libSVM 源码解读

邢存远, https://welts.xyz

2021 年 8 月 19 日

目录

1 引言

早在学习 SVM 时，笔者便有亲手实现一个 SVM 的想法。后来发现其实现难度与数学技巧远高于单隐层神经网络，这对于只能写出一个二分类感知机的我不亚于小学生做高考题。在老师的建议下，笔者决定去阅读当前最流行的 SVM 代码库：libSVM 的源代码，不仅是学习 SVM 怎么写，也是学习一个合格的代码框架应该如何去设计。在此之前，笔者已经对 SVM 的 SMO 算法和实现技巧进行了一些零散的了解，这里打算将它们串联起来，同时为阅读源码提供一定的数学基础。

2 SVM，以及 libSVM 简介

支持向量机(SVM, Support Vector Machine) 属于一种线性分类器，是建立在统计学习理论的 VC 维理论和结构风险最小原理的基础上，根据有限的训练集，在模型的复杂性和学习性之间寻求最佳的折中，以获得最好的泛化能力的经典分类方法。[?]

libSVM是由国立台湾大学的林智仁教授等开发的一款利用支持向量机用于分类、回归和区间估计等机器学习任务的多语言（C++、Java、Python、MATLAB 等）、跨平台（Windows、Linux、mac OS）的工具包，最新版本为 Version 3.25。

3 数学基础

SVM 本质上是一种统计学习模型，libSVM 的实现中涉及到很多矩阵，导数，概率论，尤其是优化方面的知识。假设读者已经有掌握线性代数和微积分包括概率论的基本知识，我们将 libSVM 涉及到的优化相关知识先在这里进行整理。

3.1 优化问题

我们常用拉格朗日对偶性去求解优化问题，这也在 SVM 中被频繁使用。关于优化问题我们参考的是《Convex optimization》这本书 [?]。先来看优化问题的形式化定义。

3.1.1 原始问题

我们将

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & g_i(x) \le 0, i = 1, \cdots, m \\ & h_j(x) = 0, j = 1 \cdots, n \end{aligned} \tag{1}$$

其中 $f(x)$ 和 $g_i(x)$ 都是凸函数，$h_j(x)$ 都是仿射函数，具有这种形式的问题称作凸优化问题。又由于我们常常不会直接求解它，因此也称其为 “初始问题”。定义拉格朗日函数：

$$\mathcal{L}(x, \lambda, \mu) = f(x) + \sum_{i=1}^{m} \lambda_i g_i(x) + \sum_{j=1}^{n} \mu_i h_j(x), \quad \lambda_i \ge 0, \mu_j \in \mathbb{R} \tag{2}$$

定义函数 θ_P：

$$\theta_P(x) = \begin{cases} f(x) & x\text{满足约束} \\ +\infty & \text{else} \end{cases} \tag{3}$$

可以证明对任意 x，我们有

$$\begin{aligned} \theta_P(x) &= \max_{\lambda,\mu:\lambda_i \geq 0} \mathcal{L}(x,\lambda,\mu) \\ \min_x \theta_P(x) &= \min_x \max_{\lambda,\mu:\lambda_i \geq 0} \mathcal{L}(x,\lambda,\mu) \end{aligned} \tag{4}$$

可以证明，原问题与拉格朗日函数的“极小极大问题”有相同的解，是等价的。我们将原始问题的最优值

$$p^* = \min_x \theta_P(x) \tag{5}$$

称为**原始问题的值**。

3.1.2 对偶问题

我们先设函数

$$\theta_D(\lambda,\mu) = \min_x \mathcal{L}(x,\lambda,\mu) \tag{6}$$

然后对其求极大，也就是“极大极小问题”：

$$\max_{\lambda,\mu} \min_x \mathcal{L}(x,\lambda,\mu) = \max_{\lambda,\mu} \theta_D(x) \tag{7}$$

这个“极大极小问题”就是原始问题 (1) 的**对偶问题**，类似的，设其最优值为 d^*。

3.1.3 强对偶，弱对偶，以及 KKT 条件

弱对偶性，也就是极大极小问题的最优值必然不大于极小极大问题的最优值，这是普遍存在的：

$$d^* \leq p^* \tag{8}$$

一个形象的理解是，矮子中最高的还是矮子，身高不超过高个子中最矮的。但我们最想要的其实是只有等号成立，方便问题的求解。当两个最优值相等时强对偶性成立。幸运的是强对偶性有一个充分必要条件：KKT 条件，即 x^* 和 λ^*, μ^* 分别是原问题和对偶问题的解当且仅当下面各式成立：

