Logistic regression

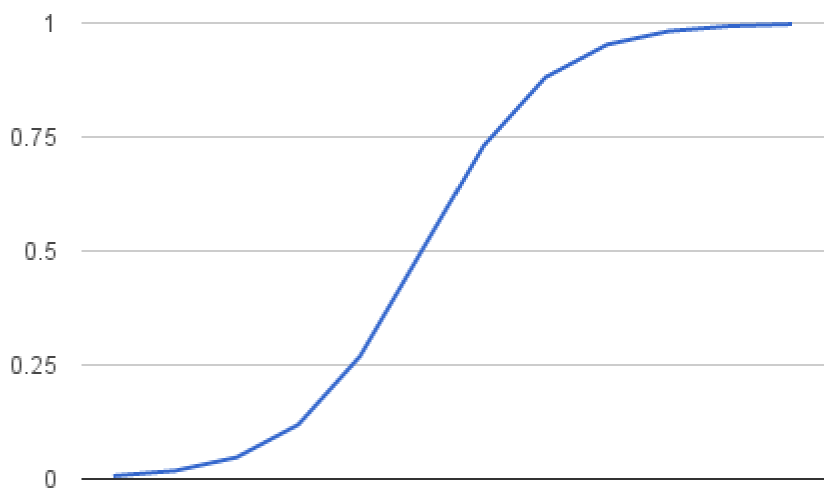
Logistic regression is a statistical model that in its basic form uses a logistic function to model a binary dependent variable, although many more complex extensions exist. Mathematically, a binary logistic model has a dependent variable with two possible values, such as pass/fail which is represented by an indicator variable, where the two values are labelled "0" and "1". Logistic Regression is used when the dependent variable(target) is categorical.

For example,

To predict whether an email is spam (1) or (0)

Whether the tumour is malignant (1) or not (0)

Logistic Function



Logistic regression is named for the function used at the core of the method, the logistic function.

The logistic function, also called the sigmoid function was developed by statisticians to describe properties of population growth in ecology, rising quickly and maxing out at the carrying capacity of the environment. It’s an S-shaped curve that can take any real-valued number and map it into a value between 0 and 1, but never exactly at those limits.

1 / (1 + e^-value)

Where e is the base of the natural logarithms and value is the actual numerical value that you want to transform. Below is a plot of the numbers between -5 and 5 transformed into the range 0 and 1 using the logistic function.

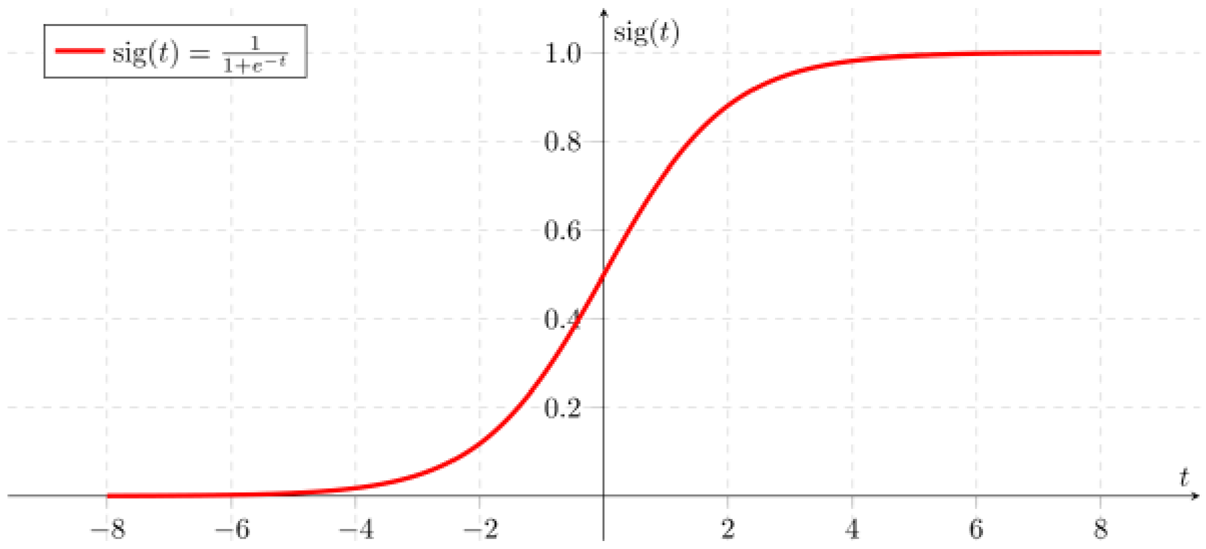
Model

Output = 0 or 1

Hypothesis => Z = WX + B

hΘ(x) = sigmoid (Z)

Sigmoid Function



If ‘Z’ goes to infinity, Y(predicted) will become 1 and if ‘Z’ goes to negative infinity, Y(predicted) will become 0.

