第 14 章 支持向量机

支持向量机(Support Vector Machine，简记 SVM)出现于 1990 年代，正式发表于 Cortes and Vapnik (1995)。

SVM 的基本思想是，通过寻找最优的“分离超平面”(separating hyperplane)，将两类数据分离开。

SVM 特别适用于变量很多的数据，因为在高维空间，数据被“打散”，故更容易用超平面进行分离。

SVM 在变量较多的数据中有很多成功的应用，比如文本分析与图像识别。例如，SVM 曾在手写数字识别的 MNIST 数据集取得巨大成功。

14.1 分离超平面

考虑以下两类数据，参见图 14.1。

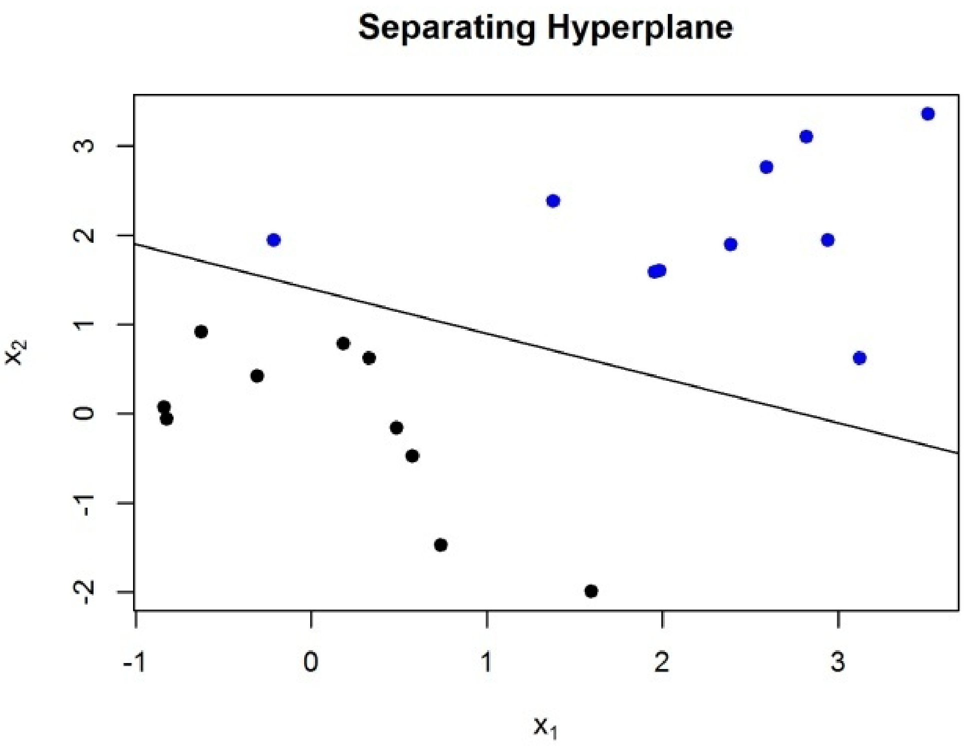

图 14.1 分离超平面

如果在三维空间(有 3 个特征变量)，则可由一个平面分离。

更一般地，推广到高维空间，则可由一个“超平面”(hyperplane)分离，称为**分离超平面**(separating hyperplane)。

假设有 p 个特征变量，则分离超平面 L 的方程可写为

$$\beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p = \beta_0 + \boldsymbol{\beta}'\mathbf{x} = 0 \qquad (14.1)$$

此分离超平面 L 将 p 维特征空间(feature space)一分为二。

如果$\beta_0 + \boldsymbol{\beta}'\mathbf{x} > 0$，则观测值$\mathbf{x}$落于超平面$L$的一边。

反之，如果$\beta_0 + \boldsymbol{\beta}'\mathbf{x} < 0$，则观测值$\mathbf{x}$落于超平面$L$的另一边。

$\left|\beta_0 + \boldsymbol{\beta}'\mathbf{x}\right|$(绝对值)的大小可用于度量观测值$\mathbf{x}$到超平面$L$的距离远近。

如果两类数据之间存在分离超平面，则称数据为**线性可分**(linearly separable)。

在线性可分的情况下，分离超平面一般并不唯一，因为总可以稍微移动超平面，而依然将两类数据分离，参见图 14.2。

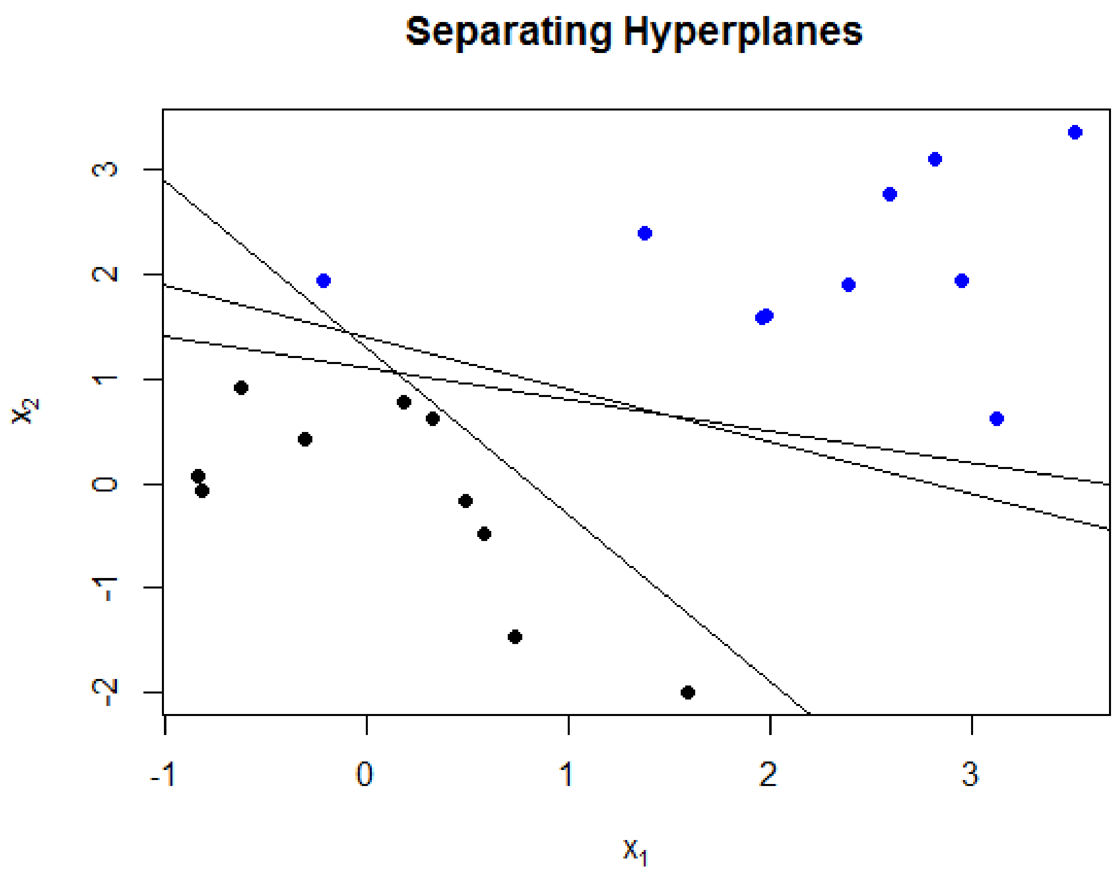

图 14.2 分离超平面并不唯一

14.2 最大间隔分类器

针对分离超平面不唯一的问题，一种解决方法是使分离超平面离两类数据尽量远。

希望在两类数据之间有一条“隔离带”，而且这条隔离带越宽越好。

这就是所谓**最大间隔分类器**(maximal margin classifier)；俗称“最宽街道法”(widest street approach)，即在两类数据之间建一条最宽的街道。

对于训练数据$\left\{\mathbf{x}_i, y_i\right\}_{i=1}^{n}$，考虑二分类问题。

记响应变量，即“类别标签”(class label)为$y_i \in \{-1, 1\}$，参见图 14.3。

其中，“$y_i = 1$”为一类数据（称为“正例”，图中的蓝点）。

而“$y_i = -1$”为另一类数据(称为“反例”，图中的黑点)。

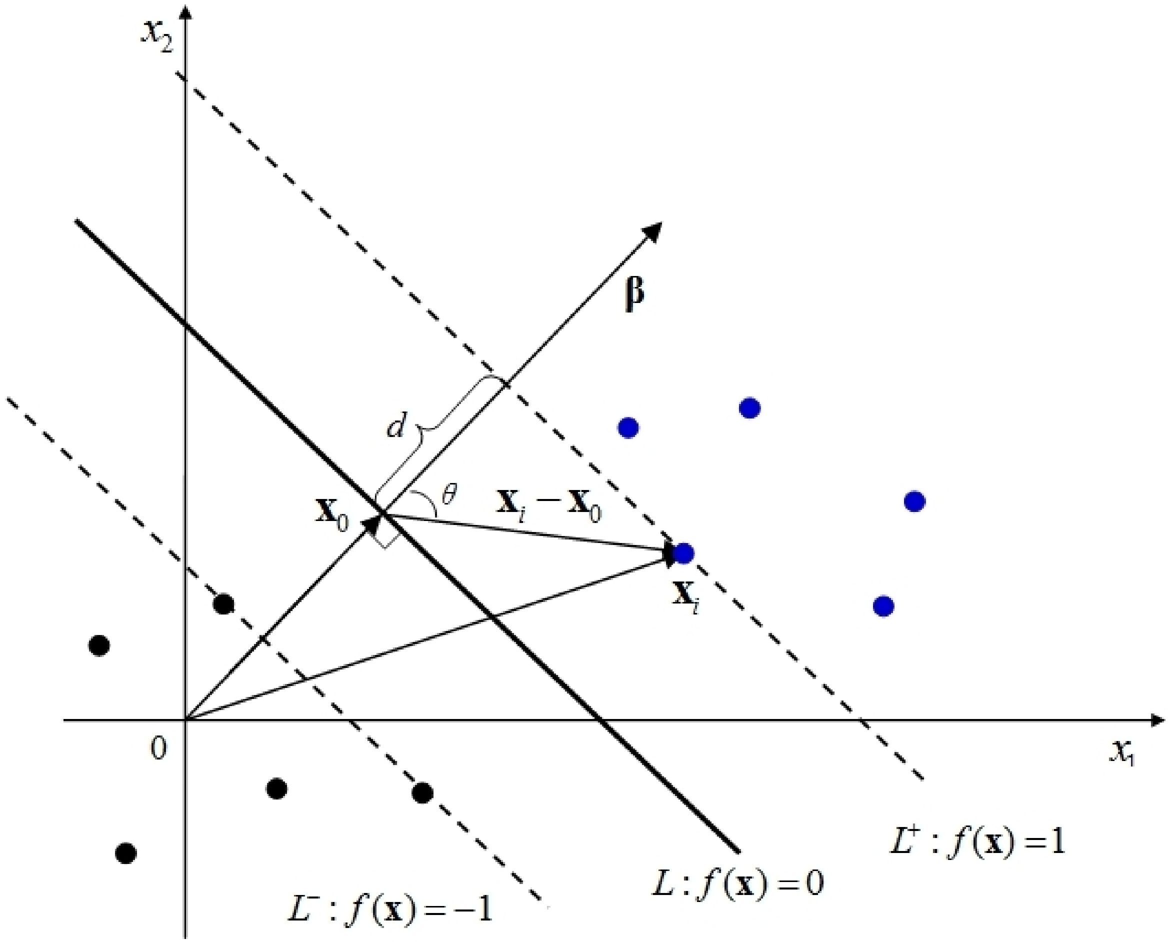

图 14.3 观测数据到分离超平面的符号距离

希望用超平面分离这两类数据，即找到一个函数 $f(\mathbf{x}) = \beta_0 + \mathbf{x}'\boldsymbol{\beta}$，若 $f(\mathbf{x}) > 0$，则预测 $\hat{y} = 1$；反之，若 $f(\mathbf{x}) < 0$，预测 $\hat{y} = -1$。

记分离超平面为 $L \equiv \{\mathbf{x} : f(\mathbf{x}) = 0\}$(图 14.3 中蓝色粗线),则超平面 L 的方程为

$$f(\mathbf{x}) \equiv \beta_0 + \boldsymbol{\beta}'\mathbf{x} = 0 \qquad (14.2)$$

其中，$\boldsymbol{\beta}$ 为垂直于此超平面的“法向量”(normal vector)，证明如下。

任给在超平面L上的两点$\mathbf{x}$与$\tilde{\mathbf{x}}$，则向量$(\mathbf{x}-\tilde{\mathbf{x}})$也位于此超平面上，而且与$\boldsymbol{\beta}$正交(即内积为 0)：

$$\beta_0+\boldsymbol{\beta}'\mathbf{x}=0,\quad \beta_0+\boldsymbol{\beta}'\tilde{\mathbf{x}}=0 \quad\Rightarrow\quad \boldsymbol{\beta}'(\mathbf{x}-\tilde{\mathbf{x}})=0 \tag{14.3}$$

其中，上式的左边两式相减，即得右边。记法向量$\boldsymbol{\beta}$方向与超平面L的交点为$\mathbf{x}_0$，则从观测值$\mathbf{x}_i$到超平面L的最短(垂直)距离为(参见图 14.3)：

$$
\begin{aligned}
d(\mathbf{x}_i, L) &= \left\|\mathbf{x}_i - \mathbf{x}_0\right\| \cos\theta \qquad &&(\text{向量夹角的余弦公式}) \\
&= \left\|\mathbf{x}_i - \mathbf{x}_0\right\| \cdot \frac{\boldsymbol{\beta}'(\mathbf{x}_i - \mathbf{x}_0)}{\left\|\mathbf{x}_i - \mathbf{x}_0\right\| \left\|\boldsymbol{\beta}\right\|} \qquad &&(\text{消去} \left\|\mathbf{x}_i - \mathbf{x}_0\right\|) \\
&= \frac{\boldsymbol{\beta}'(\mathbf{x}_i - \mathbf{x}_0)}{\left\|\boldsymbol{\beta}\right\|} \qquad &&(\text{乘积展开}) \\
&= \frac{\boldsymbol{\beta}'\mathbf{x}_i - \boldsymbol{\beta}'\mathbf{x}_0}{\left\|\boldsymbol{\beta}\right\|}
\end{aligned}
\tag{14.4}
$$

其中，由于$-1 \leq \cos\theta \leq 1$，故距离$d$可正可负，取决于$\mathbf{x}_i$在超平面$L$的哪一侧，称为“符号距离”(signed distance)。

由于$\mathbf{x}_0$在超平面L上，满足超平面的方程，故$-\boldsymbol{\beta}'\mathbf{x}_0=\beta_0$，代入上式可得：

$$d(\mathbf{x}_i, L)=\frac{\beta_0+\boldsymbol{\beta}'\mathbf{x}_i}{\|\boldsymbol{\beta}\|}=\frac{f(\mathbf{x}_i)}{\|\boldsymbol{\beta}\|} \tag{14.5}$$

$f(\mathbf{x}_i)$离 0 越远，则观测值$\mathbf{x}_i$离超平面$L=\{\mathbf{x}: f(\mathbf{x})=0\}$越远，只是需要将此符号距离进行标准化，即除以$\|\boldsymbol{\beta}\|$(向量$\boldsymbol{\beta}$的长度)。

假设超平面L可将两类数据完全分离，即所谓“分离超平面”(separating hyperplane)，则对于所有“$y_i = 1$”的正例(positive sample)，都有$f(\mathbf{x}_i) > 0$；而对于所有“$y_i = -1$”的反例(negative sample)，都有$f(\mathbf{x}_i) < 0$。

故分离超平面就是决策边界(decision boundary)。因此，分类规则为

$$\hat{y} = \mathrm{sign}\left(\beta_0 + \boldsymbol{\beta}'\mathbf{x}\right) \qquad (14.6)$$

其中，$\mathrm{sign}(\cdot)$为“符号函数”(sign function)，即

$$\text{sign}(z) = \begin{cases} 1 & if \quad z > 0 \\ -1 & if \quad z \le 0 \end{cases} \tag{14.7}$$

希望所有样本点到分离超平面L的距离越远越好。

在所有样本点中，到分离超平面L的最小距离之两倍，称为**间隔** (margin)。

在此例中，这三个样本点(向量)完全决定了“最优分离超平面”(optimal separating hyperplane)与“最大间隔”(maximal margin)的位置，故称为**支持向量**(support vectors)。

称蓝色支持向量(正例)所处的间隔为“正间隔”(positive margin)，记为 L^+。

称黑色支持向量(反例)所处的间隔为“负间隔”(negative margin)，记为 L^-。

由于 $f(\mathbf{x}) = \beta_0 + \boldsymbol{\beta}'\mathbf{x}$ 为线性函数，故可通过线性变换(比如，乘以某常数 c)，使得对于所有在正间隔 L^+ 上的样本点，都有 $f(\mathbf{x}_i) = 1$；而对于所有在负间隔 L^- 上的样本点，都有 $f(\mathbf{x}_i) = -1$。

考虑正间隔L^+上的某样本点$\mathbf{x}^*$，则$\mathbf{x}^*$到分离超平面L的距离之两倍，即为正间隔L^+与负间隔L^-之间的**最大间隔**(maximal margin)：

$$2d(\mathbf{x}^*, L) = 2\frac{f(\mathbf{x}^*)}{\|\boldsymbol{\beta}\|} = \frac{2}{\|\boldsymbol{\beta}\|} \tag{14.8}$$

其中，$f(\mathbf{x}^*) = 1$，因为$\mathbf{x}^*$位于正间隔L^+上。

上式为最大化的目标函数，而约束条件则是所有样本点都能正确分类。

在完全正确分类的情况下，对于所有“$y_i = 1$”的正样例，都有 $f(\mathbf{x}_i) \geq 1$，故 $y_i f(\mathbf{x}_i) \geq 1$。

