第6章

支持向量机

支持向量机（Support Vector Machines，SVM）是一种二元分类模型。其核心思想是，训练阶段在特征空间中寻找一个超平面，它能（或尽量能）将训练样本中的正例和负例分离在它的两侧，预测时以该超平面作为决策边界判断输入实例的类别。寻找超平面的原则是，在可分离的情况下使超平面与数据集间隔最大化。支持向量机是一类模型的统称，其中包括线性可分支持向量机、线性支持向量机以及非线性支持向量机。

6.1 线性可分支持向量机

我们先从最简单的线性可分支持向量机讲起，它是其他复杂支持向量机的基础。

6.1.1 分离超平面

假设有数据集 $\boldsymbol{D}$，其中的样本 (x_i, y_i) 有两种类别，分别称为正例（$y_i = +1$）和负例（$y_i = -1$）。如果特征空间内存在某个超平面能将正例和负例完全正确地分离到它的两侧，则称数据集 $\boldsymbol{D}$ 为线性可分数据集；如果不存在，则称数据集 $\boldsymbol{D}$ 为线性不可分数据集。

如图 6-1 所示为一个线性可分数据集。

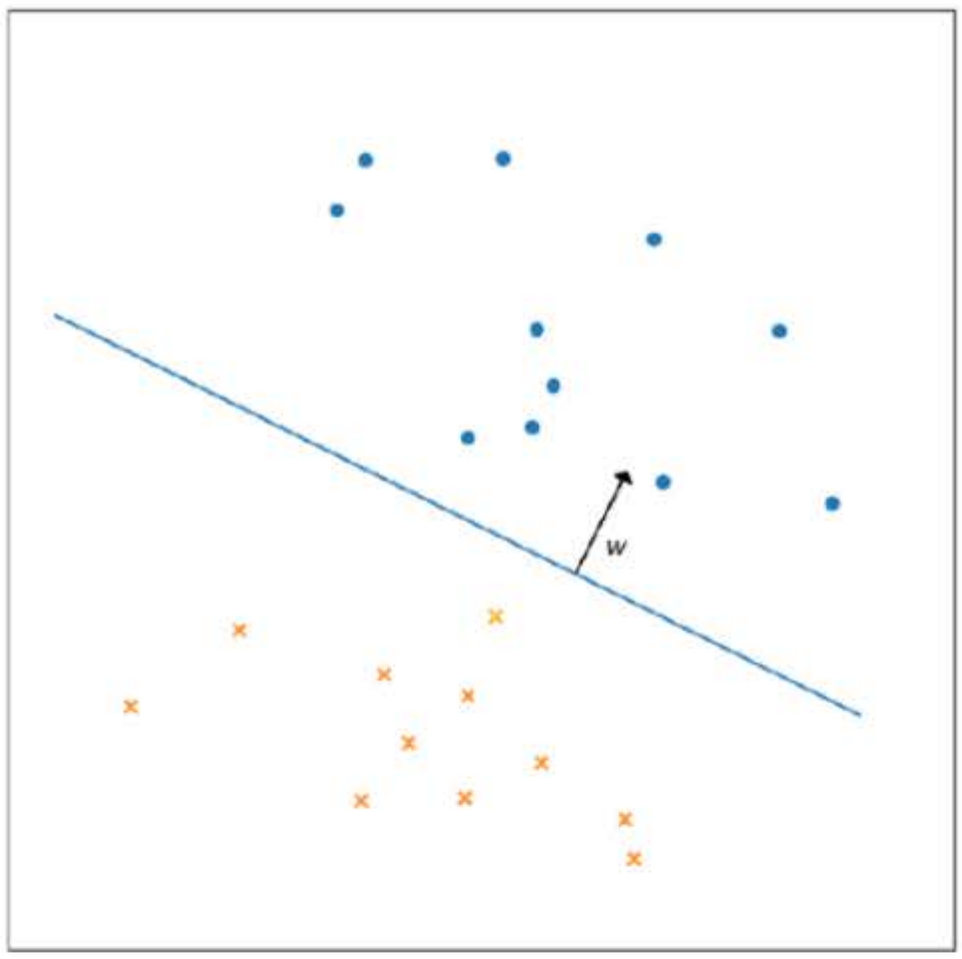

图 6-1

空间 $\mathbf{R}^n$ 中一个超平面，可用如下方程表示：

$$w^{\mathrm{T}}x+b=0$$

其中，$w\in\mathbf{R}^n$ 为平面的法向量，$b\in\mathbf{R}^1$ 为截距。

空间中的一点 x 与一个超平面 (w,b) 的相对位置可依据以下公式进行判断：

$$\begin{cases} w^{\mathrm{T}}x+b>0, & x\text{在平面上方} \\ w^{\mathrm{T}}x+b=0, & x\text{在平面上} \\ w^{\mathrm{T}}x+b<0, & x\text{在平面下方} \end{cases}$$

下面推导上述判断公式。在超平面上任意选取一点 x'，连接 x'（尾）和 x（头）构成向量$x-x'$，则向量 $x-x'$ 在法向量 w 方向上的标量投影为：

$$\begin{aligned} \alpha &= \frac{w^{\mathrm{T}}(x-x')}{\|w\|} \\ &= \frac{w^{\mathrm{T}}x-w^{\mathrm{T}}x'}{\|w\|} \\ &= \frac{w^{\mathrm{T}}x+b}{\|w\|} \end{aligned}$$

$\|w\|$ 总为正值，α 的符号取决于 $w^{\mathrm{T}}x+b$ 的符号。标量投影是点到超平面带符号的距离，超平面一侧的点向量投影总与 w 同方向，距离为正；超平面另一侧的点向量投影总与 w 反方向，距离为负。由此得出上述判断公式。

6.1.2 间隔最大化

对于一个线性可分数据集，实际上有无穷多个超平面可以完全正确地将正例和负例分离，我们要从中挑选一个作为决策边界。一个样本点距分离超平面越远，我们越能确信它会被正确分类。因此，挑选分离超平面的原则是该超平面尽量远离数据集中的每一个样本点。为刻画样本点与超平面的远近关系，下面定义两种间隔。

定义样本点 (x_i, y_i) 到超平面 (w, b) 的几何间隔（Geometric Margin）为：

$$\gamma_i = \frac{y_i (w^{\mathrm{T}} x + b)}{\|w\|}$$

定义样本点 (x_i, y_i) 到超平面 (w, b) 的函数间隔（Functional Margin）为：

$$\hat{\gamma}_i = y_i (w^{\mathrm{T}} x + b)$$

几何间隔与函数间隔的关系为：

$$\gamma_i = \frac{\hat{\gamma}_i}{\|w\|}$$

因为 α 是带符号的距离，在超平面对样本点 (x_i, y_i) 正确分类的情况下，几何间隔 γ_i 即为点到超平面的距离，则：

$$\gamma_i = y_i \alpha_i = |\alpha_i| > 0$$

再定义数据集与超平面的几何间隔为数据集中所有样本点与超平面的几何间隔的最小值，即：

$$\gamma = \min_i \gamma_i$$

回到分离超平面的挑选问题。用以上概念描述，我们所要挑选的是与数据集几何间隔最大的超平面，即求解如下约束最优化问题：

$$\max_{w,b} \ \gamma$$

$$s.t. \quad \frac{y_i (w^{\mathrm{T}} x_i + b)}{\|w\|} \geqslant \gamma \qquad i = 1, 2, \ldots, m$$

将 $\gamma = \frac{\hat{\gamma}}{\|w\|}$ 代入上式，使用函数间隔描述该问题，得：

$$\max_{w,b} \ \frac{\hat{\gamma}}{\|w\|}$$

$$s.t. \quad y_i (w^{\mathrm{T}} x_i + b) \geqslant \hat{\gamma} \qquad i = 1, 2, \ldots, m$$

再来思考，若将 (w,b) 按比例缩放至 $(\lambda w,\lambda b)$，其中 $\lambda \neq 0$，此时由 $(\lambda w,\lambda b)$ 确定的与之前由 (w,b) 确定的是同一个超平面，但数据集函数间隔变为之前的 λ 倍，即 $\hat{\gamma}_{new} = \lambda \hat{\gamma}_{old}$。由此可发现，在超平面确定的情况下，按比例缩放 (w,b) 并不影响 $\frac{\hat{\gamma}}{\|w\|}$ 的值。这就意味着可以在给定 $\hat{\gamma}$ 为任意正值的情况下，来计算以上约束最优化问题，为计算方便，可令 $\hat{\gamma} = 1$（换句话说，一旦超平面确定，总可以通过调整 λ 使得 $\hat{\gamma} = 1$）。此时约束最优化问题变为：

$$\begin{aligned} &\max_{w,b} \quad \frac{1}{\|w\|} \\ &s.t. \quad y_i (w^{\mathrm{T}} x_i + b) \geqslant 1 \qquad i = 1,2,\ldots,m \end{aligned}$$

求 $\max \frac{1}{\|w\|}$，即求 $\min \frac{1}{2} \|w\|^2$，问题又等价于：

$$\begin{aligned} &\min_{w,b} \quad \frac{1}{2} \|w\|^2 \\ &s.t. \quad y_i (w^{\mathrm{T}} x_i + b) \geqslant 1 \qquad i = 1,2,\ldots,m \end{aligned}$$

以上形式的约束最优化问题为凸二次规划问题，求解该问题便可得到最大间隔分离超平面。下一节我们来学习求解方法。

6.1.3 拉格朗日对偶法

求解凸二次规划问题，可先利用拉格朗日对偶性（Lagrange Duality）将原始问题转换为对偶问题，再通过求解对偶问题得到原始问题的解。下面使用该方法求解上一节最后得出的凸二次规划问题。

1. 对偶问题

首先，令：

$$\begin{aligned} f(w) &= \frac{1}{2} \|w\|^2 = \frac{1}{2} w^{\mathrm{T}} w \\ c_i(w,b) &= 1 - y_i (w^{\mathrm{T}} x_i + b) \qquad i = 1,2,\ldots,m \end{aligned}$$

作广义拉格朗日函数：

$$\begin{aligned} L(w,b,\alpha) &= f(w) + \sum_{i=1}^{m} \alpha_i c_i(w,b) \\ &= \frac{1}{2} w^{\mathrm{T}} w - \sum_{i=1}^{m} \alpha_i y_i (w^{\mathrm{T}} x_i + b) + \sum_{i=1}^{m} \alpha_i \end{aligned}$$

其中，α_i为拉格朗日乘子，每个约束条件对应一个拉格朗日乘子，且有$\alpha_i \geqslant 0$。数据集有 m 个样本，则约束条件和拉格朗日乘子也有 m 个。

