Week 3 Lecture Notes

ML Logistic Regression

Now we can extend the binary classification problem to **classification problem**. Note that sometimes the name "Logistic Regression" is a misnomer that may be historical reasons and is actually an approach for 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.

$y \in \{0, 1\}$

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

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

Given a training set of pairs (feature/output) pairs of predictors/responses (x, y), x and y drawn from X and Y, the mathematical goal is to find a function h such that $h(x) \approx y$.

We'll begin with the simplest case of a linear function of the form:

$h(x) = x_0 + x_1 x$

Notice that we're working with just one variable, except that we could add higher order terms, such as x^2, x^3, etc.

Plot the data from the training set. If the data looks like the following, it is obvious it should be a linear function:

The linear function that minimizes the squared error is the one that passes through the means of x and y. In this case, it looks like it might be the line we've drawn above.

We can add a threshold to our function, $h(x)$, so that it returns 0 if $h(x) < 0.5$ and 1 if $h(x) \geq 0.5$.

Now we can use this function to make predictions. We'll classify everything to the right of the threshold as 1 and everything to the left as 0.

This seems to work fine for this instance of the problem.

However, if we add another training example, it drastically changes the slope of our line, and the threshold no longer classifies our data well.

Fortunately, this kind of behavior is unusual for classification problems.

But it does illustrate why using linear regression is a bad idea for classification problems.

Linear regression can be thought of as finding the most probable function, given the training data. In this case, it looks like the training data is well-modeled by a linear function. But, this logic breaks down when dealing with classes.

This is because the probabilities that we're interested in for classification problems are not linear functions of the input variables.

Decision Boundary

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

$h_\theta(x) \geq 0 \Rightarrow y = 1$

$h_\theta(x) < 0 \Rightarrow y = 0$

The input to our hypothesis function is linear. It can be thought of as the decision boundary.

$z = \theta^T x$

$z = 0$

For example:

$\theta_0 = 5$

$\theta_1 = 1$

$\theta_2 = 0$

So $z = 5 + x_1 + 0x_2 = 0$ is the decision boundary.

We can plot the decision boundary by plotting $x_1 = -5$, as follows:

Our hypothesis function can also be non-linear:

$z = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2$

$z = 5 + x_1^2 + x_2^2$

$z = 0$

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:

$\theta = \begin{bmatrix} 5 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$

$z = 5 + x_1^2 + x_2^2$

$z = 5 + x_1^2 + x_2^2 \geq 0 \Rightarrow y = 1$

$z = 5 + x_1^2 + x_2^2 < 0 \Rightarrow y = 0$

In this case, our decision boundary is a circle with a radius of 5 around the origin.

Cost Function

We cannot use the same cost function that we used 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 convex function.

Instead, our cost function for logistic regression looks like:

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

$$Cost(h_\theta(x), y) = -\log(1 - h_\theta(x)) \quad \text{if } y = 0$$

Notice that when y is equal to 1, then we get a slight penalty for a prediction that is also 1, but we incur a big penalty if the prediction is 0. And the other way around if y is equal to 0.

We can also write our cost function as a piecewise function:

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

Notice that when y is equal to 1, then we get the left side of the function, $-\log(h_\theta(x))$, and when y is equal to 0, then we get the right side, $-\log(1 - h_\theta(x))$.

Given how we can always predict something with certainty given the following scenarios:

1. $y = 1$, $h_\theta(x) = 1$
2. $y = 0$, $h_\theta(x) = 0$

Then we can see that the cost function will be 0 for these scenarios.

But as the predicted probability approaches 0 for $y = 1$, and approaches 1 for $y = 0$, the cost function approaches infinity.

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

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

Notice that when y is equal to 1, then the left side of the equation drops out. When y is equal to 0, then the right side drops out.

We can now write our entire cost function as follows:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)})$$

Which factors into:

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

A vectorized implementation is:

$$h = g(X\theta)$$

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

Simplified Cost Function and Gradient Descent

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

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

Notice that when y is equal to 1, then the second term $(1 - y) \log(1 - h_\theta(x))$ will zero out, leaving $-\log(h_\theta(x))$, and when y is equal to 0, then the first term $-y \log(h_\theta(x))$ will zero out, leaving $-\log(1 - h_\theta(x))$.