Types of Logistic Regression

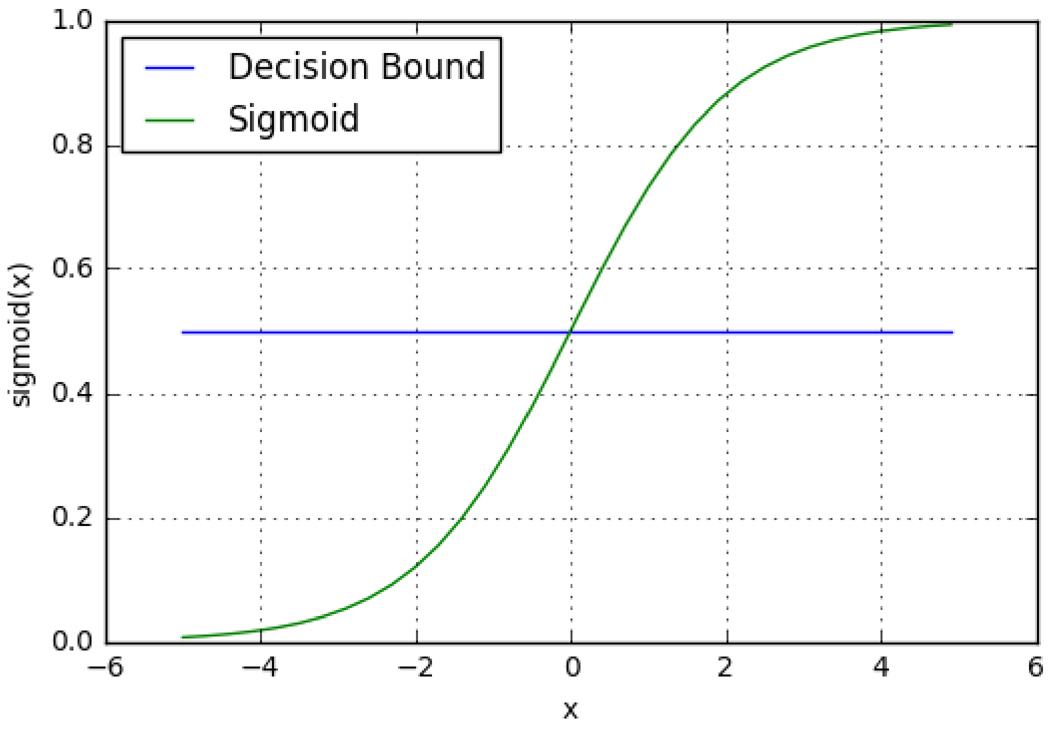
1. Binary Logistic Regression
   The categorical response has only two 2 possible outcomes. Example: Spam or Not
2. Multinomial Logistic Regression
   Three or more categories without ordering. Example: Predicting which food is preferred more (Veg, Non-Veg, Vegan)
3. Ordinal Logistic Regression
   Three or more categories with ordering. Example: Movie rating from 1 to 5

Decision boundary

Our current prediction function returns a probability score between 0 and 1. In order to map this to a discrete class (true/false, cat/dog), we select a threshold value or tipping point above which we will classify values into class 1 and below which we classify values into class 2.

- $p \geq 0.5$, class=1
- $p < 0.5$, class=0

For example, if our threshold was 0.5 and our prediction function returned 0.7, we would classify this observation as positive. If our prediction was 0.2 we would classify the observation as negative. For logistic regression with multiple classes we could select the class with the highest predicted probability.

