

6.4 代价函数

参考视频: 6 - 4 - Cost Function (11 min).mkv

在这段视频中，我们要介绍如何拟合逻辑回归模型的参数 θ 。具体来说，我要定义用来拟合参数的优化目标或者叫**代价函数**，这便是监督学习问题中的逻辑回归模型的拟合问题。

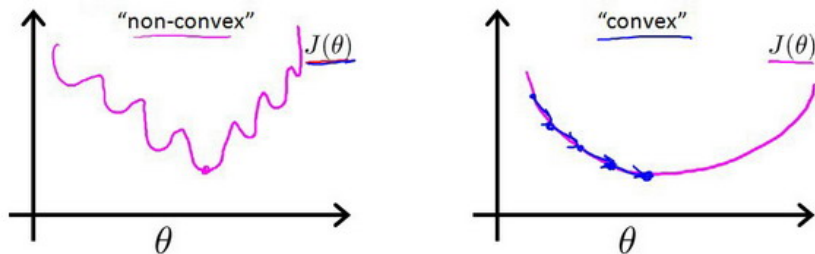
Training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$

m examples $x \in \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_n \end{bmatrix} \quad x_0 = 1, y \in \{0, 1\}$

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

How to choose parameters θ ?

对于线性回归模型，我们定义的**代价函数是所有模型误差的平方和**。理论上来说，我们也可以对逻辑回归模型沿用这个定义，但是问题在于，当我们将 $h_{\theta}(x) = \frac{1}{1+e^{-\theta^T x}}$ 带入到这样定义的代价函数中时，我们得到的代价函数将是一个**非凸函数 (non-convex function)**。



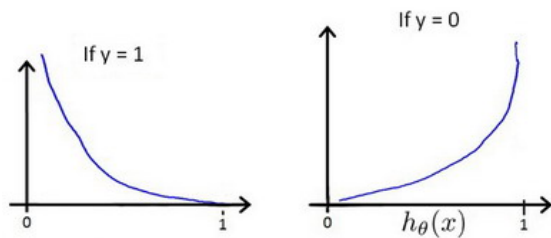
这意味着我们的代价函数有许多局部最小值，这将影响梯度下降算法寻找全局最小值。

线性回归的代价函数为: $J(\theta) = \frac{1}{m} \sum_{i=1}^m \frac{1}{2} (h_{\theta}(x^{(i)}) - y^{(i)})^2$ 。

我们**重新定义逻辑回归的代价函数为**: $J(\theta) = \frac{1}{m} \sum_{i=1}^m \text{Cost}(h_{\theta}(x^{(i)}), y^{(i)})$ ，其中

$$\text{Cost}(h_{\theta}(x), y) = \begin{cases} -\log(h_{\theta}(x)) & \text{if } y = 1 \\ -\log(1 - h_{\theta}(x)) & \text{if } y = 0 \end{cases}$$

$h_{\theta}(x)$ 与 $\text{Cost}(h_{\theta}(x), y)$ 之间的关系如下图所示:



这样构建的 $Cost(h_{\theta}(x), y)$ 函数的特点是：当实际的 $y = 1$ 且 $h_{\theta}(x)$ 也为 1 时误差为 0，当 $y = 1$ 但 $h_{\theta}(x)$ 不为 1 时误差随着 $h_{\theta}(x)$ 变小而变大；当实际的 $y = 0$ 且 $h_{\theta}(x)$ 也为 0 时代价为 0，当 $y = 0$ 但 $h_{\theta}(x)$ 不为 0 时误差随着 $h_{\theta}(x)$ 的变大而变大。

将构建的 $Cost(h_{\theta}(x), y)$ 简化如下：

$$Cost(h_{\theta}(x), y) = -y \times \log(h_{\theta}(x)) - (1 - y) \times \log(1 - h_{\theta}(x))$$

带入代价函数得到：

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

$$\text{即： } J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(h_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))]$$

