章节 1：无约束最优化问题的求解算法

使用梯度下降法目的和原因

目的

梯度下降法(Gradient Descent)是一个算法，但不是像多元线性回归那样是一个具体做回归任务的算法，而是一个非常通用的优化算法来帮助一些机器学习算法求解出最优解的，所谓的通用就是很多机器学习算法都是用它，甚至深度学习也是用它来求解最优解。所有优化算法的目的都是期望以最快的速度把模型参数θ求解出来，梯度下降法就是一种经典常用的优化算法。

原因

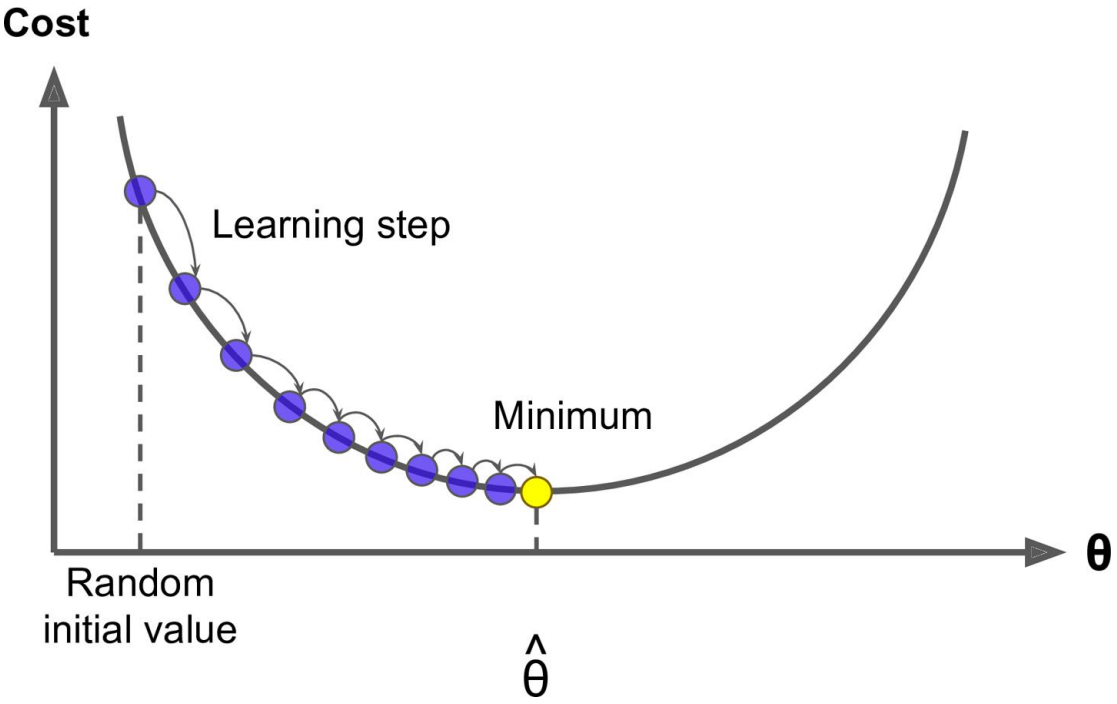
之前利用θ的解析解公式求解出来的解我们就直接说是最优解的一个原因是因为 MSE 这个损失函数是凸函数，但是如果我们机器学习的损失函数是非凸函数的话，设置梯度为 0 会得到很多个极值，甚至是极大值都有可能。

之前利用θ的解析解公式求解的另一个原因是特征维度并不多，但是细致分析一下公式里面 $X^T X$ 对称阵是 N 维乘以 N 维的，复杂度是是 O(N)的三次方，换句话说，就是如果你的特征数量翻倍，你的计算时间大致上要 2 的三次方，8 倍的慢

比如，4 个特征 1 秒，8 个特征就是 8 秒，16 个特征就是 64 秒，当维度更多的时候呢？

所以其实之前一步求出最优解并不是机器学习甚至深度学习常用的手段，如下图，之前我们是设置梯度为 0，反过来求解最低点的时候θ是多少，而梯度下降法是一点点去逼近最优解！

梯度下降法的思想



其实这就跟生活中的情形很像，比如你问一个朋友的工资是多少，他说你猜？然后你会觉得很难猜，他说你猜完我告诉你是猜高了还是猜低了，这样你就可以奔着对的方向一直猜下去，最后总有一下你能猜对。梯度下降法就是这样的，你得去试很多次，而且是不是我们在试的过程中还得想办法知道是不是在猜对的路上，说白了就是得到正确的反馈再调整然后继续猜才有意义。

一般你玩儿这样的游戏的时候，一开始第一下都是随机瞎猜一个对吧，那对于计算机来说是不是就是随机取值，也就是说你有$\theta = W1 \ldots Wn$，这里θ强调一下不是一个值，而是一个向量就是一组 W，一开始的时候我们通过随机把每个值都给它随机出来。有了θ我们可以去根据算法就是公式去计算出来$\hat{y}$，比如$\hat{y} = X\theta$，然后根据计算$\hat{y}$和真实 y 之间的损失比如 MSE，然后调整θ再去计算 MSE。

这个调整正如咱们前面说的肯定不是瞎调整，当然这个调整的方式很多，你可以整体θ每个值调大一点，也可以整体θ每个值调小一点，也可以一部分调大一部分调小。第一次θ_0我们可以得到第一次的 MSE 就是 Loss0，调整后第二次θ_1对应可以得到第二次的 MSE 就是 Loss1，如果 loss 变小是不是调对了，就应该继续调，如果 loss 反而变大是不是调反了，就应该反过来调。直到 MSE 我们找到最小值时计算出来的$\hat{\theta}$就是我们的最优解。

这个就好比道士下山，我们把 loss 看出是曲线就是山谷，如果走过来就再往回调，所以是一个迭代的过程。

作业：

思考有几种方式我们可以知道迭代过程中我们是调对了还是调错了？

1. Loss 是否在逐渐变小

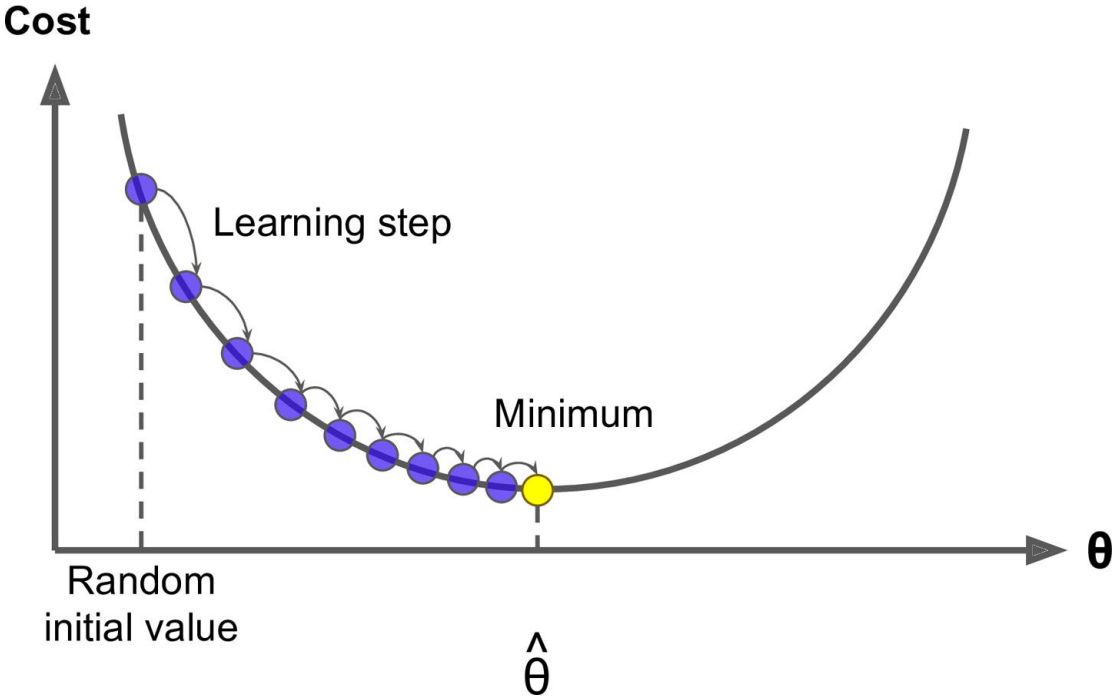
2. 切线的斜率，gradient 绝对值是否在变小

梯度下降法公式

这里梯度下降法的公式就是一个式子指导计算机迭代过程中如何去调整θ，不需要推导和证明，就是总结出来的

$$W_j^{t+1} = W_j^t - \eta \cdot gradient_j$$

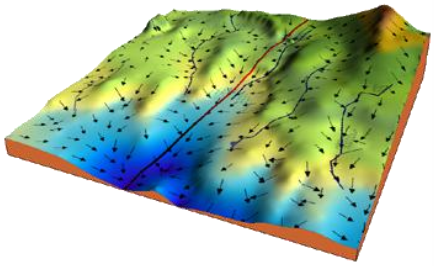
这里的 Wj 就是θ中的某一个 j=0…n，这里的η就是图里的 learning step，很多时候也叫学习率 learning rate，很多时候也用α表示，这个学习率我们可以看作是下山迈的步子的大小，步子迈的大下山就快。

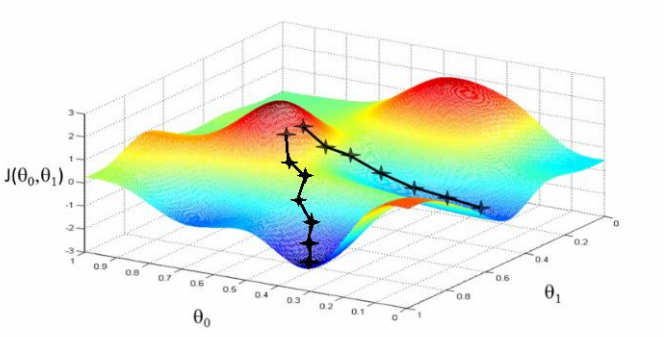


学习率一般都是正数，那么在山左侧梯度是负的，那么这个负号就会把 W 往大了调，如果在山右侧梯度就是正的，那么负号就会把 W 往小了调。每次 Wj 调整的幅度就是η*gradient，就是横轴上移动的距离。

如果特征或维度越多，那么这个公式用的次数就越多，也就是每次迭代要应用的这个式子 n+1 次，所以其实上面的图不是特别准，因为θ对应的是很多维度，应该每一个维度都可以画一个这样的图，或者是一个多维空间的图。

$$W_0^{t+1} = W_0^t - \eta \cdot g_0$$
$$W_1^{t+1} = W_1^t - \eta \cdot g_1$$
$$W_j^{t+1} = W_j^t - \eta \cdot g_j$$
$$W_n^{t+1} = W_n^t - \eta \cdot g_n$$



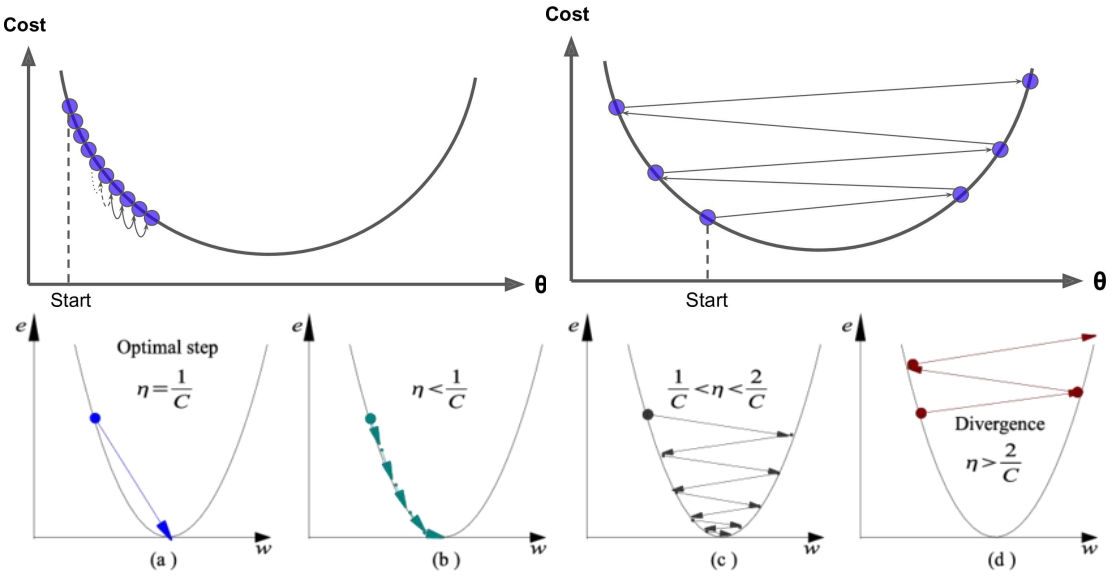


所以观察图我们可以发现不是某一个 θ_0 或 θ_1 找到最小值就是最优解，而是它们一起找到 J 最小的时候才是最优解。

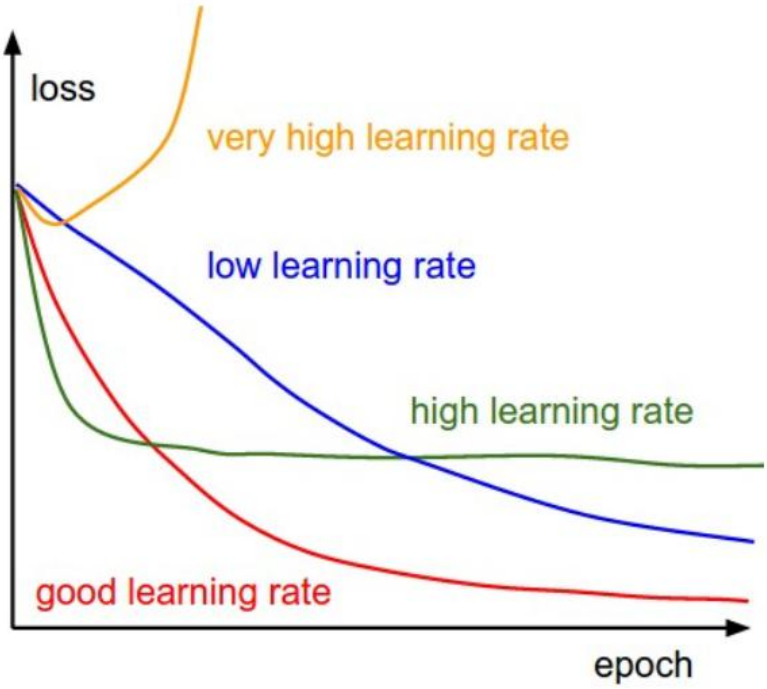
学习率设置的学问

$$W_j^{t+1} = W_j^t - \eta \cdot gradient_j$$

根据我们上面讲的梯度下降法公式，我们知道η是学习率，设置大的学习率 Wj 每次调整的幅度就大，设置小的学习率 Wj 每次调整的幅度就小，然而如果步子迈的太大也会有问题其实，俗话说步子大了容易扯着蛋，可能一下子迈到山另一头去了，然后一步又迈回来了，使得来来回回震荡。步子太小呢就一步步往前挪，也会使得整体迭代次数增加。

