支持向量机(Support Vector Machines，SVM)

线性可分支持向量机

线性可分支持向量机

给定训练数据：

$$T = \{(x_1, y_1), (x_2, y_2), \cdots, (x_N, y_N)\} \tag{1}$$

其中$x_i \in R^n, y_i \subseteq \{1, -1\}, i = 1, 2, \cdots, N$可以被一超平面$w \cdot x + b = 0$根据类别$y_i$分开。

> 给定线性可分训练数据集，通过间隔最大化或等价地求解相应的凸二次规划问题学习得到的分离超平面为
>
> $w^* \cdot x + b^* = 0$
>
> 以及相应的分类决策函数
> $f(x) = sign(w^* \cdot x + b^*)$
> 称为线性可分支持向量机

函数间隔和几何间隔

对于给定的训练集T以及超平面(w, b)，任意的(x_i, y_i)到超平面的距离为：

$$l = \frac{|w \cdot x_i + b|}{||w||} \tag{2}$$

其中分子|w·x_i+b|可以相对的表示(x_i, y_i)到超平面的距离，可以将其认为分类的确信程度(离超平面越远确信程度越高)，而y_i和$w \cdot x_i + b$的乘积可以表示分类是否正确，所以可以用$y_i(w \cdot x_i + b)$来表示分类的正确性以及确信度。

函数间隔

> 对于给定的训练集T以及超平面(w,b)，定义超平面关于样本点(x_i,y_i)的函数间隔为:
>
> $\hat{\gamma}_i = y_i(w \cdot x_i + b)$
> 而超平面关于训练集T的函数间隔定义为：
> $\hat{\gamma} = min_{i=1,2,\cdots,N} \quad \hat{\gamma}_i$

几何间隔

注意到，当w, b成倍数增长时，函数间隔γ也会成倍数增长，即：

$$\begin{aligned} w &= \lambda w \\ b &= \lambda b \\ \gamma &\to \lambda \gamma \end{aligned} \tag{3}$$

但是超平面(w, b)是没有发生变化的：

$$\lambda w \cdot x + \lambda b = 0 \tag{4}$$

为了唯一的确定超平面，我们可以对w做一些约束，使得$||w|| = 1$，从而可以得到唯一的w, b。

> 定义超平面对样本点(x_i,y_j)的几何间隔为：

> $\gamma_i = y_i(\frac{w}{||w||} \cdot x + \frac{b}{||w||})$
>
> 定义超平面对训练集的几何间隔为：
> $\gamma = min_{i=1,2,\cdots,N} \ \gamma_i$

根据函数间隔和几何间隔的定义可以得出两者的关系：

$$\gamma_i = \frac{\hat{\gamma}_i}{||w||} \tag{5}$$
$$\gamma = \frac{\hat{\gamma}}{||w||}$$

间隔最大化

对线性可分的训练数据集而言，线性可分分离超平面有无穷多个(等价于感知机)，但是几何间隔最大的分离超平面却是唯一的。这里的间隔最大化又称为硬间隔最大化。

间隔最大化的直观解释是：对训练数据集找到几何间隔最大的超平面意味着以充分大的确信度对训练数据进行分类。也就是说，不仅将正负实例点分开，而且对最难分的实例点(离超平面最近的点)也有足够大的确信度将他们分开。这样的超平面应该对未知的新实例有很好的分类预测能力。

最大间隔分离超平面

> 几何间隔最大的分离超平面，并且满足一些约束条件，等价于下面的约束最优化问题：
>
> $$max_{w,b} \ \gamma \tag{6}$$
> $$s.t. \quad y_i(\frac{w \cdot x}{||w||} + \frac{b}{||w||}) \geq \gamma$$
>
> 考虑函数间隔和几何间隔的关系，这个问题也等价于：
>
> $$max_{w,b} \quad \frac{\hat{\gamma}}{||w||} \tag{7}$$
> $$s.t. \quad y_i(w \cdot x + b) \geq \hat{\gamma}$$

函数间隔的取值并不影响最优化问题的解(不影响模型)，也就是说，当w, b成倍数增长时，函数间隔也成相同倍数增长，但是模型(超平面)是不变的。反之，如果函数间隔不确定，w, b会有无穷多个解(都成一定的倍数关系)，为了确定出唯一解，这里我们设定$\hat{\gamma} = 1$，那么问题就转化为：

