

机器学习导论

第四章

王小航

本章概要

- ▶ 线性回归
 - ▶ 闭式 (closed-form) 方程
 - ▶ 梯度下降
 - ▶ 批量梯度下降
 - ▶ 小批量梯度下降
 - ▶ 随机梯度下降
- ▶ 多项式回归
- ▶ 逻辑 (Logistic) 回归
- ▶ Softmax 回归

机器学习导论

线性回归

- ▶ 在第1章中，我们学过一个简单的生活满意度的回归模型：
$$\text{life satisfaction} = \theta_0 + \theta_1 \times GDP$$
- ▶ 这个模型就是输入特征GDP的线性函数， θ_0 和 θ_1 是模型的参数
- ▶ 更为概括地说，线性模型就是对输入特征加权求和，再加上一个我们称为偏置项（也称为截距项）的常数，以此进行预测

线性回归

► 线性回归模型预测

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_p x_p$$

在此等式中：

- \hat{y} 是预测值。
- n 是特征数量。
- x_i 是第 i 个特征值。
- θ_j 是第 j 个模型参数（包括偏差项 θ_0 和特征权重 $\theta_1, \theta_2, \dots, \theta_p$ ）。

线性回归

- ▶ 线性回归模型预测（向量化形式）

$$\hat{y} = h_{\theta}(x) = \theta \cdot x$$

- ▶ θ 是模型的参数向量，其中包含偏差项 θ_0 和特征权重 θ_1 至 θ_p 。
- ▶ x 是实例的特征向量，包含从 x_0 到 x_p ， x_0 始终等于1。
- ▶ $\theta \cdot x$ 是向量 θ 和 x 的点积，它当然等于 $\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_p x_p$ 。
- ▶ h_{θ} 是假设函数，使用模型参数 θ 。

线性回归

- ▶ 线性回归模型的MSE成本函数：

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

- ▶ 均方误差MSE是均方根误差RMSE的平方
- ▶ 将均方误差（MSE）最小化比最小化RMSE更为简单，二者效果相同（因为使函数最小化的值，同样也使其平方根最小）

标准方程

- ▶ 线性回归有一个闭式解方法 (closed-form solution)，即标准方程：

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

这个方程中：

- ▶ $\hat{\theta}$ 是使成本函数最小的 θ 值。
- ▶ y 是包含 $y^{(1)}$ 到 $y^{(n)}$ 的目标值向量。
- ▶ X 为矩阵：

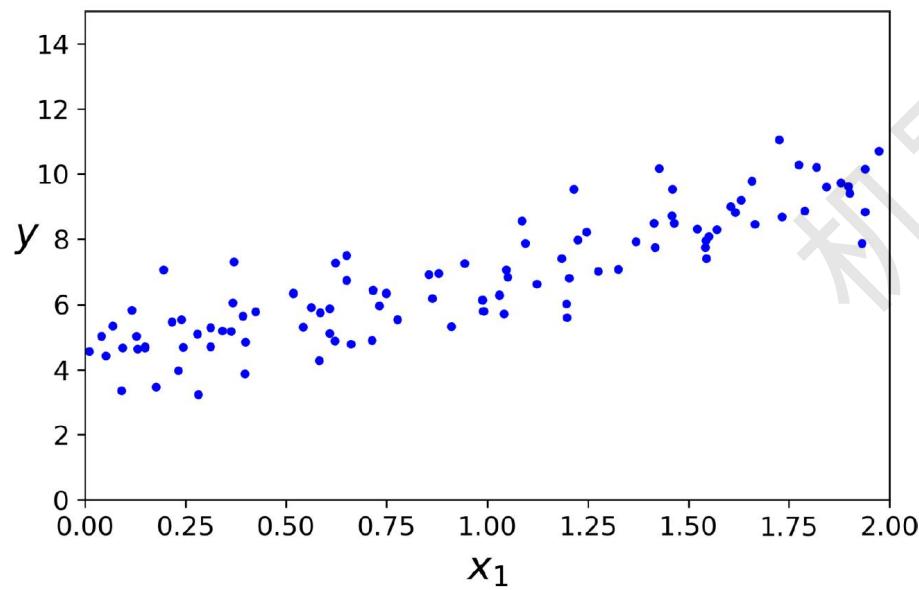
$$X = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1p} \\ 1 & x_{21} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{np} \end{pmatrix},$$

代码实现

- ▶ 生成一些线性数据：

```
import numpy as np

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



代码实现

- ▶ 现在我们使用标准方程来计算 $\hat{\theta}$ 。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] # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

- ▶ 参数估计结果（与 $y = 4 + 3x_1 + \epsilon$ 对比）：

```
>>> theta_best
array([[4.21509616],
       [2.77011339]])
```

预测

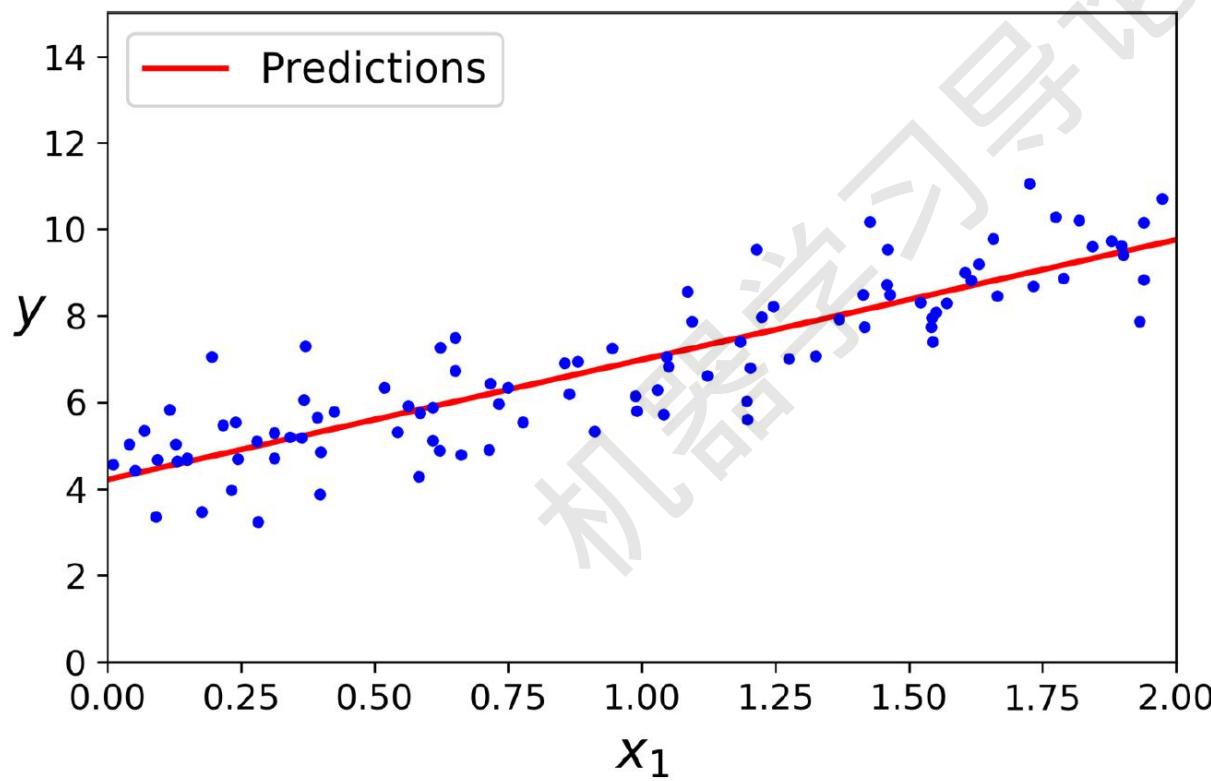
- ▶ 用 $\hat{\theta}$ 做出预测：

```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = np.c_[np.ones((2, 1)), X_new] # add x0 = 1 to each instance
>>> y_predict = X_new_b.dot(theta_best)
>>> y_predict
array([[4.21509616],
       [9.75532293]])
```

- ▶ 绘制模型的预测结果：

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```

预测



Scikit-Learn 实现

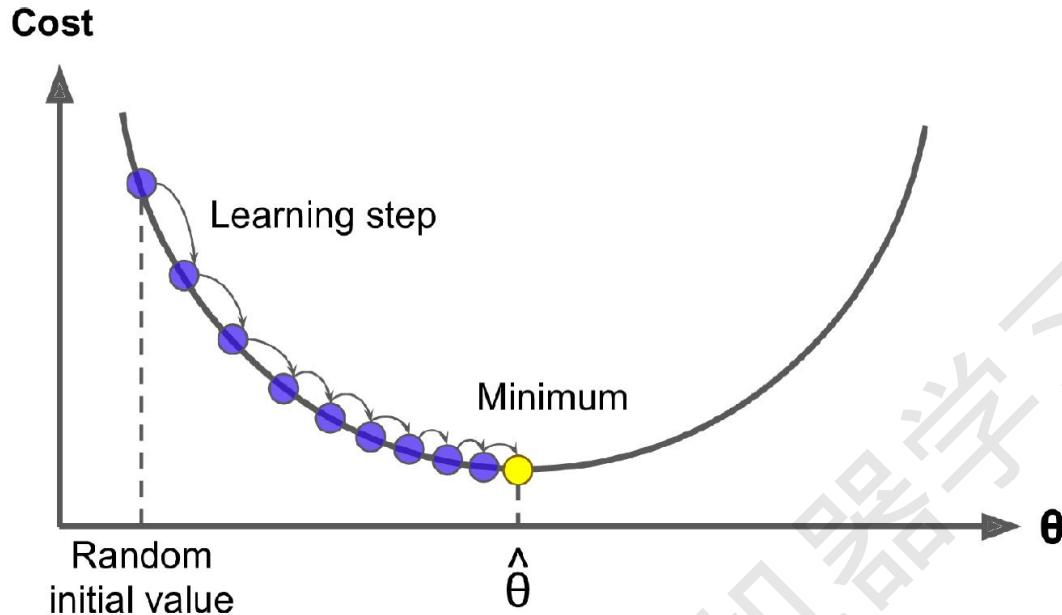
- ▶ 使用Scikit-Learn执行线性回归：

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([4.21509616]), array([[2.77011339]]))
>>> lin_reg.predict(X_new)
array([[4.21509616],
       [9.75532293]])
```