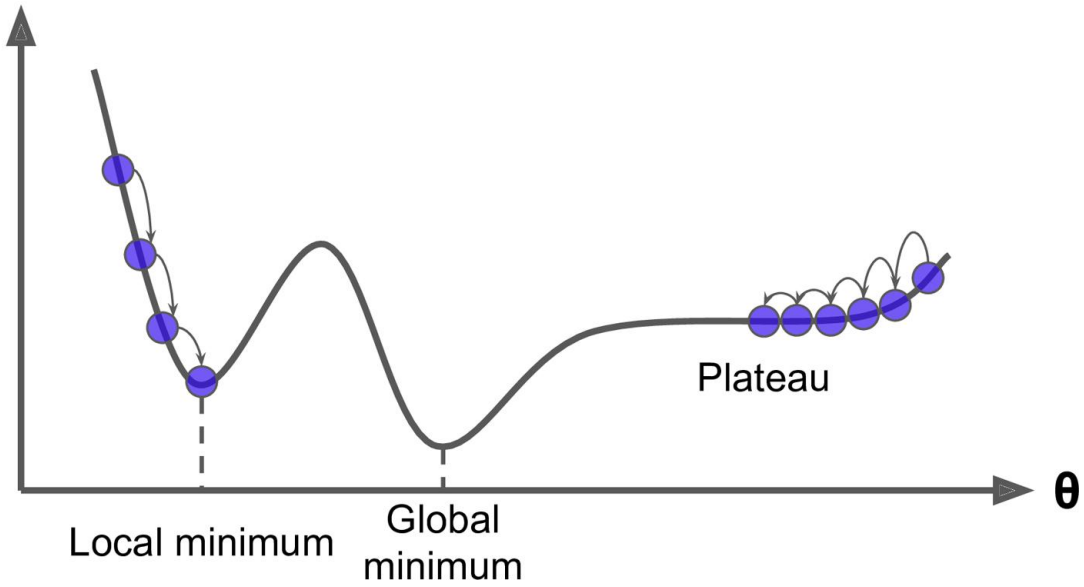


学习率的设置是门学问，一般我们会把它设置成一个比较小的正整数，0.1、0.01、0.001、0.0001，都是常见的设定数值，一般情况下学习率在整体迭代过程中是一直不变的数，但是也可以设置成随着迭代次数增多学习率逐渐变小，因为越靠近山谷我们就可以步子迈小点，省得走过，还有一些深度学习的优化算法会自己控制调整学习率这个值，后面学习过程中这些策略在讲解代码中我们会一一讲到。



全局最优解

Cost

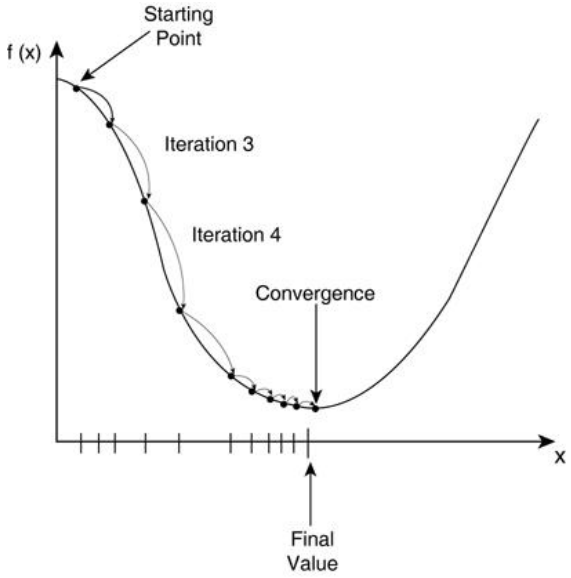


上图可以看出如果损失函数是非凸函数，梯度下降法是有可能落到局部最小值的，所以其实步长不能设置的太小太稳健，那样就很容易落入局部最优解，虽说局部最小值也没大问题，因为模型只要是堪用的就好嘛，但是我们肯定还是尽量要奔着全局最优解去。

梯度下降法流程

梯度下降法几步流程就是猜正确答案的过程：

1. 瞎蒙，Random 随机θ，随机一组数值 W0...Wn
2. 求梯度，为什么是梯度？因为梯度代表曲线某点上的切线的斜率，沿着切线往下下降就相当于沿着坡度最陡峭的方向下降
3. if g<0, theta 变大，if g>0, theta 变小
4. 判断是否收敛 convergence，如果收敛跳出迭代，如果没有达到收敛，回第 2 步继续



Question?

1. 如何随机？
2. 怎么求梯度？
3. 如何调整θ或 W？
4. 怎么判断收敛？

Answer?

1. np.random.rand()或者 np.random.randn()
2. $gradient_j = \frac{\partial Loss}{w_j}$
3. $W_i^{t+1} = W_i^t - \eta \cdot gradient_i$
4. 判断收敛这里使用 g=0 其实并不合理，因为当损失函数是非凸函数的话 g=0 有可能是极大值对吗！所以其实我们判断 loss 的下降收益更合理，当随着迭代 loss 减小的幅度即收益不再变化就可以认为停止在最低点，收敛！

总结，讲解完梯度下降法流程同学们会发现这里仍然第 2 步不是很清楚怎么去做，但是其它步骤其实都已经清楚比如用编程如何去做对吧，下面就来讲第 2 步该怎么去进一步推导出来表达式。

损失函数的导函数

接着我们来讲解如何求解上面梯度下降的第 2 步，即我们要推导出损失函数的导函数来。

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

下面公式推导 J(θ)是损失函数，θ_j 是某个特征维度 Xj 对应的权值系数，也可以写成 Wj。这里我们还是在接着之前的多元线性回归往下讲，所以损失函数是 MSE，所以下面公式表达的是因为我们的 MSE 中 X、y 是已知的，θ是未知的，而θ不是一个变量而是一堆变量，所以我们只能对含有一推变量的函数 MSE 中的一个变量求导，即偏导，下面就是对θj 求偏导。

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

x^2 的导数就是 2x，根据链式求导法则，我们可以推出第一步。然后是多元线性回归，所以 $h_\theta(x)$ 就是 $w^T x$ 也是 $w_0 \cdot x_0 + w_1 \cdot x_1 + \ldots + w_n \cdot x_n$ 亦是 $\sum_{i=0}^{n} w_i x_i$ ，到这里我们是对θj 来求偏导，那么和 Wj 没有关系的可以忽略不计，所以只剩下 Xj。

我们可以得到结论就是θj 对应的 gradient 与预测值 $\hat{y}$ 和真实值 y 有关，这里 $\hat{y}$ 和 y 是列向量，同时还与θj 对应的特征维度 Xj 有关，这里 Xj 是原始数据集矩阵的第 j 列。如果我们分别去对每个维度 W0...Wn 求偏导，即可得到所有维度对应的梯度值。

$$g_0 = (h_\theta(x) - y) \cdot X_0$$

$$g_1 = (h_\theta(x) - y) \cdot X_1$$

$$g_j = (h_\theta(x) - y) \cdot X_j$$

$$g_n = (h_\theta(x) - y) \cdot X_n$$

总结：

$$\theta_j^{t+1} = \theta_j^t - \eta \cdot g_j = \theta_j^t - \eta \cdot (h_\theta(x) - y) \cdot x_j$$

作业：

1，自行完成多元线性回归下的梯度下降法求导推导

2，这里我们可以发现 gradient 是和对应的 X 紧密相关的，而不管是哪一个维度对应的 gradient，其实 $h_\theta(x)-y$ 都是一样的，这会带来什么问题吗？

章节 2：三种梯度下降法

三种梯度下降区别和优缺点

在讲三种梯度下降区别之前，我们先来总结一下梯度下降法的步骤：

1. 瞎蒙，Random 随机θ，随机一组数值 W0...Wn
2. 求梯度，为什么是梯度？因为梯度代表曲线某点上的切线的斜率，沿着切线往下下降就相当于沿着坡度最陡峭的方向下降
3. if g<0, theta 往大调，if g>0, theta 往小调
4. 判断是否收敛 convergence，如果收敛跳出迭代，如果没有达到收敛，回第 2 步继续

四步骤对应计算方式：

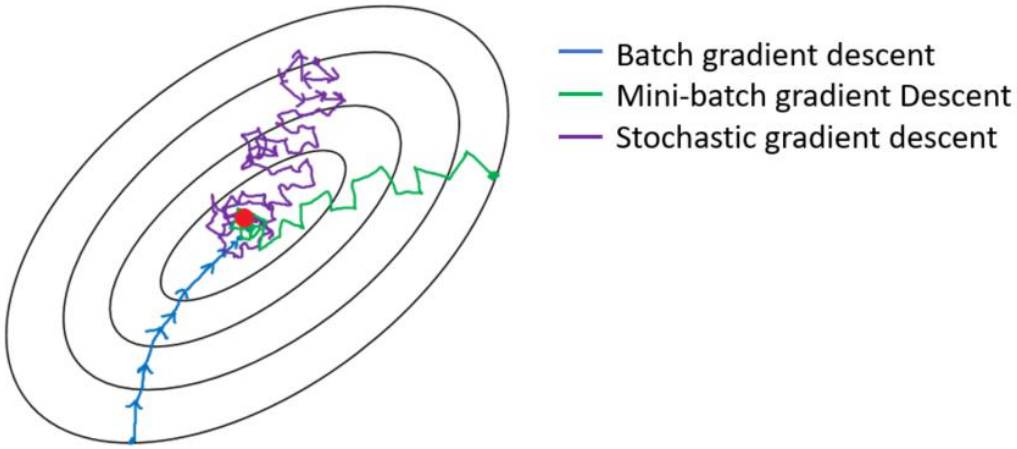
a. np.random.rand()或者 np.random.randn()

b. $gradient_i = (h_w(x)-y)\cdot x_j$

c. $W_i^{t+1} = W_i^t - \eta \cdot gradient_i$

d. 判断收敛这里使用 g=0 其实并不合理，因为当损失函数是非凸函数的话 g=0 有可能是极大值对吗！所以其实我们判断 loss 的下降收益更合理，当随着迭代 loss 减小的幅度即收益不再变化就可以认为停止在最低点，收敛！

区别：其实三种梯度下降的区别仅在于第 2 步求梯度所用到的 X 数据集的样本数量不同！它们每次学习(更新模型参数)使用的样本个数，每次更新使用不同的样本会导致每次学习的准确性和学习时间不同。

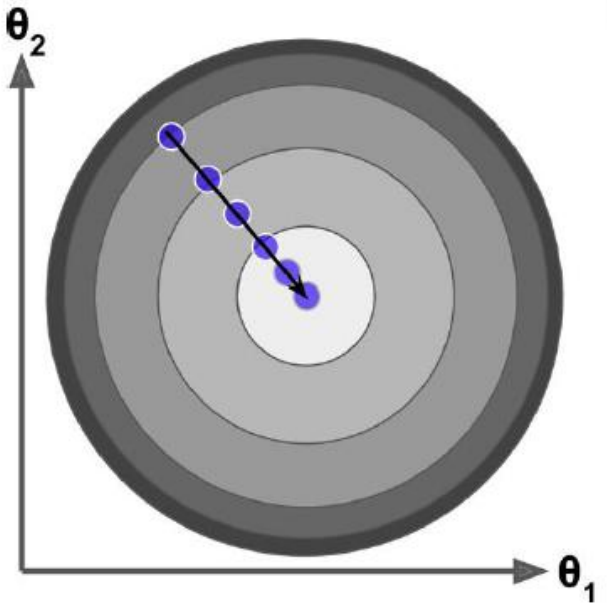


全量梯度下降

Batch Gradient Descent

$$\theta_j^{t+1} = \theta_j^{t} - \eta \cdot \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

在梯度下降中，对于θ的更新，所有的样本都有贡献，也就是参与调整θ。其计算得到的是一个标准梯度。因而理论上来说一次更新的幅度是比较大的。如果样本不多的情况下，当然是这样收敛的速度会更快啦。全量梯度下降每次学习都使用整个训练集，因此其优点在于每次更新都会朝着正确的方向进行，最后能够保证收敛于极值点(凸函数收敛于全局极值点，非凸函数可能会收敛于局部极值点)，但是其缺点在于每次学习时间过长，并且如果训练集很大以至于需要消耗大量的内存，并且全量梯度下降不能进行在线模型参数更新。



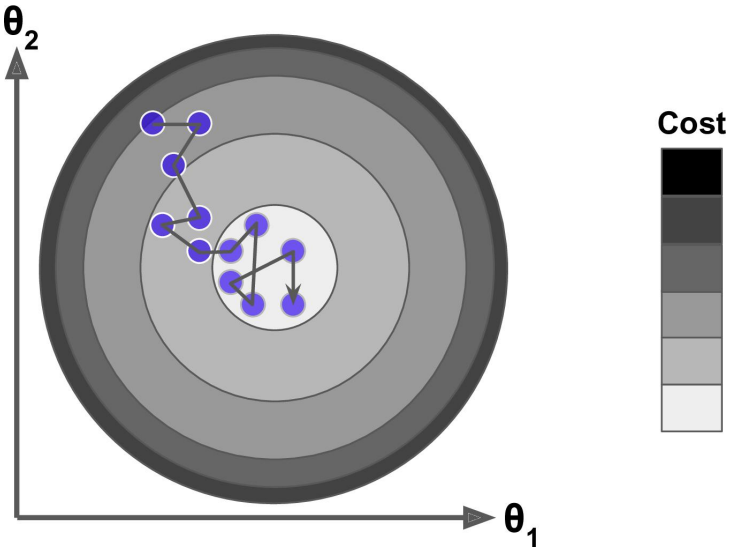
随机梯度下降

Stochastic Gradient Descent

$$\theta_j^{t+1} = \theta_j^{t} - \eta \cdot (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

梯度下降算法每次从训练集中随机选择一个样本来进行学习。批量梯度下降算法每次都会使用全部训练样本，因此这些计算是冗余的，因为每次都使用完全相同的样本集。而随机梯度下降算法每次只随机选择一个样本来更新模型参数，因此每次的学习是非常快速的，并且可以进行在线更新。随机梯度下降最大的缺点在于每次更新可能并不会按照正确的方向进行，因此可以带来优化波动(扰动)。

不过从另一个方面来看，随机梯度下降所带来的波动有个好处就是，对于类似盆地区域（即很多局部极小值点）那么这个波动的特点可能会使得优化的方向从当前的局部极小值点跳到另一个更好的局部极小值点，这样便可能对于非凸函数，最终收敛于一个较好的局部极值点，甚至全局极值点。由于波动，因此会使得迭代次数（学习次数）增多，即收敛速度变慢。不过最终其会和全量梯度下降算法一样，具有相同的收敛性，即凸函数收敛于全局极值点，非凸损失函数收敛于局部极值点。