根据拉格朗日对偶性，原始问题的对偶问题为：

$$\max_{\alpha} \min_{w,b} L(w,b,\alpha)$$

先求 $\min_{w,b} L(w,b,\alpha)$，分别求 L 对 w,b 的梯度，并令其为 0：

$$\nabla_w L(w,b,\alpha) = w - \sum_{i=1}^{m} \alpha_i y_i x_i = 0$$

$$\nabla_b L(w,b,\alpha) = -\sum_{i=1}^{m} \alpha_i y_i = 0$$

得：

$$w = \sum_{i=1}^{m} \alpha_i y_i x_i$$

$$\sum_{i=1}^{m} \alpha_i y_i = 0$$

满足以上条件时 L 取到极值：

$$\begin{aligned}
\min_{w,b} L(w,b,\alpha) &= \frac{1}{2} w^{\mathrm{T}} w - \sum_{i=1}^{m} \alpha_i y_i (w^{\mathrm{T}} x_i + b) + \sum_{i=1}^{m} \alpha_i \\
&= \frac{1}{2} w^{\mathrm{T}} w - w^{\mathrm{T}} \sum_{i=1}^{m} \alpha_i y_i x_i + b \sum_{i=1}^{m} \alpha_i y_i + \sum_{i=1}^{m} \alpha_i \\
&= \frac{1}{2} w^{\mathrm{T}} w - w^{\mathrm{T}} w + \sum_{i=1}^{m} \alpha_i \\
&= -\frac{1}{2} w^{\mathrm{T}} w + \sum_{i=1}^{m} \alpha_i \\
&= -\frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j (x_i^{\mathrm{T}} x_j) + \sum_{i=1}^{m} \alpha_i
\end{aligned}$$

此时，对偶问题变为：

$$\max_{\alpha} -\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j (x_i^{\mathrm{T}} x_j) + \sum_{i=1}^{m}\alpha_i$$

$$s.t. \quad \sum_{i=1}^{m}\alpha_i y_i = 0$$

$$\alpha_i \geqslant 0, \qquad i = 1, 2, \ldots, m$$

等价于：

$$\min_{\alpha} \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j (x_i^{\mathrm{T}} x_j) - \sum_{i=1}^{m}\alpha_i$$

$$s.t. \quad \sum_{i=1}^{m}\alpha_i y_i = 0$$

$$\alpha_i \geqslant 0, \qquad i = 1, 2, \ldots, m$$

以上对偶问题比原始问题容易求解。

2. 原始问题的解

接下来学习在已求得对偶问题的最优解后，如何通过该解求出原始问题的最优解。在原始问题中，$f(w)$ 和 $c_i(w,b)$ 满足以下条件：

- $f(w)$ 和 $c_i(w,b)$ 是凸函数。
- 不等式约束 $c_i(w,b)$ 是严格可行的，即存在 w,b 使得所有 $c_i(w,b)<0$。

可以证明存在原始问题的最优解 (w^*,b^*)，以及对偶问题的最优解 α^*，使得原始问题和对偶问题的最优值相等，并且 (w^*,b^*) 和 α^* 满足 KKT（Karush-Kunh-Tucker）条件：

$$\nabla_w L(w^*,b^*,\alpha^*) = w^* - \sum_{i=1}^{m}\alpha_i^* y_i x_i = 0$$

$$\nabla_b L(w^*,b^*,\alpha^*) = -\sum_{i=1}^{m}\alpha_i^* y_i = 0$$

$$\alpha_i^*(y_i(w^{*\mathrm{T}}x_i + b^*) - 1) = 0$$

$$y_i(w^{*\mathrm{T}}x_i + b^*) - 1 \geqslant 0$$

$$\alpha_i^* \geqslant 0 \qquad i = 1, 2, \ldots, m$$

以上结论表明，如果我们求出了对偶问题最优解 α^*，便可计算出原始问题最优解 (w^*,b^*)，计算过程如下：

由 KKT 条件中的第 1 个等式，可推导出：

$$w^* = \sum_{i=1}^{m} \alpha_i^* y_i x_i$$

其中至少存在一个 $\alpha_j^* > 0$，否则 $w^* = 0$，而它不是原始问题的解。

若某 α_j^* 满足 $\alpha_j^* > 0$，由 KKT 条件中的第 3 个等式，可推导出：

$$y_j (w^{*\mathrm{T}} x_j + b^*) = 1$$

其中与 α_j^* 对应的 x_j 被称为支持向量。图 6-2 中两个实心的（实例）圆点为支持向量，它们在间隔边界上（函数间隔为 1）。

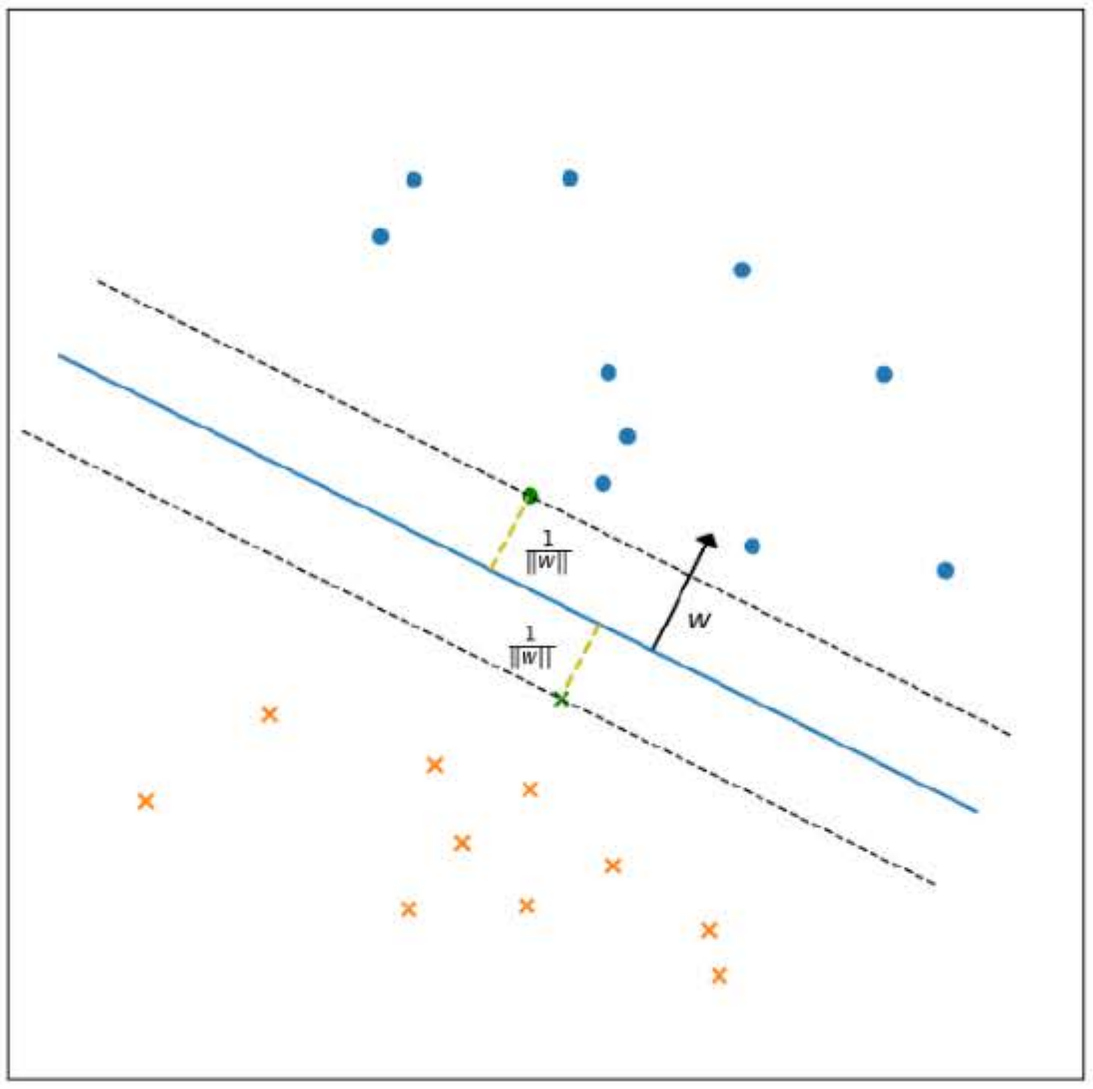

图 6-2

b^* 可由任意一个支持向量 ($\alpha_j^* > 0$) 所对应的等式求出（等式两端同乘以 y_j，且有 $y_j^2 = 1$）：

$$\begin{aligned} b^* &= y_j - w^{*\mathrm{T}} x_j \\ &= y_j - \sum_{i=1}^{m} \alpha_i^* y_i (x_i^{\mathrm{T}} x_j) \end{aligned}$$

6.1.4 分类决策函数

如果找到了数据集 D 的最大间隔分离超平面 (w^*, b^*)，便可以用其构建分类器：根据输入实例 x 位于超平面 (w^*, b^*) 的哪一侧，将 x 判为正例或负例。

分类决策函数为：

$$h(x) = sign(w^{*\mathrm{T}}x + b^*) = \begin{cases} +1, & \text{如果} w^{*\mathrm{T}}x + b^* > 0 \\ -1, & \text{如果} w^{*\mathrm{T}}x + b^* < 0 \end{cases}$$

6.1.5 线性可分支持向量机算法

最后总结一下线性可分支持向量机算法。

假设数据集 $\boldsymbol{D}$ 有 m 个样本，其中 $x_i \in \mathbf{R^n}, y \in \{+1, -1\}$。线性可分支持向量机算法如下：

（1）构造并求解约束最优化问题：

$$\min_{\alpha} \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j (x_i^{\mathrm{T}} x_j) - \sum_{i=1}^{m}\alpha_i$$

$$s.t. \quad \sum_{i=1}^{m}\alpha_i y_i = 0$$

$$\alpha_i \geqslant 0, \qquad i = 1, 2, \ldots, m$$

求得最优解 α^*。

（2）计算原始问题最优解的 w^*：

$$w^* = \sum_{i=1}^{m}\alpha_i^* y_i x_i$$

并选择任意 $a_j^* > 0$，计算原始问题最优解的 b^*：

$$b^* = y_j - \sum_{i=1}^{m}\alpha_i^* y_i (x_i^{\mathrm{T}} x_j)$$

（3）构造分类决策函数：

$$h(x) = sign(w^{*\mathrm{T}}x + b^*)$$

$$= sign(\sum_{i=1}^{m}\alpha_i^* y_i (x_i^{\mathrm{T}} x) + b^*)$$

其中：

$$sign(z) = \begin{cases} +1, & \text{如果} z > 0 \\ -1, & \text{如果} z < 0 \end{cases}$$

6.2 线性支持向量机

在现实问题中，数据集通常是线性不可分的。请看图 6-3 中的数据集，其中包含少量特异点（图中标示出的）使得数据集线性不可分，如果将特异点去掉，剩余数据子集依然是线性可分的。本节介绍的线性支持向量机是在线性可分支持向量机的基础上稍作改进得到的，它可以使用线性不可分数据集进行训练。