反之，对于所有“$y_i = -1$”的负样例，都有 $f(\mathbf{x}_i) \leq -1$，故依然 $y_i f(\mathbf{x}_i) \geq 1$。

无论是正样例，还是负样例，约束条件都是 $y_i f(\mathbf{x}_i) \geq 1$。

这正是令 $y_i \in \{-1, 1\}$ 的方便之处。

求解最大间隔之超平面的约束极值问题为

$$\max_{\boldsymbol{\beta},\beta_0} \frac{2}{\|\boldsymbol{\beta}\|} \qquad (14.9)$$
$$s.t.\ \ y_i f(\mathbf{x}_i) \geq 1,\ \ i = 1, \cdots, n$$

根据此问题所得的分类器，称为**最大间隔分类器**(maximal margin classifier)，参见图 14.4。

在训练集中的间隔越大，则我们期待在测试集中的间隔也越大，由此带来更好的泛化能力。

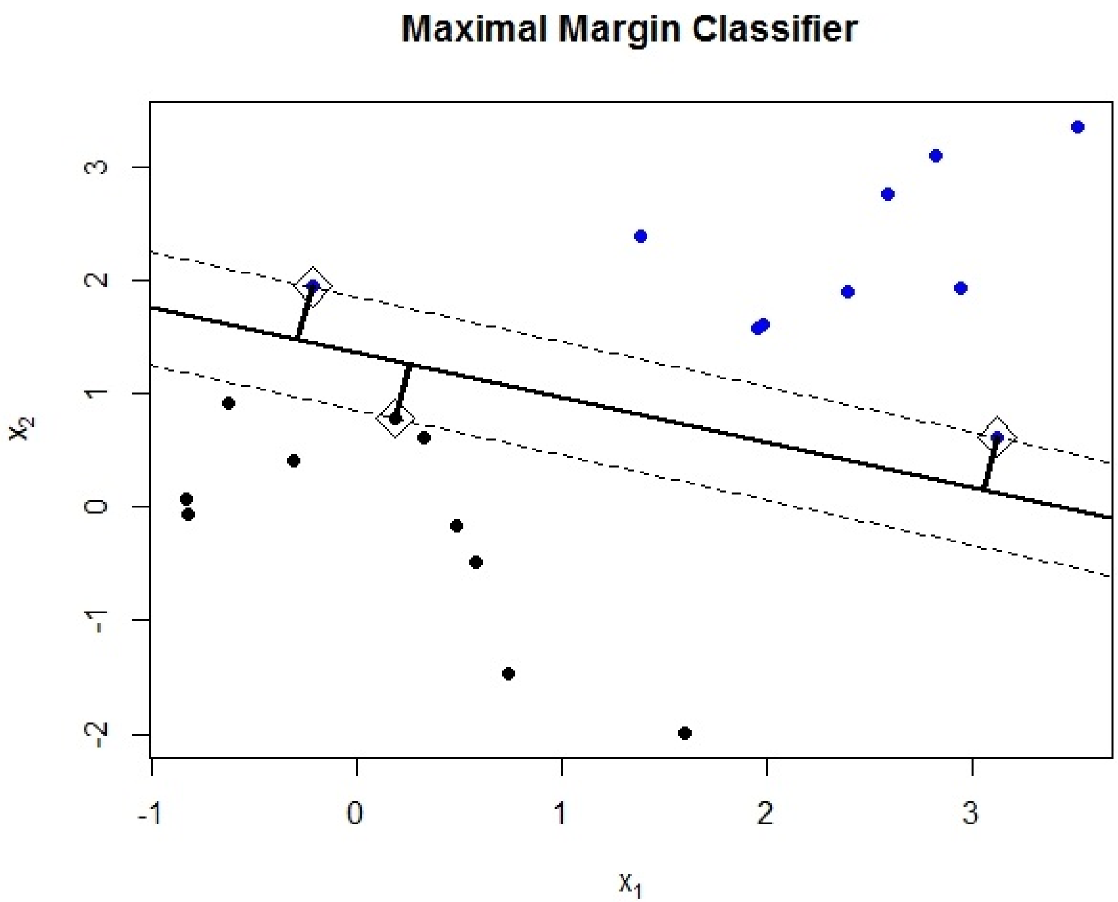

图 14.4 最大间隔分类器

最大化$\dfrac{2}{\|\boldsymbol{\beta}\|}$等价于最小化$\|\boldsymbol{\beta}\|$，而后者又等价于最小化$\dfrac{1}{2}\|\boldsymbol{\beta}\|^2 = \dfrac{1}{2}\boldsymbol{\beta}'\boldsymbol{\beta}$。

将“$f(\mathbf{x}_i) = \beta_0 + \boldsymbol{\beta}'\mathbf{x}_i$”代入约束条件，则最优化问题可写为

$$\begin{aligned} &\min_{\boldsymbol{\beta},\beta_0} \ \frac{1}{2}\boldsymbol{\beta}'\boldsymbol{\beta} \\ &s.t. \ \ y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) \geq 1, \ \ i = 1, \cdots, n \end{aligned} \tag{14.10}$$

由于目标函数$\frac{1}{2}\boldsymbol{\beta}'\boldsymbol{\beta} = \frac{1}{2}(\beta_1^2 + \cdots + \beta_p^2)$为二次型，而约束条件$y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) \geq 1$为线性不等式约束，故为“凸二次规划” (convex quadratic programming)问题。

为求解此问题，引入“原问题”(primal problem)的拉格朗日乘子函数L_P：

$$\min_{\boldsymbol{\beta},\,\beta_0,\,\boldsymbol{\alpha}} \ L_P(\boldsymbol{\beta},\beta_0,\boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{\beta}'\boldsymbol{\beta} - \sum_{i=1}^{n}\alpha_i\left[y_i(\beta_0+\boldsymbol{\beta}'\mathbf{x}_i)-1\right] \qquad (\text{乘积展开})$$

$$= \frac{1}{2}\boldsymbol{\beta}'\boldsymbol{\beta} - \beta_0\sum_{i=1}^{n}\alpha_i y_i - \boldsymbol{\beta}'\sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i + \sum_{i=1}^{n}\alpha_i$$

(14.11)

其中，$\boldsymbol{\alpha} \equiv (\alpha_1 \cdots \alpha_n)'$为对应于约束条件$y_i(\beta_0+\boldsymbol{\beta}'\mathbf{x}_i)\geq 1$的$n$个拉格朗日乘子。

使用向量微分规则，将此拉格朗日函数分别对$(\boldsymbol{\beta},\beta_0)$求偏导数，可得一阶条件：

$$\frac{\partial L_P}{\partial \boldsymbol{\beta}} = \boldsymbol{\beta} - \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i = 0 \quad \Rightarrow \quad \boldsymbol{\beta} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i \qquad (14.12)$$

$$\frac{\partial L_P}{\partial \beta_0} = -\sum_{i=1}^{n} \alpha_i y_i = 0 \quad \Rightarrow \quad \sum_{i=1}^{n} \alpha_i y_i = 0 \qquad (14.13)$$

从方程(14.12)可知，最优$\boldsymbol{\beta}$为各样本点数据的线性组合。

由于原问题(14.10)包含不等式约束，故最优解还需满足以下“Karush-Kuhn-Tucker”(简记 KKT)条件(参见第 3 章)：

$$\begin{cases} \alpha_i \geq 0\,; \\ y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) \geq 1\,; \\ \alpha_i\left[y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) - 1\right] = 0 \end{cases} \tag{14.14}$$

其中，$i = 1, \cdots, n$。从 KKT 条件的第 3 个方程可知，要么$\alpha_i = 0$，要么$y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) = 1$。

如果$\alpha_i = 0$, 则$y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) > 1$, 说明观测值$\mathbf{x}_i$在间隔之外；而且，观测值$\mathbf{x}_i$不影响拉格朗日函数(因为$\alpha_i = 0$)。

反之，如果$\alpha_i > 0$，则$y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) = 1$，说明观测值$\mathbf{x}_i$正好在间隔之上，即所谓“支持向量”(support vectors)；而且，支持向量会影响拉格朗日函数的最优化(因为$\alpha_i > 0$)。

在估计完模型后，大部分的训练数据都无须保留，最终模型只与支持向量有关。

但究竟哪些样本点是支持向量，依然取决于全部数据。

可以证明，原问题的最优解为拉格朗日乘子函数(14.11)的“鞍点”(saddle point)，故单独从拉格朗日乘子$\boldsymbol{\alpha}$来看，则为最大化问题。

将一阶条件(14.12)与(14.13)代回拉格朗日函数(14.11)，可得到一个最大化的“对偶问题” (dual problem)L_D：

$$
\begin{aligned}
\max_{\boldsymbol{\alpha}} \ L_D &= \frac{1}{2}\underbrace{\left(\sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i\right)'}_{=\hat{\boldsymbol{\beta}}'}\underbrace{\left(\sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i\right)}_{=\hat{\boldsymbol{\beta}}} - \beta_0 \underbrace{\sum_{i=1}^{n}\alpha_i y_i}_{=0} \\
&\quad - \underbrace{\left(\sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i\right)'}_{=\hat{\boldsymbol{\beta}}'}\underbrace{\left(\sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i\right)}_{=\hat{\boldsymbol{\beta}}} + \sum_{i=1}^{n}\alpha_i \\
&= \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\left(\sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i\right)'\left(\sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i\right) \qquad \text{(合并同类项)} \\
&= \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\left(\sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i'\right)\left(\sum_{j=1}^{n}\alpha_j y_j \mathbf{x}_j\right) \qquad \text{(转置、最后一项下标}\, i \,\text{改为}\, j\text{)} \\
&= \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j \mathbf{x}_i'\mathbf{x}_j \qquad \text{(乘积展开)} \\
&= \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j \left\langle \mathbf{x}_i, \mathbf{x}_j \right\rangle
\end{aligned}
\tag{14.15}
$$

其中，$\mathbf{x}_i'\mathbf{x}_j$为$\mathbf{x}_i$与$\mathbf{x}_j$的内积，可记为$\left\langle \mathbf{x}_i, \mathbf{x}_j \right\rangle$。

特征向量$\left\{\mathbf{x}_i\right\}_{i=1}^n$仅通过相互之间内积的方式而影响最优解。

这为第 14.5 节在支持向量机中使用“核技巧”(kernel trick)提供了方便。

最大化问题(14.15)是关于拉格朗日乘子$\boldsymbol{\alpha}$的二次(型)规划问题。

求解此对偶问题，并将所得解$\hat{\boldsymbol{\alpha}} = (\hat{\alpha}_1 \cdots \hat{\alpha}_n)'$，代回最优$\boldsymbol{\beta}$的表达式(14.12)可得，$\hat{\boldsymbol{\beta}} = \sum_{i=1}^{n} \hat{\alpha}_i y_i \mathbf{x}_i$。

对于截距项β_0，可通过支持向量来求解。假设$(\mathbf{x}_s, y_s)$为任意支持向量，则该支持向量在间隔上，故满足

$$y_s(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_s) = 1 \qquad (14.16)$$

在上式中，代入$\hat{\boldsymbol{\beta}}$，即可求得$\hat{\beta}_0$：

$$\hat{\beta}_0 = \frac{1}{y_s} - \hat{\boldsymbol{\beta}}'\mathbf{x}_s \tag{14.17}$$

此式对所有支持向量均成立。故更稳健(robust)的作法是，针对方程(14.17)，对所有支持向量进行平均：

$$\hat{\beta}_0 = \frac{1}{|S|}\sum_{s\in S}\left(\frac{1}{y_s} - \hat{\boldsymbol{\beta}}'\mathbf{x}_s\right) \tag{14.18}$$

其中，$S = \{s \mid \alpha_s > 0\}$为所有支持向量的下标集，而$|S|$为支持向量的个数。

由此可得最优分离超平面的估计方程为

$$\hat{f}(\mathbf{x}) = \hat{\beta}_0 + \hat{\boldsymbol{\beta}}'\mathbf{x} = \hat{\beta}_0 + \left(\sum_{i=1}^{n} \hat{\alpha}_i y_i \mathbf{x}_i\right)\mathbf{x} \tag{14.19}$$

然后，可用$\text{sign}\left(\hat{f}(\mathbf{x})\right) = \text{sign}\left(\hat{\beta}_0 + \hat{\boldsymbol{\beta}}'\mathbf{x}\right)$进行分类预测。

14.3 软间隔分类器

并非所有数据都是线性可分的，例如图 14.5 中的两类数据。

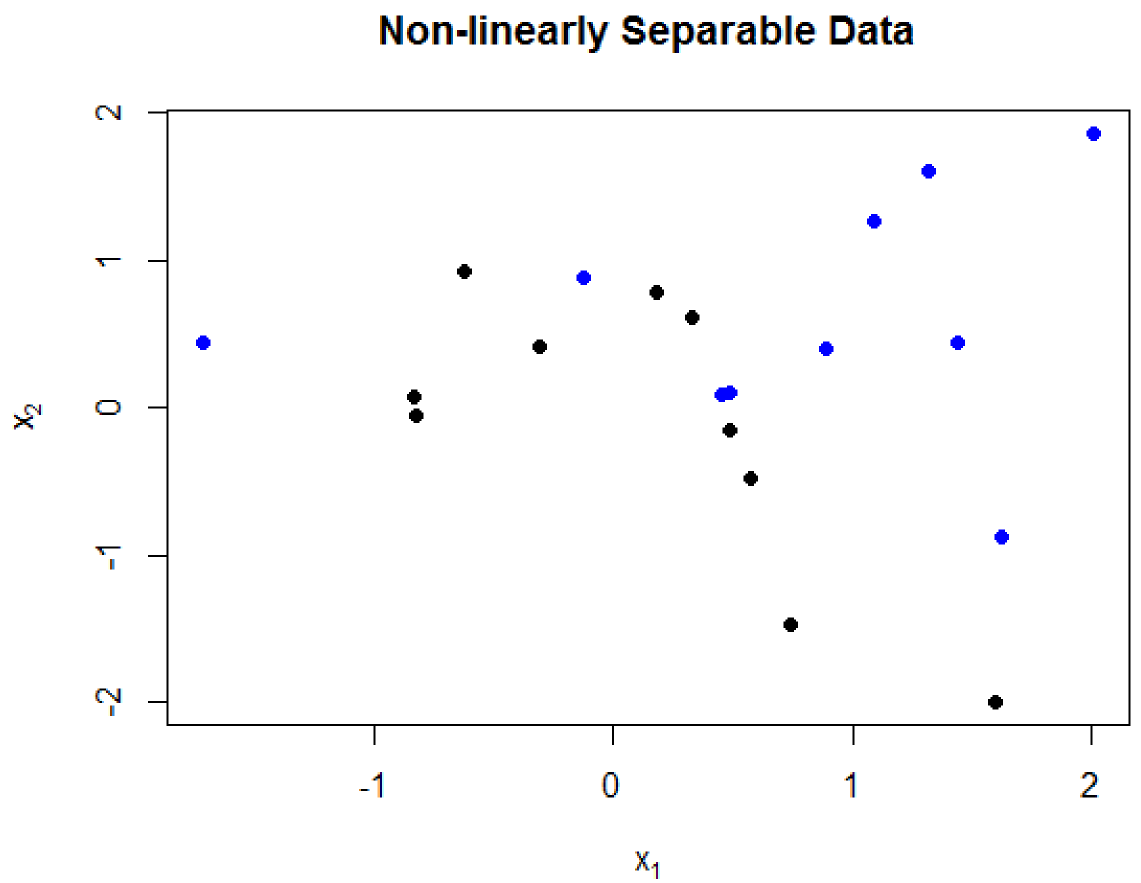

图 14.5 线性不可分的数据

对于线性不可分的数据，可以放松对于约束条件的要求，即只要求分离超平面将大多数观测值正确分离，而允许少量错误分类(或落入间隔之内)的观测值。

引入“松弛变量”(slack variable) $\xi_i \geq 0$，而将约束条件变为 $y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) \geq 1 - \xi_i$；但对所有观测值的松弛变量之和 $\sum_{i=1}^{n} \xi_i$ 进行惩罚。

此最小化问题可写为

$$
\begin{aligned}
&\min_{\boldsymbol{\beta},\,\beta_0,\,\xi_i} \ \frac{1}{2}\boldsymbol{\beta}'\boldsymbol{\beta} + C\sum_{i=1}^{n}\xi_i \\
&s.t.\ \ y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) \geq 1 - \xi_i,\ \ \xi_i \geq 0,\ \ \forall i
\end{aligned}
\tag{14.20}
$$

其中，$C \geq 0$为调节变量(C表示 Cost)，用来惩罚过大的松弛变量总和(太多错误)。

由于存在松弛变量$\xi_i \geq 0$，故允许$\mathbf{x}_i$落在间隔的错误一边(wrong side of the margin)，甚至分离超平面的错误一边(wrong side of the separating hyperplane)，因此称为**软间隔分类器**(soft margin classifier)，或**支持向量分类器**(support vector classifier)。