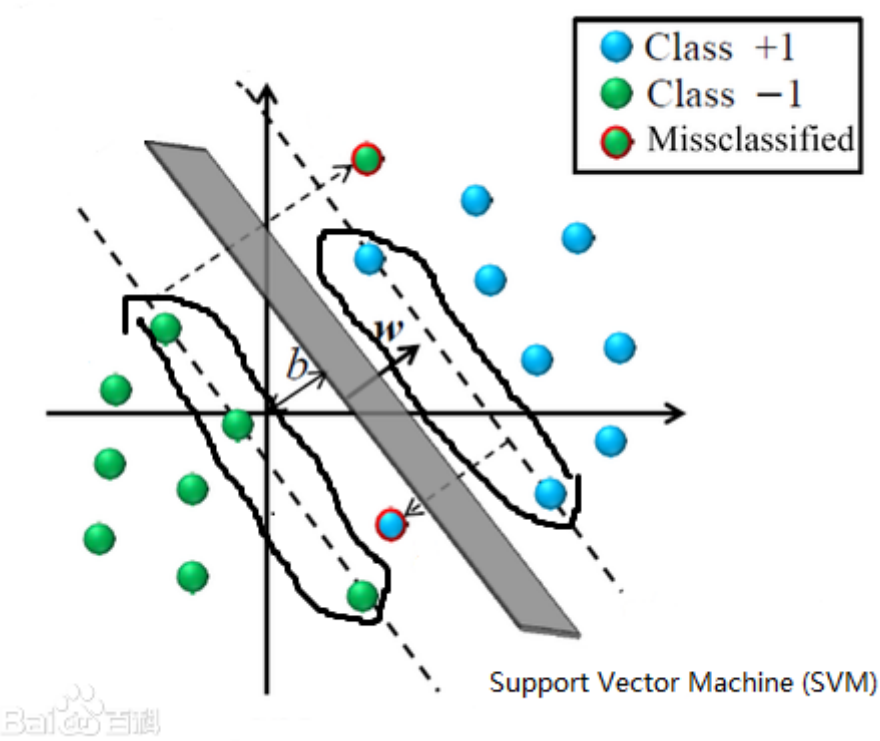
> $$max_{w,b} \ \frac{1}{||w||} \tag{8}$$
> $$s.t. \quad y_i(w \cdot x + b) \geq 1$$
>
> 为了方便后续计算，上述问题等价于：
>
> $$min_{w,b} \ ||w|| \tag{9}$$
> $$s.t. \quad y_i(w \cdot x + b) \geq 1$$
>
> 为了使构造函数是凸函数，将上述问题继续等价：
>
> $$min_{w,b} \quad \frac{1}{2}||w||^2 \tag{10}$$
> $$s.t. \quad 1 - y_i(w \cdot x + b) \leq 0$$

得到上述的等价问题就可以应用拉格朗日数乘法了。

支持向量和间隔边界

在线性可分的情况下，训练数据集的样本点中与分离超平面距离最近的样本点的实例称为支持向量(support vector)。支持向量是使约束条件$1-y_i(w\cdot x+b)\leq 0$成立的点，也即$y_i(w\cdot x+b)=1$：

> 对$y_i=+1$的正例点，支持向量在超平面$H_1:w\cdot x+b=1$上
>
> 对$y_i=-1$的负例点，支持向量在超平面$H_2:w\cdot x+b=-1$上



> H_1和H_2即为间隔边界，而H_1和H_2之间的距离称为间隔(margin)。间隔依赖于分离超平面的法向量w，等于$\frac{2}{||w||}$。
>
> 在决定分离超平面时，实际上只有支持向量起作用，所以这种分类模型称之为“支持向量机(SVM)”。

学习的对偶算法

为了求解出超平面(w,b)，我们需要满足约束条件以及求解目标，原始问题为：

> $$\begin{aligned} &min_{w,b}\ \ \frac{1}{2}||w||^2 \qquad (11)\\ &s.t.\quad 1-y_i(w\cdot x+b)\leq 0 \end{aligned}$$

构建拉格朗日函数$L(w,b,a)$：

> $$\begin{aligned} L(w,b,a)&=\frac{1}{2}||w||^2+\sum_{i=1}^{N}a_i(1-y_i(w\cdot x_i+b))\\ &=\frac{1}{2}||w||^2-\sum_{i=1}^{N}a_iy_i(w\cdot x_i+b)+\sum_{i=1}^{N}a_i \end{aligned}$$
>
> 原始问题为：
>
> $$min_{w,b}\ \ max_a\ \ L(w,b,a) \qquad (12)$$
>
> 对偶问题为：
>
> $$max_a\ \ min_{w,b}\ \ L(w,b,a) \qquad (13)$$

