Machine Learning Lecture Notes

by Andrew Ng

Transcripted by: Fernando García de la Cruz

Chapter 1

Week 1

1.1 What is Machine Learning?

Two definitions of Machine Learning are offered. **Arthur Samuel** described it as: **“the field of study that gives computers the ability to learn without being explicitly programmed.”** This is an older, informal definition.

Tom Mitchell provides a more modern definition:

> “A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.”

Example: playing checkers.

E = the experience of playing many games of checkers
T = the task of playing checkers.
P = the probability that the program will win the next game.

In general, any machine learning problem can be assigned to one of two broad classifications: **supervised learning, or unsupervised learning**.

1.2 Supervised Learning

In supervised learning, we are given a data set and already know what our correct output should look like, having the idea that there is a relationship between the input and the output.

Supervised learning problems are categorized into “regression” and “classification” problems. In a regression problem, we are trying to predict results within a continuous output, meaning that we are trying to map input variables to some continuous function. In a classification problem, we are instead trying to predict results in a discrete output. In other words, we are trying to map input variables into discrete categories. Here is a description on Math is Fun on Continuous and Discrete Data.

1.2.1 Example 1

Given data about the size of houses on the real estate market, try to predict their price. Price as a function of size is a continuous output, so this is a regression problem.

We could turn this example into a classification problem by instead making our output about whether the house "sells for more or less than the asking price." Here we are classifying the houses based on price into two discrete categories.

1.2.2 Example 2

A) **Regression** - Given a picture of Male/Female, We have to predict his/her age on the basis of given picture.

B) **Classification** - Given a picture of Male/Female, We have to predict whether He/She is of High school, College, Graduate age. Another Example for Classification - Banks have to decide whether or not to give a loan to someone on the basis of his credit history.

1.3 Unsupervised Learning

Unsupervised learning, on the other hand, allows us to approach problems with little or no idea what our results should look like. We can derive structure from data where we don't necessarily know the effect of the variables.

We can derive this structure by clustering the data based on relationships among the variables in the data.

With unsupervised learning there is no feedback based on the prediction results, i.e., **there is no teacher to correct you**.

1.3.1 Example

Clustering: Take a collection of 1000 essays written on the US Economy, and find a way to automatically group these essays into a small number that are somehow similar or related by different variables, such as word frequency, sentence length, page count, and so on.

Non-clustering: The "Cocktail Party Algorithm", which can find structure in messy data (such as the identification of individual voices and music from a mesh of sounds at a cocktail party) . Here is an answer on Quora to enhance your understanding: click here!.

1.4 ML:Linear Regression with One Variable

1.4.1 Model Representation

Recall that in regression problems, we are taking input variables and trying to fit the output onto a continuous expected result function.

Linear regression with one variable is also known as "univariate linear regression".

Univariate linear regression is used when you want to predict a **single output** value y from a **single input** value x. We're doing **supervised learning** here, so that means we already have an idea about what the input/output cause and effect should be.

1.4.2 The Hypothesis Function

Our hypothesis function has the general form:

$$\hat{y} = h_\theta(x) = \theta_0 + \theta_1 x \tag{1.1}$$

Note that this is like the equation of a straight line. We give to $h_\theta(x)$ values for θ_0 and θ_1 to get our estimated output output $\hat{y}$ In other words, we are trying to create a function called h_θ that is trying to map our input data (the x's) to our output data (the y's).

Example:
Suppose we have the following set of training data:

Input x	**Output** y
0	4
1	7
2	7
3	8

Now we can make a random guess about our h_θ function $\theta_0 = 2$ and $\theta_1 = 2$. The hypothesis function becomes $h_\theta = 2 + 2x$

So for input of 1 to our hypothesis, y will be 4. This is off by 3. Note that we will be trying out various values of θ_0 and θ_1 to try to find values which provide the best possible "fit" or the most representative "straight line" through the data points mapped on the x-y plane.

1.4.3 Cost Function

We can measure the accuracy of our hypothesis function by using a **cost function**. This takes an average (actually a fancier version of an average) of all the results of the hypothesis with inputs from x's compared to the actual output y's.

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)^2 \tag{1.2}$$

To break it apart, it is $\frac{1}{2}\bar{x}$ where $\bar{x}$ is the mean of the squares of $(h_\theta(x_i) - y_i)$, or the difference between the predicted value and the actual value.

This function is otherwise called the **"Squared error function"**, or **"Mean squared error"**. The mean is halved $\left(\frac{1}{2m}\right)$ as a convenience for the computation of the gradient descent, as the derivative term of the square function will cancel out the $\frac{1}{2}$ term.

Now we are able to concretely measure the accuracy of our predictor function against the correct results we have so that we can predict new results we don't have.

If we try to think of it in visual terms, our training data set is scattered on the x-y plane. We are trying to make straight line (defined by $h_\theta(x)$) which passes through this scattered set of data. Our objective is to get the best possible line. The best possible line will be such so that the average squared vertical distances of the scattered points from the line will be the least. In the best case, the line should pass through all the points of our training data set. In such a case the value of $J(\theta_0, \theta_1)$ will be 0.

1.5 ML: Gradient Descent

So we have our hypothesis function and we have a way of measuring how well it fits into the data. Now we need to estimate the parameters in hypothesis function. That's where gradient descent comes in.

Imagine that we graph our hypothesis function based on its fields θ_0 and θ_1 (actually we are graphing the cost function as a function of the parameter estimates). This can be kind of confusing; we are moving up to a higher level of abstraction. We are not graphing x and y itself, but the parameter range of our hypothesis function and the cost resulting from selecting particular set of parameters.

We put θ_0 on the x axis and θ_1 on the y axis, with the cost function on the vertical z axis. The points on our graph will be the result of the cost function using our hypothesis with those specific theta parameters.

We will know that we have succeeded when our cost function is at the very bottom of the pits in our graph, i.e. when its value is the minimum.

The way we do this is by taking the derivative (the tangential line to a function) of our cost function. The slope of the tangent is the derivative at that point and it will give us a direction to move towards. We make steps down the cost function in the direction with the steepest descent, and the size of each step is determined by the parameter α, which is called the learning rate.

