4、训练模型

2019年1月23日　　10:01

本章我们将从最简单的模型之一——线性回归模型，开始介绍两种非常不同的模型训练方法：

a. 通过“闭式”方程——直接计算出最适合训练集的模型参数（也就是使训练集上的成本函数最小化的模型参数）
b. 使用迭代优化的方法，即梯度下降，逐渐调整模型参数直至训练集上的成本函数降至最低，最终趋同于第一种方法计算出来的模型参数。我们还会研究几个梯度下降的变体，包括批量梯度下降、小批量梯度下降以及随机梯度下降。

接着我们会进入多项式回归的讨论，这是一个更为复杂的模型，更适合非线性数据集。由于该模型的参数比线性模型更多，因此更容易造成对训练数据的过度拟合，我们将使用学习曲线来分辨这种情况是否发生。然后再介绍几种正则化技巧，降低过拟合训练数据的风险。

1. 线性回归

   概括地说，线性模型就是对输入特征加权求和，再加上一个称为偏置项的常数，以此进行预测。我们怎样训练线性回归模型呢？回想一下，训练模型就是设置模型参数直到模型最适应训练数据集的过程。要达到这个目的，我们首先需要知道怎么衡量模型对训练数据的拟合程度是好还是差。在此我们可以使用MSE作为性能指标。在训练集X上，使用如下公式计算线性回归的MSE，hθ为假设函数。为了简化符号，我们将MSE（X， hθ）直接写作MSE（θ）。

$$\mathrm{MSE}(\mathbf{X}, h_\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\theta^T \cdot \mathbf{X}^{(i)} - y^{(i)}\right)^2$$

   为了得到使成本函数最小的θ值，有一个闭式解法——也就是一个直接得出结果的数学方程，即标准方程

   公式4-4：标准方程

$$\hat{\theta} = \left(\mathbf{X}^T \cdot \mathbf{X}\right)^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

   可以生成一些线性数据来测试这个公式

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

   现在使用标准方程来计算θ 尖，使用numpy的线性代数模块（np.linalg）中的inv()函数来对矩阵求逆，并用dot方法计算矩阵的内积：

```
X_b = np.c_[np.ones((100,1)),X]
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
#加一列1是为了计算截距
```

```
theta_best
```

```
array([[4.13412277],
       [2.78653397]])
```

我们期待的θ0是4，θ1是3，我们得到的参数与之非常接近，噪声的存在使得我们不可能完全还原。现在可以使用θ尖做出预测

```
X_new = np.array([[0],[2]])
X_new_b = np.c_[np.ones((2,1)),X_new]
y_predict = X_new_b.dot(theta_best)
y_predict
```

```
array([[4.13412277],
       [9.7071907 ]])
```

sklearn的等效代码如下所示：

```
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
lr.fit(X,y)
lr.intercept_,lr.coef_
```

```
(array([4.13412277]), array([[2.78653397]]))
```

```
lr.predict(X_new)
```

```
array([[4.13412277],
       [9.7071907 ]])
```

标准方程求逆的矩阵XT X是一个nXn的矩阵（n是特征数量）。对这种矩阵求逆的计算复杂度通常是O(n^2.4)到O(n^3)之间。换句话说，如果特征数量翻倍，那么计算时间将乘以大约2^2.4=5.3到2^3=8倍之间。

#当特征数量比较大时，标准方程的计算将极其缓慢。好的一面是，相对于训练集中的实例数量（O(m)）来说，方程是线性的，所以能够有效地处理大量的训练集，只要内存足够。同样，线性回归模型一经训练（不论是标准方程还是其他算 法），预测就非常快速：因为计算复杂度相对于想要预测的实例数量 和特征数量来说，都是线性的。换句话说，对两倍的实例（或者是两 倍的特征数）进行预测，大概需要两倍的时间。

现在，我们再看几个截然不同的线性回归模型的训练方法，这些方法更适合特征数或者训练实例数量大到内存无法满足要求的场景。

2. 梯度下降

首先使用一个随机的θ值（这被称为随机初始化），然后逐步改进，每次踏出一步，每一步都尝试降低一点成本函数（如 MSE），直到算法收敛出一个最小值。梯度下降中一个重要参数是每一步的步长，这取决于超参数学习率。如果学习率太低，算法需要经过大量迭代才能收敛，这将耗费很长时间。反过来说，如果学习率太高，那你可能会越过山谷直接到达山的另一边，甚至有可能比之前的起点还要高。这会导致算法发散，值越来越大，最后无法找到好的解决方案。

图4-6显示了梯度下降的两个主要挑战：如果随机初始化，算法从左侧起步，那么会收敛到一个局部最小值，而不是全局最小值。如果算法从右侧起步，那么需要经过很长时间才能越过整片高原，如果你停下得太早，将永远达不到全局最小值。