接下来对对偶问题进行求解，从里到外进行求解

> 1. 先求$min_{w,b}\ \ L(w,b,a)$
>
> 对w,b求偏导并令其等于0。
>
> $$\nabla_w L(w,b,a)=w-\sum_{i=1}^{N}a_i y_i x_i \tag{14}$$
> $$\nabla_b L(w,b,a)=-\sum_{i=1}^{N}a_i y_i$$
>
> (其中$||w||^2=w^Tw=w^TIw,\ \ (I_{为单位矩阵})$，根据矩阵求导法则$\frac{\partial(X^TAX)}{\partial X}=(A+A^T)X$可知，其求导结果为$2Iw=2w$，而根据$\frac{\partial AX}{\partial X}=A$可求出其余项的偏导)
>
> $$w=\sum_{i=1}^{N}a_i y_i x_i \tag{15}$$
> $$\sum_{i=1}^{N}a_i y_i=0$$
>
> 将其带回到$L(w,b,a)$得到：
>
> $$\begin{aligned}
> &\sum_{i=1}^{N}a_i y_i(w\cdot x_i+b)+\sum_{i=1}^{N}a_i \\
> min_{w,b}\ \ L(w,b,a)&=\frac{1}{2}\sum_{i,j=1}^{N}a_i a_j y_i y_j(x_i\cdot x_j)-\sum_{i=1}^{N}a_i y_i(\sum_{j=1}^{N}a_j y_j(x_j\cdot x_i)+b)+\sum_{i=1}^{N}a_i \\
> &=\frac{1}{2}\sum_{i,j=1}^{N}a_i a_j y_i y_j(x_i\cdot x_j)-\sum_{i,j=1}^{N}a_i a_j y_i y_j(x_i\cdot x_j)-b\sum_{i=1}^{N}a_i y_i+\sum_{i=1}^{N}a_i \\
> &=-\frac{1}{2}\sum_{i,j=1}^{N}a_i a_j y_i y_j(x_i\cdot x_j)+\sum_{i=1}^{N}a_i
> \end{aligned}$$
>
> 2. 求$min_{w,b}\ \ L(w,b,a)$的极大，即是对偶问题：
>
> $$max_a\quad -\frac{1}{2}\sum_{i,j=1}^{N}a_i a_j y_i y_j(x_i\cdot x_j)+\sum_{i=1}^{N}a_i \tag{16}$$
> $$s.t.\quad \sum_{i=1}^{N}a_i y_i=0$$
> $$a_i\geq 0,\ \ i=1,2,\cdots,N$$

线性支持向量机和软间隔最大化

线性支持向量机

线性可分问题的支持向量机学习方法，对线性不可分训练数据是不适用的，因为这时候原不等式约束$y_i(w\cdot x+b)\geq 1$并不能都成立。所以引入了软间隔最大化。

给定训练数据：

$$T=\{(x_1,y_1),(x_2,y_2),\cdots,(x_N,y_N)\} \tag{17}$$

其中$x_i\in R^n, y_i\subseteq\{1,-1\}, i=1,2,\cdots,N$。假定这些训练数据集**不是线性可分**的。一般情况下，训练数据中可能会混有一些**特异点(outlier)**，而将这些特异点除去后，剩下的大部分样本点组成的集合是线性可分的。

为了解决特异点造成的线性不可分问题，可以在原来约束的基础上引入**松弛变量**$\xi_i\geq 0$，使得函数间隔加上ξ_i能大于等于1，这样的话，约束条件就变成了：

$$y_i(w \cdot x_i + b) + \xi_i \geq 1$$
$$y_i(w \cdot x_i + b) \geq 1 - \xi_i$$

当出现特异点的时候$\xi_i > 0$，特别地当出现误分类点时$\xi_i > 1$。但是如果ξ_i很大很大时，随便一个超平面都满足上面的条件，所以我们需要尽量减小ξ_i的值来确定正确的超平面。

所以将目标函数设置为：

$$min_{w,b,\xi} \quad \frac{1}{2}||w||^2 + C\sum_{i=1}^{N}\xi_i \tag{18}$$

其中$C > 0$为惩罚参数，是一个调和w、ξ_i的权重系数，当C比较大时，为了保证整体比较小，那么ξ_i尽可能小的取值也就是尽可能取0，也就是说此时要求模型的**特异点尽可能少。**

原始问题

$$\begin{aligned} min_{w,b,\xi} \quad & \frac{1}{2}||w||^2 + C\sum_{i=1}^{N}\xi_i \\ s.t. \quad & y_i(w \cdot x_i + 0) \geq 1 - \xi_i \\ & \xi_i \geq 0 \quad i = 1, 2, \ldots, N \end{aligned}$$

线性支持向量机

对于给定的线性不可分的训练数据集，通过求解上述原始的二次凸优化问题，也即软间隔最大化问题，得到的分离超平面为：

$$w^* \cdot x + b^* = 0 \tag{19}$$

以及相应的分类决策函数：

$$f(x) = sign(w^* \cdot x + b^*) \tag{20}$$

称为线性支持向量机

学习的对偶算法

构造拉格朗日函数L：

$$L(w, b, a, \mu, \xi) = \frac{1}{2}||w||^2 + C\sum_{i=1}^{N}\xi_i + \sum_{i=1}^{N}a_i(1 - \xi_i - y_i(w \cdot x_i + b)) - \sum_{i=1}^{N}\mu_i\xi_i \tag{21}$$