We can vectorize our cost function as follows:

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

Gradient Descent

Remember that the general form of gradient descent is:

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

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

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

We can then write our update step as:

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

Advanced Optimization

"Conjugate gradient", "BFGS", and "L-BFGS" are more sophisticated, better ways to optimize θ that can be used instead of gradient descent. We don't need to manually pick α and they often make more progress per iteration. Unfortunately they are more complicated to understand and use than the simple gradient descent algorithm, so we'll be treating them as black-box optimizers for these problem sheets.

Suppose we have a function that calculates 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 need to send to "fminunc()". (Note: the value for "MaxIter" should be an integer, and we're telling "fminunc()" to use our cost function that we've created above.)

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

We give it 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

We can extend our logistic regression to handle more than two classes categories. Instead of $y = \{0, 1\}$ we'll expand our definition to be $y = \{0, 1, \ldots, n\}$.

In this case we divide our problem into $n+1$ classes (+1 because the index of 0 is also a classification class), in our notes, we'll predict the probability that y is a member of each of our classes.

$y \in \{0, 1, \ldots, n\}$

$h_\theta^{(0)}(x) = P(y = 0 | x; \theta)$

$h_\theta^{(1)}(x) = P(y = 1 | x; \theta)$

$\cdots$

$h_\theta^{(n)}(x) = P(y = n | x; \theta)$

$\text{prediction} = \max_i(h_\theta^{(i)}(x))$

We extensively checking one class versus all other classes one at a time. We train a logistic regression classifier for each class i to predict the probability that $y = i$. To make a prediction on a new input x, we pick the class that maximizes: $\max_i(h_\theta^{(i)}(x))$

ML Regularization

The Problem of Overfitting

Regularization is an approach to address the problem of overfitting.

High bias is a underfitting problem and is the result of a polynomial function or a linear function that is too rigid to fit the data. It is usually caused by a function that is too simple or using too few features. In the case of linear regression, it means that the data cannot be accurately represented by a straight line, and the resulting model will be a poor predictor.

Underfitting or high bias is when the form of our hypothesis function h maps poorly to the trend of the data. It is usually caused by a function too simple or using too few features. For example, if our data is in the form of a curve, but we try to fit a straight line to it, then we are underfitting the data. In this case, we need to add more features to our hypothesis function in order to capture the data.

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 in order to fit all the data points.

This is happening to a polynomial function and logistic regression. There are two 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 (details 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 bear by increasing their cost.

For example, we can make the following function more quadratic:

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

We 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 cause the values of $\theta_3 x^3$ and $\theta_4 x^4$ to vanish or at least become very small, leaving us with a much more simple function.

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

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

The 1000 is a bit arbitrary. We'll talk about how to choose this later.

Regularized Linear Regression

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

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

The second sum, $\sum_{j=1}^{n} \theta_j^2$ means to explicitly exclude the bias term, θ_0. I.e. we start the sum from $j = 1$ instead of 0. This is because we've added a column of 1's to our data set so that $x_0 = 1$, and θ_0 is a special term in our hypothesis function.

Even without the bias term, θ_0, we still have $n+1$ parameters to θ: $\theta_0, \theta_1, \ldots, \theta_n$.

Thus, when we perform our regularization:

$$\theta_j = \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \right] \quad \text{for } j = 0 \\ \theta_j = \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} \theta_j \right] \quad \text{for } j \geq 1$$

We can also write our cost function in a single line as:

$$\theta := \theta(1 - \alpha \frac{\lambda}{m}) - \alpha \frac{1}{m} X^T (g(X\theta) - y)$$

The factor, λ, is the **regularization parameter**. It determines how much the cost of our theta parameters are inflated. You can imagine that if λ is very large, then all of our thetas will be very large, and we'll have a high penalty.

Using the above cost function with the extra summation, we can smooth the output of our hypothesis function to reduce overfitting. If we decide to choose for λ to be very large, it may underfitting the data and we'll need to choose a smaller value for λ.

Regularized Linear Regression

We can extend our linear regression to handle more features and logistic regression. We will expand on 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.

$$\text{Repeat} \{ \\ \theta_0 := \theta_0 - \frac{\alpha}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)} \\ \theta_j := \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} \theta_j \right] \quad \frac{\lambda}{m} \theta_j \\ \}$$

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 term $1 - \alpha \frac{\lambda}{m}$ will always be less than 1. Intuitively you can see it as reducing the value of θ_j on every iteration of gradient descent.

Notice that the second term is now exactly the same as it used to be.

Normal Equation

Now let's represent regularization using the alternative vectorized non-iterative normal equation.