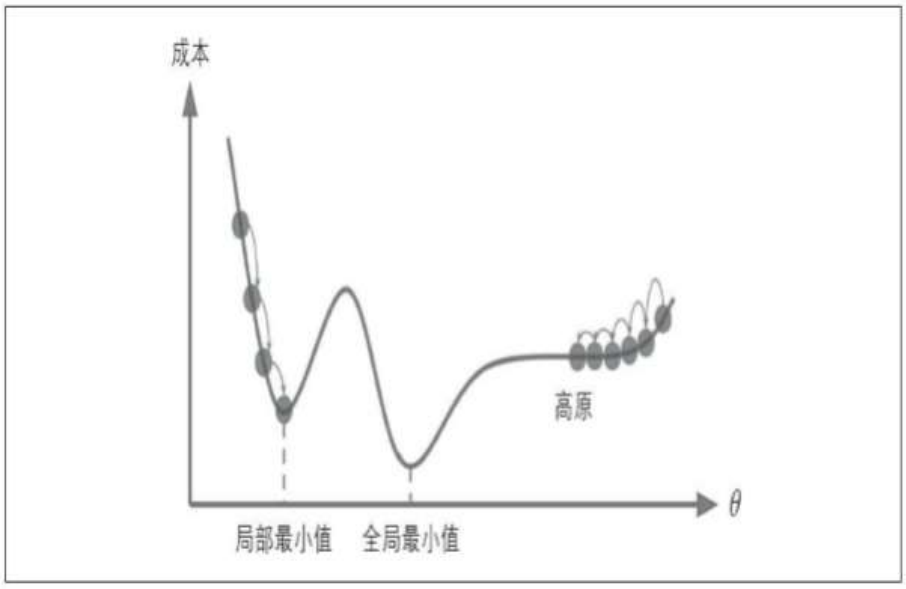


图4-6：梯度下降陷阱

幸好，线性回归模型的MSE成本函数恰好是个凸函数，这意味着连接曲线上任意两个点的线段永远不会跟曲线相交。也就是说不存在 局部最小，只有一个全局最小值。它同时也是一个连续函数，所以斜率不会产生陡峭的变化。这两件事保证的结论是：即便是乱走，梯度下降都可以趋近到全局最小值（只要等待时间足够长，学习率也不 是太高）。

但线性回归的成本函数虽然是碗状的，但如果不同特征的尺寸差别巨大，那它可能是一个非常细长的碗。如图4-7所示的梯度下降，左边的训练集上特征1和特征2具有相同的数值规模，而右边的训练集上，特征1的 值则比特征2要小得多。（注：因为特征1的值较小，所以θ1需要更大 的变化来影响成本函数，这就是为什么碗形会沿着θ1轴拉长。）

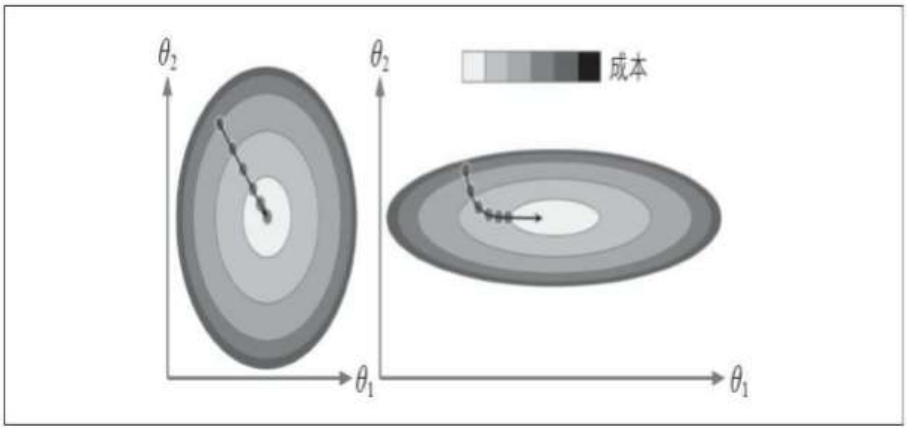


图4-7：特征值缩放和特征值无缩放的梯度下降

正如你所见，左图的梯度下降算法直接走向最小值，可以快速到达。而在右图中，先是沿着与全局最小值方向近乎垂直的方向前进， 接下来是一段几乎平坦的长长的山谷。最终还是会抵达最小值，但是这需要花费大量的时间。

#应用梯度下降时，需要保证所有特征值的大小比例都差不多 （比如使用Scikit-Learn的StandardScaler类），否则收敛的时间会长很多。（因为你的对于θ2而言，合适的学习率对于θ1来说太小了，但是又不能把学习率调大，否则对θ2又不合适了）

这张图也说明，训练模型也就是搜寻使成本函数（在训练集上）最小化的参数组合。这是模型参数空间层面上的搜索：模型的参数越多，这个空间的维度就越多，搜索就越难。同样是在干草堆里寻找一根针，在一个三百维的空间里就比在一个三维空间里要棘手得多。幸运的是，线性回归模型的成本函数是凸函数，

针就躺在碗底。

a. 批量梯度下降

要实现梯度下降，你需要计算每个模型关于参数θj的成本函数的梯度。换言之，你需要计算的是如果改变θj，成本函数会改变多少。 这被称为偏导数。公式4-5计算了关于参数θj的成本函数的偏导数

公式4-5：成本函数的偏导数

$$\frac{\partial}{\partial \theta_j}\mathrm{MSE}(\theta) = \frac{2}{m}\sum_{i=1}^{m}\left(\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}\right)x_j^{(i)}$$

如果不想单独计算这些梯度，可以使用公式4-6对其进行一次性 计算。

公式4-6：成本函数的梯度向量

$$\nabla_\theta \mathrm{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0}\mathrm{MSE}(\theta) \\ \frac{\partial}{\partial \theta_1}\mathrm{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n}\mathrm{MSE}(\theta) \end{pmatrix} = \frac{2}{m}\mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - \mathbf{y})$$

