Optimization Introduction

Jin Xie <jin.xie@uky.edu>

Slides mostly by Andrew Ng

February 22, 2018

Outline

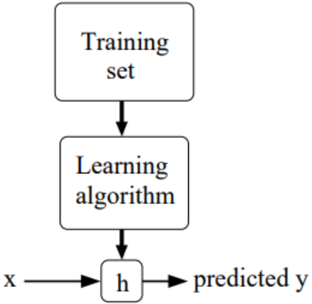
Outline

Formulas in this note

- **Features**: $x^{(i)}$
- "Output" or **target** variable: $y^{(i)}$
- A pair $(x^{(i)}, y^{(i)})$ is called a **training example**
- A list of m training examples $\{(x^{(i)}, y^{(i)}) : i = 1, \dots, m\}$ is called a **training set**

Supervised Learning Problem

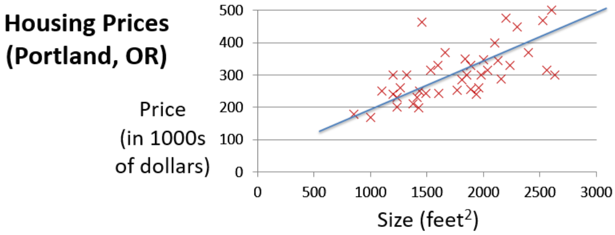
In supervised learning problems, our goal is, given a training set, to learn function $h : \mathcal{X} \mapsto \mathcal{Y}$ so that $h(x)$ is a "good" predictor for the corresponding value of y.



For historical reasons, h is called a **hypothesis**.

Example: regression problem

When y is continuous, we call the learning problem a **regression** problem.



Example: classification problem

When y takes on only a small number of discrete values, we call the learning problem a **classification** problem.

Outline

Supervised Learning

Learning Regression

Gradient Descent

Normal Equations

Logistic Regression

Newton's Method

Learning Regression

We want to predict the house price using living area and number of bedrooms. Here, x's are 2-dimensional.

- $x_1^{(i)}$ is the living area of i-th house in the training set
- $x_2^{(i)}$ is its number of bedrooms

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

Learning Regression

If we approximate y as a linear function of x:

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

- θ_i's are the **parameters** (also called **weights**)
- By letting $x_0 = 1$ (**intercept**), we simplify the notation as

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

To quantify how close $h(x^{(i)})$'s are to the corresponding $y^{(i)}$'s, we define the **cost function**:

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

Outline

Target: Minimizing $J(\theta)$

Let's 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 $J(\theta)$.

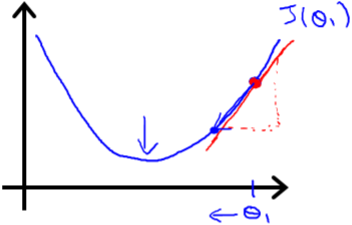
Gradient Descent

Starting with some initial θ, the Gradient Descent algorithm repeatedly performs the update:

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

This update is simultaneously performed for all θ_j $j = 0, \ldots, n$. Here, α is called the **learning rate.**

Logic for 1-dimensional GD

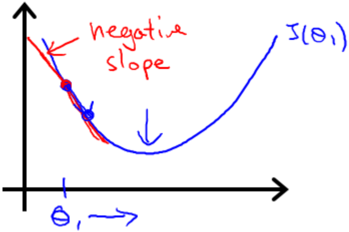


$(\theta_1 \in \mathbb{R})$

$$\theta_1 := \theta_1 - \alpha \boxed{\frac{d}{d\theta_1} J(\theta_1)} \quad \frac{\partial}{\partial \theta_1}$$

≥ 0

$$\theta_1 := \theta_1 - \underline{\alpha} \cdot (\text{positive number})$$



$$\underline{\frac{d}{d\theta_1} J(\theta_1)}$$

≤ 0

$$\theta_1 := \underline{\theta_1} - \alpha\, (\text{negative number})$$

Calculate $\frac{\partial}{\partial\theta_j}J(\theta)$

In order to update, we need $\frac{\partial}{\partial\theta_j}J(\theta)$.

