线性模型

Linear model and SVM

线性模型

能很好地理解系统如何工作是非常有帮助的。针对你的任务，它有助于快速定位到合适的模型、正确的训练算法，以及一套适当的超参数。不仅如此，后期还能让你更高效地执行错误调试和错误分析。本节我们讨论最简单的模型 — 线性回归模型和支持向量机模型。

线性回归

- 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，如公式4-1所示。

*Equation 4-1.*线性回归模型预测

$$\hat{y} = \vartheta_0 + \vartheta_1 x_1 + \vartheta_2 x_2 + \cdots + \vartheta_n x_n$$

- $\hat{y}$ 是预测值
- n是特征的数量
- x_i是第i个特征值
- ϑ_j是第j个模型参数（包括偏置项θ_0及特征权重θ_1，θ_2，…，θ_n）

线性回归

- 在训练集X上，使用公式4-3计算线性回归的MSE，h_θ为假设函数。

*Equation 4-3.*线性回归模型的MSE成本函数

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

标准方程

- 为了得到使成本函数最小的θ值，有一个封闭解（解析解）方法 — 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）。

*Equation 4-4.*标准方程

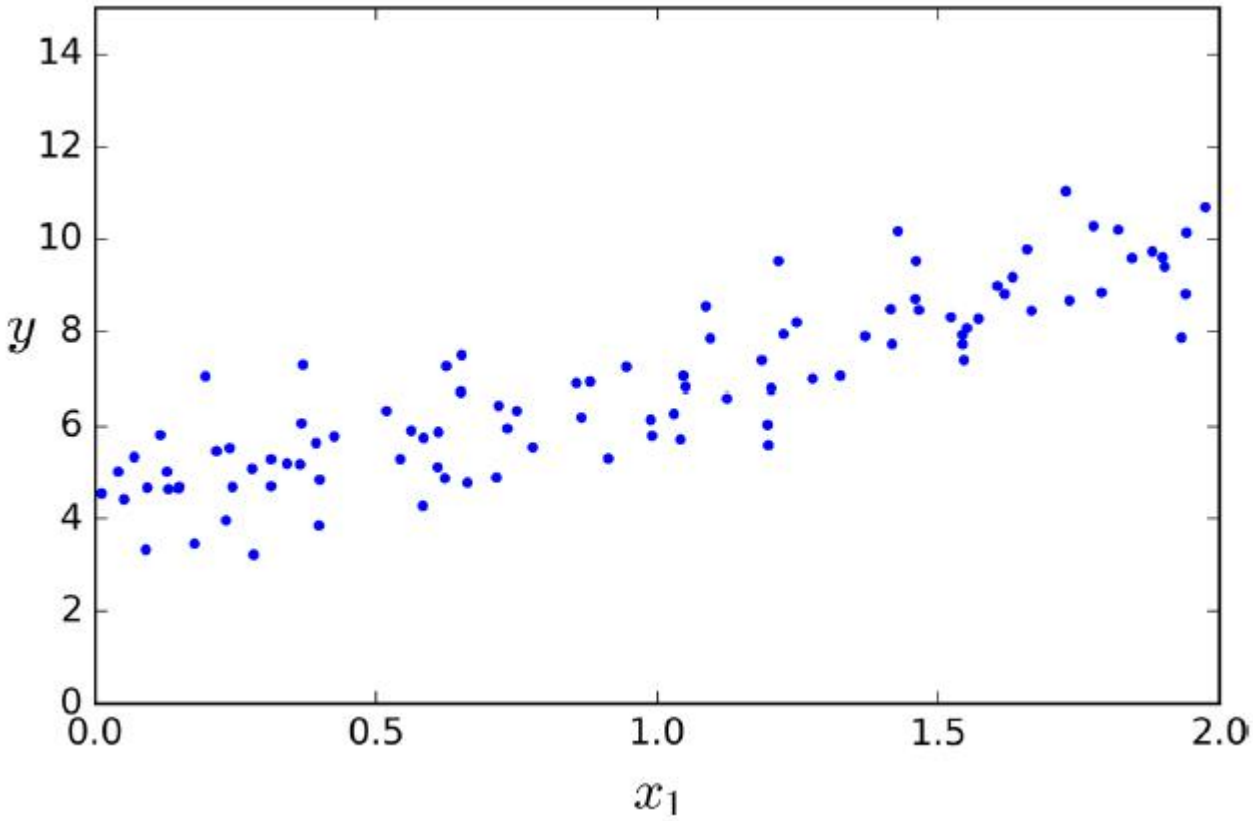
$$\hat{\theta} = \left(\mathbf{X}^T \cdot \mathbf{X}\right)^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

标准方程

import numpy as np

X = 2 * np.random.rand(100, 1)

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



标准方程

- 现在我们使用标准方程来计算。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)
```

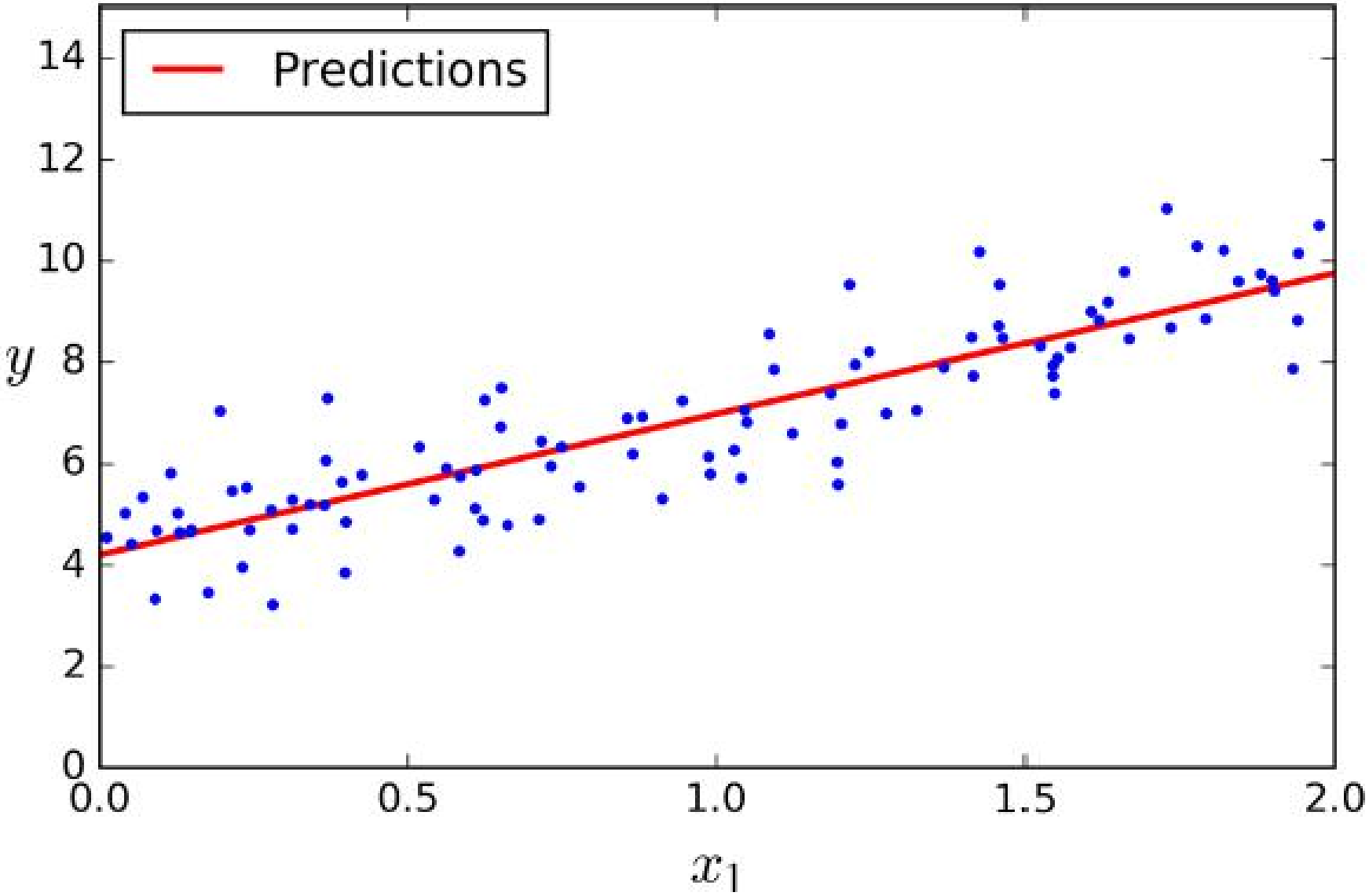
```
>>> theta_best