请注意，公式4-6在计算梯度下降的每一步时，都是基于完整的训练集X的。这就是为什么该算法会被称为批量梯度下降：每一 步都使用整批训练数据。因此，面对非常庞大的训练集时，算法会变得极慢（不过我们即将看到快得多的梯度下降算法）。但是，梯度下降算法随特征数量扩展的表现比较好：如果要训练的线性模型拥有几十万个特征，使用梯度下降比标准方程要快得多。

一旦有了梯度向量，哪个点向上，就朝反方向下坡。这时学习率η就发挥作用了：用梯度向量乘以η确定下坡步长的大小（公式4-7）。

公式4-7：梯度下降步长

$$\theta^{(\text{next step})} = \theta - \eta\nabla_\theta \mathrm{MSE}(\theta)$$

这个算法的快速实现如下：

```
eta = 0.1
n_iterations = 1000
m = 100
```

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

for iteration in range(n_iterations):
   gradients = 2/m*X_b.T.dot(X_b.dot(theta)-y)
   theta = theta - eta*gradients
```

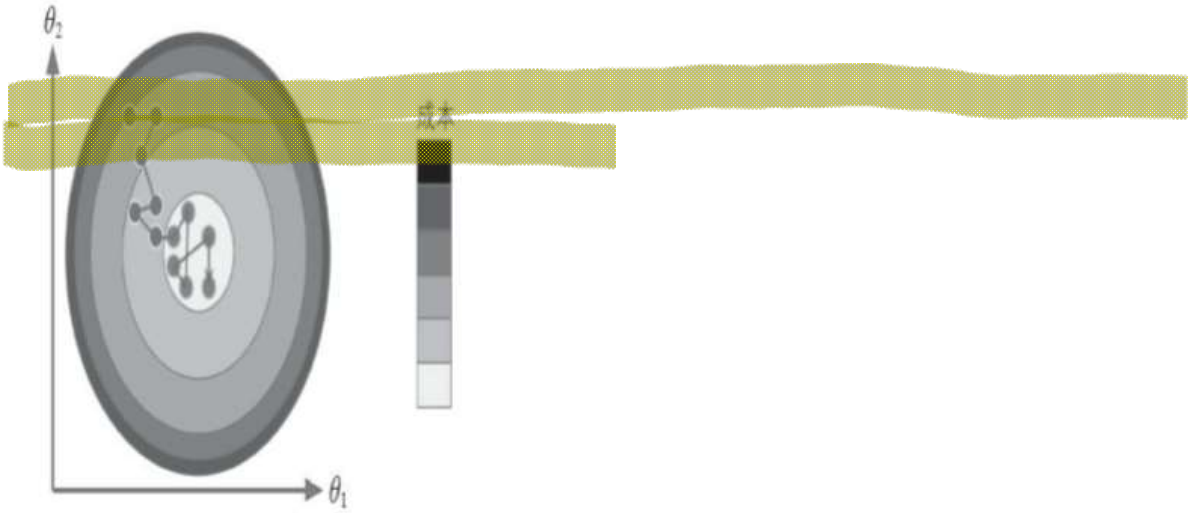
```
theta
```

```
array([[4.13412277],
       [2.78653397]])
```

b. 随机梯度下降

批量梯度下降的主要问题是它要用整个训练集来计算每一步的梯度，所以训练集很大时，算法会特别慢。与之相反的极端是随机梯度下降，每一步在训练集中随机选择一个实例，并且仅基于该单个实例来计算梯度。显然，这让算法变得快多了，因为每个迭代都只需要操作少量的数据。它也可以被用来训练海量的数据集，因为每次迭代只需要在内存中运行一个实例即可。

另一方面，由于算法的随机性质，它比批量梯度下降要不规则得多。成本函数将不再是缓缓降低直到抵达最小值，而是不断上上下下，但是从整体来看，还是在慢慢下降。随着时间推移，最终会非常接近最小值，但是即使它到达了最小值，依旧还会持续反弹，永远不会停止（见图4-9）。所以算法停下来的参数值肯定是足够好的，但不是最优的。



当成本函数非常不规则时（见图4-6），随机梯度下降其实可以帮助算法跳出局部最小值，所以相比批量梯度下降，它对找到全局最 小值更有优势。

因此，随机性的好处在于可以逃离局部最优，但缺点是永远定位不出最小值。要解决这个困境，有一个办法是逐步降低学习率。开始 的步长比较大（这有助于快速进展和逃离局部最小值），然后越来越小，让算法尽量靠近全局最小值。这个过程叫作模拟退火，因为它类似于冶金时熔化的金属慢慢冷却的退火过程。确定每个迭代学习率的函数叫作学习计划。如果学习率降得太快，可能会陷入局部最小值， 甚至是停留在走向最小值的半途中。如果学习率降得太慢，你需要太长时间才能跳到差不多最小值附近，如果提早结束训练，可能只得到 一个次优的解决方案。

下面这段代码使用了一个简单的学习计划实现随机梯度下降：

```
n_epochs = 50
t0, t1 = 5, 50

def learning_schedule(t):
    return t0/(t+t1)

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

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2*xi.T.dot(xi.dot(theta)-yi)
        eta = learning_schedule(epoch*m+i)
        theta = theta - eta*gradients
theta
```