对偶问题

$$max_{a,\mu} \quad min_{w,b,\xi} \quad L(w, b, a, \mu, \xi) \tag{22}$$

1. 先求$min_{w,b,\xi} \quad L(w, b, a, \mu, \xi)$：

$$\nabla_w L = w - \sum_{i=1}^{N}a_i y_i x_i = 0$$
$$\nabla_b L = -\sum_{i=1}^{N}a_i y_i = 0$$
$$\nabla_{\xi_i} L = C - a_i - \mu_i = 0$$

得到：

$$w=\sum_{i=1}^{N}a_iy_ix_i \tag{23}$$

$$\sum_{i=1}^{N}a_iy_i=0$$

$$C-a_i-\mu_i=0$$

带回到$min_{w,b,\xi}\ L(w,b,a,\mu,\xi)$中的到：

$$min_{w,b,\xi}\ L(w,b,a,\mu,\xi)=-\frac{1}{2}\sum_{i,j=1}^{N}a_ia_jy_iy_j(x_i\cdot x_j)+\sum_{i=1}^{N}a_i \tag{24}$$

2. 求$min_{w,b,\xi}\ L(w,b,a,\mu,\xi)$的极大，即求$max_{a,\mu}\ min_{w,b,\xi}\ L(w,b,a,\mu,\xi)$:

$$max_{a,\mu}\quad -\frac{1}{2}\sum_{i,j=1}^{N}a_ia_jy_iy_j(x_i\cdot x_j)+\sum_{i=1}^{N}a_i$$

$$s.t.\quad \sum_{i=1}^{N}a_iy_i=0$$

$$C-a_i-\mu_i=0$$

$$a_i\geq 0$$

$$\mu_i\geq 0$$

因为$min_{w,b,\xi}\ L(w,b,a,\mu,\xi)$中$\mu$以及被消掉，整理上述式子可以得到：

$$max_{a}\quad -\frac{1}{2}\sum_{i,j=1}^{N}a_ia_jy_iy_j(x_i\cdot x_j)+\sum_{i=1}^{N}a_i$$

$$s.t.\quad \sum_{i=1}^{N}a_iy_i=0$$

$$0\leq a_i\leq C$$

支持向量

讨论一下上述对偶问题需要满足的KKT条件：

$$\text{原始拉格朗日函数：}\ L(w,b,a,\mu,\xi)=\frac{1}{2}||w||^2+C\sum_{i=1}^{N}\xi_i+\sum_{i=1}^{N}a_i(1-\xi_i-y_i(w\cdot x_i+b))-\sum_{i=1}^{N}\mu_i\xi_i$$

$$\nabla_{w^*}\ L=w^*-\sum_{i=1}^{N}a_i^*y_ix_i=0$$

$$\nabla_{b^*}\ L=\sum_{i=1}^{N}a_i^*y_i=0$$

$$\nabla_{\xi_i^*}\ L=C-a_i^*-\mu_i^*=0$$

$$a_i^*(1-\xi_i-y_i(w_i^*\cdot x+b^*)=0$$

$$\mu_i\xi_i=0$$

$$\xi_i^*\geq 0$$

$$1-\xi_i^*-y_i(w_i^*\cdot x+b^*)\leq 0$$

$$a_i^*\geq 0$$

$$\mu_i^*\geq 0$$

对偶问题以及原始问题通解需要满足上述KKT条件，所以：

- $a_i^*=0$时，$u_i^*=C$，必有$\xi_i=0$，所以有$y_i(w_i^*\cdot x+b^*)\geq 1$:

  此时样本点在间隔边界上或者分类正确

- $a_i^*=C$时，$u_i^*=0$，必有$\xi_i^*\geq 0$，且有$y_i(w_i^*\cdot x+b^*)=1-\xi_i^*$:

  此时根据ξ_i^*的取值样本点的位置可分为分类错误、超平面上、超平面和间隔边界之间、间隔边界上

- 当$0<a_i^*<C$时，必有$\xi_i^*=0$，所以有$y_i(w_i^*\cdot x+b^*)=1$:

  此时直线在间隔边界上，所以此时的样本点(x_i,y_i)为支持向量

非线性支持向量机和核函数

写出核函数篇幅较长，而这里需要的核函数知识较少，读者可参考这位大佬的文章

序列最小最优算法(SMO)

数据集较大时，训练SVM需要求解一个大规模的**二次规划(quadratic programming, QP)**问题。SMO算法的大致思路是：将这个大的QP问题分解成一系列尽可能小的QP子问题，然后依次求解这些小的QP子问题(Osuna等人已经证明当满足一定条件时可以将大QP问题分解为小QP问题，并且可以收敛)