    array([[ 4.21509616],

           [ 2.77011339]])
```

标准方程

- Scikit-Learn的等效代码如下所示：

```
>>> 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]])
```

标准方程



计算复杂度

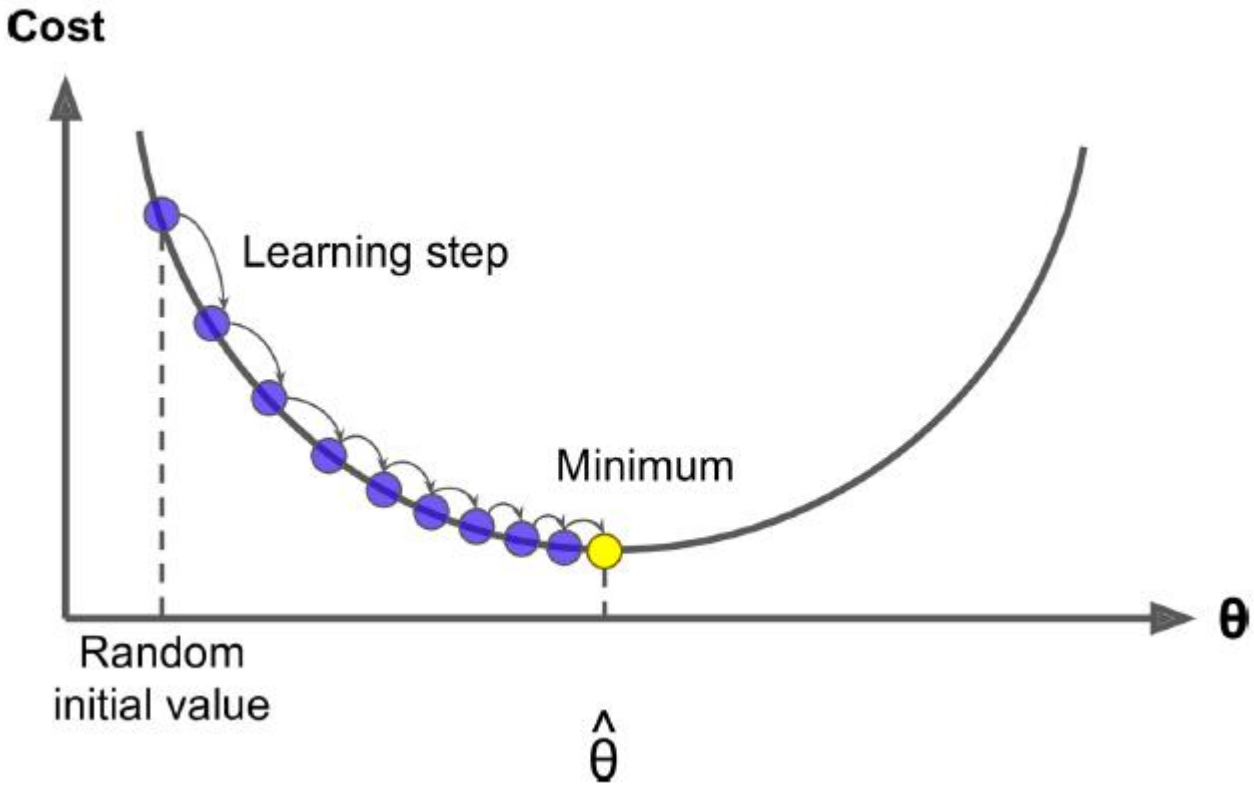
- 标准方程求逆的矩阵$X^T \cdot X$，是一个n×n矩阵（n是特征数量）。对这种矩阵求逆的计算复杂度通常为O（$n^{2.4}$）到O（n^3）之间（取决于计算实现）。换句话说，如果将特征数量翻倍，那么计算时间将乘以大约$2^{2.4}$＝5.3倍到2^3＝8倍之间。特征数量比较大（例如100000）时，标准方程的计算将极其缓慢。
- 好的一面是，相对于训练集中的实例数量（O（m））来说，方程是线性的，所以能够有效地处理大量的训练集，只要内存足够。

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。

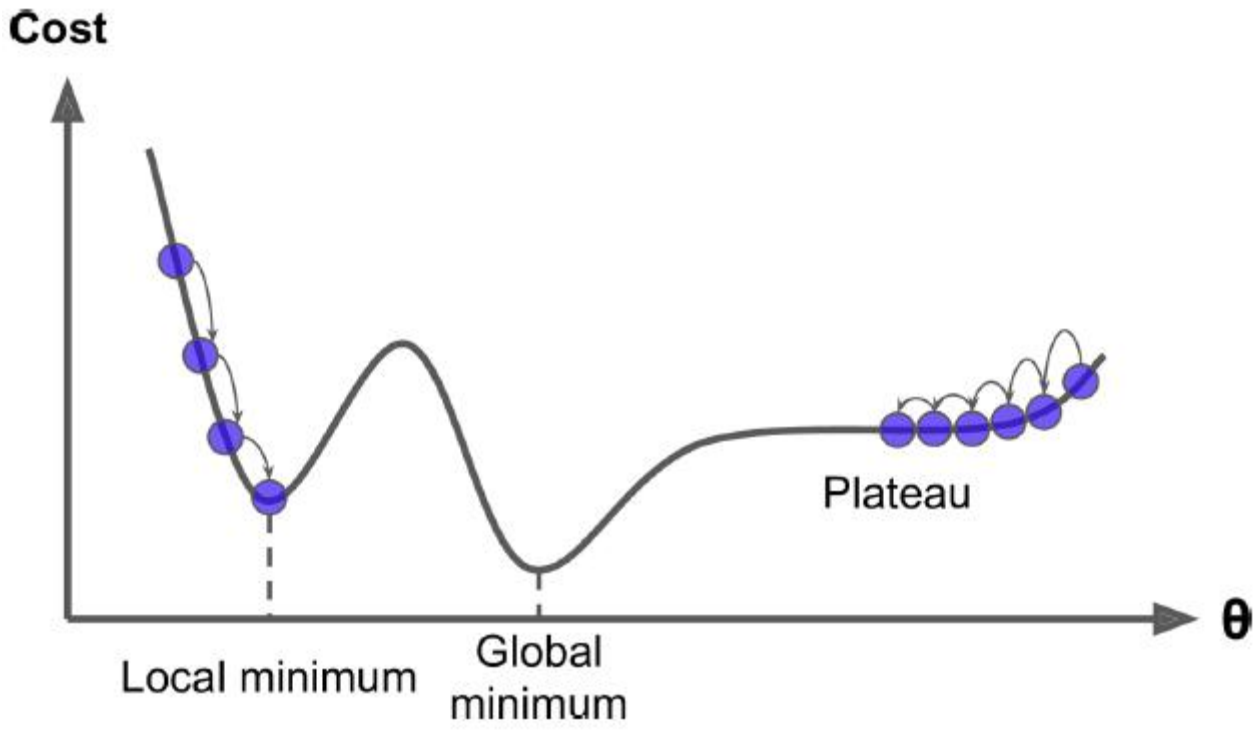
梯度下降

- 具体来说，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（参见图4-3）。



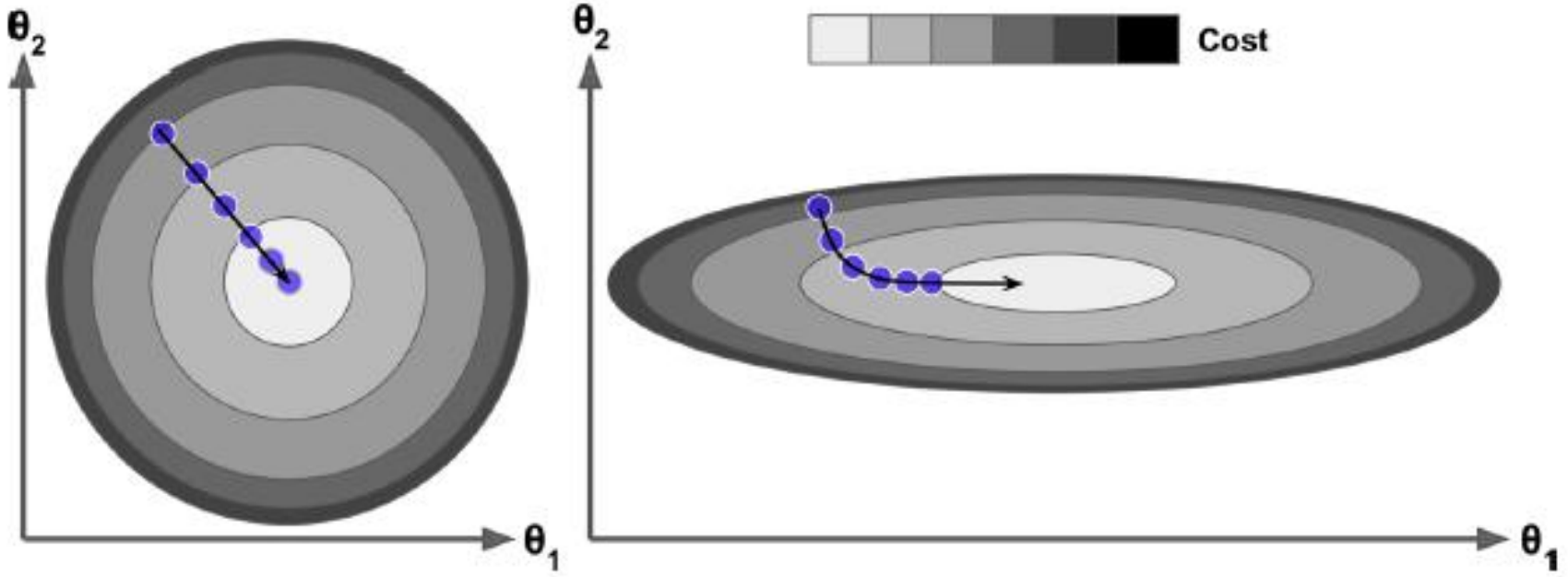
梯度下降

- 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、像山脉、像高原或者是各种不规则的地形，导致很难收敛到最小值。



梯度下降

- 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。



批量梯度下降

- 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。梯度向量，记作$\nabla_{\vartheta}\mathrm{MSE}(\vartheta)$，包含所有成本函数（每个模型参数一个）的偏导数。

Equation 4-6. Gradient vector of the cost function

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

批量梯度下降

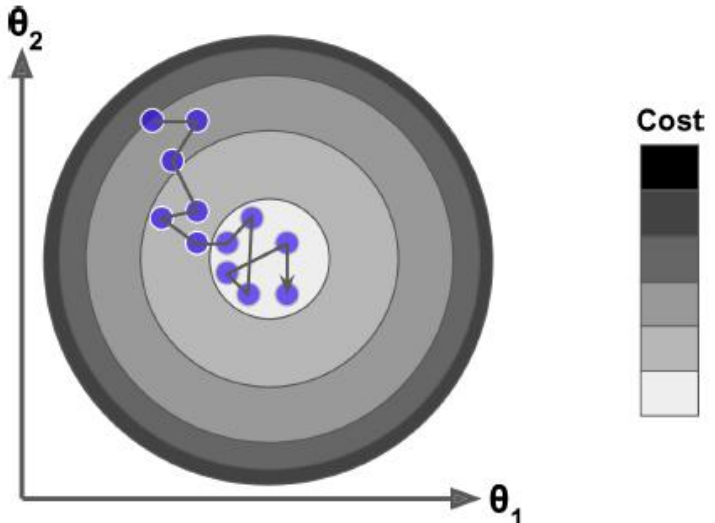
- 请注意，公式4-6在计算梯度下降的每一步时，都是基于完整的训练集X的。这就是为什么该算法会被称为批量梯度下降：每一步都使用整批训练数据。因此，面对非常庞大的训练集时，算法会变得极慢（不过我们即将看到快得多的梯度下降算法）。但是，梯度下降算法随特征数量扩展的表现比较好：如果要训练的线性模型拥有几十万个特征，使用梯度下降比标准方程要快得多。

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（SGD可以作为核外算法实现）。

随机梯度下降

- 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。



随机梯度下降

- 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。

随机梯度下降

