CS229 Lecture notes

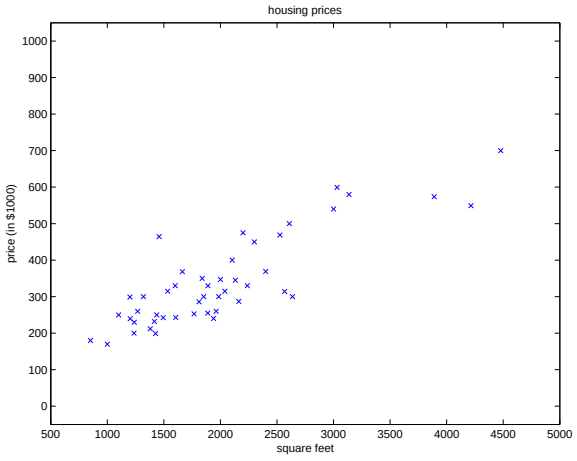
Andrew Ng

Supervised learning

Lets start by talking about a few examples of supervised learning problems. Suppose we have a dataset giving the living areas and prices of 47 houses from Portland, Oregon:

Living area (feet^2)	Price (1000$s)
2104	400
1600	330
2400	369
1416	232
3000	540
⋮	⋮

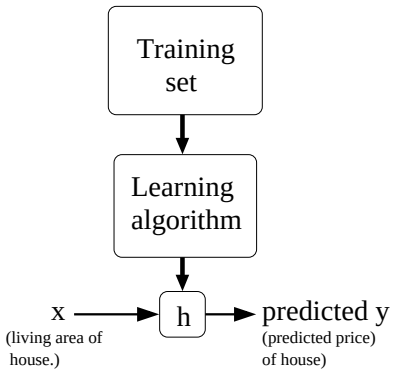
We can plot this data:



Given data like this, how can we learn to predict the prices of other houses in Portland, as a function of the size of their living areas?

To establish notation for future use, we'll use $x^{(i)}$ to denote the "input" variables (living area in this example), also called input **features**, and $y^{(i)}$ to denote the "output" or **target** variable that we are trying to predict (price). A pair $(x^{(i)}, y^{(i)})$ is called a **training example**, and the dataset that we'll be using to learn—a list of m training examples $\{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$—is called a **training set**. Note that the superscript "(i)" in the notation is simply an index into the training set, and has nothing to do with exponentiation. We will also use $\mathcal{X}$ denote the space of input values, and $\mathcal{Y}$ the space of output values. In this example, $\mathcal{X} = \mathcal{Y} = \mathbb{R}$.

To describe the supervised learning problem slightly more formally, our goal is, given a training set, to learn a function $h : \mathcal{X} \mapsto \mathcal{Y}$ so that $h(x)$ is a "good" predictor for the corresponding value of y. For historical reasons, this function h is called a **hypothesis**. Seen pictorially, the process is therefore like this:



When the target variable that we're trying to predict is continuous, such as in our housing example, we call the learning problem a **regression** problem. When y can take on only a small number of discrete values (such as if, given the living area, we wanted to predict if a dwelling is a house or an apartment, say), we call it a **classification** problem.

Part I
Linear Regression

To make our housing example more interesting, lets consider a slightly richer dataset in which we also know the number of bedrooms in each house:

Living area (feet^2)	#bedrooms	Price (1000$s)
2104	3	400
1600	3	330
2400	3	369
1416	2	232
3000	4	540
$\vdots$	$\vdots$	$\vdots$

Here, the x's are two-dimensional vectors in $\mathbb{R}^2$. For instance, $x_1^{(i)}$ is the living area of the i-th house in the training set, and $x_2^{(i)}$ is its number of bedrooms. (In general, when designing a learning problem, it will be up to you to decide what features to choose, so if you are out in Portland gathering housing data, you might also decide to include other features such as whether each house has a fireplace, the number of bathrooms, and so on. We'll say more about feature selection later, but for now lets take the features as given.)