梯度下降

- ▶ 梯度下降是一种非常通用的优化算法，能够为大范围的问题找到最优解。梯度下降的中心思想就是迭代地调整参数从而使成本函数最小化
- ▶ 通过测量参数向量 θ 相关的误差函数的局部梯度，并不断沿着降低梯度的方向调整，直到梯度降为0，到达最小值
- ▶ 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



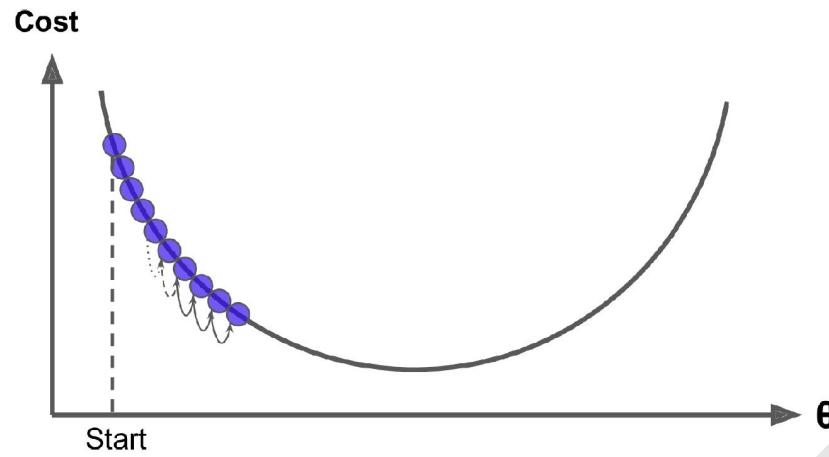
梯度下降步骤：

$$\theta^{(next\ step)} = \theta - \eta \nabla_{\theta} L(\theta)$$

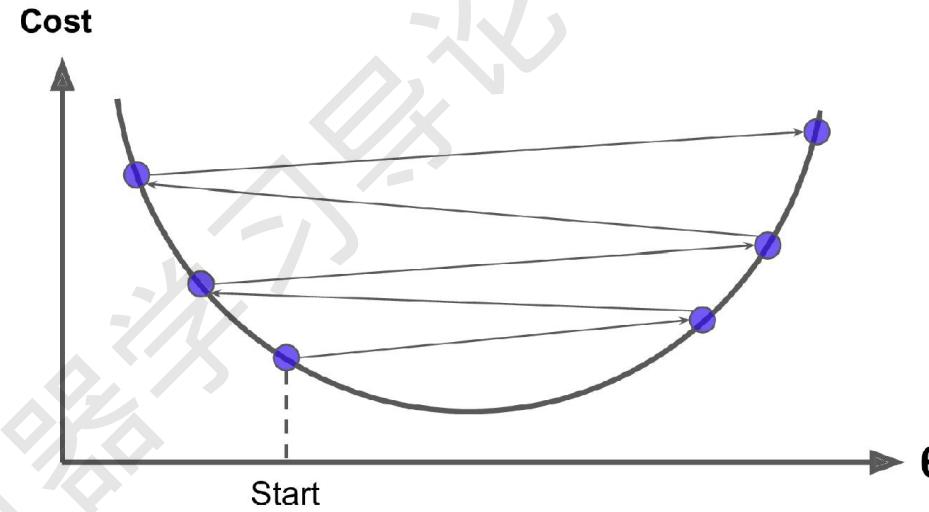
其中 η 为学习率， $L(\theta)$ 为损失函数

在梯度下降的描述中，模型参数被随机初始化并反复调整使成本函数最小化。学习步长与成本函数的斜率成正比，因此，当参数接近最小值时，步长逐渐变小

梯度下降

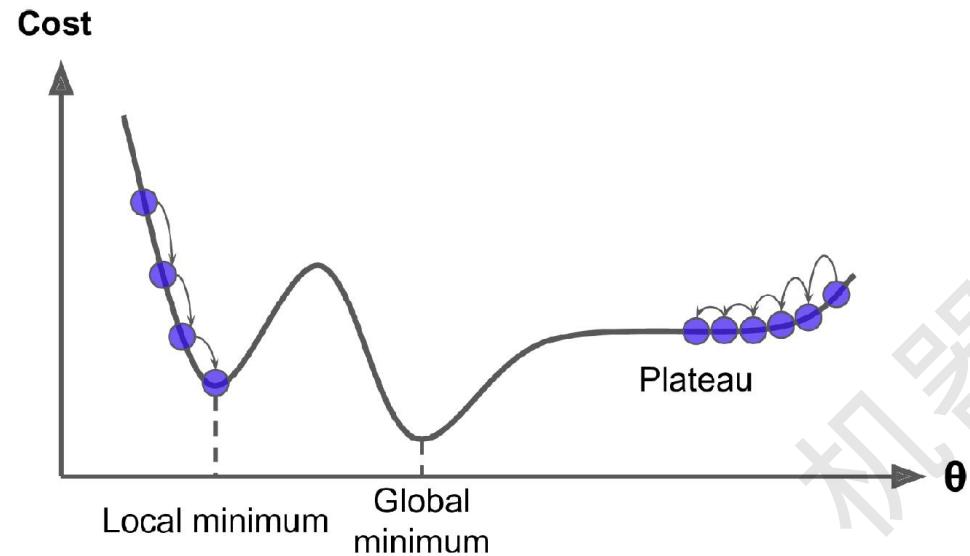


梯度下降中一个重要参数是每一步的步长，这取决于超参数学习率。如果学习率太低，算法需要经过大量迭代才能收敛，这将耗费很长时间。



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。

梯度下降



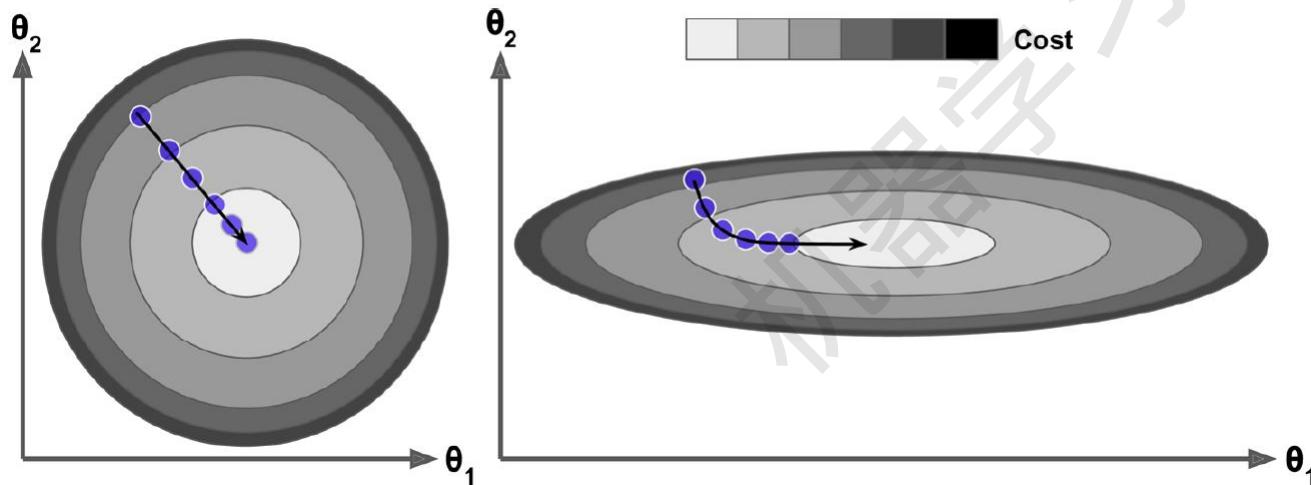
并不是所有的成本函数看起来都像一个漂亮的碗。有的可能看着像洞、山脉、高原或者各种不规则的地形，导致很难收敛到最小值。左图显示了梯度下降的两个主要挑战：如果随机初始化，算法从左侧起步，那么会收敛到一个局部最小值，而不是全局最小值。如果算法从右侧起步，那么需要经过很长时间才能越过整片高原，如果你停下得过早，将永远达不到全局最小值。

梯度下降

- ▶ 线性回归模型的MSE成本函数恰好是个凸函数，不存在局部最小值，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
- ▶ 这两点保证的结论是：即便是乱走，梯度下降都可以趋近到全局最小值

梯度下降

- 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



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 θ_j 的成本函数的梯度，也即偏导数。
- ▶ 成本函数的偏导数：

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

批量梯度下降

- ▶ 如果不想单独计算这些偏导数，可以使用以下公式对其进行一次性计算。梯度向量记作 $\nabla_{\theta}MSE(\theta)$ ，包含所有成本函数（每个模型参数一个）的偏导数

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

批量梯度下降

- ▶ 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。
- ▶ 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。

批量梯度下降

▶ 梯度下降步骤:

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

其中 η 为学习率

▶ 代码实现:

```
eta = 0.1 # learning rate
n_iterations = 1000
m = 100

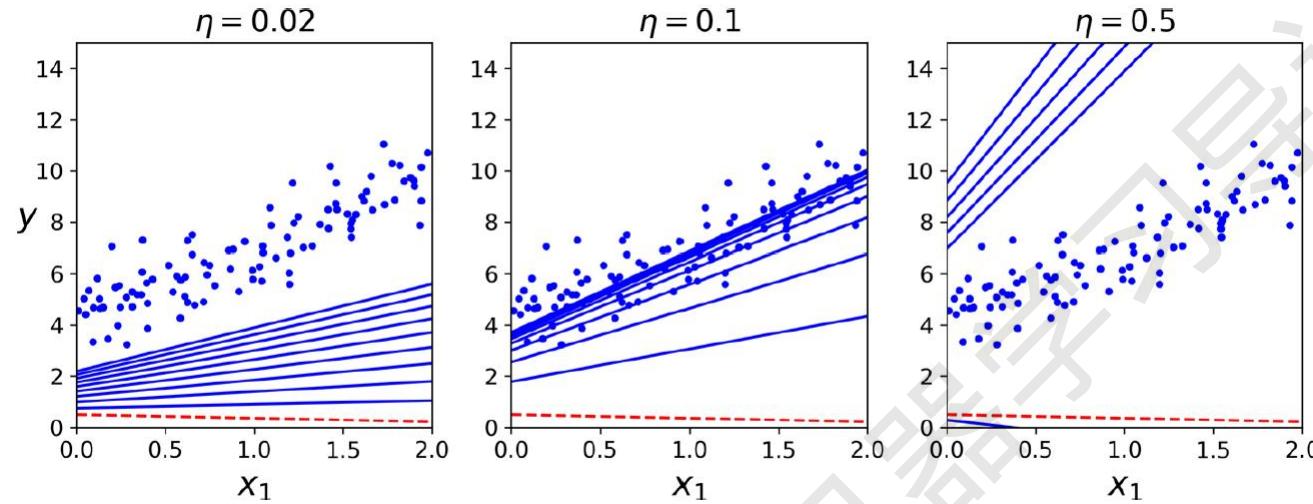
theta = np.random.randn(2,1) # random initialization

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

估计结果:

```
>>> theta
array([[4.21509616],
       [2.77011339]])
```

批量梯度下降

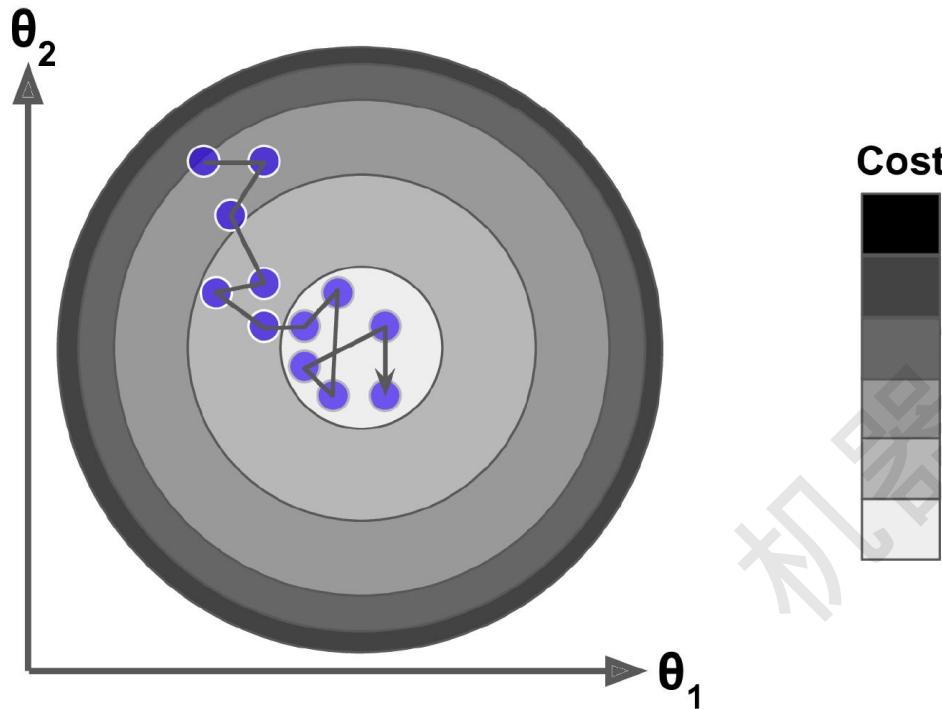


- 图展现了分别使用三种不同的学习率时，梯度下降的前十步（虚线表示起点）
- 左图的学习率太低：算法最终还是能找到解决方法，就是需要太长时间。
- 中间图的学习率看起来非常棒：几次迭代就收敛出了最终解。
- 右图的学习率太高：算法发散，直接跳过了数据区域，并且每一步都离实际解决方案越来越远。

随机梯度下降

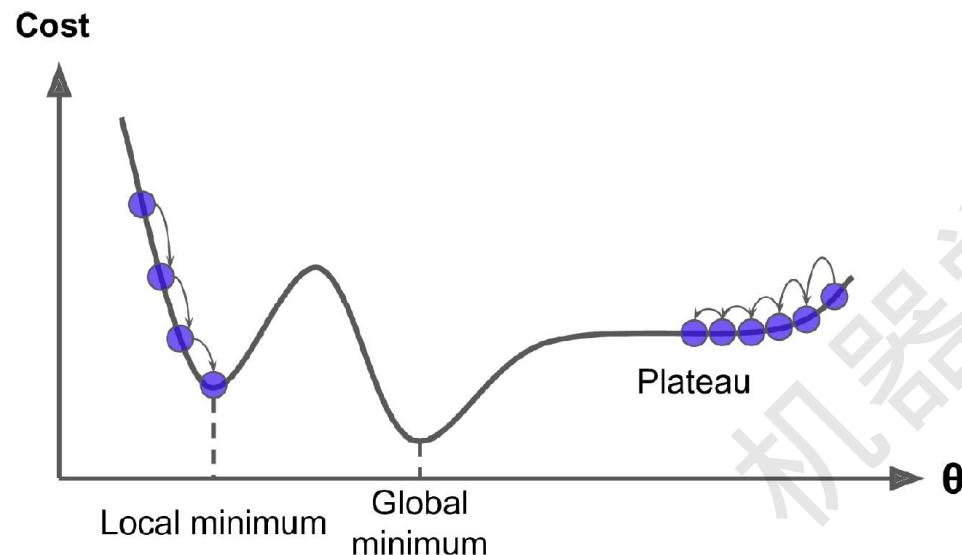
- ▶ 批量梯度下降的主要问题是它要用整个训练集来计算每一步的梯度，所以训练集很大时，算法会特别慢。
- ▶ 与之相反的极端是随机梯度下降，每一步在训练集中随机选择一个实例，并且仅基于该单个实例来计算梯度。
- ▶ 由于算法的随机性质，随机梯度比批量梯度下降要不规则得多。
- ▶ 成本函数将不再是缓缓降低直到抵达最小值，而是不断上上下下，但是从整体来看，还是在慢慢下降

随机梯度下降



- 随着时间的推移，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。
- 所以算法停下来参数值肯定是足够好的，但不是最优的。

随机梯度下降



当成本函数非常不规则时，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。

随机梯度下降

- ▶ 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**
- ▶ 确定每个迭代学习率的函数叫作**学习率调度**。
- ▶ 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。

代码实现

```
n_epochs = 50
t0, t1 = 5, 50 # learning schedule hyperparameters

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

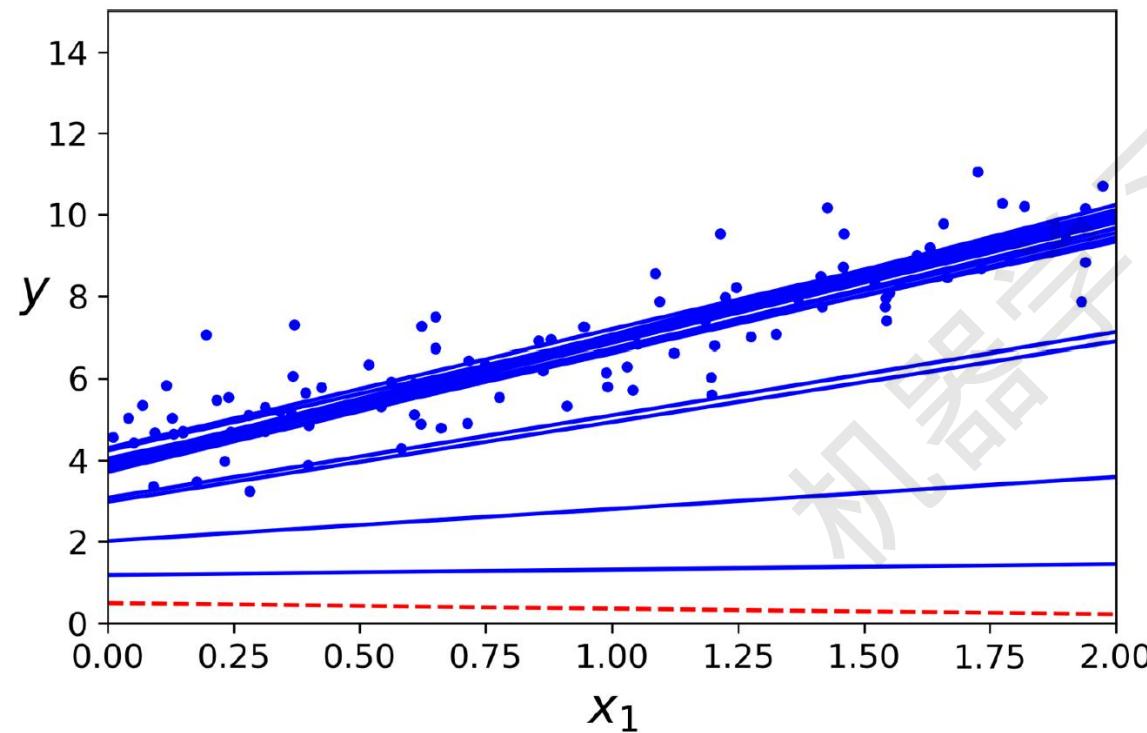
theta = np.random.randn(2,1) # random initialization

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
```

按照惯例，我们进行 m 个回合的迭代。每个回合称为一个轮次。虽然批量梯度下降代码在整个训练集中进行了1000次迭代，但此代码仅在训练集中遍历了50次，并达到了一个很好的解决方案