Python 代码实现：

```
import numpy as np
def cost(theta, X, y):
    theta = np.matrix(theta)
    X = np.matrix(X)
    y = np.matrix(y)
    first = np.multiply(-y, np.log(sigmoid(X* theta.T)))
    second = np.multiply((1 - y), np.log(1 - sigmoid(X* theta.T)))
    return np.sum(first - second) / (len(X))
```

在得到这样一个代价函数以后，我们便可以用梯度下降算法来求得能使代价函数最小的参数了。算法为：

```
Repeat {  $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$ 
        (simultaneously update all  $\theta_j$  )
    }
```

求导后得到：

```
Repeat {  $\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$ 
        (simultaneously update all  $\theta_j$  )
    }
```

在这个视频中，我们定义了单训练样本的代价函数，凸性分析的内容是超出这门课的范围的，但是可以证明我们所选的代价值函数会给我们一个凸优化问题。代价函数 $J(\theta)$ 会是一个凸函数，并且没有局部最优值。

推导过程：

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(h_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))]$$

$$\text{考虑： } h_{\theta}(x^{(i)}) = \frac{1}{1 + e^{-\theta^T x^{(i)}}}$$

$$\text{则： } y^{(i)} \log(h_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))$$

$$= y^{(i)} \log\left(\frac{1}{1 + e^{-\theta^T x^{(i)}}}\right) + (1 - y^{(i)}) \log\left(1 - \frac{1}{1 + e^{-\theta^T x^{(i)}}}\right)$$

$$= -y^{(i)} \log(1 + e^{-\theta^T x^{(i)}}) - (1 - y^{(i)}) \log(1 + e^{\theta^T x^{(i)}})$$

$$\text{所以： } \frac{\partial}{\partial \theta_j} J(\theta) = \frac{\partial}{\partial \theta_j} \left[-\frac{1}{m} \sum_{i=1}^m [-y^{(i)} \log(1 + e^{-\theta^T x^{(i)}}) - (1 - y^{(i)}) \log(1 + e^{\theta^T x^{(i)}})] \right]$$

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

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

$$= -\frac{1}{m} \sum_{i=1}^m \frac{y^{(i)} x_j^{(i)} - x_j^{(i)} e^{\theta^T x^{(i)}} + y^{(i)} x_j^{(i)} e^{\theta^T x^{(i)}}}{1 + e^{\theta^T x^{(i)}}}$$

$$= -\frac{1}{m} \sum_{i=1}^m \frac{y^{(i)} (1 + e^{\theta^T x^{(i)}}) - e^{\theta^T x^{(i)}}}{1 + e^{\theta^T x^{(i)}}} x_j^{(i)}$$

$$= -\frac{1}{m} \sum_{i=1}^m \left(y^{(i)} - \frac{e^{\theta^T x^{(i)}}}{1 + e^{\theta^T x^{(i)}}} \right) x_j^{(i)}$$

$$= -\frac{1}{m} \sum_{i=1}^m \left(y^{(i)} - \frac{1}{1 + e^{-\theta^T x^{(i)}}} \right) x_j^{(i)}$$

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

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

注：虽然得到的梯度下降算法表面上看上去与线性回归的梯度下降算法一样，但是这里的 $h_{\theta}(x) = g(\theta^T x)$ 与线性回归中不同，所以实际上是不一样的。另外，在运行梯度下降算法

之前，进行特征缩放依旧是非常必要的。

一些梯度下降算法之外的选择：除了梯度下降算法以外，还有一些常被用来令代价函数最小的算法，这些算法更加复杂和优越，而且通常不需要人工选择学习率，通常比梯度下降算法要更加快速。这些算法有：**共轭梯度（Conjugate Gradient）**，**局部优化法(Broyden fletcher goldfarb shann,BFGS)**和**有限内存局部优化法(LBFGS)**，**fminunc** 是 **matlab** 和 **octave** 中都带的一个最小值优化函数，使用时我们需要提供代价函数和每个参数的求导，下面是 **octave** 中使用 **fminunc** 函数的代码示例：

```
function [jVal, gradient] = costFunction(theta)
    jVal = [...code to compute J(theta)...];
    gradient = [...code to compute derivative of J(theta)...];
end
options = optimset('GradObj', 'on', 'MaxIter', '100');
initialTheta = zeros(2,1);

[optTheta, functionVal, exitFlag] = fminunc(@costFunction, initialTheta, options);
```

