coursera

Week 3 Lecture Notes

ML:Logistic Regression

Now we are switching from regression problems to **classification problems**. Don't be confused by the name "Logistic Regression"; it is named that way for historical reasons and is actually an approach to classification problems, not regression problems.

Binary Classification

Instead of our output vector y being a continuous range of values, it will only be 0 or 1.

y∈{0,1}

Where 0 is usually taken as the "negative class" and 1 as the "positive class", but you are free to assign any representation to it.

We're only doing two classes for now, called a "Binary Classification Problem."

One method is to use linear regression and map all predictions greater than 0.5 as a 1 and all less than 0.5 as a 0. This method doesn't work well because classification is not actually a linear function.

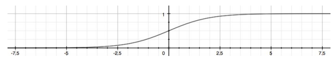
Hypothesis Representation

Our hypothesis should satisfy:

$0 \leq h_\theta(x) \leq 1$

Our new form uses the "Sigmoid Function," also called the "Logistic Function":

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



The function g(z), shown here, maps any real number to the (0, 1) interval, making it useful for transforming an arbitrary-valued function into a function better suited for classification. Try playing with interactive plot of sigmoid function : (https://www.desmos.com/calculator/bgontvxotm).

We start with our old hypothesis (linear regression), except that we want to restrict the range to 0 and 1. This is accomplished by plugging $\theta^T x$ into the Logistic Function.

h_θ will give us the **probability** that our output is 1. For example, $h_\theta(x) = 0.7$ gives us the probability of 70% that our output is 1.

$$h_\theta(x) = P(y=1|x;\theta) = 1 - P(y=0|x;\theta)$$
$$P(y=0|x;\theta) + P(y=1|x;\theta) = 1$$

Our probability that our prediction is 0 is just the complement of our probability that it is 1 (e.g. if probability that it is 1 is 70%, then the probability that it is 0 is 30%).

Decision Boundary

In order to get our discrete 0 or 1 classification, we can translate the output of the hypothesis function as follows:

$$h_\theta(x) \geq 0.5 \rightarrow y = 1$$
$$h_\theta(x) < 0.5 \rightarrow y = 0$$

The way our logistic function g behaves is that when its input is greater than or equal to zero, its output is greater than or equal to 0.5:

$$g(z) \geq 0.5$$
$$when\ z \geq 0$$

Remember.-

$$z = 0, e^0 = 1 \Rightarrow g(z) = 1/2$$
$$z \rightarrow \infty, e^{-\infty} \rightarrow 0 \Rightarrow g(z) = 1$$
$$z \rightarrow -\infty, e^{\infty} \rightarrow \infty \Rightarrow g(z) = 0$$

So if our input to g is $\theta^T X$, then that means:

$$h_\theta(x) = g(\theta^T x) \geq 0.5$$
$$when\ \theta^T x \geq 0$$

From these statements we can now say:

$$\theta^T x \geq 0 \Rightarrow y = 1$$
$$\theta^T x < 0 \Rightarrow y = 0$$

The **decision boundary** is the line that separates the area where y = 0 and where y = 1. It is created by our hypothesis function.

Example:

$$\begin{array}{l} \quad 5 \\ \theta = -1 \\ \quad 0 \end{array}$$
$$y = 1\ if\ 5 + (-1)x_1 + 0x_2 \geq 0$$
$$5 - x_1 \geq 0$$
$$-x_1 \geq -5$$
$$x_1 \leq 5$$

In this case, our decision boundary is a straight vertical line placed on the graph where $x_1 = 5$, and everything to the left of that denotes y = 1, while everything to the right denotes y = 0.

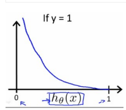
Again, the input to the sigmoid function g(z) (e.g. $\theta^T X$) doesn't need to be linear, and could be a function that describes a circle (e.g. $z = \theta_0 + \theta_1 x_1^2 + \theta_2 x_2^2$) or any shape to fit our data.

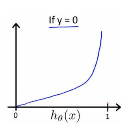
Cost Function

We cannot use the same cost function that we use for linear regression because the Logistic Function will cause the output to be wavy, causing many local optima. In other words, it will not be a convex function.

