[Cost function in logistic regression gives NaN as a result](http://stackoverflow.com/questions/35419882/cost-function-in-logistic-regression-gives-nan-as-a-result)

Logistic regression using batch gradient descent. There are two classes into which the input samples are to be classified. The classes are 1 and 0. While training the data, I am using the following sigmoid function:

t = 1 ./ (1 + exp(-z));

where

z = x\*theta

And I am using the following cost function to calculate cost, to determine when to stop training.

htheta = sigmoid(x\*theta);

cost = sum(-y .\* log(htheta) - (1-y) .\* log(1-htheta));

I am getting the cost at each step to be NaN as the values of htheta are either 1 or zero in most cases. What should I do to determine the cost value at each iteration?

This is the gradient descent code for logistic regression:

function [theta,cost\_history] = batchGD(x,y,theta,alpha)

cost\_history = zeros(1000,1);

for iter=1:1000

htheta = sigmoid(x\*theta);

new\_theta = zeros(size(theta,1),1);

for feature=1:size(theta,1)

new\_theta(feature) = theta(feature) - alpha \* sum((htheta - y) .\*x(:,feature))

end

theta = new\_theta;

cost\_history(iter) = computeCost(x,y,theta);

end

end

**The data is not normalized**

This is because when you apply the sigmoid / logit function to your hypothesis, the output probabilities are almost all approximately 0s or all 1s and with your cost function, log(1 - 1) or log(0) will produce -Inf. The accumulation of all of these individual terms in your cost function will eventually lead to NaN.

Specifically, if y = 0 for a training example and if the output of your hypothesis is log(x) where x is a very small number which is close to 0, examining the first part of the cost function would give us 0\*log(x) and will in fact produce NaN. Similarly, if y = 1 for a training example and if the output of your hypothesis is also log(x) where x is a very small number, this again would give us 0\*log(x) and will produce NaN. Simply put, the output of your hypothesis is either very close to 0 or very close to 1.

This is most likely due to the fact that the dynamic range of each feature is widely different and so a part of your hypothesis, specifically the weighted sum of x\*theta for each training example you have will give you either very large negative or positive values, and if you apply the sigmoid function to these values, you'll get very close to 0 or 1.

One way to combat this is to **normalize** the data in your matrix before performing training using gradient descent. A typical approach is to normalize with zero-mean and unit variance. Given an input feature x\_k where k = 1, 2, ... n where you have n features, the new normalized feature x\_k^{new} can be found by:

m\_k is the mean of the feature k and s\_k is the standard deviation of the feature k. This is also known as **standardizing** data. You can read up on more details about this on another answer I gave here: [How does this code for standardizing data work?](http://stackoverflow.com/questions/28689807/how-does-this-code-for-standardizing-data-work/28690441#28690441)

Because you are using the linear algebra approach to gradient descent, I'm assuming you have prepended your data matrix with a column of all ones. Knowing this, we can normalize your data like so:

mX = mean(x,1);

mX(1) = 0;

sX = std(x,[],1);

sX(1) = 1;

xnew = bsxfun(@rdivide, bsxfun(@minus, x, mX), sX);

The mean and standard deviations of each feature are stored in mX and sX respectively. You can learn how this code works by reading the post I linked to you above. I won't repeat that stuff here because that isn't the scope of this post. To ensure proper normalization, I've made the mean and standard deviation of the first column to be 0 and 1 respectively. xnew contains the new normalized data matrix. Use xnew with your gradient descent algorithm instead. Now once you find the parameters, to perform any predictions you **must** normalize any new test instances with the mean and standard deviation from the **training set**. Because the parameters learned are with respect to the statistics of the training set, you must also apply the same transformations to any test data you want to submit to the prediction model.

Assuming you have new data points stored in a matrix called xx, you would do normalize then perform the predictions:

xxnew = bsxfun(@rdivide, bsxfun(@minus, xx, mX), sX);

Now that you have this, you can perform your predictions:

pred = sigmoid(xxnew\*theta) >= 0.5;

You can change the threshold of 0.5 to be whatever you believe is best that determines whether examples belong in the positive or negative class.

**The learning rate is too large**

As you mentioned in the comments, once you normalize the data the costs appear to be finite but then suddenly go to NaN after a few iterations. Normalization can only get you so far. If your learning rate or alpha is too large, each iteration will overshoot in the direction towards the minimum and would thus make the cost at each iteration oscillate or even diverge which is what is appearing to be happening. In your case, the cost is diverging or increasing at each iteration to the point where it is so large that it can't be represented using floating point precision.

As such, one other option is to decrease your learning rate alpha until you see that the cost function is decreasing at each iteration. A popular method to determine what the best learning rate would be is to perform gradient descent on a range of logarithmically spaced values of alpha and seeing what the final cost function value is and choosing the learning rate that resulted in the smallest cost.

Using the two facts above together should allow gradient descent to converge quite nicely, assuming that the cost function is convex. In this case for logistic regression, it most certainly is.

### Your formula for the cost function is wrong because of a subtle 0 and Inf, numerical math issue

As @rayryeng pointed out, 0 \* log(0) produces a NaN (0 \* Inf isn't kosher). This is actually a huge problem: if your algorithm believes it can predict a value perfectly, it incorrectly assigns a cost of NaN. Instead of writing your cost function as:

cost = sum(-y .\* log(htheta) - (1-y) .\* log(1-htheta));

It should instead be something like:

y\_logical = y == 1;

cost = sum(-log(htheta(y\_logical))) + sum( - log(1 - htheta(~y\_logical)));

The idea is if y\_i is 1, we add -log(htheta\_i) to the cost, but if y\_i is 0, we add -log(1 - htheta\_i) to the cost. This is mathematically equivalent to -y\_i \* log(htheta\_i) - (1 - y\_i) \* log(1- htheta\_i) but without running into numerical problems that essentially stem from htheta\_i being equal to 0 or 1 within the limits of double precision floating point.

The reason why is because this explicitly avoids the multiplication of 0\*log(0) if the situation were to arise. By indexing into your hypothesis and selecting out those values that belong to each class respectively, this avoids having any NaN values that may result in the sum computation. Your true reason why you are getting NaN is because your learning rate is too large.

You may get a dimension mismatch because indexing using y\_logical and ~y\_logical may produce different sized vectors. I would suggest splitting up the sum into two separate operations... those where y == 1 and those where y == 0 then adding the two results together.

Something like: cost = sum(-log(htheta(y == 1))) + sum(-log(1 - htheta(y == 0))); should do nicely.