In logistic regression, the cost function for our hypothesis outputting (predicting) $h_{\theta}(x)$ on a training example that has label $y \in \{0, 1\}$ is:

$$\text{cost}(h_{\theta}(x), y) = \begin{cases} -\log h_{\theta}(x) & \text{if } y = 1 \\ -\log(1 - h_{\theta}(x)) & \text{if } y = 0 \end{cases}$$

Which of the following are true? Check all that apply.

- ☒ If $h_{\theta}(x) = y$, then $\text{cost}(h_{\theta}(x), y) = 0$ (for $y = 0$ and $y = 1$).

正确

- ☒ If $y = 0$, then $\text{cost}(h_{\theta}(x), y) \rightarrow \infty$ as $h_{\theta}(x) \rightarrow 1$.

正确

☐ If $y = 0$, then $\text{cost}(h_\theta(x), y) \rightarrow \infty$ as $h_\theta(x) \rightarrow 0$.

未选择的是正确的

☒ Regardless of whether $y = 0$ or $y = 1$, if $h_\theta(x) = 0.5$, then $\text{cost}(h_\theta(x), y) > 0$.

正确

6.5 简化的成本函数和梯度下降

参考视频: 6 - 5 - Simplified Cost Function and Gradient Descent (10 min).mkv

Logistic regression cost function

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \text{Cost}(h_{\theta}(x^{(i)}), y^{(i)})$$

$$\text{Cost}(h_{\theta}(x), y) = \begin{cases} -\log(h_{\theta}(x)) & \text{if } y = 1 \\ -\log(1 - h_{\theta}(x)) & \text{if } y = 0 \end{cases}$$

Note: $y = 0$ or 1 always

这个式子可以合并成:

$$\text{Cost}(h_{\theta}(x), y) = -y \times \log(h_{\theta}(x)) - (1 - y) \times \log(1 - h_{\theta}(x))$$

即, 逻辑回归的代价函数:

$$\text{Cost}(h_{\theta}(x), y) = -y \times \log(h_{\theta}(x)) - (1 - y) \times \log(1 - h_{\theta}(x))$$

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

根据这个代价函数, 为了拟合出参数, 该怎么做呢? 我们要试图找尽量让 $J(\theta)$ 取得最小值的参数 θ 。

$$\min_{\theta} J(\theta)$$

所以我们想要尽量减小这一项, 这将会得到某个参数 θ 。

最小化代价函数的方法, 是使用**梯度下降法**(gradient descent)。

Want $\min_{\theta} J(\theta)$:

Repeat {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

(simultaneously update all θ_j)

}

Want $\min_{\theta} J(\theta)$:

Repeat {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

(simultaneously update all θ_j)

}

如果你计算一下的话，你会得到这个等式：

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

我把它写在这里，将后面这个式子，在 $i = 1$ 到 m 上求和，其实就是预测误差乘以 $x_j^{(i)}$ ，所以你把这个偏导数项 $\frac{\partial}{\partial \theta_j} J(\theta)$ 放回到原来式子这里，我们就可以将梯度下降算法写作如下形式：

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \dots \\ \theta_n \end{bmatrix}$$

所以，如果你有 n 个特征，也就是说：，参数向量 θ 包括 θ_0 θ_1 θ_2 一直到 θ_n ，那么你就需要用这个式子：

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \text{ 来同时更新所有 } \theta \text{ 的值。}$$