如果$0<\xi_i<1$，则$\mathbf{x}_i$落在间隔的错误一边(即间隔之内)，但依然在超平面的正确一边。

如果$\xi_i=1$，则$\mathbf{x}_i$正好落在超平面上。

如果$\xi_i>1$，则$\mathbf{x}_i$落在分离超平面的错误一边，参见图 14.6。

对于软间隔分类器，所有在间隔上、间隔内与分类错误的样本点，都是支持向量，因为它们都对最优解有影响。

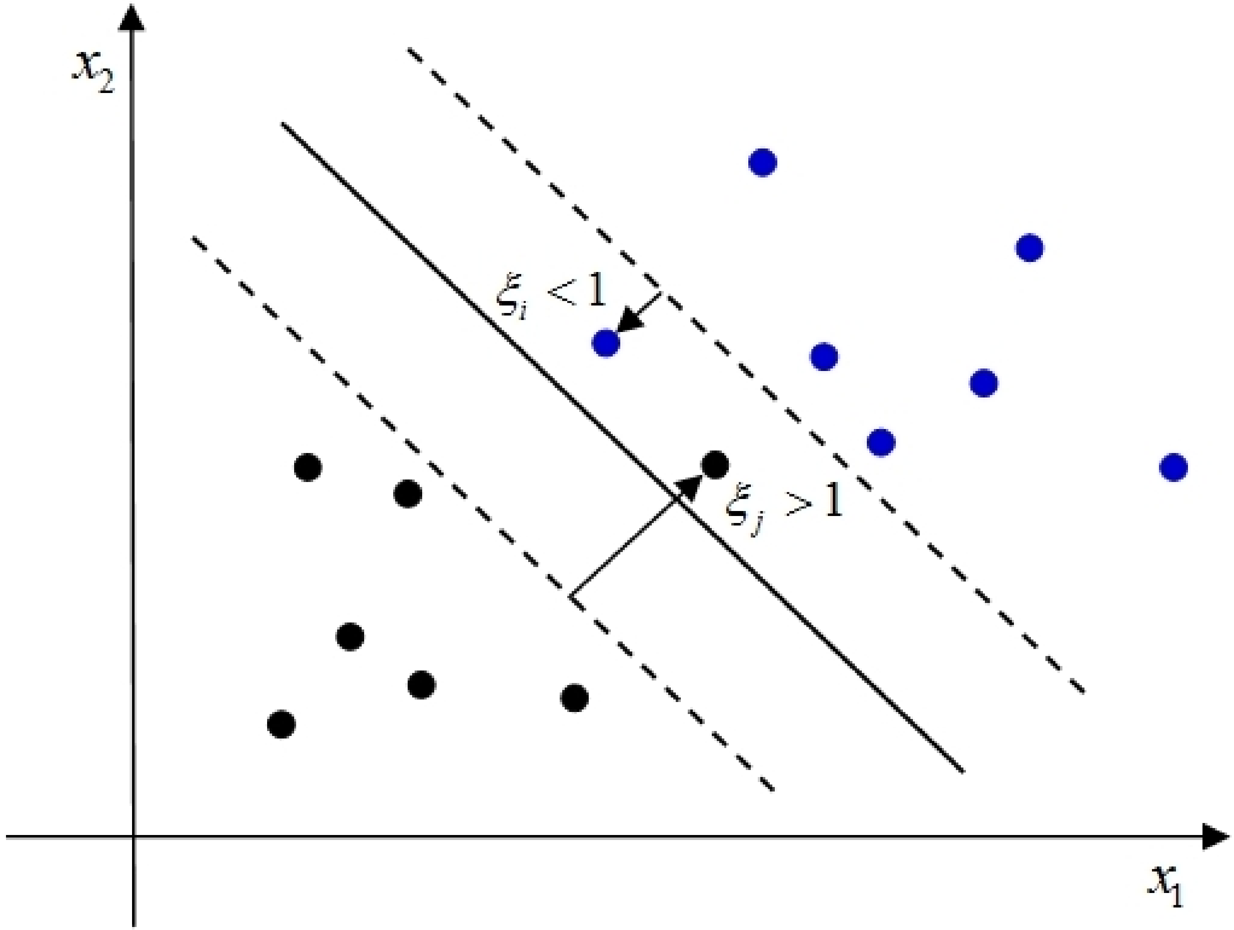

图 14.6 软间隔分类器的松弛变量

如果惩罚参数C为无穷大，则意味着算法不容忍训练样本中的任何分类错误。

这就是上节的最大间隔分类器，也称为**硬间隔分类器**(hard margin classifier)。

即使对于线性可分的数据，硬间隔分类器也可能不稳健，容易受到极端值(outlier)的影响，参见图 14.7。

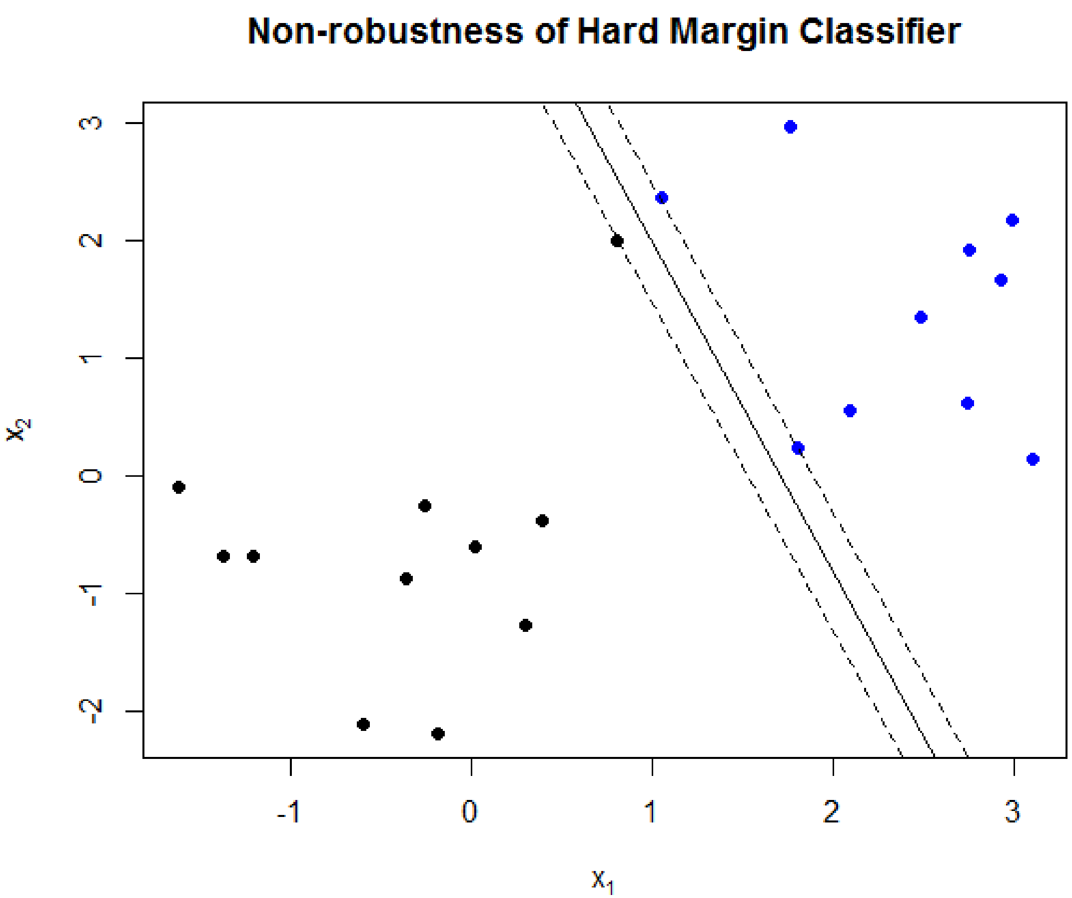

图 14.7 硬间隔分类器易受极端值影响

在图 14.7 中，最上方的黑点为极端值。此极端值对于分离超平面的位置有很大影响，而且导致最大间隔非常狭窄。此极端值很可能为“噪音”，而硬间隔分类器很好地拟合了噪音，导致过拟合，使得模型泛化能力下降。

即使对于线性可分的数据，一般也使用软间隔分类器，并将惩罚力度C视为调节参数，通过交叉验证来确定其最优值。

如果C很大，则对于犯错的惩罚力度很大，即几乎不允许犯错，故易导致过拟合(参见图 14.7)，而针对过拟合的正则化(regularization)程度较低。

反之，若C很小，则对于犯错的惩罚力度很小，故不易导致过拟合，而针对过拟合的正则化程度较高。因此，惩罚力度C与正则化程度呈反比。

对于软间隔分类器(14.20)的求解，依然可使用拉格朗日函数，但须引入松弛变量ξ_i的拉格朗日乘子μ_i：

$$\min_{\boldsymbol{\beta},\, \beta_0,\, \boldsymbol{\alpha},\, \boldsymbol{\mu}} \quad L_P(\boldsymbol{\beta}, \beta_0, \boldsymbol{\alpha}, \boldsymbol{\mu})$$

$$= \frac{1}{2}\boldsymbol{\beta}'\boldsymbol{\beta} + C\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n}\alpha_i\left[y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) - 1 + \xi_i\right] - \sum_{i=1}^{n}\mu_i\xi_i \tag{14.21}$$

此问题的求解方法与最优解的形式，均类似于硬间隔分类器。

14.4 软间隔分类器的统计解释

在使用软间隔分类器时，最简单的一种优化方法为，仅惩罚分类错误的观测值个数。

如果“$y_i f(\mathbf{x}_i) < 0$”，则分类错误。因此，目标函数可写为

$$\min_{\boldsymbol{\beta}, \beta_0} \ \frac{1}{2}\boldsymbol{\beta}'\boldsymbol{\beta} + C\sum_{i=1}^{n} I\Big(y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i) < 0\Big) \quad (14.22)$$

其中，$\lambda > 0$为调节变量，控制对分类错误的惩罚力度；$I(\cdot)$为示性函数。

定义裕度(margin)$z_i \equiv y_i(\beta_0 + \boldsymbol{\beta}'\mathbf{x}_i)$，则上式可通过 0-1 损失函数来表达：

$$\min_{\boldsymbol{\beta},\ \beta_0} \ \frac{1}{2}\boldsymbol{\beta}'\boldsymbol{\beta} + C\sum_{i=1}^{n} \ell_{0/1}(z_i) \qquad (14.23)$$

其中，$\ell_{0/1}(z_i)$为 0−1 损失函数，是裕度z_i的函数，其定义为

$$\ell_{0/1}(z_i) = \begin{cases} 0 & if \ z_i \geq 0 \\ 1 & if \ z_i < 0 \end{cases} \qquad (14.24)$$

其中，如果裕度z_i非负，则损失为 0；而如果裕度z_i为负，则损失为 1；参见图 14.8。

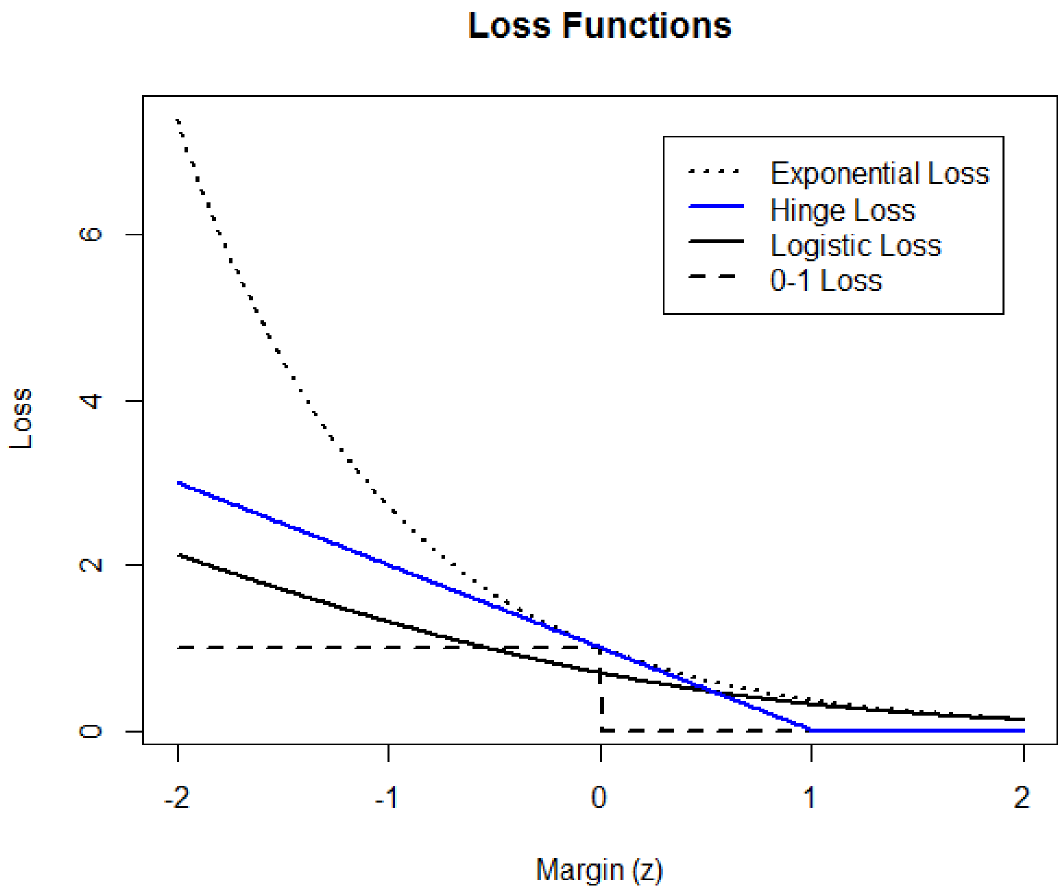

图 14.8 损失函数的比较

但 0-1 损失函数既不连续，也非凸函数，故不易进行最优化。

一般使用其他函数来替代 0-1 损失函数，称为**替代损失函数**(surrogate loss function)。

可以证明，支持向量机使用的替代损失函数为如下**合页损失函数**(hinge loss function)：

$$\ell_{hinge}(z_i) = \begin{cases} 0 & if \;\; z_i \geq 1 \\ 1 - z_i & if \;\; z_i < 1 \end{cases} \qquad (14.25)$$

其中，如果裕度$z_i \geq 1$，则损失为 0(不惩罚)；反之，如果$z_i < 1$，则其损失为$(1 - z_i)$(斜率为-1，故惩罚力度为 1 对 1)。

由于此函数的形状类似于门框的合页，故称为“合页损失函数”。

更简洁地，可将分段函数(14.25)写为统一的表达式：

$$\ell_{hinge}(z_i) = \max(0, 1 - z_i) \qquad (14.26)$$

将合页损失函数代入目标函数，并记松弛变量$\xi_i = 1 - z_i$，可得

$$
\begin{aligned}
\min_{\boldsymbol{\beta},\, \beta_0} \quad & \frac{1}{2}\boldsymbol{\beta}'\boldsymbol{\beta} + C\sum_{i=1}^{n} \ell_{hinge}(z_i) \\
&= \frac{1}{2}\boldsymbol{\beta}'\boldsymbol{\beta} + C\sum_{i=1}^{n} \max(0, 1-z_i) \\
&= \frac{1}{2}\boldsymbol{\beta}'\boldsymbol{\beta} + C\sum_{i=1}^{n} (1-z_i) I(1-z_i \geq 0) \\
&= \frac{1}{2}\boldsymbol{\beta}'\boldsymbol{\beta} + C\sum_{i=1}^{n} \xi_i I(\xi_i \geq 0)
\end{aligned}
\tag{14.27}
$$

上式与软间隔分类器的最优化问题(14.20)等价。

尽管合页损失函数依然不光滑(在$z_i = 1$处有尖点)，但至少是连续的凸函数，其数学性质优于 0-1 损失函数。

光滑的替代损失函数则包括“指数损失函数” (用于 AdaBoost 算法)，以及“逻辑损失函数” (用于二分类问题的梯度提升法)。

支持向量机的合页损失函数，与逻辑回归的逻辑损失函数最为接近。

14.5 支持向量机

在数据线性不可分的情况下，一般存在非线性的决策边界(nonlinear decision boundary)，参见图 14.9。

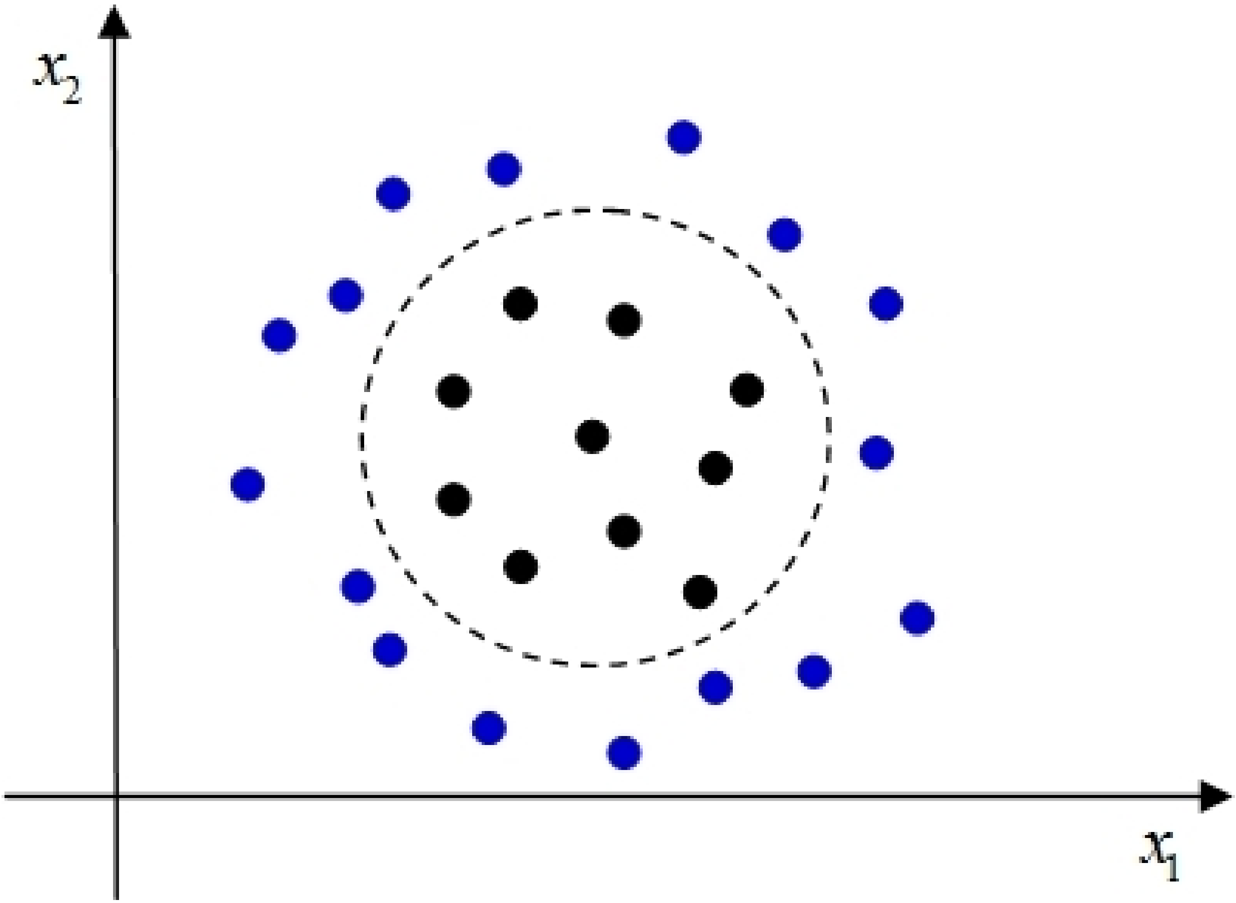

图 14.9 非线性决策边界

在图 14.9 中，不存在任何线性的分离超平面，但存在一个近乎圆形的决策边界。

可做一个极坐标变换，将特征向量(x_1, x_2)变换为(r, θ)：

$$\begin{cases} x_1 - \bar{x}_1 = r\cos\theta \\ x_2 - \bar{x}_2 = r\sin\theta \end{cases} \tag{14.28}$$

