Building a Logistic Regression in Python

Animesh Agarwal
Oct 17, 2018 · 8 min read

Suppose you are given the scores of two exams for various applicants and the objective is to classify the applicants into two categories based on their scores i.e, into Class-1 if the applicant can be admitted to the university or into Class-0 if the candidate can't be given admission. Can this problem be solved using Linear Regression? Let's check.

Note: I suggest you read Linear Regression before going ahead with this blog.

Table of Contents

What is Logistic Regression?

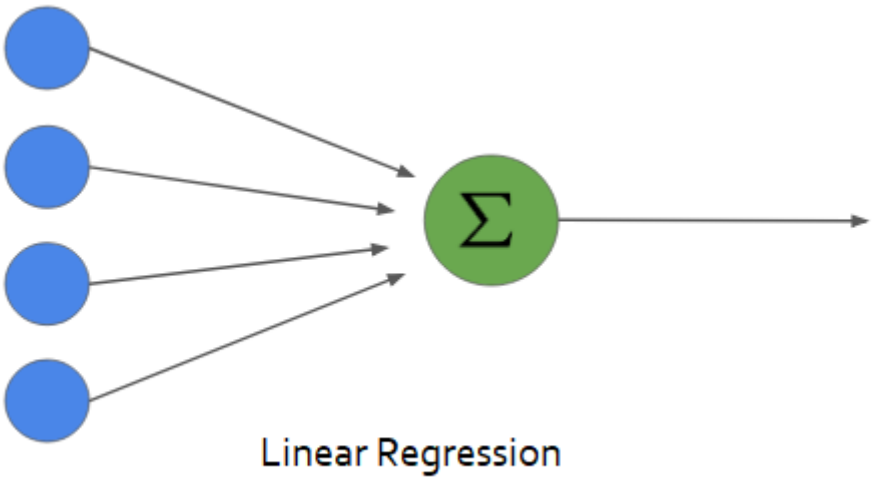
If you recall Linear Regression, it is used to determine the value of a continuous dependent variable. Logistic Regression is generally used for classification purposes. Unlike Linear Regression, the dependent variable can take a limited number of values only i.e, the dependent variable is **categorical**. When the number of possible outcomes is only two it is called **Binary Logistic Regression**.

Let's look at how logistic regression can be used for classification tasks.

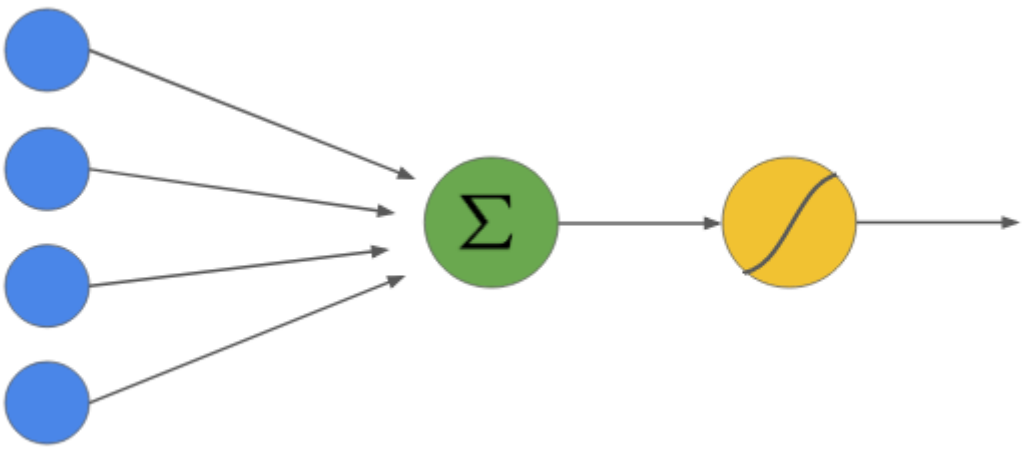
In Linear Regression, the output is the weighted sum of inputs. Logistic Regression is a generalized Linear Regression in the sense that we don't output the weighted sum of inputs directly, but we pass it through a function that can map any real value between 0 and 1.