```
array([[3.85378407],
       [3.0301703 ]])
```

因为实例是随机挑选，所以在同一轮里某些实例可能被挑选多次，而有些实例则完全没被选到。如果你希望每一轮算法都能遍历每个实例，有一种办法是将训练集洗牌打乱，然后一个接一个的使用实例，用完再重新洗牌，以此继续。不过这种方法通常收敛得更慢。

在Scikit-Learn里，用SGD执行线性回归可以使用SGDRegressor类，其默认优化的成本函数是平方误差。下面这段代码从学习率0.1 开始（eta0=0.1），使用默认的学习计划（跟前面的学习计划不同）运行了50轮，而且没有使用任何正则化（penalty=None，后面即将介 绍更多与此相关的细节）：

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(n_iter=50,penalty=None,eta0=0.1)
sgd_reg.fit(X,y.ravel())
#ravel前的shape是（100，1），ravel后（100，）
```

```
sgd_reg.intercept_, sgd_reg.coef_
```

```
(array([3.86126963]), array([2.97756551]))
```

c. 小批量梯度下降

我们要了解的最后一个梯度下降算法叫作小批量梯度下降。一旦理解了批量梯度下降和随机梯度下降，这个算法就非常容易理解了： 每一步的梯度计算，既不是基于整个训练集（如批量梯度下降）也不是基于单个实例（如随机梯度下降），而是基于一小部分随机的实例集也就是小批量。相比随机梯度下降，小批量梯度下降的主要优势在于可以从矩阵运算的硬件优化中获得显著的性能提升，特别是需要用到图形处理器时。

这个算法在参数空间层面的前进过程也不像SGD那样不稳定，特别是批量较大时。所以小批量梯度下降最终会比SGD更接近最小值一 些。但是另一方面，它可能更难从局部最小值中逃脱（不是我们前面 看到的线性回归问题，而是对于那些深受局部最小值陷阱困扰的问题）。图4-11显示了三种梯度下降算法在训练过程中参数空间里的行进路线。它们最终都汇聚在最小值附近，批量梯度下降最终停在了最小值上，而随机梯度下降和小批量梯度下降还在继续游走。但是，别忘了批量梯度可是花费了大量时间来计算每一步的，如果用好了学习计划，随机梯度下降和小批量梯度下降也同样能到达最小值。

最后，我们来比较一下到目前为止所讨论过的线性回归算法（m是训练实例的数量，n是特征数量）

表4-1：线性回归算法比较

算法	m很大	是否支持核外	n很大	超参数	是否需要缩放	Scikit-Learn

算法	m很大	是否支持核外	n很大	超参数	是否需要缩放	Scikit-Learn
标准方程	快	否	慢	0	否	LinearRegression
批量梯度下降	慢	否	快	2	是	n/a
随机梯度下降	快	是	快	≥2	是	SGDRegressor
小批量梯度下降	快	是	快	≥2	是	n/a

如果include_bias为true，则会添加0次幂

训练后的模型几乎无差别：所有这些算法最后出来的模型都非常相似，并且以完全相同的方式做出预测。

3. 多项式回归

如果数据比简单的直线更为复杂，该怎么办？令人意想不到的是，其实你也可以用线性模型来拟合非线性数据。一个简单的方法就 是将每个特征的幂次方添加为一个新特征，然后在这个拓展过的特征 集上训练线性模型。这种方法被称为多项式回归。

我们看下面这个例子，首先，基于简单的二次方程（注：二次方 程的形式为y=ax2+bx+c）制造一些非线性数据。显然，直线永远不可能拟合这个数据。所以我们使用Scikit-Learn 的PolynomialFeatures类来对训练数据进行转换，将每个特征的平方（二次多项式）作为新特征加入训练集（这个例子中只有一个特征）：

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

X_poly现在包含原本的特征X和该特征的平方。现在对这个扩展 后的训练集匹配一个LinearRegression模型

```
lin_reg = LinearRegression()
lin_reg.fit(X_poly,y)
lin_reg.intercept_,lin_reg.coef_
```

```
(array([2.00157353]), array([[0.97481145, 0.45425412]]))
```

注意，当存在多个特征时，多项式回归能够发现特征和特征之间 的关系（纯线性回归模型做不到这一点）。这是因为 PolynomialFeatures会在给定的多项式阶数下，添加所有特征组合。例 如，有两个特征a和b，阶数degree=3，PolynomialFeatures不只会添加 特征a2、a3、b2和b3，还会添加组合ab、a2b以及ab2。

4. 学习曲线

高阶多项式回归对训练数据的拟合，很可能会比简单线性回归要 好。例如，图4-14使用了一个300阶多项式模型来处理训练数据，并 将结果与一个纯线性模型和一个二次模型（二阶多项式）进行对比。 注意看这个300阶模型是如何波动以使其尽可能贴近训练实例的。

当然，这个高阶多项式回归模型严重地过度拟合了训练数据，而线性模型则是拟合不足。这个案例中泛化结果最好的是二次模型。这很合理，因为数据本身是用二次模型生成的。但是一般来说，你不会 知道生成数据的函数是什么，那么该如何确定模型的复杂程度呢？怎么才能判断模型是过度拟合还是拟合不足呢？

在第2章中，我们使用了交叉验证来评估模型的泛化性能。如果模型在训练集上表现良好，但是交叉验证的泛化表现非常糟糕，那么模型就是过度拟合。如果在二者上的表现都不佳，那就是拟合不足。 这是判断模型太简单还是太复杂的一种方法。