其中，$\bar{x}_1$与$\bar{x}_2$分别为x_1与x_2的样本均值；而半径$r = \sqrt{(x_1 - \bar{x}_1)^2 + (x_2 - \bar{x}_2)^2}$。

在变换后的(r,θ)特征空间，两类数据变为线性可分。

因为蓝色数据的半径(r)显然更大，而红色数据的半径更小，故只要选择形如“$r \geq t$”的超平面即可进行分离。

更一般地，对于决策边界非线性的数据，考虑对特征向量$\mathbf{x}_i$进行变换，比如将$\mathbf{x}_i$变换为$\boldsymbol{\varphi}(\mathbf{x}_i)$；其中，$\boldsymbol{\varphi}(\mathbf{x}_i)$为多维函数(维度可以高于$\mathbf{x}_i$)，甚至无限维函数。

这意味着，将训练样本$\left\{\mathbf{x}_i, y_i\right\}_{i=1}^n$变换为$\left\{\boldsymbol{\varphi}(\mathbf{x}_i), y_i\right\}_{i=1}^n$。

目的是希望在$\boldsymbol{\varphi}(\mathbf{x}_i)$的特征空间(feature space)中，可以得到线性可分的情形，参见图 14.10。

难点在于，对于高维数据，一般并不知道变换$\boldsymbol{\varphi}(\cdot)$的具体形式。

例 在桌上叠放一张黑纸与白纸。黑纸上的黑点属于一类，而白纸上的白点属于另一类。显然，这两类点可用超平面分离。现将这两张纸揉成一团，则无法再用超平面分离黑点与白点。然而，若将这两张纸再摊平捋顺(特征变换$\boldsymbol{\varphi}(\cdot)$)，则又可用超平面分离黑点与白点。

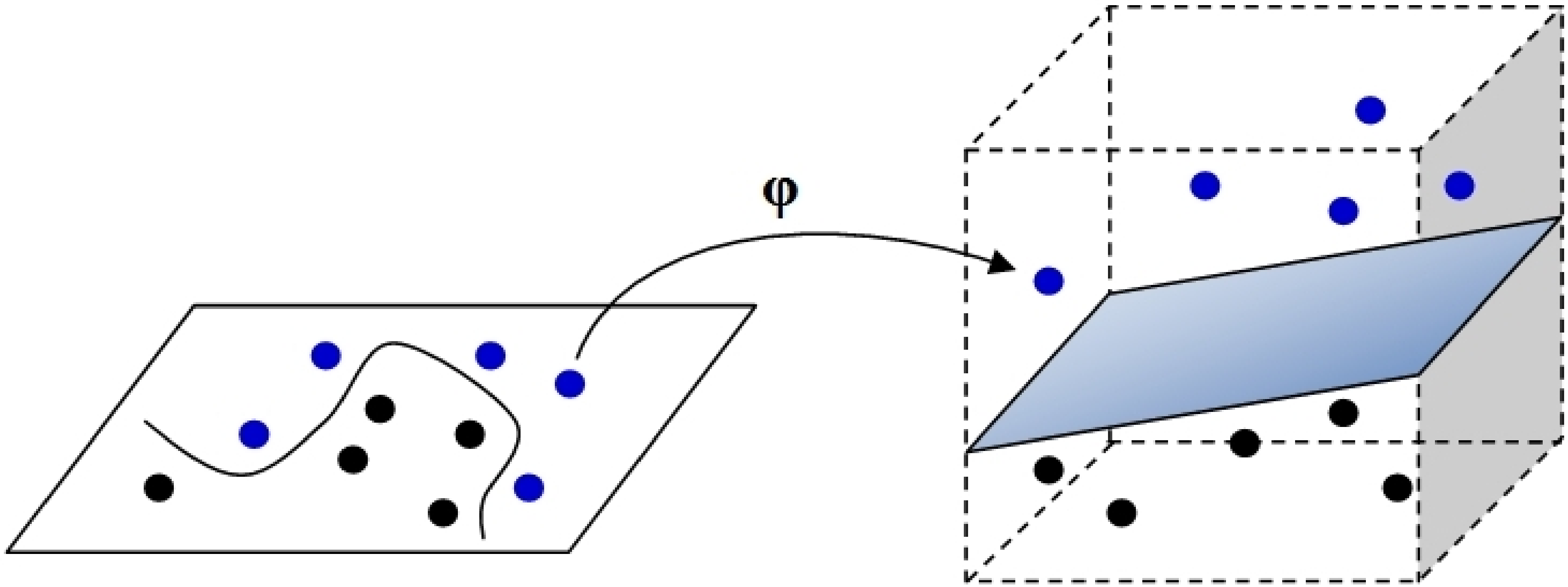

图 14.10 特征变换

根据上述推导，支持向量机的估计结果仅依赖于$\left\langle \boldsymbol{\varphi}(\mathbf{x}_i), \boldsymbol{\varphi}(\mathbf{x}_j) \right\rangle$，即$\boldsymbol{\varphi}(\mathbf{x}_i)$与$\boldsymbol{\varphi}(\mathbf{x}_j)$的内积，而不必知道$\boldsymbol{\varphi}(\cdot)$。

为此，将此内积定义为如下**核函数**(kernel function)：

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) \equiv \left\langle \boldsymbol{\varphi}(\mathbf{x}_i), \boldsymbol{\varphi}(\mathbf{x}_j) \right\rangle = \boldsymbol{\varphi}(\mathbf{x}_i)' \boldsymbol{\varphi}(\mathbf{x}_j) \tag{14.29}$$

只要直接指定核函数$\kappa(\mathbf{x}_i, \mathbf{x}_j)$的具体形式即可；而无须预先知道$\boldsymbol{\varphi}(\cdot)$，再计算$\left\langle \boldsymbol{\varphi}(\mathbf{x}_i), \boldsymbol{\varphi}(\mathbf{x}_j) \right\rangle$(此内积可能不易计算，尤其在高维空间中)。

这种方法称为**核技巧**(kernel trick)。

$\kappa(\mathbf{x}_i, \mathbf{x}_j)$必须关于$\mathbf{x}_i$与$\mathbf{x}_j$是对称的，即$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \kappa(\mathbf{x}_j, \mathbf{x}_i)$。

常用的核函数包括：

(1) 多项式核(polynomial kernel)：

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = (1 + \gamma \mathbf{x}_i' \mathbf{x}_j)^d \qquad (14.30)$$

其中，$d \geq 1$为多项式的次数，而$\gamma > 0$为参数。如果$d = 1$，则为“线性核函数”(linear kernel)，即不作变换，可直接令$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i' \mathbf{x}_j$。

(2) 径向核(radial kernel)，也称为“径向基函数”(Radial Basis Function，简记 RBF)：

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\gamma \left\|\mathbf{x}_i - \mathbf{x}_j\right\|^2\right), \quad \gamma > 0 \qquad (14.31)$$

其中，$\|\cdot\|$为 2-范数，即欧氏距离；而$\gamma > 0$为径向核的参数。

径向核非常接近于正态分布的密度函数，故也称为“高斯核”(Gaussian kernel)。

参数γ控制着一个样本点的影响范围。如果γ很大，则一个样本点的影响范围很小，可能导致过拟合。反之，若γ很小，则一个样本点的影响范围很大，可能导致欠拟合。

(3) 拉普拉斯核(Laplacian kernel)：

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\gamma \left\| \mathbf{x}_i - \mathbf{x}_j \right\|\right), \quad \gamma > 0 \qquad (14.32)$$

拉普拉斯核类似于径向核，但在指数$\exp(\cdot)$中直接使用欧氏距离，而非欧氏距离的平方。

(4) S 型核(Sigmoid kernel)：

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \tanh\left(\beta \mathbf{x}_i' \mathbf{x}_j + \theta\right), \quad \beta > 0,\ \theta < 0 \tag{14.33}$$

其中，$\tanh(\cdot)$ 为“双曲正切函数”(hyperbolic tangent)，定义为

$$\tanh(z) \equiv \frac{e^z - e^{-z}}{e^z + e^{-z}}。$$

在神经网络中，双曲正切函数也作为“激活函数”(activation function)使用，参见第 15 章。

在最大间隔分类器最优解 $\hat{f}(\mathbf{x})$ 的表达式(14.19)中，以变换之后的 $\boldsymbol{\varphi}(\mathbf{x})$ 替代 $\mathbf{x}$ 可得：

$$\begin{aligned}\hat{f}(\mathbf{x}) &= \hat{\beta}_0 + \hat{\boldsymbol{\beta}}'\boldsymbol{\varphi}(\mathbf{x}) = \hat{\beta}_0 + \left(\sum_{i=1}^{n} \hat{\alpha}_i y_i \boldsymbol{\varphi}(\mathbf{x}_i)\right)' \boldsymbol{\varphi}(\mathbf{x}) \\ &= \hat{\beta}_0 + \left(\sum_{i=1}^{n} \hat{\alpha}_i y_i \boldsymbol{\varphi}(\mathbf{x}_i)' \boldsymbol{\varphi}(\mathbf{x})\right) = \hat{\beta}_0 + \left(\sum_{i=1}^{n} \hat{\alpha}_i y_i \kappa(\mathbf{x}_i, \mathbf{x})\right)\end{aligned} \tag{14.34}$$

这意味着，支持向量机的最优解 $\hat{f}(\mathbf{x})$ 可通过训练样本的核函数 $\kappa(\mathbf{x}_i, \mathbf{x})$ 展开，称为“支持向量展开”(support vector expansion)。

更一般地，“核技巧” (kernel trick)的应用也不局限于支持向量机。

后来发展出一系列基于核函数的学习方法，统称为“核方法” (kernel methods)。

比如，先作特征变换$\mathbf{x}_i \to \boldsymbol{\varphi}(\mathbf{x}_i)$，然后在$\boldsymbol{\varphi}(\mathbf{x}_i)$的特征空间进行线性判别分析，即为“核线性判别分析”(Kernelized Linear Discriminant Analysis，简记 KLDA)。

14.6 多分类问题的支持向量机

由于支持向量机使用超平面分离不同类别的数据，故并没有推广到多分类问题的自然方法。

对于多分类问题，通常的解决方法为**1 对 1 分类**(one-to-one classification)，也称为**全配对法**(all-pairs approach)。

假设数据共分为K类，其中$K > 2$。

从这K类数据中，取出两类(不考虑排序)，则可能的取法组合数目为

$$\binom{K}{2} = C_K^2 = \frac{K(K-1)}{2} \qquad (14.35)$$

对这C_K^2个二分类问题均使用支持向量机，可得到C_K^2个 SVM 模型。

对于一个新的“测试观测值”(test observation)，使用这C_K^2个 SVM 模型进行预测，然后以最常见的预测类别作为最终的预测结果。

这也是一种两两对决(PK)，最终以多数票规则加总的方法。

14.7 支持向量回归

支持向量机最初仅用于分类问题，但后来也推广到回归问题，即所谓**支持向量回归**(Support Vector Regression，简记 SVR)。

SVR 的基本思想是，将支持向量机的合页损失函数移植到回归问题。

记回归函数(超平面)为 $f(\mathbf{x}) = \beta_0 + \mathbf{x}'\boldsymbol{\beta}$，并以此函数预测连续型响应变量 y。

SVR 的目标函数为

$$\min_{\boldsymbol{\beta},\, \beta_0} \ \frac{1}{2}\boldsymbol{\beta}'\boldsymbol{\beta} + C\sum_{i=1}^{n} \ell_{\varepsilon}\big(y_i - f(\mathbf{x}_i)\big) \tag{14.36}$$

其中，$C \geq 0$为正则化参数(regularization parameter)，$z_i \equiv y_i - f(\mathbf{x}_i)$为残差(residual)；而$\ell_{\varepsilon}(\cdot)$为$\varepsilon$-不敏感损失函数($\varepsilon$-insensitive loss function)，其定义为

$$\ell_{\varepsilon}(z_i) = \begin{cases} 0 & if \ |z_i| \leq \varepsilon \\ |z_i| - \varepsilon & if \ |z_i| > \varepsilon \end{cases} \tag{14.37}$$

其中，$\varepsilon > 0$也是调节参数。

如果残差$z_i \equiv y_i - f(\mathbf{x}_i)$的绝对值小于或等于$\varepsilon$，则损失为 0。

故在一个宽度为2ε的间隔带中，损失函数对残差不敏感，故名“ε-不敏感损失函数”。

反之，如果残差z_i的绝对值大于ε，则损失为$|z_i| - \varepsilon$，呈 1 对 1 的线性增长，参见图 14.11。

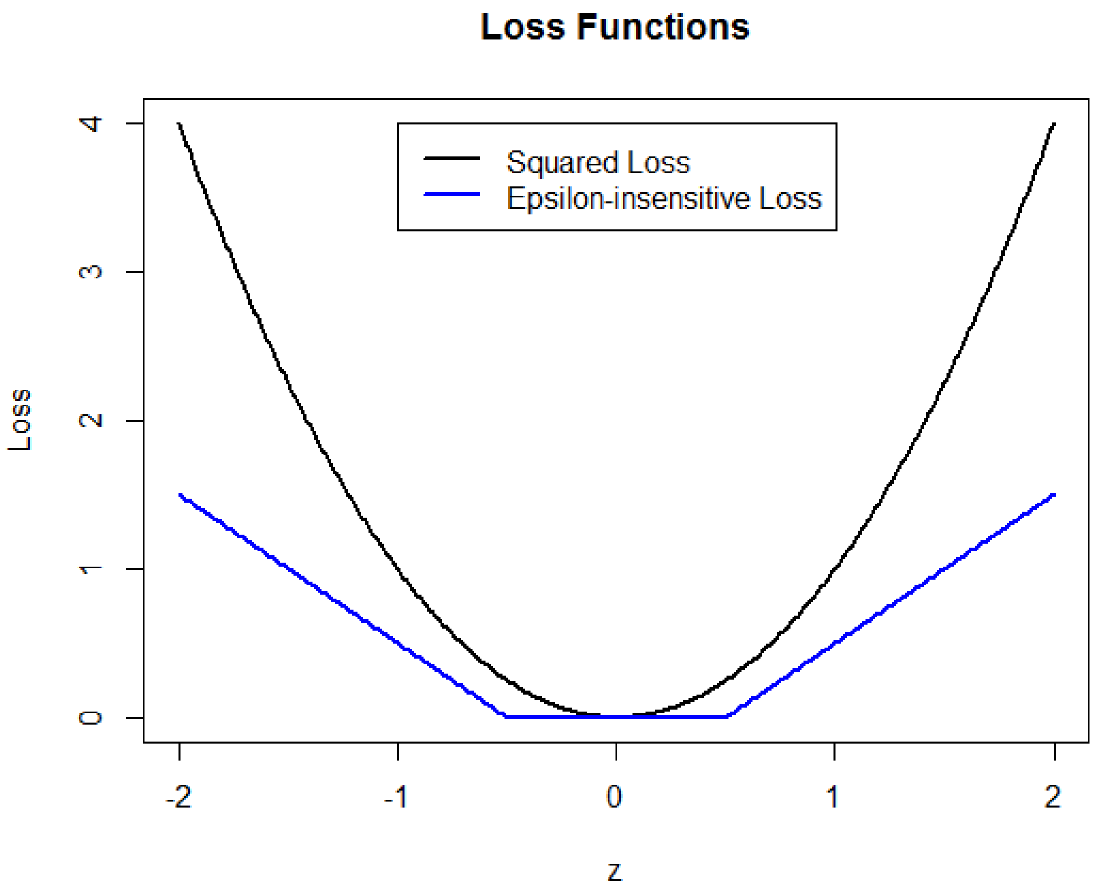

图 14.11 ε-不敏感损失函数

与平方损失函数相比，ε-不敏感损失函数存在一个半径为ε的不敏感带，其损失函数为 0。出了此敏感带后，ε-不敏感损失函数呈线性增长，其增长速度不及平方损失的二次函数。

如果将$\dfrac{1}{2}\boldsymbol{\beta}'\boldsymbol{\beta}$视为惩罚项，则 SVR 的目标函数类似于岭回归：

$$\min_{\boldsymbol{\beta},\ \beta_0} \; C\sum_{i=1}^{n}\ell_{\varepsilon}\big(y_i - f(\mathbf{x}_i)\big) + \frac{1}{2}\boldsymbol{\beta}'\boldsymbol{\beta} \qquad (14.38)$$

上式与岭回归的区别在于，岭回归以平方损失函数替代ε-不敏感损失函数$\ell_{\varepsilon}(\cdot)$。由于使用$\varepsilon$-不敏感损失函数，这种支持向量回归也称为$\varepsilon$−回归($\varepsilon$-regression)。