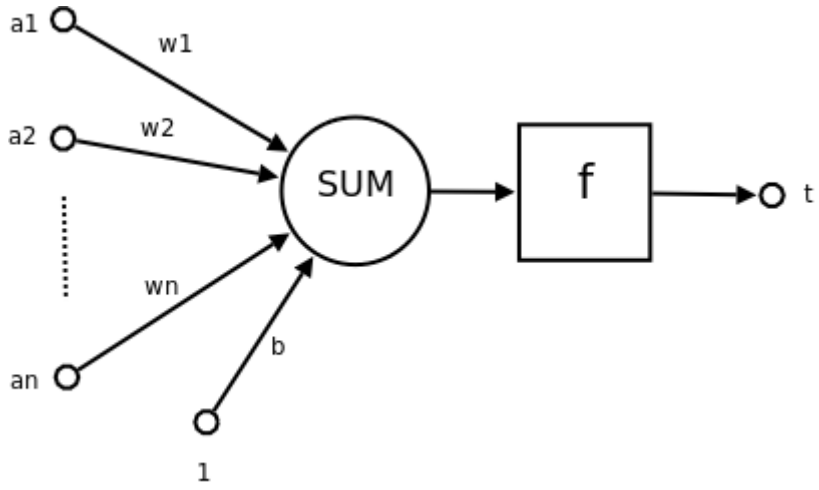
$$\begin{aligned} \nabla_x \mathcal{L}(x^*,\lambda^*,\mu^*) &= 0 \\ \lambda_i^* g_i(x^*) &= 0, i = 1,\cdots,m \\ g_i(x^*) &\leqslant 0, i = 1,\cdots,m \\ \lambda_i &\geqslant 0, i = 1,\cdots,m \\ h_j(x^*) &= 0, i = 1,\cdots,n \end{aligned} \tag{9}$$

其中第二个条件称作 KKT 的**对偶互补条件**，如果 $\lambda_i > 0$ 则必有 $g_i(x^*) = 0$。

4 支持向量机

4.1 感知机

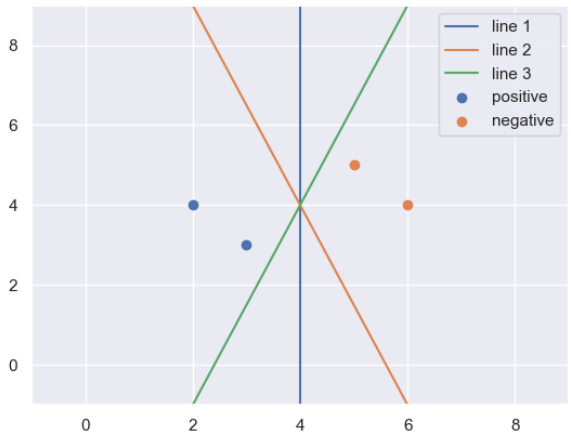
感知机（Perceptron）是最简单的人工神经网络：



也是一种二元线性分类器。给定线性可分的数据集，感知机可以找到一个将样本分开的超平面：

$$\sum_{i=1}^{n} w_i x_i + b = \boldsymbol{w}^\top \boldsymbol{x} + b \tag{10}$$

实际上对于同一个数据集，我们常常可以得到多个超平面：



上面三条直线 l1、l2 和 l3 都可以将正负样本分开，而我们更倾向于选择位于两类样本“中间”的划分超平面 l2，因为它对训练样本的扰动“容忍”性最好。换言之，泛化能力最强。支持向量机便是这样的一种较优的感知机。

4.2 形式化支持向量机

我们不能凭肉眼在感知机中找到支持向量机，我们需要将求解它的过程形式化。我们实际上要找的是这样一个超平面：

$$\boldsymbol{w}^\top \boldsymbol{x} + b = 0 \tag{11}$$

$\boldsymbol{w}$ 为超平面法向量，b 为位移项。由简单的解析几何得到空间中点 $\boldsymbol{x}$ 到平面的距离：

$$r = \frac{|\boldsymbol{w}^\top \boldsymbol{x} + b|}{\|\boldsymbol{w}\|} \tag{12}$$