现在，如果你把这个更新规则和我们之前用在线性回归上的进行比较的话，你会惊讶地发现，这个式子正是我们用来做线性回归梯度下降的。

那么，线性回归和逻辑回归是同一个算法吗？要回答这个问题，我们要观察逻辑回归看看发生了哪些变化。实际上，假设的定义发生了变化。

对于线性回归假设函数：

$$h_{\theta}(x) = \theta^T X = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

而现在逻辑函数假设函数： $h_{\theta}(x) = \frac{1}{1 + e^{-\theta^T x}}$

Gradient Descent

$$J(\theta) = -\frac{1}{m} \left[\sum_{i=1}^m y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_{\theta}(x^{(i)})) \right]$$

Want $\min_{\theta} J(\theta)$:

Repeat {

$$\rightarrow \theta_j := \theta_j - \alpha \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

(simultaneously update all θ_j)

}

$$\Theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix}$$

$$h_{\theta}(x) = \Theta^T x$$

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

Algorithm looks identical to linear regression!

Andrew Ng

Andrew Ng

因此，即使更新参数的规则看起来基本相同，但由于假设的定义发生了变化，所以逻辑函数的梯度下降，跟线性回归的梯度下降实际上是两个完全不同的东西。

当使用梯度下降法来实现逻辑回归时，我们有这些不同的参数 θ ，就是 θ_0 θ_1 θ_2 一直到 θ_n ，我们需要用这个表达式来更新这些参数。我们还可以使用 **for 循环**来更新这些参数值，用 **for i=1 to n**，或者 **for i=1 to n+1**。当然，不用 **for 循环**也是可以的，理想情况下，我们更提倡使用向量化的实现，可以把所有这些 n 个参数同时更新。

最后还有一点，我们之前在谈线性回归时讲到的**特征缩放**，我们看到了特征缩放是如何提高梯度下降的收敛速度的，这个特征缩放的方法，也适用于逻辑回归。如果你的特征范围差距很大的话，那么应用特征缩放的方法，同样也可以让逻辑回归中，梯度下降收敛更快。

就是这样，现在你知道如何实现逻辑回归，这是一种非常强大，甚至可能世界上使用最广泛的一种分类算法。

Suppose you are running gradient descent to fit a logistic regression model with parameter $\theta \in \mathbb{R}^{n+1}$. Which of the following is a reasonable way to make sure the learning rate α is set properly and that gradient descent is running correctly?

- ☐ Plot $J(\theta) = \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$ as a function of the number of iterations (i.e. the horizontal axis is the iteration number) and make sure $J(\theta)$ is decreasing on every iteration.
- ☒ Plot $J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))]$ as a function of the number of iterations and make sure $J(\theta)$ is decreasing on every iteration.

正确

- ☐ Plot $J(\theta)$ as a function of θ and make sure it is decreasing on every iteration.
- ☐ Plot $J(\theta)$ as a function of θ and make sure it is convex.

One iteration of gradient descent simultaneously performs these updates:

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)}$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) \cdot x_1^{(i)}$$

\vdots

$$\theta_n := \theta_n - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) \cdot x_n^{(i)}$$

We would like a vectorized implementation of the form $\theta := \theta - \alpha \delta$ (for some vector $\delta \in \mathbb{R}^{n+1}$).

What should the vectorized implementation be?

- ☒ $\theta := \theta - \alpha \frac{1}{m} \sum_{i=1}^m [(h_{\theta}(x^{(i)}) - y^{(i)}) \cdot x^{(i)}]$

正确

6.6 高级优化

参考视频: 6 - 6 - Advanced Optimization (14 min).mkv

在上一个视频中，我们讨论了用梯度下降的方法最小化逻辑回归中代价函数 $J(\theta)$ 。在本次视频中，我会教你们一些高级优化算法和一些高级的优化概念，利用这些方法，我们就能够通过梯度下降，进行逻辑回归的速度大大提高，而这也将使算法更加适合解决大型的机器学习问题，比如，我们有数目庞大的特征量。现在我们换个角度来看什么是梯度下降，我们有个代价函数 $J(\theta)$ ，而我们想要使其最小化，那么我们需要做的是编写代码，当输入参数 θ 时，它们会计算出两样东西： $J(\theta)$ 以及 J 关于 θ 的偏导数项。

Optimization algorithm

Cost function $J(\theta)$. Want $\min_{\theta} J(\theta)$.