由于 SVR 使用了类似于支持向量机的合页损失函数($\ell_{\varepsilon}(\cdot)$可视为两片对称合页的组合)，故 SVR 比较适用于变量较多的数据(正如 SVM)。

SVR 也可使用“核技巧”(kernel trick)，即将特征向量$\mathbf{x}_i$变换为$\boldsymbol{\varphi}(\mathbf{x}_i)$，由此得到非线性回归的结果。

ε-不敏感损失函数为线性函数(出了间隔带之后)，故对于极端值不敏感，比较稳健。这些是使用支持向量回归的主要理由。

对于 SVR 问题(14.36)的求解，依然可引入松弛变量，通过拉格朗日函数函数求解。

14.8 支持向量机的优缺点

支持向量机的最大优点是，它比较适用于变量很多的数据(high dimensionality)。

当特征向量的维度 p 很大时，数据被“打散”，使得散布于 p 维空间的样本点比较容易用超平面进行分离。

这使得 SVM 在文本分析方面得到广泛应用，比如“文件分类”(document classification)与“情感分析”(sentiment analysis)。

由于语言中的词汇很多，故文本数据通常有很丰富的特征变量。在 2012 年之前，SVM 在图像识别领域是领先算法；但之后被卷积神经网络超越。

其次，SVM 在数据存储方面较有效率(memory efficient)，这是因为在进行预测时，SVM 仅需使用一部分数据(即支持向量)即可。

另外，由于可使用核技巧，使得 SVM 具有通用性(versatility)，适合于高度非线性的决策边界。

支持向量机的缺点则包括，有时它对于核函数中的参数比较敏感。

对于真正的高维数据($p > n$，即变量个数大于样本容量)，SVM 可能表现较差。此时，特征空间的维度远超样本容量，故只有相对较少的支持向量来决定更高维度的分离超平面，这使得模型的泛化能力变差。

由于 SVM 使用分离超平面进行分类，故无法从概率的角度进行解释。

14.9 支持向量机的 Python 案例：模拟数据

我们首先使用模拟数据演示支持向量机的操作。

使用模拟数据的好处在于，我们知道真实的数据生成过程，可在低维空间直观展示 SVM 模型，并可任意生成测试集，不受样本容量的限制。、

首先，导入本章所需模块：

```
In [1]: import numpy as np
   ...: import pandas as pd
   ...: import matplotlib.pyplot as plt
   ...: import seaborn as sns
```

```
...: from sklearn.preprocessing import
         StandardScaler
...: from sklearn.model_selection import
         train_test_split
...: from sklearn.model_selection import KFold,
         StratifiedKFold
...: from sklearn.model_selection import
         GridSearchCV
...: from sklearn.metrics import
         plot_confusion_matrix
...: from sklearn.svm import SVC
...: from sklearn.svm import SVR
...: from sklearn.svm import LinearSVC
```

```
...: from sklearn.datasets import load_boston
...: from sklearn.datasets import load_digits
...: from sklearn.datasets import make_blobs
...: from mlxtend.plotting import
     plot_decision_regions
```

其次，使用 sklearn 模块的 make_blobs()函数(“blob”意为一点、一滴或一团)，生成模拟数据：

```
In [2]: X, y = make_blobs(n_samples=40, centers=2,
                  n_features=2, random_state=6)
```

其中，参数“n_samples=40”指定样本容量为 40；

参数“centers=2”指定数据共有两个中心(中心位置不同的两个正态分布)；

参数“n_features=2”表示共有 2 个特征变量。

在返回的结果中，数据矩阵 X 包含 2 个特征变量，而响应变量 y 取值为 0 或 1(对应于两类数据)。

根据支持向量机的惯例，将响应变量的取值$\{0,1\}$变换为$\{-1,1\}$，这可通过函数变换$(2y-1)$来实现(因为$2\cdot 0-1=-1$，而$2\cdot 1-1=1$)：

```
In [3]: y = 2 * y - 1
```

为画图方便，将数据矩阵 X 设为数据框，并将两个特征变量命名为 x1 与 x2：

```
In [4]: data = pd.DataFrame(X, columns=['x1', 'x2'])
```

画模拟数据的散点图：

```
In [5]: sns.scatterplot(x='x1', y='x2', data=data,
                    hue=y, palette=['blue', 'black'])
```

其中，参数“hue=y”表示，根据响应变量 y 的取值上色，而参数“palette=['blue', 'black']”将调色板设为蓝色与黑色，结果参见图 14.12。

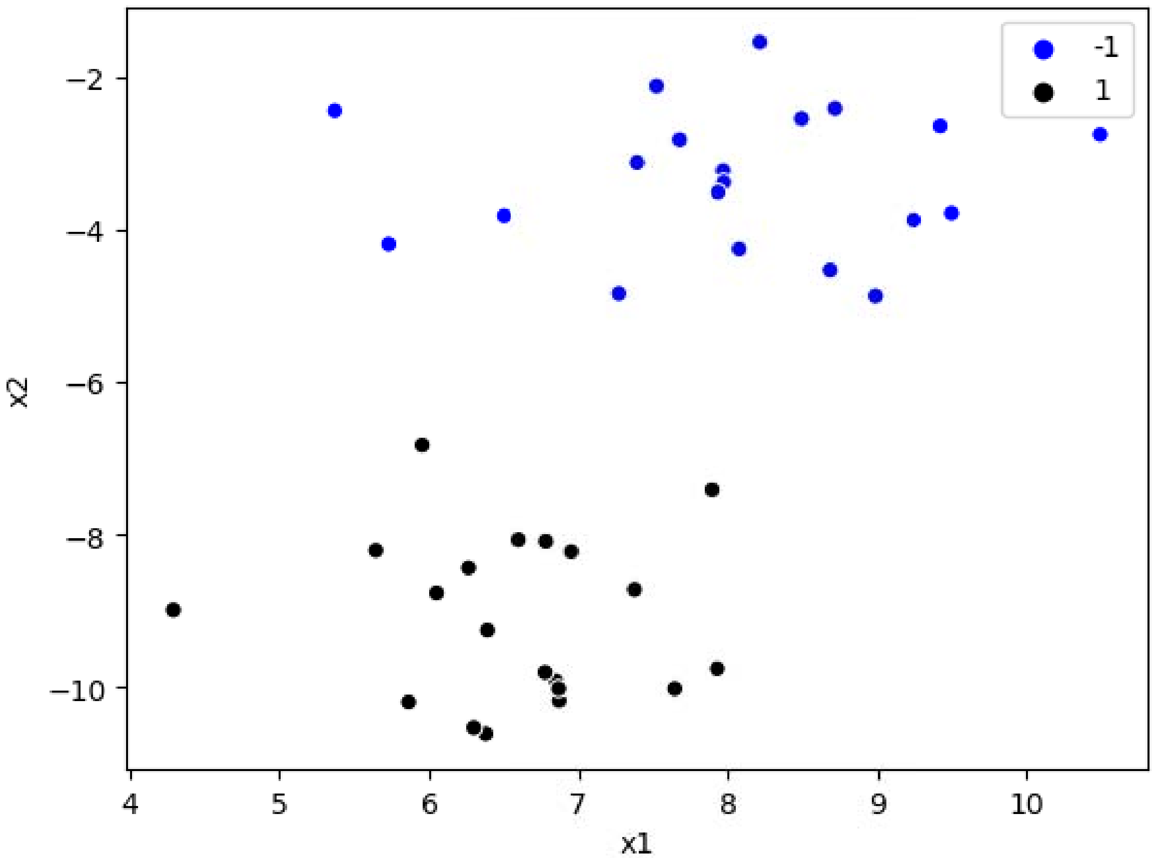

图 14.12 模拟数据的散点图

从图 14.12 可见，两类数据为线性可分。

下面，使用线性的支持向量分类器(Support Vector Classifier)进行分类，这可通过 sklearn 的 `LinearSVC` 类来实现：

```
In [6]: model = LinearSVC(C=1000, loss='hinge',
random_state=123)
```

其中，参数“`C=1000`”设定惩罚力度为$C=1000$(默认 `C=1`)。

由于这是线性可分数据，故为演示目的，设定一个很大惩罚参数C，即几乎不允许犯错误。

参数“loss='hinge'”表示使用合页损失函数,即标准的支持向量机；默认“loss='squared_hinge'”，以合页损失的平方作为损失函数。

此命令创建了 LinearSVC 的一个实例 model。使用 fit()方法进行估计：

```
In [7]: model.fit(X, y)
Out[7]:
LinearSVC(C=1000, class_weight=None, dual=True,
          fit_intercept=True, intercept_scaling=1,
          loss='hinge', max_iter=1000,
          multi_class='ovr', penalty='l2',
          random_state=123, tol=0.0001,verbose=0)
```

使用 model 的 get_params()方法，可得该模型的所有参数(以字典形式呈现)：

```
In [8]: model.get_params()
Out[8]:
{'C': 1000,
 'class_weight': None,
 'dual': True,
 'fit_intercept': True,
 'intercept_scaling': 1,
 'loss': 'hinge',
 'max_iter': 1000,
 'multi_class': 'ovr',
```

```
'penalty': 'l2',
'random_state': 123,
'tol': 0.0001,
'verbose': 0}
```

使用 `model` 的 `decision_function()` 方法，可计算观测值到分离超平面的“符号距离”(signed distance)，即上文的 $f(\mathbf{x}_i)=\beta_0+\boldsymbol{\beta}'\mathbf{x}_i$：

```
In [9]: dist = model.decision_function(X)
   ...: dist
Out[9]:
array([ 6.70285875, -2.26465076,  6.70517693, -4.07655804,  4.64851151,
       -3.54941209, -2.91067542, -4.9731406 , -1.        , -4.52544248,
        3.22779397,  3.00847884, -6.96459463,  5.3228811 ,  4.31545722,
       -7.69770417, -1.71993691,  6.6787946 , -6.75976738, -3.41656848,
        3.16409965,  2.20412747,  5.63590245,  3.98659281, -4.33205347,
       -4.7564515 ,  4.01887803,  6.68354561, -3.29486083,  3.00622782,
        0.99985517,  4.63308498,  5.24719306, -6.34222035, -5.94350419,
       -5.34793345, -5.51536708,  4.92321219, -4.11978593,  5.43329239])
```

为了判断某观测值是否为支持向量(support vector)，须考察条件“$y_i f(\mathbf{x}_i) \le 1$”是否满足，其中$y_i$取值为-1 或 1。

使用 Numpy 的 `where()`函数，可找出支持向量的位置索引：

```
In [10]: index = np.where(y * dist <= (1 + 1e-10))
    ...: index
Out[10]: (array([ 8, 30], dtype=int64),)
```

其中，只有 1 个参数的 `np.where()`函数，返回满足不等式条件“ `y * dist <= (1 + 1e-10)`”的位置索引。此不等式的右边为“$1+10^{-10}$”(比 1 略大)，以防止数值计算的偏差(而无法识别支持向量)。

结果显示，第 8 与 30 个观测值为支持向量。将此位置索引代回数据矩阵 X，可得支持向量：

```
In [11]: X[index]
Out[11]:
array([[ 5.73005848, -4.19481136],
       [ 7.89359985, -7.41655113]])
```

为便于后续调用，我们定义一个计算支持向量的函数 support_vectors()：

```
In [12]: def support_vectors(model, X, y):
    ...:     dist = model.decision_function(X)
    ...:     index = np.where(y * dist <= (1 + 1e-10))
    ...:     return X[index]
```

其中，函数 support_vectors()接受 3 个参数，即已估计的支持向量机模型 model，数据矩阵 X 与响应变量 y，并返回支持向量。

尝试调用此函数：

```
In [13]: support_vectors(model, X, y)
Out[13]:
array([[ 5.73005848, -4.19481136],
       [ 7.89359985, -7.41655113]])
```

更直观地，基于二维的特征空间，可将支持向量画图展示。

为方便后续调用，将画图过程封装为一个自定义函数 svm_plot()：

```
In [14]: def svm_plot(model, X, y):
    ...:     data = pd.DataFrame(X,columns=['x1','x2'])
    ...:     data['y'] = y
    ...:     sns.scatterplot(x='x1', y='x2', data=data,
                  s=30, hue=y, palette=['blue', 'black'])
    ...:     s_vectors = support_vectors(model, X, y)
    ...:     plt.scatter(s_vectors[:, 0],
                        s_vectors[:, 1], s=100, linewidth=1,
                        facecolors='none', edgecolors='k')
    ...:     ax = plt.gca()
    ...:     xlim = ax.get_xlim()
```

```
...:     ylim = ax.get_ylim()
...:     xx, yy = np.meshgrid(np.linspace(xlim[0],
    xlim[1], 50), np.linspace(ylim[0], ylim[1], 50))
...:     Z = model.decision_function(
                    np.c_[xx.ravel(),yy.ravel()])
...:     Z = Z.reshape(xx.shape)
...:     plt.contour(xx, yy, Z, colors='k',
                    levels=[-1, 0, 1], alpha=0.5,
                    linestyles=['--', '-', '--'])
...:     C = model.get_params()['C']
...:     plt.title(f'SVM (C = {C})')
```

在以上 `def` 语句的函数体(function body)中，前 3 个命令画散点图，其中参数“`s=30`”设定散点的大小为 30。

第 4-5 个命令调用 `support_vectors()`函数计算支持向量，并画支持向量的散点图；其中，参数“`s=100`”设定散点的大小为 100(更大的圆圈)，参数“`facecolors='none'`”表示无填充色(故为空心圆圈)，参数“`edgecolors='k'`”表示圆圈的边界为黑色，而参数“`linewidth=1`”设定圆圈边界的线宽为 1。

命令“`ax = plt.gca()`”获取当前画轴(get current axis)，并记为 `ax`；而命令“`xlim = ax.get_xlim()`”与“`ylim = ax.get_ylim()`”则获取当前画轴的 `x` 取值范围 `xlim`(由其最小值与最大值所构成的元组)与 `y` 取值范围 `ylim`。

接着，使用 `np.meshgrid()`函数在 x 与 y 的取值范围，生成50×50的网格，并记此网格的横坐标与纵坐标分别为 xx 与 yy(二者均为50×50的矩阵)。

然后，使用 `decision_function()`方法，计算每个观测值 xx 与 yy 的符号距离 Z，并以此作为网格(xx， yy)的第三维度，即高度。

其中，“`xx.ravel()`”将横坐标的矩阵 xx 展开为向量，“`yy.ravel()`”将纵坐标的矩阵 yy 展开为向量，而“`np.c_[xx.ravel(), yy.ravel()]`”则将二者作为列向量并列(“c”表示 concatenate，即并列)，变成数据矩阵的形式。

然而，作为 `decision_function()`方法的返回值，`Z` 依然是一个向量，故使用命令“`Z = Z.reshape(xx.shape)`”将其变为与 `xx` 一样的形状(50×50的矩阵)。

为了画分离超平面与间隔，使用 Matplotlib 的 `contour()`函数，以 `xx` 为横坐标，`yy` 为纵坐标，而 `Z` 为高度坐标，画三个水平(即高度 `Z` 等于-1，0 与 1)的等值线(即等高线)。

在最后 2 个命令，通过 `get_params()`方法，获得 `model` 的惩罚参数 `C`，并以“`f`-字符串”添加标题。调用 `svm_plot()`函数，即可展示支持向量，结果参见图 14.13：

```
In [15]: svm_plot(model, X, y)
```

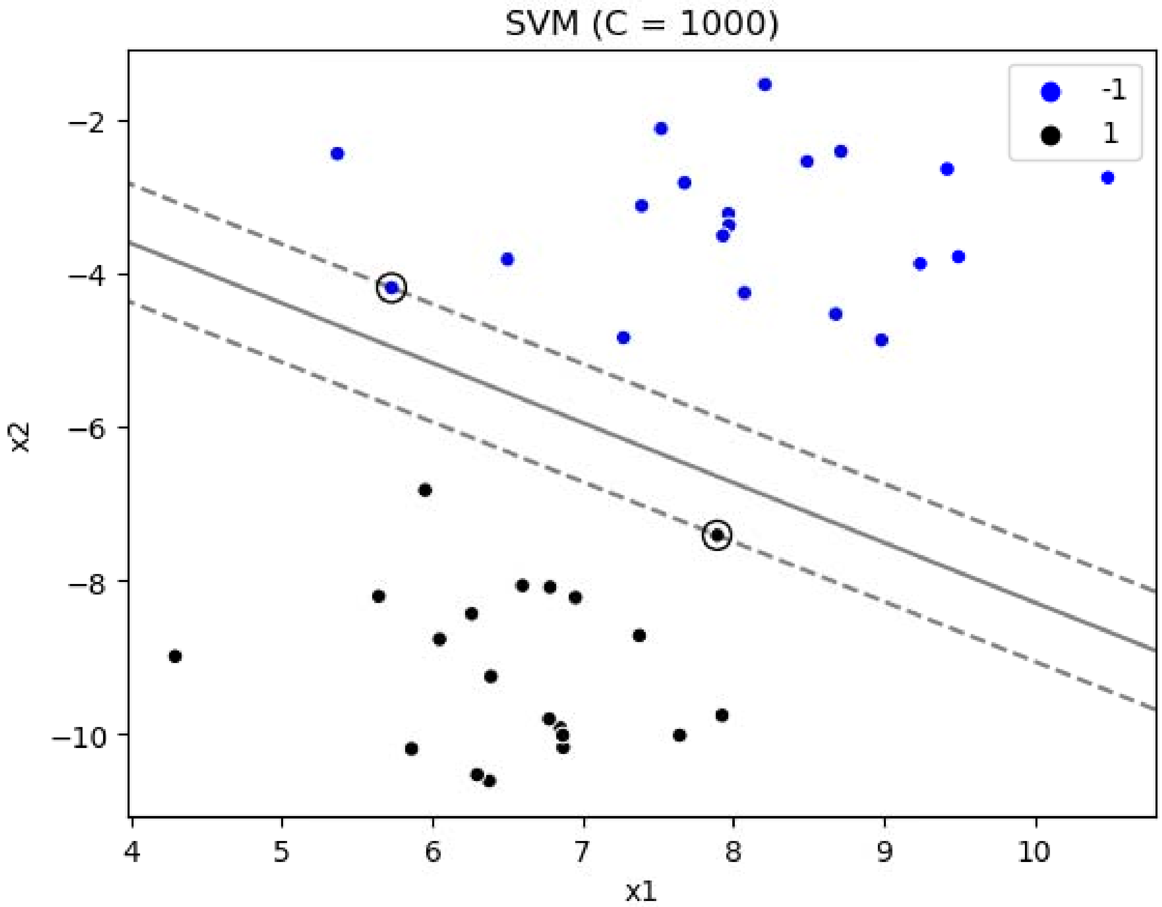

图 14.13 SVM 的估计结果(C=1000)