The gradient descent algorithm is:

$$\begin{aligned} &\text{repeat until convergence: } \{ \\ &\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \\ &\} \end{aligned}$$

where

j=0,1 represents the feature index number.

Intuitively, this could be thought of as:

repeat until convergence:

$$\theta_j := -\alpha\delta$$

where: δ = [Slope of tangent aka derivative in j dimension]

1.5.1 Gradient Descent for Linear Regression

When specifically applied to the case of linear regression, a new form of the gradient descent equation can be derived. We can substitute our actual cost function and our actual hypothesis function and modify the equation to (the derivation of the formulas are out of the scope of this course, but a really great one can be found here):

$$\begin{aligned} &\text{repeat until convergence: } \{ \\ &\qquad \theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i) \\ &\qquad \theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} ((h_\theta(x_i) - y_i) x_i) \\ &\qquad \} \end{aligned}$$

where m is the size of the training set, θ_0 a constant that will be changing simultaneously with θ_1 and x_i, y_i are values of the given training set (data).

Note that we have separated out the two cases for θ_j into separate equations for θ_0 and θ_1; and that for θ_1 we are multiplying x_i at the end due to the derivative.

The point of all this is that if we start with a guess for our hypothesis and then repeatedly apply these gradient descent equations, our hypothesis will become more and more accurate.

1.5.2 Gradient Descent for Linear Regression: visual worked example

Some may find the following video useful as it visualizes the improvement of the hypothesis as the error function reduces.

1.6 ML:Linear Algebra Review

Khan Academy has excellent Linear Algebra Tutorials: click here!

1.6.1 Matrices and Vectors

Matrices are 2-dimensional arrays:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix}$$

The above matrix has four rows and three columns, so it is a 4 x 3 matrix.

A vector is a matrix with one column and many rows:

$$\begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix}$$

So vectors are a subset of matrices. The above vector is a 4 x 1 matrix.

Notation and terms:

- A_{ij} refers to the element in the ith row and jth column of matrix A.
- A vector with 'n' rows is referred to as an 'n'-dimensional vector
- v_i refers to the element in the ith row of the vector.
- In general, all our vectors and matrices will be 1-indexed. Note that for some programming languages, the arrays are 0-indexed.
- Matrices are usually denoted by uppercase names while vectors are lowercase.
- "Scalar" means that an object is a single value, not a vector or matrix.
- $\mathbb{R}$ refers to the set of scalar real numbers
- $\mathbb{R}^n$ refers to the set of n-dimensional vectors of real numbers

1.6.2 Addition and Scalar Multiplication

Addition and subtraction are **element-wise**, so you simply add or subtract each corresponding element:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} a+w & b+x \\ c+y & d+z \end{bmatrix}$$

To add or subtract two matrices, their dimensions **must be the same**.

In scalar multiplication, we simply multiply every element by the scalar value:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} * x = \begin{bmatrix} a*x & b*x \\ c*x & d*x \end{bmatrix}$$

1.6.3 Matrix-Vector Multiplication

We map the column of the vector onto each row of the matrix, multiplying each element and summing the result.

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a*x+b*y \\ c*x+d*y \\ e*x+f*y \end{bmatrix}$$

The result is a vector. The vector must be the second term of the multiplication. The number of columns of the matrix must equal the number of rows of the vector.

An m x n matrix multiplied by an n x 1 vector results in an m x 1 vector.

1.6.4 Matrix-Matrix Multiplication

We multiply two matrices by breaking it into several vector multiplications and concatenating the result

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} * \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} a*w+b*y & a*x+b*z \\ c*w+d*y & c*x+d*z \\ e*w+f*y & e*x+f*z \end{bmatrix}$$

An m x n matrix multiplied by an n x o matrix results in an m x o matrix. In the above example, a 3 x 2 matrix times a 2 x 2 matrix resulted in a 3 x 2 matrix.

To multiply two matrices, the number of columns of the first matrix must equal the number of rows of the second matrix.

1.6.5 Matrix Multiplication Properties

- Not commutative. $A * B \neq B * A$
- Associative. $(A * B) * C = A * (B * C)$

The identity matrix, when multiplied by any matrix of the same dimensions, results in the original matrix. It's just like multiplying numbers by 1. The identity matrix simply has 1's on the diagonal (upper left to lower right diagonal) and 0's elsewhere.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

When multiplying the identity matrix after some matrix $(A*I)$, the square identity matrix should match the other matrix's columns. When multiplying the identity matrix before some other matrix $(I * A)$, the square identity matrix should match the other matrix's rows.

1.6.6 Inverse and Transpose

The inverse of a matrix A is denoted A-1. Multiplying by the inverse results in the identity matrix.

A non square matrix does not have an inverse matrix. We can compute inverses of matrices in octave with the `pinv(A)` function [1] and in matlab with the `inv(A)` function. Matrices that don't have an inverse are singular or degenerate.

The transposition of a matrix is like rotating the matrix 90° in clockwise direction and then reversing it. We can compute transposition of matrices in matlab with the `transpose(A)` function or A':

$$A = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}$$

$$A^T = \begin{bmatrix} a & c & e \\ b & d & f \end{bmatrix}$$

In other words:

$$A_{ij} = A^T_{ji}$$

Chapter 2

Week 2

2.1 ML:Linear Regression with Multiple Variables

Linear regression with multiple variables is also known as “multivariate linear regression”.

We now introduce notation for equations where we can have any number of input variables.

$$\begin{aligned} x_j^{(i)} &= \text{value of feature } j \text{ in the } i^{th} \text{ training example} \\ x^{(i)} &= \text{the column vector of all the feature inputs of the } i^{th} \text{ training example} \\ m &= \text{the number of training examples} \\ n &= \left| x^{(i)} \right| ; \text{(the number of features)} \end{aligned}$$

Now define the multivariable form of the hypothesis function as follows, accommodating these multiple features:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n \tag{2.1}$$

In order to develop intuition about this function, we can think about θ_0 as the basic price of a house, θ_1 as the price per square meter, θ_2 as the price per floor, etc. x_1 will be the number of square meters in the house, x_2 the number of floors, etc.