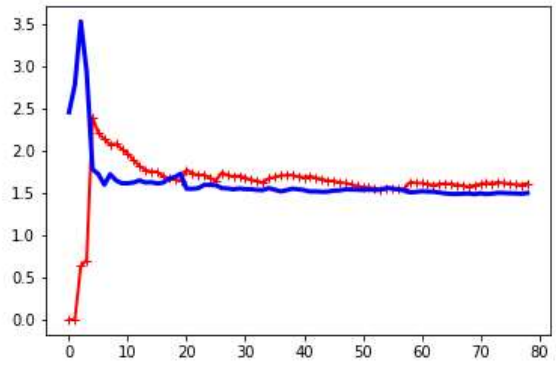
还有一种方法是观察学习曲线：这个曲线绘制的是模型在训练集和验证集上，关于“训练集大小”的性能函

数。要生成这个曲线，只需要在不同大小的训练子集上多次训练模型即可。下面这段代码，在给定训练集下定义了一个函数，绘制模型的学习曲线：

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
%matplotlib inline
import matplotlib.pyplot as plt
def plot_learning_curves(model,X,y):
    X_train,X_val,y_train,y_val = train_test_split(X,y,test_size=0.2)
    train_errors,val_errors = [],[]
    for m in range(1,len(X_train)):
        model.fit(X_train[:m],y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train_predict,y_train[:m]))
        val_errors.append(mean_squared_error(y_val_predict,y_val))
    plt.plot(np.sqrt(train_errors),'r-+',linewidth=2,label='train')
    plt.plot(np.sqrt(val_errors),'b-',linewidth=3,label='val')
```

看看纯线性回归模型（一条直线）的学习曲线

```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg,X,y)
```

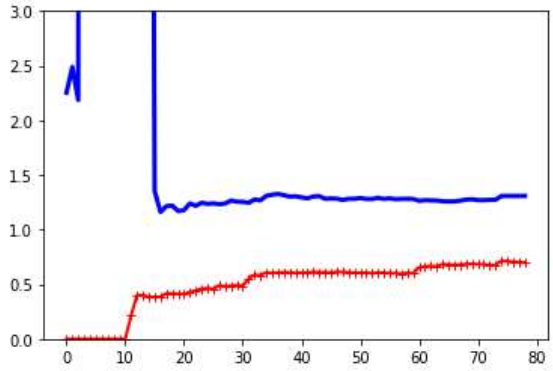


这里值得解释一二，首先，我们来看训练数据上的性能：当训练 集中只包括一两个实例时，模型可以完美拟合，这是为什么曲线是从 0开始的。但是，随着新的实例被添加进训练集中，模型不再能完美 拟合训练数据了，因为数据有噪声，并且根本就不是线性的。所以训 练集的误差一路上升，直到抵达一个高地，从这一点开始，添加新实例到训练集中不再使平均误差上升或下降。然后我们再来看看验证集 的性能表现。当训练集实例非常少时，模型不能很好地泛化，这是为 什么验证集误差的值一开始非常大，随着模型经历更多的训练数据， 它开始学习，因此验证集误差慢慢下降。但是仅靠一条直线终归不能 很好地为数据建模，所以误差也停留在了一个高值，跟另一条曲线十分接近。这条学习曲线是典型的模型拟合不足。两条曲线均到达高地，非常接近，而且相当高。

#如果你的模型对训练数据拟合不足，添加更多训练示例也于事无补。你需要使用更复杂的模型或者找到更好的特征

现在我们再来看看在同样的数据集上，一个10阶多项式模型的学习曲线