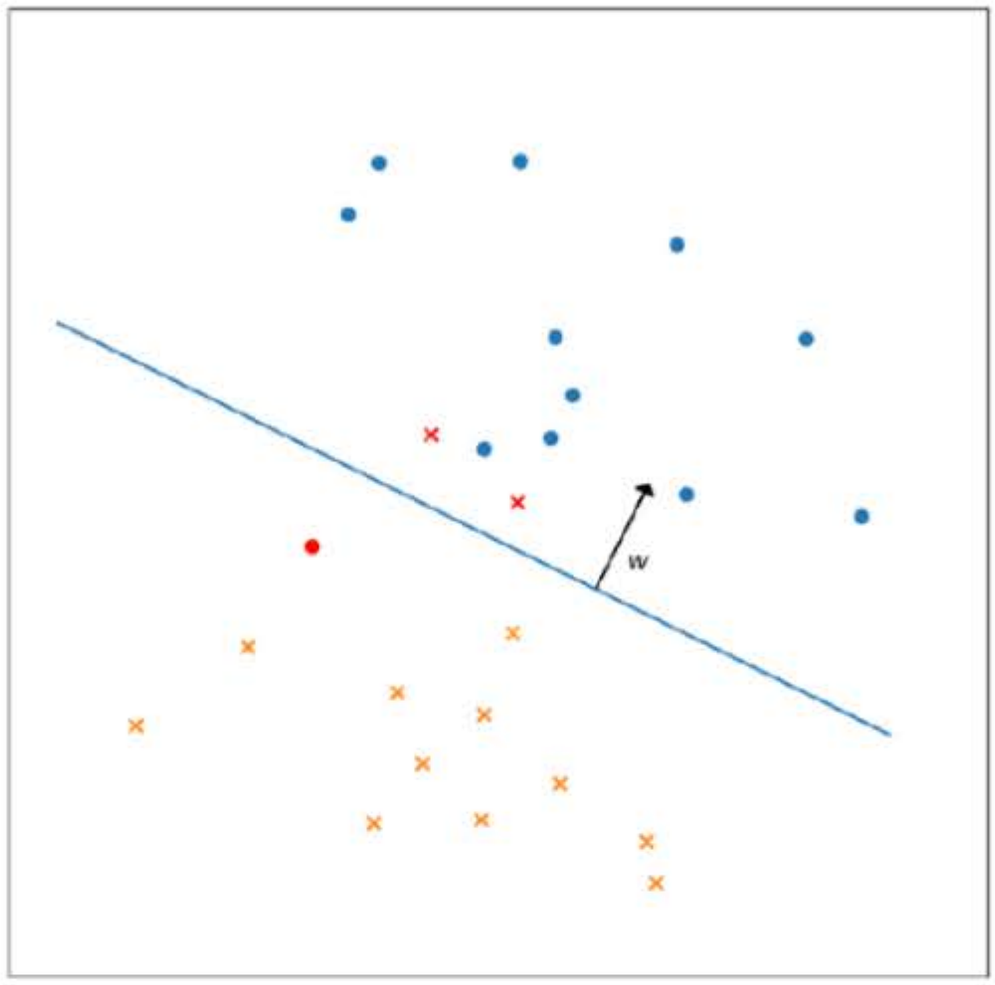

图 6-3

6.2.1 软间隔最大化

对于图 6-3 中的线性不可分数据集，无法找到一个超平面使得其中所有样本点都满足：

$$y_j(w^{\mathrm{T}}x_j + b) \geqslant 1$$

为了解决这个问题，线性支持向量机对每个样本点引进松弛变量$\xi_i \geqslant 0$，放宽约束条件：

$$y_j(w^{\mathrm{T}}x_j + b) \geqslant 1 - \xi_i$$

为了使这种放宽适度，需要对每一个 ξ_i 进行一个代价为 $C\xi_i$ 的“惩罚”。其中的 $C > 0$ 称为惩罚系数，其大小视具体应用问题而定，C 越大，对于错误分类的惩罚越重。

线性支持向量机挑选分离超平面的准则称为使“软间隔最大化”，其原始约束最优化问题为：

$$\begin{aligned}
&\min_{w,b,\xi} \ \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{m}\xi_i \\
&s.t. \quad y_i\left(w^{\mathrm{T}}x_i + b\right) \geqslant 1-\xi_i \qquad i=1,2,\ldots,m \\
&\qquad \xi_i \geqslant 0 \qquad\qquad\qquad\qquad i=1,2,\ldots,m
\end{aligned}$$

作广义格朗日函数：

$$L(w,b,\xi,\alpha,\mu) = \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{m}\xi_i + \sum_{i=1}^{m}\alpha_i(1-\xi_i - y_i\left(w^{\mathrm{T}}x_i + b\right)) + \sum_{i=1}^{m}\mu_i(-\xi_i)$$

其中，α_i 和 μ_i 为拉格朗日乘子，且有 $\alpha_i \geqslant 0$，$\mu_i \geqslant 0$。

利用拉格朗日对偶性可得到原始问题的对偶问题为（此处省略推导过程）：

$$\begin{aligned}
&\min_{\alpha} \ \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j (x_i^{\mathrm{T}}x_j) - \sum_{i=1}^{m}\alpha_i \\
&s.t. \quad \sum_{i=1}^{m}\alpha_i y_i = 0 \\
&\qquad C-\alpha_i-\mu_i = 0, \\
&\qquad \alpha_i \geqslant 0, \\
&\qquad \mu_i \geqslant 0, \qquad\qquad i=1,2,\ldots,m
\end{aligned}$$

根据约束条件中的第二个等式有 $\mu_i = C-\alpha_i$，因此可将 μ_i 消去。最终对偶问题变为：

$$\begin{aligned}
&\min_{\alpha} \ \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j (x_i^{\mathrm{T}}x_j) - \sum_{i=1}^{m}\alpha_i \\
&s.t. \quad \sum_{i=1}^{m}\alpha_i y_i = 0 \\
&\qquad 0 \leqslant \alpha_i \leqslant C, \qquad\qquad i=1,2,\ldots,m
\end{aligned}$$

此时，原始问题最优解 (w^*,b^*) 和对偶 α^* 满足 KKT 条件：

$$\nabla_w L(w^*,b^*,\xi^*,\alpha^*,\mu^*) = w^* - \sum_{i=1}^{m}\alpha_i^* y_i x_i = 0$$

$$\nabla_b L(w^*,b^*,\xi^*,\alpha^*,\mu^*) = -\sum_{i=1}^{m}\alpha_i^* y_i = 0$$

$$\nabla_{\xi} L(w^*, b^*, \xi^*, \alpha^*, \mu^*) = C - \alpha^* - \mu^* = 0$$
$$\alpha_i^* (y_i (w^{*\mathrm{T}} x_i + b^*) - 1 + \xi_i^*) = 0$$
$$\mu_i^* \xi_i^* = 0$$
$$y_i (w^{*\mathrm{T}} x_i + b^*) - 1 + \xi_i^* \geqslant 0$$
$$\xi_i^* \geqslant 0$$
$$\alpha_i^* \geqslant 0$$
$$\mu_i^* \geqslant 0 \qquad i = 1, 2, \ldots, m$$

由 KKT 条件中的第 1 个等式，可推导出：

$$w^* = \sum_{i=1}^{m} \alpha_i^* y_i x_i$$

若某 α_j^* 满足 $0 < \alpha_j^* < C$，由 KKT 条件中的第 3 个和第 5 个等式可推导出 $\xi_j = 0$，再由第 4 个等式可推导出：

$$b^* = y_j - \sum_{i=1}^{m} \alpha_i^* y_i (x_i^{\mathrm{T}} x_j)$$

可以看出 (w^*, b^*) 计算公式与之前线性可分支持向量机中的计算公式完全相同。计算 b^* 时依然需要使用支持向量。线性支持向量机的支持向量不仅仅位于间隔边界上（函数间隔为 1），也可能位于间隔边界与分离超平面之间，甚至位于分离超平面误分类的一侧，图 6-4 所示的中间带的 5 个点（实例）都是支持向量。当 $0 < \alpha_j^* < C$ 时，相应 $\xi_j = 0$，支持向量 x_j 位于间隔边界上，使用这样的支持向量可计算出 b^*（如上式）。

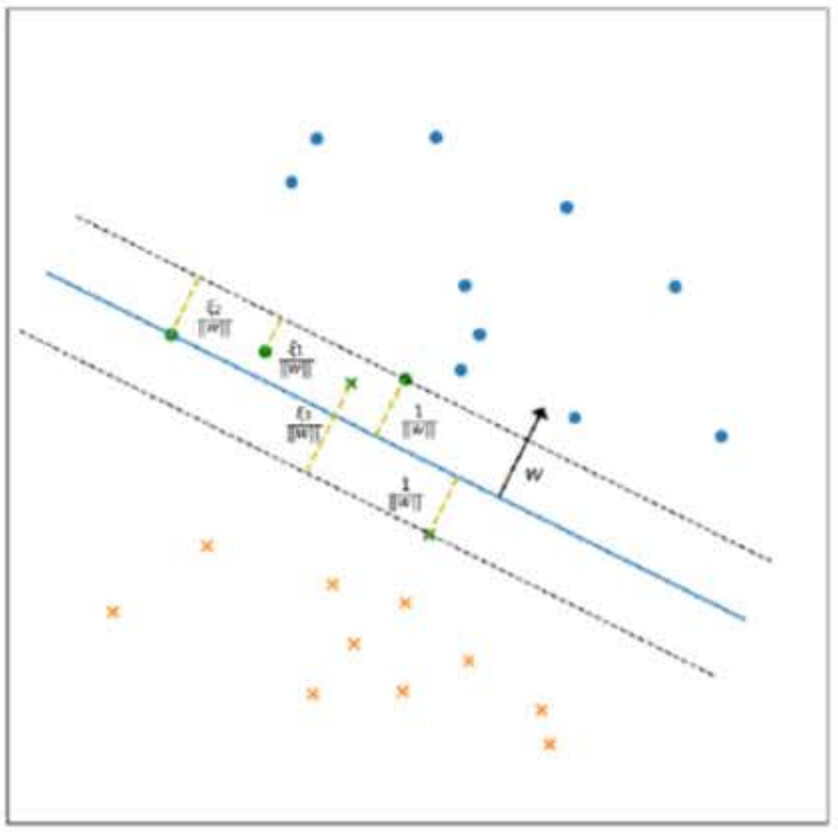

图 6-4

6.2.2 线性支持向量机算法

总结一下线性支持向量机算法。

假设数据集 $\boldsymbol{D}$ 有 m 个样本，其中$x_i \in \mathbf{R}^n, y \in \{+1, -1\}$。线性支持向量机算法如下：