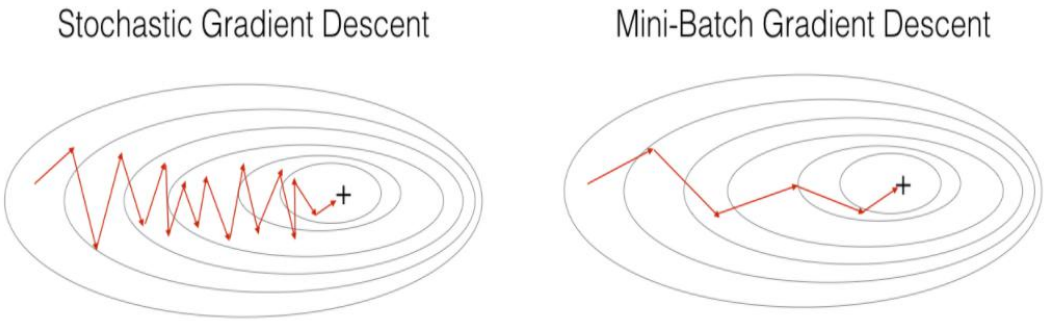
小批量梯度下降

Mini-Batch Gradient Descent

$$\theta_j^{t+1} = \theta_j^{t} - \eta \cdot \sum_{i=1}^{batch\_size} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

Mini-batch 梯度下降综合了 batch 梯度下降与 stochastic 梯度下降，在每次更新速度与更新次数中间取得一个平衡，其每次更新从训练集中随机选择 batch_size，batch_size < m 个样本进行学习。相对于随机梯度下降算法，小批量梯度下降算法降低了收敛波动性，即降低了参数更新的方差，使得更新更加稳定。相对于全量梯度下降，其提高了每次学习的

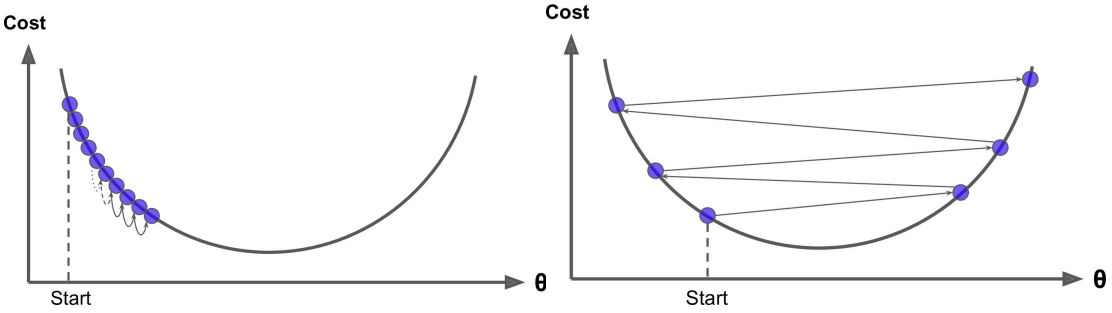
速度。并且其不用担心内存瓶颈从而可以利用矩阵运算进行高效计算。一般而言每次更新随机选择[50,256]个样本进行学习，但是也要根据具体问题而选择，实践中可以进行多次试验，选择一个更新速度与更次次数都较适合的样本数。



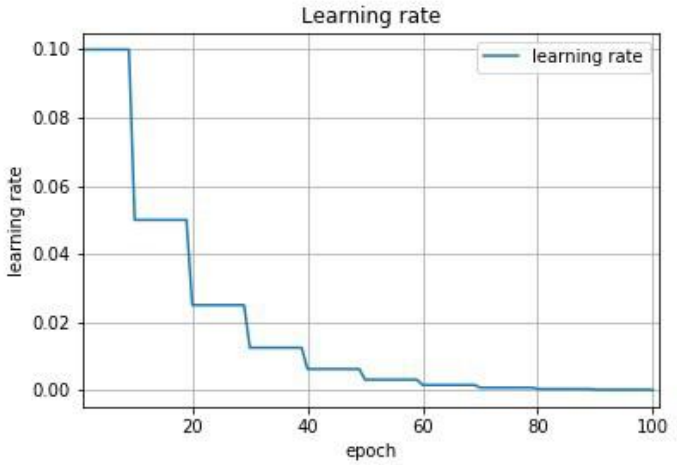
梯度下降法的问题与挑战

虽然梯度下降算法效果很好，并且广泛使用，但是不管用上面三种哪一种，都存在一些挑战与问题需要解决：

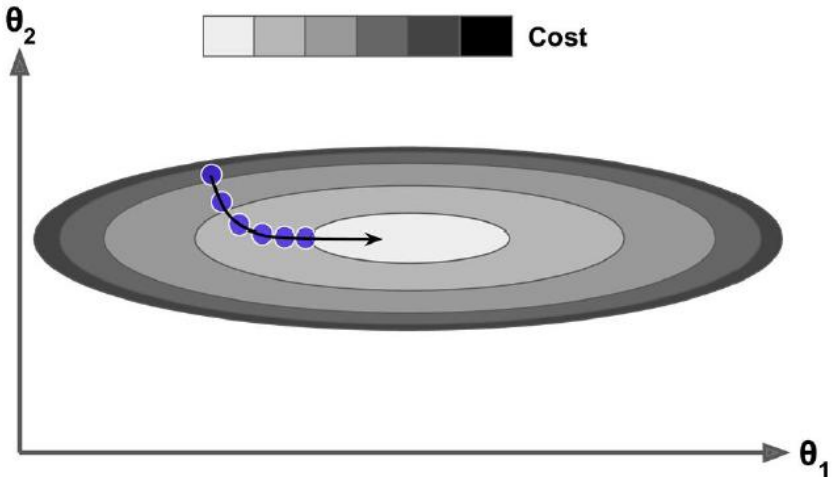
1. 选择一个合理的学习速率很难。如果学习速率过小，则会导致收敛速度很慢。如果学习速率过大，那么其会阻碍收敛，即在极值点附近会振荡。



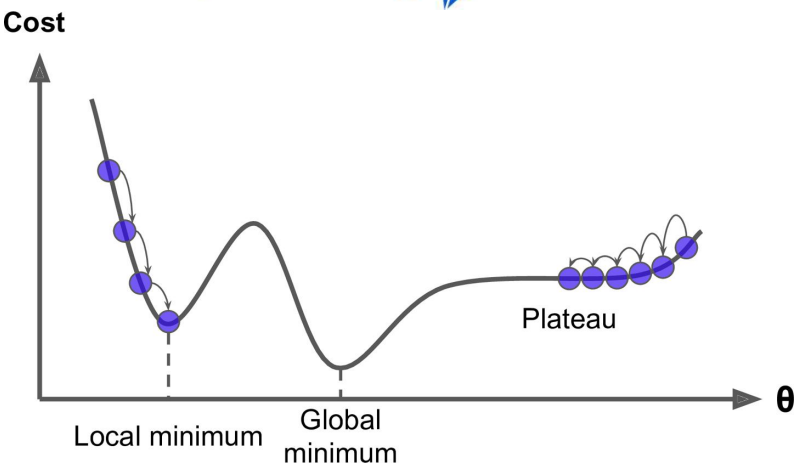
2. 学习速率调整(又称学习速率调度，Learning rate schedules 试图在每次更新过程中，改变学习速率，如退火。一般使用某种事先设定的策略或者在每次迭代中衰减一个较小的阈值。无论哪种调整方法，都需要事先进行固定设置，这边便无法自适应每次学习的数据集特点。



3. 模型所有的参数每次更新都是使用相同的学习速率。如果数据特征是稀疏的或者每个特征有着不同的取值统计特征与空间 ,那么便不能在每次更新中每个参数使用相同的学习速率，那些很少出现的特征应该使用一个相对较大的学习速率。



4. 对于非凸目标函数，容易陷入那些次优的局部极值点中，如在神经网路中。那么如何避免呢。Dauphin 指出更严重的问题不是局部极值点，而是鞍点(These saddle points are usually surrounded by a plateau of the same error, which makes it notoriously hard for SGD to escape, as the gradient is close to zero in all dimensions.)



轮次和批次

轮次：epoch，轮次顾名思义是把我们已有的训练集数据学习多少轮

批次：batch，批次这里指的的我们已有的训练集数据比较多的时候，一轮要学习太多数据，那就把一轮次要学习的数据分成多个批次，一批一批数据的学习

举个例子，这就好比有一本唐诗 300 首需要大家背诵，如果要给你一周时间要你背诵完，或许你很聪明可以背完，但是估计也不敢说一下子全都记得特别牢，甚至到可以出口成章的地步吧。那通常人是怎么做的呢？是不是就是接下来多花几周，重复之前一周的动作，把 300 首唐诗多背几次，比如花 10 周把 300 首唐诗全部倒背如流了，然后又花了 10 周时间把 300 首唐诗又背诵到了滚瓜烂熟的地步终于可以出口成唐诗了。那么刚提到的 10 周、又 10 周放在 AI 领域那么就是所谓的训练了 10 个轮次又 10 个轮次，总共 20 个轮次。

再回顾上面一段话，我们假设你很聪明，也就是 AI 训练的电脑性能处理能力非常好，如果是普通人或者一般的电脑，很有可能一周都不可能背诵完 300 首，也就是内存一下子装不下那么大的数据量，或者处理器计算能力没有那么快。当数据量大的情况下，这是很常见的现在，那么为了可以顺利背下来 300 首唐诗到举一反三出口成章的地方，也就是为了训练出来 model 模型，我们只能把一个轮次需要的数据分成多个批次来一点点计算，放到背唐诗的例子中，说白了就是一周背不下来 300 首唐诗，那么我们就一周背 50 首唐诗吧比如说，这样我们就需要 6 个批次把一轮数据背完，一个批次所需要的数据 batch_size 就是 50。

还有就是对于三种梯度下降来说，全量梯度下降就是每一轮次用到全量的数据，然后一次迭代就是一个轮次，然后用全量数据计算梯度来更新一下 W。随机梯度下降每一个轮次也需要计算所有数据，但是有多少数据就会分为多少个批次，即是一个批次一次迭代，就只用一条数据计算梯度来更新一下 W，所以随机梯度下降一个轮次中的更新 W 的次数等于样本总数。最后就是 mini-batch 梯度下降，每一个轮次也需要计算所有数据，但是轮次分成多少个批次取决于 batch_size，batch_size 大需要的轮次就少，比如 batch_size=num_samples，那就等价于全量批量梯度下降，batch_size=1 那就等价于随机梯度下降。

num_batches=math.ceil(num_samples/batch_size)

章节 3：代码实战梯度下降法与优化

全量梯度下降

创建数据集 X、y

```
np.random.seed(1)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
X_b = np.c_[np.ones((100, 1)), X]
```

创建超参数

```
learning_rate = 0.001
n_iterations = 10000
```

1，初始化 theta，w0...wn，正太分布创建 W

```
theta = np.random.randn(2, 1)
```

这里 range()函数在 python 中可以返回给我们一个可迭代的对象

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
# 4，不会设置阈值，直接设置超参数，迭代次数，迭代次数到了，我们就认为收敛了
for _ in range(n_iterations):
    # 2，接着求梯度 gradient
    gradients = X_b.T.dot(X_b.dot(theta)-y)
    # 3，应用公式调整 theta 值，theta_t + 1 = theta_t - grad * learning_rate
    theta = theta - learning_rate * gradients
```

上面代码值得一提的是，X_b.T 是 X 的转置，也就是 n 行 m 列的矩阵，然后(X_b.dot(theta)-y)是一个 m 行 1 列的列向量，这样矩阵乘以向量会得到一个 n 行一列的列向量，相当于一下子计算出来所有维度对应的梯度值。然后第 3 步直接去应用所有的梯度到所有的 theta 上面。

运行结果：

```
[[4.23695725]
 [2.84246254]]
```

这里同学们可以调整一下迭代次数或者学习率看看计算结果是否有不同？

随机梯度下降

创建数据集 X、y

```
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
X_b = np.c_[np.ones((100, 1)), X]
```

设置超参数，n_epochs 是迭代的轮次，m 是样本总数，也就是每个轮次迭代的批次

```
n_epochs = 10000
m = 100
learning_rate = 0.001
```

1，初始化 theta，w0...wn，正太分布创建 W

```
theta = np.random.randn(2, 1)
```

这里是双层 for 循环，第一层 for 循环是分轮次来计算，第二层 for 循环是分批次来计算 random_index = np.random.randint(m)是随机的从数据集中取一条数据的索引，然后根据索引来切片操作取随机出来数据的 Xi 和 yi ,每次 W 更新所需要的梯度的求解只用到一条样本的信息。

```
for epoch in range(n_epochs):
    for i in range(m):
        # 2, 求 gradient Xi.T * (Xi * theta - yi)
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = xi.T.dot(xi.dot(theta)-yi)
        # 3, 调整 theta
        theta = theta - learning_rate * gradients
```

运行结果：

```
[[3.97093672]
 [2.96575641]]
```

小批量梯度下降

创建数据集 X、y

```
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
X_b = np.c_[np.ones((100, 1)), X]
```

设置超参数，n_epochs 是迭代的轮次，m 是样本总数，batch_size 是每个批次迭代需要的数据量，然后 num_batches 是一轮需要多少个迭代需要相除再向上取整

```
learning_rate = 0.001
```

```
n_epochs = 10000
m = 100
batch_size = 10
num_batches = int(m/batch_size)
```

1，初始化 theta，w0...wn，正太分布创建 W

```
theta = np.random.randn(2, 1)
```

和之前代码差别无非就是切片取出来的是 batch_size 个样本数据，然后求解梯度而已

```
for epoch in range(n_epochs):
    for i in range(num_batches):
        # 2, 求 gradient Xi.T * (Xi * theta - yi)
        random_index = np.random.randint(m)
        x_batch = X_b[random_index:random_index+batch_size]
        y_batch = y[random_index:random_index+batch_size]
        gradients = x_batch.T.dot(x_batch.dot(theta)-y_batch)
        # 3, 调整 theta
        theta = theta - learning_rate * gradients
```

思考题 1

这里的随机梯度下降和小批量梯度下降的随机方式，会不会有可能有些数据一直都取不到的可能？代码如何操作解决这个问题？

在双层 for 循环之间，也就是每个轮次开始分批次迭代之前打乱数据索引顺序

```
arr = np.arange(len(X))
np.random.shuffle(arr)
X_b = X_b[arr]
y = y[arr]
```

取值的时候按照顺序依次切片操作即可

```
x_batch = X_b[i*batch_size: i*batch_size + batch_size]
y_train = y[i*batch_size: i*batch_size + batch_size]
```

思考题 2

如何可以通过代码让学习率随着迭代次数增多而逐渐变小？

初始学习率等于 5/500