```
from sklearn.pipeline import Pipeline
plt.ylim(0,3)
polynomial_regression = Pipeline((('poly_features',PolynomialFeatures(degree=10,include_bias=False)),
                                  ('sgd_reg',LinearRegression()),))
plot_learning_curves(polynomial_regression,X,y)
```



这条学习曲线看起来跟前一条差不多，但是有两个非常重大的区别：

·训练数据的误差远低于线性回归模型。

·两条曲线之间有一定差距。这意味着该模型在训练数据上的表现比验证集上要好很多，这正是过度拟合的标志。但是，如果你使用更大的训练集，那么这两条曲线将会越来越近。改进模型过度拟合的方法之一是提供更多的训练数据，直到 验证误差接近训练误差

在统计机器学习领域，一个重要的理论是，模型的泛化误差可以被表示为三个截然不同的误差之和：

a. 偏差

这部分泛化误差的原因在于错误的假设，比如假设数据是线性的，而实际上是二次的（模型拟合能力不足）。高偏差模型最有可能对训练数据拟合不足。

b. 方差

这部分误差是由于模型对训练数据的微小变化过度敏感导致的。 具有高自由度的模型（例如高阶多项式模型）很可能也有高方差，所 以很容易对训练数据过度拟合。

c. 噪声

这部分误差是因为数据本身的噪声所致。减少这部分误差的唯一方法就是清理数据（比如修复数据源，如损坏的传感器，或者是检测并移除异常值）

增加模型的复杂度通常会显著提升模型的方差，减少偏差。反过 来，降低模型的复杂度则会提升模型的偏差，降低方差。这就是为什 么称其为权衡。

5. 正则线性模型

减少过度拟合的一个办法就是对模型正则化（即约束它）：它拥有的自由度越低，就越不容易过度拟合数据。比如，将多项式模型正则化的简单方法就是降低多项式的阶数。

对线性模型来说，正则化通常通过约束模型的权重来实现。下面将介绍岭回归（ridge regression）、套索回归（lasso regression）以及弹性网络（elastic net）这三种不同的实现方法对权重进行约束。

a. 岭回归

即在损失函数中加入参数的L2范数作为正则项。这使得学习中的算法不仅需要拟合数据，同时还要让模型权重保持最小。注意，正则项只能在训练的时候添加到成本函数中，一旦训练完成，需要使用未经正则化的性能指标来评估模型性能。

#训练阶段使用的成本函数与测试时使用的成本函数不同是非常常见的现象。除了正则化以外，还有

一个导致这种不同的原因是， 训练时的成本函数通常都可以使用优化过的衍生函数，而测试用的性能指标需要尽可能接近最终目标。举例来说，一个使用对数损失函数 （log loss，后文即将讨论）作为成本函数来训练的分类器，最后评估时使用的指标却是精度/召回率

超参数α控制的是对模型进行正则化的程度。如果α=0，则岭回归就是线性模型，如果α非常大，那么所有的权重都将非常接近于0，结果是一条穿过数据平均值的水平线。下面给出了岭回归模型的成本函数：

$$J(\theta) = \mathrm{MSE}(\theta) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

注意，这里偏置项θ0没有正则化（求和从i=1开始，不是i=0）。 如果我们将w定义为特征权重的向量（θ1到θn），那么正则项即等于 1/2（||w||2）2其中||w||2为权重向量的l2范数。[2]而对于梯度下降，只需要在MSE梯度向量（公式4-6）上添加αw即可。

在执行岭回归之前，必须对数据进行缩放（例如使用 StandardScaler），因为它对输入特征的大小非常敏感。大多数正则化模型都是如此。

下面是如何使用sklearn执行闭式解的岭回归（使用的是公式4-9的一种变体，利用一种矩阵因式分解法）

公式4-9：闭式解的岭回归

$$\hat{\theta} = \left(\mathbf{X}^T \cdot \mathbf{X} + \alpha \mathbf{A}\right)^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

```
from sklearn.linear_model import Ridge
ridge_reg = Ridge(alpha=1,solver='cholesky')
ridge_reg.fit(X,y)
ridge_reg.predict([[1.5]])
```

使用随机梯度下降：

```
sgd_reg = SGDRegressor(penalty='l2')
sgd_reg.fit(X,y.ravel())
sgd_reg.predict([[1.5]])
```

超参数penalty设置的是使用正则项的类型。设为"l2"表示希望 SGD在成本函数中添加一个正则项，等于权重向量的l2范数的平方的 一半，即岭回归。

b. 套索回归

线性回归的另一种正则化，叫作最小绝对收缩和选择算子回归 （Least Absolute Shrinkage and Selection Operator Regression，简称 Lasso回归，或套索回归）。与岭回归一样，它也是向成本函数增加 一个正则项，但是它增加的是权重向量的l1范数，而不是l2范数的平方的一半（参见公式4-10）。

公式4-10：Lasso回归成本函数

$$J(\theta) = \mathrm{MSE}(\theta) + \alpha \sum_{i=1}^{n} |\theta_i|$$

Lasso回归的一个重要特点是它倾向于完全消除掉最不重要特征的权重（也就是将它们设置为零）。换句话说，Lasso回归会自动执行特征选择并输出一个稀疏模型（即只有很少的特征有非零权重）。下面是一个使用Scikit-Learn的Lasso类的小例子。你还可以使用 SGDRegressor（penalty="l1"）。

```
from sklearn.linear_model import Lasso
lasso_reg = Lasso(alpha=0.1)
lasso_reg.fit(X,y)
lasso_reg.predict([[1.5]])
```

c. 弹性网络

弹性网络是岭回归与Lasso回归之间的中间地带。其正则项就是岭回归与Lasso回归的正则项的混合，混合比例通过r来控制。当r=0时，弹性网络等于岭回归，而当r=1时，即相当于Lasso回归

公式4-12：弹性网络成本函数

$$J(\theta) = \mathrm{MSE}(\theta) + r\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^{n} \theta_i^2$$

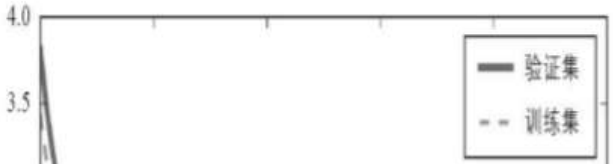
那么，到底如何选用线性回归、岭回归、Lasso回归和弹性网络呢？通常来说，有正则化——哪怕是很小，总是比没有更可取一些。所以大多数情况下，你应该避免使用纯线性回归。岭回归是个不错的默认选择，但是如果你觉得实际用到的特征只有少数几个，那就应该 更倾向于Lasso回归或是弹性网络，因为它们会将无用特征的权重降为零。一般而言，弹性网络优于Lasso回归，因为当特征数量超过训练实例数量，又或者是几个特征强相关时，Lasso回归的表现可能非常不稳定。下面是一个使用Scikit-Learn的ElasticNet的小例子（l1_ratio对应 混合比例r）：

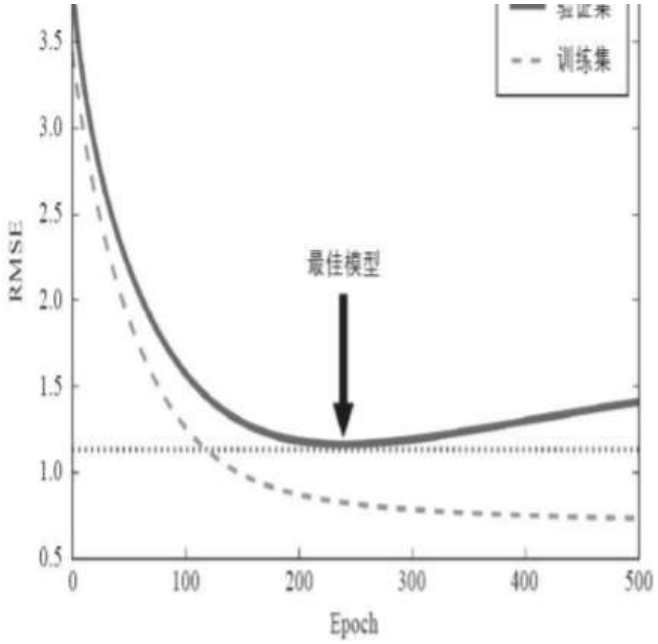
```
from sklearn.linear_model import ElasticNet
elastic_net = ElasticNet(alpha=0.1,l1_ratio=0.5)
elastic_net.fit(X,y)
elastic_net.predict([[1.5]])
```