Given θ , we have code that can compute

→ $J(\theta)$
→ $\frac{\partial}{\partial \theta_j} J(\theta)$ (for $j = 0, 1, \dots, n$)

Gradient descent:

Repeat {

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

}

假设我们已经完成了可以实现这两件事的代码，那么梯度下降所做的就是反复执行这些更新。

另一种考虑梯度下降的思路是：我们需要写出代码来计算 $J(\theta)$ 和这些偏导数，然后把它们插入到梯度下降中，然后它就可以为我们最小化这个函数。

对于梯度下降来说，我认为从技术上讲，你实际并不需要编写代码来计算代价函数 $J(\theta)$ 。你只需要编写代码来计算导数项，但是，如果你希望代码还要能够监控这些 $J(\theta)$ 的收敛性，那么我们就需要自己编写代码来计算代价函数 $J(\theta)$ 和偏导数项 $\frac{\partial}{\partial \theta_j} J(\theta)$ 。所以，在写完能够计算这两者的代码之后，我们就可以使用梯度下降。

然而梯度下降并不是我们可以使用的唯一算法，还有其他一些算法，更高级、更复杂。如果我们能用这些方法来计算代价函数 $J(\theta)$ 和偏导数项 $\frac{\partial}{\partial \theta_j} J(\theta)$ 两个项的话，那么这些算法就是为我们优化代价函数的不同方法，共轭梯度法 BFGS (变尺度法) 和 L-BFGS (限制变尺度

法) 就是其中一些更高级的优化算法, 它们需要有一种方法来计算 $J(\theta)$, 以及需要一种方法计算导数项, 然后使用比梯度下降更复杂的算法来最小化代价函数。这三种算法的具体细节超出了本门课程的范畴。实际上你最后通常会花费很多天, 或几周时间研究这些算法, 你可以专门学一门课来提高数值计算能力, 不过让我来告诉你他们的一些特性:

这三种算法有许多优点:

一个是使用这其中任何一个算法, 你通常不需要手动选择学习率 α , 所以对于这些算法的一种思路是, 给出计算导数项和代价函数的方法, 你可以认为算法有一个智能的内部循环, 而且, 事实上, 他们确实有一个智能的内部循环, 称为**线性搜索(line search)**算法, 它可以自动尝试不同的学习速率 α , 并自动选择一个好的学习速率 α , 因此它甚至可以为每次迭代选择不同的学习速率, 那么你就不需要自己选择。这些算法实际上在做更复杂的事情, 而不仅仅是选择一个好的学习率, 所以它们往往最终收敛得远远快于梯度下降, 不过关于它们到底做什么的详细讨论, 已经超过了本门课程的范围。

实际上, 我过去使用这些算法已经很长一段时间了, 也许超过十年了, 使用得相当频繁, 而直到几年前我才真正搞清楚**共轭梯度法 BFGS** 和 **L-BFGS** 的细节。

我们实际上完全有可能成功使用这些算法, 并应用于许多不同的学习问题, 而不需要真正理解这些算法的内环间在做什么, 如果说这些算法有缺点的话, 那么我想说主要缺点是它们比梯度下降法复杂多了, 特别是你最好不要使用 **L-BFGS**、**BFGS** 这些算法, 除非你是数值计算方面的专家。实际上, 我不会建议你们编写自己的代码来计算数据的平方根, 或者计算逆矩阵, 因为对于这些算法, 我还是会建议你直接使用一个软件库, 比如说, 要求一个平方根, 我们所能做的就是调用一些别人已经写好用来计算数字平方根的函数。幸运的是现在我们有 **Octave** 和与它密切相关的 **MATLAB** 语言可以使用。