```
>>> theta
array([[4.21076011],
       [2.74856079]])
```

随机梯度下降



- 由于实例是随机选取的，因此某些实例可能每个轮次中被选取几次，而其他实例则可能根本不被选取。
- 如果要确保算法在每个轮次都遍历每个实例，则另一种方法是对训练集进行混洗（确保同时对输入特征和标签进行混洗），然后逐个实例进行遍历，然后对其进行再次混洗，以此类推。但是，这种方法通常收敛较慢

Scikit-Learn 实现

- ▶ 使用SGDRegressor类，该类默认优化平方误差成本函数。

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

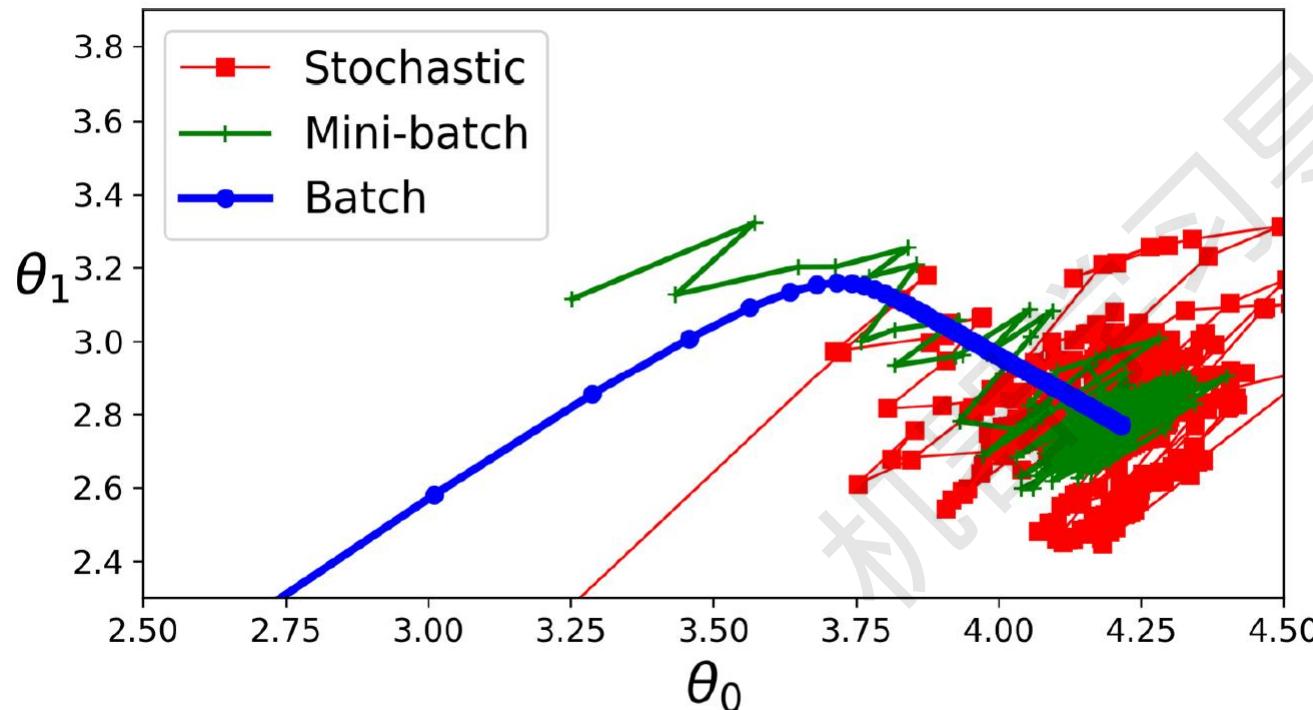
- ▶ 以上代码最多可运行1000个轮次，或者直到一个轮次期间损失下降小于0.001为止 (max_iter=1000, tol=1e-3)。
- ▶ 它使用默认的学习调度（与前一个学习调度不同）以0.1 (eta0=0.1) 的学习率开始。
- ▶ 最后，它不使用任何正则化

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([4.24365286]), array([2.8250878]))
```

小批量梯度下降

- ▶ 在每一步中，不是根据完整的训练集（如批量梯度下降）或仅基于一个实例（如随机梯度下降）来计算梯度，小批量梯度下降在称为小型批量的随机实例集上计算梯度
- ▶ 与随机梯度下降相比，该算法在参数空间上的进展更稳定，尤其是在相当大的小批次中
- ▶ 小批量梯度下降最终将比随机梯度下降走得更接近最小值，但它可能很难摆脱贫局部分最小值

小批量梯度下降



图显示了训练期间参数空间中三种梯度下降算法所采用的路径。它们最终都接近最小值，但是批量梯度下降的路径实际上是在最小值处停止，而随机梯度下降和小批量梯度下降都继续走动。

多项式回归

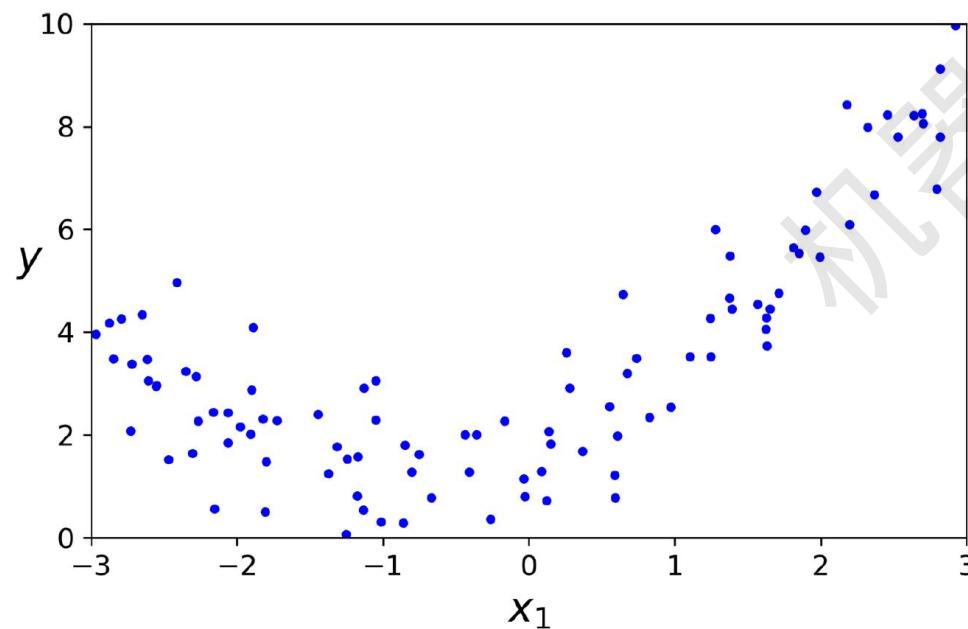
- ▶ 可以使用线性模型来拟合非线性数据
- ▶ 一个简单的方法就是将每个特征的幂次方添加为一个新特征，然后在此扩展特征集上训练一个线性模型
- ▶ 这种技术称为 **多项式回归**
- ▶ 一元m次多项式回归方程为：

$$\hat{y} = \theta_0 + \theta_1 x + \theta_2 x^2 + \dots + \theta_p x^p$$

代码实现

- ▶ 基于一个简单的二次方程式（注：二次方程的形式为 $y = ax^2 + bx + c$ 。）
(加上一些噪声，见下图) 生成一些非线性数据：

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



Scikit-Learn 实现

- ▶ 一条直线永远无法正确地拟合此数据。
- ▶ 因此，让我们使用Scikit-Learn的PolynomialFeatures类来转换训练数据，将训练集中每个特征的平方（二次多项式）添加为新特征