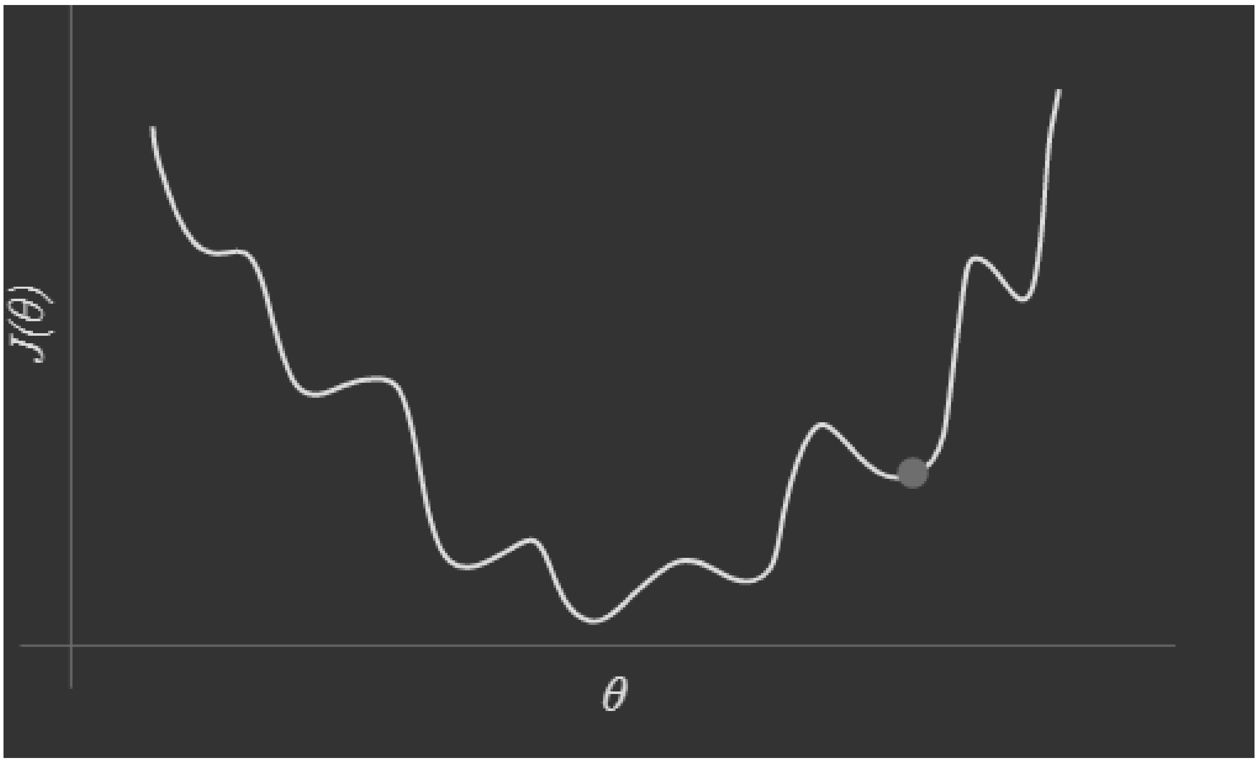


Making predictions

Using our knowledge of sigmoid functions and decision boundaries, we can now write a prediction function. A prediction function in logistic regression returns the probability of our observation being positive, True, or “Yes”. We call this class 1 and its notation is P(class=1). As the probability gets closer to 1, our model is more confident that the observation is in class 1.

Cost function

You might remember the original cost function J(θ) used in linear regression. I can tell you right now that it's not going to work here with logistic regression. If you try to use the linear regression's cost function to generate J(θ) in a logistic regression problem, you would end up with a non-convex function: a weirdly-shaped graph with no easy to find minimum global point, as seen in the picture below.



This strange outcome is due to the fact that in logistic regression we have the sigmoid function around, which is non-linear (i.e. not a line). With the J(θ) depicted in figure 1. the gradient descent algorithm might get stuck in a local minimum point. That's why we still need a neat convex function as we did for linear regression: a bowl-shaped function that eases the gradient descent function's work to converge to the optimal minimum point.

A better cost function for logistic regression:

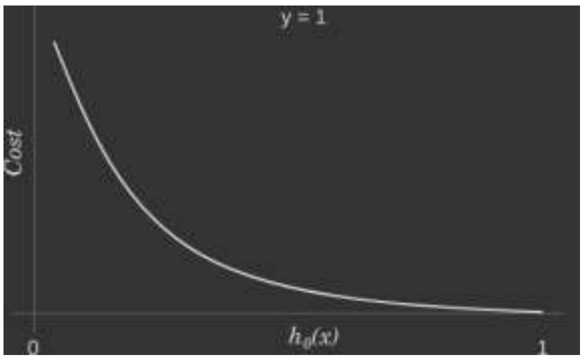
For logistic regression, the Cost function is defined as:

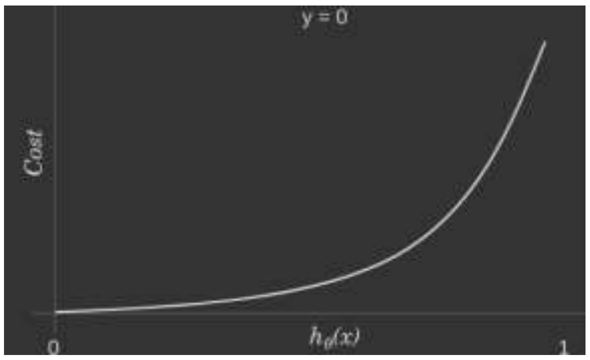
$\mathrm{Cost}(h\theta(x),y)=\{-\log(h\theta(x))\}$ if $y = 1$ and $\{-\log(1-h\theta(x))\}$ if $y = 0$