SMO算法的停机条件

前面谈到的对偶问题是：

$$
\begin{aligned}
min_a \ \ & \frac{1}{2}\sum_{i,j=1}^{N} a_i a_j y_i y_j K(\mathbf{x\_i}, \mathbf{x\_j}) - \sum_{i=1}^{N} a_i \quad (25) \\
& s.t. \quad 0 \le a_i \le C \\
& \sum_{i=1}^{N} a_i y_i = 0
\end{aligned}
$$

$K(\mathbf{x_i}, \mathbf{x_j})$为正定核函数，下面简记为$K_{ij}$。

KKT条件重温：

$$
\begin{aligned}
\nabla_{w^*} \ \ & L = w^* - \sum_{i=1}^{N} a_i^* y_i x_i = 0 \\
\nabla_{b^*} \ \ & L = \sum_{i=1}^{N} a_i^* y_i = 0 \\
\nabla_{\xi_i^*} \ \ & L = C - a_i^* - \mu_i^* = 0 \\
& a_i^*(1 - \xi_i - y_i(w_i^* \cdot x + b^*) = 0 \\
& \mu_i \xi_i = 0 \\
& \xi_i^* \ge 0 \\
& 1 - \xi_i^* - y_i(w_i^* \cdot x + b^*) \le 0 \\
& a_i^* \ge 0 \\
& \mu_i^* \ge 0
\end{aligned}
$$

以及：

$$
\begin{aligned}
a_i = 0 &\Rightarrow y_i(\sum_{j=1}^{N} a_j y_j K_{ij} + b) \ge 1 \\
0 < a_i < C &\Rightarrow y_i(\sum_{j=1}^{N} a_j y_j K_{ij} + b) = 1 \\
a_i = C &\Rightarrow y_i(\sum_{j=1}^{N} a_j y_j K_{ij} + b) \le 1
\end{aligned}
$$

为方便表示，令$g(x_i) = \sum_{j=1}^{N} a_j y_j K_{ij} + b$，

根据Osuna以及Platt的研究，停机条件为：

$$0 \leq a_i \leq C$$
$$\sum_{i=1}^{N} a_i y_i = 0$$
$$y_i g(x_i) = \begin{cases} \geq 1, \{x_i | a_i = 0\} \\ = 1, \{x_i | 0 < a_i < C\} \\ \leq 1, \{x_i | a_i = C\} \end{cases}$$

SMO算法的思路

如果训练集中的所有样本都满足上述停机条件，那么就得到了问题的最优解；否则将大QP问题拆分成一系列小的QP问题，每次对小的QP问题求解，然后一步步向最优解优化。

最小的QP子问题包含两个变量，但是这两个变量是互相受限的，因为有等式$\sum_{i=1}^{N} a_i y_i = 0$的存在。如果每个最小QP问题只优化一个变量则无法满足上述等式约束，所以可以两个变量一起进行优化，从而可以保证等式约束是成立的。

所以现在我们的重点在于：**两个变量二次规划的解析方法** and **选择变量的启发式方法。**

两个变量二次规划的解析方法

不失一般性，假设选择的两个变量为a_1, a_2，其他变量视为常量。

原目标函数为：

$$min_a \ \frac{1}{2} \sum_{i,j=1}^{N} a_i a_j y_i y_j K(\mathbf{x\_i}, \mathbf{x\_j}) - \sum_{i=1}^{N} a_i \qquad (26)$$

现目标函数为:

$$min_{a_1,a_2} \ W(a_1, a_2) = \frac{1}{2} a_1^2 K_{11} + \frac{1}{2} a_2^2 K_{22} + a_1 a_2 y_1 y_2 K_{12} + c_1 y_1 a_1 + c_2 y_2 a_2 - a_1 - a_2 + const \quad (27)$$

其中$c_i = \sum_{j=3}^{N} a_j y_j K_{ij}$，$const$为其他变量组成的常数，与优化目标无关，所以我们可以将目标函数写为：

$$min_{a_1,a_2} \ W(a_1, a_2) = \frac{1}{2} a_1^2 K_{11} + \frac{1}{2} a_2^2 K_{22} + a_1 a_2 y_1 y_2 K_{12} + c_1 y_1 a_1 + c_2 y_2 a_2 - a_1 - a_2 \quad (28)$$

单变量约束

为了保证满足停机条件之一：$\sum_{i=1}^{N} a_i y_i = 0$，将$a_1$用$a_2$表示：

$$a_1 y_1 + a_2 y_2 = -\sum_{i=3}^{N} a_i y_i = z \qquad (29)$$
$$a_1 = y_1 (z - a_2 y_2)$$