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929,  0.56664654])
```

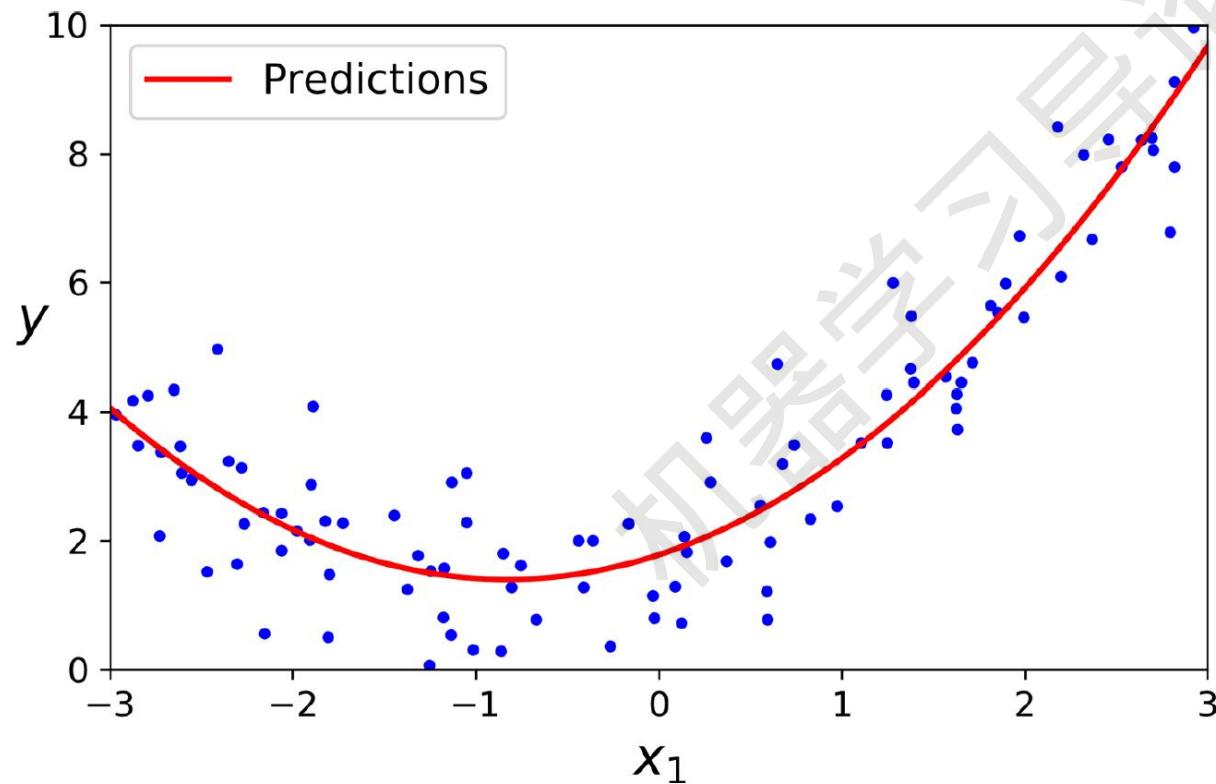
Scikit-Learn 实现

- ▶ X_{poly} 现在包含 X 的原始特征以及该特征的平方。现在，你可以将 `LinearRegression` 模型拟合到此扩展训练数据中

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

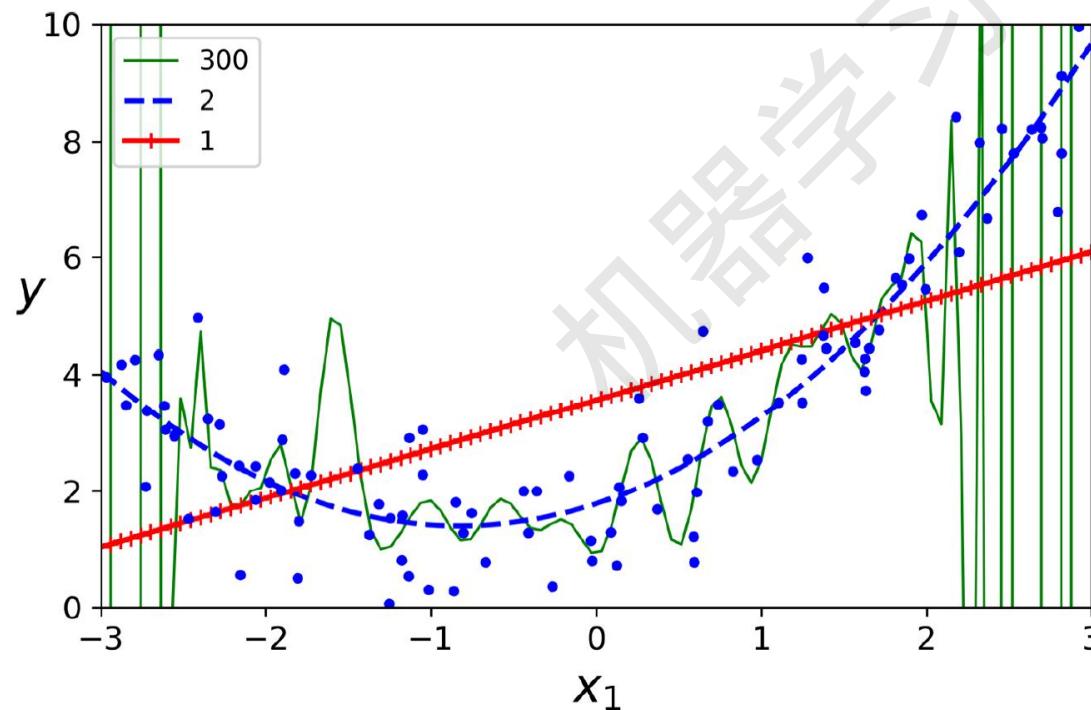
- ▶ 模型估算 $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$ ，而实际上原始函数为 $y = 0.5x_1^2 + 1.0x_1 + 2 + \text{高斯噪声}$

Scikit-Learn 实现



学习曲线

- ▶ 你如果执行高阶多项式回归，与普通线性回归相比，拟合数据可能会更好。
- ▶ 例如，下图将300阶多项式模型应用于先前的训练数据，将结果与纯线性模型和二次模型（二次多项式）进行比较。



学习曲线

- ▶ 这种高阶多项式回归模型严重过拟合训练数据，而线性模型则欠拟合。
- ▶ 在这种情况下，最能泛化的模型是二次模型，因为数据是使用二次模型生成的。
- ▶ 但是总的来说，你不知道数据由什么函数生成，那么如何确定模型的复杂性呢？
- ▶ 你如何判断模型是过拟合数据还是欠拟合数据呢？

学习曲线

- ▶ 在第2章中，你使用交叉验证来估计模型的泛化性能。
- ▶ 如果模型在训练数据上表现良好，但根据交叉验证的指标泛化较差，则你的模型过拟合。
- ▶ 如果两者的表现均不理想，则说明欠拟合。这是一种区别模型是否过于简单或过于复杂的方法。
- ▶ 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（或训练迭代）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

代码实现

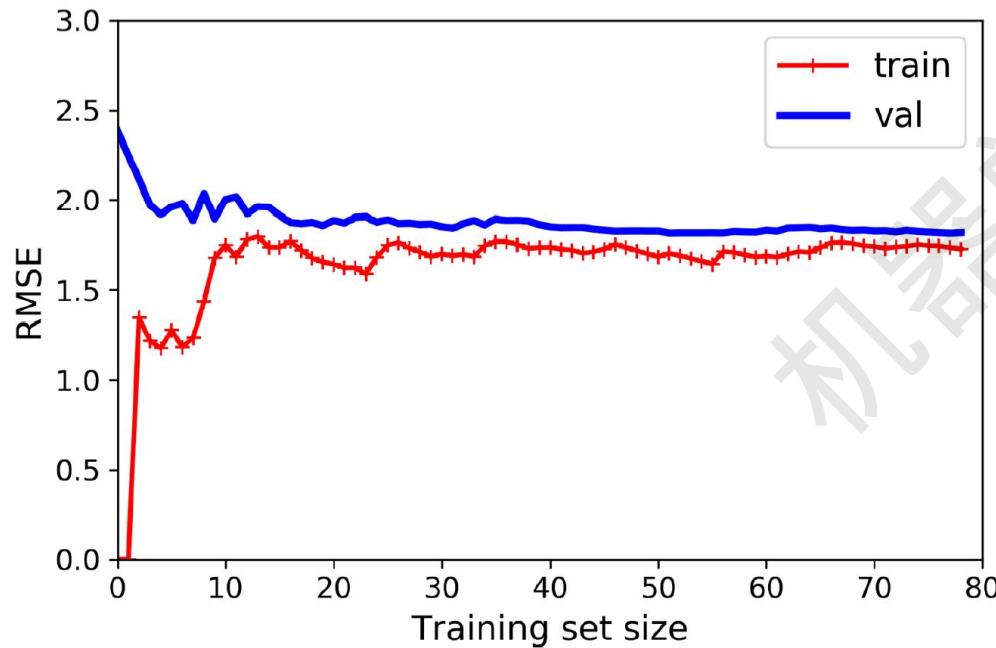
```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

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[:m], y_train_predict))
        val_errors.append(mean_squared_error(y_val, y_val_predict))
    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
```

代码实现

▶ 普通线性回归模型的学习曲线：

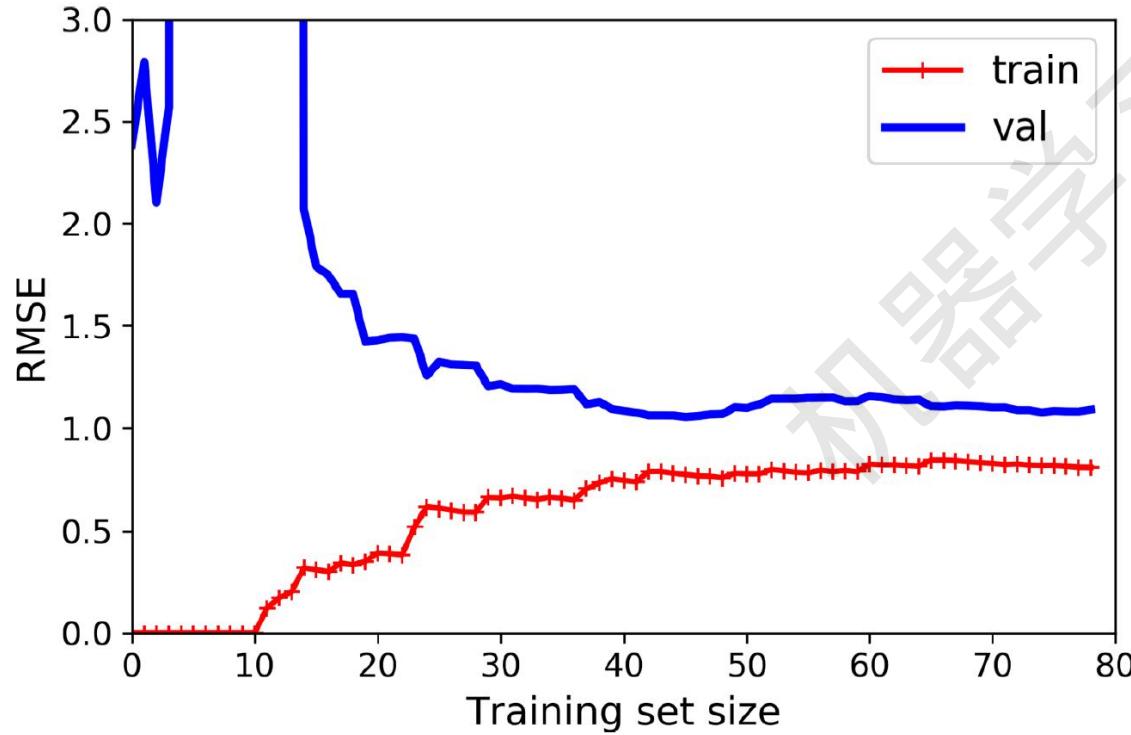
```
lin_reg = LinearRegression()  
plot_learning_curves(lin_reg, X, y)
```



- 当训练集中只有一个或两个实例时，模型可以很好地拟合它们。随着将新实例添加到训练集中，模型就不可能完美地拟合训练数据，这既因为数据有噪声，又因为它根本不是线性的。因此，训练数据上的误差会一直上升，直达到到平稳状态。
- 当在很少的训练实例上训练模型时，它无法正确泛化，这就是验证误差最初很大的原因。然后，随着模型经历更多的训练示例，它开始学习，因此验证错误逐渐降低。但是，直线不能很好地对数据进行建模，因此误差最终达到一个平稳的状态，非常接近另外一条曲线。

学习曲线

- ▶ 在相同数据上的10阶多项式模型的学习曲线



- 与线性回归模型相比，训练数据上的误差要低得多。
- 曲线之间存在间隙。这意味着该模型在训练数据上的性能要比在验证数据上的性能好得多，这是过拟合模型的标志。但是，如果你使用更大的训练集，则两条曲线会继续接近。

正则化线性模型

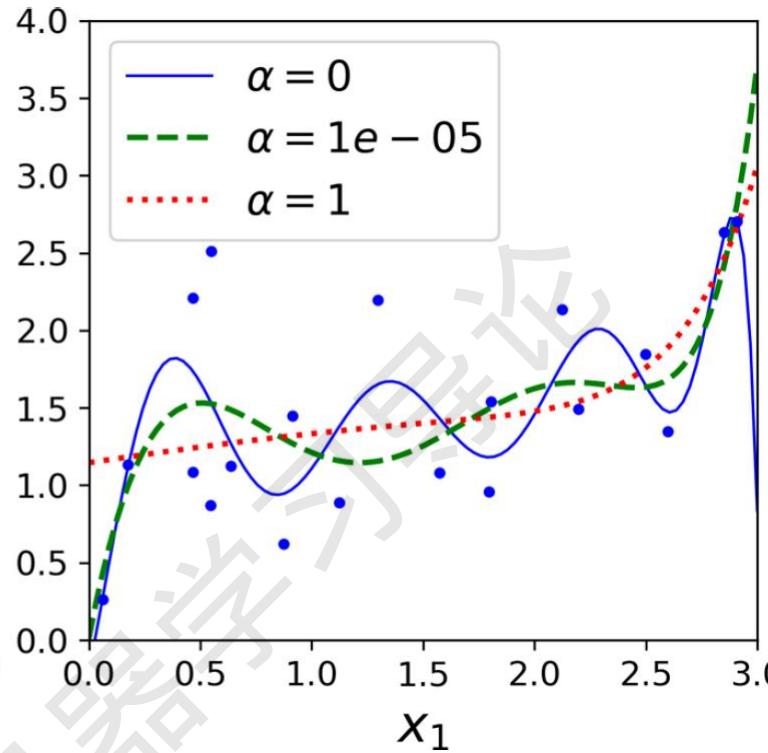
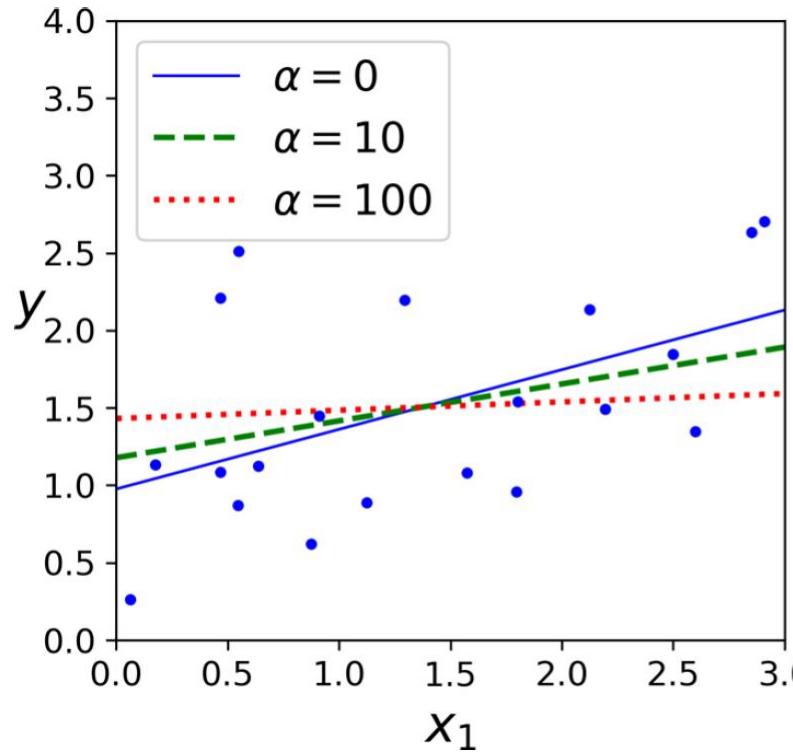
- ▶ 正如我们在第1章和第2章中看到的那样，减少过拟合的一个好方法是对模型进行正则化（即约束模型）：它拥有的自由度越少，则过拟合数据的难度就越大。
 - ▶ 岭回归 (Ridge Regression)
 - ▶ Lasso回归 (Lasso Regression)
 - ▶ 弹性网络 (Elastic Net)

岭 回 归

- ▶ 岭回归成本函数：

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

- ▶ 将正则化项添加到成本函数，迫使学习算法不仅拟合数据，而且使模型权重尽可能小
- ▶ 超参数 α 控制要对模型进行正则化的程度。如果 $\alpha=0$ ，则岭回归仅是线性回归。如果 α 非常大，则所有权重最终都非常接近于零，结果是一条经过数据均值的平线
- ▶ 偏置项 θ_0 没有进行正则化
- ▶ 在执行岭回归之前缩放数据（例如使用StandardScaler）很重要，因为它对输入特征的缩放敏感。大多数正则化模型都需要如此



左侧使用普通岭模型，导致了线性预测。在右侧，首先使用 `PolynomialFeatures (degree=10)` 扩展数据，然后使用 `StandardScaler` 对其进行缩放，最后将岭模型应用于结果特征：这是带有岭正则化的多项式回归。请注意， α 的增加会导致更平坦的预测。

岭回归的闭式解

► 岭回归的闭式解：

$$\hat{\theta} = (X^T X + \alpha I)^{-1} X^T y$$