Using the definition of matrix multiplication, our multivariable hypothesis function can be concisely represented as:

$$h_\theta(x) = \begin{bmatrix} \theta_0 & \theta_1 & \ldots & \theta_n \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} = \theta^T x$$

This is a vectorization of our hypothesis function for one training example; see the lessons on vectorization to learn more.

Remark: Note that for convenience reasons in this course Mr. Ng assumes:

$$x_0^{(i)} = 1 \text{ for } (i \in 1, \ldots, m)$$

[Note: So that we can do matrix operations with theta and x, we will set $x_0^{(i)} = 1$, for all values of i. This makes the two vectors 'theta' and $x_{(i)}$ match each other element-wise (that is, have the same number of elements: n+1).]

The training examples are stored in X row-wise, like such:

$$X = \begin{bmatrix} x_0^{(1)} & x_1^{(1)} \\ x_0^{(2)} & x_1^{(2)} \\ x_0^{(3)} & x_1^{(3)} \end{bmatrix}, \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$$

You can calculate the hypothesis as a column vector of size (m x 1) with:

$$h_\theta(X) = X\theta \tag{2.2}$$

For the rest of these notes, and other lecture notes, X will represent a matrix of training examples $x_{(i)}$ **stored row-wise.**

2.2 Cost Function

For the parameter vector θ (of type $\mathbb{R}^{n+1}$ or in $\mathbb{R}^{(n+1)\times 1}$), the cost function is:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 \tag{2.3}$$

The vectorized version is:

$$J(\theta) = \frac{1}{2m} (X\theta - \vec{y})^T (X\theta - \vec{y}) \tag{2.4}$$

Where $\vec{y}$ denotes the vector of all y values.

2.3 Gradient Descent for Multiple Variables

The gradient descent equation itself is generally the same form; we just have to repeat it for our “n” features:

$$\begin{aligned} & \text{repeat until convergence: } \{ \\ & \theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)} \\ & \theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_1^{(i)} \\ & \theta_2 := \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_2^{(i)} \\ & \cdots \\ \} \end{aligned}$$

In other words:

```
repeat until convergence: {
    $\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$        for j := 0..n
}
```

2.4 Matrix Notation

The Gradient Descent rule can be expressed as:

$$\theta := \theta - \alpha \nabla J(\theta) \tag{2.5}$$

Where $\nabla J(\theta)$ is a column vector of the form:

$$\nabla J(\theta) = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_0} \\ \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix} \tag{2.6}$$

The j-th component of the gradient is the summation of the product of two terms:

$$\begin{aligned} \frac{\partial J(\theta)}{\partial \theta_j} =& \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot x_j^{(i)} \\ =& \frac{1}{m} \sum_{i=1}^{m} x_j^{(i)} \cdot \left(h_\theta(x^{(i)}) - y^{(i)} \right) \end{aligned}$$

Sometimes, the summation of the product of two terms can be expressed as the product of two vectors.

Here, $x_j^{(i)}$, for i = 1,...,m, represents the m elements of the j-th column, $\vec{x_j}$, of the training set X.

The other term $\left(h_\theta(x^{(i)}) - y^{(i)} \right)$ is the vector of the deviations between the predictions $h_\theta(x^{(i)})$ and the true values $y^{(i)}$. Re-writing $\frac{\partial J(\theta)}{\partial \theta_j}$, we have:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m} \vec{x_j}^T (X\theta - \vec{y})$$

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

Finally, the matrix notation (vectorized) of the Gradient Descent rule is:

$$\theta := \theta - \frac{\alpha}{m} X^T (X\theta - \vec{y}) \tag{2.7}$$

2.5 Feature Normalization

We can speed up gradient descent by having each of our input values in roughly the same range. This is because θ will descend quickly on small ranges and slowly on large ranges, and so will oscillate inefficiently down to the optimum when the variables are very uneven.

The way to prevent this is to modify the ranges of our input variables so that they are all roughly the same. Ideally:

$$-1 \leq x_{(i)} \leq 1$$

or

$$-0.5 \leq x_{(i)} \leq 0.5$$

These aren't exact requirements; we are only trying to speed things up. The goal is to get all input variables into roughly one of these ranges, give or take a few.

Two techniques to help with this are **feature scaling** and **mean normalization**. Feature scaling involves dividing the input values by the range (i.e. the maximum value minus the minimum value) of the input variable, resulting in a new range of just 1. Mean normalization involves subtracting the average value for an input variable from the values for that input variable, resulting in a new average value for the input variable of just zero. To implement both of these techniques, adjust your input values as shown in this formula:

$$x_i := \frac{x_i - \mu_i}{s_i} \tag{2.8}$$

Where μ_i is the **average** of all the values for feature (i) and s_i is the range of values (max - min), or s_i is the standard deviation.

Note that dividing by the range, or dividing by the standard deviation, give different results. The quizzes in this course use range - the programming exercises use standard deviation.

Example: x_i is housing prices with range of 100 to 2000, with a mean value of 1000.

Then, $x_i := \frac{price - 1000}{1900}$

2.6 Gradient Descent Tips

Debugging gradient descent. Make a plot with number of iterations on the x-axis. Now plot the cost function, $J(\theta)$ over the number of iterations of gradient descent. If $J(\theta)$ ever increases, then you probably need to decrease α.

Automatic convergence test. Declare convergence if $J(\theta)$ decreases by less than E in one iteration, where E is some small value such as 10-3. However in practice it's difficult to choose this threshold value.

It has been proven that if learning rate α is sufficiently small, then $J(\theta)$ will decrease on every iteration. Andrew Ng recommends decreasing α by multiples of 3.

2.6.1 Features and Polynomial Regression

We can improve our features and the form of our hypothesis function in a couple different ways.

We can combine multiple features into one. For example, we can combine x_1 and x_2 into a new feature x_3 by taking $x_1 \cdot x_2$

Polynomial Regression

Our hypothesis function need not be linear (a straight line) if that does not fit the data well.

We can **change the behavior or curve** of our hypothesis function by making it a quadratic, cubic or square root function (or any other form).

For example, if our hypothesis function is $h_\theta(x) = \theta_0 + \theta_1 x_1$ then we can create additional features based on x_1, to get the quadratic function $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2$ or the cubic function $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^3$

In the cubic version, we have created new features x_2 and x_3 where $x_2 = x_1^2$ and $x_3 = x_1^3$.