> ***If we take the weighted sum of inputs as the output as we do in Linear Regression, the value can be more than 1 but we want a value between 0 and 1. That's why Linear Regression can't be used for classification tasks.***

We can see from the below figure that the output of the linear regression is passed through an **activation** function that can map any real value between 0 and 1.

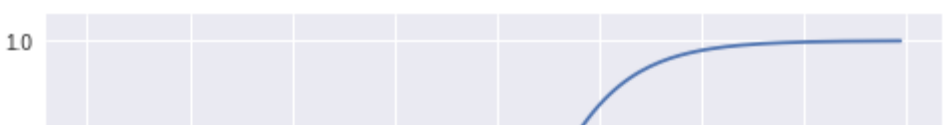


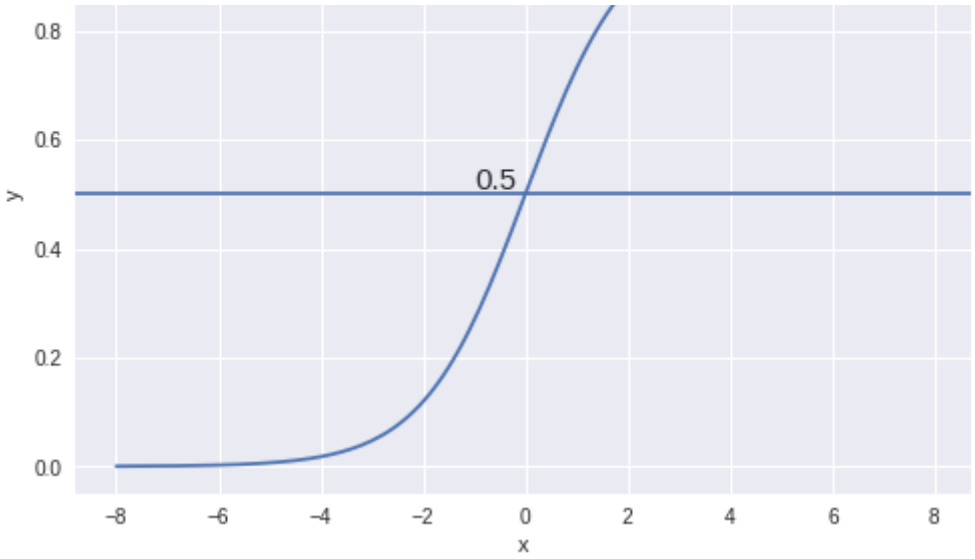
Linear Regression



Logistic Regression

The activation function that is used is known as the **sigmoid** function. The plot of the sigmoid function looks like





sigmoid function

We can see that the value of the sigmoid function always lies between 0 and 1. The value is exactly 0.5 at X=0. We can use 0.5 as the probability threshold to determine the classes. If the probability is greater than 0.5, we classify it as **Class-1(Y=1)** or else as **Class-0(Y=0)**.

Before we build our model let's look at the assumptions made by Logistic Regression

- The dependent variable must be categorical
- The independent variables(features) must be independent (to avoid multicollinearity).