```
from sklearn.linear_model import Ridge
m = 20
X = 3 * np.random.rand(m, 1)
y = 1 + 0.5 * X + np.random.randn(m, 1) / 1.5
ridge_reg = Ridge(alpha=1, solver="cholesky")
ridge_reg.fit(X, y)
ridge_reg.predict([[1.5]])
```

随机梯度下降法

▶ 使用随机梯度下降法：

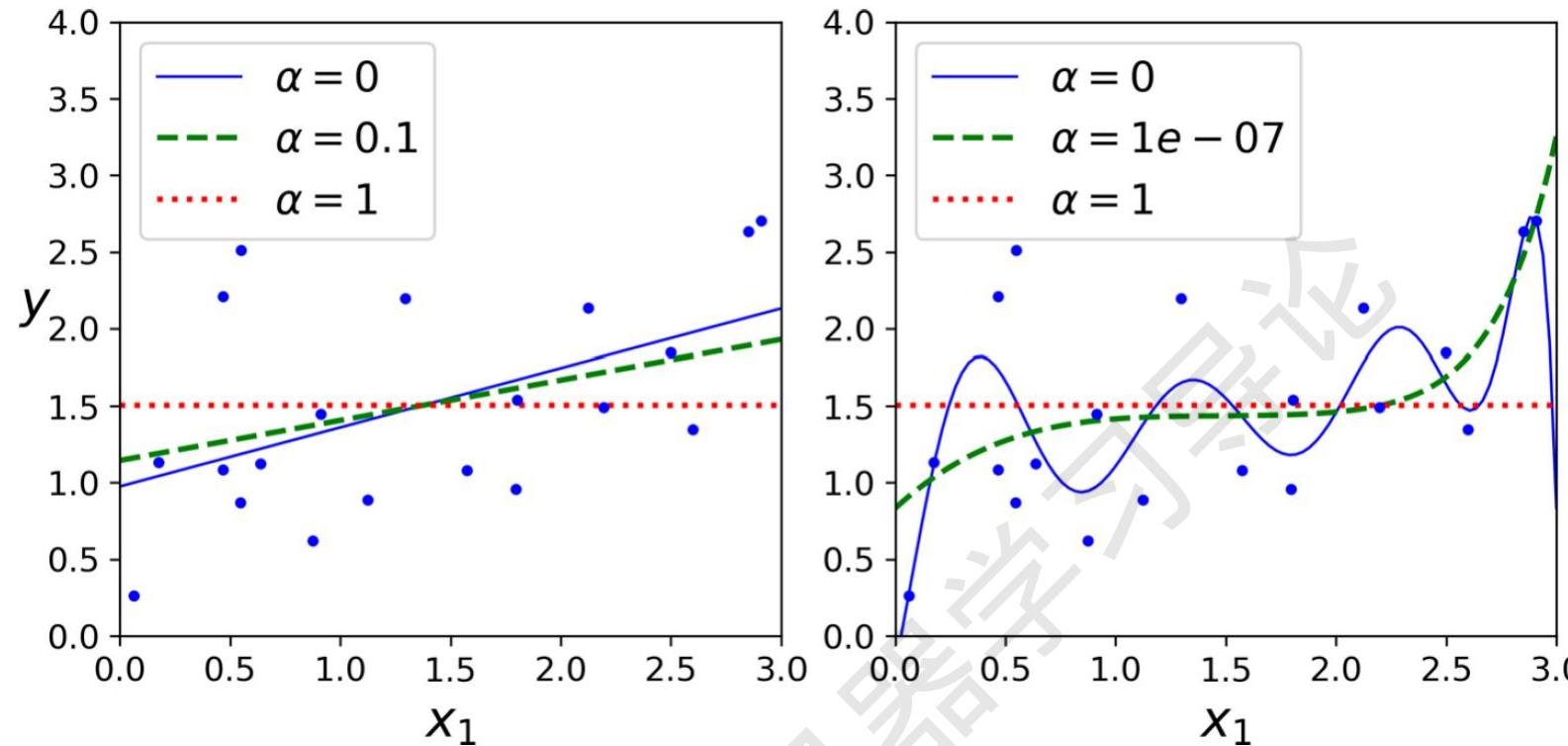
```
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([1.47012588])
```

Lasso 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)
- ▶ Lasso回归成本函数：

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

- ▶ 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



右图中的虚线 ($\alpha = 10^{-7}$) 看起来像是二次的，快要接近于线性：因为所有高阶多项式的特征权重都等于零。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）

Lasso的代码实现

- ▶ Lasso没有闭式解
- ▶ 使用Scikit-Learn拟合Lasso回归：

```
>>> from sklearn.linear_model import Lasso  
>>> lasso_reg = Lasso(alpha=0.1)  
>>> lasso_reg.fit(X, y)  
>>> lasso_reg.predict([[1.5]])  
array([1.53788174])
```

- ▶ 你可以改用SGDRegressor (penalty="l1")

弹性网络

- ▶ 弹性网络是介于岭回归和Lasso回归之间的中间地带。
- ▶ 正则项是岭和Lasso正则项的简单混合，你可以控制混合比r。
- ▶ 当r=0时，弹性网络等效于岭回归，而当r=1时，弹性网络等效于Lasso回归
- ▶ 弹性网络成本函数

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

Scikit-Learn的代码实现