```
t0, t1 = 5, 500  # 超参数
```

定义调整学习率的函数

```
def learning_schedule(t):
```

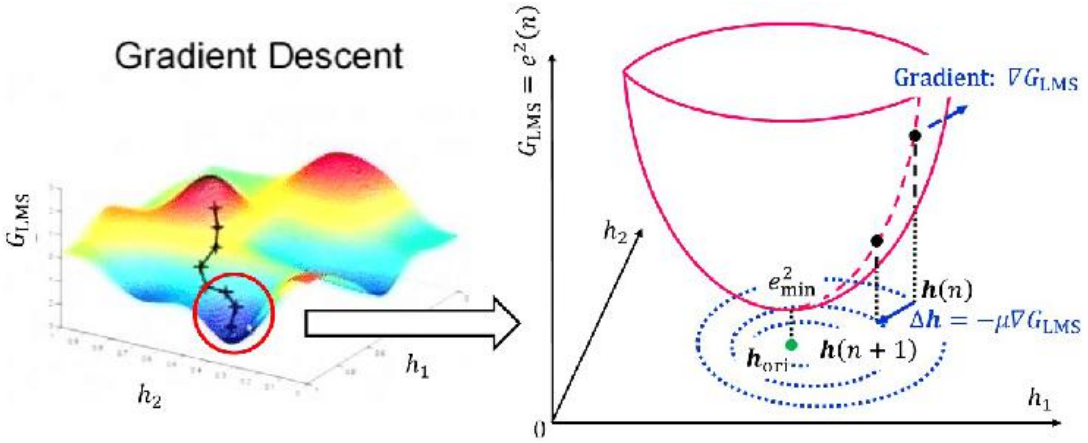
```
    return t0 / (t + t1)
```

在更新 W 参数前随着 epoch 调整学习率

```
learning_rate = learning_schedule(epoch*m + i)
```

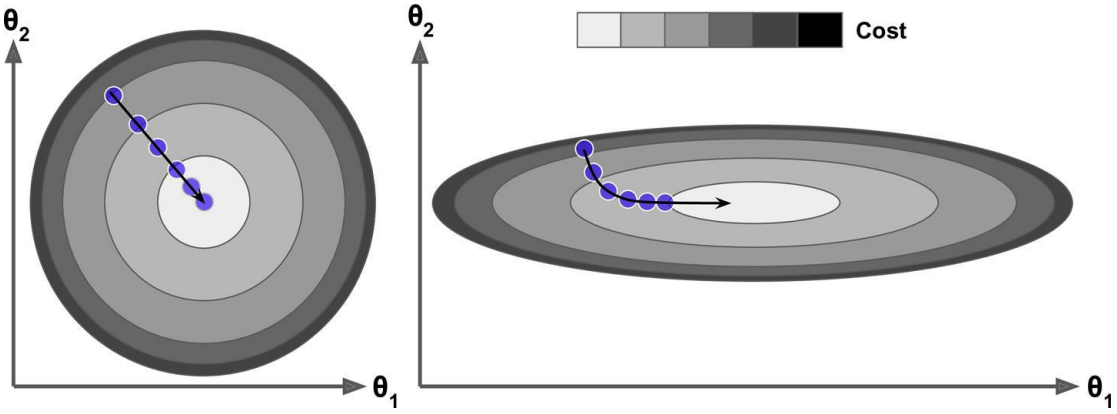
章节 4：归一化 normalization

归一化的目的



提到归一化，还是少不了梯度下降，如果维度多了，就是超平面，很难去画出来了，所以我们选择只有两个维度的情况，那么我们可以把损失函数看作是山峰山谷，如果拿多元线性回归举例的话，因为多元线性回归的损失函数 MSE 是凸函数，所以我们可以把损失函数看成是一个碗。然后下面的图就是从碗上方去俯瞰！哪里是损失最小的地方呢？当然对应的就是碗底的地方！所以下图碗中心的地方颜色较浅的区域就是 cost 损失较小的地方。

下图左是做了归一化的俯瞰图，下图右是没有做归一化的俯瞰图。



我们先来说一下为什么没做归一化是右侧图示，举个例子假如我们有一人客户信息，然后里面有两个维度，一个是用户的年龄，一个是用户的月收入，不管目标变量是什么多元线

性回归的式子我们可以里面写出来，不考虑截距项 $y=\theta_1 X_1+\theta_2 X_2$ ，那么这样每一条样本的不同维度对应的数量级不同，原因是每个维度对应的物理含义不同嘛，但是计算机能理解这 25 和 10000 分别是年龄和收入吗？计算机只是拿到一堆数字。

name	age	income
张三	25	10000
李四	36	50000
王五	49	30000
...	...	...

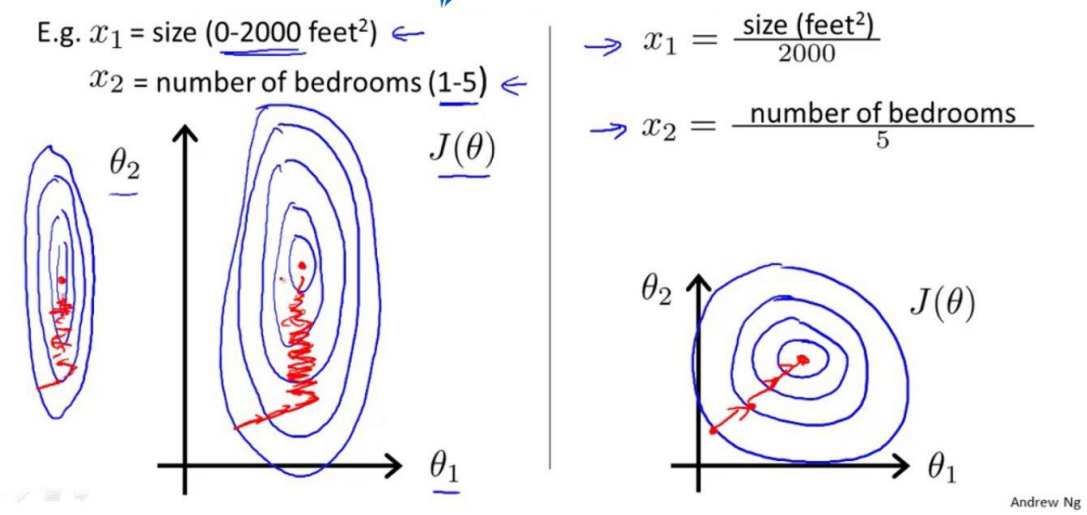
我们把 X1 看成是年龄，X2 看成是收入，同时我们是知道 y 的，机器学习就是知道 X,y 的情况下解方程组调整出最优解的过程嘛。根据公式我们也可以发现 y 是两部分贡献之和，按常理来说，一开始并不知道两个部分谁更重要的情况下，可以想象为两部分对 y 的贡献是一样的即 $\theta_1 X_1=\theta_2 X_2$ ，如果 $X_1<<X_2$ ，那么最终 $\theta_1^{'}>>\theta_2^{'}$ 。

这样是不是就比较好理解为什么之前图里面右图为什么会画的 θ_1 这个轴比 θ_2 这个轴要长一些了。再思考一下，我们在梯度下降第 1 步的时候是不是所有的维度θ都是根据在期望 μ为 0 方差σ为 1 的正太分布随机在 0 附近的 ,说白了就是一开始的 θ_1 和 θ_2 数值是差不多的。所以可以发现 θ_1 从初始值到目标位置 $\theta_1^{'}$ 的距离要远大于 θ_2 从初始值到目标位置 $\theta_2^{'}$ 。

再者，根据前面的 $X_1<<X_2$ ，我们可以知道梯度是和 X 有关的，所以根据梯度公式 $g_j=(h(\theta)-y)\cdot Xj$ 可以推出 $g_1<<g_2$ 。根据梯度下降公式 $\theta_1^{t+1}=\theta_1^{t}-\eta\cdot g_1$ $\theta_2^{t+1}=\theta_2^{t}-\eta\cdot g_2$ ，可以推出 $|\theta_1^{t+1}-\theta_1^{t}|=|\eta\cdot g_1|$ $|\theta_2^{t+1}-\theta_2^{t}|=|\eta\cdot g_2|$ ，也可以说每次调整 θ_1 的幅度要远小于 θ_2 的调整幅度。

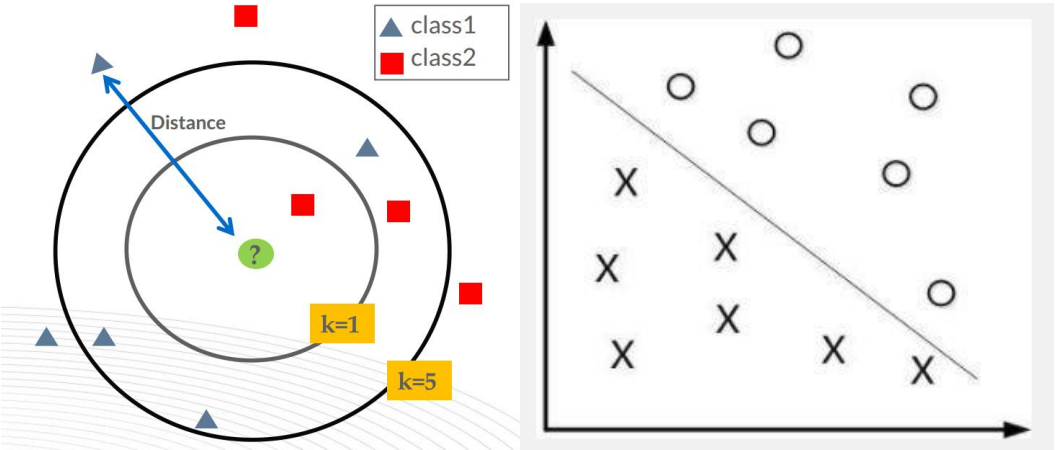
总结，根据上面得到的两个结论，我们可以发现它们互相之间是矛盾的，意味着最后 θ_2 需要比 θ_1 更少的迭代次数就可以收敛，而我们要最终求得最优解，就必须每个维度θ都收敛才可以，所以会出现 θ_2 等待 θ_1 收敛的情况。讲到这里对应图大家应该可以理解为什么右图是先顺着 θ_2 的坐标轴往下走再往右走的原因了吧。

结论：归一化的一个目的是使得最终梯度下降的时候可以不同维度θ参数可以在接近的调整幅度上。这就好比社会主义，可以一小部分先富裕起来，至少 loss 整体可以下降，然后只是会等另外一部分人富裕起来。但是更好的是要实现共同富裕 ,最后每个人都不能落下，优化的步伐是一致的。



扩展

归一化的其它好处是有可能提高精度！一些分类器需要计算样本之间的距离(如欧氏距离)，例如 KNN。如果一个特征值域范围非常大，那么距离计算就主要取决于这个特征，从而与实际情况相悖 (比如这时实际情况是值域范围小的特征更重要)。同样的再举例分类器的例子，由于特征维度的量纲不同，所以很可能 X1 在 10000 到 20000 的区间，X2 在 1 到 2 的区间，这样所有点都在一条直线上，出现无法找到分界线的情况。



归一化的本质

前面一小节我们知道了做归一化的目的是要共同富裕 ,而之所以梯度下降优化时不能达到步调一致的根本原因其实还是 X1 和 X2 的数量级不同。所以什么是归一化？

答案自然就出来了，就是把 X1 和 X2 的数量级给它统一，扩展一点说，如果有更多特征维度，就要把各个特征维度 X1...Xn 的数量级统一，来做到无量纲化。

那接下来怎么去做归一化的问题，就是怎么去把数据中的 X1...Xn 的数量级统一。

最大值最小值归一化

min max scaling

$$x_{i,j}^{*} = \frac{x_{i,j} - x_j^{min}}{x_j^{max} - x_j^{min}}$$

这里 x_j^{min} 是对应 X 矩阵中第 j 列特征值的最小值，x_j^{max} 是对应 X 矩阵中第 j 列特征值的最大值，$x_{i,j}$ 是 X 矩阵中第 i 行第 j 列的数值，$x_{i,j}^{*}$ 是归一化之后的 X 矩阵中第 i 行第 j 列的数值。

举个例子，比如第 j 列的数值是[1, 2, 3, 5, 5]，x_j^{min} 就是 1，x_j^{max} 就是 5，那么归一化之后是[0, 0.25, 0.75, 1, 1]。如果第 j 列的数值是[1, 2, 3, 5, 50001]，那么归一化之后是[0, 0.00004, 0.00006, 0.0001, 1]。同学们会发现什么规律吗？

其实我们很容易发现使用最大值最小值归一化的时候，优点是一定可以把数值归一到 0 到 1 之间，缺点是如果有一个离群值，正如我们举的例子一样，会使得一个数值为 1，其它数值都几乎为 0，所以受离群值的影响比较大。

```
from sklearn.preprocessing import MinMaxScaler
```

标准归一化

通常标准归一化中包含了均值归一化和方差归一化。经过处理的数据符合标准正态分布，即均值为 0，标准差为 1，其转化函数为：

$$X_{new} = \frac{X_i - X_{mean}}{\text{Standard Deviation}}$$

其中μ为所有样本数据的均值，σ为所有样本数据的标准差。

$$Mean(population) = \mu = \frac{\sum_{i=1}^{k} f_i x_i}{n}$$

$$StandardDeviation(population) = \sigma = \sqrt{\sum_{i=1}^{k} \frac{f_i (x_i - \mu)^2}{n}}$$

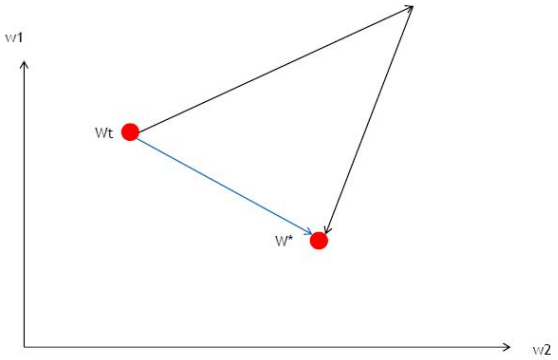
$$Variance(population) = \sigma^2 = \sum_{i=1}^{k} \frac{f_i (x_i - \mu)^2}{n}$$

相对于最大值最小值归一化来说，因为标准归一化是除以的是标准差，而标准差的计算会考虑到所有样本数据，所以受到离群值的影响会小一些，这就是除以方差的好处！但是如果是使用标准归一化不一定会把数据缩放到 0 到 1 之间了。

那为什么要减去均值呢？其实做均值归一化还有一个特殊的好处，我们来看下面梯度下

降法的式子 ,我们会发现α是正数 ,不管 A 是正还是负 ,比如多元线性回归的话 A 就是 $\hat{y}-y$ ，对于所有的维度 X，比如这里的 x_{i1} 和 x_{i2} 来说α乘上 A 都是一样的符号，那么每次迭代的时候 w_0^{t+1} 和 w_1^{t+1} 的更新幅度符号也必然是一样的，这样就会像下图有右图所以，要想从 w^t 更新到 w^* 就必然要么 W1 和 W2 同时变大再同时变小 ,或者就 W1 和 W2 同时变小再同时变大。不能如图上所示蓝色的最优解路径，即 W1 变小的时候 W2 变大。

$$\begin{pmatrix} w_0^{t+1} \\ w_1^{t+1} \\ w_2^{t+1} \end{pmatrix} = \begin{pmatrix} w_0^{t} \\ w_1^{t} \\ w_2^{t} \end{pmatrix} + \alpha A \begin{pmatrix} 1 \\ x_{i1} \\ x_{i2} \end{pmatrix}$$



那我们如何才能做到让 W1 变小的时候 W2 变大呢？归其根本还是大部分数据集 X 矩阵中的数据均为正数，同学们可以想想比如人的信息，年龄、收入等不都是正数值嘛，公式中 x_{i1} 和 x_{i2} 如果都是正数的话就会出问题，所以如果我们可以让 x_{i1} 和 x_{i2} 它们符号不同，比如有正有负 ,其实就可以在做梯度下降的时候有更多的可能性去让更新尽可能沿着最优解路径去做。