- 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。

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(n_iter=50, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

- 再次得到一个跟标准方程的解非常相近的解决方案：

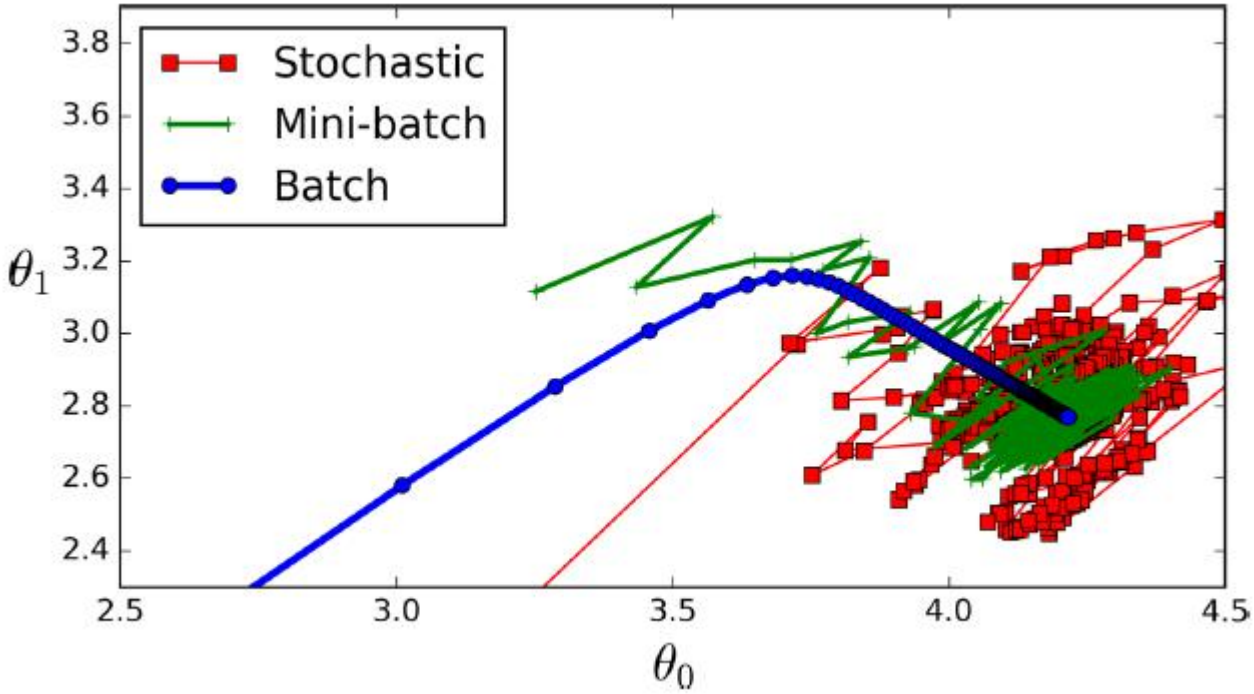
```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([ 4.18380366]), array([ 2.74205299]))
```

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。

小批量梯度下降

- 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。



小批量梯度下降

*Table 4-1.*线性回归算法比较

Algorithm	Large *m*	Out-of-core support	Large *n*	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	n/a
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	n/a

多项式回归

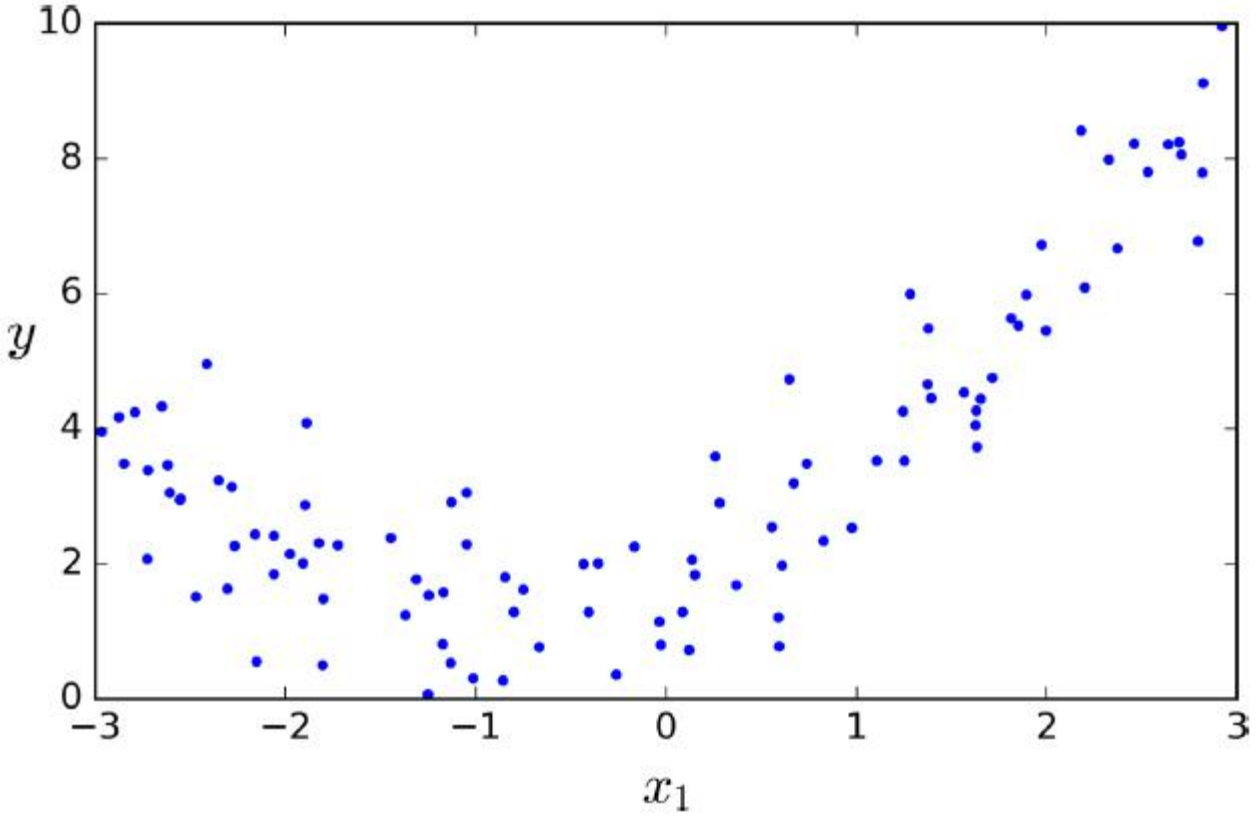
- 如果数据比简单的直线更为复杂，该怎么办？令人意想不到的是，其实你也可以用线性模型来拟合非线性数据。一个简单的方法就是将每个特征的幂次方添加为一个新特征，然后在这个拓展过的特征集上训练线性模型。这种方法被称为多项式回归。

多项式回归

m = 100

X = 6 * np.random.rand(m, 1) - 3

y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)



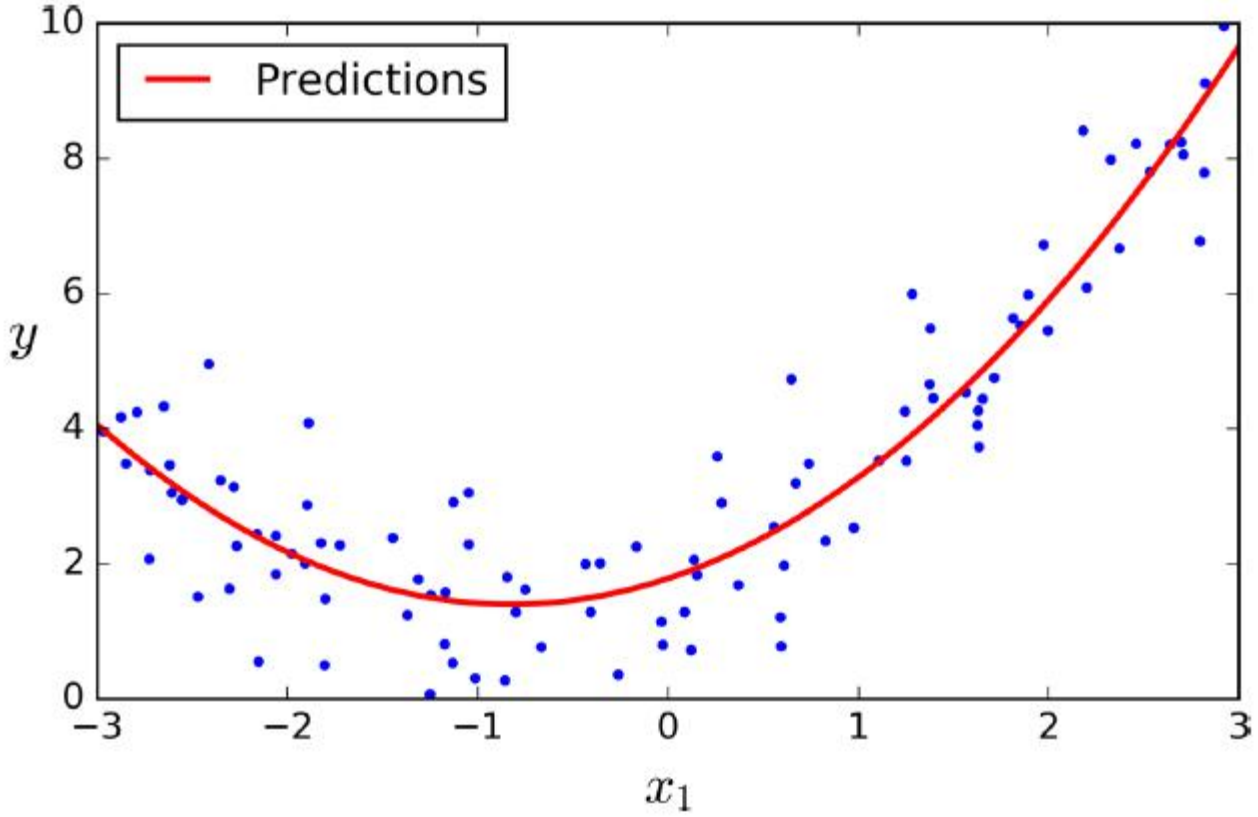
多项式回归

- 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）：

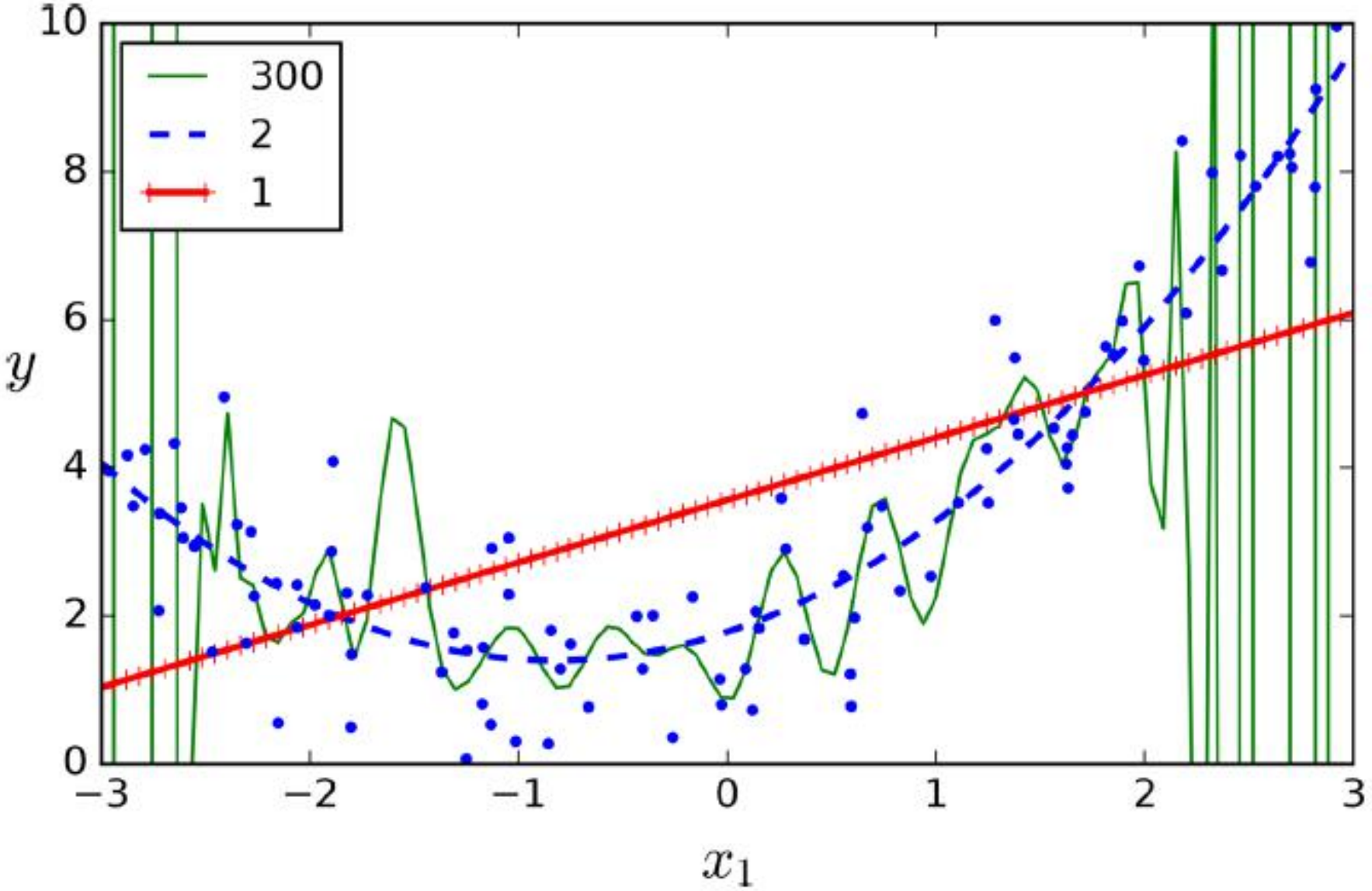
```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929, 0.56664654])
```

多项式回归

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



学习曲线

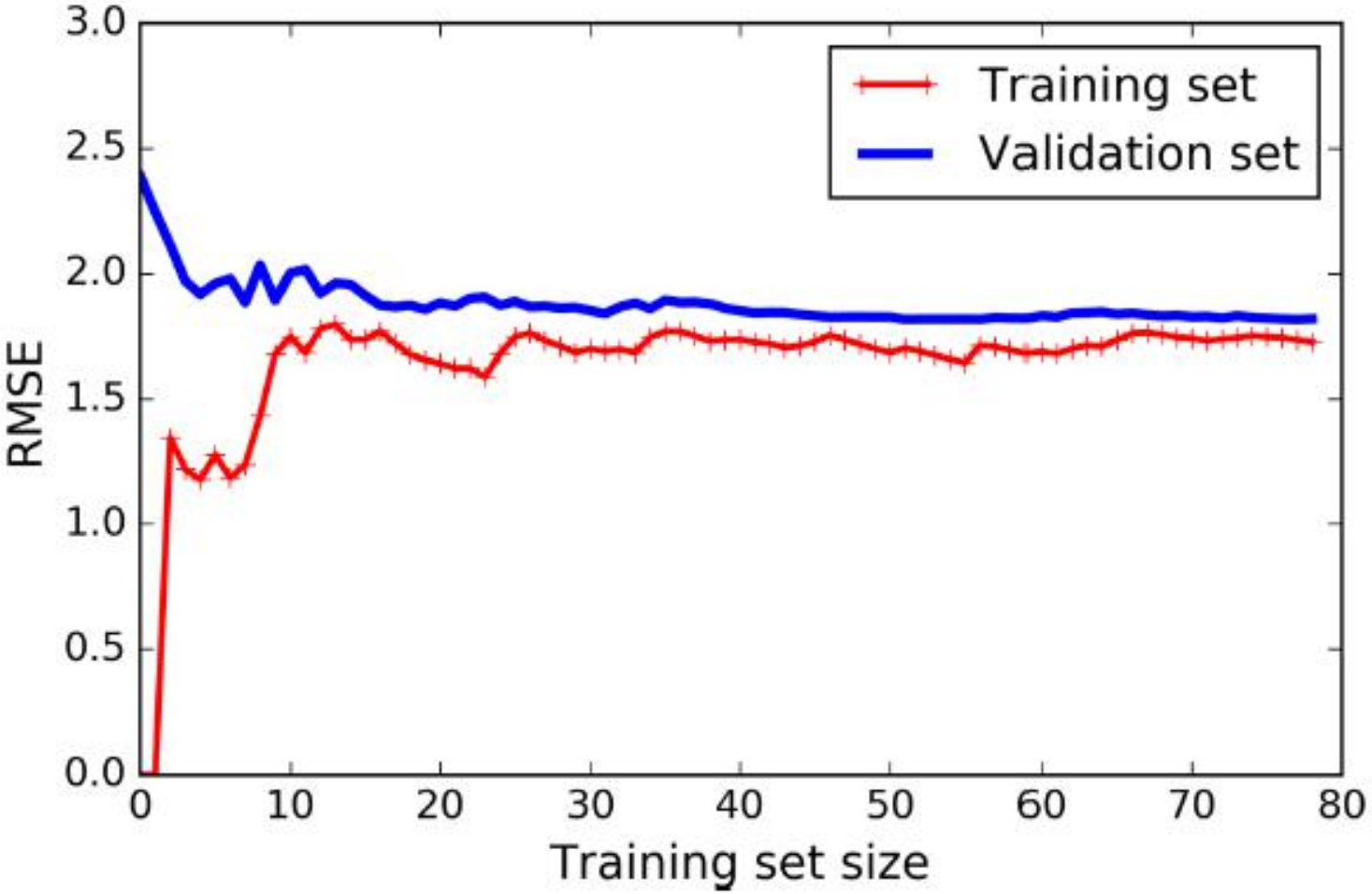


学习曲线

- 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，那么模型就是过拟合。如果在二者上的表现都不佳，那就是欠拟合。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。

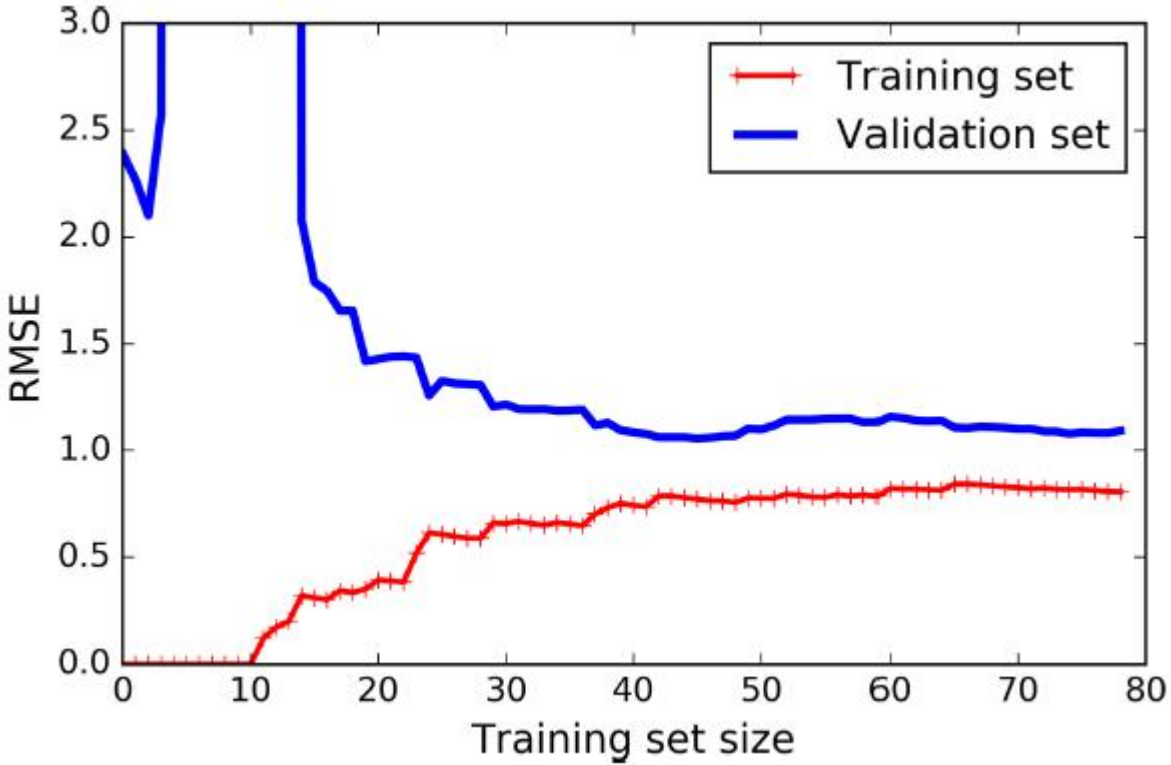
学习曲线

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



学习曲线