Instead, our cost function for logistic regression looks like:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \mathrm{Cost}(h_\theta(x^{(i)}), y^{(i)})$$
$$\mathrm{Cost}(h_\theta(x), y) = -\log(h_\theta(x)) \qquad \text{if y = 1}$$
$$\mathrm{Cost}(h_\theta(x), y) = -\log(1 - h_\theta(x)) \qquad \text{if y = 0}$$





The more our hypothesis is off from y, the larger the cost function output. If our hypothesis is equal to y, then our cost is 0:

$$\mathrm{Cost}(h_\theta(x), y) = 0 \text{ if } h_\theta(x) = y$$
$$\mathrm{Cost}(h_\theta(x), y) \rightarrow \infty \text{ if } y = 0 \text{ and } h_\theta(x) \rightarrow 1$$
$$\mathrm{Cost}(h_\theta(x), y) \rightarrow \infty \text{ if } y = 1 \text{ and } h_\theta(x) \rightarrow 0$$

If our correct answer 'y' is 0, then the cost function will be 0 if our hypothesis function also outputs 0. If our hypothesis approaches 1, then the cost function will approach infinity.

If our correct answer 'y' is 1, then the cost function will be 0 if our hypothesis function outputs 1. If our hypothesis approaches 0, then the cost function will approach infinity.

Note that writing the cost function in this way guarantees that J(θ) is convex for logistic regression.

Simplified Cost Function and Gradient Descent

We can compress our cost function's two conditional cases into one case:

$\mathrm{Cost}(h_\theta(x), y) = -y\ \log(h_\theta(x)) - (1-y)\log(1 - h_\theta(x))$

Notice that when y is equal to 1, then the second term $(1-y)\log(1-h_\theta(x))$ will be zero and will not affect the result. If y is equal to 0, then the first term $-y\log(h_\theta(x))$ will be zero and will not affect the result.

We can fully write out our entire cost function as follows:

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

A vectorized implementation is:

$$h = g(X\theta)$$
$$J(\theta) = \frac{1}{m} \cdot \left(-y^T\log(h) - (1-y)^T\log(1-h)\right)$$

Gradient Descent

Remember that the general form of gradient descent is:

$$Repeat\ \{$$
$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$
$$\}$$

We can work out the derivative part using calculus to get:

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

Notice that this algorithm is identical to the one we used in linear regression. We still have to simultaneously update all values in theta.

A vectorized implementation is:

$\theta := \theta - \frac{\alpha}{m} X^T (g(X\theta) - \vec{y})$

Partial derivative of J(θ)

First calculate derivative of sigmoid function (it will be useful while finding partial derivative of J(θ)):

$$\sigma(x)' = \left(\frac{1}{1+e^{-x}}\right)' = \frac{-(1+e^{-x})'}{(1+e^{-x})^2} = \frac{-1' - (e^{-x})'}{(1+e^{-x})^2} = \frac{0 - (-x)'(e^{-x})}{(1+e^{-x})^2} = \frac{-(-1)(e^{-x})}{(1+e^{-x})^2} = \frac{e^{-x}}{(1+e^{-x})^2}$$
$$= \left(\frac{1}{1+e^{-x}}\right)\left(\frac{e^{-x}}{1+e^{-x}}\right) = \sigma(x)\left(\frac{+1-1+e^{-x}}{1+e^{-x}}\right) = \sigma(x)\left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}\right) = \sigma(x)(1-\sigma(x))$$

Now we are ready to find out resulting partial derivative:

