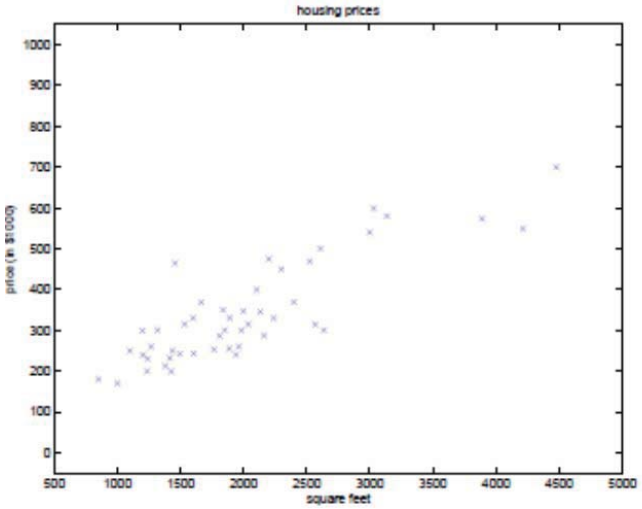
Linear Regression

Machine learning for house hunting

- Suppose we have a dataset giving the living areas and prices of some houses :

Living area (feet^2)	Price(1000$s)
2104	400
1600	330
2400	369
1416	232
3000	540
…	
2005	?
3200	?
1280	?

We can plot this data set:



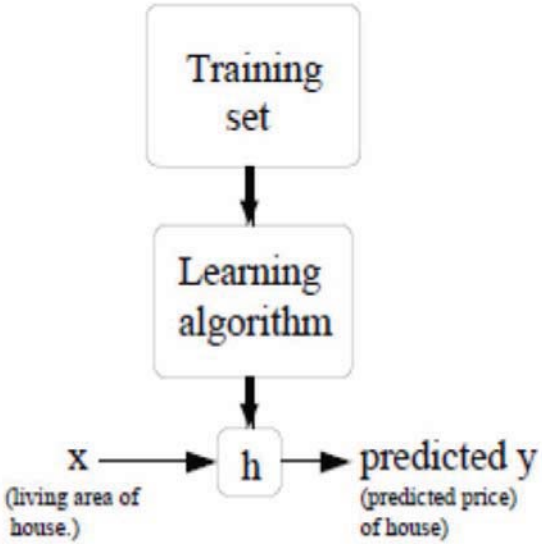
- How can we learn to predict the prices of other houses, as a function of the size of their living areas?

The learning problem

- $x^{(i)}$ denotes the "input" variables/features
- $y^{(i)}$ denotes the "output" or target variable that we are trying to predict(price)
- A pair $(x^{(i)}, y^{(i)})$ is called a training example
- A list of m training examples $\{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$—is called a training set.
- **X** denote the space of input values, and **Y** the space of output values. In this example, **X** = **Y** = **R**.
- Our goal:

Given a training set, learn a function h : **X** → **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.



A slightly richer dataset

- If you want to find the most reasonably priced house satisfying your needs: square-ft, # of bedroom, distance to work place…

Living area (feet^2)	# bedrooms	Price(1000$s)
2104	3	400
1600	3	330
2400	3	369
1416	2	232
3000	4	540
…		
2005	3	?
3200	4	?
1280	2	?

The learning problem

- Features:
  - Living area, # bedroom, distance to work place …
  - Denote as $x=[x_1,x_2,\ldots,x_n]^T$
- Target:
  - Price
  - Denoted as y
- Training set:

$$X=\begin{bmatrix} --(x^{(1)})^T-- \\ --(x^{(2)})^T-- \\ \vdots \\ --(x^{(m)})^T-- \end{bmatrix}=\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{bmatrix} \qquad Y=\begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

m: # examples/samples
n: # features

Linear Regression

- Assume that Y (target) is a linear function of X(features):

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

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

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

The Least-Mean-Square (LMS) method

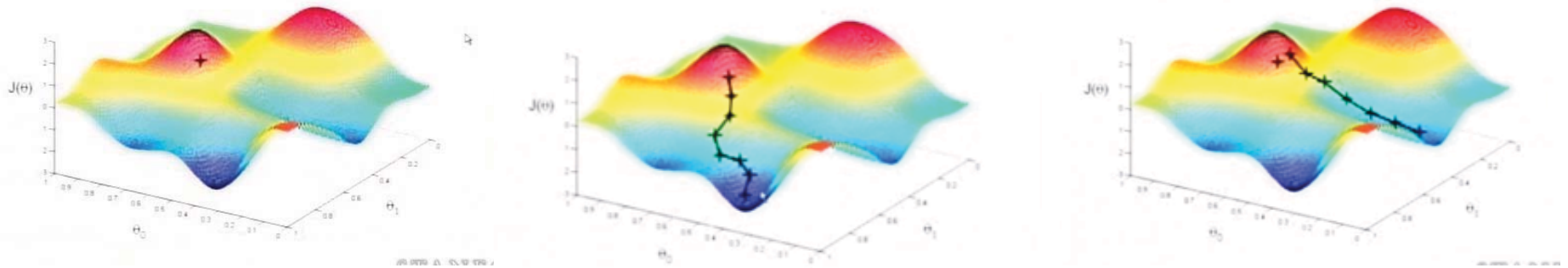
- The Cost Function:

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

- Consider a gradient descent algorithm:

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

Gradient descent



The Least-Mean-Square (LMS) method

- For a single training example, this gives the update rule:

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

- This is known as the LMS update rule, or the Widrow-Hoff learning rule
- If the training set has more than one example

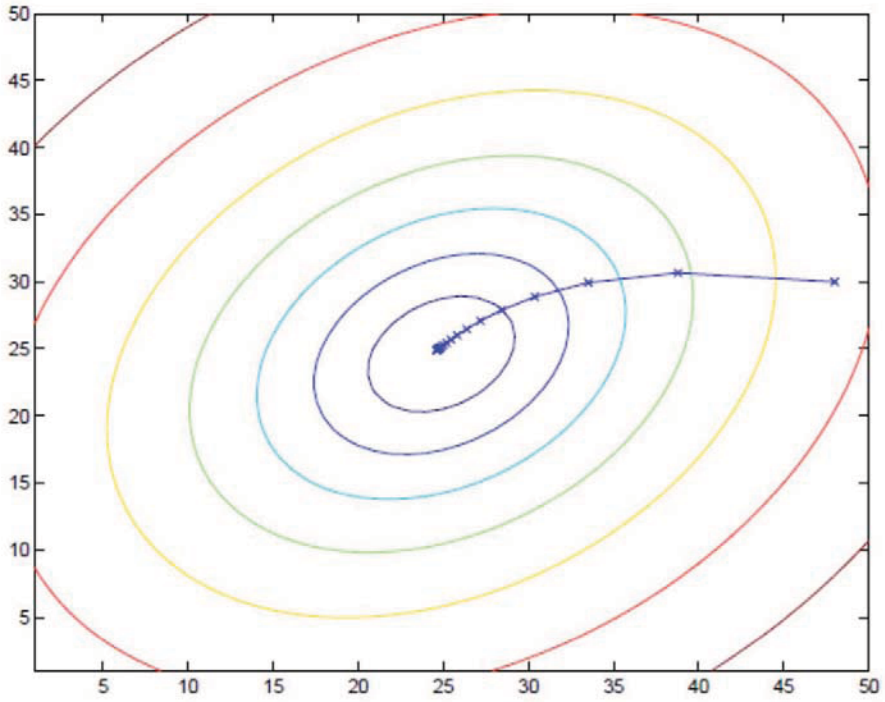
Repeat until convergence {

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

}

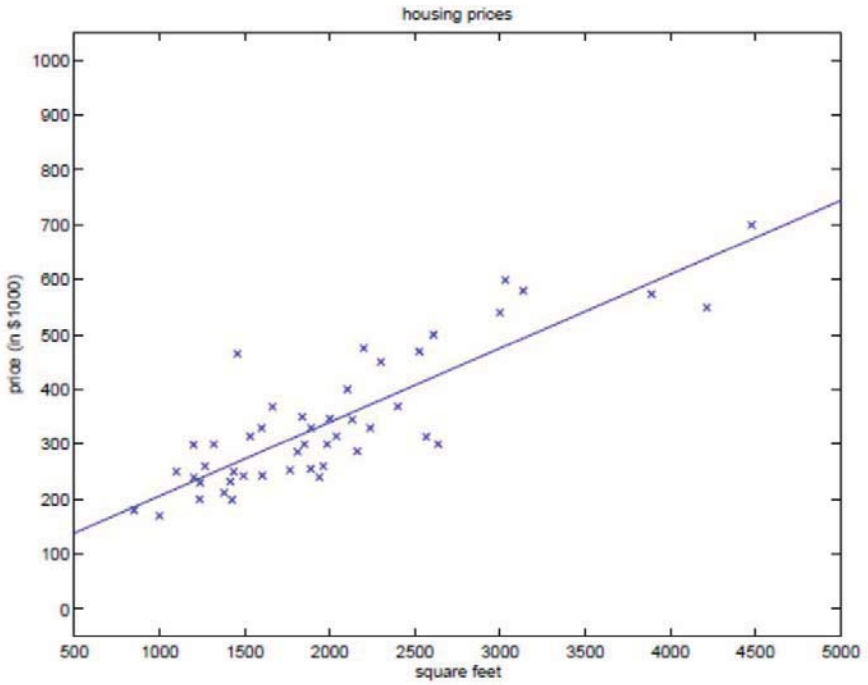
Batch gradient descent

- The ellipses shown right are the contours of a quadratic function.
- Also shown is the trajectory taken by gradient descent, which 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.



Results of batch gradient descent

- 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(x) as a function of x (area), along with the training data, we obtain the figure shown right
- If the number of bedrooms were included as one of the input features as well, we get $\theta_0 = 89.60$, $\theta_1 = 0.1392$, $\theta_2 = -8.738$



Stochastic gradient descent

- 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, {

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

}

}

- When the training set is large, stochastic gradient descent is often preferred over batch gradient descent.