Dataset

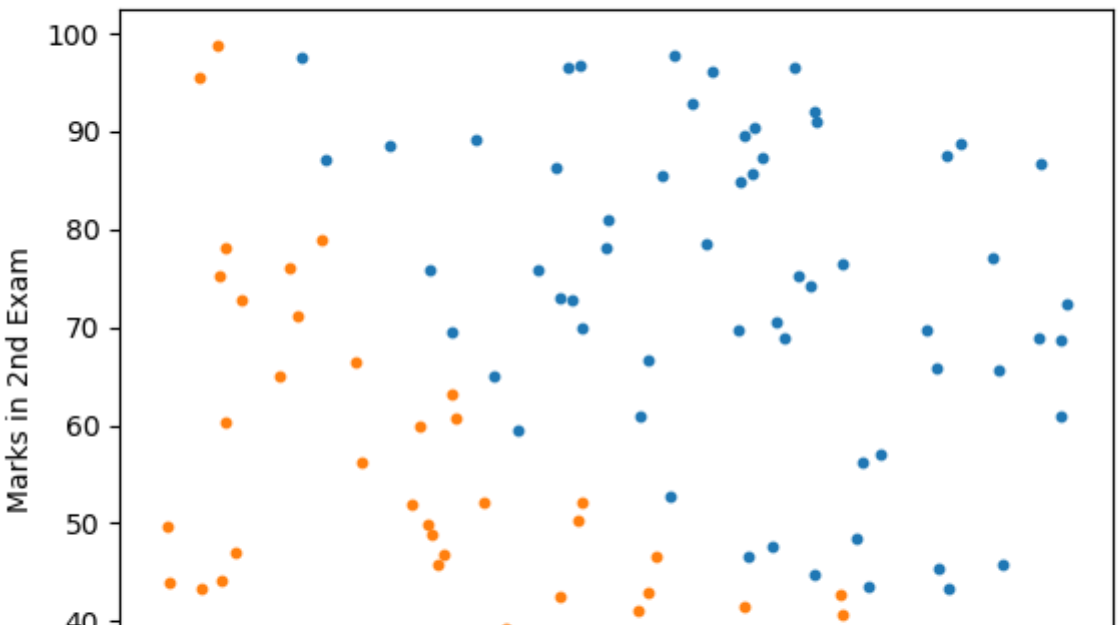
The data used in this blog has been taken from Andrew Ng's Machine Learning course on Coursera. The data can be downloaded from here. The data consists of marks of two exams for 100 applicants. The target value takes on binary values 1,0. 1 means the applicant was admitted to the university whereas 0 means the applicant didn't get an admission. The objective is to build a classifier that can predict whether an application will be admitted to the university or not.

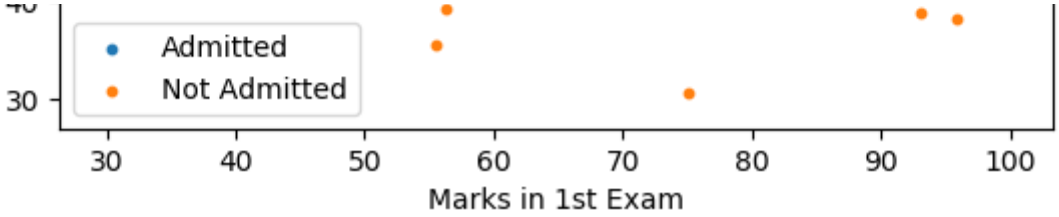
Let's load the data into `pandas` Dataframe using the `read_csv` function. We will also split the data into `admitted` and `non-admitted` to visualize the data.

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

```
5
6
7  def load_data(path, header):
8      marks_df = pd.read_csv(path, header=header)
9      return marks_df
10
11
12 if __name__ == "__main__":
13     # load the data from the file
14     data = load_data("data/marks.txt", None)
15
16     # X = feature values, all the columns except the last column
17     X = data.iloc[:, :-1]
18
19     # y = target values, last column of the data frame
20     y = data.iloc[:, -1]
21
22     # filter out the applicants that got admitted
23     admitted = data.loc[y == 1]
24
25     # filter out the applicants that din't get admission
26     not_admitted = data.loc[y == 0]
27
28     # plots
29     plt.scatter(admitted.iloc[:, 0], admitted.iloc[:, 1], s=10, label='Admitted')
30     plt.scatter(not_admitted.iloc[:, 0], not_admitted.iloc[:, 1], s=10, label='Not Admi
31     plt.legend()
32     plt.show()
```

dataset.py hosted with ❤ by GitHub view raw





Now that we have a clear understanding of the problem and the data, let's go ahead and build our model.