```
array([5.23168991])
```

d. 早停法

对于梯度下降这一类迭代学习的算法，还有一个与众不同的正则 化方法，就是在验证误差达到最小值时停止训练，该方法叫作早期停 止法。图4-20展现了一个用批量梯度下降训练的复杂模型（高阶多项 式回归模型）。经过一轮一轮的训练，算法不断地学习，训练集上的 预测误差（RMSE）自然不断下降，同样其在验证集上的预测误差也 随之下降。但是，一段时间之后，验证误差停止下降反而开始回升。 这说明模型开始过度拟合训练数据。通过早期停止法，一旦验证误差 达到最小值就立刻停止训练。这是一个非常简单而有效的正则化技 巧，所以Geoffrey Hinton称其为“美丽的免费午餐”。





$$\frac{e^{\theta \cdot x}}{1+e^{\theta \cdot x}}$$

对随机梯度下降和小批量梯度下降来说，曲线没有这么平 滑，所以很难知道是否已经达到最小值。一种解决方法是等验证误差 超过最小值一段时间之后再停止（这时你可以确信模型不会变得更好了），然后将模型参数回滚到验证误差最小时的位置。下面是早期停止法的基本实现：

```
from sklearn.base import clone
sgd_reg = SGDRegressor(n_iter=1,warm_start=True,penalty=None)
minimum_val_error = float('inf')
best_epoch = None
best_model = None
for epoch in range(1000):
   sgd_reg.fit(X_train,y_train)
   y_val_predict = sgd_reg.predict(X_test)
   val_error = mean_squared_error(y_val_predict,y_test)
   if val_error<minimum_val_error:
      minimum_val_error = val_error
      best_epoch = epoch
      best_model = clone(sgd_reg)
```

注意，当warm_start=True时，调用fit（）方法，会从停下的地方 继续开始训练，而不会重新开始。

e. 逻辑回归

逻辑回归模型也是计算输入特征的加权和（加上偏置项），但是不同于线性回归模型直接输出结果，它输出的时结果的数理函数（参见公式4-13）

公式4-13：逻辑回归模型概率估算（向量化形式）

$$\hat{p} = h_\theta(\mathbf{x}) = \sigma(\theta^T \cdot \mathbf{x})$$

其中，

公式4-14：逻辑函数

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

一旦逻辑回归模型估算出实例x属于正类的概率p尖号，就可以做出预测y尖号

公式4-15：逻辑回归模型预测

$$\hat{y} = \begin{cases} 0 (\hat{p} < 0.5) \\ 1 (\hat{p} \geqslant 0.5) \end{cases}$$

整个训练集的成本函数即为所有训练实例的平均成本。它可以记 成一个单独的表达式（可以轻松验证），如公式4-17所示，这个函数 被称为log损失函数。

公式4-17：逻辑回归成本函数（log损失函数）

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

但是坏消息是，这个函数没有已知的闭式方程（不存在一个标准 方程的等价方程）来计算出最小化成本函数的θ值。而好消息是，这是个凸函数，所以通过梯度下降（或是其他任意优化算法）保证能够 找出全局最小值（只要学习率不是太高，你又能长时间等待）。公式 4-18给出了成本函数关于第j个模型参数θj的偏导数方程。

公式4-18：Logistic成本函数的偏导数

$$\frac{\partial}{\partial\theta_j}J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\sigma(\theta^T \cdot \mathbf{x}^{(i)}) - y^{(i)}\right)x_j^{(i)}$$

公式4-18与公式4-5看起来非常相似：计算出每个实例的预测误 差，并将其乘以第j个特征值，然后再对所有训练实例求平均值。一旦你有了包含所有偏导数的梯度向量就可以使用梯度下降算法了。就是这样，现在你知道如何训练逻辑模型了。对随机梯度下降，一次使 用一个实例；对小批量梯度下降，一次使用一个小批量。

f. 决策边界

这里我们用鸢尾植物数据集来说明逻辑回归。我们试试仅基于花瓣宽度这一个特征，创建一个分类器来检测 Virginica鸢尾花。首先加载数据：