（1）选取适当惩罚系数 $\boldsymbol{C}$，构造并求解约束最优化问题：

$$\min_{\alpha} \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j (x_i^{\mathrm{T}} x_j) - \sum_{i=1}^{m}\alpha_i$$

$$s.t. \quad \sum_{i=1}^{m}\alpha_i y_i = 0$$

$$0 \leqslant \alpha_i \leqslant C, \qquad i = 1, 2, \ldots, m$$

求得最优解 α^*。

（2）计算原始问题最优解的 w^*：

$$w^* = \sum_{i=1}^{m}\alpha_i^* y_i x_i$$

并选择任意 $0 < \alpha_j^* < C$，计算原始问题最优解的 b^*：

$$b^* = y_j - \sum_{i=1}^{m}\alpha_i^* y_i (x_i^{\mathrm{T}} x_j)$$

（3）构造分类决策函数：

$$h(x) = sign(w^{*\mathrm{T}} x + b^*)$$

$$= sign(\sum_{i=1}^{m}\alpha_i^* y_i (x_i^{\mathrm{T}} x) + b^*)$$

其中：

$$sign(z) = \begin{cases} +1, & \text{如果} z > 0 \\ -1, & \text{如果} z < 0 \end{cases}$$

6.3 非线性支持向量机

6.3.1 空间变换

之前讲过的两种支持向量机只能用于处理线性分类问题，但有时我们还会面对非线性分类问题。请看图 6-5 中的数据集，在 $\mathbf{R}^2$ 空间中，我们无法用一条直线（线性）将该数据集中的正例和负例正确地分隔开，但可以用一条圆形曲线（非线性）将它们分隔开。同样，$\mathbf{R}^n$ 空间中的数据集也有类似情况：不能用超平面进行分类，但可以用超曲面进行分类。这种使用非线性模型（超曲面）进行分类的问题称为非线性分类问题。

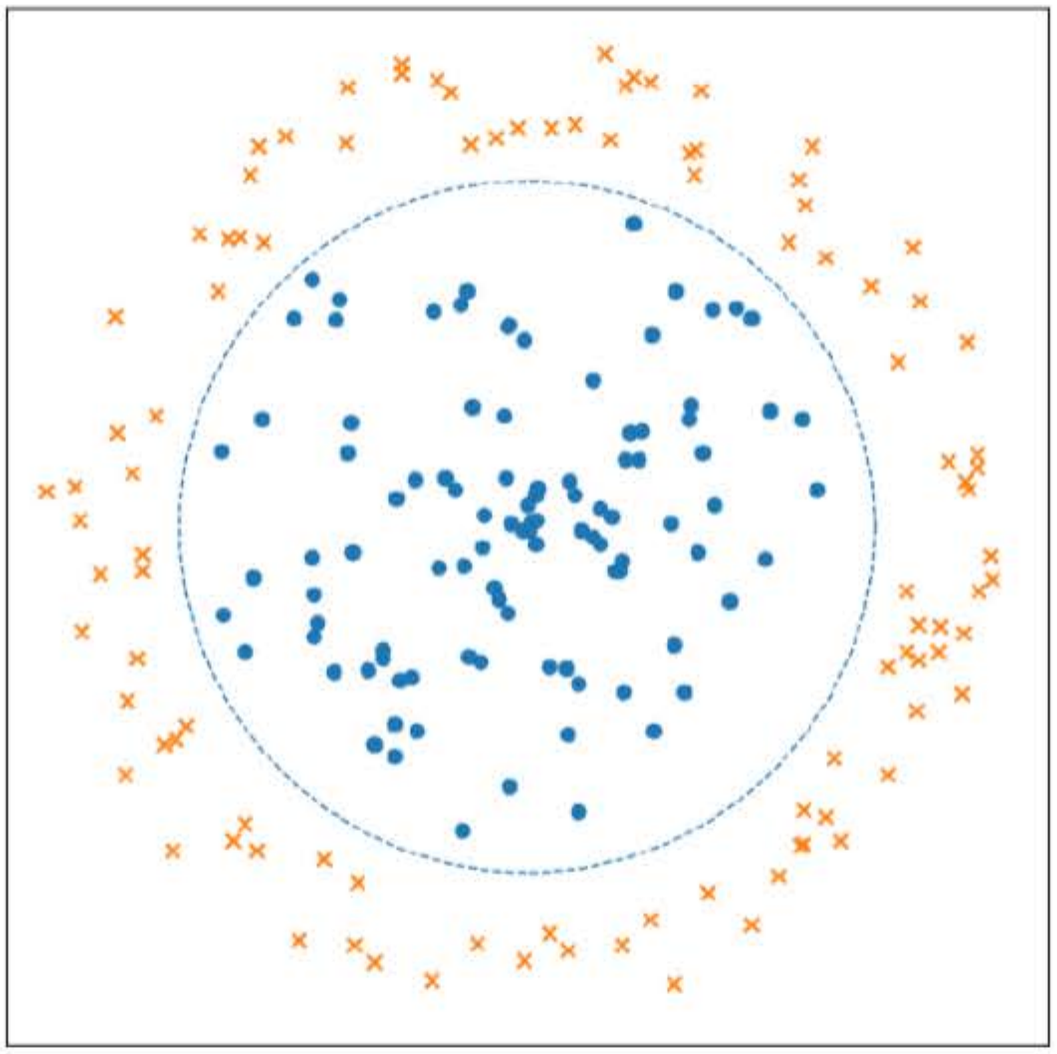

图 6-5

求解分离超曲面往往要比求解分离超平面困难很多，因此面对一个非线性分类问题时，我们希望能将其转化为一个线性分类问题，从而降低求解难度。转化问题的方法是使用某种非线性变换 ϕ，将原来空间 x 中的数据集映射到空间 H（通常是更高维的）中。以图 6-5 中的数据集为例，令非线性变换函数为：

$$\phi(x) = (x_1, x_2, x_1^2 + x_2^2)$$

变换 ϕ 为原数据 x 增加了一个维度，大小为 $\|x\|^2$。映射后的数据集在 $\mathbf{R}^3$ 空间中的情形如图 6-6 所示。

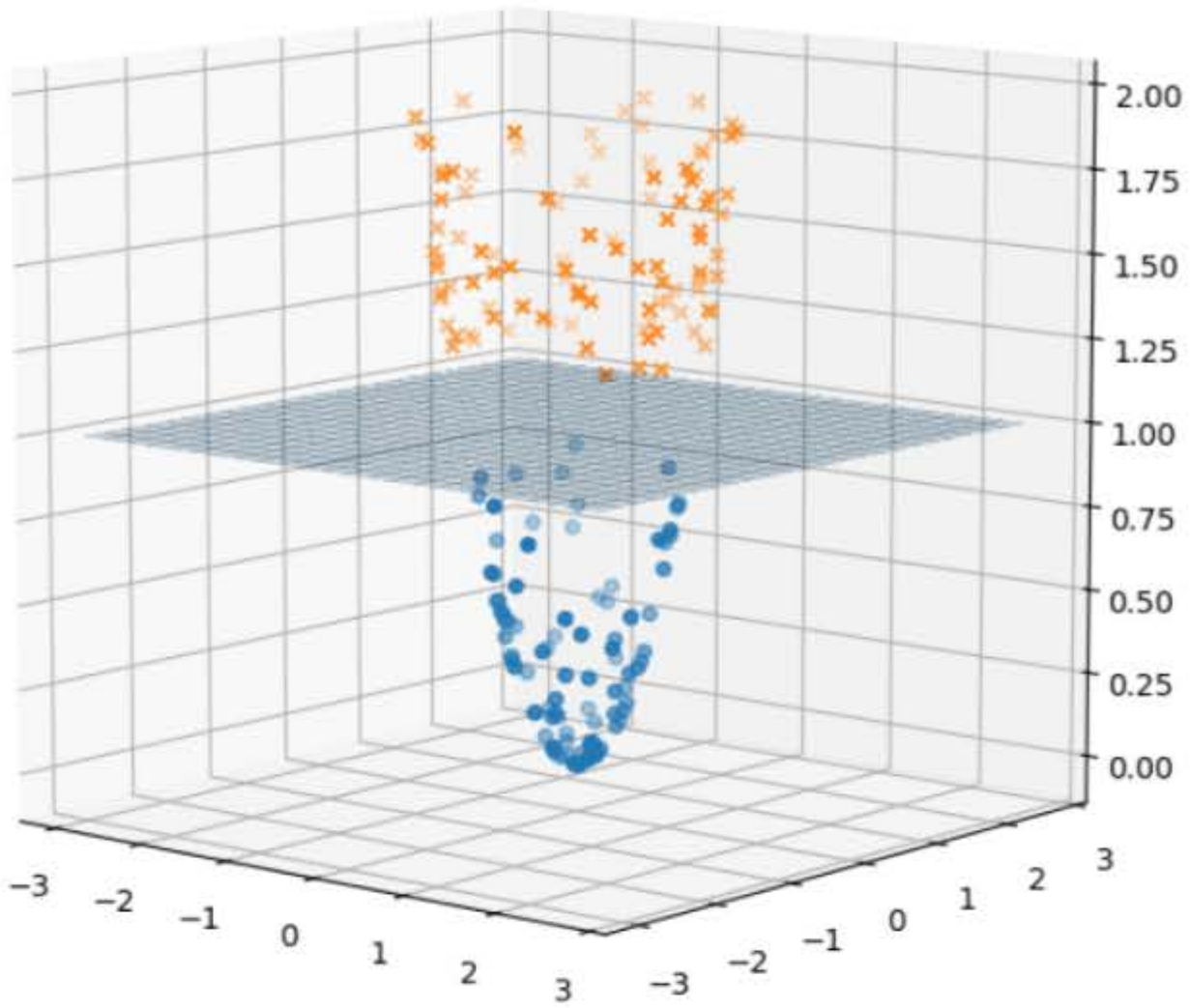

图 6-6

可以看出，映射到 $\mathbf{R}^3$ 空间后，原非线性分类问题变成了线性分类问题，此时再应用线性支持向量机算法，便可求得图中的最优分离超平面。

6.3.2 核技巧

虽然我们可以利用非线性变换处理非线性分类问题，但使用非线性变换时又面临新的问题：如果映射后的空间 H 维度非常高（甚至是无限维），将导致进行非线性变换所使用的存储空间和计算资源开销过大，有时甚至是无法实现的。然而，利用核技巧，可以解决这个问题。

请回顾线性支持向量机算法，可以发现无论是求解对偶问题还是使用模型预测，实际上只需要使用实例间的内积 $\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x}_j$，而并不需要单独使用每一个实例 $\boldsymbol{x}_i$，这就意味着针对每一个实例 $\boldsymbol{x}_i$ 进行线性变换 $\boldsymbol{\phi}(\boldsymbol{x}_i)$ 并不是必需的。核技巧的核心思想是，利用核函数直接计算映射到空间 H 后实例间的内积，以此替代先作映射再计算内积。下面给出核函数的定义。

