Machine Learning
Regression

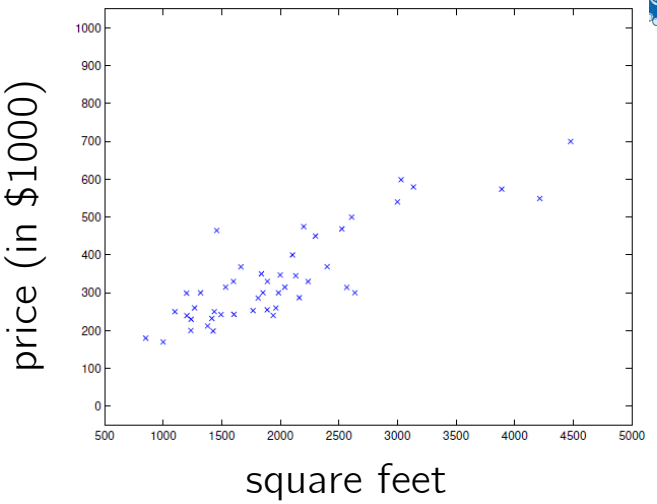
AI Summer School

University of Tehran

Slides adopted from Stanford CS229

Prices of houses

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



- Given data like this, how can we **learn** to **predict** the prices of other houses, **as a function** of the size of their living areas?
- A pair $(x^{(i)}, y^{(i)})$ is called a **training example**, and the dataset that we'll be using to learn—a list of n training examples $\{(x^{(i)}, y^{(i)}); i=1, \ldots, n\}$—is called a **training set**.
- We used superscript "(i)" in the notation for regression to denote an index into the data set. In other section, we usually use **subscript**.

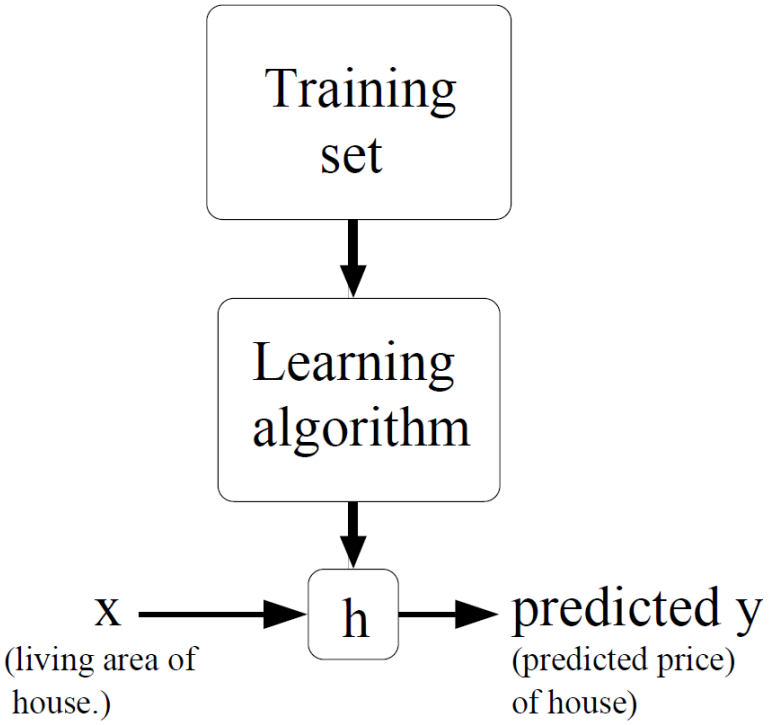
Linear Regression

- We approximate y as **a linear function** of x:

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

- To perform **supervised learning**, we must decide how we're going to **represent** functions/hypotheses h in a computer.
- The θ_i's are the **parameters** (also called weights) **parameterizing** the space of linear functions mapping from X to Y
- Letting $x_0 = 1$ (this is the **intercept** term), so that the (the new **convention**)

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



$$h : \mathcal{X} \mapsto \mathcal{Y}$$

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.
- 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** (the ordinary least squares):

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

- We want to choose θ so as to **minimize** J(θ).