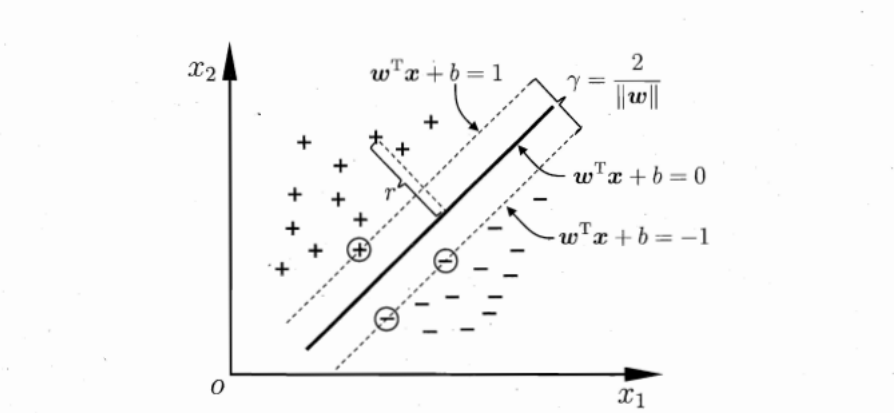
将二分类问题中的数据集标签 y_i 映射到 $\{-1, +1\}$，也就是正类 $y_i = 1$，负类 $y_i = -1$，我们想让该超平面正确分类，则有：

$$\begin{cases} \boldsymbol{w}^\top \boldsymbol{x} + b \geq +1, y_i = +1 \\ \boldsymbol{w}^\top \boldsymbol{x} + b \leq -1, y_i = -1 \end{cases} \tag{13}$$

我们的目标其实是，给定一个分离超平面，距离该超平面最近的正类样本和负类样本（也就是上述约束中的不等号为等号时的样本点，称作**支持向量**）到超平面的距离之和

$$\gamma = \frac{2}{\|\boldsymbol{w}\|} \tag{14}$$

最大，如下图所示：



其中满足 $\boldsymbol{w}^\top \boldsymbol{x} + b = 1$ 或者 -1 的点就是支持向量。

我们将这个问题形式化成前面提到的优化原始问题 ：

$$\begin{aligned} \min_{\boldsymbol{w},b} \quad & \frac{1}{2}\|\boldsymbol{w}\|^2 \\ \text{s.t.} \quad & 1 - y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) \leq 0, i = 1, 2, \cdots, m \end{aligned} \tag{15}$$

有两点值得注意：

- 原来是将 γ 极大化，为了问题的标准和求导的方便，将目标函数写成 $\|\boldsymbol{w}\|^2/2$；
- 这里约束条件里的 y_i 调换了位置，是由 $y_i^2 = 1$ 导出，后面还会用到。

4.3 用拉格朗日对偶求解

先有原问题的拉格朗日函数：

$$\mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\lambda}) = \frac{1}{2}\|\boldsymbol{w}\|^2 + \sum_{i=1}^{m} \lambda_i (1 - y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b)) \tag{16}$$

其中 $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, ..., \lambda_m), \lambda_i \geq 0$. 我们先求 $\min_{\boldsymbol{w},b} \mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\lambda})$，对变量求偏导：

$$\begin{cases} \dfrac{\partial}{\partial \boldsymbol{w}} \mathcal{L} = \boldsymbol{w} - \sum_{i=1}^{m} \lambda_i y_i \boldsymbol{x}_i \\ \dfrac{\partial}{\partial b} \mathcal{L} = \sum_{i=1}^{m} \lambda_i y_i \end{cases} \tag{17}$$

令上面两式为 0：

$$\begin{cases} \boldsymbol{w} = \sum_{i=1}^{m} \lambda_i y_i \boldsymbol{x}_i \\ 0 = \sum_{i=1}^{m} \lambda_i y_i \end{cases} \tag{18}$$

将 (18) 带入 (16)，则可消去变量：

$$\begin{aligned} \mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\lambda}) &= \frac{1}{2}\|\boldsymbol{w}\|^2 + \sum_{i=1}^{m} \lambda_i (1 - y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b)) \\ &= \frac{1}{2}(\sum_{i=1}^{m} \lambda_i y_i \boldsymbol{x_i})^2 + \sum_{i=1}^{m} \lambda_i - b\sum_{i=1}^{m} \lambda_i y_i - \sum_{i=1}^{m} \lambda_i y_i \boldsymbol{w}^\top \boldsymbol{x}_i \\ &= \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i^\top \boldsymbol{x}_j - \sum_{i=1}^{m}\sum_{j=1}^{m} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i^\top \boldsymbol{x}_j + \sum_{i=1}^{m} \lambda_i \\ &= \sum_{i=1}^{m} \lambda_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i^\top \boldsymbol{x}_j \end{aligned} \tag{19}$$