假设对于非线性变换函数 $\boldsymbol{\phi}$，有函数 $\boldsymbol{K}$ 使得：

$$\boldsymbol{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{\phi}(\boldsymbol{x}_i)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_j)$$

则称 $\boldsymbol{K}$ 为核函数，它的值等于将向量 $\boldsymbol{x}_i$ 和 $\boldsymbol{x}_j$ 使用函数 $\boldsymbol{\phi}$ 映射至 H 后，两个新向量的内积。

通常直接计算 $K(x_i, x_j)$ 开销比较小，因此我们使用它来避免开销巨大的线性变换运算 $\phi(x)$。另外需要说明的是，如果先给定核函数 K，可能存在多个满足条件的 ϕ 和 H。在实际应用中，一般根据经验直接选择核函数，核函数隐式地定义了特征空间 H，我们不必关心 ϕ 和 H 具体是什么以及核函数是否有效通过实验验证。关于函数 K 需满足怎样的条件才是核函数，在此不再深入讨论。

以下是几种常用的核函数。

（1）多项式核

$$K(x_i, x_j) = (x_i^{\mathrm{T}} x_j)^d$$

其中 $d \geqslant 1$，为多项式的次数。

（2）高斯核

$$K(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right)$$

其中 $\sigma > 0$，为高斯核的带宽。

（3）拉普拉斯核

$$K(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|}{\sigma}\right)$$

其中 $\sigma > 0$。

（4）Sigmoid 核

$$K(x_i, x_j) = \tanh(\beta x_i^{\mathrm{T}} x_j + \theta)$$

其中 tanh 为双曲正切函数，$\beta > 0$, $\theta < 0$。

6.3.3 非线性支持向量机算法

在线性支持向量机算法的基础上，只需使用 $K(x_i, x_j)$ 替代 $x_i^{\mathrm{T}} x_j$ 便可得到非线性支持向量机算法。

假设数据集 D 有 m 个样本，其中 $x_i \in \mathbf{R}^{\mathbf{n}}, y \in \{+1, -1\}$。非线性支持向量机算法如下：

（1）选取适当惩罚系数 C 以及核函数 K，构造并求解约束最优化问题：

$$\min_{\alpha} \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j K(x_i,x_j) - \sum_{i=1}^{m}\alpha_i$$

$$s.t. \quad \sum_{i=1}^{m}\alpha_i y_i = 0$$

$$0 \leqslant \alpha_i \leqslant C, \qquad i = 1,2,\ldots,m$$

求得最优解 α^*。

（2）选择任意 $0 < \alpha_j^* < C$，计算原始问题最优解的 b^*：

$$b^* = y_j - \sum_{i=1}^{m}\alpha_i^* y_i K(x_i,x_j)$$

（3）构造分类决策函数：

$$h(x) = sign(\sum_{i=1}^{m}\alpha_i^* y_i K(x_i,x) + b^*)$$

其中：

$$sign(z) = \begin{cases} +1, & \text{如果} z > 0 \\ -1, & \text{如果} z < 0 \end{cases}$$

实际上，可将线性支持向量机算法视为核函数取 $K(x_i,x_j) = x_i^{\mathrm{T}}x_j$ 时的非线性支持向量机算法。

6.4 SMO 算法

本节我们学习一种实现支持向量机的具体算法，它被称为序列最小最优化算法（SMO）。回顾非线性支持向量机的约束最优化问题：

$$\min_{\alpha} \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j K(x_i,x_j) - \sum_{i=1}^{m}\alpha_i$$

$$s.t. \quad \sum_{i=1}^{m}\alpha_i y_i = 0$$

$$0 \leqslant \alpha_i \leqslant C, \qquad i = 1,2,\ldots,m$$

它是一个具有全局最优解的凸二次规划问题，虽然有很多优化算法可以求解该问题，但在训练集很大时，这些算法需要很长的训练时间，以致无法使用。SMO 算法是一种快速实现算法，使用它可提升训练速度。

约束最优化问题的变量为 m 个拉格朗日乘子$\alpha_1, \alpha_2, \ldots, \alpha_m$。SMO 算法的核心思想是：每次挑选两个变量 α_i, α_j，固定其他变量不动，针对 α_i, α_j 进行优化，且使它们满足 KKT 条件。每进行一次优化，$\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_m)$ 将越接近最优解 α^*。反复执行优化过程，直至所有变量都满足 KKT 条件时，最优解 α^* 便得到了。

6.4.1 两个变量最优化问题的求解

下面我们讨论两个变量的约束最优化问题的求解。

假设在某一次优化过程中，选择针对优化的变量为 α_1, α_2，固定其他变量 $\alpha_3, \alpha_4, \ldots, \alpha_m$不动（这里选择 α_1, α_2 只是为了阐述方便，求解并不具有特殊性，选择任意 α_i, α_j 方法是相同的）。

因为只针对 α_1, α_2 进行优化，所以其他变量为常数。可将约束最优化问题中不包含 α_1, α_2 的常数项去掉（最优解不变），问题变为：

$$\min_{\alpha_1,\alpha_2} W(\alpha_1,\alpha_2) = \frac{1}{2}y_1^2\alpha_1^2K_{11} + \frac{1}{2}y_2^2\alpha_2^2K_{22} + y_1y_2\alpha_1\alpha_2K_{12} + y_1\alpha_1\sum_{i=3}y_i\alpha_iK_{1i} + y_2\alpha_2\sum_{i=3}y_i\alpha_iK_{2i} - (\alpha_1+\alpha_2)$$

$$s.t. \quad \sum_{i=1}^{m}\alpha_i y_i = 0$$

$$0 \leqslant \alpha_i \leqslant C, \qquad i = 1,2,\ldots,m$$

其中，$K_{ij} = K(x_i, x_j)$。

由第一个约束条件 $\sum_{i=1}^{m}\alpha_i y_i = 0$ 可推导出：

$$y_1\alpha_1 + y_2\alpha_2 = -\sum_{i=3}y_i\alpha_i$$

其他变量是固定不动的，因此上式右端为常数，第一个约束条件也可写成：

$$y_1\alpha_1 + y_2\alpha_2 = \epsilon$$

上式表明 (α_1, α_2) 位于一条直线上，又因为 $y_i \in \{+1, -1\}$，所以直线的斜率只能为+1 （$y_1 \neq y_2$ 时）或-1（$y_1 = y_2$时）。再由第二个约束条件 $0 \leqslant a_i \leqslant C$ 可知，(α_1, α_2)在$[0, C] \times [0, C]$ 的方形区域内。综合两个约束条件分析得出，(α_1, α_2) 可取值位于图 6-7 所示的线段上。

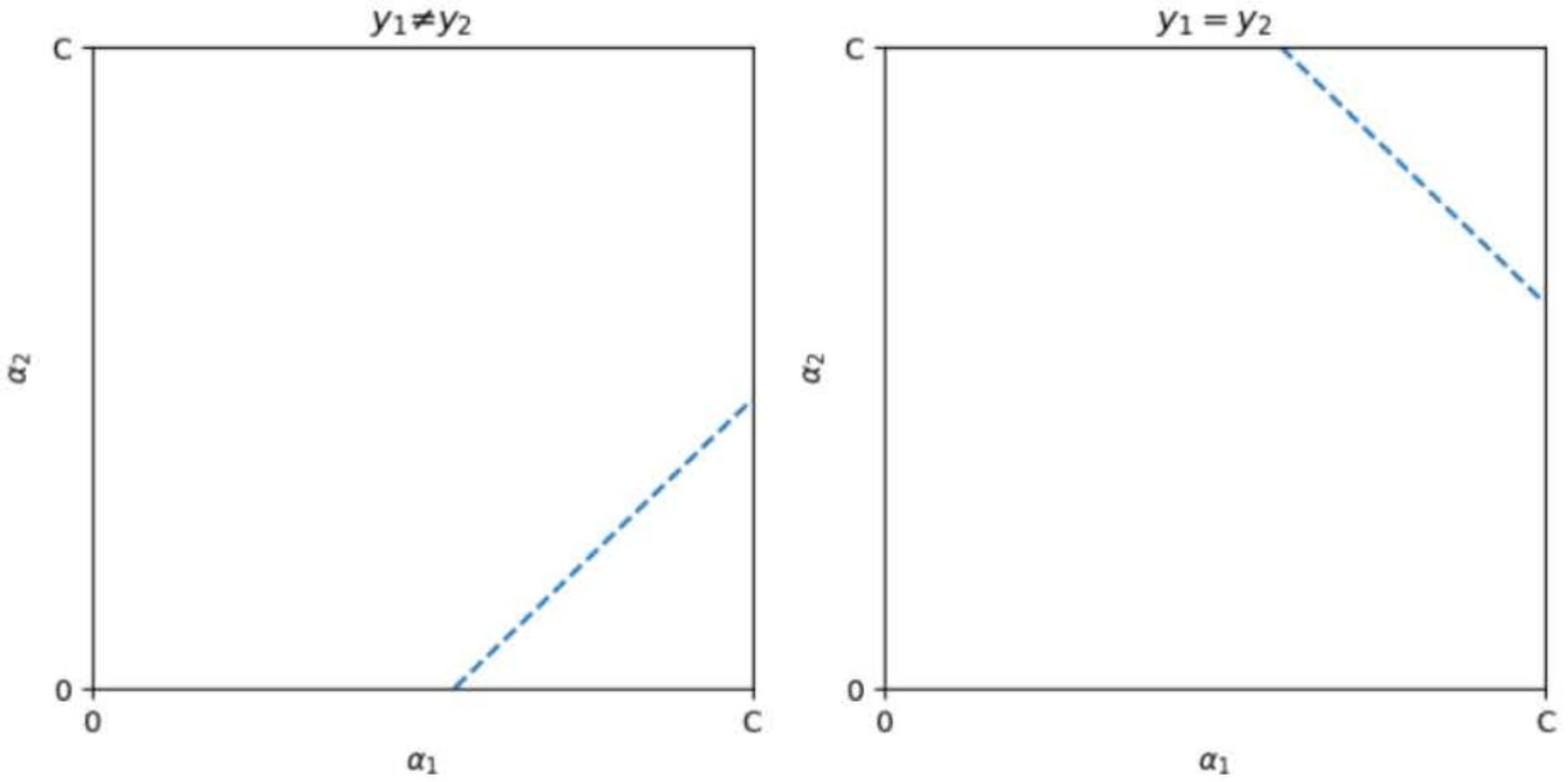

图 6-7

对于以上优化问题，我们可以先忽略第二个约束条件 $0 \leqslant a_i \leqslant C$（即去掉方形边界的约束，让 (α_1, α_2) 在直线上任意移动），求出其中一个变量的解 $\alpha_2^{new,unc}$，再根据第二个约束条件对 $\alpha_2^{new,unc}$ 裁剪得到 α_2^{new}，有了 α_2^{new}，便可根据第一个约束条件计算出 α_1^{new}，最终得到优化问题的解 $(\alpha_1^{new}, \alpha_2^{new})$。下面叙述具体求解过程。