```
from sklearn.pipeline import Pipeline
polynomial_regression = Pipeline((
    ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
    ("sgd_reg", LinearRegression()), ))
plot_learning_curves(polynomial_regression, X, y)
```



偏差/方差权衡

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

- 偏差 ***Bias:***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。高偏差模型最有可能对训练数据拟合不足。
- 方差 ***Variance:***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。
- 不可避免的误差 ***Irreducible error:***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（例如修复数据源，如损坏的传感器，或者是检测并移除异常值）。

正则线性模型

- 减少过度拟合的一个好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。接下来我们将会使用岭回归（Ridge Regression）、套索回归（Lasso Regression）及弹性网络（Elastic Net）这三种不同的实现方法对权重进行约束。

岭回归

- 岭回归（也叫作吉洪诺夫正则化）是线性回归的正则化版：在成本函数中添加一个等于$\alpha\sum_{i=1}^{n}\theta_i^2$的正则项。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，你需要使用未经正则化的性能指标来评估模型性能。

岭回归

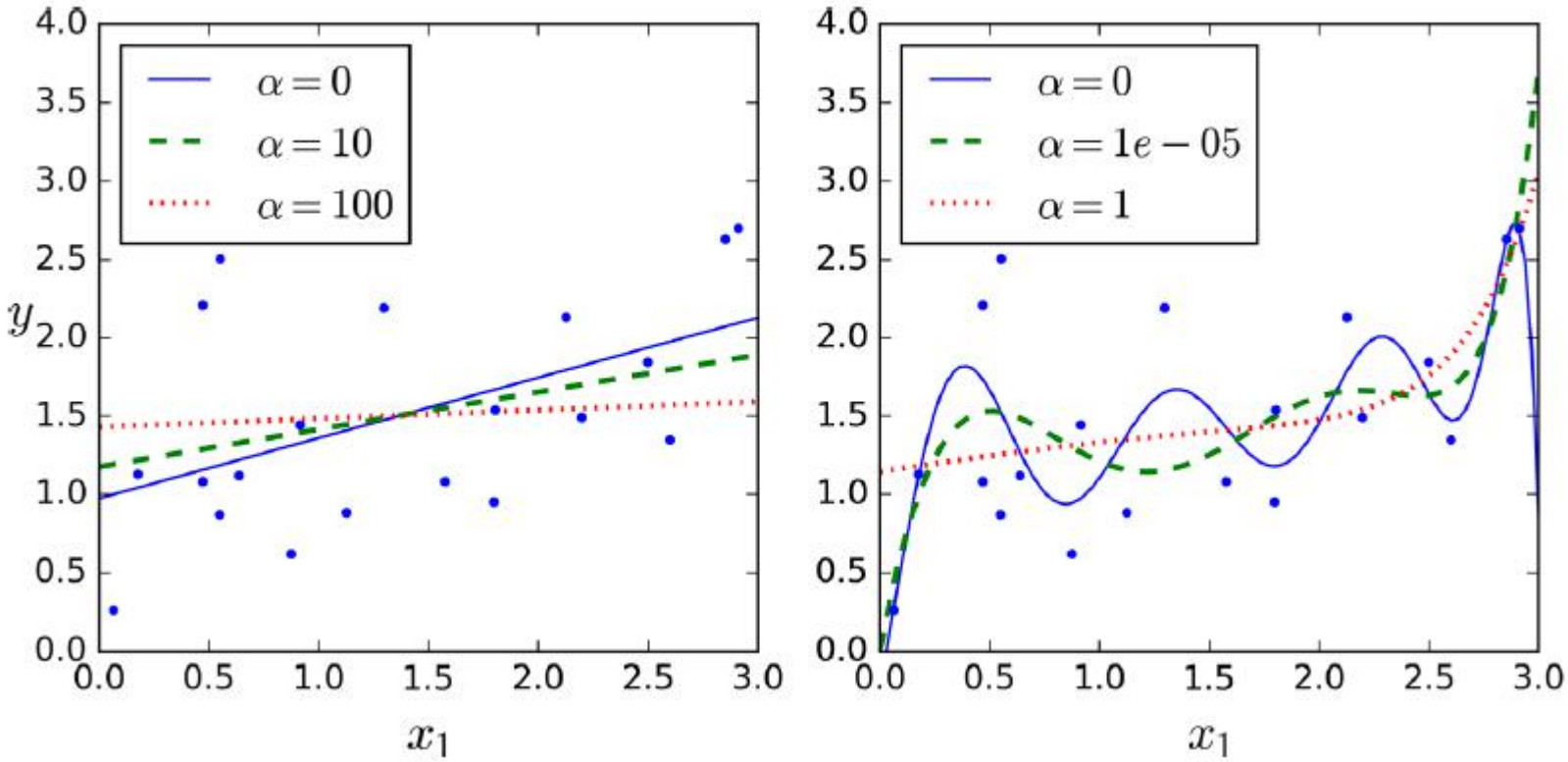
- 超参数α控制的是对模型进行正则化的程度。如果α=0，则岭回归就是线性模型。如果α非常大，那么所有的权重都将非常接近于零，结果是一条穿过数据平均值的水平线。公式4-8给出了岭回归模型的成本函数。

Equation 4-8. Ridge Regression cost function

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

岭回归

- 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。



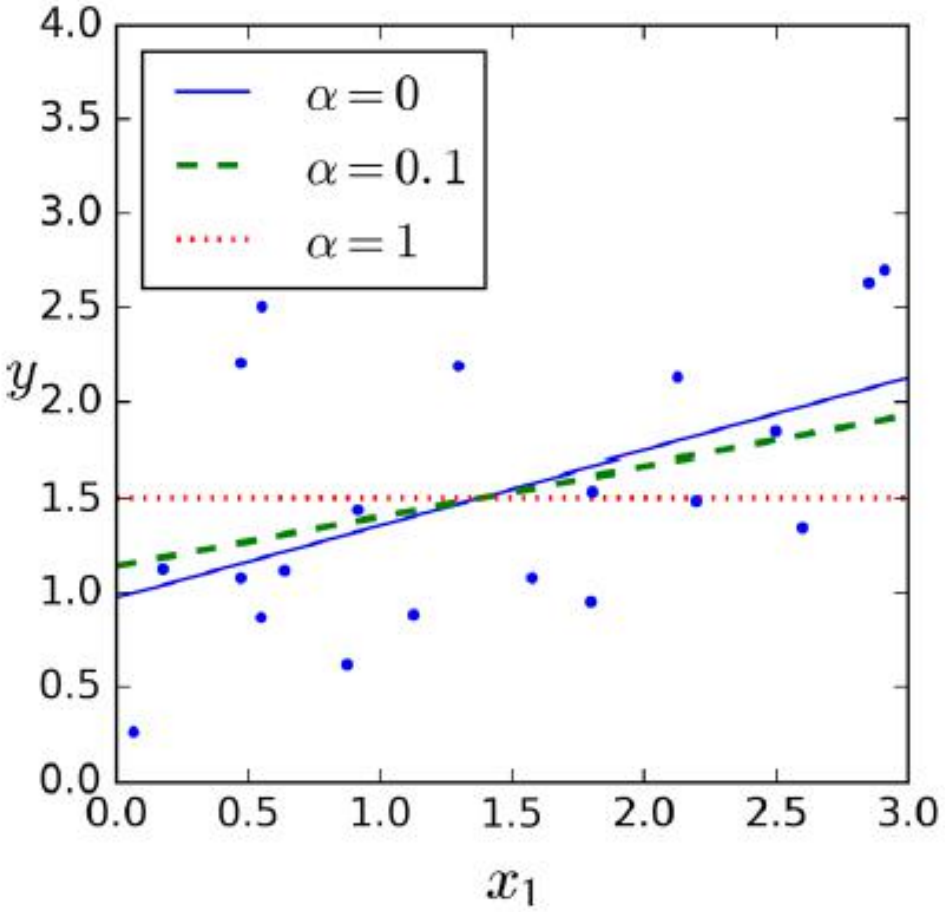
套索回归

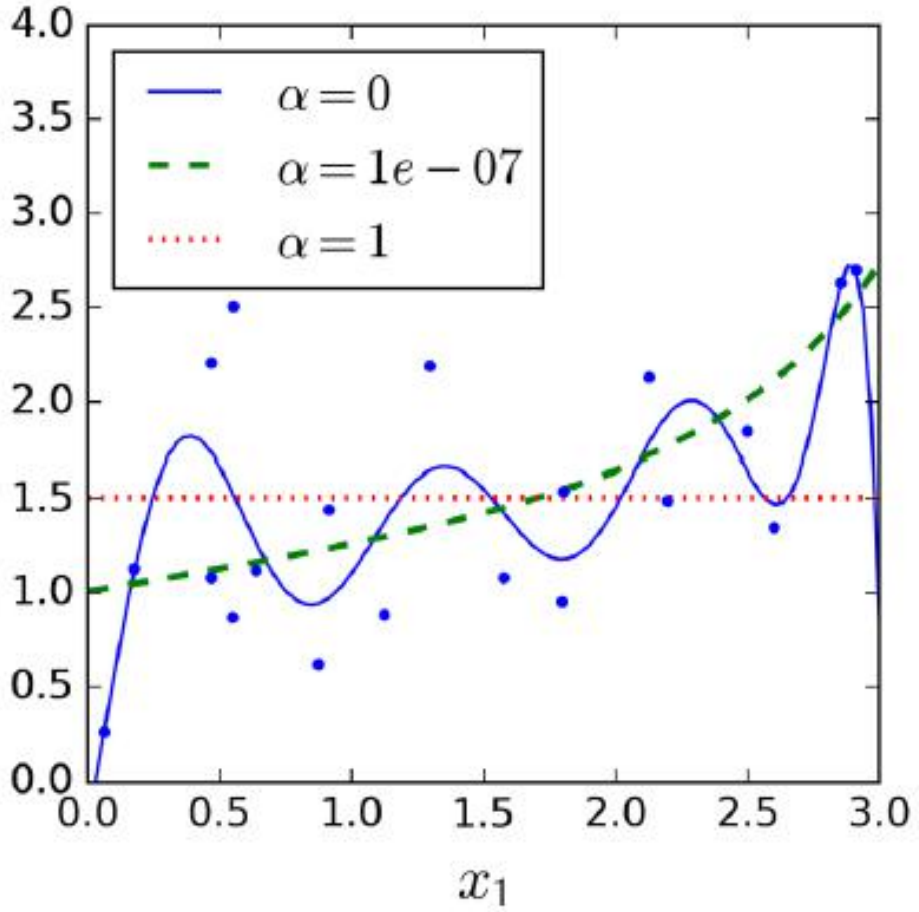
- 线性回归的另一种正则化，叫作最小绝对收缩和选择算子回归（Least Absolute Shrinkage and Selection Operator Regression，简称Lasso回归，或套索回归）。与岭回归一样，它也是向成本函数增加一个正则项，但是它增加的是权重向量的ℓ_1范数，而不是ℓ_2范数的平方的一半（参见公式4-10）。

Equation 4-10. Lasso Regression cost function

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

套索回归





套索回归

- 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）。例如，在图4-18的右图中的虚线（$\alpha = 10^{-7}$）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）。

弹性网络

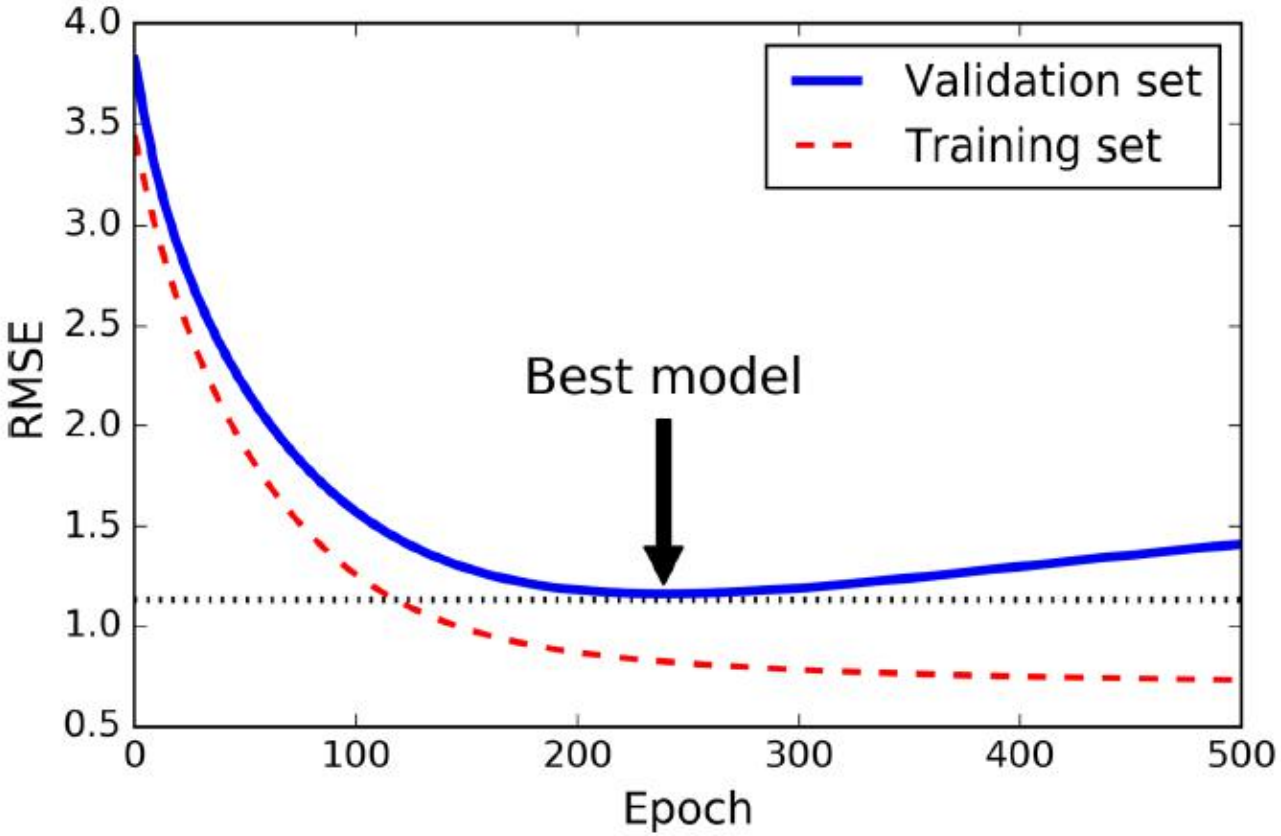
- 弹性网络是岭回归与Lasso回归之间的中间地带。其正则项就是岭回归和Lasso回归的正则项的混合，混合比例通过r来控制。当r＝0时，弹性网络即等同于岭回归，而当r＝1时，即相当于Lasso回归（见公式4-12）。

Equation 4-12. Elastic Net cost function

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

早期停止法

- 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。



逻辑回归

- 逻辑回归（Logistic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（也就是负类，标记为“0”）。这样它就成了一个二元分类器。

概率估算

- 所以它是怎么工作的呢？跟线性回归模型一样，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，它输出的是结果的数理逻辑（参见公式4-13）。

Equation 4-13. Logistic Regression model estimated probability (vectorized form)

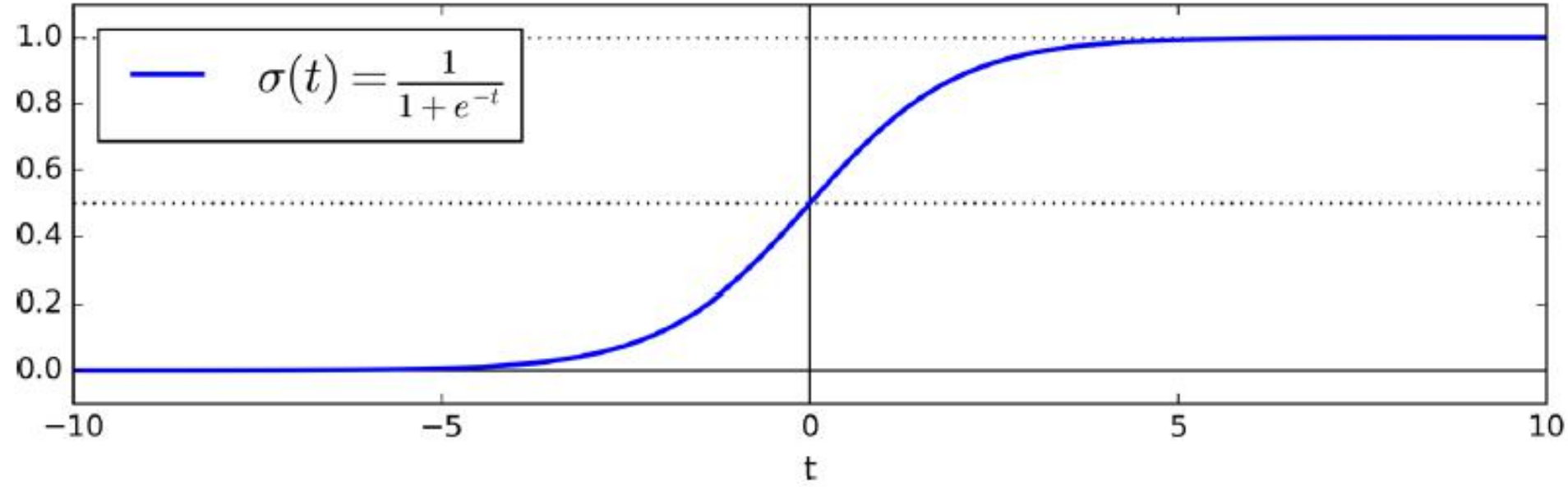
$$\hat{p} = h_\theta(\mathbf{x}) = \sigma\left(\theta^T \cdot \mathbf{x}\right)$$

- 逻辑模型（也称为*logit*），是一个sigmoid函数（即S形），记作σ（·），它的输出为一个0到1之间的数字。

概率估算

Equation 4-14. Logistic function

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



概率估算

- 一旦逻辑回归模型估算出实例x属于正类的概率 $p = h_\theta(x)$，就可以轻松做出预测（见公式4-15）。

Equation 4-15. Logistic Regression model prediction

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

- 注意，当t<0时，σ（t）<0.5；当t≥0时，σ（t）≥0.5。所以如果$\theta^T \cdot x$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

- 现在你知道逻辑回归模型是如何估算概率并做出预测了。但是要怎么训练呢？训练的目的就是设置参数向量θ，使模型对正类实例做出高概率估算（y=1），对负类实例做出低概率估算（y=0）。公式4-16所示为单个训练实例x的成本函数，正说明了这一点。

Equation 4-16. Cost function of a single training instance

$$c(\theta) = \begin{cases} -\log\left(\hat{p}\right) & \text{if } y = 1, \\ -\log\left(1 - \hat{p}\right) & \text{if } y = 0. \end{cases}$$

训练和成本函数