Hypothesis and Cost Function

Till now we have understood how Logistic Regression can be used to classify the instances into different classes. In this section, we will define the hypothesis and the cost function.

A Linear Regression model can be represented by the equation.

$$h(x) = \theta^T x$$

We then apply the sigmoid function to the output of the linear regression

$$h(x) = \sigma(\theta^T x)$$

where the sigmoid function is represented by,

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

The hypothesis for logistic regression then becomes,

$$h(x) = \frac{1}{1 + e^{-\theta^T x}}$$

$$h(x) = \begin{cases} > 0.5, & \text{if } \theta^T x > 0 \\ < 0.5, & \text{if } \theta^T x < 0 \end{cases}$$

If the weighted sum of inputs is greater than zero, the predicted class is 1 and vice-versa. So the decision boundary separating both the classes can be found by setting the

weighted sum of inputs to 0.

Cost Function

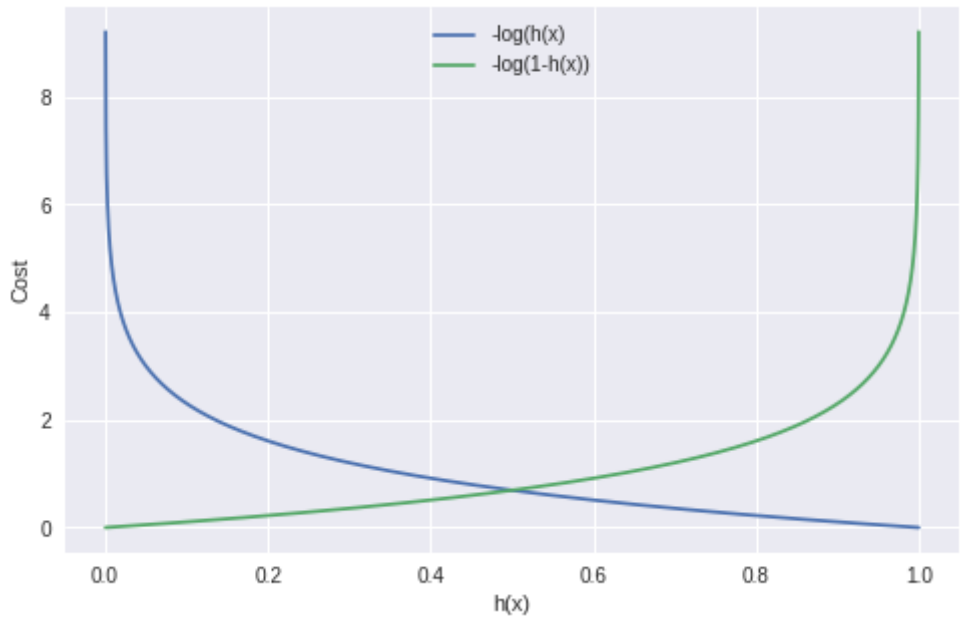
Like Linear Regression, we will define a cost function for our model and the objective will be to minimize the cost.

The cost function for a single training example can be given by:

$$cost = \begin{cases} -log(h(x), & \text{if } y = 1 \\ -log(1 - h(x)), & \text{if } y = 0 \end{cases}$$

Cost function intuition

If the actual class is 1 and the model predicts 0, we should highly penalize it and vice-versa. As you can see from the below picture, for the plot `-log(h(x))` as h(x) approaches 1, the cost is 0 and as h(x) nears 0, the cost is infinity(that is we penalize the model heavily). Similarly for the plot `-log(1-h(x))` when the actual value is 0 and the model predicts 0, the cost is 0 and the cost becomes infinity as h(x) approaches 1.



We can combine both of the equations using:

$$cost(h(x), y) = -ylog(h(x)) - (1 - y)log(1 - h(x))$$

The cost for all the training examples denoted by *J(θ)* can be computed by taking the average over the cost of all the training samples

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^i log(h(x^i)) + (1-y^i) log(1-h(x^i))]$$

where ***m*** is the number of training samples.

