# Machine Learning

Herrick Fang

## Contents



# 1 Machine Learning vs Statistical Learning

Machine Learning: applies more toward large scale applications focusing on prediction and accuracy
Statistical Learning: focuses more on models, interpretation, precision, and uncertainty

# 2 Supervised Learning

The algorithms receive "Right Answers."

## 2.1 Linear Regression

Linear regressions is one of the most fundamental tools for machine learning. It predicts real continuous valued output in a finite ordered set (e.g. survived, digit 0-9, class).

### 2.1.1 Model and Cost Function

The goal of this model is to minimize $\theta_i's$, the **parameters**, such that the hypothesis function,

$$h(x) = \sum_{i=0}^{n} \theta_i x_i = \theta^T x \tag{1}$$

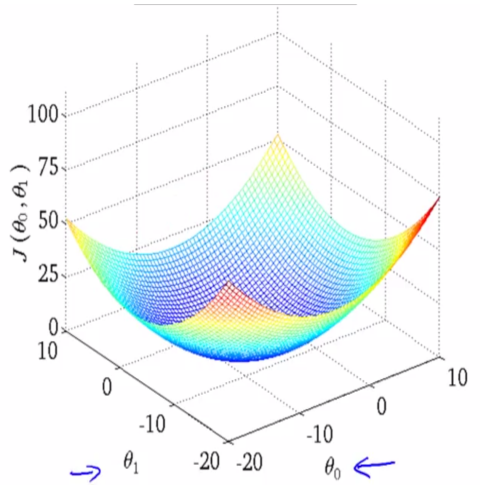
is close to $y$ given the input variables $x's$, so that our set of $m$ number of training examples $\left(x^{(i)}, y^{(i)}\right)$, will be minimized. In multivariate linear regression, $n$ denotes the number of features and $x^{(i)}$

denotes the row of the training set. $x_j^{(i)}$ denotes the output element within the row of the training set. Think of $\theta$ as a $\mathbb{R}^{n+1}$ vector and the same with x.

**Cost Function** The squared error function is a reasonable choice in minimizing the hypothesis function,

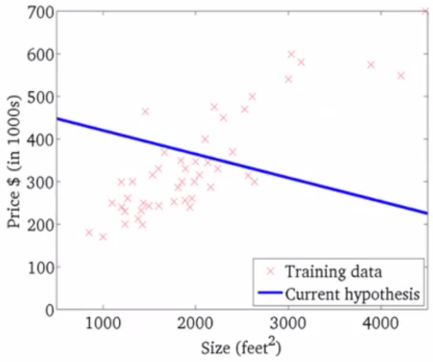
$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2. \tag{2}$$

where we aim to minimize $\theta_0$ and $\theta_1$.
A way to visualize the cost function is a bow-shaped function or a contour plot.



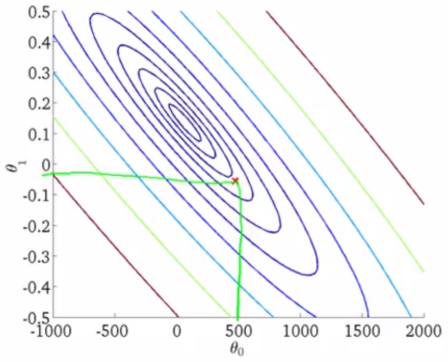

$h_\theta(x)$

(for fixed $\theta_0, \theta_1$, this is a function of x)




$J(\theta_0, \theta_1)$

(function of the parameters $\theta_0, \theta_1$)




**Gradient Descent** The algorithm starts with a set $\theta_0$ and $\theta_1$ then these values keep changing to reduce $J(\theta_0, \theta_1)$ until we end up at a minimum. Essentially, we try to find a way to walk down a hill in the fastest way possible by looking around and choosing the farthest step.

The **gradient descent** algorithm starts with some initial $\theta$, and repeats

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta). \tag{3}$$

where $\alpha$ is the **learning rate** until convergence.

To implement this algorithm correctly, we must simultaneously update by assigning $ALL$ dummy variables to the values of $\theta_j$ before assigning the new $\theta_j$ to the dummy variables.

Let's work out one update of one training example $(x, y)$. We have

$$\begin{aligned}
\frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \frac{1}{2(1)} (h_\theta(x) - y)^2 \\
&= 2 \cdot \frac{1}{2} (h_\theta(x) - y) \cdot \frac{\partial}{\partial \theta_j} (h_\theta(x) - y) \\
&= (h_\theta(x) - y) \cdot \frac{\partial}{\partial \theta_j} \left( \sum_{i=0}^{n} \theta_i x_i - y \right) \\
&= (h_\theta(x) - y) x_j
\end{aligned}$$

Some intuition includes knowing that a positive derivative leads to a decrease in $\theta_j$ and vice versa.