$$\begin{aligned}
\frac{\partial}{\partial\theta_j}J(\theta) &= \frac{\partial}{\partial\theta_j}\frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 \\
&= \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})\frac{\partial}{\partial\theta_j}(h_\theta(x^{(i)}) - y^{(i)}) \\
&= \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})\frac{\partial}{\partial\theta_j}\left[\sum_{k=0}^{n}\theta_k x_k^{(i)} - y^{(i)}\right] \\
&= \sum_{i=1}^{m}(\theta^T x^{(i)} - y^{(i)})x_j^{(i)}
\end{aligned}$$

Gradient Descent for LMS

Batch Gradient Descent for LMS

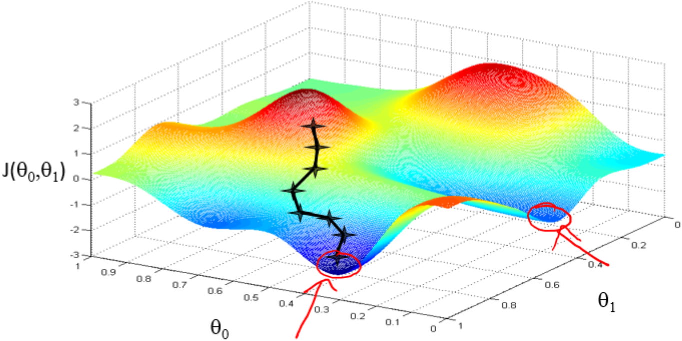
Repeat until convergence {

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

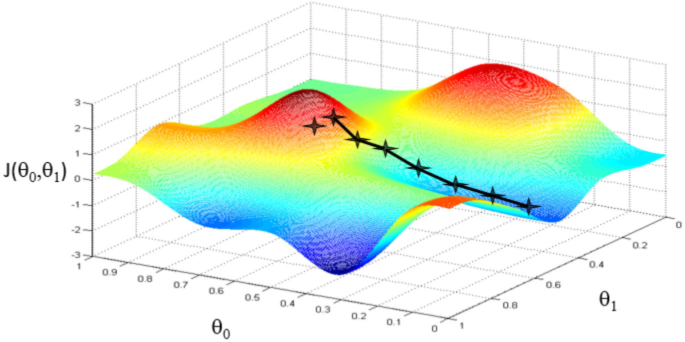
}

This method looks at every example in the entire training set on every step, and is called **batch gradient descent.**

Local minimum in 2-dimensional $J(\theta)$

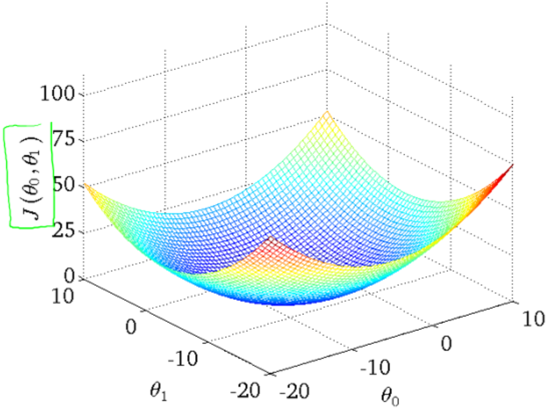


Local minimum in 2-dimensional $J(\theta)$



Global minimum in 2-dimensional $J(\theta)$

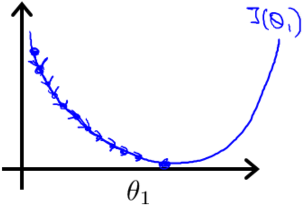
In order to avoid the local minimum, $J(\theta)$ needs to be "**convex**" (or "**convex downward**" or "**concave upward**").



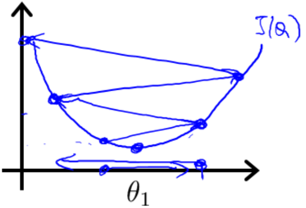
Learning Rate

$$\boxed{\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)}$$

If α is too small, gradient descent can be slow.