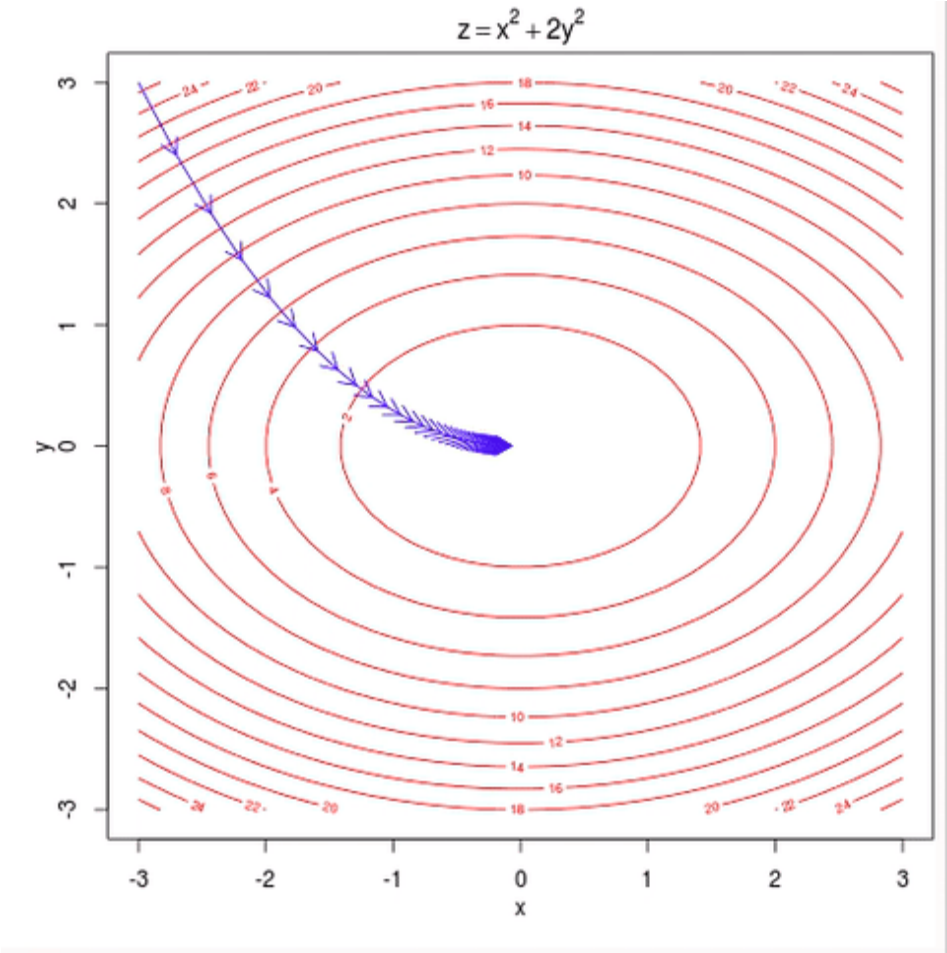
Gradient descent algorithm to find θ

- We update all values of θ_j , $j = 0, \ldots, d$

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

- With some "**initial guess**" for θ, and that **repeatedly** changes θ to make J(θ) smaller, until **hopefully** we **converge to a value of θ** that minimizes J(θ).
- **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

Gradient descent



Partial derivative term

- For the case of if we have **only one training example** (x, y) (neglect the sum in the definition of J)

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

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

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

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

least mean squares (**LMS**) update rule or **Widrow-Hoff** learning rule.

Widrow-Hoff learning rule.

- The magnitude of the update is **proportional** to the **error** term $(y^{(i)} - h(x^{(i)}))$;
  - 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 $h(x^{(i)})$ has **a large error** (i.e., if it is very far from $y^{(i)}$).

Modifying method for a training set of more than one example (Solution I)

- **Batch** gradient descent
  - Looks at every example in the **entire training set** on every step, and is called.

Repeat until convergence {

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

}

- **Vector notation**:

$$\theta := \theta + \alpha \sum_{i=1}^{n} \left(y^{(i)} - h_\theta(x^{(i)})\right) x^{(i)}$$

Solution II: stochastic gradient descent

- To update the parameters according to the **gradient of the error** with respect to that s**ingle training example only**.

Loop {

for $i = 1$ to n, {

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

}

}

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

$$\theta := \theta + \alpha \sum_{i=1}^{n} \left(y^{(i)} - h_\theta(x^{(i)})\right) x^{(i)}$$

- Often, stochastic gradient descent gets θ "close" to the minimum **much faster** than batch gradient descent.
- When training **set is large**, stochastic gradient descent is **often preferred** over batch gradient descent

The normal equations

- Performing the minimization explicitly and **without resorting to an iterative algorithm**.
- Define the **design matrix** X to be the n-by-d matrix that contains the training examples' input values in its rows
- Also, let $\vec{y}$ be the n-dimensional vector containing all the target values from the training set

$$X = \begin{bmatrix} — & (x^{(1)})^T & — \\ — & (x^{(2)})^T & — \\ & \vdots & \\ — & (x^{(n)})^T & — \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}$$