（1）求解 $\alpha_2^{new,unc}$

在第一个约束条件下求目标函数的极值点，得到 $\alpha_2^{new,unc}$。

利用拉格朗日乘数法，作拉格朗日函数：

$$L(\alpha_1, \alpha_2, \lambda) = W(\alpha_1, \alpha_2) + \lambda(y_1\alpha_1 + y_2\alpha_2 - \epsilon)$$

求各偏导数，并令其为 0，得：

$$\begin{cases} \frac{\partial L}{\partial \alpha_1} = \alpha_1 K_{11} + y_1 y_2 \alpha_2 K_{12} + y_1 \sum_{i=3} y_i \alpha_i K_{1i} - 1 + \lambda y_1 = 0 & \text{（1 式）} \\ \frac{\partial L}{\partial \alpha_2} = \alpha_2 K_{22} + y_1 y_2 \alpha_1 K_{12} + y_2 \sum_{i=3} y_i \alpha_i K_{2i} - 1 + \lambda y_2 = 0 & \text{（2 式）} \\ \frac{\partial L}{\partial \lambda} = y_1 \alpha_1 + y_2 \alpha_2 - \epsilon = 0 & \text{（3 式）} \end{cases}$$

为简化，令：

$$v_1 = \sum_{i=3} y_i \alpha_i K_{1i}$$

$$v_2 = \sum_{i=3} y_i \alpha_i K_{2i}$$

经观察可发现 v_1, v_2 不包含 α_1, α_2，为常数。

1 式两端乘以y_1，2 式两端乘以y_2，再由 $y_i^2 = 1$ 得：

$$\begin{cases} y_1\alpha_1 K_{11} + y_2\alpha_2 K_{12} + v1 - y_1 + \lambda = 0 \\ y_2\alpha_2 K_{22} + y_1\alpha_1 K_{12} + v2 - y_2 + \lambda = 0 \\ y_1\alpha_1 + y_2\alpha_2 - \epsilon = 0 \end{cases}$$

1 式减 2 式，并根据 3 式消除其中的 $y_1\alpha_1$ 得：

$$(v_1 - y_1) - (v_2 - y_2) + \epsilon K_{11} - \epsilon K_{12} = y_2\alpha_2(K_{11} + K_{22} - 2K_{12})$$

在以上等式中，除了α_2之外都为常数，因此 $\alpha_2^{new,unc}$ 可由该等式计算，但解的形式比较复杂，不便于算法描述。我们进一步整理，使得解的形式更加简洁。

在优化过程中，由当前 α 求得的分离超平面方程为 $g(x) = 0$，则：

$$g(x) = \sum_{i=1}^{m} \alpha_i y_i K(x_i, x) + b$$

设：

$$E_i = E(x_i) = g(x_i) - y_i, \quad i = 1, 2$$

E_i 可理解 x_i 的预测值 $g(x_i)$ 与实际值 y_i 之差。

再设 α_1, α_2 优化前的值为 $\alpha_1^{old}, \alpha_2^{old}$，根据第一个约束条件，它们同样满足：

$$y_1\alpha_1^{old} + y_2\alpha_2^{old} = \epsilon$$

再观察 v_1, v_2 的形式，可发现：

$$v_1 = g(x_1) - b - y_1\alpha_1^{old} K_{11} - y_2\alpha_2^{old} K_{12}$$
$$v_2 = g(x_2) - b - y_1\alpha_1^{old} K_{12} - y_2\alpha_2^{old} K_{22}$$

将以上 3 个等式代回之前的等式，得：

$$\begin{aligned} &(g(x_1) - b - y_1\alpha_1^{old} K_{11} - y_2\alpha_2^{old} K_{12} - y_1) \\ &- (g(x_2) - b - y_1\alpha_1^{old} K_{12} - y_2\alpha_2^{old} K_{22} - y_2) \\ &+ y_1\alpha_1^{old} K_{11} + y_2\alpha_2^{old} K_{11} - y_1\alpha_1^{old} K_{12} - y_2\alpha_2^{old} K_{12} = y_2\alpha_2(K_{11} + K_{22} - 2K_{12}) \end{aligned}$$

经整理，得到：

$$y_2\alpha_2^{old}(K_{11} + K_{22} - 2K_{12}) + (E_1 - E_2) = y_2\alpha_2(K_{11} + K_{22} - 2K_{12})$$

最终求得形式简洁的 $\alpha_2^{new,unc}$：

$$\alpha_2^{new,unc} = \alpha_2^{old} + \frac{y_2(E_1 - E_2)}{K_{11} + K_{22} - 2K_{12}}$$

（2）求解 α_2^{new}

对 $\alpha_2^{new,unc}$ 裁剪得到 α_2^{new}。

根据第二个约束条件，可推导出 α_2^{new} 需满足：

$$L \leqslant \alpha_2^{new} \leqslant H$$

其中 L 和 H 为直线与方形边界交点处 α_2 的值。L 和 H可根据 $\alpha_1^{old}, \alpha_2^{old}$进行计算。

当 $y_1 \neq y_2$ 时：

$$L = \max(0, \alpha_2^{old} - \alpha_1^{old})$$
$$H = \min(C, \alpha_2^{old} - \alpha_1^{old} + C)$$

当 $y_1 = y_2$ 时：

$$L = \max(0, \alpha_1^{old} + \alpha_2^{old} - C)$$
$$H = \min(C, \alpha_1^{old} + \alpha_2^{old})$$

如果 $\alpha_2^{new,unc}$ 不在 $[L, H]$ 范围内（不在线段上），则优化问题的解位于边界处。最终得出：

$$\alpha_2^{new} = \begin{cases} L, & \text{如果} \alpha_2^{new,unc} < L \\ \alpha_2^{new,unc}, & \text{如果} L \leqslant \alpha_2^{new,unc} \leqslant H \\ H, & \text{如果} \alpha_2^{new,unc} > H \end{cases}$$

（3）求解 α_1^{new}

通过 α_2^{new} 和第一个约束条件计算α_1^{new}。

根据第一个约束条件，有：

$$y_1\alpha_1^{new} + y_2\alpha_2^{new} = y_1\alpha_1^{old} + y_2\alpha_2^{old}$$

等式两端同乘以 y_1，整理得：

$$\alpha_1^{new} = \alpha_1^{old} + y_1y_2(\alpha_2^{old} - \alpha_2^{new})$$

6.4.2 变量选择

我们已经了解了如何针对已选定的两个变量进行优化，下面再来讨论如何选择两个优化变量。

SMO 算法的最终目标是使所有变量都满足KKT 条件，因此每次优化所选择的两个变量，其中至少有一个变量是违反以下 KKT 条件的：

$$\begin{cases} y_i g(x_i) \geqslant 1, & \text{如果} \alpha_i = 0 \\ y_i g(x_i) = 1, & \text{如果} 0 < \alpha_i < C \\ y_i g(x_i) \leqslant 1, & \text{如果} \alpha_i = C \end{cases}$$

上式中 (x_i, y_i) 为 α_i 对应的样本，$g(x_i) = \sum_{k=1}^{m} \alpha_k y_k K_{ik} + b$。

SMO 算法的一轮优化（包含多次）由两重循环构成，外层循环选择第一个变量 α_1，它必须是违反 KKT 条件的；内层循环根据已选 α_1 选择第二个变量 α_2；然后尝试对 α_1, α_2 进行优化。大体流程如下：

```
1.  SMO 算法的一轮优化：
2.      外层循环：
3.          每次选择一个违反 KKT 条件的 alpha_1
4.          内层循环：
5.              每次根据已选 alpha_1，选择一个 alpha_2，并尝试对 alpha_1、alpha_2
进行优化
6.              IF 优化成功：
7.                  跳出内层循环，继续外层循环，迭代下一个 alpha_1
8.              ELSE:
9.                  继续内层循环，迭代下一个 alpha_2
```

选择第一个变量的外层循环，优先遍历所有非边界的 α_k（满足 $0 < \alpha_k < C$），其中违反 KKT 条件的可被选作 α_1；如果所有非边界 α_k 都满足 KKT 条件，则遍历所有的 α_k，其中违反 KKT 条件的可被选作 α_1。选择第一个变量的过程可描述为：

（1）遍历所有非边界的 α_k 依次判断它们是否违反 KKT 条件，违反的作为 α_1。

（2）当方法 1 无效时，遍历所有的 α_k，依次判断它们是否违反 KKT 条件，违反的作为 α_1。

选择第二个变量的内层循环，在外层循环已选出 α_1 的情况下，应尽量使 α_2 在优化后有足够大的变化。由之前推导出的 α_2 更新公式可看出，α_2 的变化大小与 $|E_1 - E_2|$ 大小相关，因此在 E_1 已确定（因为 α_1 已确定）的情况下，优先选择使得 $|E_1 - E_2|$ 最大的 α_2。在特殊情况下，上述方法选择的 α_2 不能使目标函数有足够的下降，还需在所有非边界或所有的 α_k 中继续寻找一个能使目标函数有足够下降的 α_2。 选择第二个变量的过程可描述为：

（1）根据 α_1 计算 E_1，寻找使得 $|E_1 - E_2|$ 最大的 α_2。

（2）当方法 1 无效时，遍历所有非边界的 α_k，依次作为 α_2 进行尝试，直到目标函数有足够的下降。

（3）当方法 1 和方法 2 都无效时，遍历所有的α_k，依次作为α_2 进行尝试，直到目标函数有足够的下降。

（4）当以上方法都无效时，放弃当前α_1。

6.4.3 更新 b

每一次针对 α_1, α_2 成功优化后，都需要重新计算 b。b 的计算依然由以下 KKT 条件确立：

$$\begin{cases} y_i g(x_i) \geqslant 1, & \text{如果} \alpha_i = 0 \\ y_i g(x_i) = 1, & \text{如果} 0 < \alpha_i < C \\ y_i g(x_i) \leqslant 1, & \text{如果} \alpha_i = C \end{cases}$$