```
from sklearn.preprocessing import StandardScaler
```

```
>>> from sklearn.preprocessing import StandardScaler
>>> data = [[1], [2], [3], [5], [5]]
>>> scaler=StandardScaler()
>>> print(scaler.fit(data))
StandardScaler(copy=True, with_mean=True, with_std=True)
>>> print(scaler.mean_)
[3.2]
>>> print(scaler.scale_)
[1.6]
>>> print(scaler.var_)
[2.56]
>>> print(scaler.transform(data))
[[-1.375]
 [-0.75 ]
 [-0.125]
 [ 1.125]
 [ 1.125]]
```

```
>>> from sklearn.preprocessing import StandardScaler
>>> data = [[1],[2],[3],[5],[50001]]
>>> scaler=StandardScaler()
>>> print(scaler.fit(data))
StandardScaler(copy=True, with_mean=True, with_std=True)
>>> print(scaler.mean_)
[10002.4]
>>> print(scaler.scale_)
[19999.30004375]
>>> print(scaler.var_)
[3.99972002e+08]
>>> print(scaler.transform(data))
[[-0.5000875 ]
 [-0.5000375 ]
 [-0.4999875 ]
 [-0.49988749]
 [ 2.        ]]
```

强调

我们在做特征工程的时候，很多时候如果对训练集的数据进行了预处理，比如这里讲的归一化，那么未来对测试集的时候，和模型上线来新的数据的时候，都要进行相同的数据预处理流程，而且所使用的均值和方差是来自当时训练集的均值和方差！

因为我们人工智能要干的事情就是从训练集数据中找规律，然后利用找到的规律去预测未来。这也就是说假设训练集和测试集以及未来新来的数据是属于同分布的！从代码上面来说如何去使用训练集的均值和方差呢？如果是上面代码的话，就需要把 scaler 对象持久化，回头模型上线的时候再加载进来去对新来的数据使用。

```
>>> print(scaler.transform([[4.8]]))
[[-0.4998975]]
```

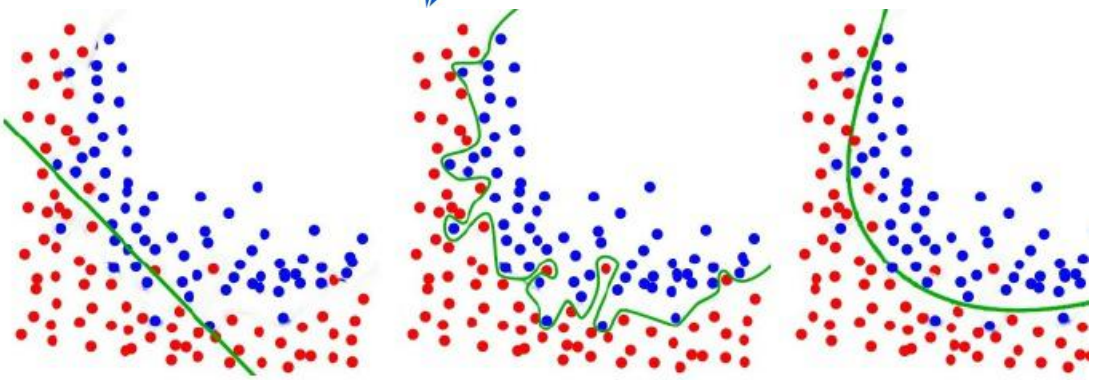
章节 5：正则化 regularization

过拟合和欠拟合

(1) under fit：还没有拟合到位，训练集和测试集的准确率都还没有到达最高。学的还不到位。

(2) over fit：拟合过度，训练集的准确率升高的同时，测试集的准确率反而降低。学的过度了，做过的卷子都能再次答对，考试碰到新的没见过的题就考不好。

(3) just right：过拟合前训练集和测试集准确率都达到最高时刻。学习并不需要花费很多时间，理解的很好，考试的时候可以很好的把知识举一反三。真正工作中我们是奔着过拟合的状态去调的，但是最后要的模型肯定是没有过拟合的。



(1) (2) (3)

正则化就是防止过拟合，增加模型的鲁棒性 robust，鲁棒是 Robust 的音译，也就是强壮的意思。就像计算机软件在面临攻击、网络过载等情况下能够不死机不崩溃，这就是该软件的鲁棒性。鲁棒性调优就是让模型拥有更好的鲁棒性，也就是让模型的泛化能力和推广能力更加的强大。举以下例子：下面两个式子描述同一条直线那个更好？

$$0.5x_1 + 0.4x_2 + 0.3=0$$

$$5x_1 + 4x_2 + 3=0$$

第一个更好，因为下面的公式是上面的十倍，当 w 越小公式的容错的能力就越好。因为把测试集带入公式中如果测试集原来是 100 在带入的时候发生了一些偏差，比如说变成了 101，第二个模型结果就会比第一个模型结果的偏差大的多。公式中 $\hat{y} = w^T x$ 当 x 有一点错误，这个错误会通过 w 放大从而影响 z。但是 w 不能太小，当 w 太小时正确率就无法保证，就没法做分类。想要有一定的容错率又要保证正确率就要由正则项来决定。

所以正则化（鲁棒性调优）的本质就是牺牲模型在训练集上的正确率来提高推广能力，W 在数值上越小越好，这样能抵抗数值的扰动。同时为了保证模型的正确率 W 又不能极小。故而人们将原来的损失函数加上一个惩罚项，这里面损失函数就是原来固有的损失函数，比如回归的话通常是 MSE，分类的话通常是 cross entropy 交叉熵，然后在加上一部分惩罚项来使得计算出来的模型 W 相对小一些来带来泛化能力。

常用的惩罚项有 L1 正则项或者 L2 正则项

$$L_1 = \sum_{i=0}^{m} |w_i|$$

$$L_2 = \sum_{i=0}^{m} w_i^2$$

其实 L1 和 L2 正则的公式数学里面的意义就是范数，代表空间中向量到原点的距离

1、 L-P范数

与闵可夫斯基距离的定义一样，L-P范数不是一个范数，而是一组范数，其定义如下：

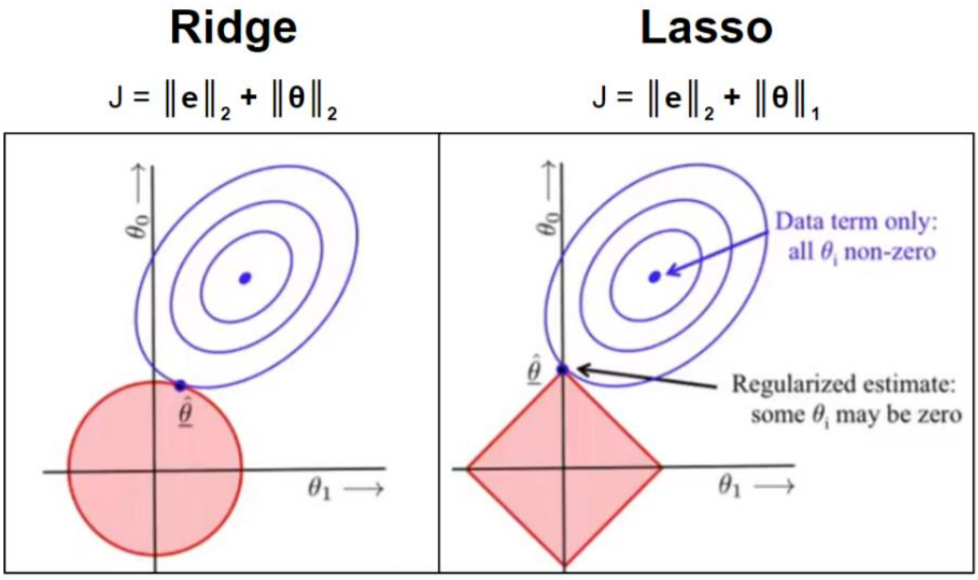
$$L_p = \|\mathbf{x}\|_p = \sqrt[p]{\sum_{i=1}^{n} x_i^p},\ \mathbf{x} = (x_1, x_2, \cdots, x_n)$$

根据P 的变化，范数也有着不同的变化，一个经典的有关P范数的变化图如下：

上图表示了p从无穷到0变化时，三维空间中到原点的距离（范数）为1的点构成的图形的变化情况。以常见的L-2范数（p=2）为例，此时的范数也即欧氏距离，空间中到原点的欧氏距离为1的点构成了一个球面。

实际上，在0时，Lp并不满足三角不等式的性质，也就不是严格意义下的范数。以p=0.5，二维坐标(1,4)、(4,1)、(1,9)为例，$\sqrt[0.5]{1+\sqrt{4}}+\sqrt[0.5]{1+\sqrt{4}}<\sqrt[0.5]{1+\sqrt{9}}$。因此这里的L-P范数只是一个概念上的宽泛说法。

当我们把多元线性回归损失函数加上 L2 正则的时候，就诞生了 Ridge 岭回归。当我们把多元线性回归损失函数加上 L1 正则的时候，就孕育出来了 Lasso 回归。其实 L1 和 L2 正则项惩罚项可以加到任何算法的损失函数上面去提高计算出来模型的泛化能力的。



L1 稀疏 L2 平滑

当我们看到上图时，其实里面学问很大，通常我们会说 L1 正则会使得计算出来的模型有的 W 趋近于 0，有的 W 相对较大，而 L2 会使得 W 参数整体变小，这是为什么呢？这得从梯度下降更新得角度去考虑

如果损失函数加上正则项，那么导函数就等于多了正则项的导函数，即原来比如 MSE 的导函数和 L1、L2 的导函数，那么梯度 gradient 就是这两部分组成，每次减小的幅度就是学习率η×(MSE 的导+L1/L2 的导)，说白了就是比不加惩罚项要多更新一部分，这部分就使得 W 奔着原点 0 去。

$$\theta^* = argmin_\theta \left(\sum_i \|f(X_i) - Y_i\|^2 + \lambda \sum_i |\theta_i| \right)$$

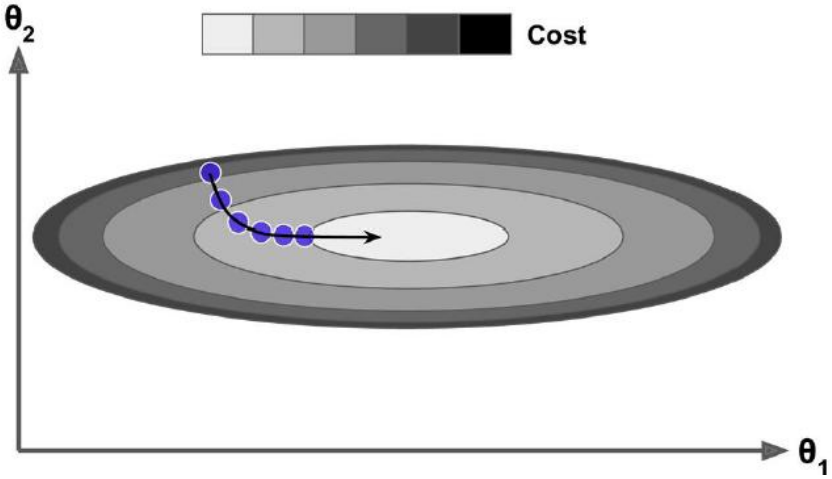
上式中λ是正则项系数，λ越大，说明我们算法工程师越看重模型的泛化能力，经验值是设置 0.4，也可以看成是保障正确率的 base loss 的权值是 1，保障泛化能力的正则项权值是 0.4。既然现在要去最小化的整体损失函数变成了两部分加和，那么θ就既得满足原有的损失函数比如 MSE，同时还得满足 L1 或者 L2 这部分，从图上来说就是θ得在紫色的损失函数等高线上同时又得在红色的 L1 或 L2 的等高线上，说白了就是要出现在交点上面。因为更高维度空间的图形很难去绘制和观看，所以这里选择两个特征维度对应θ1 和θ2 两个参数的图来解释。

以下公式就是 L1 或者 L2 的导函数，也就是每次梯度下降会多走的幅度，我们可以看到如果是 L1 每次会多走η*1 的幅度，我们知道η学习率一开始设置一个常数那么每次多走的幅度对应不同的维度来说是一样的一个常数，学习率是 0.5 的话，比如θ1 多走 0.5，θ2 也是多走 0.5。反观 L2 每次多走的幅度是之前的 2 倍的 w_i^t，学习率是 0.5 的话，那么就θ1 就多走 θ_1^t，θ2 就多走 θ_2^t。

$$L_1 = |w_1| + |w_2| + ... + |w_n|, \quad \frac{\partial L_1}{\partial w_i} = sign(w_i) \quad = 1 \text{ or } -1$$

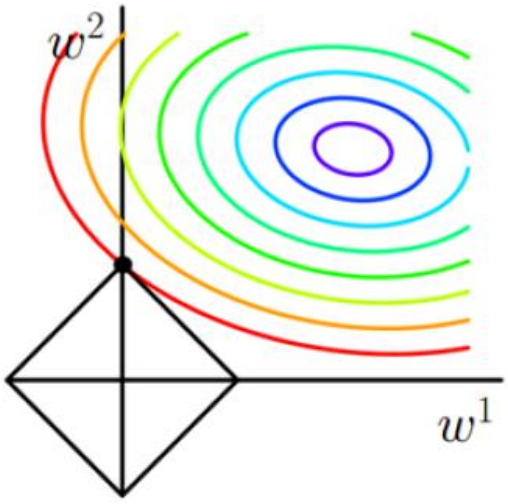
$$L_2 = \frac{1}{2}w_1^2 + w_2^2 + ... + w_n^2, \quad \frac{\partial L_2}{\partial w_i} = w_i$$

故事还没有结束，同学们根据前面讲过的知识，知道下图意味着θ1 的数量级比θ2 的数量级大，但是θ2 比θ1 每次更新的幅度更大，也就是说如果是加了 L1 正则项，θ2 比θ1 每次一样还是更新的幅度更大。但是如果是加了 L2 正则项，因为θ1 的数量级比θ2 的数量级大，所以每次θ1 会多走的幅度是 0.5* θ_1^t 要远大于 0.5* θ_2^t。

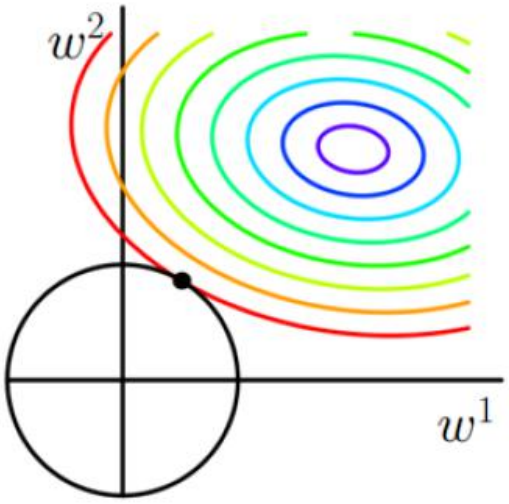


综上所述我们再来理解下图并不难，L1 更容易相交于坐标轴上，而 L2 更容易相交于非坐标轴上。如果相交于坐标轴上，如图 L1 就使得是 W2 非 0，W1 是 0，这个就体现出 L1 的稀疏性。如果没相交于坐标轴，那 L2 就使得 W 整体变小。通常我们为了去提高模型的泛化能力 L1 和 L2 都可以使用。L1 的稀疏性在做机器学习的时候，还有一个副产品就是可

以帮忙去做特征的选择。



(a) ℓ_1-ball meets quadratic function.



(b) ℓ_2-ball meets quadratic function.

章节 6：多元线性回归的衍生算法

Ridge Lasso ElasticNet

Ridge

$$\min_w ||Xw - y||_2^2 + \alpha||w||_2^2$$

同样从 sklearn.linear_model 下导入，并且创建 X,y 数据集