Octave 有一个非常理想的库用于实现这些先进的优化算法, 所以, 如果你直接调用它自带的库, 你就能得到不错的结果。我必须指出这些算法实现得好或不好是有区别的, 因此, 如果你正在你的机器学习程序中使用一种不同的语言, 比如如果你正在使用 **C**、**C++**、**Java** 等等, 你可能会想尝试一些不同的库, 以确保你找到一个能很好实现这些算法的库。因为在 **L-BFGS** 或者等高线梯度的实现上, 表现得好与不太好是有差别的, 因此现在让我们来说明: 如何使用这些算法:

Example:

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$$

$$J(\theta) = (\theta_1 - 5)^2 + (\theta_2 - 5)^2$$

$$\frac{\partial}{\partial \theta_1} J(\theta) = 2(\theta_1 - 5)$$

$$\frac{\partial}{\partial \theta_2} J(\theta) = 2(\theta_2 - 5)$$

比方说，你有一个含两个参数的问题，这两个参数是 θ_0 和 θ_1 ，因此，通过这个代价函数，你可以得到 θ_1 和 θ_2 的值，如果你将 $J(\theta)$ 最小化的话，那么它的最小值将是 $\theta_1 = 5$ ， $\theta_2 = 5$ 。代价函数 $J(\theta)$ 的导数推出来就是这两个表达式：

$$\frac{\partial}{\partial \theta_1} J(\theta) = 2(\theta_1 - 5)$$

$$\frac{\partial}{\partial \theta_2} J(\theta) = 2(\theta_2 - 5)$$

如果我们不知道最小值，但你想要代价函数找到这个最小值，是用比如梯度下降这些算法，但最好是用比它更高级的算法，你要做的就是运行一个像这样的 **Octave** 函数：

```
function [jVal, gradient]=costFunction(theta)
    jVal=(theta(1)-5)^2+(theta(2)-5)^2;
    gradient=zeros(2,1);
    gradient(1)=2*(theta(1)-5);
    gradient(2)=2*(theta(2)-5);
end
```

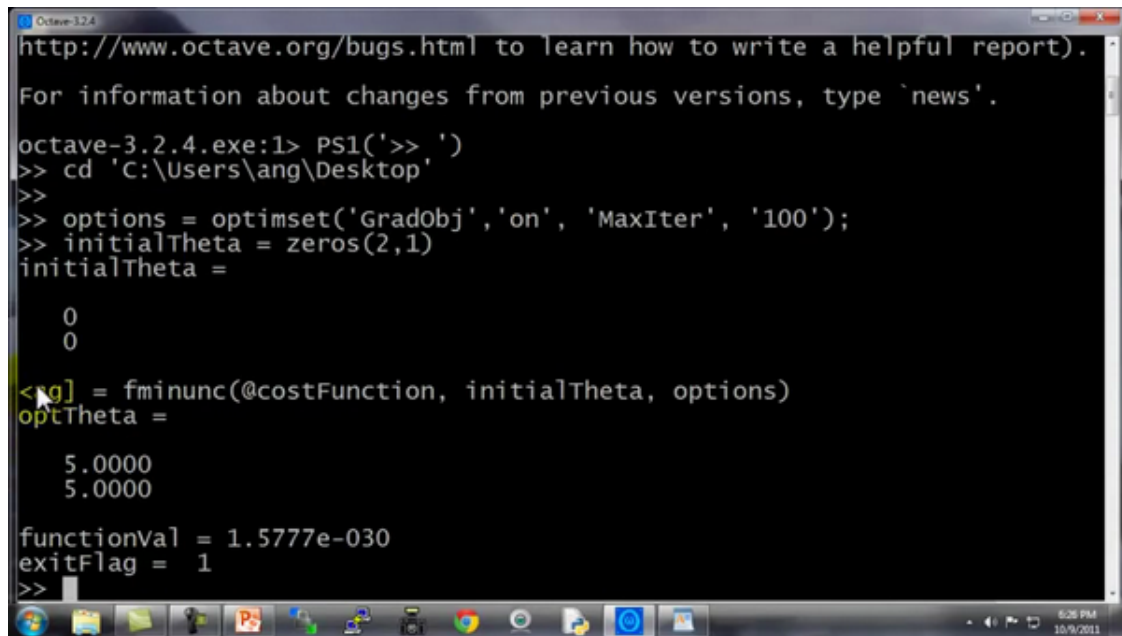
这样就计算出这个代价函数，函数返回的第二个值是梯度值，梯度值应该是一个 2×1 的向量，梯度向量的两个元素对应这里的两个偏导数项，运行这个 **costFunction** 函数后，你就可以调用高级的优化函数，这个函数叫 **fminunc**，它表示 **Octave** 里无约束最小化函数。调用它的方式如下：

```
options=optimset('GradObj','on','MaxIter',100);
initialTheta=zeros(2,1);
[optTheta, functionVal, exitFlag]=fminunc(@costFunction, initialTheta, options);
```