将$a_1 = z - a_2$带回到目标函数得到：

$$min_{a_2} \ W(a_2) = \frac{1}{2} y_1^2 (z - a_2 y_2)^2 K_{11} + \frac{1}{2} a_2^2 K_{22} + y_1 (z - a_2 y_2) a_2 y_1 y_2 K_{12} + c_1 y_1^2 (z - a_2 y_2) + c_2 y_2 a_2 - y_1 (z - a_2 y_2) - a_2 \quad (30)$$

并且$y_i^2 = 1$，可将上式化简得：

$$min_{a_2} \ W(a_2) = \frac{1}{2} (z - a_2 y_2)^2 K_{11} + \frac{1}{2} a_2^2 K_{22} + (z - a_2 y_2) a_2 y_2 K_{12} + c_1 (z - a_2 y_2) + c_2 y_2 a_2 - y_1 (z - a_2 y_2) - a_2 \quad (31)$$

令其导数为0得到：

$$\frac{dW}{da_2} = -y_2 K_{11} (z - a_2 y_2) + K_{22} a_2 + K_{12} (y_2 (z - a_2 y_2) - y_2^2 a_2) - c_1 y_2 + c_2 y_2 + y_1 y_2 - 1 = 0$$
$$(K_{11} - 2K_{12} + K_{22}) a_2 - y_2 (K_{11} z - 2K_{12} z + c_1 - c_2 - y_1 + y_2) = 0$$
$$(K_{11} - 2K_{12} + K_{22}) a_2 = y_2 (K_{11} z - 2K_{12} z + c_1 - c_2 - y_1 + y_2)$$

由于是迭代算法，每一步需要更新a_1, a_2的值，所以上述的z, c_i为：

$$z = a_1^{old} y_1 + a_2^{old} y_2 \tag{32}$$
$$c_i = \sum_{j=3}^{N} a_j y_j K_{ij} = g(x_i^{old}) - b^{old} - a_1^{old} y_1 K_{1i} - a_2^{old} y_2 K_{2i}$$

带回去得到：

$$(K_{11} - 2K_{12} + K_{22})a_2 = y_2(K_{11}z - 2K_{12}z + c_1 - c_2 - y_1 + y_2)$$
$$(K_{11} - 2K_{12} + K_{22})a_2 = y_2(K_{11}(a_1^{old}y_1 + a_2^{old}y_2) - 2K_{12}(a_1^{old}y_1 + a_2^{old}y_2) + (g(x_1^{old}) - b^{old} - a_1^{old}y_1K_{11} - a_2^{old}y_2K_{21}) - (g(x_2^{old}) - b^{old} - a_1^{old}y_1K_{12} - a_2^{old}y_2K_{22}) - y_1 + y_2)$$
$$(K_{11} - 2K_{12} + K_{22})a_2 = (K_{11} - 2K_{12} + K_{22})a_2^{old} + y_2(g(x_1^{old}) - y_1) - y_2(g(x_2^{old} - y_2))$$

> 令$\eta = K_{11} - 2K_{12} + K_{22}$，记$E_i = g(x_i) - y_i$为第$i$个样本的差值。
>
> 所以有
>
> $$a_2^{new} = a_2^{old} + y_2 \frac{E_1 - E_2}{\eta} \tag{33}$$

修剪

上述的a_2^{new}不一定满足$0 \le a_2^{new} \le C$的条件，所以我们需要对其进行修剪：

等式约束：

$$a_1^{new} y_1 + a_2^{new} y_2 = a_1^{old} y_1 + a_2^{old} y_2 \tag{34}$$

可以反解出a_1^{new}：

$$a_1^{new} = y_1(a_1^{old} y_1 + a_2^{old} y_2 - a_2^{new} y_2) \tag{35}$$

不等式约束：

$$0 \le a_1^{new} \le C \tag{36}$$

> - y_1, y_2同号时：
>
> $$a_1^{new} = a_1^{old} - a_2^{old} + a_2^{new} \tag{37}$$
> $$0 \le a_1^{new} \le C \Leftrightarrow a_2^{old} - a_1^{old} \le a_2^{new} \le a_2^{old} - a_1^{old} + C$$
>
> 为了保证$0 \le a_2^{new} \le C$，令：
>
> $$L = max(0, a_2^{old} - a_1^{old})$$
> $$H = min(C, a_2^{old} - a_1^{old} + C)$$
>
> 所以必须有：
>
> $$L \le a_2^{new} \le H \tag{38}$$
>
> 当a_2^{new}越过边界时，取a_2^{new}为边界
>
> - y_1, y_2异号时：
>
> $$a_1^{new} = a_1^{old} + a_2^{old} - a_2^{new} \tag{39}$$
> $$0 \le a_1^{new} \le C \Leftrightarrow a_2^{old} + a_1^{old} - C \le a_2^{new} \le a_2^{old} + a_1^{old}$$
>
> 同理，令：
>
> $$L = max(0, a_2^{old} + a_1^{old} - C)$$
> $$H = min(C, a_2^{old} + a_1^{old})$$
>
> 所以必须有：
>
> $$L \le a_2^{new} \le H \tag{40}$$
>
> 当a_2^{new}越过边界时，取a_2^{new}为边界