We will use gradient descent to minimize the cost function. The gradient w.r.t any parameter can be given by

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h(x^i) - y^i)\ x_j^i$$

The equation is similar to what we achieved in Linear Regression, only h(x) is different in both the cases.

Training the model

Now we have everything in place we need to build our model. Let's implement this in code.

Let's first prepare the data for our model.

```
X = np.c_[np.ones((X.shape[0], 1)), X]
y = y[:, np.newaxis]
theta = np.zeros((X.shape[1], 1))
```

We will define some functions that will be used to compute the cost.

```
def sigmoid(x):
    # Activation function used to map any real value between 0 and 1
    return 1 / (1 + np.exp(-x))

def net_input(theta, x):
    # Computes the weighted sum of inputs
    return np.dot(x, theta)

def probability(theta, x):
    # Returns the probability after passing through sigmoid
    return sigmoid(net_input(theta, x))
```

Next, we define the `cost` and the `gradient` function.

```
def cost_function(self, theta, x, y):
    # Computes the cost function for all the training samples
    m = x.shape[0]
    total_cost = -(1 / m) * np.sum(
        y * np.log(probability(theta, x)) + (1 - y) * np.log(
            1 - probability(theta, x)))
    return total_cost

def gradient(self, theta, x, y):
    # Computes the gradient of the cost function at the point theta
    m = x.shape[0]
    return (1 / m) * np.dot(x.T, sigmoid(net_input(theta,   x)) - y)
```

Let's also define the `fit` function which will be used to find the model parameters that minimizes the cost function. In this blog, we coded the gradient descent approach to compute the model parameters. Here, we will use `fmin_tnc` function from the `scipy` library. It can be used to compute the minimum for any function. It takes arguments as

- func: the function to minimize
- x0: initial values for the parameters that we want to find
- fprime: gradient for the function defined by 'func'
- args: arguments that needs to be passed to the functions.

```
def fit(self, x, y, theta):
    opt_weights = fmin_tnc(func=cost_function, x0=theta,
                  fprime=gradient,args=(x, y.flatten()))
    return opt_weights[0]

parameters = fit(X, y, theta)
```

The model parameters are `[-25.16131856 0.20623159 0.20147149]`

To see how good our model performed, we will plot the decision boundary.

Plotting the decision boundary

As there are two features in our dataset, the linear equation can be represented by,

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

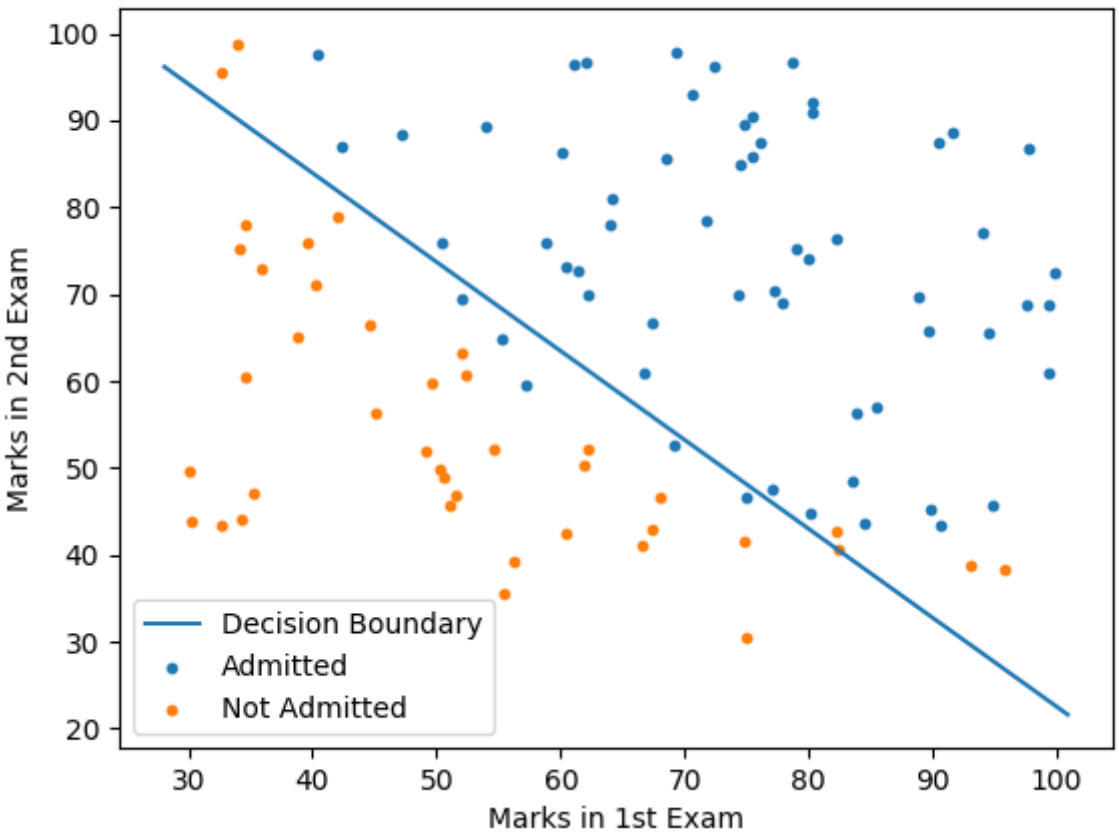
As discussed earlier, the decision boundary can be found by setting the weighted sum of inputs to 0. Equating h(x) to 0 gives us,