- 这个成本函数是有道理的，因为当t接近于0时，-log（t）会变得非常大，所以如果模型估算一个正类实例的概率接近于0，成本将会变得很高。同理估算出一个负类实例的概率接近1，成本也会变得非常高。那么反过来，当t接近于1的时候，-log（t）接近于0，所以对一个负类实例估算出的概率接近于0，对一个正类实例估算出的概率接近于1，而成本则都接近于0，这刚好是我们想要的。

Equation 4-16. Cost function of a single training instance

$$c(\theta) = \begin{cases} -\log\left(\hat{p}\right) & \text{if } y = 1, \\ -\log\left(1 - \hat{p}\right) & \text{if } y = 0. \end{cases}$$

训练和成本函数

- 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。

Equation 4-17. Logistic Regression cost function (log loss)

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

训练和成本函数

- 坏消息是，这个函数没有已知的解析解（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θ_j的偏导数方程。

Equation 4-18. Logistic cost function partial derivatives

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

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，或者叫多元逻辑回归。

Softmax回归

- 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），估算出每个类别的概率。你应该很熟悉计算s_k（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。

Equation 4-19. Softmax score for class k

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

Softmax回归

- 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 p_k，也就是实例属于类别k的概率。

Equation 4-20. Softmax function

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

Softmax回归

- 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。

Equation 4-21. Softmax Regression classifier prediction

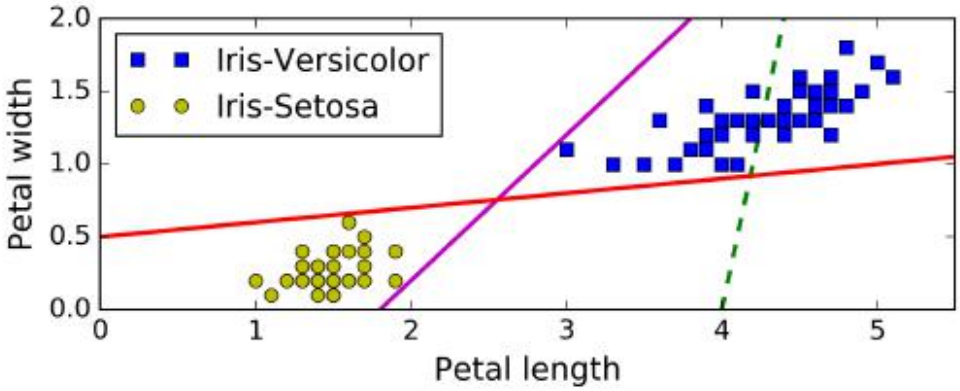
$$\hat{y} = \underset{k}{\operatorname{argmax}}\ \sigma(\mathbf{s}(\mathbf{x}))_k = \underset{k}{\operatorname{argmax}}\ s_k(\mathbf{x}) = \underset{k}{\operatorname{argmax}}\ \left(\theta_k{}^T \cdot \mathbf{x}\right)$$

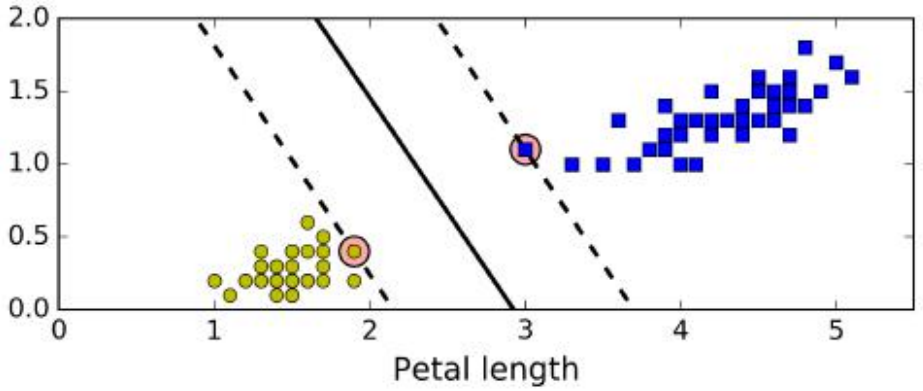
支持向量机 Support Vector Machines

- 支持向量机（SVM）是个非常强大并且有多种功能的机器学习模型，能够做线性或者非线性的分类，回归，甚至异常值检测。机器学习领域中最为流行的模型之一，是任何学习机器学习的人必备的工具。SVM 特别适合应用于复杂但中小规模数据集的分类问题。

线性支持向量机分类

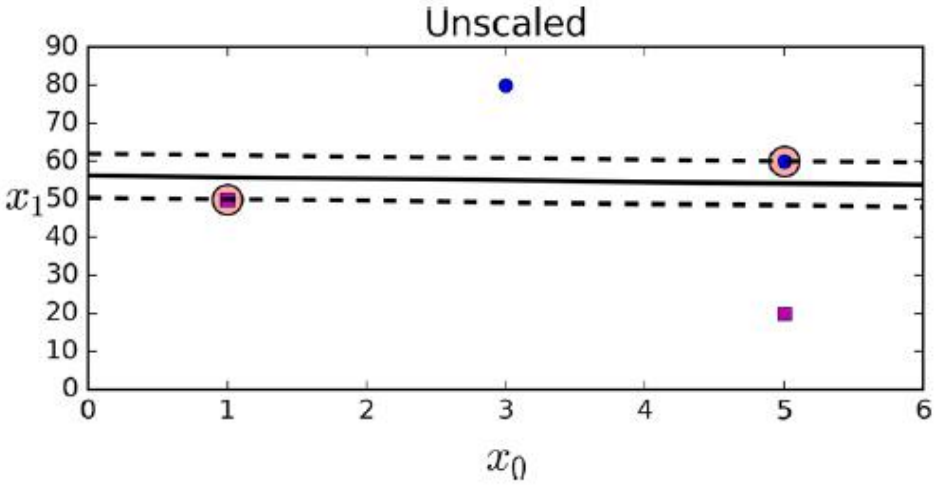
- 下图的两个种类能够非常容易的用一条直线分开（即线性可分的）。左边的图显示了三种可能的线性分类器的判定边界。其中用虚线表示的线性模型不能正确地划分类别。另外两个线性模型划分正确，但是判定边界很靠近样本点，在新数据上不会表现的很好。右边图中 SVM 分类器的判定边界实线，不仅分开了两种类别，而且还尽可能地远离了最靠近的训练数据点。即在两种类别之间保持了最大间隔分类。

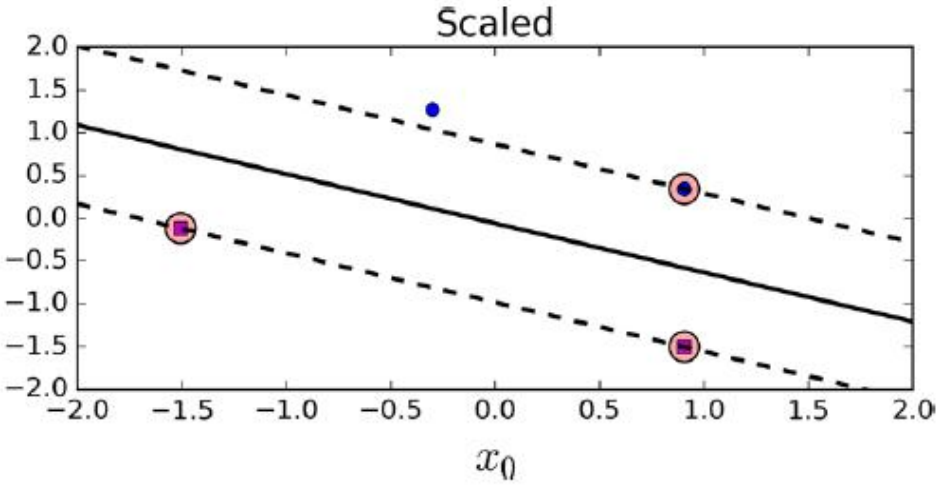




线性支持向量机分类

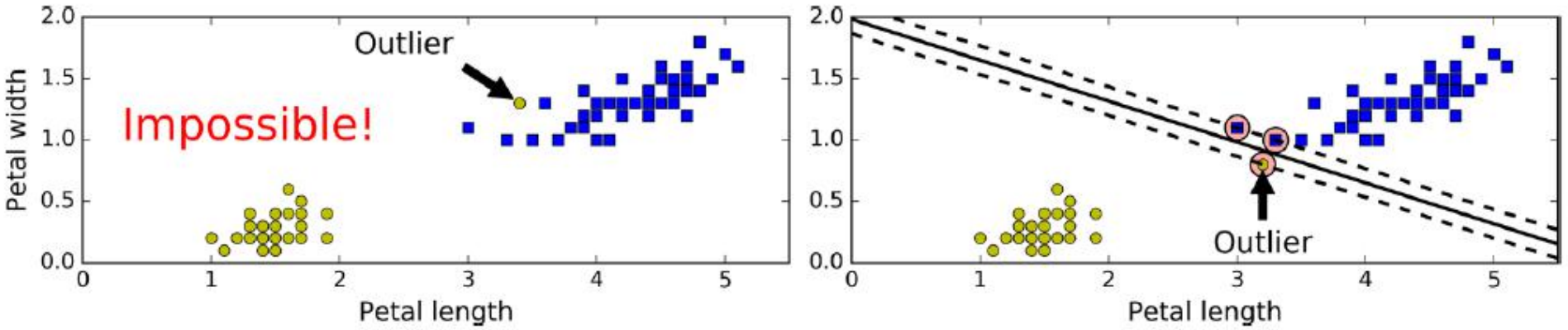
- SVM 对特征缩放比较敏感，可以看到：左边的图中，垂直的比例要更大于水平的比例，所以最宽的“街道”接近水平。但对特征缩放后（例如使用Scikit-Learn的StandardScaler），判定边界看起来要好得多，如右图。





软间隔分类

- 如果我们严格地规定所有的数据都不在“街道”上，都在正确地两边，称为硬间隔分类，硬间隔分类有两个问题，第一，只对线性可分的数据起作用，第二，对异常点敏感。下图显示了有一个异常点的鸢尾花数据集：左边的图中很难找到硬间隔，右边的图中判定边界和我们之前没有异常点的判定边界非常不一样，它很难泛化。

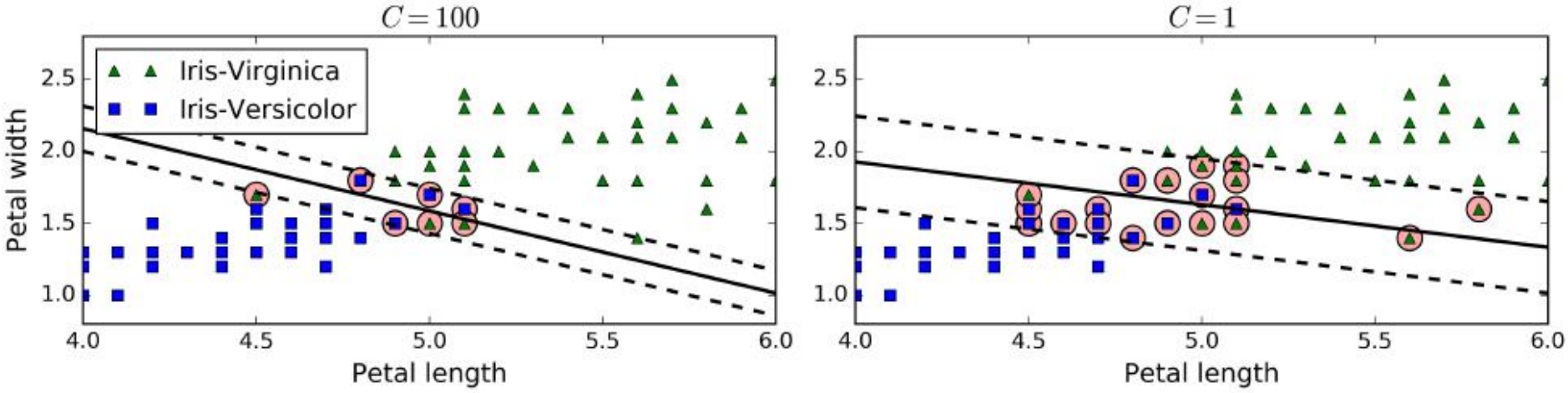


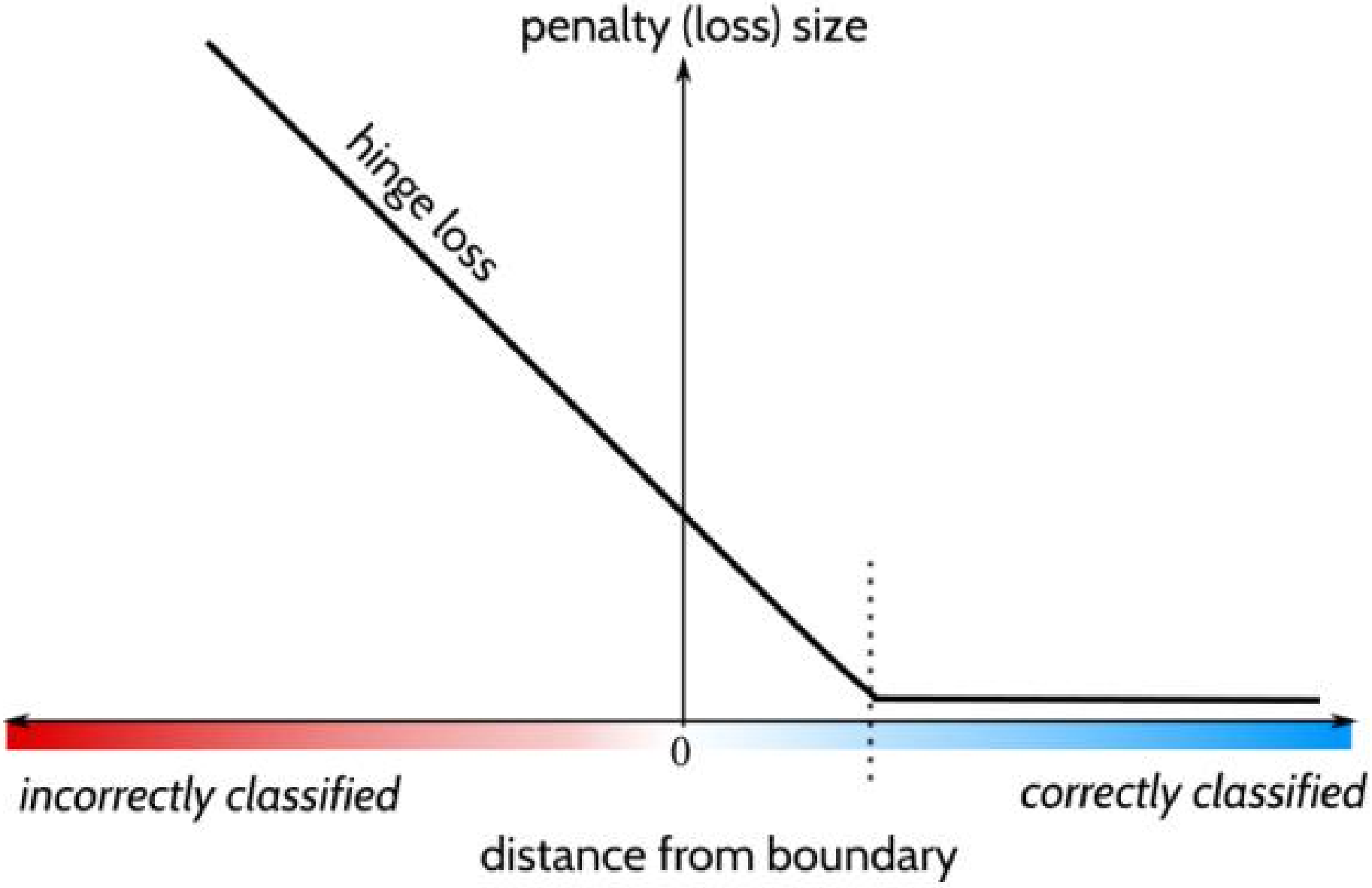
软间隔分类

- 为了避免上述的问题，我们更倾向于使用更加软性的模型。目的在保持“街道”尽可能宽敞和避免间隔违规（例如：数据点出现在“街道”中央或者甚至在错误的一边）之间找到一个良好的平衡。这就是软间隔分类。

软间隔分类

- 在 Scikit-Learn 中可以用C超参数（惩罚系数）来控制这种平衡：较小的C会导致更宽的“街道”，但允许更多的间隔违规：左边图中，使用了较大的C值，导致更少的间隔违规，但是间隔较小。右边的图，使用了较小的C值，间隔变大了，但是许多数据点出现在了“街道”上。然而，第二个分类器似乎泛化地更好：事实上，在这个训练数据集上减少了预测错误，因为实际上大部分的间隔违规点出现在了判定边界正确的一侧。

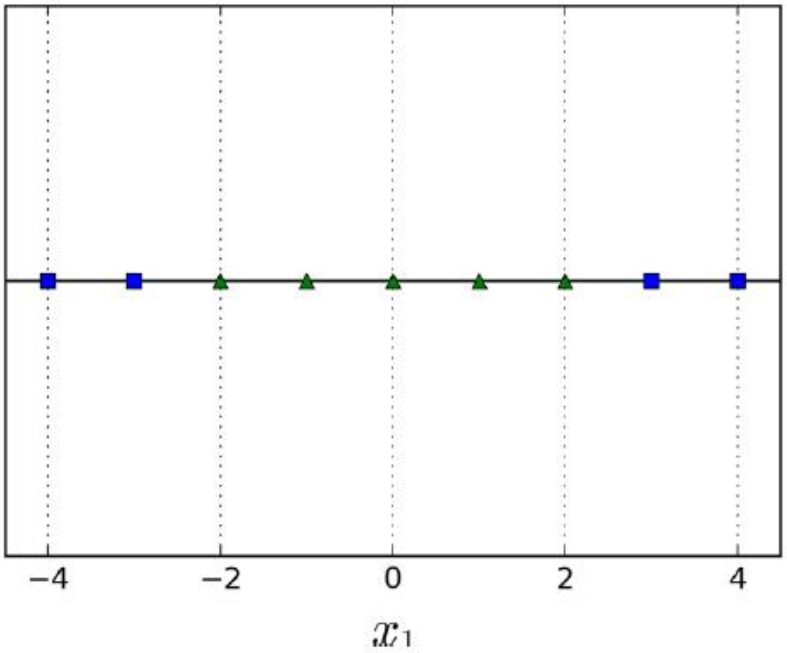


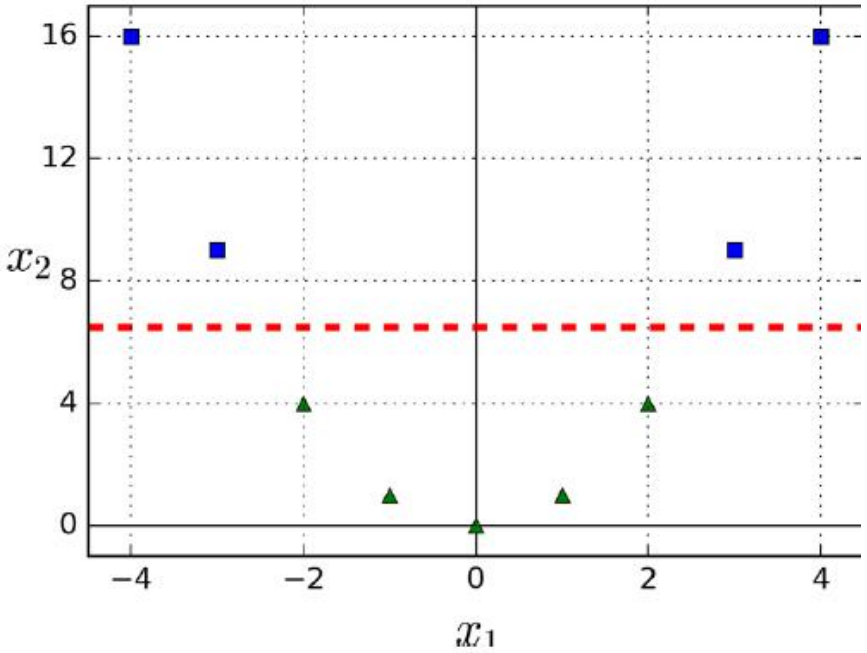

penalty (loss) size
hinge loss
0
incorrectly classified
correctly classified
distance from boundary

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC
iris = datasets.load_iris()
X = iris["data"][:, (2, 3)] # petal length, petal width
y = (iris["target"] == 2).astype(np.float64) # Iris-Virginica
svm_clf = Pipeline((
            ("scaler", StandardScaler()),
            ("linear_svc", LinearSVC(C=1, loss="hinge")),
        ))
svm_clf.fit(X_scaled, y)

>>> svm_clf.predict([[5.5, 1.7]])
array([ 1.])
```