我们根据 α_i^{new} 是否位于边界，分情况进行讨论。

一种情况是，$\alpha_1^{new}, \alpha_2^{new}$ 至少有一个不位于边界上。

如果 $0 < \alpha_1^{new} < C$，KKT 条件为：

$$y_1 g(x_1) = 1$$

上式两端乘以 y_1 并整理，得出：

$$E_1^{new} = g(x_1) - y_1 = 0$$

再将 $g(x_1) = \sum_{k=1}^{m} \alpha_k y_k K_{1k} + b$ 分别代入 E_1^{new} 和 E_1^{old} 的定义式，得：

$$E_1^{new} = \alpha_1^{new} y_1 K_{11} + \alpha_2^{new} y_2 K_{12} + \sum_{k=3}^{m} \alpha_k y_k K_{1k} + b_1^{new} - y_1$$

$$E_1^{old} = \alpha_1^{old} y_1 K_{11} + \alpha_2^{old} y_2 K_{12} + \sum_{k=3}^{m} \alpha_k y_k K_{1k} + b^{old} - y_1$$

以上两式相减并整理，可得出 b_1^{new} 的计算式为：

$$b_1^{new} = E_1^{new} - E_1^{old} - y_1 K_{11}(\alpha_1^{new} - \alpha_1^{old}) - y_2 K_{12}(\alpha_2^{new} - \alpha_2^{old}) + b^{old}$$

之前曾推导出当 $0 < \alpha_1^{new} < C$ 时，$E_1^{new} = 0$，因此：

$$b_1^{new} = -E_1^{old} - y_1 K_{11}(\alpha_1^{new} - \alpha_1^{old}) - y_2 K_{12}(\alpha_2^{new} - \alpha_2^{old}) + b^{old}$$

同理，如果 $0 < \alpha_2^{new} < C$，可得出 b_2^{new} 的计算式为：

$$b_2^{new} = -E_2^{old} - y_1 K_{21}(\alpha_1^{new} - \alpha_1^{old}) - y_2 K_{22}(\alpha_2^{new} - \alpha_2^{old}) + b^{old}$$

当 α_1^{new}，α_2^{new} 同时满足 $0 < \alpha_i^{new} < C$，则 $b_1^{new} = b_2^{new}$。

再来看另一种情况，如果α_1^{new}，α_2^{new}等于0或C，即都位于边界，KKT条件为：

$$\begin{cases} y_i g(x_i) \geqslant 1, & \text{如果} \alpha_i = 0, \\ y_i g(x_i) \leqslant 1, & \text{如果} \alpha_i = C \end{cases}$$

无论α_1^{new}，α_2^{new}取0或C的任意组合，总可根据以上不等式计算出b^{new}的范围，范围的两个边界正是b_1^{new}，b_2^{new}，也就是说b^{new}可取b_1^{new}，b_2^{new}之间的任意值，此时取两者的中点作为b^{new}：

最终，b的更新算法可描述为：

```
1.  分别计算b1_new,b2_new
2.
3.  IF 0 < alpha_1 < C:
4.      b_new = b1_new
5.  ELSE IF 0 < alpha_2 < C:
6.      b_new = b2_new
7.  ELSE:
8.      b_new = (b1_new + b2_new) / 2
```

6.4.4 更新E缓存

为加速计算SMO算法使用了一个存储E_k的缓存。对于容量较大的训练集，缓存全部E_k可能开销很大，实际应用中可只缓存非边界α_k对应的E_k，它们在优化时使用得更频繁。对于那些使用较少的边界α_k对应的E_k不进行缓存，需要时根据定义式实时计算。

首先考察E_1, E_2的更新。由于仅当$0 < \alpha_i^{new} < C$时进行缓存，且满足该条件时$E_i = 0$，因此E_1, E_2新公式为：

$$E_1 = 0$$

$$E_2 = 0$$

另外，缓存中其他有效的E_j（非边界α_j对应的E_j）受到α_1, α_2改变的影响，也需要进行更新。

E_j^{old}, E_j^{new}的定义式为：

$$E_j^{new} = \alpha_1^{new} y_1 K_{1j} + \alpha_2^{new} y_2 K_{2j} + \sum_{k=3}^{m} \alpha_k y_k K_{kj} + b^{new} - y_1$$

$$E_j^{old} = \alpha_1^{old} y_1 K_{1j} + \alpha_2^{old} y_2 K_{2j} + \sum_{k=3}^{m} \alpha_k y_k K_{kj} + b^{old} - y_1$$

以上两式相减并整理，可得出 E_j 的更新公式：

$$E_j^{new} = E_j^{old} + (b^{new} - b^{old}) + y_1 K_{1j}(\alpha_1^{new} - \alpha_1^{old}) + y_2 K_{2j}(\alpha_2^{new} - \alpha_2^{old})$$

6.5 算法实现

下面我们来实现 SMO 算法，代码如下：

```
1.  import numpy as np
2.
3.  class SMO:
4.      def __init__(self, C, tol, kernel='rbf', gamma=None):
5.          # 惩罚系数
6.          self.C = C
7.          # 优化过程中 alpha 步长的阈值
8.          self.tol = tol
9.
10.         # 核函数
11.         if kernel == 'rbf':
12.             self.K = self._gaussian_kernel
13.             self.gamma = gamma
14.         else:
15.             self.K = self._linear_kernel
16.
17.     def _linear_kernel(self, U, v):
18.         '''线性核函数'''
19.         return np.dot(U, v)
20.
21.     def _gaussian_kernel(self, U, v):
22.         '''高斯核函数'''
23.         if U.ndim == 1:
24.             p = np.dot(U - v, U - v)
25.         else:
26.             p = np.sum((U - v) * (U - v), axis=1)
27.         return np.exp(-p * self.gamma)
28.
29.     def _g(self, x):
30.         '''函数 g(x)'''
```

```
        alpha, b, X, y, E = self.args

        idx = np.nonzero(alpha > 0)[0]
        if idx.size > 0:
            return np.sum(y[idx] * alpha[idx] * self.K(X[idx], x)) +
b[0]
        return b[0]

    def _optimize_alpha_i_j(self, i, j):
        '''优化alpha_i, alpha_j'''
        alpha, b, X, y, E = self.args
        C, tol, K = self.C, self.tol, self.K

        # 优化需有两个不同alpha
        if i == j:
            return 0

        # 计算alpha[j]的边界
        if y[i] != y[j]:
            L = max(0, alpha[j] - alpha[i])
            H = min(C, C + alpha[j] - alpha[i])
        else:
            L = max(0, alpha[j] + alpha[i] - C)
            H = min(C, alpha[j] + alpha[i])

        # L == H 时已无优化空间(一个点)
        if L == H:
            return 0

        # 计算eta
        eta = K(X[i], X[i]) + K(X[j], X[j]) - 2 * K(X[i], X[j])
        if eta <= 0:
            return 0

        # 对于alpha非边界使用E缓存。 边界alpha, 动态计算E
        if 0 < alpha[i] < C:
            E_i = E[i]
        else:
```

```
            E_i = self._g(X[i]) - y[i]

        if 0 < alpha[j] < C:
            E_j = E[j]
        else:
            E_j = self._g(X[j]) - y[j]

        # 计算alpha_j_new
        alpha_j_new = alpha[j] +  y[j] * (E_i - E_j) / eta

        # 对alpha_j_new进行裁剪
        if alpha_j_new > H:
            alpha_j_new = H
        elif alpha_j_new < L:
            alpha_j_new = L
        alpha_j_new = np.round(alpha_j_new, 7)

        # 判断步长是否足够大
        if np.abs(alpha_j_new - alpha[j]) < tol * (alpha_j_new + alpha[j]
+ tol):
            return 0

        # 计算alpha_i_new
        alpha_i_new = alpha[i] + y[i] * y[j] * (alpha[j] - alpha_j_new)
        alpha_i_new = np.round(alpha_i_new, 7)

        # 计算b_new
        b1 = b[0] - E_i \
             -y[i] * (alpha_i_new - alpha[i]) * K(X[i], X[i]) \
             -y[j] * (alpha_j_new - alpha[j]) * K(X[i], X[j])

        b2 = b[0] - E_j \
             -y[i] * (alpha_i_new - alpha[i]) * K(X[i], X[j]) \
                 -y[j] * (alpha_j_new - alpha[j]) * K(X[j], X[j])

            if 0 < alpha_i_new < C:
                b_new = b1
            elif 0 < alpha_j_new < C:
```

```
105.                b_new = b2
106.            else:
107.                b_new = (b1 + b2) / 2
108.
109.            # 更新 E 缓存
110.            # 更新 E[i]，E[j]。若优化后 alpha 若不在边界，则缓存有效且值为 0
111.            E[i] = E[j] = 0
112.            # 更新其他非边界 alpha 对应的 E[k]
113.            mask = (alpha != 0) & (alpha != C)
114.            mask[i] = mask[j] = False
115.            non_bound_idx = np.nonzero(mask)[0]
116.            for k in non_bound_idx:
117.                E[k] += b_new - b[0] + y[i] * K(X[i], X[k]) *
(alpha_i_new - alpha[i]) \
118.                        + y[j] * K(X[j], X[k]) * (alpha_j_new - alpha[j])
119.
120.            # 更新 alpha_i，alpha_i
121.            alpha[i] = alpha_i_new
122.            alpha[j] = alpha_j_new
123.
124.            # 更新 b
125.            b[0] = b_new
126.
127.            return 1
128.
129.        def _optimize_alpha_i(self, i):
130.            '''优化 alpha_i， 内部寻找 alpha_j'''
131.            alpha, b, X, y, E = self.args
132.
133.            # 对于 alpha 非边界，使用 E 缓存。 边界 alpha， 动态计算 E
134.            if 0 < alpha[i] < self.C:
135.                E_i = E[i]
136.            else:
137.                E_i = self._g(X[i]) - y[i]
138.
139.            # alpha_i 仅在违反 KKT 条件时进行优化
140.            if (E_i * y[i] < -self.tol and alpha[i] < self.C) or \
141.                    (E_i * y[i] > self.tol and alpha[i] > 0):
```

```
142.                # 按优先级次序选择 alpha_j
143.
144.                # 分别获取非边界 alpha 和边界 alpha 的索引
145.                mask = (alpha != 0) & (alpha != self.C)
146.                non_bound_idx = np.nonzero(mask)[0]
147.                bound_idx = np.nonzero(~mask)[0]
148.
149.                # 优先级(-1)
150.                # 若非边界 alpha 个数大于 1，则寻找使得|E_i - E_j|
                      最大化的 alpha_j
151.                if len(non_bound_idx) > 1:
152.                    if E[i] > 0:
153.                        j = np.argmin(E[non_bound_idx])
154.                    else:
155.                        j = np.argmax(E[non_bound_idx])
156.
157.                    if self._optimize_alpha_i_j(i, j):
158.                        return 1
159.
160.                # 优先级(-2)
161.                # 随机迭代非边界 alpha
162.                np.random.shuffle(non_bound_idx)
163.                for j in non_bound_idx:
164.                    if self._optimize_alpha_i_j(i, j):
165.                        return 1
166.
167.                # 优先级(-3)
168.                # 随机迭代边界 alpha
169.                np.random.shuffle(bound_idx)
170.                for j in bound_idx:
171.                    if self._optimize_alpha_i_j(i, j):
172.                        return 1
173.
174.            return 0
175.
176.        def train(self, X_train, y_train):
177.            '''训练'''
178.            m, _ = X_train.shape
```

```
179.
180.          # 初始化向量 alpha 和标量 b
181.          alpha = np.zeros(m)
182.          b = np.zeros(1)
183.
184.          # 创建 E 缓存
185.          E = np.zeros(m)
186.
187.          # 将各方法频繁使用的参数收集到列表，供调用时传递
188.          self.args = [alpha, b, X_train, y_train, E]
189.
190.          n_changed = 0
191.          examine_all = True
192.          while n_changed > 0 or examine_all:
193.              n_changed = 0
194.
195.              # 迭代 alpha_i
196.              for i in range(m):
197.                  if examine_all or 0 < alpha[i] < self.C:
198.                      n_changed += self._optimize_alpha_i(i)
199.
200.              # 若当前迭代非边界 alpha，且没有 alpha 改变，下次迭代所有 alpha
201.              # 否则，下次迭代非边界间 alpha
202.              examine_all = (not examine_all) and (n_changed == 0)
203.
204.          # 训练完成后保存模型参数
205.          idx = np.nonzero(alpha > 0)[0]
206.          # 1.非零 alpha
207.          self.sv_alpha = alpha[idx]
208.          # 2.支持向量
209.          self.sv_X = X_train[idx]
210.          self.sv_y = y_train[idx]
211.          # 3.b.
212.          self.sv_b = b[0]
213.
214.      def _predict_one(self, x):
215.          '''对单个输入进行预测'''
216.          k = self.K(self.sv_X, x)
```

```
217.             return np.sum(self.sv_y * self.sv_alpha * k) + self.sv_b
218.
219.         def predict(self, X):
220.             '''预测'''
221.             y_pred = np.apply_along_axis(self._predict_one, 1, X)
222.             return np.squeeze(np.where(y_pred > 0, 1., -1.))
```