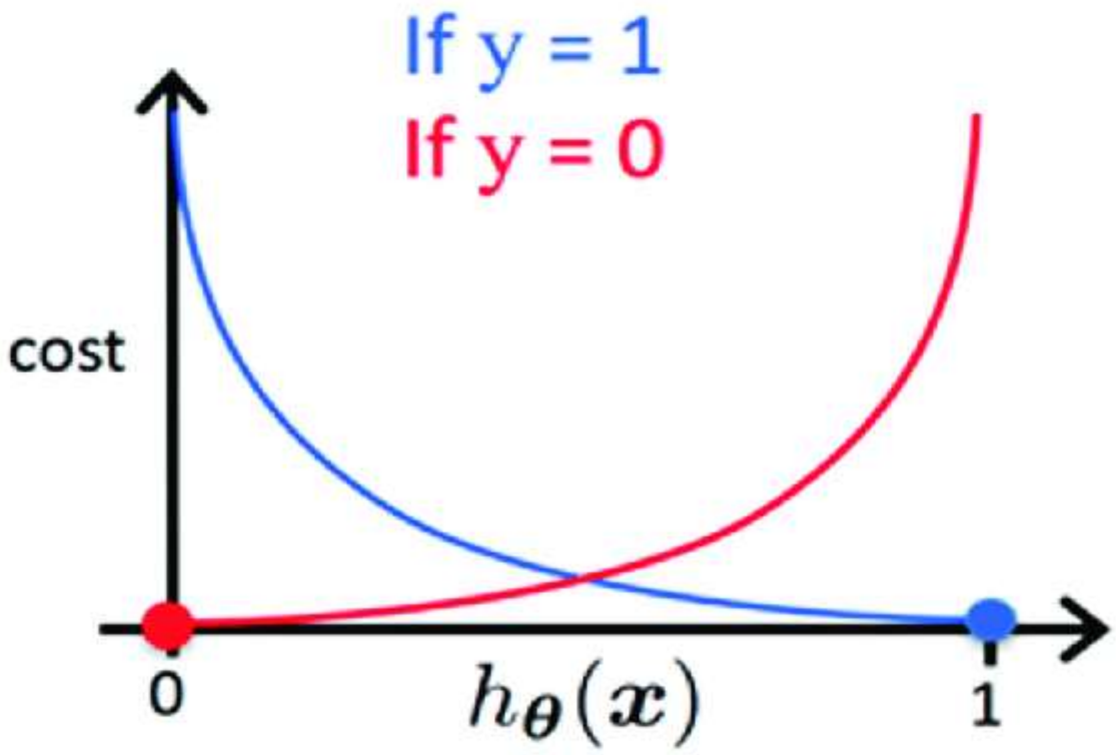
The i indexes have been removed for clarity. In words this is the cost the algorithm pays if it predicts a value hθ(x) while the actual cost label turns out to be y. By using this function we will grant the convexity to the function the gradient descent algorithm has to process, as discussed above. There is also a mathematical proof for that, which is outside the scope of this introductory course.

In case y=1, the output (i.e. the cost to pay) approaches to 0 as hθ(x) approaches to 1. Conversely, the cost to pay grows to infinity as hθ(x) approaches to 0. You can clearly see it in the plot 2. below, left side. This is a desirable property: we want a bigger penalty as the algorithm predicts something far away from the actual value. If the label is y=1 but the algorithm predicts hθ(x)=0, the outcome is completely wrong.

Conversely, the same intuition applies when y=0, depicted in the plot 2. below, right side. Bigger penalties when the label is y=0 but the algorithm predicts hθ(x)=1.







Gradient descent

We have the hypothesis function and the cost function. It's now time to find the best values for θs parameters in the cost function, or in other words to minimize the cost function by running the gradient descent algorithm. The procedure is identical to what we did for linear regression.

More formally, we want to minimize the cost function, J(θ)

Which will output a set of parameters θ, the best ones (i.e. with less error). Once done, we will be ready to make predictions on new input examples with their features x, by using the new θs in the hypothesis function:

hθ(x)=1/(1+e^(θ⊤x))

Where hθ(x) is the output, the prediction, or yet the probability that y=1.

The way we are going to minimize the cost function is by using the gradient descent. The good news is that the procedure is 99% identical to what we did for linear regression.

To minimize the cost function we have to run the gradient descent function on each parameter:

```
repeat until convergence {
    $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$
}
```

or

```
repeat until convergence {
    $\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$
}
```