Stanford Machine Learning - Week I

Eric N Johnson

July 18, 2016

1 Supervised Learning

"Right answers" are given to the computer to guess what other answers are

- Linear and Quadratic Regression
- Logistic Regression
- Classification (in as many dimensions as needed)

2 Unsupervised Learning

"Right answers" aren't given already. Instead the ML algorithm looks for structure.

- Clustering algorithms

  Organizing computer clusters
  Social networking analysis
  Market segmentation
  Astronomical data analysis

- The 'cocktail party' problem

  i Two microphones are used to record speakers in a room
  ii The algorithm is able to split out each speaker in spite of overlapping signals, noise
  iii The algorithm is very short and uses Singular Value Decomposition

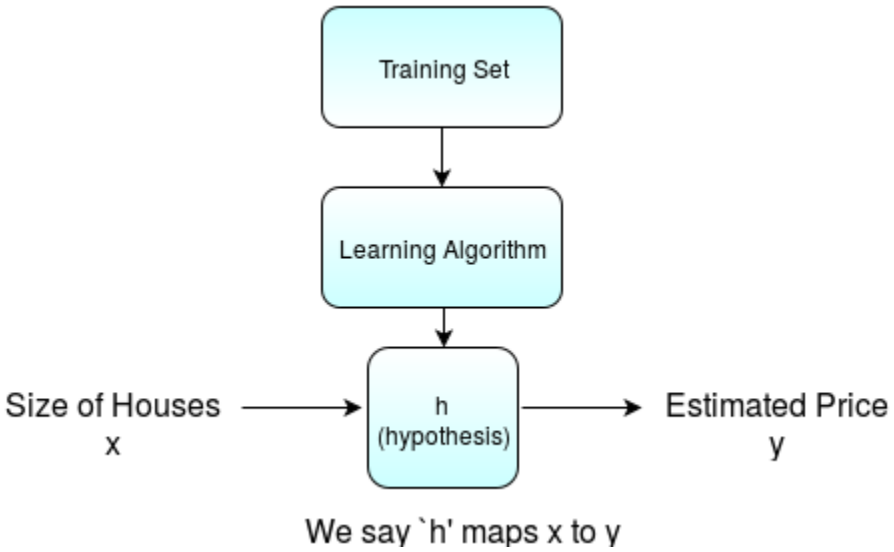
3 Model and Cost Function (linear regression)

For example, we have x = size and y = price for homes or whatever. This is a supervised learning problem since we know *price* for our dataset. We predict real-valued output. Contrast this with classification where we predict what set something belongs in which is 'discrete output.'

Variables used throughout this course:

- $m \equiv$ Number of training examples
- $x \equiv$ input variable or features
- $y \equiv$ output or target variable
- $(x, y) \equiv$ one training example
- $(x^{(i)}, y^{(i)}) \equiv i^{th}$ example

Here is a diagram for this kind of model:



Note: using *hypothesis* may not be the best word – but it is what people say in practice. We write univariate linear regression by writing:

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

or in shorthand, $h(x)$. The *parameters* of the model are θ_0 and θ_1.

In linear regression we have a training set used to find our parameters. We choose θ_0 and θ_1 so that $h_\theta(x)$ is close to y for our training examples (x, y).

This is an optimization problem. We minimize θ_0 and θ_1:

$$\min(\theta_0, \theta_1) \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

which is equivalent to minimizing a cost function $J(\theta_0, \theta_1)$:

$$\min(J(\theta_0, \theta_1)) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

This is sometimes called a *squared error function*.

Minimizing $J(\theta_0, \theta_1)$ involves using the parameters θ_0 and θ_1 as function inputs. Below is an example.

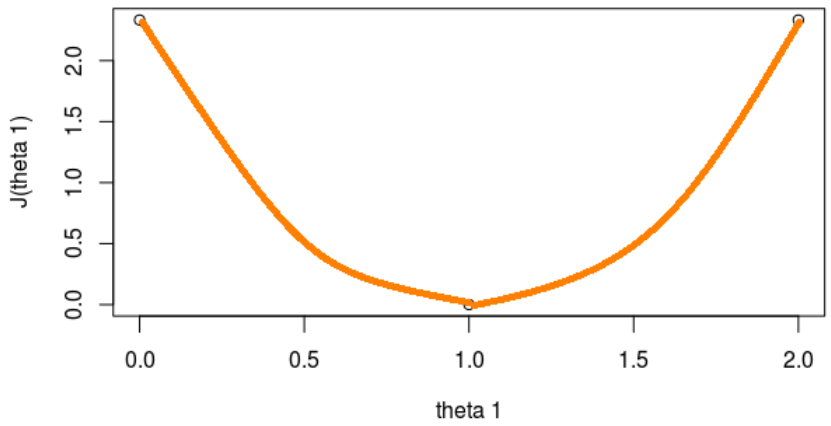
Suppose, for example that we have this:

x	y
1	1
2	2
3	3

And suppose we let $h_\theta(x) = \theta_1 x$ (the y-int is zero). Then we calculate $J(\theta_1)$:

$$\begin{aligned}
J(\theta_1 = 0) &= \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)} - y^{(i)})\right)^2 \\
&= \frac{1}{6}\left[(-1)^2 + (-2)^2 + (-3)^2\right] \\
&= \frac{14}{6} \\
J(\theta_1 = 1) &= \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)} - y^{(i)})\right)^2 \\
&= \frac{1}{6}\left[(0)^2 + (0)^2 + (0)^2\right] \\
&= 0 \\
J(\theta_1 = 2) &= \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)} - y^{(i)})\right)^2 \\
&= \frac{1}{6}\left[(1)^2 + (2)^2 + (3)^2\right] \\
&= \frac{14}{6}
\end{aligned}$$

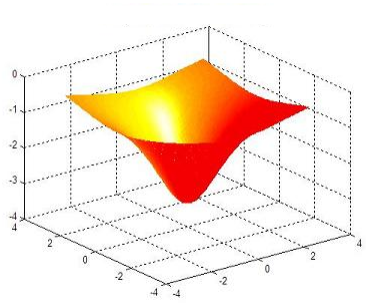
The result is clear when we plot $J(\theta_1)$:



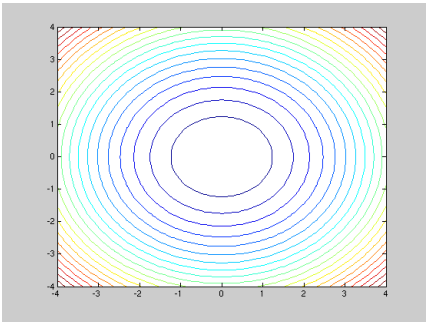
Obviously I didn't want to do a bunch of trivial algebra here but that's the idea. $\theta_1 = 1$ minimizes the cost function $J(\theta_1)$. If we have several parameters, say $\theta_0, \theta_1, ..., \theta_n$, we need a more sophisticated algorithm.

4 Gradient Descent

If we keep the parameters θ_0 and θ_1, we are going to minimize $J(\theta_0, \theta_1)$ over a plane. We get something that looks like a surface plot:



But in general we will represent these as contour plots.



With several parameters $\theta_0, \theta_1, ..., \theta_n$, the data becomes much harder to visualize.

This is an algorithm to minimize a cost function over some parameters. With some cost function $J(\theta_0, \theta_1, ..., \theta_n)$, we want to minimize

$$\min(\theta_0, \theta_1, ..., \theta_n) J(\theta_0, \theta_1, ..., \theta_n)$$

Recall from graduate school that Gradient Descent is guaranteed to find local minimums assuming the problem has certain characteristics.

Here is the Gradient Descent algorithm:

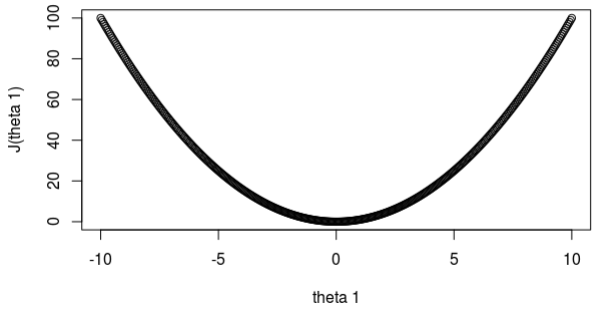
$$\begin{aligned} &\text{Repeat until convergence } \{ \\ &\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1, ...) \\ &\} \end{aligned}$$

For a problem with only θ_0 and θ_1, we correct and update:

$$\begin{aligned} temp0 &:= \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) \\ temp1 &:= \theta_1 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) \\ \theta_0 &:= temp0 \\ \theta_1 &:= temp1 \end{aligned}$$

We *simultaneously* update θ_0 and θ_1 just like it is written above. Doing a simultaneous update is important! Note that $:=$ denotes *assignment*. I.e., $a := b$ means take the value of b and make a that value. Contrast that with $=$ which is just an assertion that $a = b$. The parameter α is called the *learning rate*. α determines the step size for each step of the algorithm.

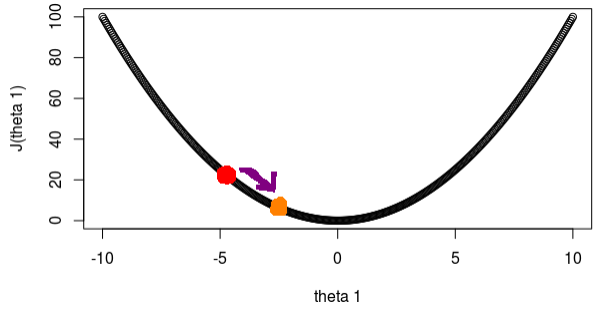
Take for example, this plot of $J(\theta_0)$:



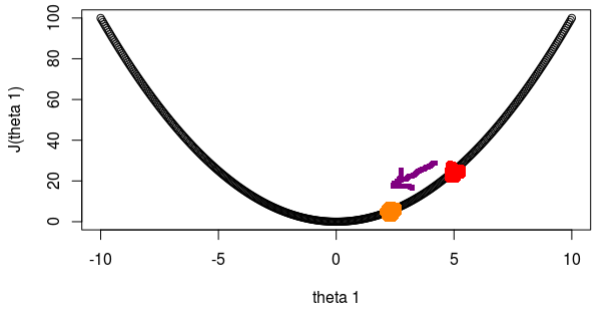
Around $\theta_1 = -5$, we will update θ_1 according to

$$temp0 := \theta_0 - \alpha \frac{d}{d\theta_0} J(\theta_0, \theta_1)$$

Since $\frac{d}{d\theta_1} J(\theta_1) < 0$, we will update by *increasing* the value of θ_1. Thus we *decrease* the value of $J(\theta_1)$.



Around $\theta_1 = 5$, however, $\frac{d}{d\theta_1} J(\theta_1) > 0$ and so we update by *decreasing* the value of θ_1. This *decreases* the value of $J(\theta_1)$.



The parameter α just specifies how large a jump we take. We want α to be neither too large nor too small. A very small α may require too many calculations to get J to a global minimum. An α that is too large may overshoot our desired minimum or even diverge. If θ_1 is already at a *local minimum*, the algorithm will not move θ_1. We cannot guarantee that Gradient Descent will find a global minimum - but changes to α may assist us. Note that as Gradient Descent moves toward a local minimum, the step sizes automatically decrease even as α remains fixed.

How can you find the partial derivatives?

We have two cases. When $j = 0$ and when $j = 1$.

$$\begin{aligned}\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) &= \frac{\partial}{\partial \theta_j} \left[\frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)} - y^{(i)} \right)^2 \right] \\ &= \frac{\partial}{\partial \theta_j} \left[\frac{1}{2m} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 x^{(i)} - y^{(i)} \right)^2 \right]\end{aligned}$$

The expression for θ_0 (where $j = 0$) simplifies to:

$$\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 x^{(i)} - y^{(i)} \right)$$

and the expression for θ_1 (where $j = 1$) simplifies to:

$$\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 x^{(i)} - y^{(i)} \right) x^{(i)}$$

That second $x^{(i)}$ in the derivative for the $j = 1$ case is from *chain rule*.
Thus for linear regression, the Gradient Descent algorithm is:

Repeat until convergence {

$$temp0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 x^{(i)} - y^{(i)} \right)$$

$$temp1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 x^{(i)} - y^{(i)} \right) x^{(i)}$$

$$\theta_0 := temp0$$

$$\theta_1 := temp1$$

}

We know that we can end up in a *local minimum* with Gradient Descent. However, we will **always** get a *global minimum* when we use Gradient Descent for linear regression because our cost function $J(\theta_0, \theta_1)$ is **convex**.

4.1 Batch Gradient Descent

"Batch" means in ML context that each step of Gradient Descent uses all of the training examples. We will hit other versions of Gradient Descent later.

5 Linear Algebra Review (With Octave)

5.1 Matrices

The matrix

$$A = \begin{bmatrix} 1 & 5 & 1 \\ 3 & 6 & 2 \\ 1 & 1 & 1 \end{bmatrix}$$

can be defined in Octave using the following commands:

```
A = [1, 5, 1; 3, 6, 2; 1, 1, 1]
```

which sets

```
A =
   1   5   1
   3   6   2
   1   1   1
```

5.2 Vectors

These are defined similarly. $x = 1, 2, 1$ can be created using

```
x = [1,2,1]
```

which sets

```
x =
   1   2   1
```

5.3 Basic Linear Algebraic Manipulation

We may refer to specific elements of the matrix. I.e., $A_{i,j}$ in Octave using

```
A(1,2)
```

which returns 5.
We may *transpose* a matrix (or vector) by using

```
At = transpose(A)
```

which returns

```
At =
   1   3   1
   5   6   1
   1   2   1
```

Similarly, we may find the *conjugate transpose* A^* by using

```
Ac = ctranspose(A)
```

Which is kind of boring in this case.

Addition, subtraction, and multiplying matrices by constants are straightforward.

Multiplication is defined sanely (it isn't element-by-element like in some languages)

```
x * A
```

returns

```
ans =
    8   18    6
```

A matrix inverse can be calculated easily using

```
Ainv = inverse(A)
```

and so we get the expected result

```
>> Ainv * A
ans =
   1   0   0
   0   1   0
   0   0   1
```

We may create an identity matrix using

```
>> eye(4)
ans =
Diagonal Matrix

   1   0   0   0
   0   1   0   0
   0   0   1   0
   0   0   0   1
```

and an array of all 1's using

```
>> ones(4)
ans =

   1   1   1   1
   1   1   1   1
   1   1   1   1
   1   1   1   1
```