The normal equations

since $h_\theta(x^{(i)}) = (x^{(i)})^T\theta,$

$$X\theta - \vec{y} = \begin{bmatrix} (x^{(1)})^T\theta \\ \vdots \\ (x^{(n)})^T\theta \end{bmatrix} - \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(n)} \end{bmatrix} = \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ \vdots \\ h_\theta(x^{(n)}) - y^{(n)} \end{bmatrix}$$

$$\frac{1}{2}(X\theta - \vec{y})^T(X\theta - \vec{y}) = \frac{1}{2}\sum_{i=1}^{n}(h_\theta(x^{(i)}) - y^{(i)})^2 = J(\theta)$$

To minimize J, let's find its derivatives with respect to θ.

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

$a^Tb = b^Ta$

$\nabla_x b^T x = b$

$\nabla_x x^T A x = 2Ax$

The normal equations

- We set its **derivatives to zero**, and obtain the normal equations:

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

- The value of θ that minimizes J(θ) is given in **closed form** by the equation

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

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

Classification and logistic regression

- Let's now talk about the classification problem.
  - This is just like the regression problem, except that the **values y** we now want to predict take on only a **small number** of **discrete values**.
- For now, we focus on the **binary classification** problem in which y can take on only two values, 0 and 1.
  - In most cases the binary classifier will also **generalize to the multiple-class** case
- $y^{(i)}$ is called the **label for the training** example.
- Logistic **regression**:
  - We could **approach the classification** problem **ignoring** the fact that y is discrete-valued

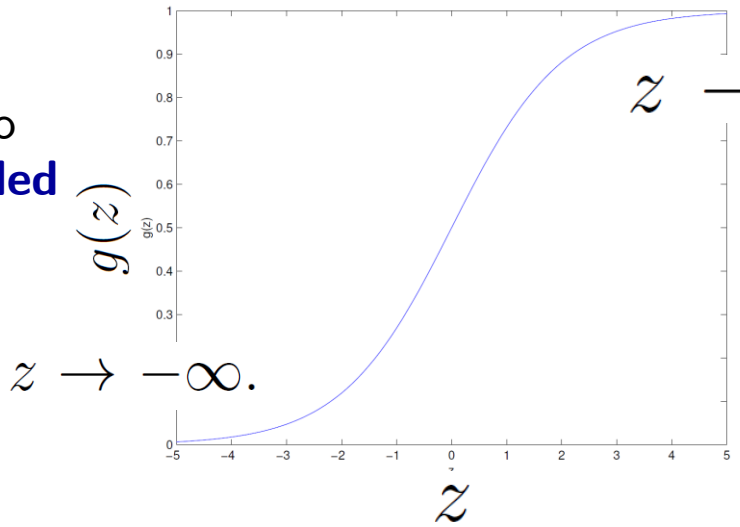
Logistic function or the sigmoid function.

- It also doesn't make sense for $h_\theta(x)$ to take values larger than 1 or smaller than 0 when we know that $y \in \{0,1\}$;
- We will choose:

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

$$\boxed{g(z) = \frac{1}{1 + e^{-z}}}$$

g(z), and hence also h(x), is always **bounded** between 0 and 1.



keeping the **convention** of letting $x_0 = 1$, so that:

$$\theta^T x = \theta_0 + \sum_{j=1}^{d} \theta_j x_j$$

Useful property of the derivative of the sigmoid function,

$$
\begin{aligned}
g'(z) &= \frac{d}{dz}\frac{1}{1+e^{-z}} \\
&= \frac{1}{(1+e^{-z})^2}\left(e^{-z}\right) \\
&= \frac{1}{(1+e^{-z})}\cdot\left(1-\frac{1}{(1+e^{-z})}\right) \\
&= g(z)(1-g(z))
\end{aligned}
$$

- **Other functions** that smoothly increase from 0 to 1 can also be used
- The choice of the logistic function is a **fairly natural** one: (GLMs, and generative learning algorithms)

Stochastic gradient ascent rule and **perceptron learning**

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

- It is **similar to LMS** update rule; but this is **not** the same algorithm, because **$h_\theta(x^{(i)})$ is now defined as a non-linear function of $\theta^T x^{(i)}$**
- There is a **deeper reason** on ending up with the same update rule for a rather different algorithm and learning problem. (GLM models)
- Digression: The **perceptron learning** algorithm
  - Modifying the logistic regression method to "**force" it** to output values that are either 0 or 1 or exactly.

$$g(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

- Using this modified definition of g, and if we use **the same update rule**, then we have the **perceptron learning algorithm**.