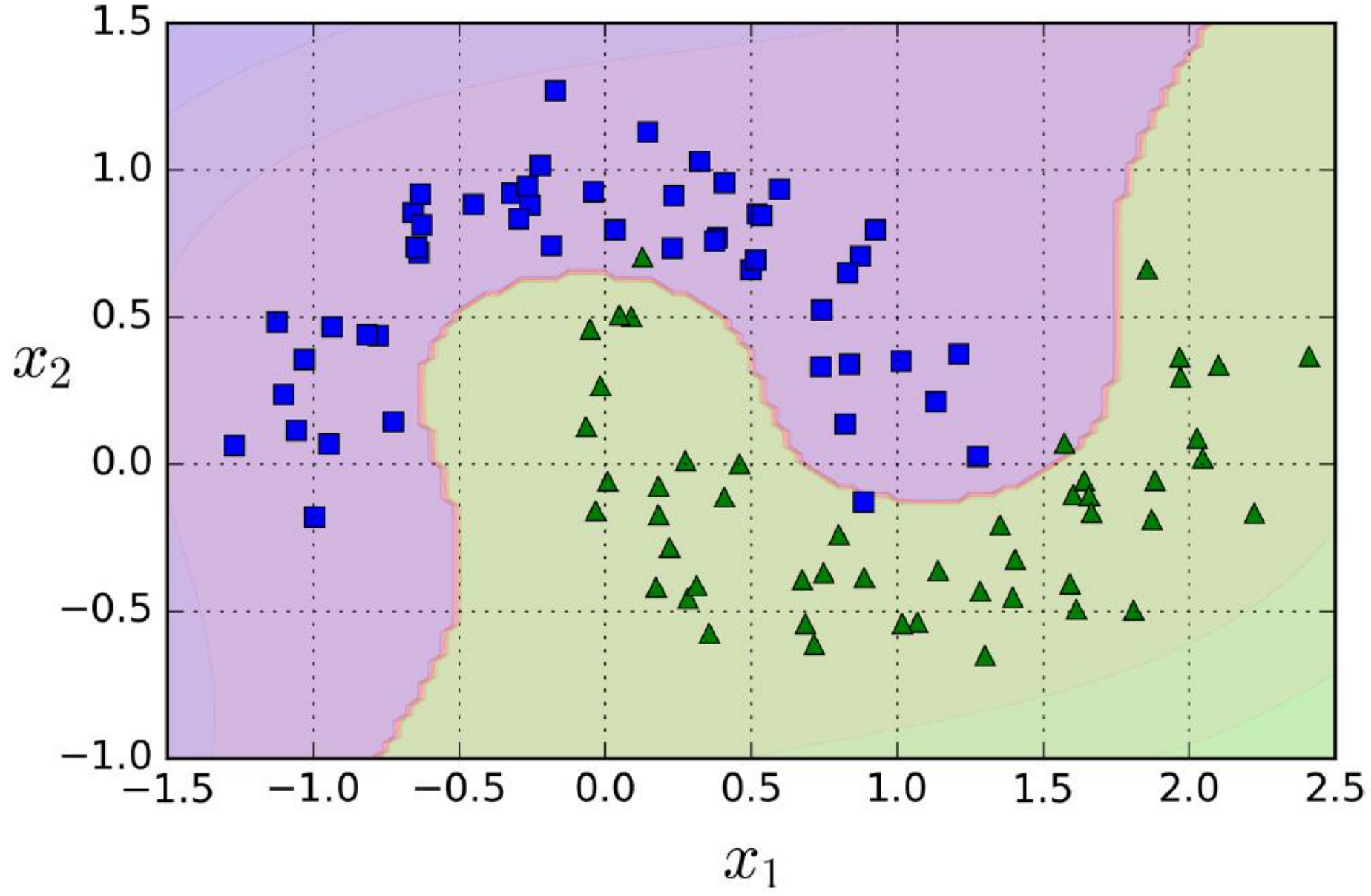
非线性支持向量机分类

- 尽管线性 SVM 分类器在许多案例上表现得出乎意料的好，但是很多数据集并不是线性可分的。一种处理非线性数据集方法是增加更多的特征，例如多项式特征；在某些情况下可以变成线性可分的数据。在左图中，它只有一个特征x1的简单的数据集，正如你看到的，该数据集不是线性可分的。但是如果增加了第二个特征 $x2=(x1)^2$，产生的 2D 数据集就线性可分。





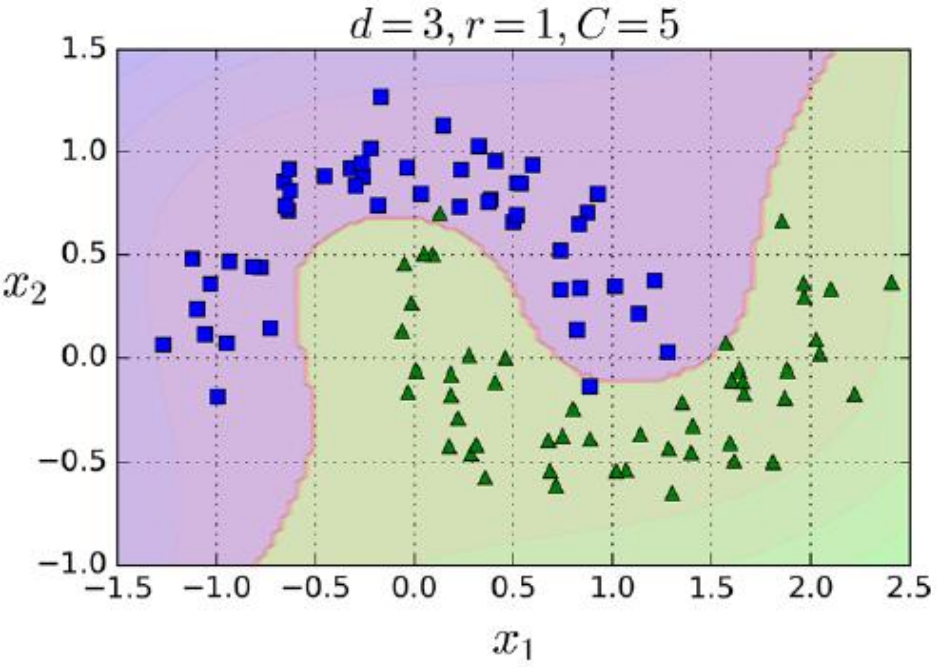
```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
polynomial_svm_clf = Pipeline((
    ("poly_features", PolynomialFeatures(degree=3)),
    ("scaler", StandardScaler()),
    ("svm_clf", LinearSVC(C=10, loss="hinge"))
  ))
polynomial_svm_clf.fit(X, y)
```

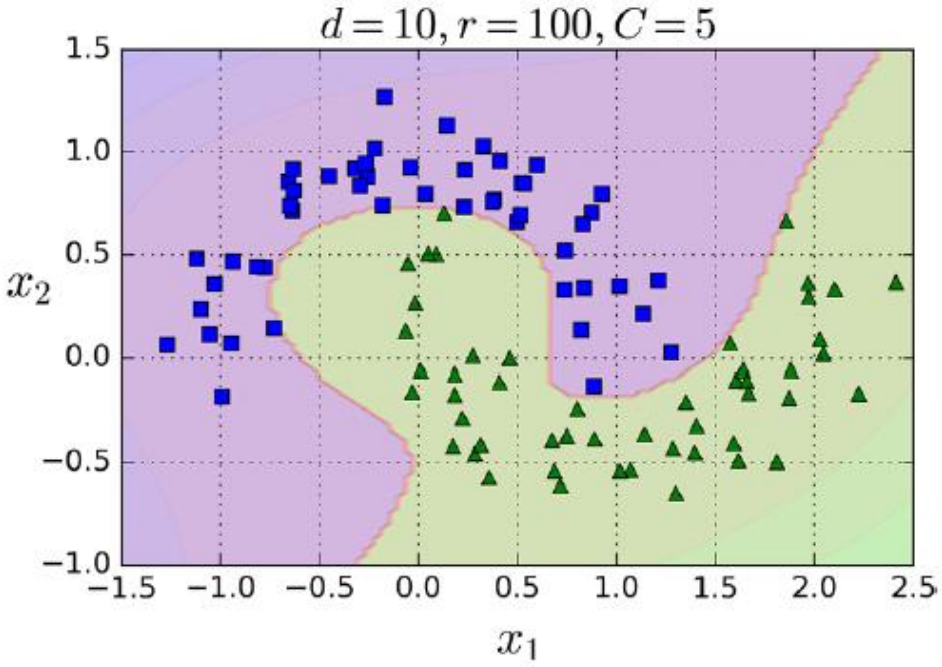

1.5
1.0
0.5
0.0
−0.5
−1.0
x_2
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
x_1

多项式核

- 添加多项式特征很容易实现，不仅仅在 SVM，在各种机器学习算法都有不错的表现，但是低次数的多项式不能处理非常复杂的数据集，而高次数的多项式却产生了大量的特征，会使模型变得慢。
- 幸运的是，当你使用 SVM 时，可以运用一个被称为“核技巧”（kernel trick）的神奇数学技巧。它可以取得就像你添加了许多多项式，甚至有高次数的多项式，一样好的结果。所以不会大量特征导致的组合爆炸，因为你并没有增加任何特征。这个技巧可以用 SVC 类来实现。让我们在卫星数据集测试一下效果。