To make it a square root function, we could do: $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 \sqrt{x_1}$

Note that at 2:52 and through 6:22 in the "Features and Polynomial Regression" video, the curve that Prof Ng discusses about "doesn't ever come back down" is in reference to the hypothesis function that uses the sqrt() function (shown by the solid purple line), not the one that uses $size^2$ (shown with the dotted blue line). The quadratic form of the hypothesis function would have the shape shown with the blue dotted line if θ_2 was negative.

One important thing to keep in mind is, if you choose your features this way then feature scaling becomes very important.

eg. if x_1 has range 1 - 1000 then range of x_1^2 becomes 1 - 1000000 and that of x_1^3 becomes 1 - 1000000000.

2.7 Normal Equation

The "Normal Equation" is a method of finding the optimum theta **without iteration**.

$$\theta = (X^T X)^{-1} X^T y \tag{2.9}$$

There is no need to do feature scaling with the normal equation.

Mathematical proof of the Normal equation requires knowledge of linear algebra and is fairly involved, so you do not need to worry about the details.

Proofs are available at these links for those who are interested:

Wikipedia

thegreenplace

The following is a comparison of gradient descent and the normal equation:

Gradient Descent	**Normal Equation**
Need to choose alpha	No need to choose alpha
Needs may iterations	No need to iterate
$\mathcal{O}(kn^2)$	$\mathcal{O}(n^3)$, need to calculate inverse of X^TX
Works well when n is large	Slow if n is very large

With the normal equation, computing the inversion has complexity $\mathcal{O}(n^3)$. So if we have a very large number of features, the normal equation will be slow. In practice, when n exceeds 10,000 it might be a good time to go from a normal solution to an iterative process.

2.7.1 Normal Equation Noninvertibility

When implementing the normal equation in octave we want to use the `pinv` function rather than `inv`.

X^TX may be **noninvertible**. The common causes are:

- Redundant features, where two features are very closely related (i.e. they are linearly dependent)
- Too many features (e.g. $m \leq n$). In this case, delete some features or use “regularization” (to be explained in a later lesson).

Solutions to the above problems include deleting a feature that is linearly dependent with another or deleting one or more features when there are too many features.

Chapter 3

Week 3

3.1 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.

3.2 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 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"**:

$$h_\theta(x) = g(\theta^T x) \tag{3.1}$$

$$z = \theta^T x \tag{3.2}$$

$$g(z) = \frac{1}{1 + e^{-z}} \tag{3.3}$$

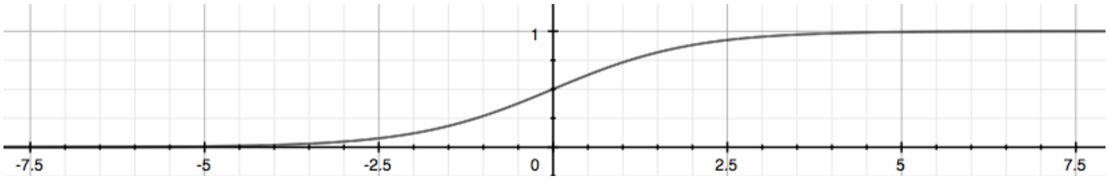

Figure 3.1: Sigmoid Function

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: click here!.

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 θ^T 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.

$$\begin{aligned} &h_\theta(x) = P(y = 1|x; \theta) = 1 - P(y = 0|x; \theta) \\ &P(y = 0|x; \theta) + P(y = 1|x; \theta) = 1 \end{aligned}$$

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%).

3.3 Decision Boundary

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

$$\begin{aligned} h_\theta(x) \geq 0.5 \rightarrow y = 1 \\ h_\theta(x) < 0.5 \rightarrow y = 0 \end{aligned}$$

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:

$$\begin{aligned} g(z) \geq 0.5 \\ \text{when } z \geq 0 \end{aligned}$$

Remember:

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

So if our input to g is θ^T, then that means:

$$\begin{aligned}&h_\theta(x) = g(\theta^T x) \geq 0.5\\ &\text{when } \theta^T x \geq 0\end{aligned}$$

From these statements we can now say:

$$\begin{aligned}&\theta^T x \geq 0 \Rightarrow y = 1\\ &\theta^T x < 0 \Rightarrow y = 0\end{aligned}$$

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:

$$\begin{aligned}&\theta = \begin{bmatrix} 5 \\ -1 \\ 0 \end{bmatrix}\\ &y = 1 \; if \; 5 + (-1)x_1 + 0x_2 \geq 0\\ &5 - x_1 \geq 0\\ &-x_1 \geq -5\\ &x_1 \leq 5\end{aligned}$$

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.

3.4 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:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \mathrm{Cost}(h_\theta(x^{(i)}), y^{(i)}) \tag{3.4}$$

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

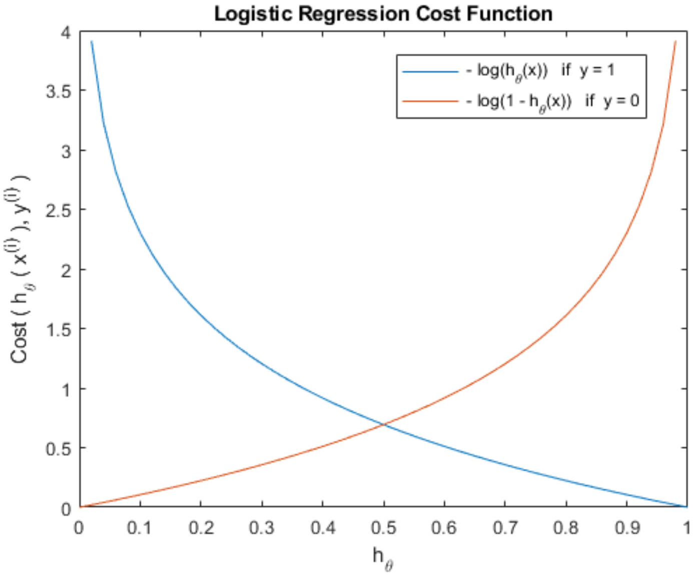

Figure 3.2: Logistic Regression Cost Function

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:

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

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 $J(\theta)$ is convex for logistic regression.