从图 14.13 可见，由于惩罚力度很大($C=1000$)，故间隔(margin)比较狭窄，所有观测值均正确分类。

除了两个支持向量在间隔之上，其余样本点均在间隔之外，故没有犯任何错误($\sum_{i=1}^{n}\xi_i=0$)。

下面，尝试使用更小的惩罚参数$C=0.1$，再次进行 SVM 估计：

```
In [16]: model = LinearSVC(C=0.1,
                            loss="hinge",random_state=123,
                            max_iter=1e4)
```

其中，参数“C=0.1”将惩罚参数 C 设为 0.1(sklearn 要求 C 严格大于 0)；参数“max_iter=1e4”将最大迭代次数增加到 10000(默认 1000 次，但不收敛)。

使用 fit()方法进行估计：

```
In [17]: model.fit(X, y)
Out[17]:
LinearSVC(C=0.1, class_weight=None, dual=True,
          fit_intercept=True, intercept_scaling=1,
          loss='hinge', max_iter=10000.0,
          multi_class='ovr', penalty='l2',
          random_state=123, tol=0.0001,verbose=0)
```

使用自定义的 support_vectors()函数，考察支持向量：

```
In [18]: support_vectors(model, X, y)
Out[18]:
array([[ 6.50072722, -3.82403586],
       [ 5.37042238, -2.44715237],
       [ 5.73005848, -4.19481136],
       [ 6.95292352, -8.22624269],
       [ 7.27059007, -4.84225716],
       [ 5.95313618, -6.82945967],
       [ 7.89359985, -7.41655113]])
```

画图展示支持向量，结果参见图 14.14：

```
In [19]: svm_plot(model, X, y)
```

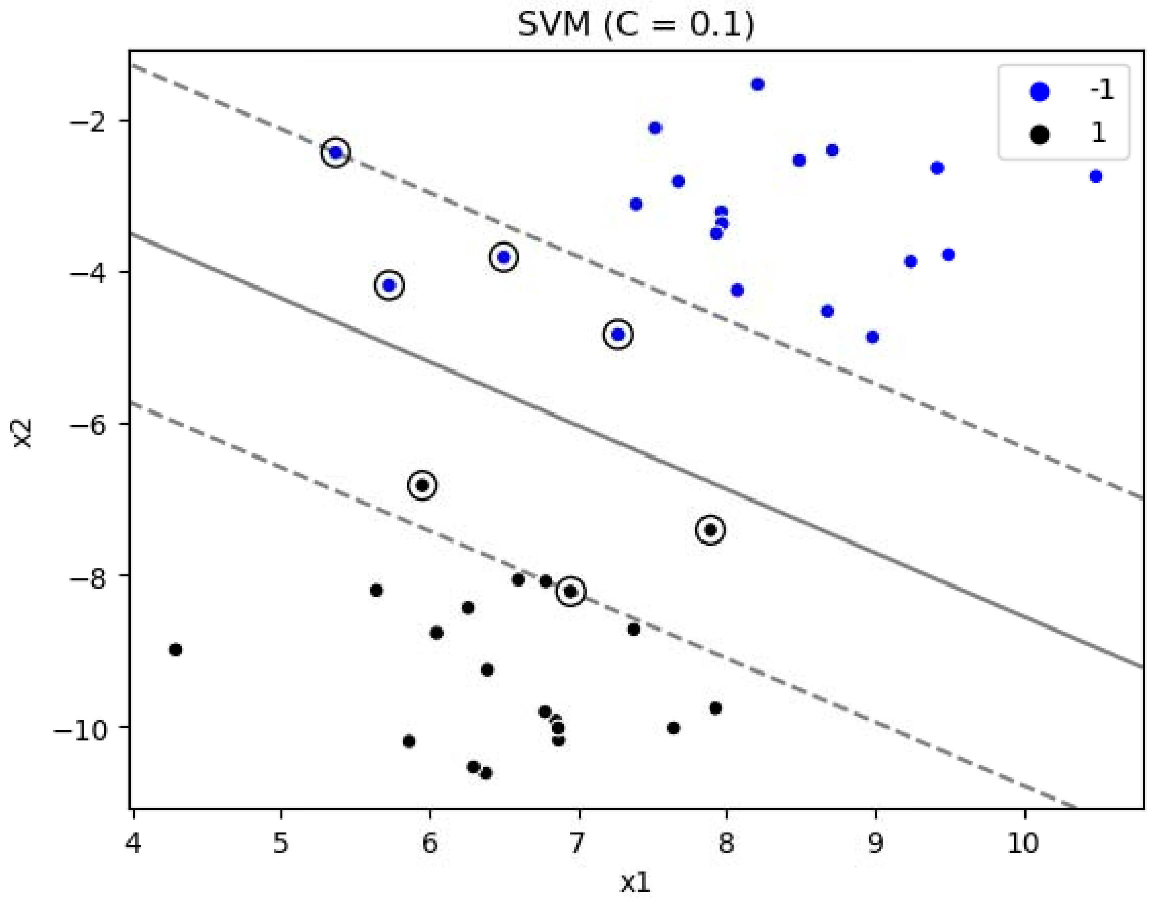

图 14.14 SVM 的估计结果(C=0.1)

从图 14.14 可见，由于惩罚力度较小($C=0.1$)，故间隔更宽，且支持向量增多。

如何确定最优的惩罚参数C？

可使用 sklearn 的 `GridSearchCV` 类进行交叉验证。

先以字典形式定义一个关于 `C` 参数的网格 `param_grid`：

```
In [20]: param_grid = {'C': [0.001, 0.01, 0.1, 1, 10,
100]}
```

由于这是分类问题且样本不大，故使用 sklearn 的 StratifiedKFold 类定义一个 5 折分层随机分组：

```
In [21]: kfold = StratifiedKFold(n_splits=5,
                    shuffle=True, random_state=1)
```

然后，使用 GridSearchCV 类进行交叉验证：

```
In [22]: model = GridSearchCV(LinearSVC(loss="hinge",
                       random_state=123, max_iter=1e4),
                       param_grid, cv=kfold)
```

此命令建立了 GridSearchCV 类的一个实例 model。

使用 fit()方法进行估计：

```
In [23]: model.fit(X, y)
Out[23]:
GridSearchCV(cv=StratifiedKFold(n_splits=5,
    random_state=1,shuffle=True), error_score=nan,
    estimator=LinearSVC(C=1.0, class_weight=None,
    dual=True, fit_intercept=True,
    intercept_scaling=1, loss='hinge',
    max_iter=10000.0, multi_class='ovr', penalty='l2',
    random_state=123, tol=0.0001, verbose=0),
    iid='deprecated', n_jobs=None,
    param_grid={'C': [0.001, 0.01, 0.1, 1, 10, 100]},
```

```
    pre_dispatch='2*n_jobs', refit=True,
    return_train_score=False, scoring=None, verbose=0)
```

使用 model 的 best_params_属性，可得最优参数：

```
In [24]: model.best_params_
Out[24]: {'C': 0.01}
```

结果显示，在网格 param_grid 中，“C = 0.01”为最优参数。

使用 `best_estimator_`属性，将 `model` 重新定义为最优模型(对应于最优参数)：

```
In [25]: model = model.best_estimator_
```

使用 `len()`函数，计算支持向量的数目：

```
In [26]: len(support_vectors(model, X, y))
Out[26]: 21
```

结果显示，支持向量的数量增加到 21 个。

通过属性 intercept_与 coef_，分别展示分离超平面的截距项与系数：

```
In [27]: model.intercept_
Out[27]: array([-0.02441972])

In [28]: model.coef_
Out[28]: array([[-0.23165701, -0.28402698]])
```

更直观地，画图展示支持向量，结果参见图 14.15：

```
In [29]: svm_plot(model, X, y)
```

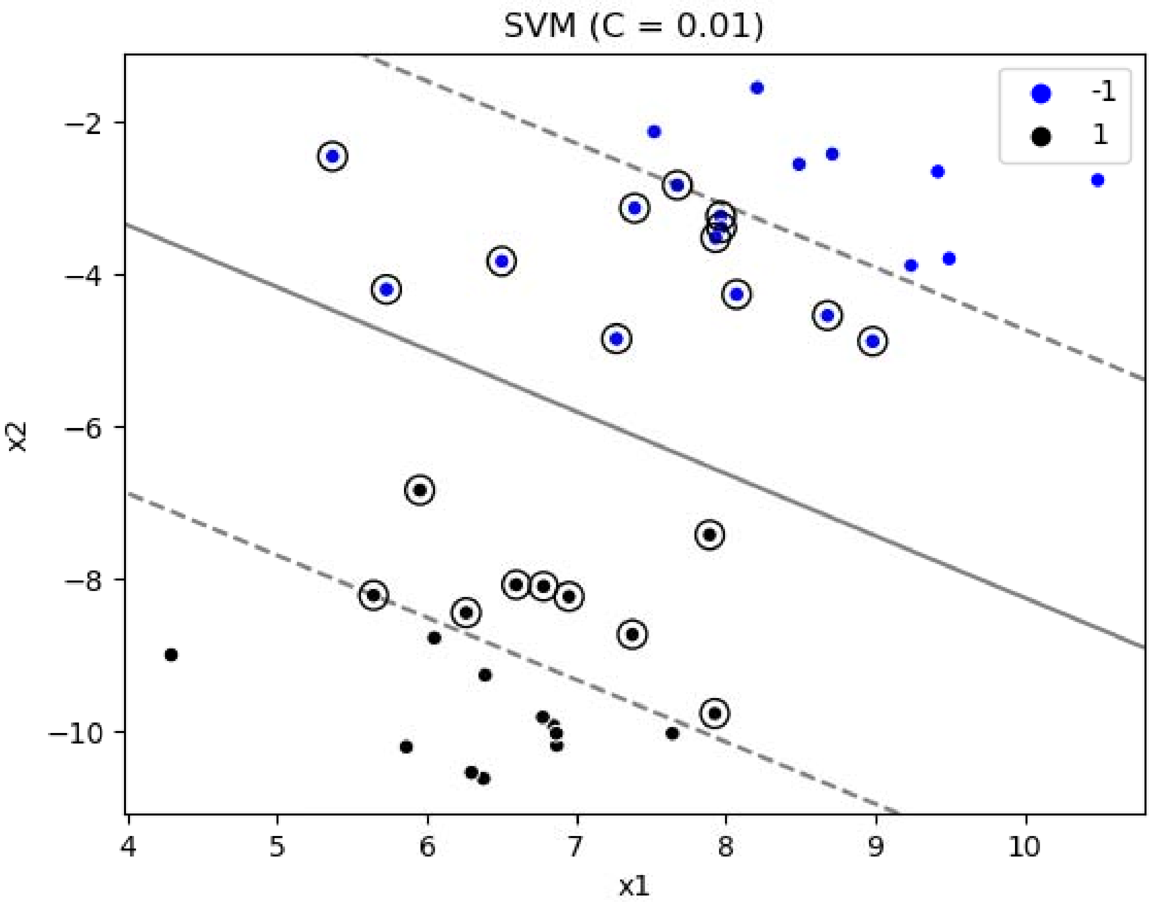

图 14.15 SVM 的估计结果(C=0.01)

从图 14.15 可见，由于惩罚参数降为 0.01，故间隔变得更宽，支持向量进一步增多，而正则化程度更高(更不易过拟合)。还可在“C = 0.01”附近，将参数网格 param_grid 进一步细化，进行交叉验证搜索。

下面，使用同样的数据生成过程(包括同样的随机数种子)，产生一个样本容量为 1000 的测试集：

```
In [30]: X_test, y_test = make_blobs(n_samples=1040,
             centers=2, n_features=2, random_state=6)
    ...: y_test = 2 * y_test - 1
```

其中，第 1 个命令的参数“n_samples=1040”表示生成 1040 个样本点。第 2 个命令将 y_test 的取值从将$\{0,1\}$变换为$\{-1,1\}$。

由于使用了同样的随机数种子，故最前面的 40 个观测值就是此前所用的训练集，而最后面的 1000 个观测值则构成测试集：

```
In [31]: X_test = X_test[40:, :]
    ...: y_test = y_test[40:]
```

画测试集的散点图，结果参见图 14.16：

```
In [32]: data_test = pd.DataFrame(X_test,
                                  columns=['x1','x2'])
    ...: sns.scatterplot(x='x1', y='x2', data=data_test,
                    hue=y_test, palette=['blue', 'black'])
```

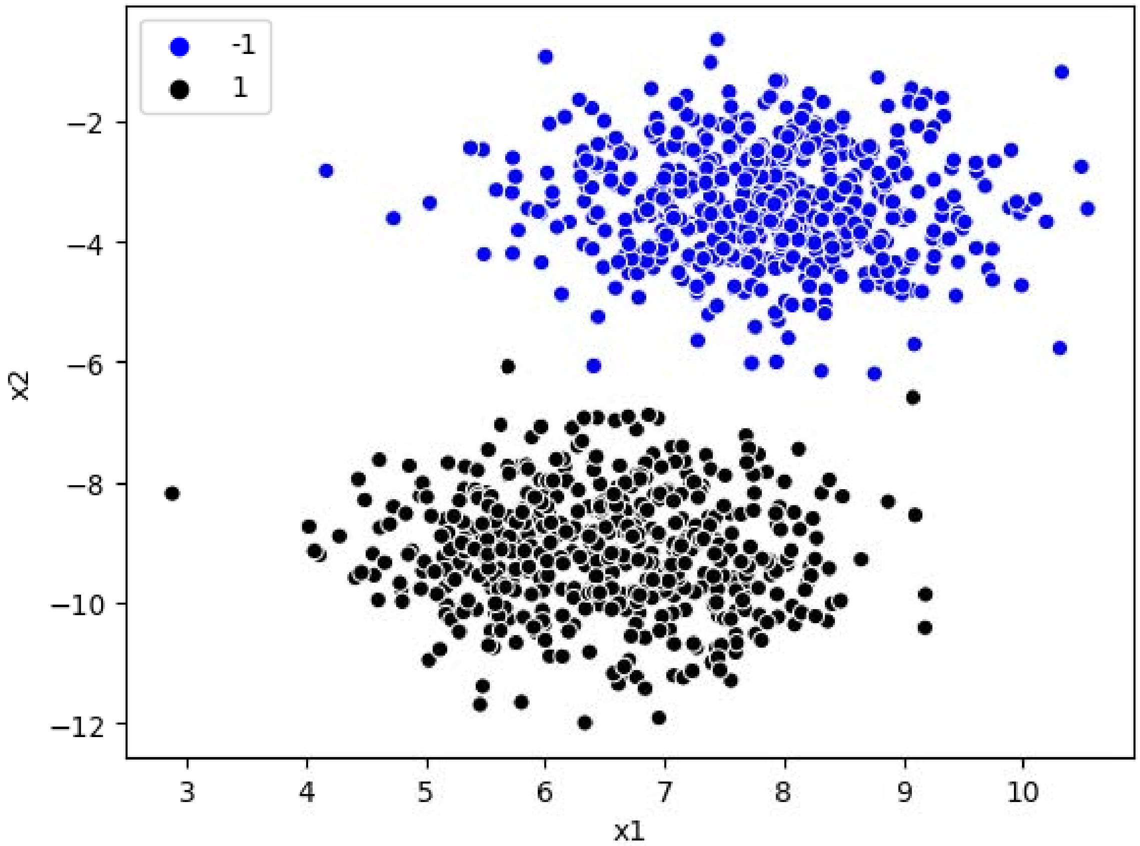

图 14.16 测试集的散点图

根据前面所估的最优模型 model，使用 score()方法，计算测试集的预测准确率：

```
In [33]: model.score(X_test, y_test)
Out[33]: 0.998
```

在测试集中进行预测，并计算混淆矩阵：

```
In [34]: pred = model.predict(X_test)
```

```
In [35]: pd.crosstab(y_test, pred, rownames=['Actual'],
                     colnames=['Predicted'])
Out[35]:
Predicted    0    1
Actual
0          501    1
1            1  497
```

在 sklearn 中，还可使用 SVC 类估计线性支持向量机，只要指定线性核即可：

```
In [36]: model = SVC(kernel='linear', C=0.01,
random_state=123)
```

其中，参数“kernel='linear'”指定线性核(linear kernel)。

此命令生成 SVC 类的一个实例 model。

使用 fit()方法进行估计：

```
In [37]: model.fit(X, y)
Out[37]:
SVC(C=0.01, break_ties=False, cache_size=200,
    class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3,
    gamma='scale', kernel='linear', max_iter=-1,
    probability=False, random_state=123,
    shrinking=True, tol=0.001, verbose=False)
```

使用 model 的属性 n_support_，可得支持向量的数目：

```
In [38]: model.n_support_
Out[38]: array([9, 9])
```

结果显示，两类数据(y=0 与 y=1)各有 9 个支持向量，总共 18 个支持向量。

使用 support_属性，可得支持向量的位置索引：

```
In [39]: model.support_
Out[39]:
array([ 1,  5,  7,  8, 16, 19, 24, 28, 38, 10, 11, 14,
20, 21, 23, 29, 30, 31])
```