This essentially means that if $\alpha$ is too small, gradient descent can be slow and if $\alpha$ is too large, gradient descent can overshoot the minimum, which means that it fails to converge.

**Batch Gradient Descent** looks at every example in the entire training set on every step.

**repeat**
  $\theta_j \leftarrow \theta_j + \alpha \sum_{i=1}^{m} \left( y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)}$
**until** convergence

**Stochastic Gradient Descent** updates the parameters according to the gradient of the error with respect to a single training example at a time.

**loop**
  **for** $i = 1$ to $m$ **do**
    $\theta_j \leftarrow \theta_j + \alpha \left( y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)}$
  **end for**
**end loop**

Linear regression automatically uses a convex function, which means that we have a bowl shaped curve, so we will find the global minimum of the function. In contrast, other functions that are not convex may fail to find the global minimum. A local optimum may be found instead.

The learning rate $\alpha$ cannot be too large or else the function may fail to converge. If it is too small, gradient descent will be too slow.

**Feature Scaling** To make gradient descent run faster, we generally want our x values to range from -1 to 1. Visually, our contour plot will be more circular if we apply this method. To do so, we

can make our features, x, take on smaller values by dividing all of them by the range S:

$$\frac{x^{(i)}}{S}$$

To further improve this scaling, we use what is called **mean normalization** which subtracts the feature, x, by the mean value, $\mu$

$$\frac{x^{(i)} - \mu}{S}$$

to get an average of zero.

**$\alpha$ and debugging** In Gradient Descent, the function J($\theta$) should be decreasing exponentially like a $\frac{1}{x}$ graph where $x > 0$ if we want an optimal algorithm.

Otherwise, we see that the the learning rate $\alpha$

- too large if the graph is increasing = overshooting
- too small if the graph is decreasing, but the trend is closer to a linear graph

Thus, a good way to test for the right $\alpha$ is to multiply by a factor of 10 until we see a good looking graph.

Automatic convergence also works if the change in each iterative step cause is less than some $\epsilon$, but finding such a value is too tedious for each set of data.

**Features and Polynomial Regression** It is possible to organize data in such a way to fit a function by adding more variables and giving them larger powers of n. Thus, feature scaling must scale to those factors. Knowing the basic structure of some graphs will be helpful in determing what can be used to optimize the curve fitting.

**Normal Equation** This algorithm finds a method to solve for $\theta$ analytically instead of iteratively like **gradient descent**. Basically, you set all the partial derivatives of the one-dimenisonal vector $\theta \in \mathbb{R}$ to 0. This will obtain all the necessary values of $\theta$ such that we find a good fitting curve.

$$\theta = \left(X^T X\right)^{-1} X^T y$$

### 2.1.2 Comparison

For $m$ training examples and $n$ features, we see

| Gradient Descent | Normal Equation |
|---|---|
| Need to choose $\alpha$ | No need to choose $\alpha$ |
| Needs many iterations | Doesn't need to iterate |
| Works well even when $n$ is large | Need to compute $\left(X^T X\right)^{-1}$ |
| | Slow if $n$ is large ($> 10000$) |

## 2.2 Classification

Discrete/Quantitative Outputs Only (e.g. price, blood pressure)

# 3 Unsupervised Learning

We are not telling the algorithm the right answer, so the computer finds its own patterns.

# 4 Linear Algebra

Matrices are two-dimensional arrays such as

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

The above matrix is a 4 x 3 matrix. For matrices, we classify them as an $m$ x $n$ matrix (rows x columns).

A **vector** is a $m$ x 1 matrix.

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

In math, the matrix is 1-indexed, but in programming, it is 0-indexed.

**Adding**

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

**Multiplying** Multiplying matrices by a scalar is easy:

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

Multiplying matrices by a vector involves having the same column of your data matrix and row of your vector:

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

An m x n matrix multiplied by an n x o matrix results in an m x o matrix.

Multiplying matrices can be thought of separating each column in the second matrix into multiple vectors.

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

**Properties** Commutativity doesn't work:

$$A \cdot B \neq B \cdot A$$

Associativity does work:

$$A \cdot (B \cdot C) = (A \cdot B) \cdot C$$

Identity Matrices have 1s across the diagonal and 0s elsewhere:

$$A \cdot I = I \cdot A = A$$

Inverse Matrices follow

$$A \cdot A^{-1} = A^{-1} \cdot A = I$$

if there is an inverse matrix. Singular/ Degenerate matrices do not have an inverse matrix.

A matrix transpose is essentially switching rows and columns:

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

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

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