```
import numpy as np
from sklearn.linear_model import Ridge
from sklearn.linear_model import SGDRegressor

X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

这里 alpha 设置为 0.4 对应的就是公式中的α正则项系数，solver=’sag’就是使用随机梯度下降法来求解最优解

```
ridge_reg = Ridge(alpha=0.4, solver='sag')
ridge_reg.fit(X, y)
```

```
print(ridge_reg.predict(1.5))
print(ridge_reg.intercept_)
print(ridge_reg.coef_)
```

源码解析

```
class Ridge(_BaseRidge, RegressorMixin):
    """Linear least squares with l2 regularization.

    Minimizes the objective function::

    ||y - Xw||^2_2 + alpha * ||w||^2_2

    This model solves a regression model where the loss function is
    the linear least squares function and regularization is given by
    the l2-norm. Also known as Ridge Regression or Tikhonov regularization.
    This estimator has built-in support for multi-variate regression
    (i.e., when y is a 2d-array of shape [n_samples, n_targets]).
```

alpha：正则项系数

fit_intercept：是否计算 W0 截距项

normalize：是否做归一化

max_iter：最大迭代次数

solver：优化算法的选择

运行结果（α=0.4）

```
[[8.51613892]]
[3.70210608]
[[3.20935523]]
```

对于结果同学们可以通过调整α的值来观察是不是α变大的同时 W 会变小

运行结果（α=4）

```
[[8.14400395]]
[4.46628333]
[[2.45181375]]
```

运行结果（α=400）

```
[[7.07379569]]
[6.76562244]
[[0.20544883]]
```

SGDRegressor 来完成类似的操作

```
sgd_reg = SGDRegressor(penalty='l2', max_iter=1000)
sgd_reg.fit(X, y.ravel())
print(sgd_reg.predict([[1.5]]))
print("W0=", sgd_reg.intercept_)
print("W1=", sgd_reg.coef_)
```

运行结果

```
[8.50077452]
W0= [4.28171675]
W1= [2.81270518]
```

这里 ravel()操作是扁平化

Examples

It is equivalent to `reshape(-1, order=order)`.

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.ravel(x)
array([1, 2, 3, 4, 5, 6])
```

```
>>> x.reshape(-1)
array([1, 2, 3, 4, 5, 6])
```

Lasso

$$\min_{w} \frac{1}{2n_{\text{samples}}} ||Xw - y||_2^2 + \alpha ||w||_1$$

```
import numpy as np
from sklearn.linear_model import Lasso
from sklearn.linear_model import SGDRegressor

X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

lasso_reg = Lasso(alpha=0.15, max_iter=30000)
lasso_reg.fit(X, y)
print(lasso_reg.predict(1.5))
print(lasso_reg.intercept_)
print(lasso_reg.coef_)

sgd_reg = SGDRegressor(penalty='l1', max_iter=10000)
sgd_reg.fit(X, y.ravel())
print(sgd_reg.predict(1.5))
print(sgd_reg.intercept_)
print(sgd_reg.coef_)
```

ElasticNet

$$\min_w \frac{1}{2n_{\text{samples}}}||Xw-y||_2^2+\alpha\rho||w||_1+\frac{\alpha(1-\rho)}{2}||w||_2^2$$

这里我们从 sklearn 官网来解释什么是 elasticnet

1.1.5. Elastic-Net

ElasticNet is a linear regression model trained with both ℓ_1 and ℓ_2-norm regularization of the coefficients. This combination allows for learning a sparse model where few of the weights are non-zero like Lasso, while still maintaining the regularization properties of Ridge. We control the convex combination of ℓ_1 and ℓ_2 using the `l1_ratio` parameter.

```
import numpy as np
from sklearn.linear_model import ElasticNet
from sklearn.linear_model import SGDRegressor

X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

elastic_net = ElasticNet(alpha=0.4, l1_ratio=0.15)
elastic_net.fit(X, y)
print(elastic_net.predict(1.5))

sgd_reg = SGDRegressor(penalty='elasticnet', max_iter=1000)
sgd_reg.fit(X, y.ravel())
print(sgd_reg.predict(1.5))
```

SGDRegressor 参数解析

```
Ridge, ElasticNet, Lasso, sklearn.svm.SVR

"""
def __init__(self, loss="squared_loss", penalty="l2", alpha=0.0001,
             l1_ratio=0.15, fit_intercept=True, max_iter=None, tol=None,
             shuffle=True, verbose=0, epsilon=DEFAULT_EPSILON,
             random_state=None, learning_rate="invscaling", eta0=0.01,
             power_t=0.25, early_stopping=False, validation_fraction=0.1,
             n_iter_no_change=5, warm_start=False, average=False,
             n_iter=None):
```

```
loss : str, default: 'squared_loss'
    The loss function to be used. The possible values are 'squared_loss',
    'huber', 'epsilon_insensitive', or 'squared_epsilon_insensitive'

    The 'squared_loss' refers to the ordinary least squares fit.
    'huber' modifies 'squared_loss' to focus less on getting outliers
    correct by switching from squared to linear loss past a distance of
    epsilon. 'epsilon_insensitive' ignores errors less than epsilon and is
    linear past that; this is the loss function used in SVR.
    'squared_epsilon_insensitive' is the same but becomes squared loss past
    a tolerance of epsilon.

penalty : str, 'none', 'l2', 'l1', or 'elasticnet'
    The penalty (aka regularization term) to be used. Defaults to 'l2'
    which is the standard regularizer for linear SVM models. 'l1' and
    'elasticnet' might bring sparsity to the model (feature selection)
    not achievable with 'l2'.

alpha : float
    Constant that multiplies the regularization term. Defaults to 0.0001
```

for 循环进行梯度下降的时候，讲一下迭代次数设置问题，设置迭代次数太少，无法找到最优解就是欠拟合了，迭代次数过多就是浪费资源，让它在山谷底部来回震荡，能不能达到最优解，使用测试集一测就知道是否达到最优解了，然后可以去用 early stopping 机制

```
max_iter : int, optional
    The maximum number of passes over the training data (aka epochs).
    It only impacts the behavior in the ``fit`` method, and not the
    `partial_fit`.
    Defaults to 5. Defaults to 1000 from 0.21, or if tol is not None.

    .. versionadded:: 0.19

tol : float or None, optional
    The stopping criterion. If it is not None, the iterations will stop
    when (loss > previous_loss - tol). Defaults to None.
    Defaults to 1e-3 from 0.21.

    .. versionadded:: 0.19

shuffle : bool, optional
    Whether or not the training data should be shuffled after each epoch.
    Defaults to True.
```

```
eta0 : double
    The initial learning rate for the 'constant', 'invscaling' or
    'adaptive' schedules. The default value is 0.0 as eta0 is not used by
    the default schedule 'optimal'.

power_t : double
    The exponent for inverse scaling learning rate [default 0.5].

early_stopping : bool, default=False
    Whether to use early stopping to terminate training when validation
    score is not improving. If set to True, it will automatically set aside
    a fraction of training data as validation and terminate training when
    validation score is not improving by at least tol for
    n_iter_no_change consecutive epochs.

    .. versionadded:: 0.20

validation_fraction : float, default=0.1
```

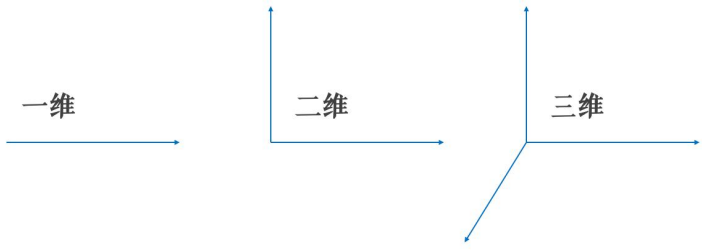
章节 7：多项式升维

升维手段 polynomial regression

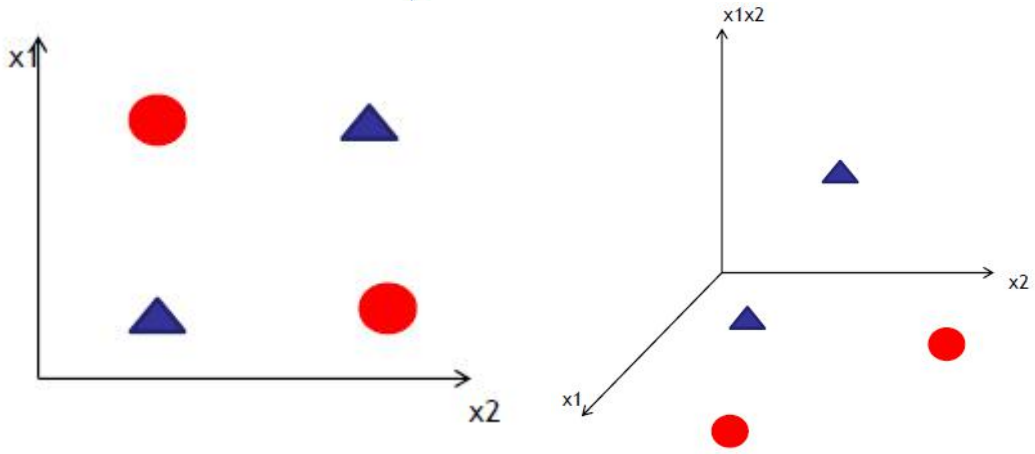
升维的目的是为了去解决欠拟合的问题的，也就是为了提高模型的准确率为目的的，因为当维度不够时，说白了就是对于预测结果考虑的因素少的话，肯定不能准确的计算出模型。

举个例子，我们看电影电视经常看到这样的场景：

女主对男主说：“我不爱你了”，于是男主很伤心，以为女主真的不爱他了，其实女主得了绝症，不想连累男主。



在做升维的时候，最常见的手段就是将已知维度进行相乘来构建新的维度，如下图所示。下图左展示的是线性不可分的情况，下图右通过升维使得变得线性可分。



多项式回归是升维的一种，它可以算是机器学习中的一种算法，不过和归一化一样一般算作数据预处理的手段，在 sklearn 模块下它处于 sklearn.preprocessing 模块下。它的目的就是将已有维度进行相乘，包括自己和自己相乘，来组成二阶的甚至更高阶的维度。

为什么它叫回归呢？主要是因为在线性回归算法下用它的时候比较多，因为线性回归是线性的模型，来拟合 X 和 y 之间的线性变化关系，当数据呈现一种非线性关系的时候，即 y 不是随着 X 线性变化的时候，我们这时有两种选择，一是选择用非线性的算法（回归树、神经网络等）去拟合非线性的数据。二是铁了心的用线性算法，但是需要把数据变成线性变化的关系才能更好的让算法发挥功能。其实也不要小看线性算法，毕竟线性的算法运算速度快是一大优势，当我们把数据通过升维变成线性变化关系后，就能让线性模型可以去应对更加广泛的数据集，也是一大幸事。比如举例根据年龄预测在医院里面的花销。

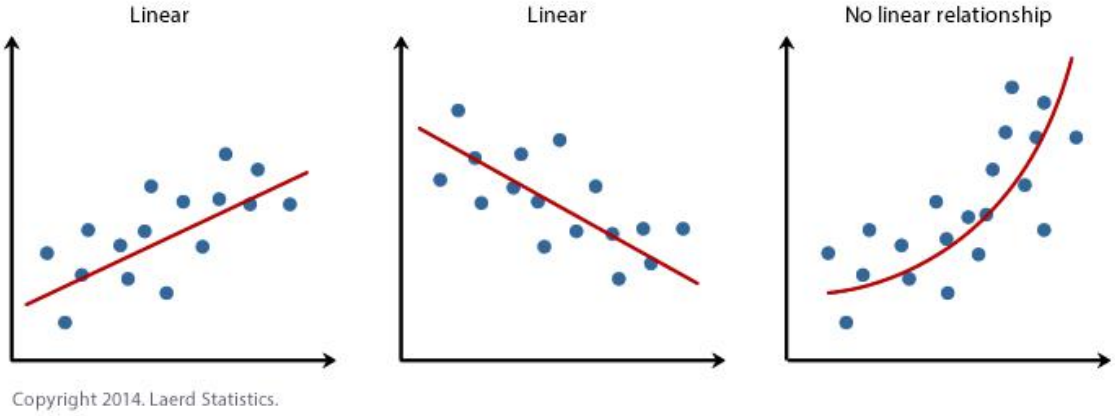
Data 数据-->Algorithm 算法-->Model 模型

Data 线性-->Algorithm 线性-->Model Good

Data 非线性-->Algorithm 非线性-->Model Good

Data 非线性-->Algorithm 线性-->Model Bad

Data 非线性-->Data 线性--Algorithm 线性-->Model Good



对于多项式回归来说主要是为了扩展线性回归算法来适应更广泛的数据集，比如我们数据集有两个维度 X1、X2，那么用多元线性回归公式就是 $\hat{y} = w_0 + w_1 x_1 + w_2 x_2$ ，当我们去使用二阶多项式升维的时候，数据集就从原来的 X1、X2 扩展成了 X1、X2、x_1^2、x_2^2、$x_1 x_2$。

因此多元线性回归就得去多计算三个维度所对应的 W 值，

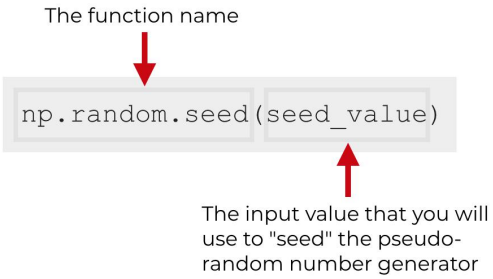
$$\hat{y} = w_0 + w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_2^2 + w_5x_1x_2$$

代码实战

导入 mean_squared_error 的目的是为了去评估一下升维的效果，导入主角 PolynomialFeatures 来做多项式回归升维，当然可以看出来它也没叫回归，而是叫什么 Features，很明显区别于那些回归算法，因为目的不同嘛，回归算法目的是为了去拟合，升维只是对数据集 X 进行变换。

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
```