$$
\begin{aligned}
\frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \frac{-1}{m} \sum_{i=1}^{m} y^{(i)} log(h_\theta(x^{(i)})) + (1 - y^{(i)}) log(1 - h_\theta(x^{(i)})) \\
&= -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \frac{\partial}{\partial \theta_j} log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \frac{\partial}{\partial \theta_j} log(1 - h_\theta(x^{(i)})) \\
&= -\frac{1}{m} \sum_{i=1}^{m} \frac{y^{(i)} \frac{\partial}{\partial \theta_j} h_\theta(x^{(i)})}{h_\theta(x^{(i)})} + \frac{(1 - y^{(i)}) \frac{\partial}{\partial \theta_j} (1 - h_\theta(x^{(i)}))}{1 - h_\theta(x^{(i)})} \\
&= -\frac{1}{m} \sum_{i=1}^{m} \frac{y^{(i)} \frac{\partial}{\partial \theta_j} \sigma(\theta^T x^{(i)})}{h_\theta(x^{(i)})} + \frac{(1 - y^{(i)}) \frac{\partial}{\partial \theta_j} (1 - \sigma(\theta^T x^{(i)}))}{1 - h_\theta(x^{(i)})} \\
&= -\frac{1}{m} \sum_{i=1}^{m} \frac{y^{(i)} \sigma(\theta^T x^{(i)})(1 - \sigma(\theta^T x^{(i)})) \frac{\partial}{\partial \theta_j} \theta^T x^{(i)}}{h_\theta(x^{(i)})} + \frac{-(1 - y^{(i)}) \sigma(\theta^T x^{(i)})(1 - \sigma(\theta^T x^{(i)})) \frac{\partial}{\partial \theta_j} \theta^T x^{(i)}}{1 - h_\theta(x^{(i)})} \\
&= -\frac{1}{m} \sum_{i=1}^{m} \frac{y^{(i)} h_\theta(x^{(i)})(1 - h_\theta(x^{(i)})) \frac{\partial}{\partial \theta_j} \theta^T x^{(i)}}{h_\theta(x^{(i)})} - \frac{(1 - y^{(i)}) h_\theta(x^{(i)})(1 - h_\theta(x^{(i)})) \frac{\partial}{\partial \theta_j} \theta^T x^{(i)}}{1 - h_\theta(x^{(i)})} \\
&= -\frac{1}{m} \sum_{i=1}^{m} y^{(i)}(1 - h_\theta(x^{(i)})) x_j^{(i)} - (1 - y^{(i)}) h_\theta(x^{(i)}) x_j^{(i)} \\
&= -\frac{1}{m} \sum_{i=1}^{m} y^{(i)}(1 - h_\theta(x^{(i)})) - (1 - y^{(i)}) h_\theta(x^{(i)}) x_j^{(i)} \\
&= -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} - y^{(i)} h_\theta(x^{(i)}) - h_\theta(x^{(i)}) + y^{(i)} h_\theta(x^{(i)}) x_j^{(i)} \\
&= -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} - h_\theta(x^{(i)}) x_j^{(i)} \\
&= \frac{1}{m} \sum_{i=1}^{m} h_\theta(x^{(i)}) - y^{(i)} x_j^{(i)}
\end{aligned}
$$

The vectorized version;

$$\nabla J(\theta) = \frac{1}{m} \cdot X^T \cdot \left(g\left(X \cdot \theta\right) - \vec{y}\right)$$

Advanced Optimization

"Conjugate gradient", "BFGS", and "L-BFGS" are more sophisticated, faster ways to optimize θ that can be used instead of gradient descent. A. Ng suggests not to write these more sophisticated algorithms yourself (unless you are an expert in numerical computing) but use the libraries instead, as they're already tested and highly optimized. Octave provides them.

We first need to provide a function that evaluates the following two functions for a given input value θ:

$$J(\theta)$$

$$\frac{\partial}{\partial \theta_j} J(\theta)$$

We can write a single function that returns both of these:

```
function [jVal, gradient] = costFunction(theta)
  jVal = [...code to compute J(theta)...];
  gradient = [...code to compute derivative of J(theta)...];
end
```

Then we can use octave's "fminunc()" optimization algorithm along with the "optimset()" function that creates an object containing the options we want to send to "fminunc()". (Note: the value for MaxIter should be an integer, not a character string - errata in the video at 7:30)

```
options = optimset('GradObj', 'on', 'MaxIter', 100);
      initialTheta = zeros(2,1);
      [optTheta, functionVal, exitFlag] = fminunc(@costFunction, initialTheta, options);
```

We give to the function "fminunc()" our cost function, our initial vector of theta values, and the "options" object that we created beforehand.

Multiclass Classification: One-vs-all

Now we will approach the classification of data into more than two categories. Instead of y = {0,1} we will expand our definition so that y = {0,1...n}.

In this case we divide our problem into n+1 (+1 because the index starts at 0) binary classification problems; in each one, we predict the probability that 'y' is a member of one of our classes.

$$
\begin{aligned}
&y \in \{0,1\ldots n\}\\
&h_\theta^{(0)}(x) = P(y = 0 | x; \theta)\\
&h_\theta^{(1)}(x) = P(y = 1 | x; \theta)\\
&\cdots\\
&h_\theta^{(n)}(x) = P(y = n | x; \theta)\\
&\text{prediction} = \max_i (h_\theta^{(i)}(x))
\end{aligned}
$$

We are basically choosing one class and then lumping all the others into a single second class. We do this repeatedly, applying binary logistic regression to each case, and then use the hypothesis that returned the highest value as our prediction.

ML:Regularization

The Problem of Overfitting

Regularization is designed to address the problem of overfitting.

