Final Last Updated June 5, 2014

Andrew Ng Coursera ML Course Reading Notes

Ilya Nepomnyashchiy

Due to the volume of notes and my desire to get on to other reading projects, I won't be finishing my transcription of my notes for this class. If you wish to learn the materials from the class, please do check it out on Coursera. It is run quite regularly.

These are notes taken for myself. I have a much higher degree of Math, CS, and ML background than I think is assumed for the course, so I will abbreviate a lot of concepts that are described in much better detail in the lecture for lower level students. I hope that my notes come in handy to someone else, though. As always, I'd be happy to clarify, debate, or correct any observations that I make here if you e-mail me from my website.

I took this course at the end of 2013 and just took forever to actually write up the notes.

Also, due to the Coursera Honor Code, I am barred from making any of my work for the course available publicly. Thus, I will only be posting my notes.

Contents

Lecture 1: What is ML? **1**

Lecture 2: Linear regression of one variable **2**

Lecture 3: Linear algebra review **3**

Lecture 4: Learning with multiple features **4**

Lecture 1: What is ML?

Machine Learning is pervasive in the world: Search engines, photo tagging (e.g. Facebook or Flickr), and spam filters are everyday examples of our use of Machine Learning. We hope to eventually have AI that is as smart as humans, and many hope that learning algorithms will pave the way for this future.

One of the goals of this course is to provide the student with practical knowledge and understanding in addition to the mathematics and theory, because without practical knowledge and implementation details, the theory is pretty much useless. (The practical knowledge is pretty much why I am taking this course, even though I know a lot of the theory already).

In addition to the above problems, ML can be useful for:

- Database mining, where we need to deal with large and unwieldy datasets in many different domains.

- Applications that cannot be done by hand, such as autonomous helicopters and handwriting recognition (I see these as domains where we are limited by the fact that in traditional programming, computers will only do exactly what we tell them to do).
- Self-customizing programs, such as product recommendations
- Understanding human learning

How can we define Machine Learning? Arthur Samuel (1959): Algorithms which give computers the "ability to learn without being explicitly programmed". Samuel wrote a good checkers AI without being able to play checkers well himself. He was able to do this because the computer could take the time to analyze many, many games. Tom Mitchell (1998): "A computer is said to learn from experience **E** with respect to some task **T** and some performance measure **P**, if its performance on **T**, as measured by **P**, improves with experience **E**." (I myself think this is a fine definition, but may be over-formalizing it a bit).

There are two main categories of Machine Learning systems and lots of other broad categories we will be covering:

- Supervised learning
- Unsupervised learning
- Reinforcement learning
- Recommender systems
- Other stuff!

The first two are most common. At this point, Professor Ng once again mentions that we will be learning lots of practical knowledge, and that he has met with people working in industry who have spent months on a problem but getting nowhere due to lacking some key practical insight.

The way **supervised learning** works is that we are given a set of points and the "correct answer" for those points. As an example, consider a situation in which you have the housing prices at sale for a number of houses in your area compared to their square footage and you wish to predict the sale price of a new house. How would you do it? You could fit a line, fit a quadractic function, fit an arbitrary polynomial, etc. We will discuss in future lectures how to choose which function. Such a problem is also called a "regression" problem because we are predicting a continuous-valued output.

Another example of supervised learning is prediction of whether or not a tumor is breast cancer based on the tumor size. Since this is a problem that requires prediction of a discrete-valued output, it is called a "classification" problem. We can think about this problem as separating values on $\mathbb{R}^1$. If we had another feature, it'd be easier to separate the points. In reality, we could have many different features. We will discuss an algorithm that can deal with arbitrary numbers of features.

In **unsupervised learning**, we merely give the algorithm a bunch of data and no labels and ask it to categories. Some examples of this are Google News' grouping of multiple articles into a set of articles about the same topic, taking genetic information about people and clustering them based on the expression of genes, organizing computer clusters, social network analysis, market segmentation, and astronomical data analysis.

One really cool example mentioned and demonstrated in the lecture is the "cocktail party problem", wherein we have k speakers and k microphones, and we wish to separate out a track from each speaker. We can do this with one line of code in MATLAB/Octave, which is sort of cheating because the outer call is `svd` which is a pretty complex thing.

Lecture 2: Linear regression of one variable

Back to the housing prices example, we want to fit e.g. a straight line. More formally: The data set has m samples, xs are the input vectors, and ys are the output or "target" variables. Therefore, (x, y) is a single training example and $(x^{(i)}, y^{(i)})$ is a specific one (the ith). We take the training set and feed it into the learning algorithm, which gives us a function h (which stands for hypothesis). We have h: size of the house $\rightarrow$ the estimated price, or $h \colon x \rightarrow y$.

How do we represent h? In this case, we want to fit a straight line, so we will represent $h_\theta(x) = \theta_0 + \theta_1 x$. Usually we will write $h_\theta(x)$ as $h(x)$ as it will be clear from context. We are starting from linear regression because it is a nice simple building block to start with. The full name of this is linear regression with one variable or univariate linear regression.