从而得到优化问题的对偶问题：

$$\begin{aligned} \max_{\boldsymbol{\lambda}} \quad & \sum_{i=1}^{m} \lambda_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i^\top \boldsymbol{x}_j \\ \text{s.t.} \quad & \sum_{i=1}^{m} \lambda_i y_i = 0 \\ & \lambda_i \geq 0, i = 1, 2, ..., m. \end{aligned} \tag{20}$$

如果能解出 $\boldsymbol{\lambda}$，我们就得到模型：

$$\begin{aligned} f(\boldsymbol{x}) &= \boldsymbol{w}^\top \boldsymbol{x} + b \\ &= \sum_{i=1}^{m} \lambda_i y_i \boldsymbol{x}_i^\top \boldsymbol{x} + b \end{aligned} \tag{21}$$

从对偶问题解出的 λ_i 式拉格朗日乘子，对应的是样本 $(\boldsymbol{x}_i, y_i)$，由于原问题有不等式约束，所以上述过程需满足 KKT 条件：

$$\begin{cases} \lambda_i \geq 0; \\ y_i f(\boldsymbol{x}_i) - 1 \geq 0 \\ \lambda_i (y_i f(\boldsymbol{x}_i) - 1) = 0 \end{cases}$$

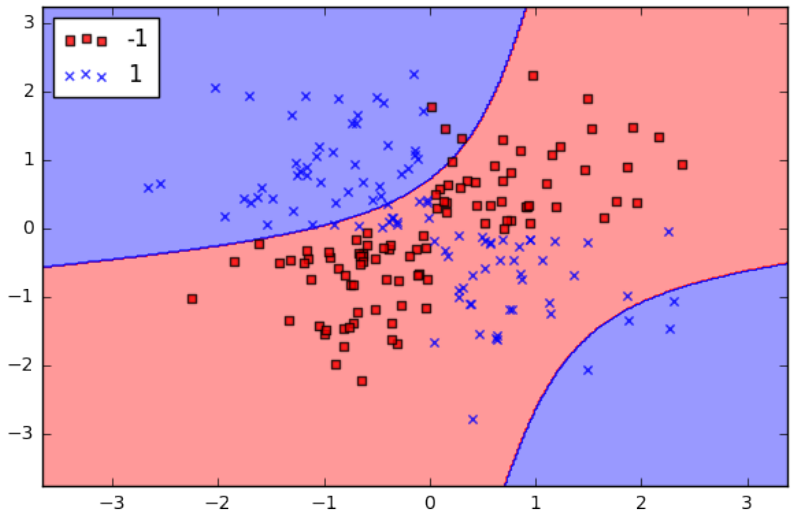
于是，对于任意训练样本 $(\boldsymbol{x_i}, y_i)$，总有 $\lambda_i = 0$ 或 $y_i f(x_i) = 1$. $\lambda_i = 0$ 的点不会对 $f(\boldsymbol{x})$ 有任何影响，否则**样本点都在最大间隔边界上**，是一个支持向量. 这为我们的训练带来启示，即大部分样本都不需要，最终模型只与支持向量有关.

4.4 核函数

以下是百度百科上关于核函数的定义：

> 支持向量机通过某非线性变换 $\phi(x)$ ，将输入空间映射到高维特征空间。特征空间的维数可能非常高。如果支持向量机的求解只用到内积运算，而在低维输入空间又存在某个函数 $K(\boldsymbol{x}, \boldsymbol{x}')$ ，它恰好等于在高维空间中这个内积，即 $K(\boldsymbol{x}, \boldsymbol{x}') = \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \rangle$ 。那么支持向量机就不用计算复杂的非线性变换，而由这个函数 $K(\boldsymbol{x}, \boldsymbol{x}')$ 直接得到非线性变换的内积，使大大简化了计算。这样的函数 $K(\boldsymbol{x}, \boldsymbol{x}')$ 称为核函数。

而简单地说，通过用 $K(\boldsymbol{x}, \boldsymbol{x}')$ 去替换简单的向量内积，使得决策边界不再是分离超平面，而是一个曲面，有效解决了线性不可分的问题，比如下图：