3.5 Simplified Cost Function and Gradient Descent

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

$$\mathrm{Cost}(h_\theta(x), y) = -y \; \log(h_\theta(x)) - (1 - y)\log(1 - h_\theta(x)) \tag{3.5}$$

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:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))] \tag{3.6}$$

A vectorized implementation is:

$$h = g(X\theta) \tag{3.7}$$

$$J(\theta) = \frac{1}{m}\cdot\left(-y^T\log(h) - (1-y)^T\log(1-h)\right) \tag{3.8}$$

3.5.1 Gradient Descent

Remember that the general form of gradient descent is:

$$\begin{aligned}&\mathit{Repeat}\ \{\\ &\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta)\\ &\}\end{aligned}$$

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

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

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:

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

3.5.2 Partial derivative of Cost Function

First calculate derivative of sigmoid function (it will be useful while finding partial derivative of $J(\theta)$

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

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

The vectorized version:

$$\nabla J(\theta) = \frac{1}{m} \cdot X^T \cdot (g(X \cdot \theta) - \vec{y}) \tag{3.9}$$

3.6 Advances 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] = costFunc(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);
initTheta = zeros(2,1);
[optTheta, funcVal, exitFlag] = fminunc(@costFunc, initTheta, options);
```

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

3.7 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.

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

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.

3.8 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
   - A) Keep all the features, but reduce the parameters θ_j
   - B) Regularization works well when we have a lot of slightly useful features.

3.9 Cost Function

f 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**:

$$min_\theta \ \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 + 1000 \cdot \theta_3^2 + 1000 \cdot \theta_4^2 \tag{3.10}$$

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:

$$min_\theta \ \frac{1}{2m} \left[\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n} \theta_j^2 \right] \tag{3.11}$$

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

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.

3.10 Regularized Linear Regression

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

3.10.1 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.

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

The term $\frac{\lambda}{m}\theta_j$ performs our regularization. With some manipulation our update rule can also be represented as:

$$\theta_j := \theta_j(1 - \alpha\frac{\lambda}{m}) - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \tag{3.12}$$

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.

3.10.2 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:

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

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 $(n+1) \times (n+1)$. Intuitively, this is the identity matrix (though we are not including x_0), multiplied with a single real number λ.

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

3.11 Regularized Logistic Regression

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

3.11.1 Cost Function

Recall that our cost function for logistic regression was:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\ \log(h_\theta(x^{(i)})) + (1-y^{(i)})\ \log(1-h_\theta(x^{(i)}))\right] \tag{3.13}$$

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

$$\begin{aligned}&J(\theta) = \\ &-\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\ \log(h_\theta(x^{(i)})) + (1-y^{(i)})\ \log(1-h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2\end{aligned} \tag{3.14}$$

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 θ 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.

3.11.2 Gradient Descent

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

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

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

3.12 Initial Ones Feature Vector

3.12.1 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.

3.12.2 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 $\Delta(AC + DC) = \Delta(AC)$

3.12.3 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 pretending a ones vector X_0. You may then find after training that the weight for your initial feature of ones is some value θ_0. 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 θ_0 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 θ_0 dollars (or rands) before you even use your learning machine. As with the above analogy, your constant θ_0 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.

Kholofelo Moyaba

3.12.4 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

Chapter 4

Week 4

4.1 ML: Neural Networks: Representation

4.1.1 Non-linear Hypothesis

Performing linear regression with a complex set of data with many features is very unwieldy. Say you wanted to create a hypothesis from three (3) features that included all the quadratic terms:

$$\begin{aligned}&g(\theta_0 + \theta_1 x_1^2 + \theta_2 x_1 x_2 + \theta_3 x_1 x_3\\&+ \theta_4 x_2^2 + \theta_5 x_2 x_3\\&+ \theta_6 x_3^2)\end{aligned}$$

That gives us 6 features. The exact way to calculate how many features for all polynomial terms is the combination function with repetition:

Combinations and permutations $\frac{(n+r-1)!}{r!(n-1)!}$.

In this case we are taking all two-element combinations of three features: $\frac{(3+2-1)!}{(2!\cdot(3-1)!)} = \frac{4!}{4} = 6$. (**Note**: you do not have to know these formulas, I just found helpful for understanding).

For 100 features, if we wanted to make the quadratic we would get $\frac{(100+2-1)!}{(2\cdot(100-1)!)} = 5050$ resulting new features.

We can approximate the growth of the number of new features we get with all quadratic terms with $\mathcal{O}(n^2/2)$. And if you wanted to include all cubic terms in your hypothesis, the features would grow asymptotically at $\mathcal{O}(n^3)$. These are very steep growths, so as the number of our features increase, the number of quadratic or cubic features increase very rapidly and becomes quickly impractical.