上述代码简要说明如下（详细内容参看代码注释）：

- __init__()方法：构造器，保存用户传入的超参数，并根据参数 kernel 指定核函数。
- _gaussian_kernel()方法：高斯核函数的实现。
- _linear_kernel()方法：线性核函数的实现。实际上，使用线性核函数等于不使用核函数。
- _g()方法：$g(x)$ 函数的实现。
- _optimize_alpha_i_j()方法：尝试对由外层循环和内层循环选定的一对 α_1, α_2 进行优化。
- _optimize_alpha_i()方法：尝试对外层循环选定的 α_1 进行优化。优化需要两个变量，因此在该方法内部根据4.2节描述的规则执行内层循环选择 α_2，然后调用_optimize_alpha_i_j()方法针对α_1, α_2 进行优化。
- _predict_one()方法： 对单个实例进行预测。该方法功能和_g()方法相同，只是它使用已训练好的模型参数进行计算。
- train()方法：训练模型。该方法由 3 部分构成：
  - 首先初始化训练时使用的变量 alpha, b, E 缓存。
  - 反复执行外层循环，调用_optimize_alpha_i()方法训练模型。
  - 训练完成后，保存模型参数。
- predict()方法：预测。内部调用_predict_one()方法对 X 中每个实例进行预测。

6.6 项目实战

最后，我们来做一个 SVM 实战项目：使用 SVM 分类器识别手写英文字母，如表 6-1 所示。

表 6-1　手写字母数据集（https://archive.ics.uci.edu/ml/datasets/Letter+Recognition）

列号	列名	含义	特征/类标记	可取值
1	lettr	A~Z 的字母	类标记	A~Z
2	x-box	horizontal position of box	特征	整数
3	y-box	vertical position of box	特征	整数
4	width	width of box	特征	整数
5	high	height of box	特征	整数
6	onpix	total # on pixels	特征	整数
7	x-bar	mean x of on pixels in box	特征	整数
8	y-bar	mean y of on pixels in box	特征	整数
9	x2bar	mean x variance	特征	整数
10	y2bar	mean y variance	特征	整数
11	xybar	mean x y correlation	特征	整数
12	x2ybr	mean of x * x * y	特征	整数
13	xy2br	mean of x * y * y	特征	整数
14	x-ege	mean edge count left to right	特征	整数
15	xegvy	correlation of x-ege with y	特征	整数
16	y-ege	mean edge count bottom to top	特征	整数
17	yegvx	correlation of y-ege with x	特征	整数

数据集总共有 20000 条数据，其中的每一行包含一个手写字母的类标记以及该手写字母在黑白像素长方形中的 16 个特征。有 26 个字母就有 26 个类别，但我们实现的 SVM 是一个二元分类器，只能输出+1 或-1。因此，在这个项目中，我们只识别 26 个字母中某一个指定字母，实验中以字母“C”为例。

读者可使用任意方式将数据集文件 letter-recognition.data 下载到本地。这个文件所在的 URL 为：https://archive.ics.uci.edu/ml/machine-learning-databases/letter-recognition/letter-recognition.data。

6.6.1　准备数据

调用 numpy 的 genfromtxt 函数加载数据集：

```
1.  >>> import numpy as np
2.  >>> X = np.genfromtxt('letter-recognition.data', delimiter=',',
usecols=range(1, 17))
```

```
3.  >>> X
4.  array([[ 2.,  8.,  3., ...,  8.,  0.,  8.],
5.         [ 5., 12.,  3., ...,  8.,  4., 10.],
6.         [ 4., 11.,  6., ...,  7.,  3.,  9.],
7.         ...,
8.         [ 6.,  9.,  6., ..., 12.,  2.,  4.],
9.         [ 2.,  3.,  4., ...,  9.,  5.,  8.],
10.        [ 4.,  9.,  6., ...,  7.,  2.,  8.]])
11. >>> y = np.genfromtxt('letter-recognition.data', delimiter=',',
usecols=0, dtype=np.str)
12. >>> y
13. array(['T', 'I', 'D', ..., 'T', 'S', 'A'], dtype='<U1')
```

目前 y 中是字符串形式的类标记，我们把其中的'C'转换为+1，其他字母转换为-1：

```
1.  >>> y = np.where(y == 'C', 1, -1)
2.  >>> y
3.  array([-1, -1, -1, ..., -1, -1, -1])
```

至此，数据准备完毕。

6.6.2 模型训练与测试

SMO 的超参数有：

（1）惩罚系数 C

（2）步长阈值 tol

（3）核函数 kernel

（4）高斯核函数的参数 gamma（决定带宽大小）

首先，直觉上这样复杂的数据集应该不是线性可分或近似线性可分的（实验表明确实如此），因此我们决定使用高斯核函数。其余几个参数需要通过测试比较，选出最优组合。

我们先随意使用一组超参数（C=1，tol=0.01，gamma=0.01）进行一次实验，创建模型：

```
1.  >>> from svm import SMO
2.  >>> clf = SMO(C=1, tol=0.01, kernel='rbf', gamma=0.01)
```

然后，调用 sklearn 中的 train_test_split 函数将数据集切分为训练集和测试集（比例为 7:3）：

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

训练模型：

```
>>> clf.train(X_train, y_train)
```

使用（C=1， tol=0.01，gamma=0.01）这组超参数时，只需稍等几秒训练便完成了（使用某些参数时可能需要训练很久）。

使用已训练好的模型对测试集中的实例进行预测，并调用 sklearn 中的 accuracy_score 函数计算预测的准确率：

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = clf.predict(X_test)
>>> accuracy = accuracy_score(y_test, y_pred)
>>> accuracy
0.993
```

99.3%的准确率看似很不错，但不要太乐观，要知道即使将所有测试点都预测成-1（一个字母'C'都没识别出来），也会有 95%以上的准确率，因为字母'C'在测试集中仅占 5%左右。我们进一步了解有多少字母'C'没有被识别出来，以及有多少其他字母被误认成了'C'。调用 sklearn 中的 confusion_matrix 函数可以帮助我们了解这些信息：

```
>>> from sklearn.metrics import confusion_matrix
>>> C = confusion_matrix(y_test, y_pred)
>>> C
array([[5788,    1],
       [  41,  170]])
```

以实际值和预测值为参数调用 confusion_matrix 函数，它返回了一个混淆矩阵 C，矩阵中 (i, j) 位置的元素代表：实际为第 i 个类别，被模型预测为第 j 个类别的样本的个数。也就是说，只有主对角线上的元素为各种预测正确的数量，其他位置的元素为各种预测错误的数量。那么，以上混淆矩阵的含义为：

- 有 5788 个非字母'C'被预测为非字母'C'，预测正确。
- 有 170 个字母'C'被预测为字母'C'，预测正确。

- 有 1 个非字母'C'被预测为字母'C'，预测错误。
- 有 41 个字母'C'被预测为非字母'C'，预测错误。

可以看出，字母'C'没有被识别出来的比例还是很高的，我们还需调整超参数以使性能更佳。

之后，我们尝试各种超参数的组合来训练模型并比较性能，该过程代码如下（具体参数根据实验情况随时调整）：

```
# 用于保存各超参数组合的成绩
acc_list = []
p_list = []

# 待尝试的各超参数，可先粗调再细调
C_list = [0.1, 1, 10, 100]
gamma_list = [0.1, 1, 10, 100]
for C in C_list:
    for gamma in gamma_list:
        # 迭代不同超参数组合， 创建模型
        clf = SVC(C=C, tol=0.01, kernel='rbf', gamma=gamma)
        # 训练，预测，计算准确率
        clf.fit(X_train, y_train)
        y_pred = clf.predict(X_test)
        accuracy = accuracy_score(y_pred, y_test)
        # 保存成绩
        acc_list.append(accuracy)
        p_list.append((C, gamma))

# 找到最优超参数组合
idx = np.argmax(acc_list)
print(p_list(idx))
```

经过反复实验，发现将超参数设置为（C=5，tol=0.01，gamma=0.05）时，训练时间不会太久，并且性能不错：

```
>>> clf = SMO(C=5, tol=0.01, kernel='rbf', gamma=0.05)
>>> clf.train(X_train, y_train)
>>> y_pred = clf.predict(X_test)
>>> accuracy = accuracy_score(y_test, y_pred)
>>> accuracy
```

```
6.  0.9988333333333334
7.  >>> C = confusion_matrix(y_test, y_pred)
8.  >>> C
9.  array([[5788,    1],
10.        [   6,  205]])
```

此时模型的预测准确率提高到了 99.88%，混淆矩阵给出的信息表明只有 6 个字母'C'没有被识别出来，1 个非字母'C'被误认成字母'C'，这样的性能是很令人满意的。

至此，我们这个项目就完成了。实际上，基于 SVM 也可以构建多元分类器（如识别 26 个字母），在这里就不再继续讨论了，有兴趣的读者可以查阅相关资料。