```
>>> 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([1.54333232])
```

逻辑回归

- ▶ 逻辑回归 (Logistic回归, 也称为Logit回归) 被广泛用于估算一个实例属于某个特定类别的概率。
- ▶ 比如, 这封电子邮件属于垃圾邮件的概率是多少?
- ▶ 如果预估概率超过50%, 则模型预测该实例属于该类别 (称为正类, 标记为“1”), 反之, 则预测不是 (称为负类, 标记为“0”)。
- ▶ 这样它就成了一个二元分类器。

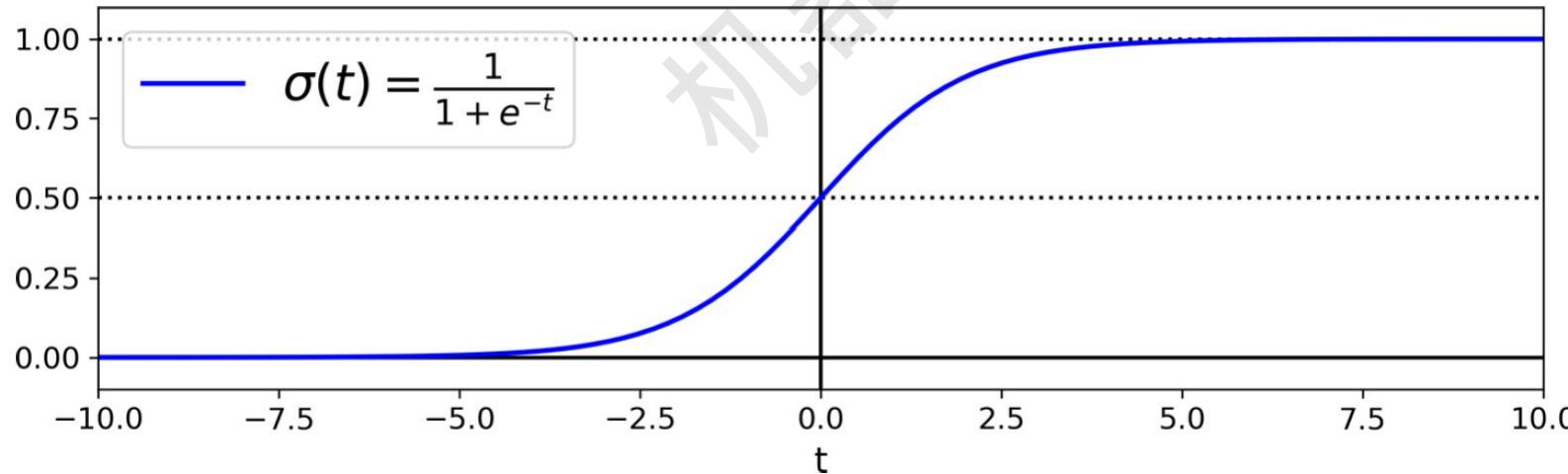
模 型

- ▶ 逻辑回归模型的估计概率（向量化形式）：

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

- ▶ 逻辑记为 $\sigma(\cdot)$ ，是一个sigmoid函数（即S型函数），输出一个介于0和1之间的数字。
- ▶ 逻辑函数：

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



预测

- ▶ 一旦逻辑回归模型估算出实例 x 属于正类的概率 $\hat{p} = h_{\theta}(x)$, 就可以做出预测 \hat{y}
- ▶ 逻辑回归模型预测

$$\hat{y} = \begin{cases} 0, & \text{如果 } \hat{p} < 0.5 \\ 1, & \text{如果 } \hat{p} \geq 0.5 \end{cases}$$

- ▶ 注意, 当 $t < 0$ 时, $\sigma(t) < 0.5$; 当 $t \geq 0$ 时, $\sigma(t) \geq 0.5$ 。所以如果 $x^T \theta$ 是正类, 逻辑回归模型预测结果是1, 如果是负类, 则预测为0。

训练和成本函数

- ▶ 逻辑回归成本函数（对数损失）：

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

- ▶ 这个函数没有已知的闭式方程。
- ▶ 这是个凸函数，所以通过梯度下降保证能够找出全局最小值
- ▶ 逻辑成本函数偏导数：

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

成本函数偏导数的推导

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

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \frac{1}{\hat{p}^{(i)}} - (1 - y^{(i)}) \frac{1}{1 - \hat{p}^{(i)}}] \frac{\partial \hat{p}^{(i)}}{\partial \boldsymbol{\theta}}$$

其中, $\hat{p}^{(i)} = \sigma(\mathbf{x}^{(i)T} \boldsymbol{\theta}) = \frac{1}{1 + \exp(-\mathbf{x}^{(i)T} \boldsymbol{\theta})}$

$$\frac{\partial \hat{p}^{(i)}}{\partial \boldsymbol{\theta}} = -\frac{\exp(-\mathbf{x}^{(i)T} \boldsymbol{\theta})}{(1 + \exp(-\mathbf{x}^{(i)T} \boldsymbol{\theta}))^2} (-\mathbf{x}^{(i)}_j) = \hat{p}^{(i)}(1 - \hat{p}^{(i)})\mathbf{x}^{(i)}_j$$

所以,
$$\begin{aligned} \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} &= -\frac{1}{m} \sum_{i=1}^m [y^{(i)}(1 - \hat{p}^{(i)}) - (1 - y^{(i)})\hat{p}^{(i)}] \mathbf{x}^{(i)}_j \\ &= \frac{1}{m} \sum_{i=1}^m (\sigma(\boldsymbol{\theta}^T \mathbf{x}^{(i)}) - y^{(i)}) \mathbf{x}^{(i)}_j \end{aligned}$$

决策边界

- ▶ 鸢尾植物 (iris) 数据集
- ▶ 这是一个非常著名的数据集，共有150朵鸢尾花
- ▶ 分别来自三个不同品种 (山鸢尾、变色鸢尾和维吉尼亚鸢尾)
 - ▶ Iris setosa
 - ▶ Iris versicolor
 - ▶ Iris virginica
- ▶ 数据里包含花的萼片以及花瓣的长度和宽度
 - ▶ sepal length in cm
 - ▶ sepal width in cm
 - ▶ petal length in cm
 - ▶ petal width in cm



Figure 4-22. Flowers of three iris plant species¹⁴

The famous Iris database, first used by Sir R.A. Fisher. The dataset is taken from Fisher's paper. Note that it's the same as in R, but not as in the UCI Machine Learning Repository, which has two wrong data points.

代码实现

- 仅基于花瓣宽度这一个特征，创建一个分类器来检测维吉尼亚鸢尾花

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']

>>> X = iris["data"][:, 3:] # petal width
>>> y = (iris["target"] == 2).astype(np.int) # 1 if Iris virginica, else 0
```

- 训练一个逻辑回归模型：

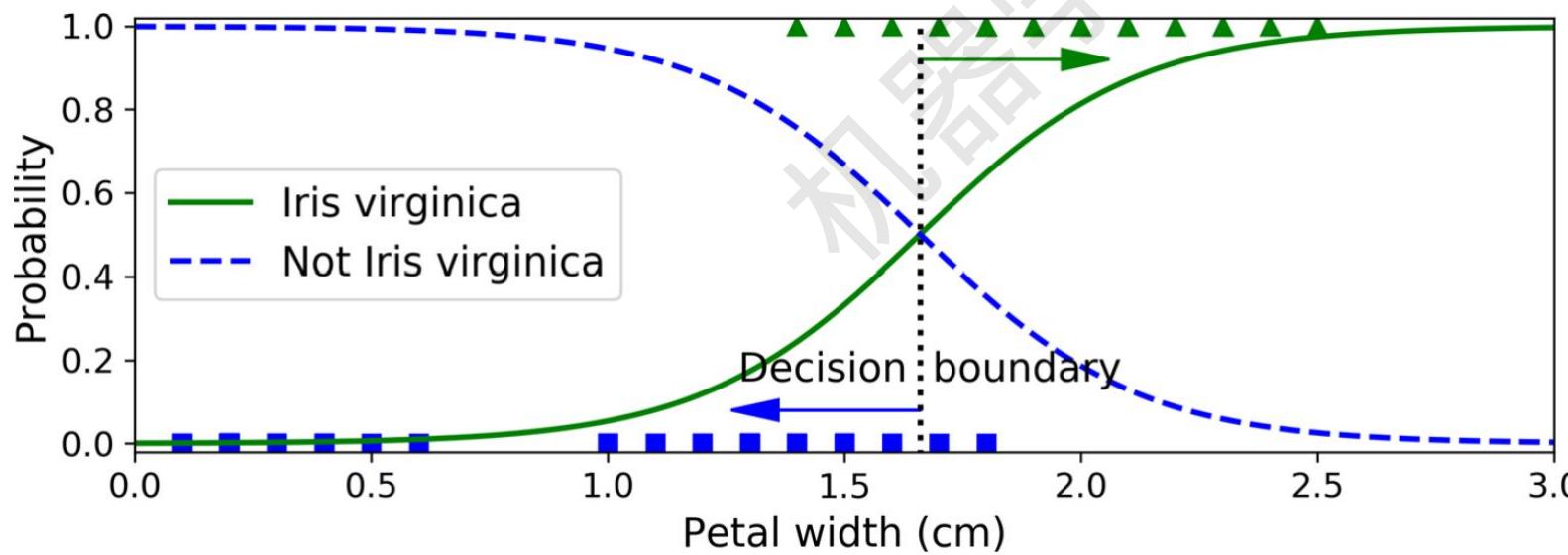
```
from sklearn.linear_model import LogisticRegression

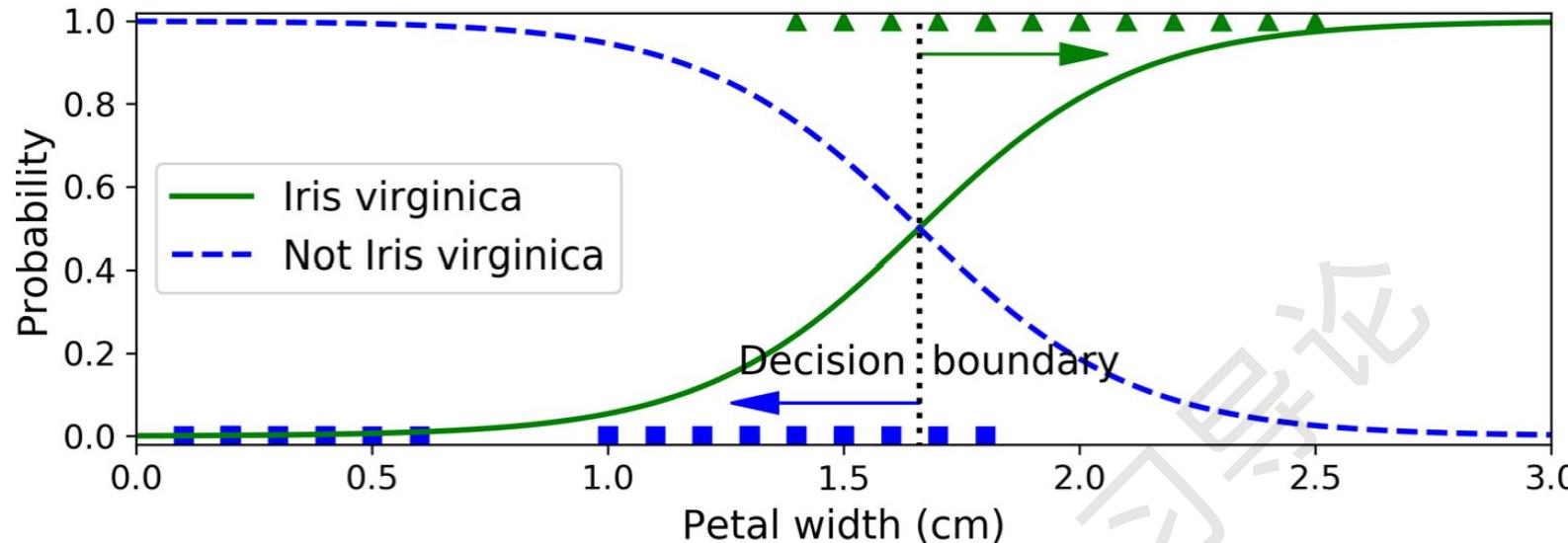
log_reg = LogisticRegression()
log_reg.fit(X, y)
```