Example: let our training set be a collection of 50 x 50 pixel black-and-white photographs, and our goal will be to classify which ones are photos of cars. Our feature set size is then n = 2500 if we compare every pair of pixels.

Now let's say we need to make a quadratic hypothesis function. With quadratic features, our growth is $\mathcal{O}(n^2/2)$. So our total features will be about $2500^2/2 = 3125000$, which is very impractical.

Neural networks offers an alternate way to perform machine learning when we have complex hypotheses with many features.

4.2 Neurons and the Brain

Neural networks are limited imitations of how our own brains work. They've had a big recent resurgence because of advances in computer hardware.

There is evidence that **the brain uses only one "learning algorithm"** for all its different functions. Scientists have tried cutting (in an animal brain) the connection between the ears and the auditory cortex and rewiring the optical nerve with the auditory cortex to find that the auditory cortex literally learns to see.

This principle is called "**neuroplasticity**" and has many examples and experimental evidence.

4.3 Model Representation I

Let's examine how we will represent a hypothesis function using neural networks.

At a very simple level, neurons are basically computational units that take input **dendrites** as electrical input (called "**spikes**") that are channeled to outputs (**axons**).

In our model, our dendrites are like the input features $x_1 \cdots x_n$, and the output is the result of our hypothesis function:

In this model our $x0$ input node is sometimes called the "bias unit." It is always equal to 1.

In neural networks, we use the same logistic function as in classification: $\frac{1}{1+e^{-\theta^T x}}$. In neural networks however we sometimes call it a **sigmoid** (logistic) activation function.

Our "theta" parameters are sometimes instead called **weights** in the neural networks model.

Visually, a simplistic representation looks like:

$$[x_0 x_1 x_2] \rightarrow [\quad] \rightarrow h_\theta(x)$$

Our input nodes (layer 1) go into another node (layer 2), and are output as the hypothesis function.

The first layer is called the **input layer** and the final layer the **output layer**, which gives the final value computed on the hypothesis.

We can have intermediate layers of nodes between the input and output layers called the **hidden layer**.

We label these intermediate or **"hidden"** layer nodes $a_0^2 \cdots a_n^2$ and call them **activation units**.

$$\begin{aligned} a_i^{(j)} &= \text{“activation” of unit } i \text{ in layer } j \\ \Theta^{(j)} &= \text{matrix of weights controlling function mapping from layer } j \text{ to layer } j+1 \end{aligned}$$

If we had one hidden layer, it would look visually something like:

$$[x_0 x_1 x_2] \to [a_1^{(2)} a_2^{(2)} a_3^{(2)}] \to h_\theta(x)$$

The values for each **activation** nodes is obtained as follows:

$$\begin{aligned} a_1^{(2)} &= g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3) \\ a_2^{(2)} &= g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3) \\ a_3^{(2)} &= g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3) \end{aligned}$$

$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

This is saying that we compute our activation nodes by using a 3x4 matrix of parameters. We apply each row of the parameters to our inputs to obtain the value for one activation node. Our hypothesis output is the logistic function applied to the sum of the values of our activation nodes, which have been multiplied by yet another parameter matrix $\Theta^{(2)}$ containing the weights for our second layer of nodes.

Each layer gets its own matrix of weights, $\Theta^{(j)}$.

The dimensions of these matrices of weights is determined as follows:

> If network has s_j units in layer j and s_{j+1} units in layer $j+1$, then Θ_0^j will be of dimension $s_{s+j} \times (s_j + 1)$

The +1 comes from the addition in $\Theta^{(j)}$ of the **bias nodes**, x_0 and $\Theta_0^{(j)}$. In other words the output nodes will not include the bias nodes while the inputs will.

Example: layer 1 has 2 input nodes and layer 2 has 4 activation nodes. Dimension of $\Theta^{(1)}$ is going to be 4×3 where $s_j = 2$ and $s_{j+1} = 4$, so:

$$s_{j+1} \times (s_j + 1) = 4 \times 3$$

4.4 Model Representation II

In this section we'll do a vectorized implementation of the above functions. We're going to define a new variable $z_k^{(j)}$ that encompasses the parameters inside our g function. In our previous example if we replaced the variable z for all the parameters we would get:

$$\begin{aligned} a_1^{(2)} &= g(z_1^{(2)}) \\ a_2^{(2)} &= g(z_2^{(2)}) \\ a_3^{(2)} &= g(z_3^{(2)}) \end{aligned}$$

In other words, for layer j=2 and node k, the variable z will be:

$$z_k^{(2)} = \Theta_{k,0}^{(1)} x_0 + \Theta_{k,1}^{(1)} x_1 + \cdots + \Theta_{k,n}^{(1)} x_n$$

The vector representation of x and z^j is:

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_n \end{bmatrix} z^{(j)} = \begin{bmatrix} z_1^{(j)} \\ z_2^{(j)} \\ \cdots \\ z_n^{(j)} \end{bmatrix}$$

Setting $x = a^{(1)}$, we can rewrite the equation as:

$$z^{(j)} = \Theta^{(j-1)} a^{(j-1)}$$

We are multiplying our matrix $\Theta^{(j-1)}$ with dimensions $s_j \times (n+1)$ (where s_j is the number of our activation nodes) by our vector $a^{(j-1)}$ with height $(n+1)$. This gives us our vector $z^{(j)}$ with height s_j.

Now we can get a vector of our activation nodes for layer j as follows:

$$a^{(j)} = g(z^{(j)})$$

Where our function g can be applied element-wise to our vector $z^{(j)}$.

We can then add a bias unit (equal to 1) to layer j after we have computed $a^{(j)}$. This will be element $a_0^{(j)}$ and will be equal to 1.

To compute our final hypothesis, let's first compute another z vector:

$$z^{(j+1)} = \Theta^{(j)} a^{(j)}$$

We get this final z vector by multiplying the next theta matrix after $\Theta^{(j-1)}$ with the values of all the activation nodes we just got.