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline((
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
   ))
poly_kernel_svm_clf.fit(X, y)
```





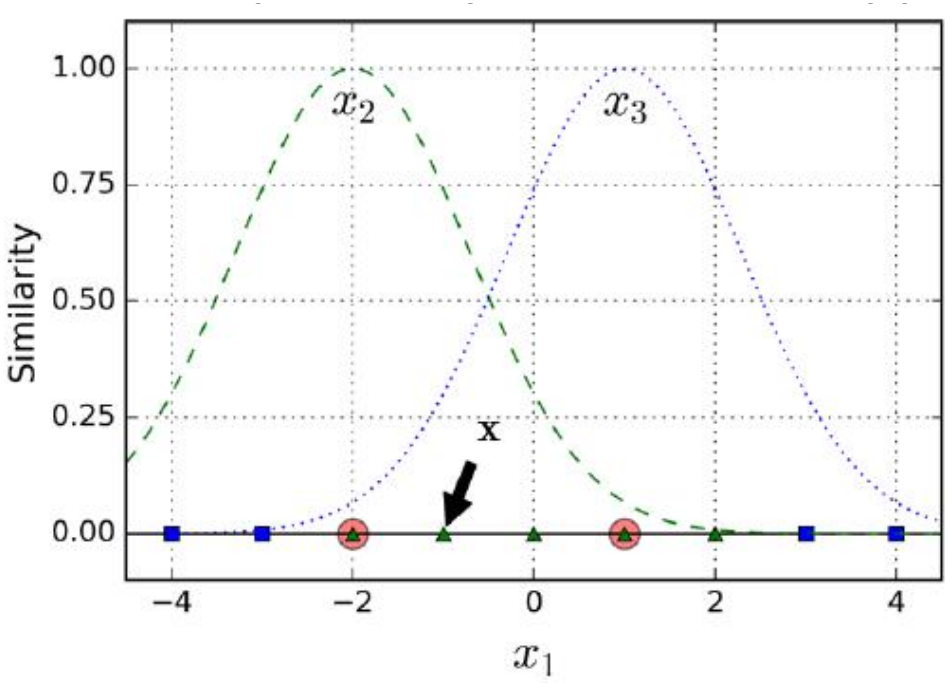
增加相似特征

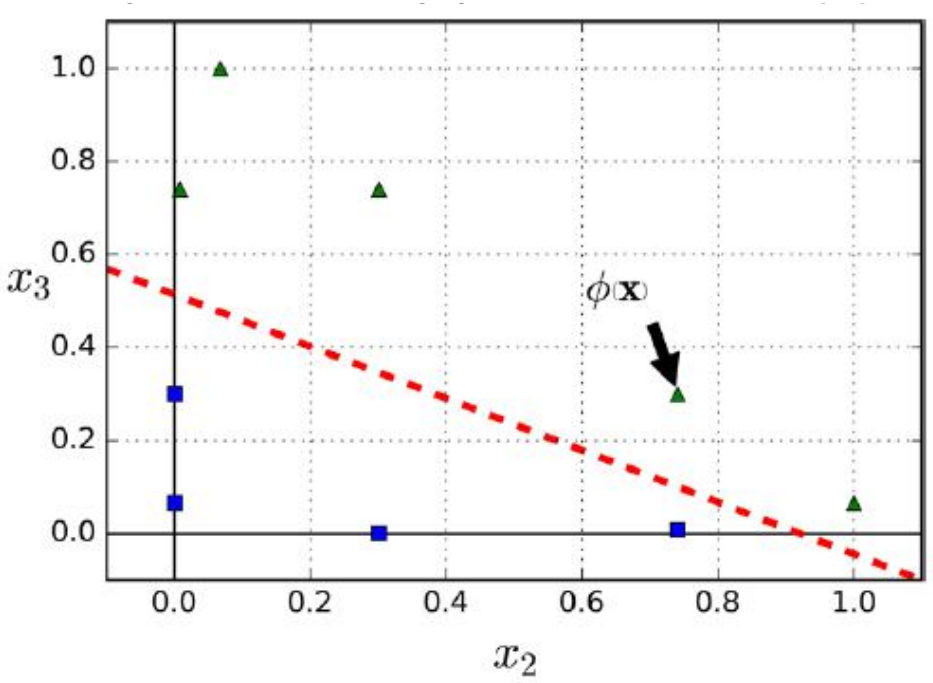
- 另一种解决非线性问题的方法是使用相似函数（similarity funtion）计算每个样本与特定地标（landmark）的相似度。例如，让我们来看看前面讨论过的一维数据集，并在x1=-2和x1=1之间增加两个地标。接下来，我们定义一个相似函数，即高斯径向基函数（Gaussian Radial Basis Function，RBF），设置γ = 0.3

Equation 5-1. Gaussian RBF

$$\phi\gamma(\mathbf{x}, \ell) = \exp\left(-\gamma \| \mathbf{x} - \ell \|^2\right)$$

增加相似特征





Equation 5-1. Gaussian RBF

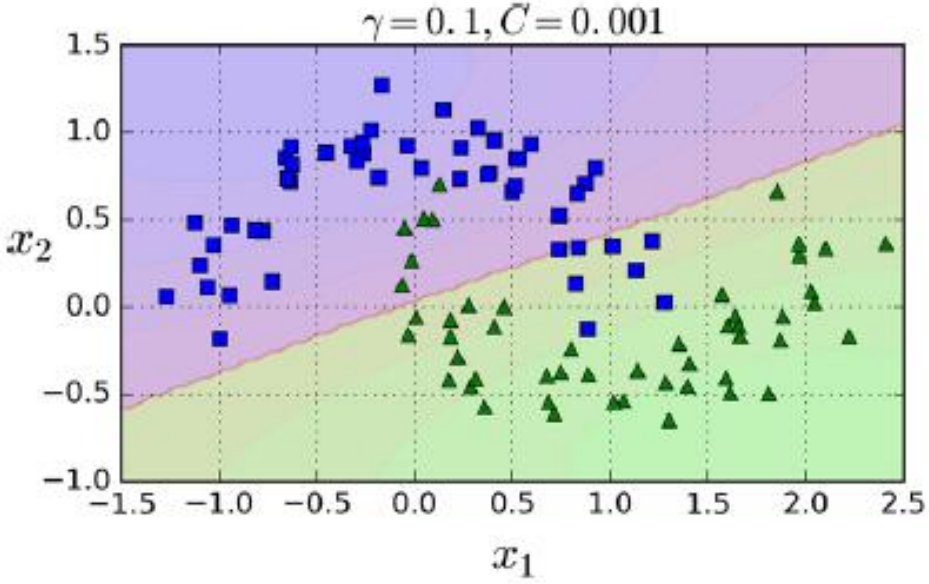
$$\phi y(\mathbf{x}, \ell) = \exp\left(-y\| \mathbf{x} - \ell \|^2\right)$$

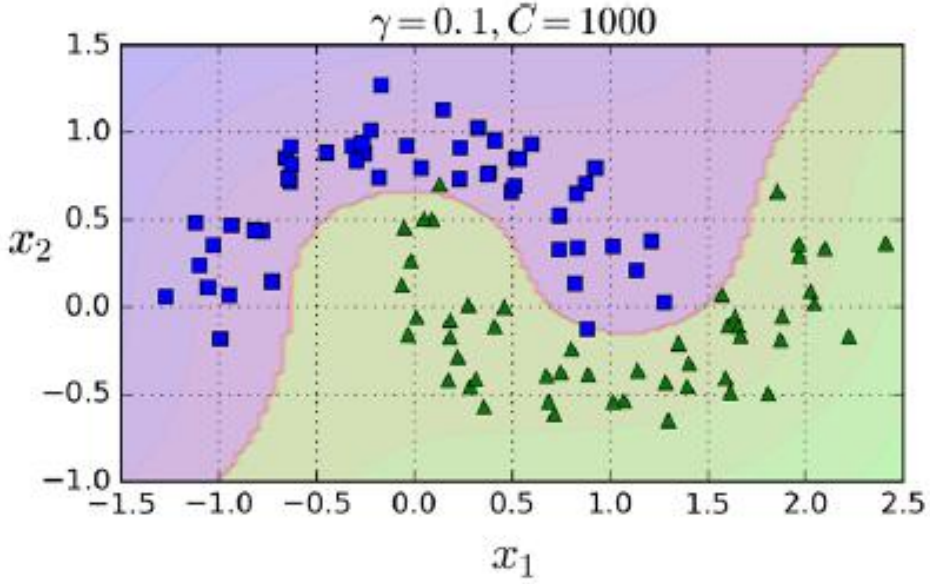
高斯 RBF 核

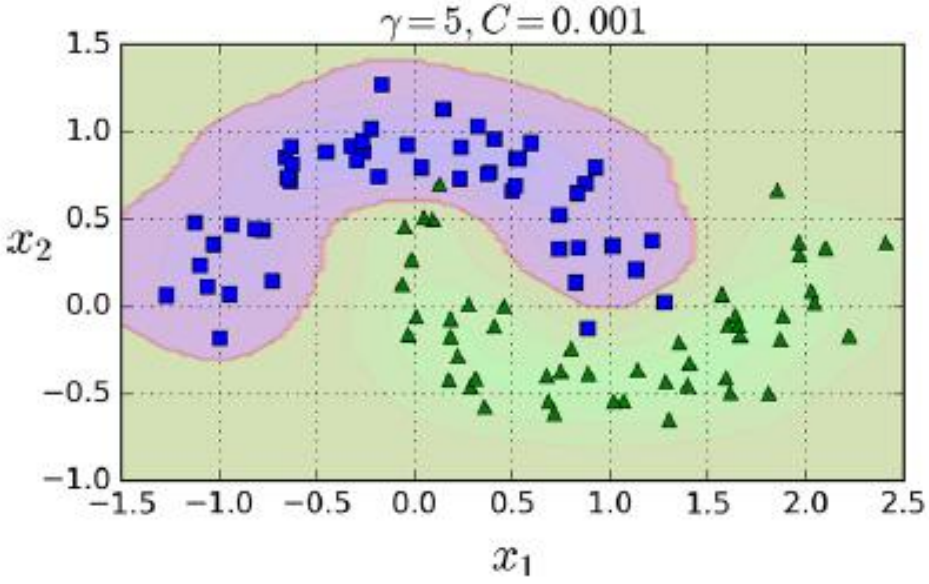
- 就像多项式特征法一样，相似特征法对各种机器学习算法同样也有不错的表现。但是在所有额外特征上的计算成本可能很高，特别是在大规模的训练集上。然而，“核”技巧再一次显现了它在 SVM 上的神奇之处：高斯核让你可以获得同样好的结果成为可能，就像你在相似特征法添加了许多相似特征一样，但事实上，你并不需要真正添加它们。

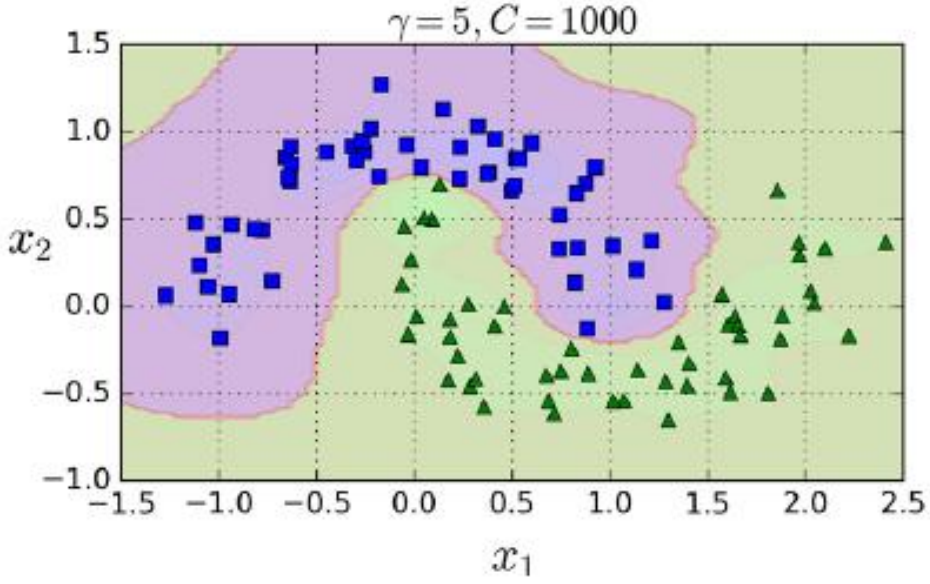
```
rbf_kernel_svm_clf = Pipeline((
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
  ))
rbf_kernel_svm_clf.fit(X, y)
```

高斯 RBF 核


$\gamma = 0.1, C = 0.001$
x_1
x_2


$\gamma = 0.1, C = 1000$
x_1
x_2


$\gamma = 5, C = 0.001$
x_1
x_2


$\gamma = 5, C = 1000$
x_1
x_2

计算复杂性

- LinearSVC类基于liblinear库，它实现了线性 SVM 的优化算法。它并不支持核技巧，但是它样本和特征的数量几乎是线性的：训练时间复杂度大约为 $O(m \times n)$.
- 如果你要非常高的精度，这个算法需要花费更多时间。这是由容差值超参数ϵ（在 Scikit-learn 称为tol）控制的。大多数分类任务中，使用默认容差值的效果是已经可以满足一般要求。

计算复杂性

- SVC 类基于libsvm库，它实现了支持核技巧的算法。训练时间复杂度通常介于$O(m^2 \times n)$和$O(m^3 \times n)$之间。不幸的是，这意味着当训练样本变大时，它将变得极其慢（例如，成千上万个样本）。这个算法对于复杂但小型或中等数量的数据集表现是完美的。然而，它能对特征数量很好的缩放，尤其对稀疏特征来说（sparse features）（即每个样本都有一些非零特征）。在这个情况下，算法对每个样本的非零特征的平均数量进行大概的缩放。

计算复杂性

Table 5-1. Comparison of Scikit-Learn classes for SVM classification

Class	Time complexity	Out-of-core support	Scaling required	Kernel trick
`LinearSVC`	$O(m \times n)$	No	Yes	No
`SGDClassifier`	$O(m \times n)$	Yes	Yes	No
`SVC`	$O(m^2 \times n)$ to $O(m^3 \times n)$	No	Yes	Yes

SVM 回归

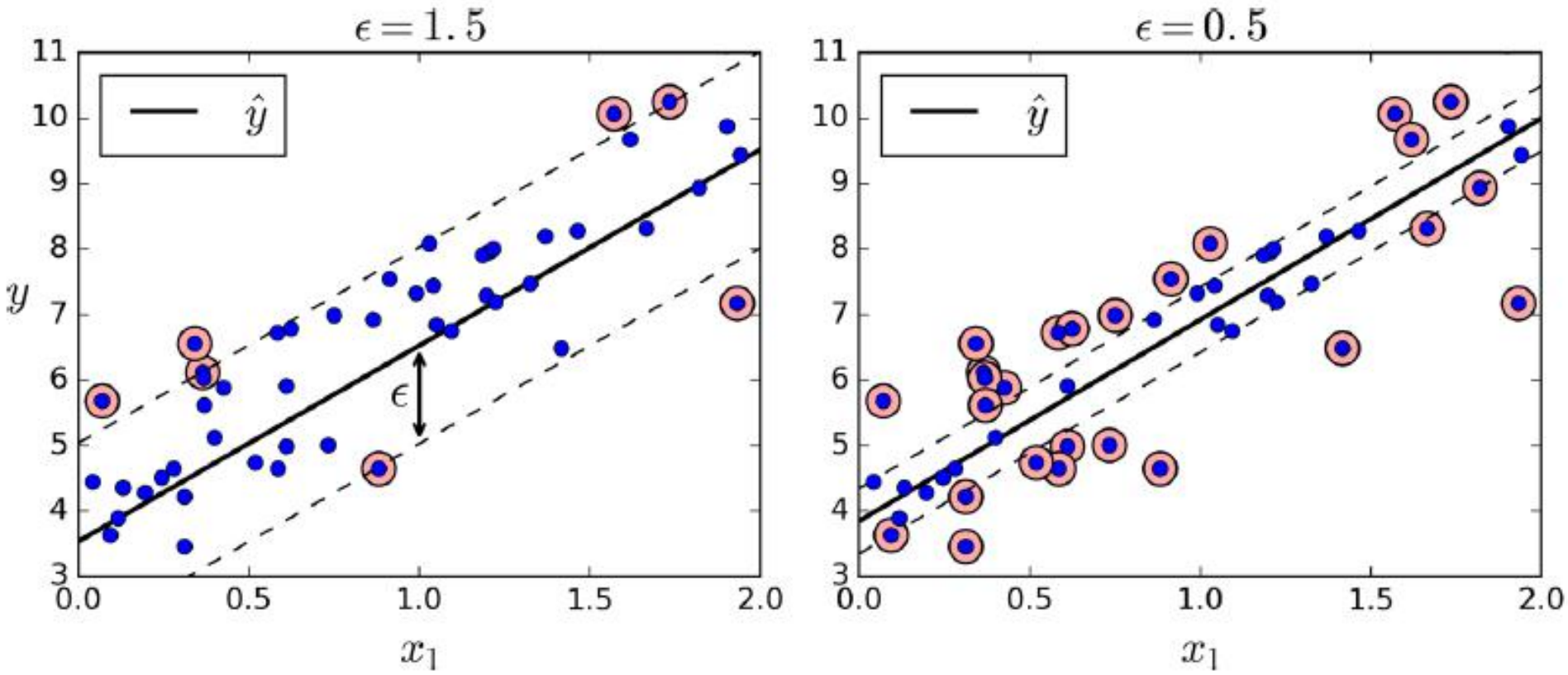
- 正如我们之前提到的，SVM 算法应用广泛：不仅仅支持线性和非线性的分类任务，还支持线性和非线性的回归任务。技巧在于逆转我们的目标：限制间隔违规的情况下，不是试图在两个类别之间找到尽可能大的“街道”（即间隔）。SVM 回归任务是限制间隔违规情况下，尽量放置更多的样本在“街道”上。“街道”的宽度由超参数ϵ控制。

SVM 回归

from sklearn.svm import LinearSVR

svm_reg = LinearSVR(epsilon=1.5)

svm_reg.fit(X, y)

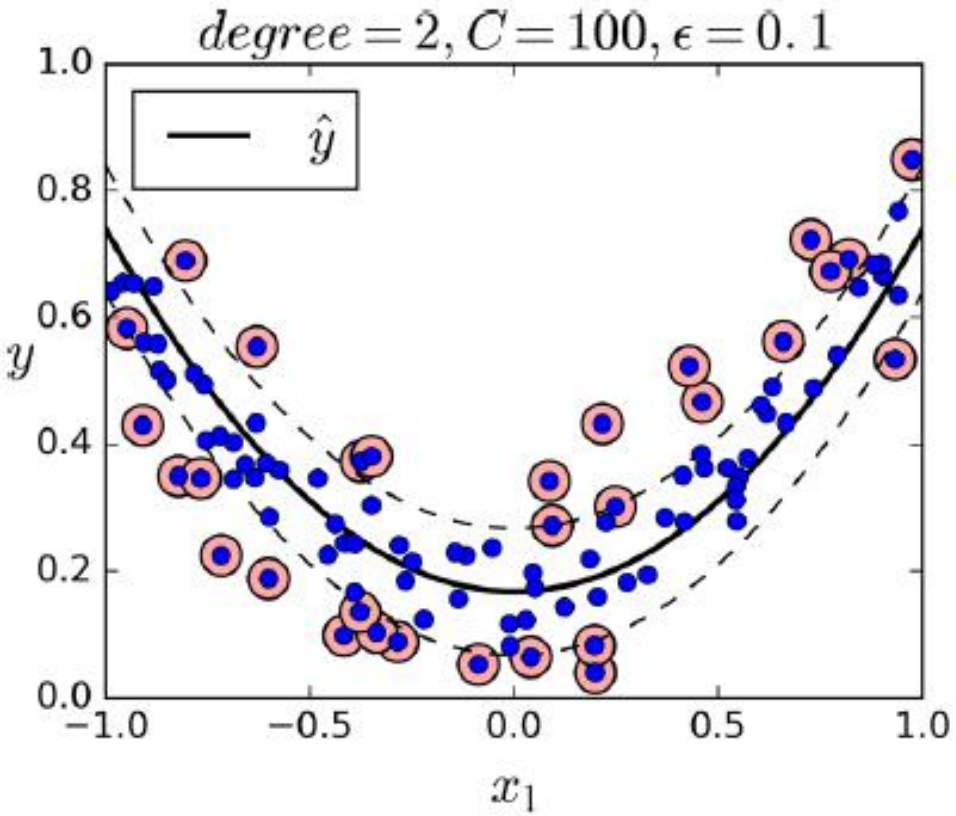


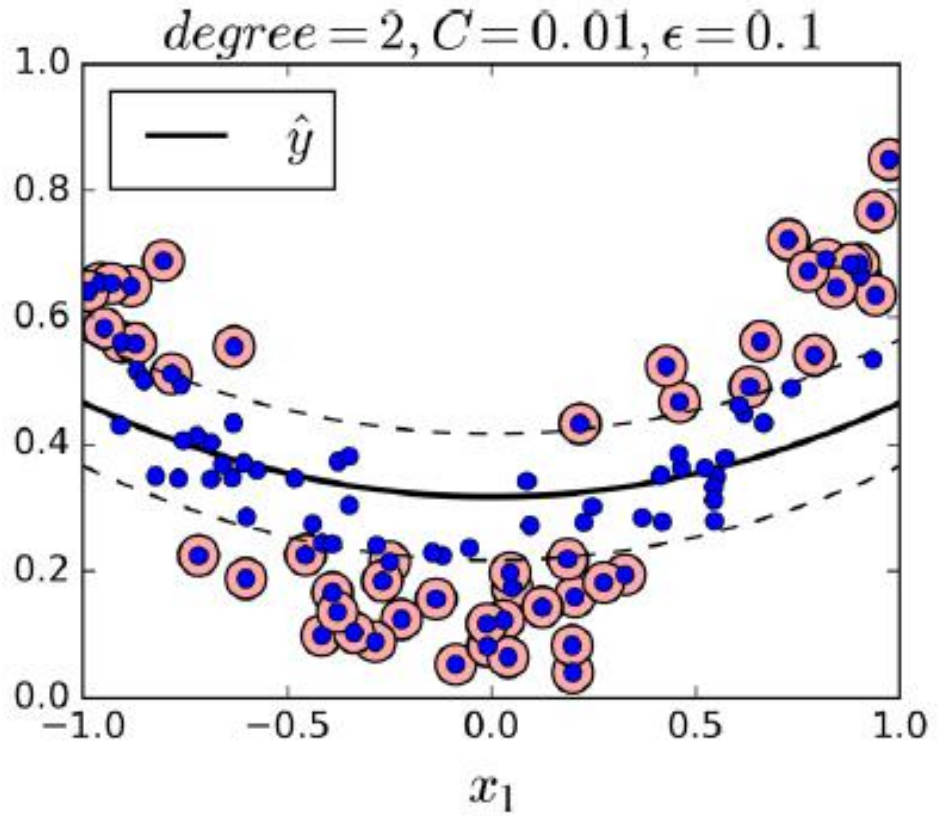
SVM 回归

from sklearn.svm import SVR

svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)

svm_poly_reg.fit(X, y)





背后机制

决策函数和预测

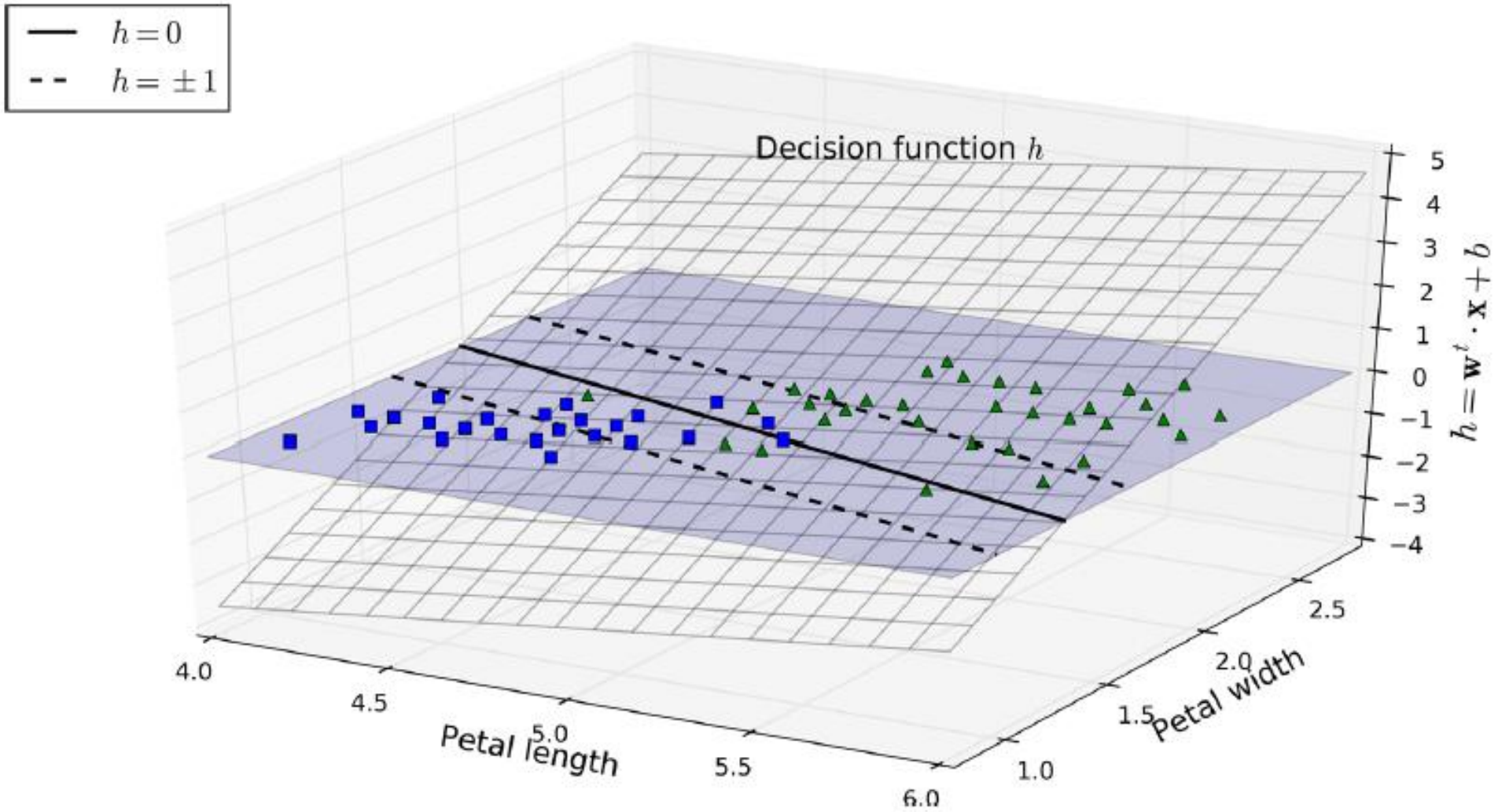
线性 SVM 分类器通过简单地计算决策函数 $\mathbf{w}^T \cdot \mathbf{x} + b = w_1 x_1 + \cdots + w_n x_n + b$ 来预测新样本的类别：如果结果是正的，预测类别$\hat{y}$是正类，为 1，否则就是负类，为 0。

Equation 5-2. Linear SVM classifier prediction

$$\hat{y} = \begin{cases} 0 \text{ if } \mathbf{w}^T \cdot \mathbf{x} + b < 0, \\ 1 \text{ if } \mathbf{w}^T \cdot \mathbf{x} + b \geq 0 \end{cases}$$

背后机制

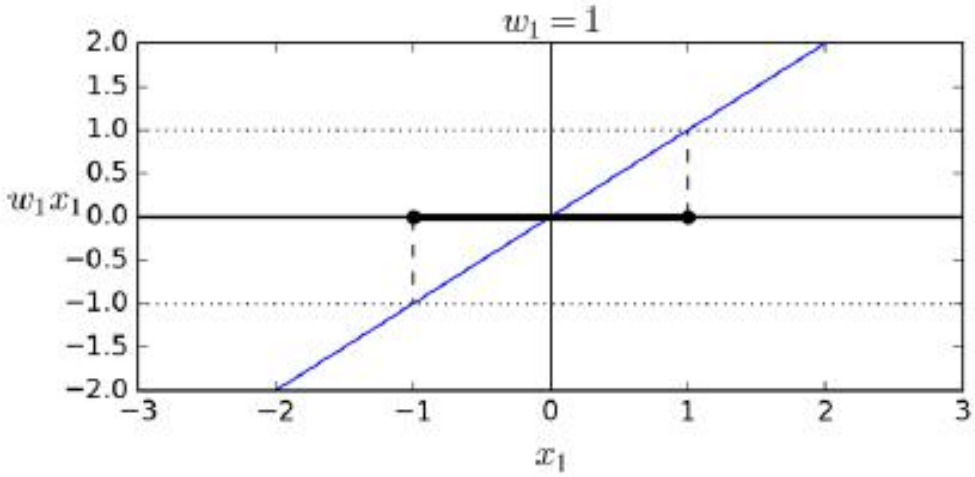
决策函数和预测

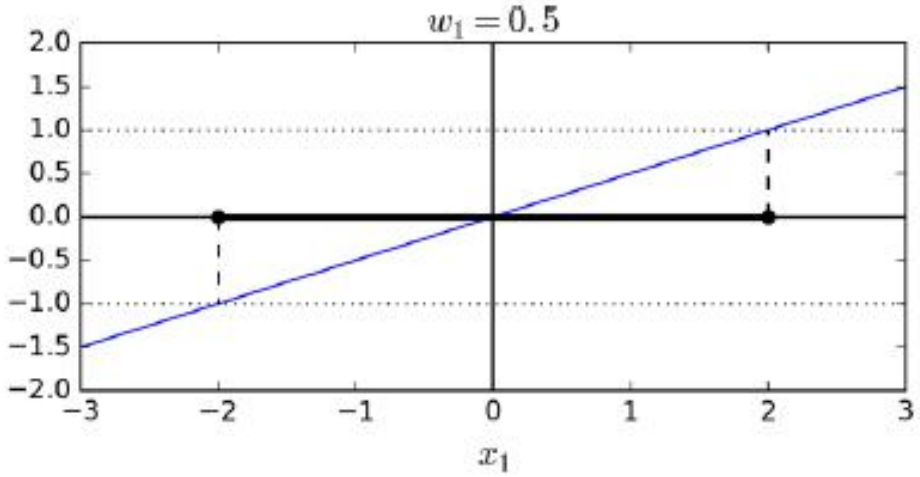


背后机制

训练目标

- 看下决策函数的斜率：它等于权重向量的范数 |w|。如果我们把这个斜率除于 2，决策函数等于 ±1 的点将会离决策边界原来的两倍大。换句话，即斜率除于 2，那么间隔将增加两倍。在下图 中，2D 形式比较容易可视化。权重向量w越小，间隔越大。





训练目标

- 所以我们的目标是最小化 |w|，从而获得大的间隔。然而，如果我们想要避免间隔违规（硬间隔），对于正的训练样本，我们需要决策函数大于 1，对于负训练样本，小于 -1。若我们对负样本（即 $y^{(i)} = 0$）定义 $t^{(i)} = -1$，对正样本（即 $y^{(i)} = 1$）定义 $t^{(i)} = 1$，那么我们可以对所有的样本表示为 $t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1$。因此，我们可以将硬间隔线性 SVM 分类器表示为如下的约束优化问题

Equation 5-3. Hard margin linear SVM classifier objective

$$\underset{\mathbf{w}, b}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w}$$

$$\text{subject to} \quad t^{(i)}\left(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b\right) \geq 1 \quad \text{for } i = 1, 2, \cdots, m$$

训练目标

- 为了获得软间隔的目标，我们需要对每个样本应用一个松弛变量（slack variable）$\zeta^{(i)} \geq 0$。$\zeta^{(i)}$ 表示了第i个样本允许违规间隔的程度。我们现在有两个不一致的目标：一个是使松弛变量尽可能的小，从而减小间隔违规，另一个是使$1/2\ \mathbf{w}^T \cdot \mathbf{w}$ 尽量小，从而增大间隔。这时C超参数发挥作用：它允许我们在两个目标之间权衡。因此我们得到了如下的约束优化问题。

Equation 5-4. Soft margin linear SVM classifier objective

$$\underset{\mathbf{w}, b, \zeta}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w} + C \sum_{i=1}^{m} \zeta^{(i)}$$

$$\text{subject to} \quad t^{(i)}\left(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b\right) \geq 1 - \zeta^{(i)} \quad \text{and} \quad \zeta^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

二次规划

- 硬间隔和软间隔都是线性约束的凸二次规划优化问题。这些问题被称之为二次规划（QP）问题。现在有许多解决方案可以使用各种技术来处理 QP 问题。

Equation 5-5. Quadratic Programming problem

$$\begin{aligned} \underset{\mathbf{p}}{\text{Minimize}} \quad & \frac{1}{2}\mathbf{p}^T \cdot \mathbf{H} \cdot \mathbf{p} + \mathbf{f}^T \cdot \mathbf{p} \\ \text{subject to} \quad & \mathbf{A} \cdot \mathbf{p} \le \mathbf{b} \end{aligned}$$

where
- $\mathbf{p}$ is an n_p-dimensional vector (n_p = number of parameters),
- $\mathbf{H}$ is an $n_p \times n_p$ matrix,
- $\mathbf{f}$ is an n_p-dimensional vector,
- $\mathbf{A}$ is an $n_c \times n_p$ matrix (n_c = number of constraints),
- $\mathbf{b}$ is an n_c-dimensional vector.

对偶问题