即：

$$a_2^{new} = \begin{cases} H, & if \quad a_2^{new} \geq H \\ a_2^{new}, & if \quad L < a_2^{new} < H \\ L, & if \quad a_2^{new} \leq L \end{cases}$$

计算a_1^{new}

根据计算出的a_2^{new}计算a_1^{new}

$$a_1^{new} = y_1(a_1^{old}y_1 + a_2^{old}y_2 - a_2^{new}y_2) \tag{41}$$

选择变量的启发式方法

SMO在每一步都是选择两个变量进行优化，其中至少有一个违反KKT条件；根据Osuna的理论，每一步优化都会使目标函数值减小，因此可以保证算法的收敛性。

为加速收敛，SMO采用启发式的方法来选择每一步中优化的两个变量。有两个启发式规则：

- 一个是选择第一个变量的规则，构成外层循环；
- 另一个是选择第二个变量的规则，构成内层循环。

选择第一个优化变量

第一个优化变量选取训练样本中违反KKT条件最严重的样本点，并将其对应的变量作为第1个变量。具体地，检验样本点(x_i, y_i)是否满足：

$$\begin{aligned} a_i = 0 &\Leftrightarrow y_i g(x_i) \geq 1 \\ a_i = C &\Leftrightarrow y_i g(x_i) \leq 1 \\ 0 < a_i < C &\Leftrightarrow y_i g(x_i) = 1 \end{aligned}$$

其中$g(x_i) = \sum_{j=1}^{N} a_j y_j K_{ij} + b$。有时为了加快收敛速度，会在$\xi$范围内进行的，即：

$$\begin{aligned} a_i = 0 &\Leftrightarrow y_i g(x_i) \geq 1 - \xi \\ a_i = C &\Leftrightarrow y_i g(x_i) \leq 1 + \xi \\ 0 < a_i < C &\Leftrightarrow 1 - \xi \leq y_i g(x_i) \leq 1 + \xi \end{aligned}$$

一般取$\xi = 10^{-3} \sim 10^{-2}$。

在检验过程中，外层循环首先遍历所有满足条件$0 < a_i < C$的样本点，即在间隔边界上的支持向量点，检验他们是否满足KKT条件，如果他们满足则遍历整个训练集，检验是否满足KKT条件。

选择第二个优化变量

选定第一个优化变量a_1之后，选择第二个优化变量a_2，根据公式$a_2^{new} = a_2^{old} + y_2 \frac{E_1 - E_2}{\eta}$，我们可以使优化的步长最大的样本点，也就是让$y_2 \frac{E_1 - E_2}{\eta}$的绝对值最大，但是计算$\eta$需要计算核函数，为了简便，将$|E_1 - E_2|$近似为步长，根据第一个优化变量可以确定$E_1$，然后根据$E_1$的符号可以选择$E_2$使得$|E_1 - E_2|$最大，具体地：

$$\begin{aligned} if \;\; E_1 < 0: \;\; max(E_2) \\ else: \;\; min(E_2) \end{aligned} \tag{42}$$

当E_1小于0选取最大的作为E_2，反之选最小的作为E_2。

需要注意的是，每个变量不一定仅仅被优化一次，可能跟随着其他变量被优化好多次。

E和b的更新

这位大佬写的很好，可以去借鉴一下，这里就不再过多叙述，只需要记住更新公式：

$$
\begin{aligned}
b_1^{new} &= b^{old} - E_1^{old} - y_1 K_{11}(a_1^{new} - a_1^{old}) - y_2 K_{12}(a_2^{new} - a_2^{old}) \\
b_2^{new} &= b^{old} - E_2^{old} - y_1 K_{12}(a_1^{new} - a_1^{old}) - y_2 K_{22}(a_2^{new} - a_2^{old}) \\
E_i^{new} &= E_i^{old} + y_1 K_{1i}(a_1^{new} - a_1^{old}) + y_2 K_{2i}(a_2^{new} - a_2^{old}) + b^{new} - b^{old}
\end{aligned} \tag{43}
$$