If α is too large, gradient descent can overshoot the minimum. It may fail to converge, or even diverge.

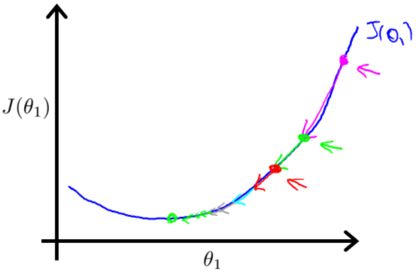


Learning Rate

Gradient descent can converge to a local minimum, even with the learning rate α fixed.

$$\theta_1 := \theta_1 - \alpha \boxed{\frac{d}{d\theta_1} J(\theta_1)}$$

As we approach a local minimum, gradient descent will automatically take smaller steps. So, no need to decrease α over time.



Stochastic Gradient Descent for LMS

Stochastic Gradient Descent (Mini Batch GD) for LMS

Repeat until convergence {

for $i = 1$ to m {

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

}

}

Batch Gradient Descent for LMS

Repeat until convergence {

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

}

Stochastic Gradient Descent for LMS

Stochastic Gradient Descent for LMS

Repeat until convergence {

for $i = 1$ to m {

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

}

}

Whereas **batch gradient descent** has to scan the entire training set before taking a single step (a costly operation if m is large), **stochastic gradient descent** starts making progress right away.

SGD vs. batch gradient descent

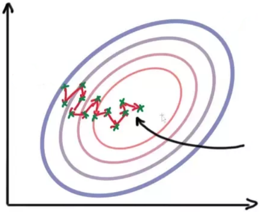


Figure: SGD

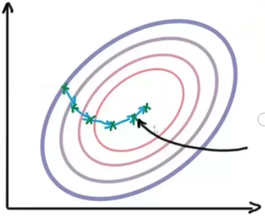


Figure: batch gradient descent

More on SGD

- Often, **SGD** is much faster than **batch gradient descent**.
- Note however that **SGD** may never "converge" to the minimum, and the parameters θ will keep oscillating around the minimum of $J(\theta)$.
- In practice, **SGD** often performs better than **batch gradient descent**.
- An alternative is **mini-batch gradient descent** which seeks a balance between **SGD** and **batch GD.**
  - **mini-batch gradient descent** splits the training set into batches, and apply **SGD** on each batch.

Outline

LMS Revisit

We all learned the closed form solution to LMS in STA 603. Let X be the design matrix, then

$$J(\theta) = \frac{1}{2}(X\theta - y)^T(X\theta - y)$$
$$\nabla_\theta J(\theta) = X^T X\theta - X^T y.$$

To minimize $J(\theta)$, we set the derivatives to zero and obtain the **normal equations**:

$$X^T X\theta = X^T y.$$

Thus the value of θ that minimizes $J(\theta)$ is given in a closed form by the equation

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

Gradient Descent vs. Normal Equation

m **training examples,** n **features.**

Gradient Descent	Normal Equation
• Need to choose α. • Needs many iterations. • Works well even when n is large.	• No need to choose α. • Don't need to iterate. • Need to compute $(X^T X)^{-1}$ • Slow if n is very large, sometimes even unable to compute.

Outline

Logistic Regression

Logistic Regression Model

Want $\boxed{0 \leq h_\theta(x) \leq 1}$

$$h_\theta(x) = g(\theta^T x)$$

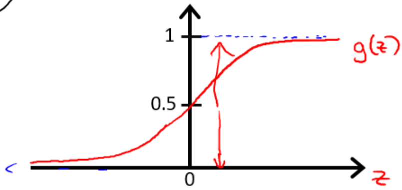
$$g(z) = \frac{1}{1+e^{-z}}$$

$\theta^T x$

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

Sigmoid function

Logistic function



Parameters θ.

Logistic Regression

Useful property of the sigmoid function:

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

Logistic Regression

Question: Which cost function should we use? Same as the square loss in linear regression?