The normal equations

- Given a training set, define the design matrix X to be the m-by-n matrix that contains the training examples' input values in its rows:

$$X=\begin{bmatrix} --(x^{(1)})^T-- \\ --(x^{(2)})^T-- \\ \vdots \\ --(x^{(m)})^T-- \end{bmatrix}=\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{bmatrix} \qquad Y=\begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

m: # samples
n: # features

The normal equations

- Write the cost function in matrix form:

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \frac{1}{2}(X\theta - \vec{y})^T (X\theta - \vec{y}) \\
&= \frac{1}{2}\nabla_\theta \left(\theta^T X^T X\theta - \theta^T X^T \vec{y} - \vec{y}^T X\theta + \vec{y}^T \vec{y}\right) \\
&= \frac{1}{2}\nabla_\theta \operatorname{tr} \left(\theta^T X^T X\theta - \theta^T X^T \vec{y} - \vec{y}^T X\theta + \vec{y}^T \vec{y}\right) \\
&= \frac{1}{2}\nabla_\theta \left(\operatorname{tr} \theta^T X^T X\theta - 2\operatorname{tr} \vec{y}^T X\theta\right) \\
&= \frac{1}{2}\left(X^T X\theta + X^T X\theta - 2X^T \vec{y}\right) \\
&= X^T X\theta - X^T \vec{y}
\end{aligned}
$$

The normal equations

- To minimize J, we set its derivatives to zero, and obtain the normal equations:

$$X^T X\theta = X^T \vec{y}$$

- Thus, the value of θ that minimizes J is given in closed form by the equation：

$$\theta = (X^T X)^{-1} X^T \vec{y}.$$

Comments on the normal equation

- In most situations of practical interest, the number of data points N is larger than the dimensionality k of the input space and the matrix $\mathbf{X}$ is of full column rank. If this condition holds, then it is easy to verify that $X^T X$ is necessarily invertible.
- The assumption that $X^T X$ is invertible implies that it is positive definite, thus at the critical point we have found is a minimum.
- What if $\mathbf{X}$ has less than full column rank? → Regularization

Direct and Iterative methods

- Direct methods: we can achieve the solution in a single step by solving the normal equation
  - Using Gaussian elimination or QR decomposition, we converge in a finite number of steps
  - It can be infeasible when data are streaming in in real time, or of very large amount
- Iterative methods: stochastic or steepest gradient
  - Converging in a limiting sense
  - But more attractive in large practical problems
  - Caution is needed for deciding the learning rate α

Probabilistic Interpretation of LMS

- Let us assume that the target variables and the inputs are related via the equation

$$y^{(i)} = \theta^T x^{(i)} + \epsilon^{(i)},$$

where $\epsilon^{(i)}$ is an error term of unmodeled effects or random noise, and distributed i.i.d.

- Now assume that ε follows a Gaussian $N(0,\sigma)$, then we have:

$$p(y^{(i)}|x^{(i)};\theta) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right)$$

- When we wish to explicitly view this as a function of θ, we will instead call it the likelihood function:

$$L(\theta) = L(\theta; X, \vec{y}) = p(\vec{y}|X; \theta)..$$

- By independence assumption:

$$\begin{aligned} L(\theta) &= \prod_{i=1}^{m} p(y^{(i)} \mid x^{(i)}; \theta) \\ &= \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right) \end{aligned}$$

Probabilistic Interpretation of LMS, cont.

- Hence the log-likelihood is:

$$\begin{aligned}
\ell(\theta) &= \log L(\theta) \\
&= \log \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right) \\
&= \sum_{i=1}^{m} \log \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right) \\
&= m \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{\sigma^2} \cdot \frac{1}{2} \sum_{i=1}^{m} (y^{(i)} - \theta^T x^{(i)})^2.
\end{aligned}$$

- Do you recognize the last term? maximizing ℓ(θ) gives the same answer as minimizing J(θ)
- Thus under independence assumption, LMS is equivalent to MLE of θ !

Locally weighted linear regression (LWR)

- The algorithm:
- Instead of minimizing $\sum_i (y^{(i)} - \theta^T x^{(i)})^2$.
- Now we fit θ to minimize $\sum_i w^{(i)} (y^{(i)} - \theta^T x^{(i)})^2$.
- Where do $w^{(i)}$'s come from $w^{(i)} = \exp\left(-\frac{(x^{(i)} - x)^2}{2\tau^2}\right)$

- where **x** is the query point for which we'd like to know its corresponding **y**
- Essentially we put higher weights on (errors on) training examples that are close to the query point (than those that are further away from the query)

Parametric vs. non-parametric

- Locally weighted linear regression is the first example we are running into of a **non-parametric** algorithm.
- The (unweighted) linear regression algorithm that we saw earlier is known as a **parametric** learning algorithm because it has a fixed, finite number of parameters (the θ), which are fit to the data;
- Once we've fit the θ and stored them away, we no longer need to keep the training data around to make future predictions.
- In contrast, to make predictions using locally weighted linear regression, we need to keep the entire training set around.
- The term "non-parametric" (roughly) refers to the fact that the amount of stuff we need to keep in order to represent the hypothesis grows linearly with the size of the training set.