- 给出一个约束优化问题，即原始问题（primal problem），它可能表示不同但是和另一个问题紧密相连，称为对偶问题（Dual Problem）。对偶问题的解通常是对原始问题的解给出一个下界约束，但在某些条件下，它们可以获得相同解。幸运的是，SVM 问题恰好满足这些条件，所以你可以选择解决原始问题或者对偶问题，两者将会有相同解。如下表示了线性 SVM 的对偶形式：

Equation 5-6. Dual form of the linear SVM objective

$$\underset{\alpha}{\text{minimize}} \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha^{(i)}\alpha^{(j)}t^{(i)}t^{(j)}\mathbf{x}^{(i)T}\cdot\mathbf{x}^{(j)} - \sum_{i=1}^{m}\alpha^{(i)}$$

$$\text{subject to} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

对偶问题

- 一旦你找到最小化公式的向量α（使用 QP 解决方案），你可以通过使用如下公式计算w和b。

Equation 5-7. From the dual solution to the primal solution

$$\widehat{\mathbf{w}} = \sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \left(1 - t^{(i)} \left(\widehat{\mathbf{w}}^T \cdot \mathbf{x}^{(i)}\right)\right)$$

核化支持向量机

- 假设你想把一个 2 次多项式变换应用到二维空间的训练集（例如卫星数据集），然后在变换后的训练集上训练一个线性SVM分类器。公式 5-8 显示了可以应用的 2 次多项式映射函数ϕ。

Equation 5-8. Second-degree polynomial mapping

$$\phi(\mathbf{x}) = \phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} {x_1}^2 \\ \sqrt{2}\, x_1 x_2 \\ {x_2}^2 \end{pmatrix}$$

核化支持向量机

- 注意到转换后的向量是 3 维的而不是 2 维。如果我们应用这个 2 次多项式映射，然后计算转换后向量的点积（见公式 5-9），让我们看下两个 2 维向量a和b会发生什么。

Equation 5-9. Kernel trick for a 2nd-degree polynomial mapping

$$\phi(\mathbf{a})^T \cdot \phi(\mathbf{b}) = \begin{pmatrix} a_1^2 \\ \sqrt{2}\,a_1 a_2 \\ a_2^2 \end{pmatrix}^T \cdot \begin{pmatrix} b_1^2 \\ \sqrt{2}\,b_1 b_2 \\ b_2^2 \end{pmatrix} = a_1^2 b_1^2 + 2a_1 b_1 a_2 b_2 + a_2^2 b_2^2$$

$$= (a_1 b_1 + a_2 b_2)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^T \cdot \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = \left(\mathbf{a}^T \cdot \mathbf{b}\right)^2$$

核化支持向量机

- 如果你应用转换ϕ到所有训练样本，那么对偶问题（见公式 5-6）将会包含点积 $\phi(\mathbf{x}^{(i)})^T \cdot \phi(\mathbf{x}^{(j)})$ 。但如果ϕ像在公式 5-8 定义的 2 次多项式转换，那么你可以将这个转换后的向量点积替换为$\left(\mathbf{x}^{(i)T} \cdot \mathbf{x}^{(j)}\right)^2$。所以实际上你根本不需要对训练样本进行转换：仅仅需要在公式 5-6 中，将点积替换成它点积的平方。结果将会和你经过麻烦的训练集转换并拟合出线性 SVM 算法得出的结果一样，但是这个技巧使得整个过程在计算上面更有效率。这就是核技巧的精髓。

核化支持向量机

- 函数 $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T \cdot \mathbf{b})^2$ 被称为二次多项式核（polynomial kernel）。在机器学习，核函数是一个能计算点积的函数，并只基于原始向量a和b，不需要计算（甚至知道）转换 ϕ。公式 5-10 列举了一些最常用的核函数。

Equation 5-10. Common kernels

$$\text{Linear:} \quad K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \cdot \mathbf{b}$$

$$\text{Polynomial:} \quad K(\mathbf{a}, \mathbf{b}) = \left(\gamma \mathbf{a}^T \cdot \mathbf{b} + r\right)^d$$

$$\text{Gaussian RBF:} \quad K(\mathbf{a}, \mathbf{b}) = \exp\left(-\gamma \| \mathbf{a} - \mathbf{b} \|^2\right)$$

$$\text{Sigmoid:} \quad K(\mathbf{a}, \mathbf{b}) = \tanh\left(\gamma \mathbf{a}^T \cdot \mathbf{b} + r\right)$$

核化支持向量机

- 我们还有一个问题要解决。公式 5-7 展示了线性 SVM 分类器如何从对偶解到原始解，如果你应用了核技巧那么得到的公式会包含$\phi(x^{(i)})$。事实上，w必须和 $\phi(x^{(i)})$ 有同样的维度，这可能会对应巨大的甚至无限的维度，所以你很难计算它。但怎么在不知道w的情况下做出预测？

核化支持向量机

- 好消息是你可以将公式 5-7 的w代入到新的样本 $\mathbf{x}^{(n)}$ 的决策函数中，你会得到一个在输入向量之间只有点积的方程式。这时，核技巧将派上用场，见公式 5-11：

Equation 5-11. Making predictions with a kernelized SVM

$$
\begin{aligned}
h_{\hat{\mathbf{w}},\hat{b}}\left(\phi\left(\mathbf{x}^{(n)}\right)\right) &= \hat{\mathbf{w}}^T \cdot \phi\left(\mathbf{x}^{(n)}\right) + \hat{b} = \left(\sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \phi\left(\mathbf{x}^{(i)}\right)\right)^T \cdot \phi\left(\mathbf{x}^{(n)}\right) + \hat{b} \\
&= \sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \left(\phi\left(\mathbf{x}^{(i)}\right)^T \cdot \phi\left(\mathbf{x}^{(n)}\right)\right) + \hat{b} \\
&= \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \hat{\alpha}^{(i)} t^{(i)} K\left(\mathbf{x}^{(i)}, \mathbf{x}^{(n)}\right) + \hat{b}
\end{aligned}
$$

核化支持向量机

- 注意到支持向量才满足$\alpha^{(i)} \neq 0$，做出预测只涉及计算为支持向量部分的输入样本$\mathbf{x}^{(n)}$ 的点积，而不是全部的训练样本。当然，你同样也需要使用同样的技巧来计算偏置项b，见公式 5-12

Equation 5-12. Computing the bias term using the kernel trick

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \left(1 - t^{(i)} \widehat{\mathbf{w}}^T \cdot \phi\left(\mathbf{x}^{(i)}\right)\right) = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \left(1 - t^{(i)} \left(\sum_{j=1}^{m} \hat{\alpha}^{(j)} t^{(j)} \phi\left(\mathbf{x}^{(j)}\right)\right)^T \cdot \phi\left(\mathbf{x}^{(i)}\right)\right)$$

$$= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \left(1 - t^{(i)} \sum_{\substack{j=1 \\ \hat{\alpha}^{(j)} > 0}}^{m} \hat{\alpha}^{(j)} t^{(j)} K\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right)\right)$$

在线支持向量机

- 对于线性SVM分类器，一种方式是使用梯度下降（例如使用SGDClassifire）最小化代价函数，如从原始问题推导出的公式 5-13。不幸的是，它比基于 QP 方式收敛慢得多。

Equation 5-13. Linear SVM classifier cost function

$$J(\mathbf{w}, b) = \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w} \quad + \quad C \sum_{i=1}^{m} max\left(0, 1 - t^{(i)}\left(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b\right)\right)$$

在线支持向量机

- 代价函数的第一部分会使模型有一个小的权重向量w，从而获得一个更大的间隔。第二部分计算所有间隔违规的总数。如果样本位于“街道”上和正确的一边，或它与“街道”正确一边的距离成比例，则间隔违规等于 0。最小化保证了模型的间隔违规尽可能小并且少。

Equation 5-13. Linear SVM classifier cost function

$$J(\mathbf{w}, b) = \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w} \quad + \quad C \sum_{i=1}^{m} max\left(0, 1 - t^{(i)}\left(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b\right)\right)$$