To perform supervised learning, we must decide how we're going to represent functions/hypotheses h in a computer. As an initial choice, lets say we decide to approximate y as a linear function of x:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

Here, the θ_i's are the **parameters** (also called **weights**) parameterizing the space of linear functions mapping from $\mathcal{X}$ to $\mathcal{Y}$. When there is no risk of confusion, we will drop the θ subscript in $h_\theta(x)$, and write it more simply as $h(x)$. To simplify our notation, we also introduce the convention of letting $x_0 = 1$ (this is the **intercept term**), so that

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

where on the right-hand side above we are viewing θ and x both as vectors, and here n is the number of input variables (not counting x_0).

Now, given a training set, how do we pick, or learn, the parameters θ? One reasonable method seems to be to make $h(x)$ close to y, at least for

the training examples we have. To formalize this, we will define a function that measures, for each value of the θ's, how close the $h(x^{(i)})$'s are to the corresponding $y^{(i)}$'s. We define the **cost function**:

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

If you've seen linear regression before, you may recognize this as the familiar least-squares cost function that gives rise to the **ordinary least squares** regression model. Whether or not you have seen it previously, lets keep going, and we'll eventually show this to be a special case of a much broader family of algorithms.

1 LMS algorithm

We want to choose θ so as to minimize $J(\theta)$. To do so, lets use a search algorithm that starts with some "initial guess" for θ, and that repeatedly changes θ to make $J(\theta)$ smaller, until hopefully we converge to a value of θ that minimizes $J(\theta)$. Specifically, lets consider the **gradient descent** algorithm, which starts with some initial θ, and repeatedly performs the update:

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

(This update is simultaneously performed for all values of $j = 0, \ldots, n$.) Here, α is called the **learning rate**. This is a very natural algorithm that repeatedly takes a step in the direction of steepest decrease of J.

In order to implement this algorithm, we have to work out what is the partial derivative term on the right hand side. Lets first work it out for the case of if we have only one training example (x, y), so that we can neglect the sum in the definition of J. We have:

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

For a single training example, this gives the update rule:[1]

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)}.$$

The rule is called the **LMS** update rule (LMS stands for "least mean squares"), and is also known as the **Widrow-Hoff** learning rule. This rule has several properties that seem natural and intuitive. For instance, the magnitude of the update is proportional to the **error** term $(y^{(i)} - h_\theta(x^{(i)}))$; thus, for instance, if we are encountering a training example on which our prediction nearly matches the actual value of $y^{(i)}$, then we find that there is little need to change the parameters; in contrast, a larger change to the parameters will be made if our prediction $h_\theta(x^{(i)})$ has a large error (i.e., if it is very far from $y^{(i)}$).

We'd derived the LMS rule for when there was only a single training example. There are two ways to modify this method for a training set of more than one example. The first is replace it with the following algorithm:

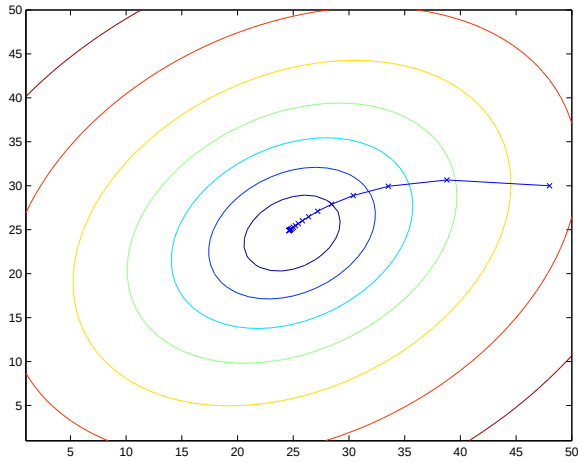
Repeat until convergence {

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)} \qquad \text{(for every } j\text{)}.$$