High bias or underfitting is when the form of our hypothesis function h maps poorly to the trend of the data. It is usually caused by a function that is too simple or uses too few features. eg. if we take $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$ then we are making an initial assumption that a linear model will fit the training data well and will be able to generalize but that may not be the case.

At the other extreme, overfitting or high variance is caused by a hypothesis function that fits the available data but does not generalize well to predict new data. It is usually caused by a complicated function that creates a lot of unnecessary curves and angles unrelated to the data.

This terminology is applied to both linear and logistic regression. There are two main options to address the issue of overfitting:

1) Reduce the number of features:

 a) Manually select which features to keep.

 b) Use a model selection algorithm (studied later in the course).

2) Regularization

Keep all the features, but reduce the parameters θ_j.

Regularization works well when we have a lot of slightly useful features.

Cost Function

If we have overfitting from our hypothesis function, we can reduce the weight that some of the terms in our function carry by increasing their cost.

Say we wanted to make the following function more quadratic:

$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

We'll want to eliminate the influence of $\theta_3 x^3$ and $\theta_4 x^4$. Without actually getting rid of these features or changing the form of our hypothesis, we can instead modify our **cost function**:

$$min_\theta \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2 + 1000 \cdot \theta_3^2 + 1000 \cdot \theta_4^2$$

We've added two extra terms at the end to inflate the cost of θ_3 and θ_4. Now, in order for the cost function to get close to zero, we will have to reduce the values of θ_3 and θ_4 to near zero. This will in turn greatly reduce the values of $\theta_3 x^3$ and $\theta_4 x^4$ in our hypothesis function.

We could also regularize all of our theta parameters in a single summation:

$$min_\theta \frac{1}{2m} \left[\sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^n \theta_j^2\right]$$

The λ, or lambda, is the **regularization parameter**. It determines how much the costs of our theta parameters are inflated. You can visualize the effect of regularization in this interactive plot : https://www.desmos.com/calculator/1hexc8ntqp

Using the above cost function with the extra summation, we can smooth the output of our hypothesis function to reduce overfitting. If lambda is chosen to be too large, it may smooth out the function too much and cause underfitting.

Regularized Linear Regression

We can apply regularization to both linear regression and logistic regression. We will approach linear regression first.

Gradient Descent

We will modify our gradient descent function to separate out θ_0 from the rest of the parameters because we do not want to penalize θ_0.

$$
\begin{aligned}
&\text{Repeat } \{\\
&\quad \theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)}\\
&\quad \theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m}\theta_j \qquad j \in \{1,2...n\}\\
&\}
\end{aligned}
$$

The term $\frac{\lambda}{m}\theta_j$ performs our regularization.

With some manipulation our update rule can also be represented as:

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

The first term in the above equation, $1 - \alpha\frac{\lambda}{m}$ will always be less than 1. Intuitively you can see it as reducing the value of θ_j by some amount on every update.

Notice that the second term is now exactly the same as it was before.

Normal Equation

Now let's approach regularization using the alternate method of the non-iterative normal equation.

To add in regularization, the equation is the same as our original, except that we add another term inside the parentheses:

$$
\begin{aligned}
&\theta = \left(X^T X + \lambda \cdot L\right)^{-1} X^T y\\
&\text{where } L = \begin{bmatrix} 0 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix}
\end{aligned}
$$

L is a matrix with 0 at the top left and 1's down the diagonal, with 0's everywhere else. It should have dimension (n+1)×(n+1). Intuitively, this is the identity matrix (though we are not including x_0), multiplied with a single real number λ.

Recall that if m ≤ n, then $X^T X$ is non-invertible. However, when we add the term λ⋅L, then $X^T X$ + λ⋅L becomes invertible.

Regularized Logistic Regression

We can regularize logistic regression in a similar way that we regularize linear regression. Let's start with the cost function.

Cost Function

Recall that our cost function for logistic regression was:

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

We can regularize this equation by adding a term to the end:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^m [y^{(i)}\ \log(h_\theta(x^{(i)})) + (1 - y^{(i)})\ \log(1 - h_\theta(x^{(i)}))] + \frac{\lambda}{2m}\sum_{j=1}^n \theta_j^2$$

Note Well: The second sum, $\sum_{j=1}^n \theta_j^2$ **means to explicitly exclude** the bias term, θ_0. I.e. the θ vector is indexed from 0 to n (holding n+1 values, θ_0 through θ_n), and this sum explicitly skips θ_0, by running from 1 to n, skipping 0.