通过 numpy 来创建数据集，np.random.seed(42)，是随机种子的设置方式，如果下次运行的时候还是 42 那随机出来的数据就是一样的。



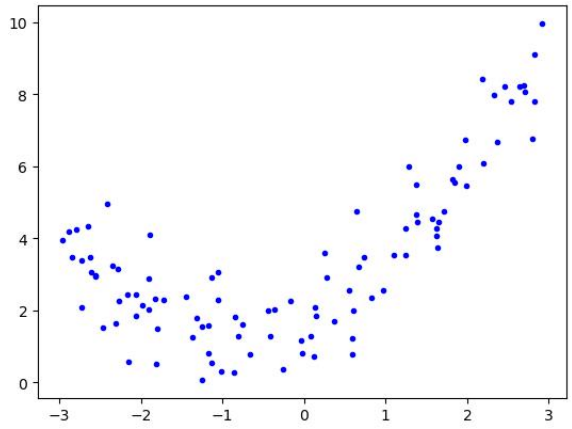

```
np.random.seed(42)
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X ** 2 + X + 2 + np.random.randn(m, 1)
```

切片操作把数据分为训练集和测试集

```
X_train = X[:80]
X_test = X[80:]
y_train = y[:80]
y_test = y[80:]
```

使用 matplotlib 下的 plot 方法来把真实数据的点画在二维平面中，同学们可以通过代码绘制的图看出来我们的 y 不是随着 X 线性变化的。是不是更像二阶的抛物线？

```
plt.plot(X, y, 'b.')
```



构建一个 dict 字典 d 来为了后面去使用一阶、二阶、十阶多项式回归的时候可以去分别使用不同的图示

b.:代表散点图，点的颜色是 blue 蓝色

g-:代表把点连成线，线的颜色是 green 绿色

r+:代表散点图，点的颜色是 red 红色，点用加号来表示

y*:代表散点图，点的颜色是 yellow 黄色，点用星号*来表示

```
d = {1: 'g-', 2: 'r+', 10: 'y*'}
```

我们把多项式回归数据预处理、多元线性回归模型的训练、模型的评估三步放入了 for 循环中去执行 ,每次迭代不同的地方只在于第一步的多项式回归进行预处理的时候使用几阶来进行升维，for 循环遍历一个字典的时候，这里 i 形参分别接的是字典中键值对的键，也就是说会分别把 1、2、10 传递给 PolynomialFeatures 中的 degree 超参数。

```
for i in d:
    poly_features = PolynomialFeatures(degree=i, include_bias=True)
    X_poly_train = poly_features.fit_transform(X_train)
    X_poly_test = poly_features.fit_transform(X_test)
    print(X_train[0])
    print(X_poly_train[0])
    print(X_train.shape)
    print(X_poly_train.shape)
```

这里我们打印训练集的一条样本在升维前和升维后的结果拿出来对比 ,并且对比前后数据集的形状。

degree=1

```
[-0.75275929]
[ 1.         -0.75275929]
(80, 1)
(80, 2)
```

degree=2

```
[-0.75275929]
[ 1.         -0.75275929  0.56664654]
(80, 1)
(80, 3)
```

degree=10

```
[-0.75275929]
[ 1.         -0.75275929  0.56664654 -0.42654845  0.32108831 -0.2417022
  0.18194358 -0.13695972  0.1030977  -0.07760775  0.05841996]
(80, 1)
(80, 11)
```

这里需要注意的是，PolynomialFeatures 中的 include_bias 设置为 True 的话，LinearRegression 中的 fit_intercept 就不需要设置为 True 了，反之亦然。

当下面画图的时候，因为我们要放在一张图里面去比较，所以用了切片操作只把升维之后的一个维度传递给 plot 函数。

```
lin_reg = LinearRegression(fit_intercept=False)
lin_reg.fit(X_poly_train, y_train)
print(lin_reg.intercept_, lin_reg.coef_)

y_train_predict = lin_reg.predict(X_poly_train)
y_test_predict = lin_reg.predict(X_poly_test)
plt.plot(X_poly_train[:, 1], y_train_predict, d[i])
```

degree=1

```
0.0 [[3.59274166 0.85841979]]
```

degree=2

```
0.0 [[1.72519842 0.89885719 0.55761947]]
```

degree=10

```
0.0 [[ 1.43216485e+00  1.50789179e+00  1.37504798e+00 -9.56344064e-01
  -5.74307442e-01  4.12979529e-01  1.73939385e-01 -6.51728008e-02
  -2.33506196e-02  3.39845254e-03  1.13043678e-03]]
```

最后我们来看看评估

```
print(mean_squared_error(y_train, y_train_predict))
print(mean_squared_error(y_test, y_test_predict))
```

degree=1

```
2.9847977077842067
3.190483392114553
```

degree=2

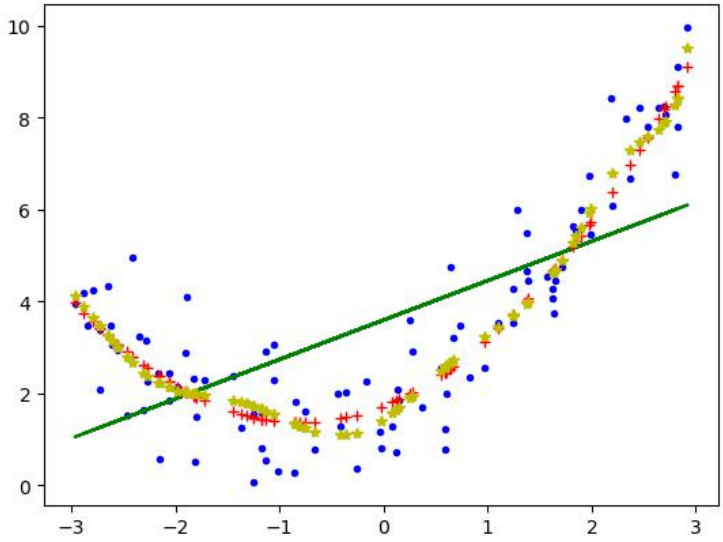
```
0.7485027751162633
0.9375214873212269
```

degree=10

```
0.7142156873482035
0.9330254117052545
```

总结

无论从下图，还是从上面的评估指标的对比，我们都可以发现使用多项式回归的时候，超参数 degree 很重要。当我们设置为 1 的时候欠拟合，当我们设置为 10 的时候过拟合，当我们设置为 2 的时候 just right



章节 8：代码实战保险花销预测(基础篇)

下面讲的这个项目脚本，把机器学习流程完整串了起来，不仅让大家掌握了多元线性回归的建模和评估，重点还有讲到特征工程里面的 np.log 改变数据为正太分布、离散型数据 one-hot 编码、去除空值、标准归一化、多项式回归升维，以及提高模型泛化能力的正则项 这些非常实用的技巧

数据

insurance.csv

```
age,sex,bmi,children,smoker,region,charges
19,female,27.9,0,yes,southwest,16884.924
18,male,33.77,1,no,southeast,1725.5523
28,male,33,3,no,southeast,4449.462
33,male,22.705,0,no,northwest,21984.47061
32,male,28.88,0,no,northwest,3866.8552
31,female,25.74,0,no,southeast,3756.6216
46,female,33.44,1,no,southeast,8240.5896
37,female,27.74,3,no,northwest,7281.5056
37,male,29.83,2,no,northeast,6406.4107
```

	A	B	C	D	E	F	G
1	age	sex	bmi	children	smoker	region	charges
2	19	female	27.9	0	yes	southwest	16884.92
3	18	male	33.77	1	no	southeast	1725.552
4	28	male	33	3	no	southeast	4449.462
5	33	male	22.705	0	no	northwest	21984.47
6	32	male	28.88	0	no	northwest	3866.855
7	31	female	25.74	0	no	southeast	3756.622
8	46	female	33.44	1	no	southeast	8240.59
9	37	female	27.74	3	no	northwest	7281.506
10	37	male	29.83	2	no	northeast	6406.411
11	60	female	25.84	0	no	northwest	28923.14
12	25	male	26.22	0	no	northeast	2721.321
13	62	female	26.29	0	yes	southeast	27808.73
14	23	male	34.4	0	no	southwest	1826.843
15	56	female	39.82	0	no	southeast	11090.72
16	27	male	42.13	0	yes	southeast	39611.76
17	19	male	24.6	1	no	southwest	1837.237
18	52	female	30.78	1	no	northeast	10797.34
19	23	male	23.845	0	no	northeast	2395.172
20	56	male	40.3	0	no	southwest	10602.39
21	30	male	35.3	0	yes	southwest	36837.47
22	60	female	36.005	0	no	northeast	13228.85
23	30	female	32.4	1	no	southwest	4149.736

读取数据

读取数据文件，这里使用 pandas 模块来读取文件，其中 read_csv()函数中 sep 可以设置成' '即可完成对 tsv 格式文本文件的读取，head()中默认显示头 5 行数据，也可以传参整数 n 就可以读取对应行数的数据，利用 pandas 读取来的数据类型是 dataframe 数据框类型，可以理解为二维的表格一样，有行有列很适合做数据分析和挖掘

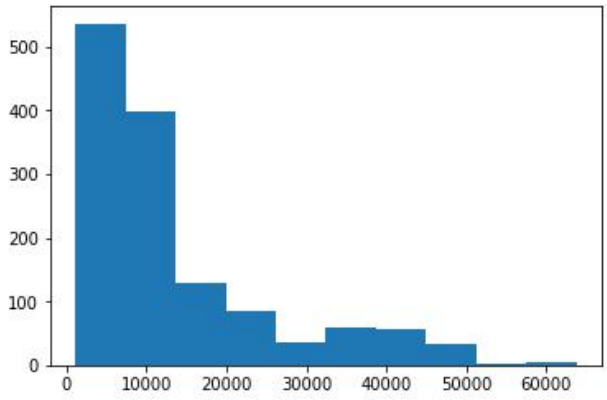
```
import pandas as pd
import numpy as np
data = pd.read_csv("./data/insurance.csv", sep=',')
data.head(n=6)
```

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520
5	31	female	25.740	0	no	southeast	3756.62160

EDA 数据探索

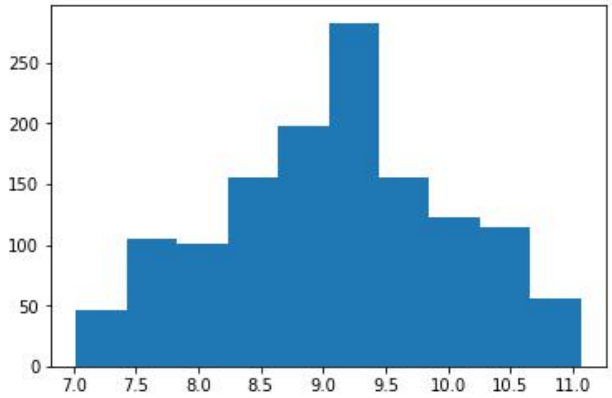
绘图分析目标变量 Y 的变化规律，看看是否发生了左偏或者右偏

```
import matplotlib.pyplot as plt
% matplotlib inline
plt.hist(data['charges'])
```



当发生如上图右偏的现象时，通常机器学习的手段是利用 np.log()函数对其进行矫正，这么做的目的是将其矫正为正太分布 ,而原因是因为线性回归的假设就是误差服从一个正太分布，而我们这章正是使用多元线性回归来实战

```
plt.hist(np.log(data['charges']))
```



特征工程

利用 pandas 模块很方便的 get_dummies()函数可以将数据非数值型的列做离散化，当然你也可以选择用 sklearn 的 LabelEncoder 和 OneHotEncoder，或者直接用 dataframe 的 map 函数来操作都可以

```
data = pd.get_dummies(data)
data.head()
```

	age	bmi	children	charges	sex_female	sex_male	smoker_no	smoker_yes	region_northeast	region_northwest	region_southeast	region_sout
0	19	27.900	0	16884.92400	1	0	0	1	0	0	0	
1	18	33.770	1	1725.55230	0	1	1	0	0	0	1	
2	28	33.000	3	4449.46200	0	1	1	0	0	0	1	
3	33	22.705	0	21984.47061	0	1	1	0	0	1	0	
4	32	28.880	0	3866.85520	0	1	1	0	0	1	0	

获取表中的 charges 花销一列作为目标变量 Y，其余列作为自变量 X1..Xn，顺便利用 fillna()将空值部分自动填充为 0

drop()函数中的 axis=1 指的是按照列来操作

```
x = data.drop('charges', axis=1)
```

```
y = data['charges']


x.fillna(0, inplace=True)
y.fillna(0, inplace=True)
```

```
x.tail()
```

	age	bmi	children	sex_female	sex_male	smoker_no	smoker_yes	region_northeast	region_northwest	region_southeast	region_southwest
1333	50	30.97	3	0	1	1	0	0	1	0	0
1334	18	31.92	0	1	0	1	0	1	0	0	0
1335	18	36.85	0	1	0	1	0	0	0	1	0
1336	21	25.80	0	1	0	1	0	0	0	0	1
1337	61	29.07	0	1	0	0	1	0	1	0	0

```
y.head()
```

```
Out[27]: 0    16884.92400
         1     1725.55230
         2     4449.46200
         3    21984.47061
         4     3866.85520
         Name: charges, dtype: float64
```

将数据切分为训练集和测试集，其中 test_size=0.3 代表测试集占数据集的百分之 30，这样剩下的训练集所占数据自动就是百分之 70

```
from sklearn.cross_validation import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3)
```

