np.mean(y*np.log(h)+(1-y)*np.log(1-h))
    #print(cost)
    return cost*-1
```

To predict for the values in our test set:

First find the hypothesis using the theta found using training dataset and the test dataset. This will give you the probability for each row in our testing data. Then for

each row, check whether it is greater than or equal to our threshold value (i.e., 0.5). If it is then the predicted value is 1 else, it is 0.

For every match between our predicted and actual y label, increment the correct variable and then find the percentage by dividing the value stored in the correct variable with the number of rows in our test set.

```
def calAccuracy(theta,X,y):
    h = hypothesis(theta,X)
    correct=0
    for i in range(y.shape[0]):
        if h[i]>=0.5:
            if y[i]==1: correct+=1
            print("predicted: ",1,end='\t\t')
        elif h[i]<0.5:
            if y[i]==0: correct+=1
            print("predicted: ",0,end='\t\t')
        print("actual: ",y[i])
    return correct*100/y.shape[0]
```

So, that wraps up the blog. Hope you have learnt something out of this.