Gradient Descent

Just like with linear regression, we will want to **separately** update θ_0 and the rest of the parameters because we do not want to regularize θ_0.

$$
\begin{aligned}
&\text{Repeat } \{\\
&\quad \theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)}\\
&\quad \theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m}\theta_j \qquad j \in \{1,2...n\}\\
&\}
\end{aligned}
$$

This is identical to the gradient descent function presented for linear regression.

Initial Ones Feature Vector

Constant Feature

As it turns out it is crucial to add a constant feature to your pool of features before starting any training of your machine. Normally that feature is just a set of ones for all your training examples.

Concretely, if X is your feature matrix then X_0 is a vector with ones.

Below are some insights to explain the reason for this constant feature. The first part draws some analogies from electrical engineering concept, the second looks at understanding the ones vector by using a simple machine learning example.

Electrical Engineering

From electrical engineering, in particular signal processing, this can be explained as DC and AC.

The initial feature vector X without the constant term captures the dynamics of your model. That means those features particularly record changes in your output y - in other words changing some feature X_i where $i \neq 0$ will have a change on the output y. AC is normally made out of many components or harmonics; hence we also have many features (yet we have one DC term).

The constant feature represents the DC component. In control engineering this can also be the steady state.

Interestingly removing the DC term is easily done by differentiating your signal - or simply taking a difference between consecutive points of a discrete signal (it should be noted that at this point the analogy is implying time-based signals - so this will also make sense for machine learning application with a time basis - e.g. forecasting stock exchange trends).

Another interesting note: if you were to play and AC+DC signal as well as an AC only signal where both AC components are the same then they would sound exactly the same. That is because we only hear changes in signals and Δ(AC+DC)=Δ(AC).

Housing price example

Suppose you design a machine which predicts the price of a house based on some features. In this case what does the ones vector help with?

Let's assume a simple model which has features that are directly proportional to the expected price i.e. if feature Xi increases so the expected price y will also increase. So as an example we could have two features: namely the size of the house in [m2], and the number of rooms.

When you train your machine you will start by prepending a ones vector X_0. You may then find after training that the weight for your initial feature of ones is some value θ0. As it turns, when applying your hypothesis function $h_\theta(X)$ - in the case of the initial feature you will just be multiplying by a constant (most probably θ0 if you not applying any other functions such as sigmoids). This constant (let's say it's θ_0 for argument's sake) is the DC term. It is a constant that doesn't change.

But what does it mean for this example? Well, let's suppose that someone knows that you have a working model for housing prices. It turns out that for this example, if they ask you how much money they can expect if they sell the house you can say that they need at least θ0 dollars (or rands) before you even use your learning machine. As with the above analogy, your constant θ0 is somewhat of a steady state where all your inputs are zeros. Concretely, this is the price of a house with no rooms which takes up no space.

However this explanation has some holes because if you have some features which decrease the price e.g. age, then the DC term may not be an absolute minimum of the price. This is because the age may make the price go even lower.

Theoretically if you were to train a machine without a ones vector $f_{AC}(X)$, it's output may not match the output of a machine which had a ones vector $f_{DC}(X)$. However, $f_{AC}(X)$ may have exactly the same trend as $f_{DC}(X)$ i.e. if you were to plot both machine's output you would find that they may look exactly the same except that it seems one output has just been shifted (by a constant). With reference to the housing price problem: suppose you make predictions on two houses $house_A$ and $house_B$ using both machines. It turns out while the outputs from the two machines would different, the difference between houseA and houseB's predictions according to both machines could be exactly the same. Realistically, that means a machine trained without the ones vector f_AC could actually be very useful if you have just one benchmark point. This is because you can find out the missing constant by simply taking a difference between the machine's prediction an actual price - then when making predictions you simply add that constant to what even output you get. That is: if $house_{benchmark}$ is your benchmark then the DC component is simply $price(house_{benchmark}) - f_{AC}(features(house_{benchmark}))$

A more simple and crude way of putting it is that the DC component of your model represents the inherent bias of the model. The other features then cause tension in order to move away from that bias position.

coursera

Kholofelo Moyaba

A simpler approach

A "bias" feature is simply a way to move the "best fit" learned vector to better fit the data. For example, consider a learning problem with a single feature X_1. The formula without the X_0 feature is just $theta_1 * X_1 = y$. This is graphed as a line that always passes through the origin, with slope y/theta. The x_0 term allows the line to pass through a different point on the y axis. This will almost always give a better fit. Not all best fit lines go through the origin (0,0) right?

Joe Cotton