```
from sklearn import datasets
iris = datasets.load_iris()
iris.keys()
```

```
dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names'])
```

```
x = iris['data'][:,3:]
y = (iris['target']==2).astype(np.int)
```

训练逻辑回归模型

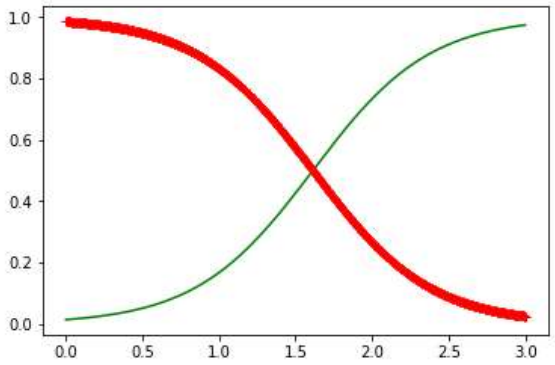
```
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(x,y)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
```

我们来看看对于花瓣宽度在0到3厘米之间的鸢尾花，模型估算出 的概率

```
X_new = np.linspace(0,3,1000).reshape(-1,1)
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new,y_proba[:,1],'g-',label='iris-virginica')
plt.plot(X_new,y_proba[:,0],'r--',label='Not iris-virginica')
```

```
[<matplotlib.lines.Line2D at 0x159b62cef28>]
```



控制Scikit-Learn的LogisticRegression模型正则化程度的超参 数不是alpha（其他线性模型使用alpha），而是它的逆反：C，C的值 越高，模型正则化程度越高

g. softmax回归

逻辑回归模型经过推广，可以直接支持多个类别，而不需要训练 并组合多个二元分类器（如第3章所述）。原理很简单：对于一个给定的实例x，Softmax回归模型首先计算 出每个类别k的分数sk（x），然后对这些分数应用softmax函数（也叫 归一化指数），估算出每个类别的概率。你应该很熟悉计算sk（x） 分数的公式（公式4-19），因为它看起来就跟线性回归预测的方程一 样。

公式4-19：类别k的Softmax分数

$$s_k(\mathbf{x}) = \theta_k^T \cdot \mathbf{x}$$

注意，每个类别都有自己特定的参数向量θk。所有这些向量通常 都作为行存储在参数矩阵Θ中。计算完实例x每个类别的分数后，就可以通过Softmax函数（公式 4-20）来计算分数：计算出每个分数的指数，然后对它们进行归一化 处理（除以所有指数的总和）即得到实例属于类别k的概 率。

公式4-20：Softmax函数

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^{K} \exp(s_j(\mathbf{x}))}$$

跟逻辑回归分类器一样，Softmax回归分类器将估算概率值最高 的类别作为预测类别（也就是分数最高的类别），如公式4-21所示。

公式4-21：Softmax回归分类器预测

$$\hat{y} = \underset{k}{\operatorname{argmax}}\, \sigma(\mathbf{s}(\mathbf{x}))_k = \underset{k}{\operatorname{argmax}}\, s_k(\mathbf{x}) = \underset{k}{\operatorname{argmax}} (\theta_k^T \cdot \mathbf{x})$$

softmax的训练目标是得到一个能对目标类别做出高概率估算 的模型（也就是其他类别的概率相应

要很低）。通过将公式4-22的成 本函数（也叫作交叉熵）最小化来实现这个目标，因为当模型对目标 类别做出较低概率的估算时，会受到惩罚。交叉熵经常被用于衡量一 组估算出的类别概率跟目标类别的匹配程度（后面的章节中还会多次 用到）。

公式4-22：交叉熵成本函数

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(\hat{p}_k^{(i)})$$

公式4-23给出了该成本函数关于θ_k的梯度向量：

公式4-23：对于类别k的交叉熵梯度向量

$$\nabla_{\theta_k} J(\Theta) = \frac{1}{m}\sum_{i=1}^{m} (\hat{p}_k^{(i)} - y_k^{(i)})\mathbf{x}^{(i)}$$

使用Softmax回归将鸢尾花分为三类。当用两个以上的类 别训练时，Scikit-Learn的LogisticRegressio默认选择使用的是一对多的训练方式，不过将超参数multi_class设置为"multinomial"，可以将其切换成Softmax回归。你还必须指定一个支持Softmax回归的求解 器，比如"lbfgs"求解器（详见Scikit-Learn文档）。默认使用l2正则 化，你可以通过超参数C进行控制。

```
X = iris['data'][:,(2,3)]
y = iris['target']

softmax_reg = LogisticRegression(multi_class='multinomial',solver='lbfgs',C=1.0)
softmax_reg.fit(X,y)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='multinomial',
          n_jobs=1, penalty='l2', random_state=None, solver='lbfgs',
          tol=0.0001, verbose=0, warm_start=False)
```

所以当你下次碰到一朵鸢尾花，花瓣长5厘米宽2厘米，你就可以 让模型告诉你它的种类，它会回答说：94.2%的概率是Virginica鸢尾 花（第2类）或者5.8%的概率为Versicolor鸢尾花：

```
softmax_reg.predict([[5,2]]),softmax_reg.predict_proba([[5,2]])
```

```
(array([2]), array([[2.42794737e-04, 2.14966717e-01, 7.84790488e-01]]))
```