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

$$\theta = (X^T X + \lambda L)^{-1} X^T y$$

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

Recall that if $m \leq n$, then $X^T X$ is non-invertible. However, when we add the term λL, then $X^T X + \lambda L$ 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:

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

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

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

The second sum, $\sum_{j=1}^{n} \theta_j^2$ means to explicitly exclude the bias term, θ_0. I.e. we start the sum from $j = 1$ instead of 0. This is because we've added a column of 1's to our data set so that $x_0 = 1$, and θ_0 is a special term in our hypothesis function.

Thus, when we perform our regularization:

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)} \\ \theta_j := \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} \theta_j \right] \quad \frac{\lambda}{m} \theta_j$$

Gradient Descent

Like linear regression, we will want to **separately** update θ_0 and the rest of the parameters because we do not want to penalize θ_0.

$$\text{Repeat} \{ \\ \theta_0 := \theta_0 - \frac{\alpha}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)} \\ \theta_j := \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} \theta_j \right] \quad \frac{\lambda}{m} \theta_j \\ \}$$

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

Initial Ones Feature Vector

Constant Feature

As it turns out it's crucial to add a constant feature to your set of features before starting gradient descent or your machine. Normally that means adding a column of 1's to your data matrix, X.

Without this, your algorithm will not be able to fit a line with the y-axis.

Below, we've attempted to adjust the cost function to this modification. The first part is the same as regular linear regression (regularizing second), the second is just a rebalancing the same cost function to include a constant feature (regularizing a bias).

Electrical Engineering

From electrical engineering, the equation for capacitance, C, can be explained as $\frac{Q}{V}$ and $\frac{dV}{dt}$.

Capacitance is a measure of how much charge a conductor can hold. Capacitance depends on the size of the conductor, length, and the material (specifically relative dielectric constant). ε is the relative dielectric constant, A is the area of the conductor, and d is the distance between the conductor and its opposite charge.

$$C = \frac{\varepsilon A}{d}$$

Capacitance is a measure of how much charge you can store by providing a potential difference. Capacitance is a measure of how much charge you can store before a conductor's current starts to leak off. Capacitance is a measure of how much charge you can store before the dielectric breaks down and allows current to flow.

Electrical engineers use the Laplace transform to analyze circuits. The Laplace transform is a function that takes a function of time and converts it into a function of complex frequency. Laplace transforms are used to convert differential equations into algebraic equations, which are easier to solve.

Electrical engineers use the Fourier transform to analyze signals. The Fourier transform is a function that takes a function of time and converts it into a function of frequency. Fourier transforms are used to convert time domain signals into frequency domain signals, which are easier to analyze.

Electrical engineers use the Z transform to analyze discrete time signals. The Z transform is a function that takes a function of time and converts it into a function of complex frequency. Z transforms are used to convert time domain signals into frequency domain signals, which are easier to analyze.

Electrical Engineering

From electrical engineering, the equation for capacitance, C, can be explained as $\frac{Q}{V}$ and $\frac{dV}{dt}$.

Capacitance is a measure of how much charge a conductor can hold. Capacitance depends on the size of the conductor, length, and the material (specifically relative dielectric constant). ε is the relative dielectric constant, A is the area of the conductor, and d is the distance between the conductor and its opposite charge.

$$C = \frac{\varepsilon A}{d}$$

Capacitance is a measure of how much charge you can store by providing a potential difference. Capacitance is a measure of how much charge you can store before a conductor's current starts to leak off. Capacitance is a measure of how much charge you can store before the dielectric breaks down and allows current to flow.

Electrical engineers use the Laplace transform to analyze circuits. The Laplace transform is a function that takes a function of time and converts it into a function of complex frequency. Laplace transforms are used to convert differential equations into algebraic equations, which are easier to solve.

Electrical engineers use the Fourier transform to analyze signals. The Fourier transform is a function that takes a function of time and converts it into a function of frequency. Fourier transforms are used to convert time domain signals into frequency domain signals, which are easier to analyze.

Electrical engineers use the Z transform to analyze discrete time signals. The Z transform is a function that takes a function of time and converts it into a function of complex frequency. Z transforms are used to convert time domain signals into frequency domain signals, which are easier to analyze.

Machine Maintenance

A simpler approach:

A "Fast" Fourier transform is a processing step chosen the "fast" Fourier transform is better for the data. For example, consider a sensing problem which outputs Brown, B_t. The time-dependence is a function that always changes. Brown's function, B_t, is a function of time. Brown's function, B_t, is a function of time. Brown's function, B_t, is a function of time