}

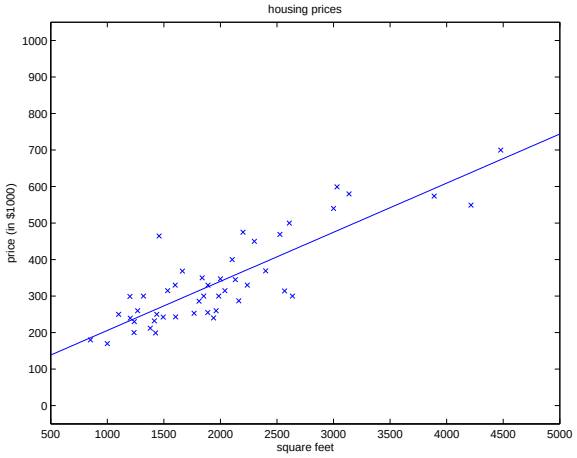
The reader can easily verify that the quantity in the summation in the update rule above is just $\partial J(\theta)/\partial \theta_j$ (for the original definition of J). So, this is simply gradient descent on the original cost function J. This method looks at every example in the entire training set on every step, and is called **batch gradient descent**. Note that, while gradient descent can be susceptible to local minima in general, the optimization problem we have posed here for linear regression has only one global, and no other local, optima; thus gradient descent always converges (assuming the learning rate α is not too large) to the global minimum. Indeed, J is a convex quadratic function. Here is an example of gradient descent as it is run to minimize a quadratic function.

[1] We use the notation "$a := b$" to denote an operation (in a computer program) in which we *set* the value of a variable a to be equal to the value of b. In other words, this operation overwrites a with the value of b. In contrast, we will write "$a = b$" when we are asserting a statement of fact, that the value of a is equal to the value of b.



The ellipses shown above are the contours of a quadratic function. Also shown is the trajectory taken by gradient descent, with was initialized at (48,30). The x's in the figure (joined by straight lines) mark the successive values of θ that gradient descent went through.

When we run batch gradient descent to fit θ on our previous dataset, to learn to predict housing price as a function of living area, we obtain $\theta_0 = 71.27$, $\theta_1 = 0.1345$. If we plot $h_\theta(x)$ as a function of x (area), along with the training data, we obtain the following figure:



If the number of bedrooms were included as one of the input features as well, we get $\theta_0 = 89.60, \theta_1 = 0.1392$, $\theta_2 = -8.738$.

The above results were obtained with batch gradient descent. There is an alternative to batch gradient descent that also works very well. Consider the following algorithm:

Loop {

for i=1 to m, {

$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)}$ (for every j).

}

}

In this algorithm, we repeatedly run through the training set, and each time we encounter a training example, we update the parameters according to the gradient of the error with respect to that single training example only. This algorithm is called **stochastic gradient descent** (also **incremental gradient descent**). Whereas batch gradient descent has to scan through the entire training set before taking a single step—a costly operation if m is large—stochastic gradient descent can start making progress right away, and continues to make progress with each example it looks at. Often, stochastic gradient descent gets θ "close" to the minimum much faster than batch gradient descent. (Note however that it may never "converge" to the minimum, and the parameters θ will keep oscillating around the minimum of $J(\theta)$; but in practice most of the values near the minimum will be reasonably good approximations to the true minimum.[2]) For these reasons, particularly when the training set is large, stochastic gradient descent is often preferred over batch gradient descent.

2 The normal equations

Gradient descent gives one way of minimizing J. Lets discuss a second way of doing so, this time performing the minimization explicitly and without resorting to an iterative algorithm. In this method, we will minimize J by explicitly taking its derivatives with respect to the θ_j's, and setting them to zero. To enable us to do this without having to write reams of algebra and pages full of matrices of derivatives, lets introduce some notation for doing calculus with matrices.

[2]While it is more common to run stochastic gradient descent as we have described it and with a fixed learning rate α, by slowly letting the learning rate α decrease to zero as the algorithm runs, it is also possible to ensure that the parameters will converge to the global minimum rather then merely oscillate around the minimum.