你要设置几个 **options**，这个 **options** 变量作为一个数据结构可以存储你想要的 **options**，所以 **GradObj** 和 **On**，这里设置梯度目标参数为打开(**on**)，这意味着你现在确实要给这个算法提供一个梯度，然后设置最大迭代次数，比方说 100，我们给出一个 θ 的猜测初始值，它是一个 2×1 的向量，那么这个命令就调用 **fminunc**，这个@符号表示指向我们刚刚定义的 **costFunction** 函数的指针。如果你调用它，它就会使用众多高级优化算法中的一个，当然你也可以把它当成梯度下降，只不过它能自动选择学习速率 α ，你不需要自己来做。然后它会

尝试使用这些高级的优化算法，就像加强版的梯度下降法，为你找到最佳的 θ 值。

让我告诉你它在 **Octave** 里什么样：

A screenshot of an Octave-3.2.4 command window. The window title is 'Octave-3.2.4'. The command prompt shows the following code being executed:

```
http://www.octave.org/bugs.html to learn how to write a helpful report).  
For information about changes from previous versions, type 'news'.  
octave-3.2.4.exe:1> PS1('>> ')  
>> cd 'C:\Users\ang\Desktop'  
>>  
>> options = optimset('GradObj','on', 'MaxIter', '100');  
>> initialTheta = zeros(2,1)  
initialTheta =  
  
    0  
    0  
  
<[g] = fminunc(@costFunction, initialTheta, options)  
optTheta =  
  
    5.0000  
    5.0000  
  
functionVal = 1.5777e-030  
exitFlag = 1  
>>
```

 The window also shows a Windows taskbar at the bottom with various icons and a system clock showing 5:26 PM on 10/9/2011.

所以我写了这个关于 **theta** 的 **costFunction** 函数，它计算出代价函数 **jval** 以及梯度 **gradient**，**gradient** 有两个元素，是代价函数对于 **theta(1)** 和 **theta(2)** 这两个参数的偏导数。

我希望你们从这个幻灯片中学习到的主要内容是：写一个函数，它能返回代价函数值、梯度值，因此要把这个应用到逻辑回归，或者甚至线性回归中，你也可以把这些优化算法用于线性回归，你需要做的就是输入合适的代码来计算这里的这些东西。

现在你已经知道如何使用这些高级的优化算法，有了这些算法，你就可以使用一个复杂的优化库，它让算法使用起来更模糊一点。因此也许稍微有点难调试，不过由于这些算法的运行速度通常远远超过梯度下降。

所以当我有一个很大的机器学习问题时，我会选择这些高级算法，而不是梯度下降。有了这些概念，你就应该能将逻辑回归和线性回归应用于更大的问题中，这就是高级优化的概念。

在下一个视频，我想要告诉你如何修改你已经知道的逻辑回归算法，然后使它在多类别分类问题中也能正常运行。