$$x_2 = -\frac{\theta_0 + \theta_1 x_1}{\theta_2}$$

We will plot our decision boundary on top of the plot we used for visualizing our dataset.

```
x_values = [np.min(X[:, 1] - 5), np.max(X[:, 2] + 5)]
y_values = - (parameters[0] + np.dot(parameters[1], x_values)) /
parameters[2]

plt.plot(x_values, y_values, label='Decision Boundary')
plt.xlabel('Marks in 1st Exam')
plt.ylabel('Marks in 2nd Exam')
plt.legend()
plt.show()
```



Looks that our model has done a decent job in predicting the classes. But how accurate is it? Let's find out.

Accuracy of the model

```
def predict(self, x):
    theta = parameters[:, np.newaxis]
    return probability(theta, x)

def accuracy(self, x, actual_classes, probab_threshold=0.5):
    predicted_classes = (predict(x) >=
                         probab_threshold).astype(int)
    predicted_classes = predicted_classes.flatten()
    accuracy = np.mean(predicted_classes == actual_classes)
    return accuracy * 100

accuracy(X, y.flatten())
```

The accuracy of the model is 89%.

Let's implement our classifier using scikit-learn and compare it with the model we built from scratch.

scikit-learn implementation

```
from sklearn.linear_model import LogisticRegressionfrom
sklearn.metrics import accuracy_score

model = LogisticRegression()
model.fit(X, y)
predicted_classes = model.predict(X)
accuracy = accuracy_score(y.flatten(),predicted_classes)
parameters = model.coef_
```

The model parameters are `[[-2.85831439, 0.05214733, 0.04531467]]` and the accuracy is 91%.

Why are the model parameters significantly different from the model we implemented from scratch? If you look at the documentation of sk-learn's Logistic Regression implementation, it takes regularization into account. Basically, regularization is used to prevent the model from overfitting the data. I won't be getting

into the details of regularization in this blog. But for now, that's it. Thanks for Reading !!

The complete code used in this blog can be found in this GitHub repo.

Please drop me a message if you are stuck anywhere or if you have any feedback.

What Next?

In the next blog, we will use the concepts learned in this blog to build a classifier on the Adult data from the UCI Machine Learning Repository. The dataset contains close to 49K samples and includes categorical, numerical and missing values. This will be an interesting dataset to explore.

Do watch this space for more.

Machine Learning Logistic Regression Sigmoid