$Python + Numpy$实现

```python
import numpy as np


class SVM():

    def __init__(self,max_iter = 200,C = 1):
        # 最多迭代数
        self.max_iter = max_iter
        # w
        self.w = 0
        # b
        self.b = 0
        # 松弛变量
        self.C = C
        # 精度(xi为其希腊字母)
        self.xi = 0.01

    def fit(self,X_train:np.ndarray,Y_train:np.ndarray):
        # 样本个数和X的维度
        self.m,self.n = X_train.shape
        # X
        self.X = X_train
        # Y
        self.Y = Y_train
        # alpha
        self.alpha = np.zeros(self.m)
        #  error_i = w·x_i+b-y_i
        self.E = np.array([self.g(i)-self.Y[i] for i in range(self.m)])

        for _iter in range(self.max_iter):

            # 非边界alpha
            non_bound_alpha = [i for i in range(self.m) if 0 < self.alpha[i] < self.C]
            # 边界alpha
            bound_alpha = [i for i in range(self.m) if i not in non_bound_alpha]

            # 先非边界alpha后边界alpha
            non_bound_alpha.extend(bound_alpha)

            # 选取的两个变量的下标
            i1 = i2 = None

            for i in non_bound_alpha:
                # 满足KKT条件则跳出
                if self.KKT(i):
                    continue

                # 第一个优化alpha下标
```

```
                i1 = i
                # 使第二个优化变量优化步长最大


                if self.E[i] < 0:
                    i2 = self.E.argmax()
                else:
                    i2 = self.E.argmin()
                break

            # 待优化变量
            E1_old = self.E[i1]
            E2_old = self.E[i2]
            alpha1_old = self.alpha[i1]
            alpha2_old = self.alpha[i2]
            Y1 = self.Y[i1]
            Y2 = self.Y[i2]

            # eta(希腊字母) = K11-2K12+K22
            eta =
self.kernel(i1,i1)-2*self.kernel(i1,i2)+self.kernel(i2,i2)

            # eta小于等于0则重新找优化变量
            if eta <=0 :
                continue

            # 找到L,H
            if Y1 == Y2:
                L = max(0,alpha1_old+alpha2_old-self.C)
                H = min(self.C,alpha1_old+alpha2_old)
            else:
                L = max(0,alpha2_old-alpha1_old)
                H = min(self.C,alpha2_old-alpha1_old+self.C)

            # 更新alpha2
            alpha2_new_unclipped = alpha2_old + Y2*(E1_old-E2_old)/eta
            alpha2_new = self.update_alpha2(alpha2_new_unclipped,L,H)

            # 更新alpha1
            alpha1_new = alpha1_old + Y1*Y2*(alpha2_old-alpha2_new)

            # 更新b
            b1_new = self.b-E1_old-Y1*self.kernel(i1,i1)*(alpha1_new-
alpha1_old)-Y2*self.kernel(i1,i2)*(alpha2_new-alpha2_old)
            b2_new = self.b - E2_old-Y1*self.kernel(i1,i2)*(alpha1_new-
alpha1_old)-Y2*self.kernel(i2,i2)*(alpha2_new-alpha2_old)

            if 0< alpha1_new <self.C:
                b_new = b1_new
            elif 0<alpha2_new <self.C:
                b_new = b2_new
            else:
                b_new = (b1_new+b2_new)/2

            # 更新E
            self.E = self.E + Y1*(alpha1_new-
alpha1_old)*np.array([self.kernel(i1,j) for j in range(self.m)])+ \
```

```
                Y2*(alpha2_new-alpha2_old)*np.array([self.kernel(i2,j) for j in range(self.m)]) + np.array([b_new-self.b for _ in range(self.m)])

            # 修改到原变量中
            self.alpha[i1] = alpha1_new
            self.alpha[i2] = alpha2_new
            self.b = b_new

        # 获取w
        self.w = np.dot(self.alpha,self.Y.reshape(-1,1)*self.X)


    # 更新alpha2
    def update_alpha2(self, alpha2_new_unclipped, L, H):
        # 越界则取边界
        if alpha2_new_unclipped < L:
            return L
        elif alpha2_new_unclipped > H:
            return H
        # 没有越界则不变
        else:
            return alpha2_new_unclipped

    # KKT条件
    def KKT(self,i):
        if self.alpha[i] == 0:
            return self.Y[i]*self.g(i) >= 1-self.xi
        elif self.alpha[i] == self.C:
            return self.Y[i]*self.g(i)<= 1+self.xi
        else:
            return 1-self.xi<= self.Y[i]*self.g(i)<= 1+self.xi

    # g(i) = w·x+b = SUM(a_j*y_j*K_ij)
    def g(self,i):
        sum = 0
        for j in range(self.m):
            sum += self.alpha[j]*self.Y[j]*self.kernel(i,j)
        return sum+self.b

    # 核函数
    def kernel(self,i,j):
        return np.dot(self.X[i],self.X[j])


    def predict(self,X):
        return np.sign(np.dot(self.w,X)+self.b)

    def score(self,X_test,Y_test):
        right_num = 0
        for i in range(len(X_test)):
            y = self.predict(X_test[i])
            if y == Y_test[i]:
                right_num+=1
        return right_num/len(X_test)
```