Now, we must define a thing called the cost function. The θ_is are parameters of the model, but how do we choose them? We want to choose them so that $h(x)$ is close to the training examples, or more formally to satisfy the following minimum:

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

We will refer to the function as $J(\theta_0, \theta_1)$, and call it our cost function. This particular cost function is called the squared error cost function. We use it here because it works well for most problems, especially regression ones. (As a side note, squared error is differentiable, wherease linear error is not, which becomes important later on).

The lectures go through several examples to help build intuition about the cost function. It's a parabola, and it has one global minimum.

Next, we describe the algorithm we will use. This algorithm is called gradient descent. The general outline is as follows:

- Start with some θ_0, θ_1.
- Keep changing them to reduce J until we hopefully end up at a global minimum.

The way we will change the θ in this case is by repeating until convergence:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

for all j. (At this point, Professor Ng completely glosses over some basic multivariable calculus and this is where I am sad). We must make sure to update the parameters simultaneously (thus, put the new values of the parameters into temp variables until all the calculations are done), or else the algorithm we are doing is not exactly gradient descent and has some wonky properties.

Notes on the learning rate: If α is too small, convergence will take forever. If α is too large, it may overshoot the minimum or fail to converge. Also the gradient gets smaller near a local minimum, so there is no need to change the learning rate over time. Finally, note that some cost functions may have local minima which are not global minima, so gradient descent can land you in the wrong solution. It turns out that the J we defined above is a convex function, so this does not happen.

This technique is also called "batch gradient descent", because all training points are used on each step. It is also possible to not batch, but we will discuss this later. Furthermore, we can use linear algebra to solve for linear regression in particular without an iterative method. However, gradient descent scales better to larger datasets. More extensions involve learning with larger number of features. We will talk about all of these methods later in the class.

It turns out the best notation for much of machine learning comes from the language of linear algebra, which is why lecture 3 is a review of linear algebra.

Lecture 3: Linear algebra review

Didn't watch this because I already know linear algebra and don't have time :P

Lecture 4: Learning with multiple features

Let's go back to our house prices example. Suppose that instead of just square feet of the house as our feature, we had square feet, number of bedrooms, number of floors, and age of the house. We will use x_1, x_2, x_3, and x_4 to denote these four features, and continue to use y to denote the target variables. As some further notation, we will use n to be the number of features, $x^(i)$ to be the input features of the ith training example, and $x_j^(i)$ to be the value of feature j in the ith training example.

Now, instead of representing $h_\theta(x)$ as $\theta_0 + \theta_1 x$, we have $h_\theta(\vec{x}) = \theta_0 + \theta_1 x_2 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4$. In general, $h_\theta(\vec{x}) = \theta_0 + \theta_1 x_1 + \ldots + \theta_n x_n$. For convenience, we define $x_0 = 1$ so that $\vec{x} \in \mathbb{R}^{n+1}$ and the parameters are a vector $\theta \in \mathbb{R}^{n+1}$. We can write the general form of our hypothesis as $h_\theta(\vec{x}) = \theta^T \vec{x}$. I think for my own convenience for typing this, I'll drop the vector notation and input vectors will just be assumed to be vectors.

Next, we look at gradient descent on multiple variables. We have the same cost function, but now J depends on all of the new θ parameters, which we think of as a vector. Thus:

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

is our new update rule, and we once again simultaneously update.

In practice, we want to do a couple of things to our data to make sure gradient descent works as well as it should. The first thing is called feature scaling. The idea here is to make sure that the various features in the feature vector are on the same scale, so that one of the features doesn't dominate the gradient descent. For example, if x_1 is your house's size, which goes on a scale of 0-2000 square feet, and x_2 is your house's number of bedrooms, which goes on a scale of 1-5 bedrooms, then the contours will be stretched out in one direction and the descent will take a while to converge (it will oscillate in one direction, similarly to when α is big).

The solution to this problem is to divide each feature by its max value and subtract a value such that each feature is approximately in $[-1, 1]$. It's okay if this isn't the exact interval, as long as it's on the same scale.

Another thing we might want to do to our data is called mean normalization. In this process, we try to make our variables have approximately zero mean (obviously, this shouldn't apply to $x_0 = 1$). This can be done merely by subtracting the mean. For example, in our house example, we might do $x_1 = \frac{\text{size}-1000}{2000}$.

For "Gradient Descent in practice II", Professor Ng discusses debugging and choosing the most effective learning rate.

Debugging: A very useful thing to do when your learning process doesn't seem to be working is to plot the cost function over the number of iterations. The cost should be going down after every iteration. This curve will also tell you the number of iterations after which the curve flattens out and you get diminishing returns (note that this number depends on the application and can vary a great deal). Instead of looking at the graph, one can also come up with automated convergence tests, e.g. stop when $J(\theta)$ decreases by less than a small value, say 10^{-3} in one iteration. Professor Ng points out that he finds it hard to pick that small value and that looking at the plot is often better.

More importantly, this curve can highlight problems in your learning method. If $J(\theta)$ is increasing, then α is too big, and the descent is overshooting the minimum. If J is oscillating, it likely means the same. Although it isn't proven in the lecture, it is the case that if α is small enough, $J(\theta)$ should decrease on every iteration (assuming differentiable cost function!), but having too small of an α will cause the descent to take too long to converge. In some cases, an α that is too big will cause slow convergence as well, but that's uncommon.

Picking α: