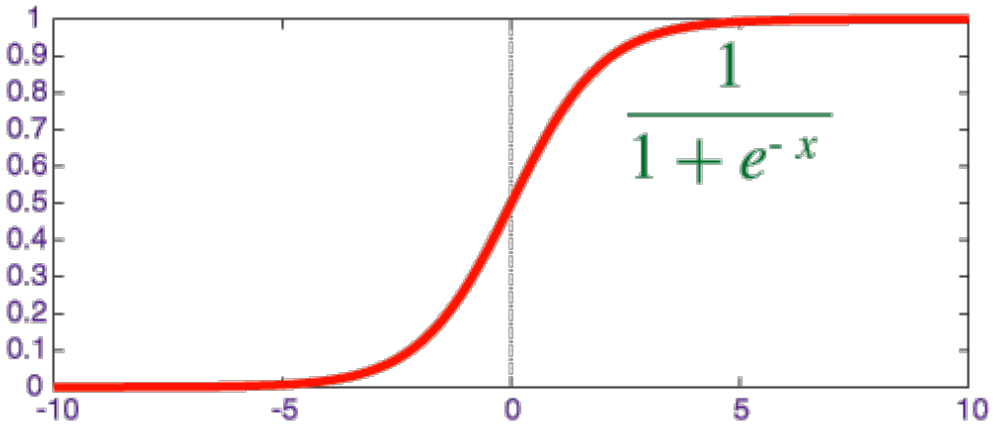
Logistic Regression and Handwritten Digits

What is the efficiency of logistic regression in predicting the correct hand-written integer? I'll use data supplied by Andrew Ng (Stanford).

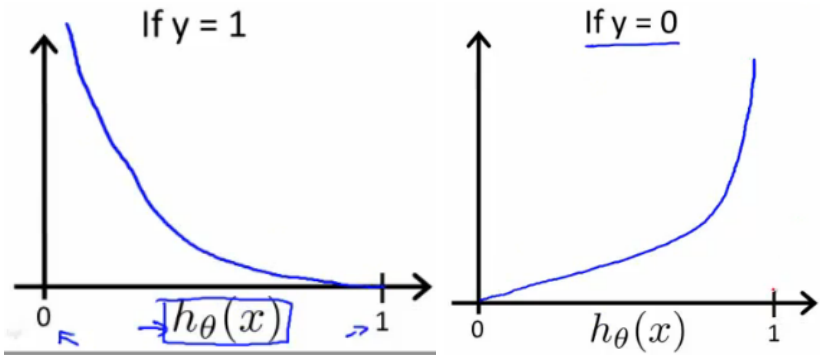
Logistic regression has a Boolean outcome (e.g. benign/cancerous young/old true/false). Put formally, its outcome o is $o \in \{0,1\}$. So, we no longer need a line of best fit; instead, we make a decision boundary. On one side of the boundary, the result is false (0), vice versa. It seems that the sigmoid function below g(x)=1/(1+e^-z) is the best choice to model logistic regression whereas in linear regression our equivalent was a straight line in the x-y plane. I'd reckon we could also use a modified arctan function; I'm still looking into this on online forums.



Courtesy: quora.com

What is z? My first thought would be the training examples put into matrix X | X= $\mathbb{R}^{m*(n+1)}$ But I later realized that we need to penalize incorrect behaviour and make sure the decision boundary isn't wrong, so we multiply it with the theta matrix θ^T. So, in MATLAB, z=θ'X.

The linear regression cost function J(θ) was basically some parabola we want to find the minimum point for, where the slope of any point on the curve is what we're looking for as the slope of the line of best fit. I realize we can't keep this going for logistic regression because $o \in \{0,1\}$ so o is binary. Naturally, we need to build a piece-wise function. From my reading, it seems that the log functions come in handy here:



This gives:

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

The intuition is that it penalizes the wrong outcome to infinity (NaN for octave). I don't know why this'd be the case but maybe we could challenge this underlying assumption if we don't want to penalize our wrong answers so heavily...maybe like in a lab where a super-expensive experiment is being done on successive generations of mice and you could keep on going to see if the next generation also produces the result you thought was incorrect just to confirm.

Combining the piece-wise function, we get our J(θ).

Gradient Descent

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

Expanding for h_θ(x):

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(-y^{(i)}\log(\frac{1}{1+e^{-x\theta}}) - (1-y)\log(\frac{1}{1+e^{-x\theta}}))$$

To get our gradient descent, we need to derive the above. The vectorized form of $\frac{\partial J(\theta)}{\partial \theta}$ is

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m}(sigmoid(X) - y) * (X')$$

Regularizing is common practise for ML algorithms having n>>10. As one Redditor puts it, regularizing is adding a scaled identity matrix to your maximum likelihood estimate. We use it when we have a lot of features and we don't want to overfit our data. Let our regularizing factor λ be 0.01.

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m}((sigmoid(X) - y) * (X') + \lambda\,\theta)$$

Theta is the parameter which means if you vary theta, you get a different hypothesis. I think of theta as the probability that a training set is of class k e K. So, to predict our classes,

---

```
h=sigmoid(X*theta')
[pval, p]=max(h,[],2);
```

Using MATLAB's advanced optimization techniques, we will use one-versus-all classification to train K = |{0,1,2,…9}| = 10 logistic regression classifiers. To do this, we will be using an advanced optimization function fmincg instead of fminunc (unconstrained) or gradient descent.

```
initial_theta = zeros(n + 1, 1);
```

```
options = optimset('GradObj', 'on', 'MaxIter', 50);
[theta] =fmincg (@(t)(lrCostFunction(t, X, (y == c), lambda)),initial_theta, options);
all_theta(c,:) = theta';
```

We get an efficiency of **95.0%.**