结果显示，第 1，5，7，…，30，31 个观测值为支持向量。

使用属性 support_vectors_，展示这些支持向量：

```
In [40]: model.support_vectors_
Out[40]:
array([[ 6.50072722, -3.82403586],
       [ 8.68185687, -4.53683537],
       [ 9.24223825, -3.88003098],
       [ 5.73005848, -4.19481136],
       [ 7.27059007, -4.84225716],
       [ 8.98426675, -4.87449712],
       [ 7.97164446, -3.38236058],
       [ 8.07502382, -4.25949569],
       [ 7.93333064, -3.51553205],
```

```
[ 7.37578372, -8.7241701 ],
[ 6.95292352, -8.22624269],
[ 5.64443032, -8.21045789],
[ 6.6008728 , -8.07144707],
[ 5.95313618, -6.82945967],
[ 6.26221548, -8.43925752],
[ 6.78335342, -8.09238614],
[ 7.89359985, -7.41655113],
[ 6.04907774, -8.76969991]])
```

展示分离超平面的截距项与系数：

```
In [41]: model.intercept_
Out[41]: array([-1.32223698])
In [42]: model.coef_
Out[42]: array([[-0.10169556, -0.33494889]])
```

计算测试集的预测准确率：

```
In [43]: model.score(X_test, y_test)
Out[43]: 0.998
```

更直观地，画图展示支持向量，结果参见图 14.17：

```
In [44]: svm_plot(model, X, y)
```

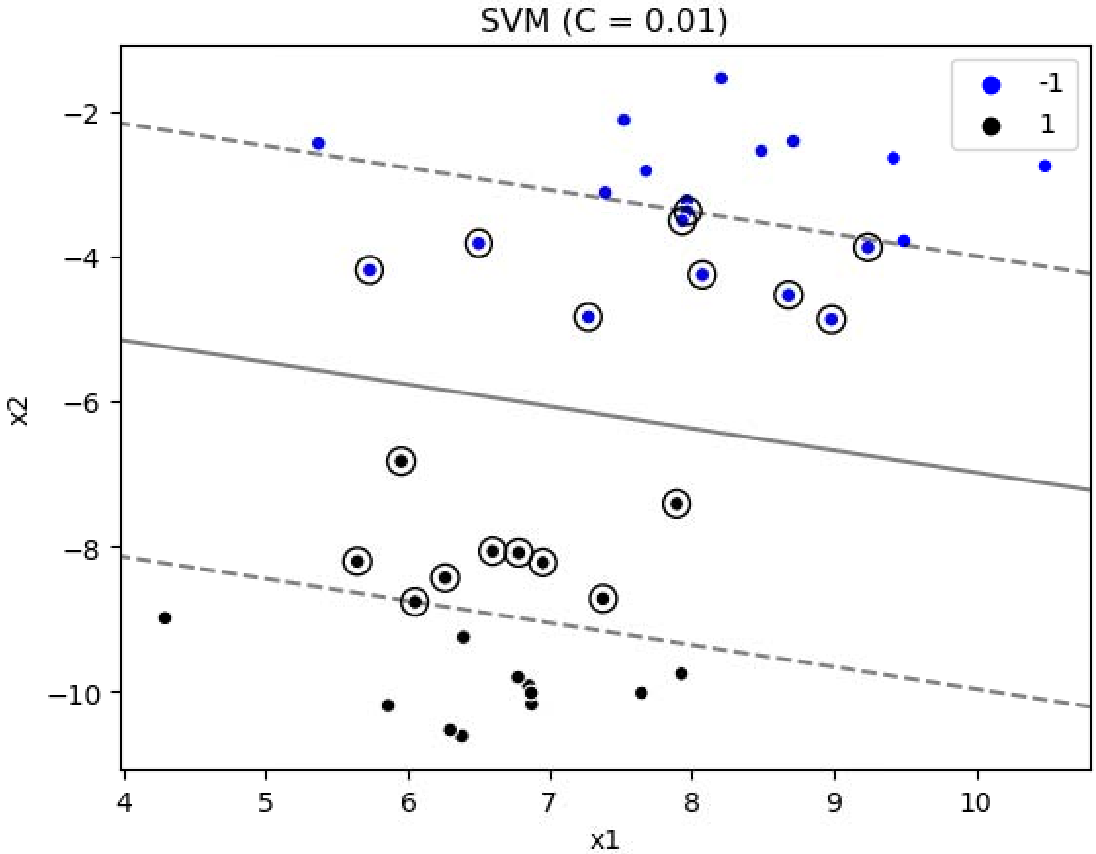

图 14.17 使用 SVC 类的估计结果(C=0.01)

对比以上结果可知，虽然设定了同样的惩罚参数“C=0.01”与随机数种子“random_state=123”，使用 SVC 类与 LinearSVC 类的结果略有不同。

比如，使用 LinearSVC 得到 21 个支持向量，而使用 SVC 类仅得到 18 个支持向量。

二者的回归超平面(截距项与系数)也有所不同，尽管在测试集的预测准确率相同(均为 0.998)。

这是因为，LinearSVC 类使用 liblinear 算法，而 SVC 类使用 libsvm 算法。

对于线性支持向量机，如果样本容量较大，建议使用 `LinearSVC` 类，因为它已针对线性问题进行优化，运行速度更快。

使用 SVC 类的好处在于，它可以使用非线性核(nonlinear kernels)，详见下文。

以上示例的决策边界为线性边界。下面以逻辑上的“异或函数”为例，考察 SVM 在非线性决策边界中的应用。

作为一种逻辑运算，**异或**(Exclusive Or，简记 XOR)是一种排他性(exclusive)的“或”，即当二者取值不同则为“真”(True)，而当二者取值相同即为“假”(False)，详见第 15 章。

首先，生成模拟数据:

```
In [45]: np.random.seed(1)
    ...: X = np.random.randn(200, 2)
    ...: y = np.logical_xor(X[:, 0] > 0, X[:, 1] > 0)
```

其中，200×2的数据矩阵 X 来自标准正态分布；而 Numpy 的 logical_xor()函数，返回逻辑值 True，当条件“X[:, 0] > 0”(X 的第 0 个变量为正)与“X[:, 1] > 0”(X 的第 1 个变量为正)不同时(一对一错)；反之，若这两个条件同对或同错，则返回逻辑值 False。

使用 Numpy 的 `where()`函数，将 `y` 赋值为 1 或-1：

```
In [46]: y = np.where(y, 1, -1)
```

其中，取值为 `True` 的 `y` 被赋值为 1，而取值为 `False` 的 `y` 被赋值为-1。

将数据矩阵 `X` 设为数据框，并画散点图，结果参见图 14.18：

```
In [47]: data = pd.DataFrame(X, columns=['x1', 'x2'])
    ...: sns.scatterplot(x='x1', y='x2', data=data,
                         hue=y, palette=['blue', 'black'])
```

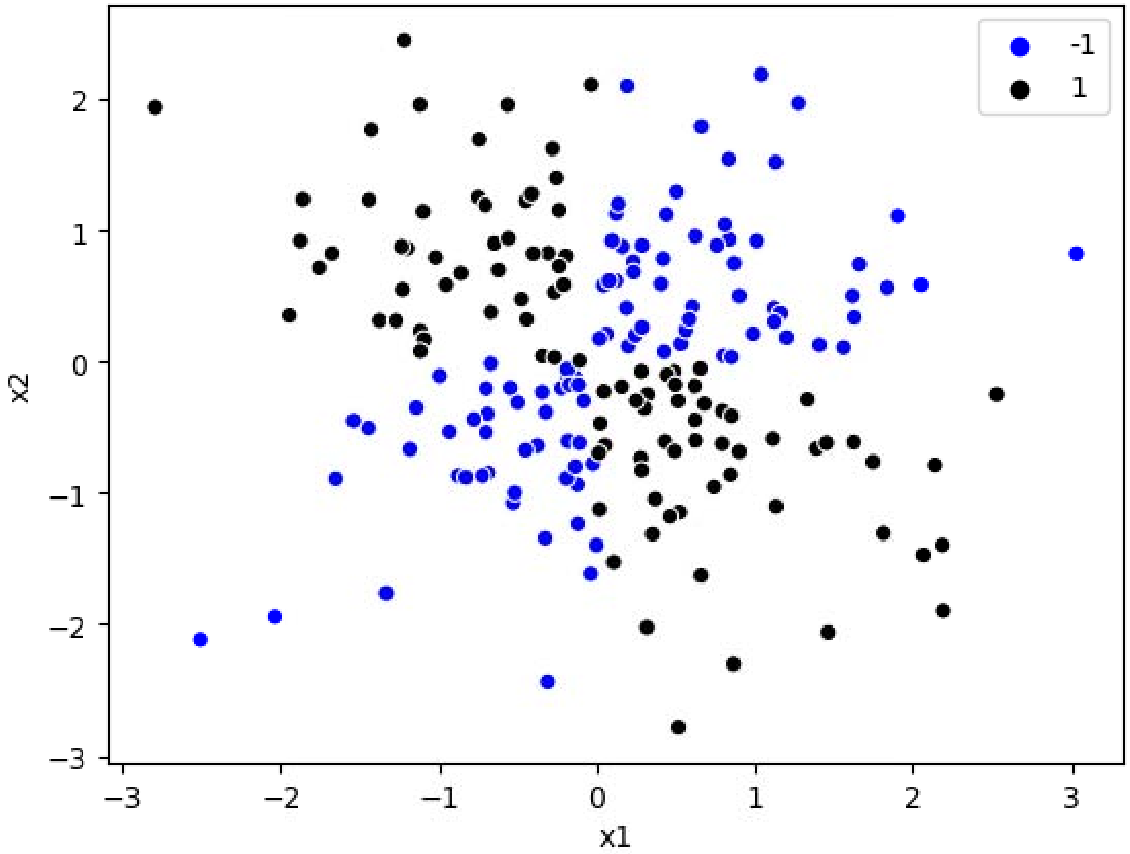

图 14.18 非线性决策边界的模拟数据

两类数据的决策边界为非线性，无法使用超平面进行分离。

下面，使用 sklearn 的 SVC 类估计支持向量机，并使用径向核：

```
In [48]: model = SVC(kernel='rbf', C=1, gamma=0.5,
random_state=123)
```

其中，参数“kernel='rbf'”表示使用径向核，也称“径向基函数”(radial basis function)；这是默认值。参数“C=1”设定惩罚参数 C 为 1，这也是默认值。参数“gamma=0.5”设定径向核的γ参数为 0.5，参见方程(14.31)。

此命令生成了 SVC 类的一个实例 model。

使用 fit()方法进行估计：

```
In [49]: model.fit(X, y)
Out[49]:
SVC(C=1, break_ties=False, cache_size=200,
    class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3,
    gamma=0.5, kernel='rbf', max_iter=-1,
    probability=False, random_state=123,
    shrinking=True, tol=0.001, verbose=False)
```

通过 `model` 的 `n_support_` 属性，可得支持向量的数目：

```
In [50]: model.n_support_
Out[50]: array([46, 47])
```

结果显示，两类数据分别有 46 与 47 个支持向量。

使用 support_属性，可得支持向量的位置索引：

```
In [51]: model.support_
Out[51]:
array([  8,   9,  13,  17,  23,  24,  36,  40,  41,  42,
44,  45,  47,        58,  60,  65,  69,  70,  72,  73,
75,  78,  92, 109, 111, 113, 114, 115, 117, 121, 125, 132,
133, 141, 142, 143, 145, 147, 157, 160, 166, 169, 173,
190, 191, 193,   4,  14,  18,  22,  25,  27, 31,  32,  37,
38,  43,  48,  52,  54,  77,  79,  81,  82,  84, 86,  87,
88,  89,  94, 100, 108, 126, 129, 130, 134, 139, 144, 152,
153, 158, 161, 164, 171, 172, 176, 178, 182, 183, 185,
192, 196, 199])
```

若要直接展示所有支持向量，可输入命令“model.support_vectors_”；为节省空间，在此从略。

更直观地，可使用 mlxtend 模块的 plot_decision_regions()函数画决策边界：

```
In [52]: plot_decision_regions(X, y, model,
                               hide_spines=False)
    ...: plt.title('SVM (C=1, gamma=0.5)')
```

其中，参数“hide_spines=False”表示显示图的四周方框(默认不显示)，结果参见图 14.19。

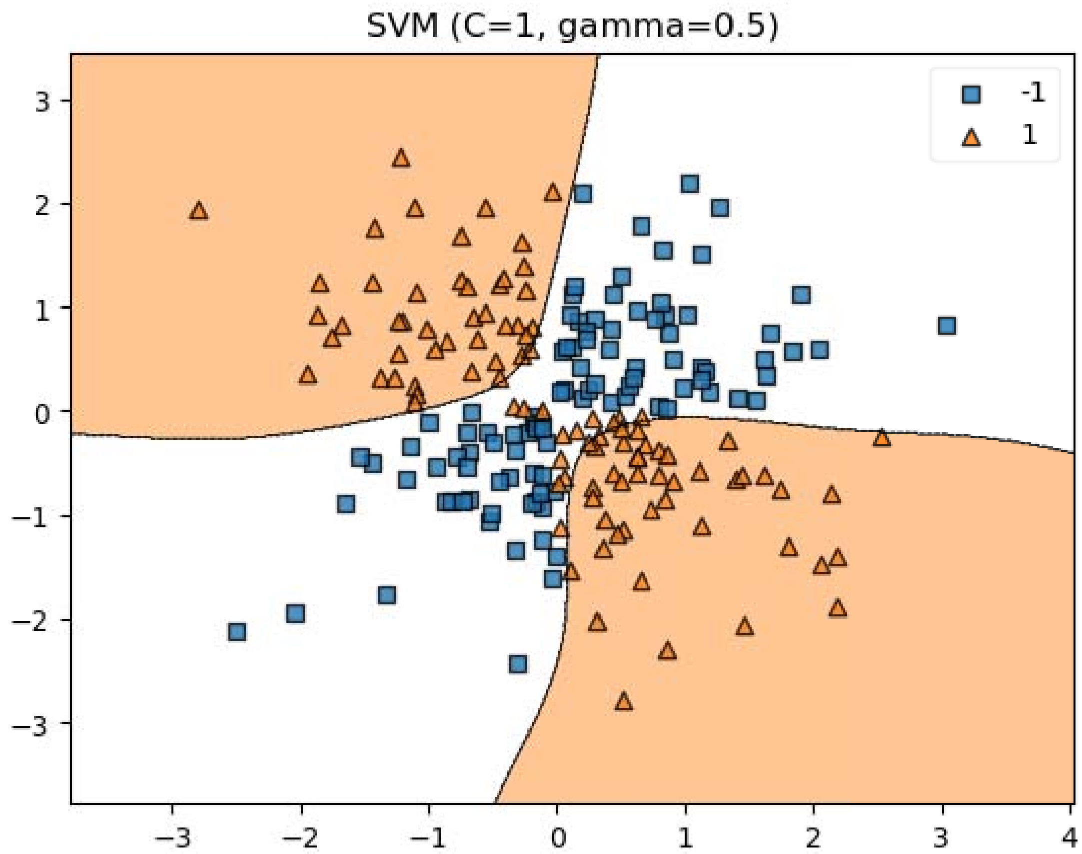

图 14.19 SVM 的决策边界(`C=1`, `gamma=0.5`)

从图 14.19 可见，决策边界为非线性，且可较好地区分两类数据。

使用 `score()`方法，计算样本内的预测准确率：

```
In [53]: model.score(X, y)
Out[53]: 0.93
```

结果显示，样本内的预测准确率为 93%，预测效果良好。

下面，将惩罚参数 C 增大至 10000(仍然保持 gamma=0.5)，考察它对决策边界的影响，结果参见图 14.20：

```
In [54]: model = SVC(kernel='rbf', C=10000, gamma=0.5,
                     random_state=123)
    ...: model.fit(X, y)
    ...: plot_decision_regions(X, y, model,
                               hide_spines=False)
    ...: plt.title('SVM (C=10000, gamma=0.5)')
```

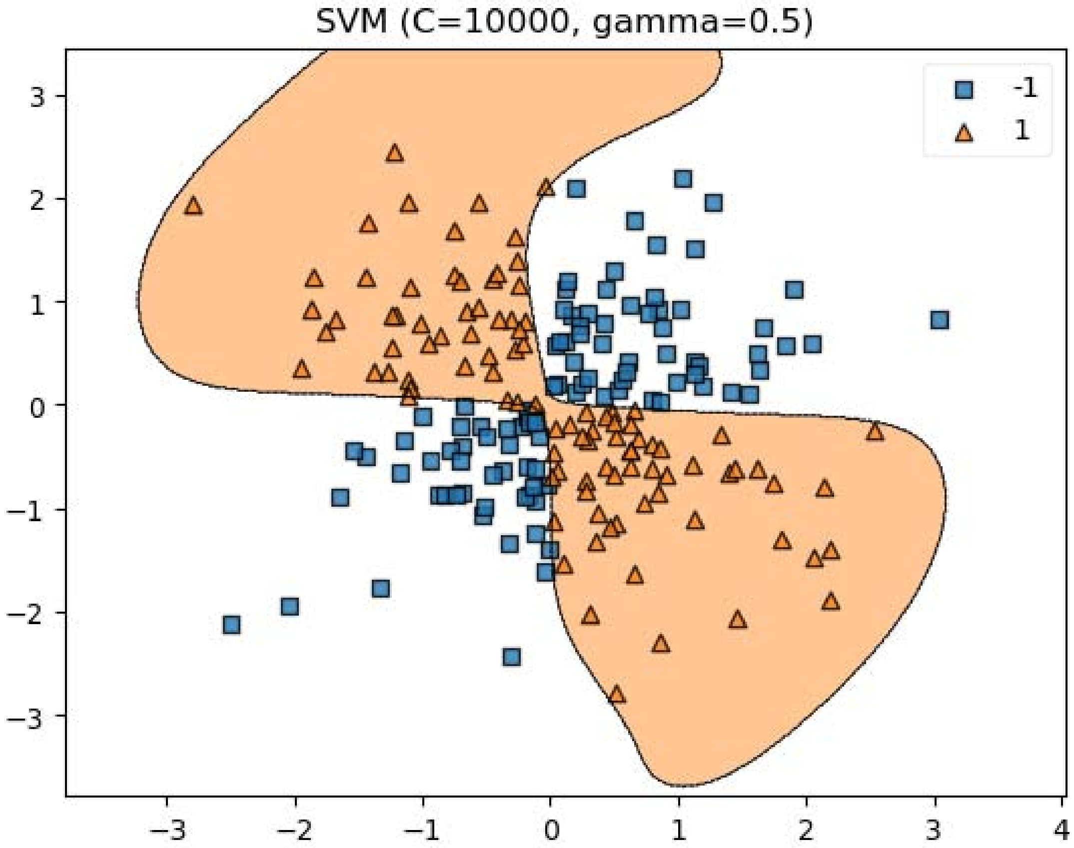

图 14.20 SVM 的决策边界(C=10000, gamma=0.5)