利用标准归一化对数据进行缩放，所谓的标准归一化就是求集合的方差和均值，然后用每条样本的每个特征字段减去相应所属维度的均值，再除以方差， 其实这里的标准归一化包含了均值归一化和方差归一化，因为机器学习领域同时做均值归一化和方差归一化已经成为了特征工程的一个标准，所以可以理解为什么这里 sklearn 叫 StandardScaler

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler(with_mean=True, with_std=True).fit(x_train)
```

这里大家要注意的是首先创建了对象后将 x_train 传给了 fit()函数返回一个 scaler，这个 scaler 本质上其实在利用训练及的数据计算每个维度的方差和均值，接下来下面用同一个 scaler 对象对训练集和测试集的数据进行变化，这里在实战操作的时候很容易直接使用 fit_transform()函数直接对训练集进行操作，那样的话测试集就无法利用训练集的数据来进行归一化了，我们总是假设训练集和测试集是属于同分布的，所以才要如此操作，紧记这一点

```
x_train_scaled = scaler.transform(x_train)
x_test_scaled = scaler.transform(x_test)
x_train_scaled
```

```
Out[30]: array([[-0.990648 ,  0.42413783,  0.7803878 , ..., -0.56418956,
                  1.6081688 , -0.5740605 ],
                [ 0.07564286, -0.17681016, -0.88538866, ..., -0.56418956,
                 -0.62182527,  1.74197668],
                [-0.06652926, -0.46117295,  0.7803878 , ..., -0.56418956,
                 -0.62182527, -0.5740605 ],
                ...,
                [ 0.5021592 , -1.74684719, -0.88538866, ...,  1.77245393,
                 -0.62182527, -0.5740605 ],
                [ 1.07084765,  0.01652431,  1.61327603, ..., -0.56418956,
                 -0.62182527,  1.74197668],
                [ 0.9997616 , -1.02748184, -0.88538866, ...,  1.77245393,
                 -0.62182527, -0.5740605 ]])
```

这里使用了多项式回归，一种升维的手段，目的之一是增加影响 Y 的维度或者叫因素，为了更好的拟合训练集数据变化规律，目的之二是因为这章我们利用的是线性模型来做回归，如果想用线性模型去拟合更加复杂的数据，我们可以选择让非线性数据通过升维变成线性的数据，然后利用线性模型去拟合很多时候就可以达到想要的效果

```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
x_train_scaled = poly_features.fit_transform(x_train_scaled)
x_test_scaled = poly_features.fit_transform(x_test_scaled)
```

模型训练

此处为了提高模型的泛化能力，防止过拟合，可以使用 Ridge 来替换 LinearRegression，这样等于加上了 L2 正则项，当然大家也可以使用 LassoRegreesion，那样就等于加上了 L1 正则项，而 alpha 这个超参正是正则项系数

```
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.ensemble import GradientBoostingRegressor
# reg = LinearRegression()
reg = Ridge(alpha=10.0)
# reg = GradientBoostingRegressor()
```

```
reg.fit(x_train_scaled, np.log1p(y_train))
```

```
y_predict = reg.predict(x_test_scaled)
```

模型评估

```
from sklearn.metrics import mean_squared_error
np.sqrt(mean_squared_error(y_true=np.log1p(y_train),
y_pred=reg.predict(x_train_scaled)))
```

```
Out[35]: 0.37533128295281942
```

```
np.sqrt(mean_squared_error(y_true=np.log1p(y_test), y_pred=y_predict))
```

```
Out[36]: 0.34375945802541824
```

```
mean_squared_error(y_true=np.log1p(y_train), y_pred=reg.predict(x_train_scaled))
```

Out[37]: 0.1408735719630094

```
mean_squared_error(y_true=np.log1p(y_test), y_pred=y_predict)
```

Out[38]: 0.11817056498192928

```
np.sqrt(mean_squared_error(y_true=y_train,
y_pred=np.exp(reg.predict(x_train_scaled))))
```

Out[39]: 4995.976799682393

```
np.sqrt(mean_squared_error(y_true=y_test, y_pred=np.exp(y_predict)))
```

Out[40]: 5253.6166608171343

章节 9：代码实战保险花销预测(进阶版)

讲完 original 脚本后，再讲这个脚本。讲这个脚本的目的是，这个脚本是优化，最终的评估指标更好，这个脚本可以让大家学到机器学习中了解数据是多么的重要，当数据探索部分做好，更了解自己的数据，并不需要前面那么全套的操作，或者并不用杀鸡用牛刀，就可以事半功倍

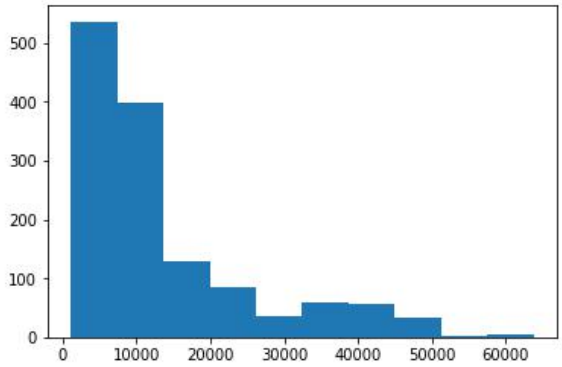
读取数据

```
import pandas as pd
import numpy as np
data = pd.read_csv("./data/insurance.csv")
data.head()
```

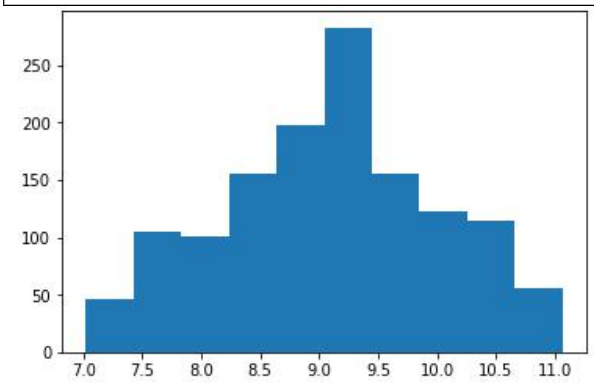
	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520

EDA 数据探索

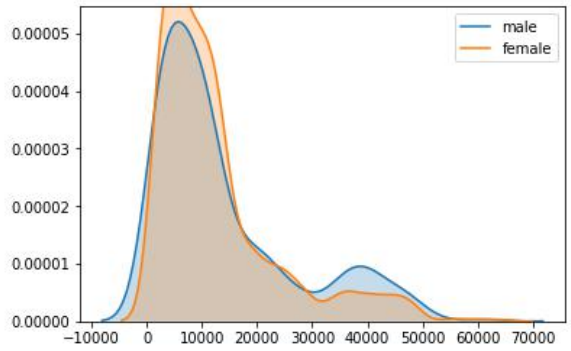
```
import matplotlib.pyplot as plt
% matplotlib inline
plt.hist(data['charges'])
```



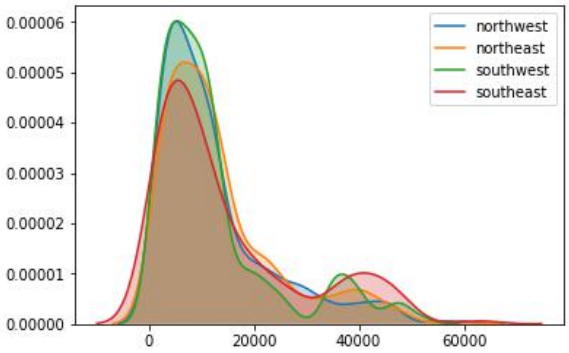

```
plt.hist(np.log(data['charges']))
```



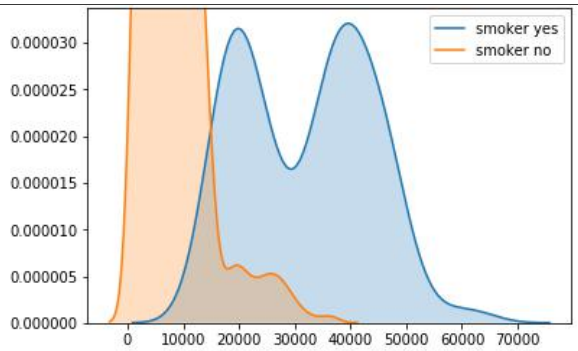

```
import seaborn as sns
sns.kdeplot(data.loc[data.sex=='male', 'charges'], shade=True, label='male')
sns.kdeplot(data.loc[data.sex=='female', 'charges'], shade=True, label='female')
```



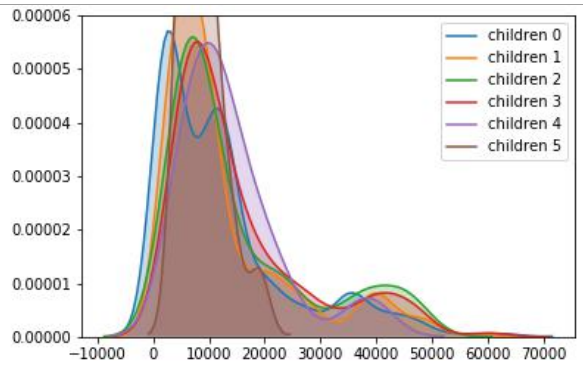

```
sns.kdeplot(data.loc[data.region=='northwest', 'charges'], shade=True, label='northwest')
sns.kdeplot(data.loc[data.region=='northeast', 'charges'], shade=True, label='northeast')
sns.kdeplot(data.loc[data.region=='southwest', 'charges'], shade=True, label='southwest')
sns.kdeplot(data.loc[data.region=='southeast', 'charges'], shade=True, label='southeast')
```




```
sns.kdeplot(data.loc[data.smoker=='yes', 'charges'], shade=True, label='smoker yes')
sns.kdeplot(data.loc[data.smoker=='no', 'charges'], shade=True, label='smoker no')
```




```
sns.kdeplot(data.loc[data.children==0, 'charges'], shade=True, label='children 0')
sns.kdeplot(data.loc[data.children==1, 'charges'], shade=True, label='children 1')
sns.kdeplot(data.loc[data.children==2, 'charges'], shade=True, label='children 2')
sns.kdeplot(data.loc[data.children==3, 'charges'], shade=True, label='children 3')
sns.kdeplot(data.loc[data.children==4, 'charges'], shade=True, label='children 4')
sns.kdeplot(data.loc[data.children==5, 'charges'], shade=True, label='children 5')
```



特征工程

这行代码很重要，因为探索数据发现 region、sex 和最后的 charges 花销基本上相关系数很低很低，所以直接特征选择给去除掉，降维可以使得模型训练和使用的时候计算速度更快，同时也更加有效的防止了过拟合

```
data = data.drop(['region','sex'], axis=1)
data.head()
```

	age	bmi	children	smoker	charges
0	19	27.900	0	yes	16884.92400
1	18	33.770	1	no	1725.55230
2	28	33.000	3	no	4449.46200
3	33	22.705	0	no	21984.47061
4	32	28.880	0	no	3866.85520

这里用到了 pandas 里面的很高级的 apply 函数，传入自定义的一个函数。

用于将样本中每个 bmi 特征大于等于 30 的变成标签 ‘over’ ,小于 30 的变成 ‘under’，这样做是建立在知道医学上 bmi 这个肥胖指数 30 是个很关键的阈值，大于等于 30 属于肥胖，小于 30 不属于肥胖。

另外探索数据发现维度 children 这个指标是 0 的和大于 0 的对最后目标函数影响不同，说明没孩子和有孩子的在保险这项花销这件事情上观点是不同的，而大于 0 甭管有几个孩子观点都是差不多的，这可能就是俗话说的有孩子以后人就不一样了吧，所以将没有孩子的打标签为 ‘no’ ，有孩子的打标签为 ‘yes’ 。

```
def greater(df,bmi,num_child):
    df['bmi'] = 'over' if df['bmi'] >= bmi else 'under'
    df['children'] = 'no' if df['children'] == num_child else 'yes'
    return df
data = data.apply(greater,axis=1,args=(30,0,))
data.head()
```

	age	bmi	children	smoker	charges
0	19	under	no	yes	16884.92400
1	18	over	yes	no	1725.55230
2	28	over	yes	no	4449.46200
3	33	under	no	no	21984.47061
4	32	under	no	no	3866.85520

```
data = pd.get_dummies(data)
data.head()
```

	age	charges	bmi_over	bmi_under	children_no	children_yes	smoker_no	smoker_yes
0	19	16884.92400	0	1	1	0	0	1
1	18	1725.55230	1	0	0	1	1	0
2	28	4449.46200	1	0	0	1	1	0
3	33	21984.47061	0	1	1	0	1	0
4	32	3866.85520	0	1	1	0	1	0

```
x = data.drop('charges', axis=1)
y = data['charges']
```

```
x.fillna(0, inplace=True)
y.fillna(0, inplace=True)
```

```
x.tail()
```

	age	bmi_over	bmi_under	children_no	children_yes	smoker_no	smoker_yes
1333	50	1	0	0	1	1	0
1334	18	1	0	1	0	1	0
1335	18	1	0	1	0	1	0
1336	21	0	1	1	0	1	0
1337	61	0	1	1	0	0	1

```
y.head()
```

```
Out[56]: 0    16884.92400
         1     1725.55230
         2     4449.46200
         3    21984.47061
         4     3866.85520
         Name: charges, dtype: float64
```

```
from sklearn.cross_validation import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3)
```

```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
x_train_poly = poly_features.fit_transform(x_train)
x_test_poly = poly_features.fit_transform(x_test)
```

模型训练

```
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.ensemble import GradientBoostingRegressor
# reg = LinearRegression()
reg = Ridge(alpha=10.0)
# reg = GradientBoostingRegressor()
```

```
reg.fit(x_train_poly, np.log1p(y_train))
```

```
Out[60]: Ridge(alpha=10.0, copy_X=True, fit_intercept=True, max_iter=None,
           normalize=False, random_state=None, solver='auto', tol=0.001)
```

```
y_predict = reg.predict(x_test_poly)
```

模型评估

```
from sklearn.metrics import mean_squared_error
np.sqrt(mean_squared_error(y_true=np.log1p(y_train),
y_pred=reg.predict(x_train_poly)))
```

Out[62]: 0.39730909472332399

```
np.sqrt(mean_squared_error(y_true=np.log1p(y_test), y_pred=y_predict))
```

Out[63]: 0.34244575479882072

```
mean_squared_error(y_true=np.log1p(y_train), y_pred=reg.predict(x_train_poly))
```

Out[64]: 0.15785451674986722

```
mean_squared_error(y_true=np.log1p(y_test), y_pred=y_predict)
```

Out[65]: 0.11726909497973406

```
np.sqrt(mean_squared_error(y_true=y_train,
y_pred=np.exp(reg.predict(x_train_poly))))
```

Out[66]: 4831.9567256042083

```
np.sqrt(mean_squared_error(y_true=y_test, y_pred=np.exp(y_predict)))
```

Out[67]: 4461.9263320090431