This last theta matrix $\Theta^{(j)}$ will have only one row so that our result is a single number.

We then get our final result with:

$$h_\theta(x) = a^{(j+1)} = g(z^{(j+1)}) \tag{4.1}$$

Notice that in this **last step**, between layer j and layer j+1, we are doing **exactly the same thing** as we did in logistic regression.

Adding all these intermediate layers in neural networks allows us to more elegantly produce interesting and more complex non-linear hypotheses.

4.5 Examples and Intuitions I

A simple example of applying neural networks is by predicting x_1 AND x_2, which is the logical '**and**' operator and is only true if both x_1 and x_2 are 1.

The graph of our functions will look like:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow \begin{bmatrix} g(z^{(2)}) \end{bmatrix} \rightarrow h_\Theta(x)$$

Remember that x_0 is our bias variable and is always 1.

Let's set our first theta matrix as:

$$\Theta^{(1)} = \begin{bmatrix} -30 & 20 & 20 \end{bmatrix}$$

This will cause the output of our hypothesis to only be positive if both x_1 and x_2 are 1. In other words:

$$\begin{aligned}
& h_\Theta(x) = g(-30 + 20x_1 + 20x_2) \\
& \\
& x_1 = 0 \;\; and \;\; x_2 = 0 \;\; then \;\; g(-30) \approx 0 \\
& x_1 = 0 \;\; and \;\; x_2 = 1 \;\; then \;\; g(-10) \approx 0 \\
& x_1 = 1 \;\; and \;\; x_2 = 0 \;\; then \;\; g(-10) \approx 0 \\
& x_1 = 1 \;\; and \;\; x_2 = 1 \;\; then \;\; g(10) \approx 1
\end{aligned}$$

So we have constructed one of the fundamental operations in computers by using a small neural network rather than using an actual **AND** gate. Neural networks can also be used to simulate all the other logical gates.

4.6 Examples and Intuitions II

The $\Theta^{(1)}$ matrices for **AND**, **NOR** and **OR** are:

$$\begin{aligned} AND: \quad & \Theta^{(1)} = \begin{bmatrix} -30 & 20 & 20 \end{bmatrix} \\ NOR: \quad & \Theta^{(1)} = \begin{bmatrix} 10 & -20 & -20 \end{bmatrix} \\ OR: \quad & \Theta^{(1)} = \begin{bmatrix} -10 & 20 & 20 \end{bmatrix} \end{aligned}$$

We can combine these to get the **XNOR** logical operator (which gives 1 if $x_1 x$ and x_2 are both 0 or both 1).

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \end{bmatrix} \rightarrow \begin{bmatrix} a^{(3)} \end{bmatrix} \rightarrow h_\Theta(x)$$

For the transition between the first and second layer, we'll use a $\Theta^{(1)}$ matrix that combines the values for **AND** and **NOR**:

$$\Theta^{(1)} = [-30 \quad 20 \quad 2010 \quad -20 \quad -20]$$

For the transition between second and third layer, we'll use a $\Theta^{(2)}$ matrix that combines the values for **OR**:

$$\Theta^{(1)} = [-30 \quad 20 \quad 2010 \quad -20 \quad -20]$$

$$\Theta^{(1)} = [-10 \quad 20 \quad 20]$$

Let's write out the values for all our nodes:

$$\begin{aligned} a^{(2)} &= g(\Theta^{(1)} \cdot x) \\ a^{(3)} &= g(\Theta^{(2)} \cdot a^{(2)}) \\ h_\Theta(x) &= a^{(3)} \end{aligned}$$

4.7 Multiclass Classification

To classify data into multiple classes, we let our hypothesis function return a vector of values. Say we wanted to classify our data into one of four final resulting classes:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix} \rightarrow \begin{bmatrix} a_0^{(2)} \\ a_1^{(2)} \\ a_2^{(2)} \\ \cdots \\ a_n^{(2)} \end{bmatrix} \rightarrow \begin{bmatrix} a_0^{(3)} \\ a_1^{(3)} \\ a_2^{(3)} \\ \cdots \\ a_n^{(3)} \end{bmatrix} \rightarrow \cdots \rightarrow \begin{bmatrix} h_\Theta(x)_1 \\ h_\Theta(x)_2 \\ h_\Theta(x)_3 \\ h_\Theta(x)_4 \end{bmatrix} \rightarrow$$

Our final layer of nodes, when multiplied by its theta matrix, will result in another vector, on which we will apply the g() logistic function to get a vector of hypothesis values.

Our resulting hypothesis for one set of inputs may look like:

$$h_\Theta(x) = [0010]$$

In which case our resulting class is the third one down, or $h_\Theta(x)_3$.

We can define our set of resulting classes as y:

$$y^{(i)} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Our final value of our hypothesis for a set of inputs will be one of the elements in y.

Chapter 5

Week 5

5.1 Neural Networks Learning

Let's first define a few variables that we will need to use:

A) L= total number of layers in the network

B) s_1 number of units(not including bias unit) in layer 1

C) K=number of outputs units/classes

Recall that the cost function for regularized logistic regression was:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\ \log\left(h_\theta(x^{(i)})\right) + (1-y^{(i)})\ \log\left(1-h_\theta(x^{(i)})\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \tag{5.1}$$

For neural networks, it is going to bee slightly more complicated:

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[y_k^{(i)}\log((h_\Theta(x^{(i)}))_k) + (1-y_k^{(i)})\log(1-(h_\Theta(x^{(i)}))_k)\right] + \cdots$$

$$\cdots + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{j,i}^{(l)})^2 \tag{5.2}$$

We have added a few nested summations to account for our multiple output nodes. In the first part of the equation, between the square brackets, we have an additional nested summation that loops through the number of output nodes.

In the regularization part, after the square brackets, we must account for multiple theta matrices. The number of columns in our current theta matrix is equal to the number of nodes in our current layer (including the bias unit). The number of rows in our current theta matrix is equal to the number of nodes in the next layer (excluding the bias unit). As before with logistic regression, we square every term.

Note:

- The double sum simply adds up the logistic regression costs calculated for each cell in the output layer; and
- The triple sum simply adds up the squares of all the individual Θ's in the entire network.
- The *i* in the triple sum **does not** refer to training example i

5.2 Backpropagation Algorithm

Backpropagation is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression.

Our goal is to compute:

$$\min_{\Theta} J(\Theta)$$

That is, we want to minimize our cost function J using an optimal set of parameters in theta.

In this section we'll look at the equations we use to compute the partial derivative of $J(\Theta)$:

$$\frac{\partial}{\partial \Theta_{i,j}^{(l)}} J(\Theta)$$