从图 14.20 可见，由于惩罚参数 C 很大，导致决策边界扭曲(受训练集影响很大)，这是过拟合的一种表现。

反之，将惩罚参数 C 降低至 0.01，考察它对决策边界的影响，结果参见图 14.21：

```
In [55]: model = SVC(kernel='rbf', C=0.01, gamma=0.5,
                     random_state=123)
    ...: model.fit(X, y)
    ...: plot_decision_regions(X, y, model,
                               hide_spines=False)
    ...: plt.title('SVM (C=0.01, gamma=0.5)')
```

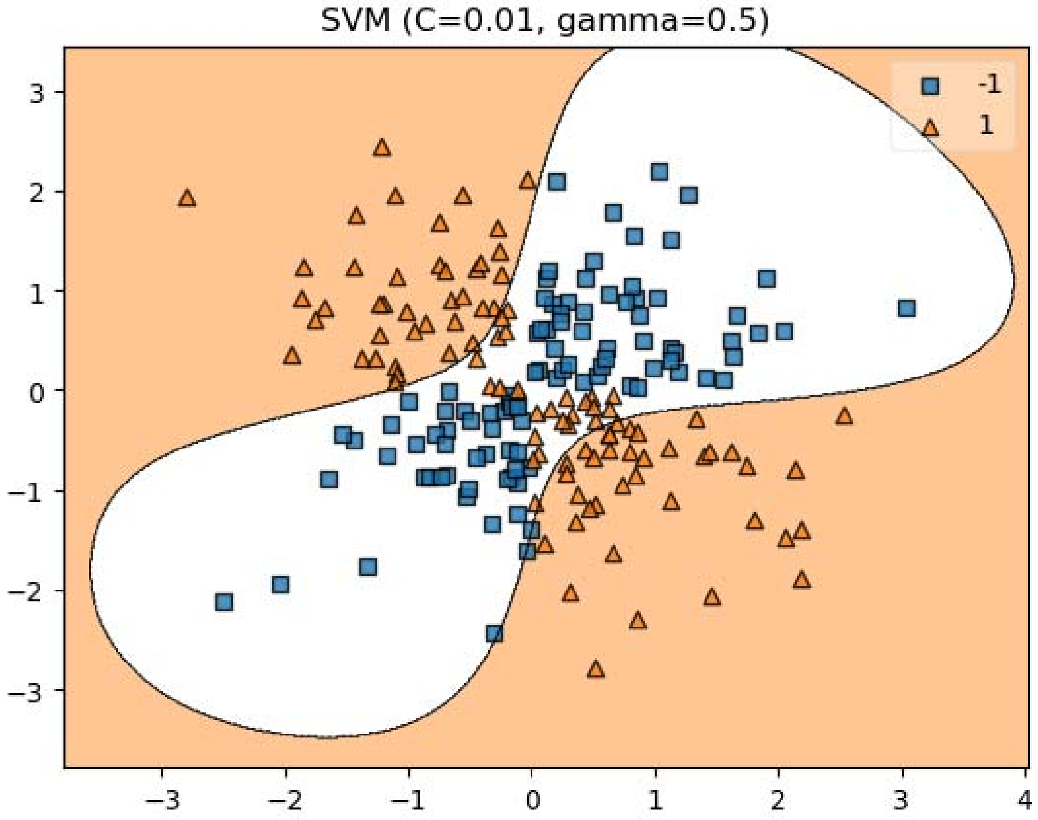

图 14.21 SVM 的决策边界(`C=0.01`, `gamma=0.5`)

从图 14.21 可见，由于惩罚参数 C 很小，导致决策边界过分光滑，未能很好地区分两类数据，出现欠拟合。

下面，考虑参数 gamma 的作用。首先，增大参数 gamma 至 50(同时保持 C=1)，考察它对决策边界的影响，结果参见图 14.22：

```
In [56]: model = SVC(kernel='rbf', C=1, gamma=50,
                     random_state=123)
    ...: model.fit(X, y)
    ...: plot_decision_regions(X, y, model,
                               hide_spines=False)
    ...: plt.title('SVM (C=1, gamma=50)')
```

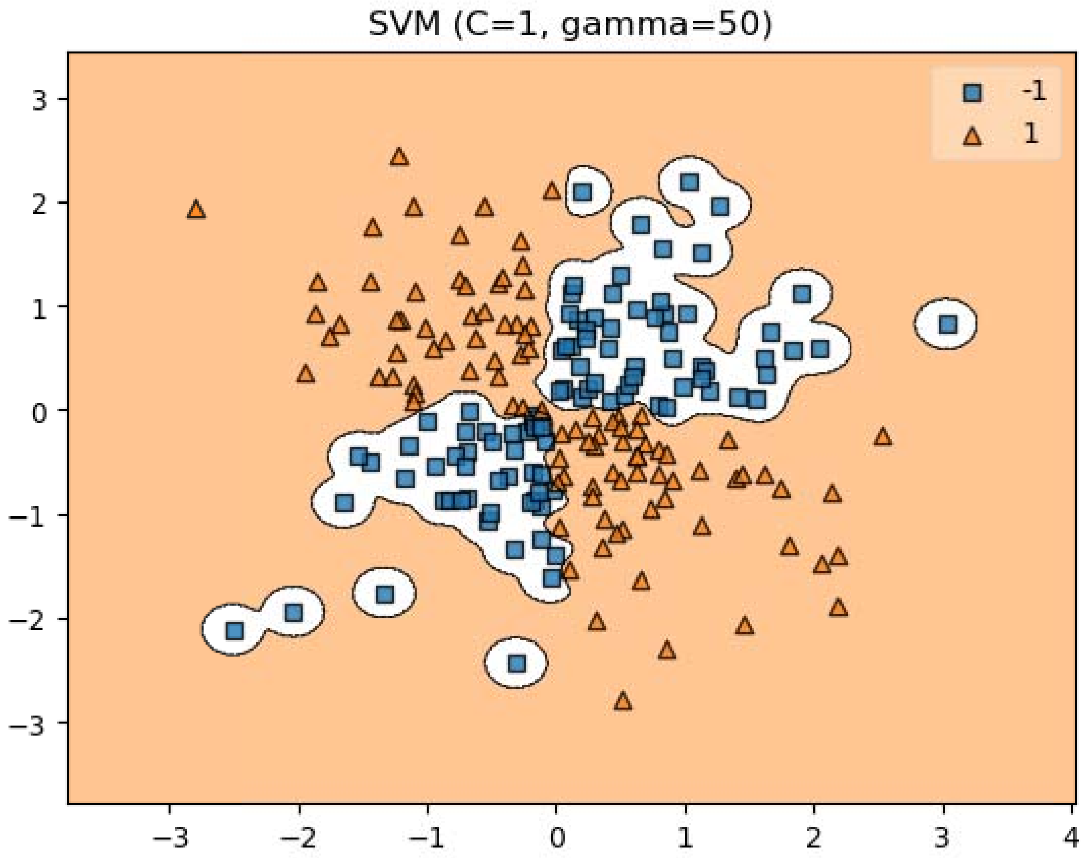

图 14.22 SVM 的决策边界(C=1, gamma=50)

从图 14.22 可见，将参数 gamma 增大到 50 后，由于每个观测值的影响范围变得很小(部分决策边界甚至由单个观测值所决定)，故决策边界可完美地区分两类数据，出现了较为严重的过拟合。使

用 score()方法，计算样本内的预测准确率：

```
In [57]: model.score(X, y)
Out[57]: 1.0
```

结果表明，样本内的预测准确率为 100%。

虽然训练误差为 0，但过拟合使得模型的泛化能力下降。

相反地，将参数 gamma 降低到 0.05，考察它对于决策边界的影响，结果参见图 14.23：

```
In [58]: model = SVC(kernel='rbf', C=1, gamma=0.05,
                     random_state=123)
    ...: model.fit(X, y)
    ...: plot_decision_regions(X, y, model,
                               hide_spines=False)
    ...: plt.title('SVM (C=1, gamma=0.05)')
```

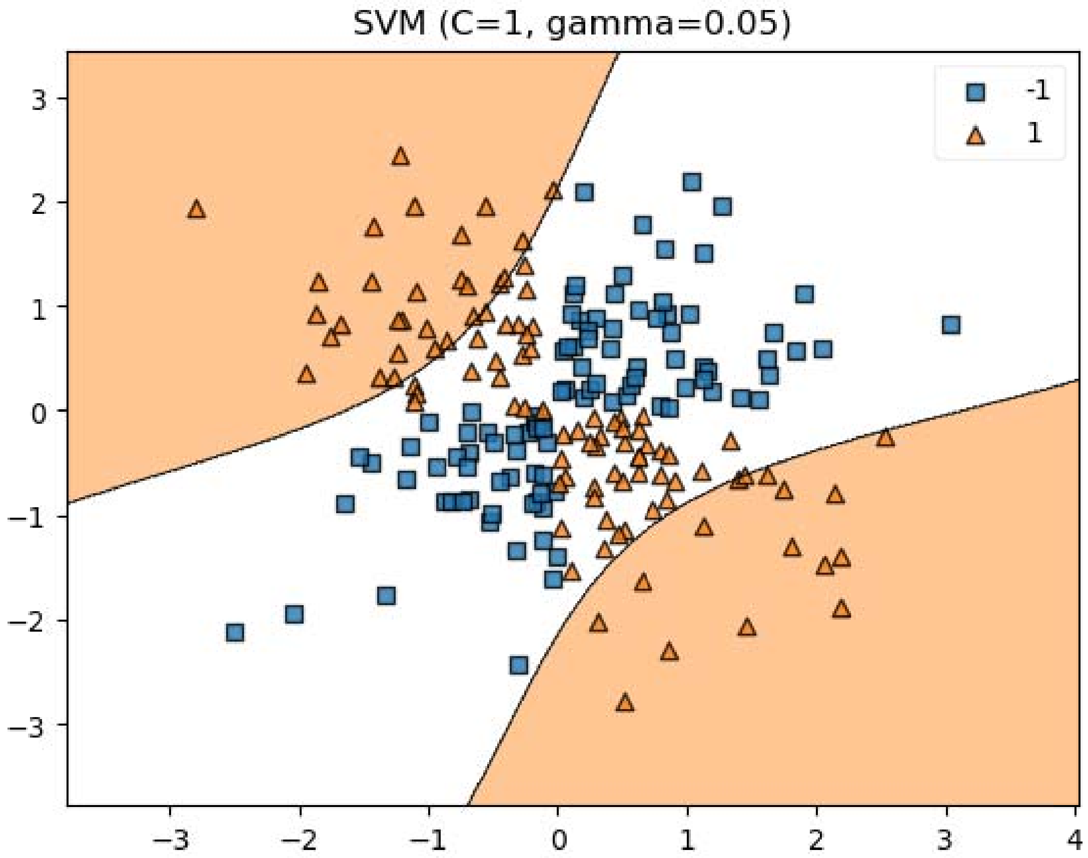

图 14.23 SVM 的决策边界(C=1, gamma=0.05)

从图 14.23 可见，将参数 `gamma` 减少到 0.05 后，由于每个观测值的影响范围很大，故决策边界变得过于光滑，出现了欠拟合。

使用 `score()`方法，计算样本内的预测准确率：

```
In [59]: model.score(X, y)
Out[59]: 0.715
```

结果显示，样本内的预测准确率只有 71.5%，训练误差大幅上升。

如何选择最优的调节参数(C, γ)？

下面，对这两个超参数同时进行交叉验证。

先设定字典形式的参数网格 `param_grid`：

```
In [60]: param_grid = {'C': [0.001, 0.01, 0.1, 1, 10,
           100], 'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
```

然后，使用 GridSearchCV 类，进行 10 折交叉验证，并展示最优参数组合：

```
In [61]: kfold = StratifiedKFold(n_splits=10,
                                 shuffle=True, random_state=1)
    ...: model = GridSearchCV(SVC(kernel='rbf',
                   random_state=123), param_grid, cv=kfold)
    ...: model.fit(X, y)
    ...: model.best_params_
Out[61]: {'C': 100, 'gamma': 0.1}
```

结果显示，最优参数为“C=100”与“gamma=0.1”。

画最优模型的决策边界，结果参见图 14.24：

```
In [62]: plot_decision_regions(X, y, model,
                               hide_spines=False)
    ...: plt.title('Optimal SVM (C=100, gamma=0.1)')
```

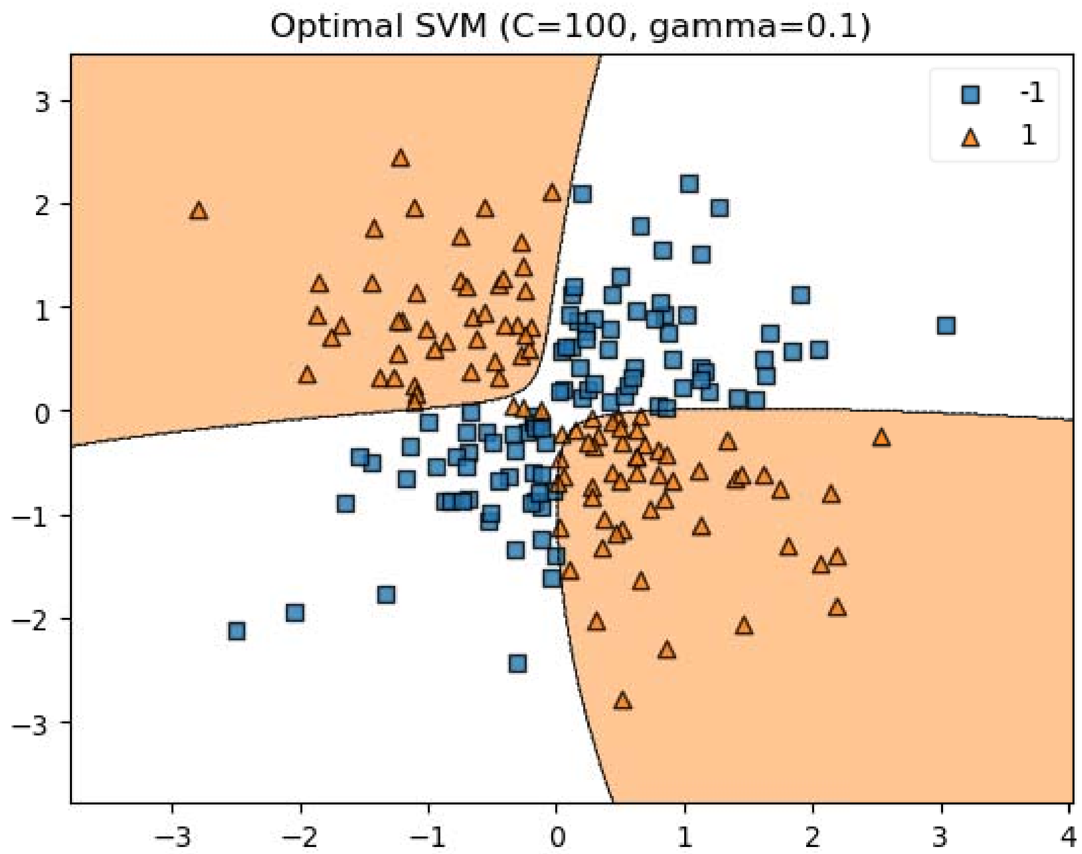

图 14.24 最优 SVM 模型的决策边界(`C=100`, `gamma=0.1`)

从图 14.24 可见，最优模型的拟合效果良好，既非欠拟合，也无过拟合。

下面，生成 1000 个测试集的观测值，并计算测试集的预测准确率：

```
In [63]: np.random.seed(369)
    ...: X_test = np.random.randn(1000, 2)
    ...: y_test = np.logical_xor(X_test[:, 0] > 0,
                                 X_test[:, 1] > 0)
    ...: y_test = np.where(y_test, 1, -1)
    ...: model.score(X_test, y_test)
Out[63]: 0.948
```

结果显示，测试集的预测准确率达到 94.8%。

在测试集中预测，并计算相应的混淆矩阵：

```
In [64]: pred = model.predict(X_test)
    ...: pd.crosstab(y_test, pred, rownames=['Actual'],
                           colnames=['Predicted'])
Out[64]:
Predicted   -1    1
Actual
-1         483    6
 1          46  465
```

14.10 支持向量机的二分类 Python 案例

下面使用真实数据演示 SVM 的 Python 操作。

对于二分类问题，我们使用过滤垃圾邮件的 `spam` 数据(参见第 8 章)。

* 详见教材，以及配套 Python 程序（现场演示）。

14.11 支持向量机的多分类 Python 案例

对于多分类问题，我们以 sklearn 模块自带的手写数字数据 `digits` 作为案例。

* 详见教材，以及配套 Python 程序（现场演示）。

14.12 支持向量回归的 Python 案例

我们以波士顿房价数据 `boston` 为例(参见第 4 章)，演示支持向量回归的操作。

* 详见教材，以及配套 Python 程序（现场演示）。