Logistic Regression

Assume that

$$P(y=1|x,\theta) = h_\theta(x)$$
$$P(y=0|x,\theta) = 1 - h_\theta(x).$$

Then we can write down the likelihood as

$$L(\theta) = \prod_{i=1}^{m} \left(h_\theta(x^{(i)})\right)^{y^{(i)}} \left(1 - h_\theta(x^{(i)})\right)^{1-y^{(i)}}.$$

The log likelihood is

$$l(\theta) = \log L(\theta) = \sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1-y^{(i)}) \log(1-h_\theta(x^{(i)})).$$

Logistic Regression

Calculate the derivative of $l(\theta)$:

$$\frac{\partial}{\partial\theta_j} l(\theta) = \sum_{i=1}^{m} \left[y^{(i)} \frac{1}{g(\theta^T x^{(i)})} - (1 - y^{(i)}) \frac{1}{1 - g(\theta^T x^{(i)})} \right] \frac{\partial}{\partial\theta_j} g(\theta^T x^{(i)})$$

$$= \sum_{i=1}^{m} \left[y^{(i)} \frac{1}{g(\theta^T x^{(i)})} - (1 - y^{(i)}) \frac{1}{1 - g(\theta^T x^{(i)})} \right]$$

$$\cdot\, g(\theta^T x^{(i)})[1 - g(\theta^T x^{(i)})] \frac{\partial}{\partial\theta_j} \theta^T x^{(i)}$$

$$= \sum_{i=1}^{m} [y^{(i)}(1 - g(\theta^T x^{(i)})) - (1 - y^{(i)}) g(\theta^T x^{(i)})] x_j^{(i)}$$

$$= \sum_{i=1}^{m} [y^{(i)} - h_\theta(x^{(i)})] x_j$$

Then we could apply batch gradient descent to calculate the MLE for logistic regression. But in practice, people use BFGS or L-BFGS instead, why?

Outline

Newton's Method

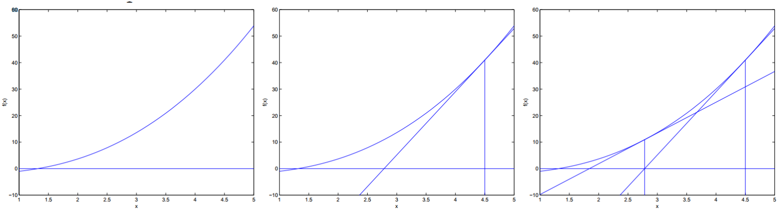
Suppose we have some function $f : \mathbb{R} \mapsto \mathbb{R}$, and we wish to find a value of θ so that $f(\theta) = 0$ ($\theta \in \mathbb{R}$). Newton's Method performs the following update:

$$\theta := \theta - \frac{f(\theta)}{f'(\theta)}.$$

Logic:

- Approximate f via a linear function that is tangent to f at the current guess θ.
- Solving for where that linear function equals to zero.
- Let the next guess for θ be where that linear function is zero.

Newton's Method



Apply Newton's method to log likelihood

We can use Newton's method to solve for $l'(\theta) = 0$ in order to get the maixima. Then the update rule is:

$$\theta := \theta - \frac{l'(\theta)}{l''(\theta)}.$$

We can generalize this to multidimensional setting (called **Newton-Raphson method**):

$$\theta := \theta - H^{-1} \nabla_\theta l(\theta).$$

Here, $\nabla_\theta l(\theta)$ is, as usual, the vector of partial derivatives of $l(\theta)$ with respect to θ_i's. And H is called the **Hessian** matrix, whose entries are given by

$$H_{ij} = \frac{\partial^2 l(\theta)}{\partial \theta_i \partial \theta_j}.$$

Newton's method vs. Gradient Descent

Newton's Method:

Pros:

- Faster convergence than (batch) gradient descent
- Requires many fewer iterations to get very close to the minimum.

Cons:

- Requires finding and inverting an n-by-n Hessian
- Slower when n becomes larger.

Different quasi Newton's method (BFGS, L-BFGS) varies in how they calculate and update the (local) Hessian matrix at each step.