# Machine Learning Day 1

Nikhil Shenoy

September 22, 2015

## 1 Installation

Before we do anything, we need to do all of our installs. We'll be doing all of our work in Python, and the best way to get the packages we need would be to install Anaconda. This powerful Python package includes NumPy, Pandas, Matplotlib, Sci-kit Learn, and iPython, which are the main packages that we'll be using. If you don't want to install all of Anaconda, you'll just have to make sure that you've installed these packages on your own. Here's how to get the environment set up:

1. Follow the instructions for installing Anaconda on your particular system here:
   http://docs.continuum.io/anaconda/install#windows-install
2. Open the iPython interpreter by typing ïpython notebookïnto the command line. It should open directly in your web browser.
3. There should be a button on the right side of the screen that says "New Notebook". Click on it
4. Click on the first cell and type the following code. If there are no errors, then you're all set!

```
%matplotlib inline
import numpy as np
from sklearn import linear_model
import matplotlib.pyplot as plt
from scipy import optimize
```

## 2 Theory: Linear Regression

### 2.1 The Hypothesis Function

Linear Regression is one of the basic algorithms of Machine Learning. You have most likely seen it already in math courses you may have taken; approximations using a tangent line and linear approximations are very similar concepts. Given a number of data points in the form $(x, y)$, we want to find a linear function that best approximates all of our data points. So, we define a **hypothesis function**:

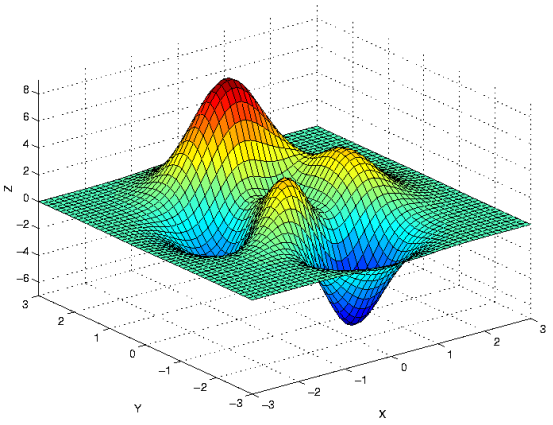


Figure 1: A basic 3D plot

$$h_\theta(x) = \theta_0 + \theta_1 x$$

Our goal is to find constants $\theta_0, \theta_1$ such that our hypothesis function fits our data as closely as possible. How do we do this? We define a **cost function**:

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

This cost function is called the **Mean Squared Error** function. As the name implies, we find the sum of the squares of the errors. We divide the equation by two as a convenience during the gradient descent algorithm. Now that we have defined our cost function, we can quantitatively determine how well our hypothesis function is doing.

## 2.2 Gradient Descent

To find the best selections for $\theta_0$ and $\theta_1$, we need to minimize $J(\theta_0, \theta_1)$. This will give us the best constants such that the cost of $h_\theta$ is the least. So how do we do that? Imagine a 3-D plot with $\theta_0$ on the $x$-axis and $\theta_1$ on the $y$-axis. We plot $J(\theta_0, \theta_1)$ on the $z$-axis. The plot will look something like Figure 1.

We want to find the global minimum of this plot. In order get a hint as to where the minimum is, we take the negative gradient f the function at the current point. The positive gradient gives us the vector that points in the direction of steepest increase; conversely, the negative gradient gives us the vector pointing in the direction of steepest descent. By finding this vector at points on the function and following the direction it points to, we will eventually reach a minimum. Here is an outline of the gradient descent algorithm:

1. Pick any starting point on the plot
2. Find the negative gradient of the cost function at that point
3. Move a specified distance in the direction of the negative gradient.
4. Repeat until we can move no lower.

The gradient is expressed as:

$$\nabla J = \begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \\ \frac{\partial J}{\partial \theta_1} \end{bmatrix} \tag{1}$$

We can also define our current and next values of $\theta_0$ and $\theta_1$ using the vector $\theta$. Mathematically, we can express the entire algorithm as:

$$\theta = \theta - \alpha \nabla J$$

We continue to iterate using this equation until $\theta$ converges on a single vector. The constant $\alpha$ is a scaling factor called the **learning rate**. The learning rate scales the gradient by some constant; we can pre-set $\alpha$ so that our algorithm converges at a particular rate.

# 3 Generalized Linear Regression

We want to be able to apply linear regression to problems of higher dimensions, so now we'll generalize the two-variable case we covered before. We'll redefine our hypothesis function as follows:

$$h_\theta(x) = \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

Using vector notation, we can write this as:

$$h_\theta(x) = \begin{pmatrix} \theta_1 & \theta_2 \cdots \theta_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \theta^T x$$

If we have multiple training examples, we can create a larger matrix $X$:

$$X = \begin{pmatrix} x_0^{(1)} & x_1^{(1)} & \cdots x_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_0^{(m)} & x_1^{(m)} & \cdots x_n^{(m)} \end{pmatrix}$$

To create a new hypothesis with $n$ **features** and $m$ **training examples**, we replace the column vector $x$ from the previous equation with our new $X$ matrix. Now we have:

$$h_\theta(X) = \begin{pmatrix} \theta_1 & \theta_2 \cdots \theta_n \end{pmatrix} \begin{pmatrix} x_0^{(1)} & x_1^{(1)} & \cdots x_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_0^{(m)} & x_1^{(m)} & \cdots x_n^{(m)} \end{pmatrix} = X\theta$$

Since our notation is now vectorized, we can adjust our cost function to be vectorized as well. We now have:

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

By taking the gradient of $J(\theta)$ and performing some algebraic magic, we get the following:

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

Finally, we can use a variant on the iteration equation we saw before in order to get our answer:

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