决策边界

- ▶ 花瓣宽度在0到3cm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], "b--", label="Not Iris virginica")
# + more Matplotlib code to make the image look pretty
```



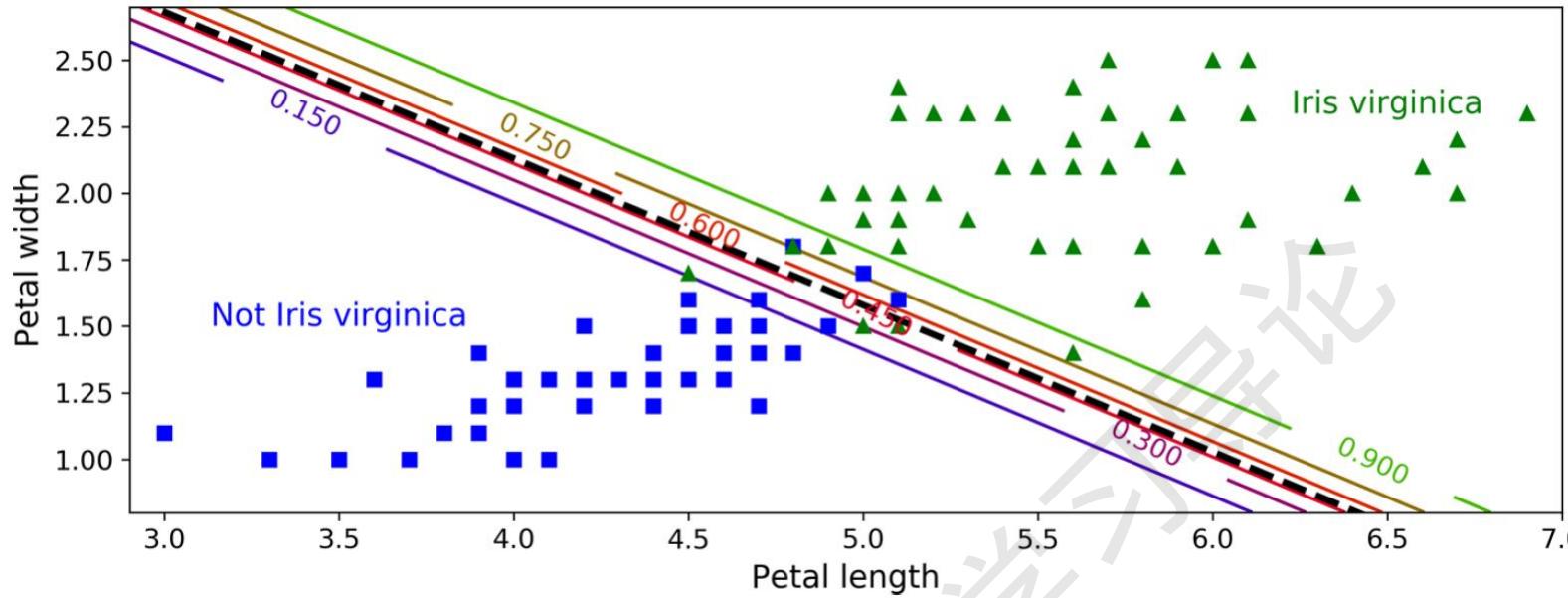


- ▶ 维吉尼亚鸢尾（三角形所示）的花瓣宽度范围为1.4~2.5cm，而其他两种鸢尾花（正方形所示）花瓣通常较窄，花瓣宽度范围为0.1~1.8cm。
- ▶ 这里有一部分重叠。对花瓣宽度超过2cm的花，分类器可以很有信心地说它是一朵维吉尼亚鸢尾花（对该类别输出一个高概率值），对花瓣宽度低于1cm以下的，也可以胸有成竹地说其不是（对“非维吉尼亚鸢尾”类别输出一个高概率值）

预测

```
>>> log_reg.predict([[1.7], [1.5]])
array([1, 0])
```

机器学习导论



- ▶ 还是同样的数据集，但是这次显示了两个特征：花瓣宽度和花瓣长度。经过训练，这个逻辑回归分类器就可以基于这两个特征来预测新花朵是否属于维吉尼亚鸢尾。
- ▶ 虚线表示模型估算概率为50%的点，即模型的决策边界。注意这里是一个线性的边界（注：这是点 X 的集合，使得 $\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$ ，它定义了一条直线。）
- ▶ 每条平行线都分别代表一个模型输出的特定概率，从左下的15%到右上的90%

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）。
- ▶ 这就是Softmax回归，或者叫作 **多元逻辑回归**。
- ▶ 给定一个实例 x ，Softmax回归模型首先计算出每个类 k 的分数 $s_k(x)$ ，然后对这些分数应用softmax函数（也叫归一化指数），估算出每个类的概率

Softmax 回归

- ▶ 类 k 的 Softmax 分数：

$$s_k(x) = x^T \theta^{(k)}$$

每个类都有自己的特定参数向量 $\theta^{(k)}$ 。

所有这些向量通常都作为行存储在参数矩阵 Θ 中

- ▶ Softmax 函数：

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

- ▶ Softmax 回归分类预测：

$$\hat{y} = \arg \max_k \sigma(s(x))_k = \arg \max_k s_k(x) = \arg \max_k ((\theta^{(k)})^T x)$$

训练模型

- ▶ 通过将成本函数（也叫作交叉熵）最小化来实现这个目标
- ▶ 交叉熵成本函数：

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

$y_k^{(i)}$ 是属于类 k 的第 i 个实例的目标概率。一般而言等于 1 或 0，具体取决于实例是否属于该类。

- ▶ 当只有两个类 ($K=2$) 时，此成本函数等效于逻辑回归的成本函数

训练模型

- ▶ 类k的交叉熵梯度向量：

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

- ▶ 可以计算每个类的梯度向量，然后使用梯度下降来找到最小化成本函数的参数矩阵 Θ
- ▶ 参考<https://zhuanlan.zhihu.com/p/98061179>

代码实现

```
X = iris["data"][:, (2, 3)] # petal length, petal width
y = iris["target"]

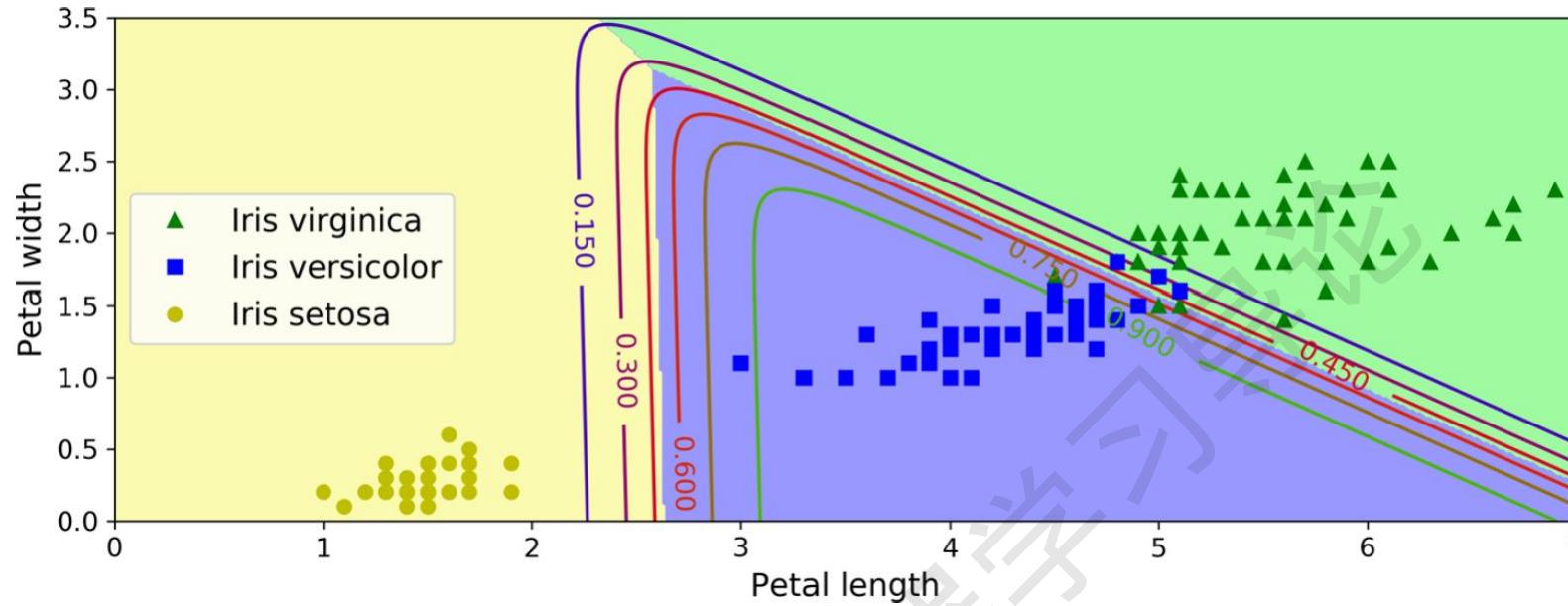
softmax_reg = LogisticRegression(multi_class="multinomial", solver="lbfgs", C=10)
softmax_reg.fit(X, y)
```

- ▶ 当用两个以上的类训练时，Scikit-Learn的LogisticRegression默认选择使用的一对多的训练方式，不过将超参数multi_class设置为"multinomial"，可以将其切换成Softmax回归。
- ▶ 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，可以通过超参数C进行控制

预测

- ▶ 花瓣长5cm宽2cm的鸢尾花预测种类：

```
>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]])
array([[6.38014896e-07, 5.74929995e-02, 9.42506362e-01]])
```



- ▶ 图展现了由不同背景色表示的决策边界。
- ▶ 任何两个类之间的决策边界都是线性的。图中的折线表示属于变色
鸢尾的概率（例如，标记为0.45的线代表45%的概率边界）。
- ▶ 该模型预测出的类，其估算概率有可能低于50%，比如，在所有决
策边界相交的地方，所有类的估算概率都为33%