In back propagation we're going to compute for every node:

$$\delta_j^{(l)}$$

Recall that $a_j^{(l)}$ is activation node j in layer l.

For the last layer, we can compute the vector of delta values with:

$$\delta^{(L)} = a^{(L)} - y$$

Where L is our total number of layers and $a^{(L)}$ is the vector of outputs of the activation units for the last layer. So our "error values" for the last layer are simply the differences of our actual results in the last layer and the correct outputs in y.

To get the delta values of the layers before the last layer, we can use an equation that steps us back from right to left:

$$\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)})\ .*\ g'(z^{(l)})$$

The delta values of layer l are calculated by multiplying the delta values in the next layer with the theta matrix of layer l. We then element-wise multiply that with a function called g', or g-prime, which is the derivative of the activation function g evaluated with the input values given by $z(l)$.

The g-prime derivative terms can also be written out as:

$$g'(u) = g(u)\ .*\ (1 - g(u))$$

The full back propagation equation for the inner nodes is then:

$$\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)})\ .*\ a^{(l)}\ .*\ (1 - a^{(l)})$$

A. Ng states that the derivation and proofs are complicated and involved, but you can still implement the above equations to do back propagation without knowing the details.

We can compute our partial derivative terms by multiplying our activation values and our error values for each training example t:

$$\frac{\partial J(\Theta)}{\partial \Theta_{i,j}^{(l)}} = \frac{1}{m} \sum_{t=1}^{m} a_j^{(t)(l)} \delta_i^{(t)(l+1)}$$

This however ignores regularization, which we'll deal with later.

Note: δ^{l+1} and a^{l+1} are vectors with s_{l+1} elements. Similarly, $a^{(l)}$ is a vector with s_l elements. Multiplying them produces a matrix that is s_{l+1} by s_l which is the same dimension as $\Theta^{(l)}$. That is, the process produces a gradient term for every element in $\Theta^{(l)}$. (Actually, $\Theta^{(l)}$ has $s_l + 1$ column, so the dimensionality is not exactly the same).

We can now take all these equations and put them together into a backpropagation algorithm:

5.2.1 Back propagation Algorithm

Given training set $\{(x^{(1)}, y^{(1)}) \cdots (x^{(m)}, y^{(m)})\}$

$$\text{Set } \Delta_{i,j}^{(l)} := 0 \quad \text{for all } (l, i, j)$$

For training example t=1 to m:

- Set $a^{(1)} := x^{(t)}$
- Perform forward propagation to compute $a^{(l)}$ for l=2,3,...,L
- Using $y^{(t)}$, compute $\delta^{(L)} = a^{(L)} - y^{(t)}$
- Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ using $\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)})\ .*\ a^{(l)}\ .*\ (1 - a^{(l)})$
- $\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ or with vectorization,$\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

- $D^{(l)}_{i,j} := \dfrac{1}{m}\left(\Delta^{(l)}_{i,j} + \lambda\Theta^{(l)}_{i,j}\right)$ If $j \neq 0$ **NOTE**: Typo in lecture slide omits outside parentheses. This version is correct.
- $D^{(l)}_{i,j} := \dfrac{1}{m}\Delta^{(l)}_{i,j}$

5.3 Backpropagation Intuition

5.4 The Problem of Overfitting

Consider the problem of predicting y from $x \in R$. The leftmost figure 5.1 shows the result of fitting a $y = \theta_0 + \theta_1 x$ to a dataset. We see that the data doesn't really lie on straight line, and so the fit is not very good.

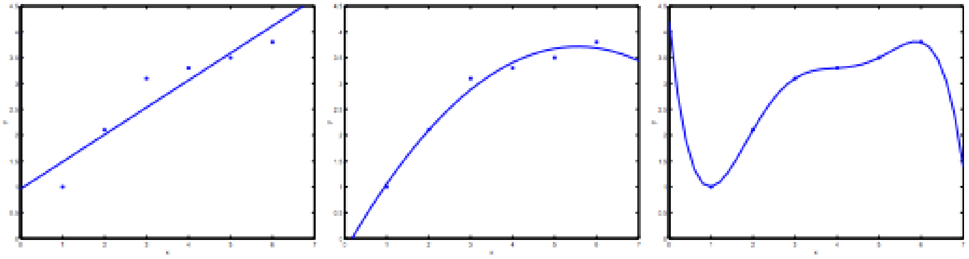

Figure 5.1: The Problem of Overfitting

Instead, if we had added an extra feature x^2, and fit $y = \theta_0 + \theta_1 x + \theta_2 x^2$, then we obtain a slightly better fit to the data (See middle figure). Naively, it might seem that the more features we add, the better. However, there is also a danger in adding too many features: The rightmost figure is the result of fitting a 5^{th} order polynomial $y = \sum_{j=0}^{5} \theta_j$. We see that even though the fitted curve passes through the data perfectly, we would not expect this to be a very good predictor of, say, housing prices (y) for different living areas (x). Without formally defining what these terms mean, we'll say the figure on the left shows an instance of **underfitting**-in which the data clearly shows structure not captured by the model-and the figure on the right is an example of **overfitting**.

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 that is too simple or uses too few features. 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:

- Manually select which features to keep.
- Use a model selection algorithm (studied later in the course).

2) Regularization

- Keep all the features, but reduce the magnitude of parameters θ_j
- Regularization works well when we have a lot of slightly useful features.

Chapter 6

Summary

6.1 Lineal Regression

6.1.1 Hypothesis

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n \tag{6.1}$$

$$h_\theta(x) = \theta^T x \tag{6.2}$$

$$h_\theta(X) = X\theta \tag{6.3}$$

6.1.2 Cost Function

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 \tag{6.4}$$

$$J(\theta) = \frac{1}{2m} (X\theta - \vec{y})^T (X\theta - \vec{y}) \tag{6.5}$$

6.1.3 Gradient Descent

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \tag{6.6}$$

$$\theta := \theta - \frac{\alpha}{m} X^T (X\theta - \vec{y}) \tag{6.7}$$

for j=0, 1,..., n:

6.1.4 Normal Equation

$$\theta = (X^T X)^{-1} X^T y \tag{6.8}$$

6.2 Logistic Regression

6.2.1 Hypothesis

$$0 \leq h_\theta(x) \leq 1 \tag{6.9}$$

"Sigmoid Function," also called the "Logistic Function":

$$h_\theta(x) = g(\theta^T x) \tag{6.10}$$

$$z = \theta^T x \tag{6.11}$$

$$g(z) = \frac{1}{1 + e^{-z}} \tag{6.12}$$

$$h_\theta(x) = P(y = 1|x; \theta) = 1 - P(y = 0|x; \theta) \tag{6.13}$$

$$P(y = 0|x; \theta) + P(y = 1|x; \theta) = 1 \tag{6.14}$$

6.2.2 Cost Function

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))] \tag{6.15}$$

$$h = g(X\theta) \tag{6.16}$$

$$J(\theta) = \frac{1}{m} \cdot \big(-y^T \log(h) - (1 - y)^T \log(1 - h)\big) \tag{6.17}$$

6.2.3 Gradient Descent

$$\theta_j := \theta_j - \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \tag{6.18}$$

$$\theta := \theta - \frac{\alpha}{m}X^T(g(X\theta) - \vec{y}) \tag{6.19}$$