利用 RBF 核，我们的决策边界变成了曲线。读者可能注意到，图中的样本点并没有被完全分开，甚至样本是不可分的。而利用我们在下面提到的 C-SVC，可以解决这个问题。

4.5 更多 SVM

上面的 SVM 是最基础的支持向量机，libSVM 能够求解比这种复杂得多的问题。这一部分我们来介绍 libSVM 中的 5 种 SVM。

4.5.1 C-SVC

C-SVC 是上面的 SVM 的“升级版”，叫做软间隔支持向量机，所谓的“软”，就是在不可分（即使是非线性）的情况下，允许部分样本点不满足约束，如上图所示，但对于这些点必然要有相应“惩罚”。

加入象征不满足约束程度的稀疏变量 $\boldsymbol{\xi}$，C-SVC 其实是解决下面的优化问题：

$$\begin{aligned} \min_{\boldsymbol{w},b,\boldsymbol{\xi}} \quad & \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{l}\xi_i \\ \text{subject to} \quad & y_i(\boldsymbol{w}^\top\phi(\boldsymbol{x}_i)+b) \geqslant 1-\xi_i \\ & \xi_i \geqslant 0, i = 1,\cdots,l \end{aligned} \tag{22}$$

我们也不难得到其对应的对偶问题（习惯上，前面提到的的拉格朗日乘子 $\boldsymbol{\lambda}$ 用 $\boldsymbol{\alpha}$ 代替）：

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & \frac{1}{2}\boldsymbol{\alpha}^\top Q\boldsymbol{\alpha} - \boldsymbol{e}^\top\boldsymbol{\alpha} \\ \text{subject to} \quad & \boldsymbol{y}^\top\boldsymbol{\alpha} = 0, \\ & 0 \leqslant \alpha_i \leqslant C, i = 1,\cdots,l \end{aligned} \tag{23}$$

其中 $\boldsymbol{e}$ 是一个全 1 向量，而 $Q_{ij} = y_i y_j K(\boldsymbol{x}_i,\boldsymbol{x}_j) = y_i y_j \phi(\boldsymbol{x}_i)^\top\phi(\boldsymbol{x}_j)$。当我们解决了上面的对偶问题后，我们就可以得到 $\boldsymbol{w}$ 的解：

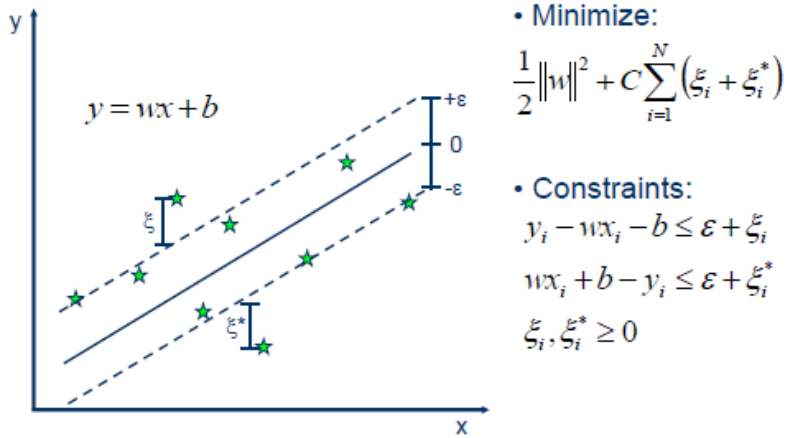
$$\boldsymbol{w} = \sum_{i=1}^{l} y_i\alpha_i\phi(\boldsymbol{x}_i) \tag{24}$$

从而决策函数：

$$\operatorname{sgn}(\boldsymbol{w}^\top\phi(\boldsymbol{x})+b) = \operatorname{sgn}\left(\sum_{i=1}^{l} y_i\alpha_i K(x_i,x)+b\right) \tag{25}$$

4.5.2 ϵ-SVR

ε-SVR，也就是 ε-Support Vector Regression，是利用支持向量机来解决回归问题：



同样，这里的稀疏变量 $\boldsymbol{\xi}$ 用来衡量给定边界 ε 后样本点对决策边界的违背程度。原始优化问题已经在上图写出，而其对偶问题为：

$$\begin{aligned} \min_{\boldsymbol{\alpha},\boldsymbol{\alpha}^*} \quad & \frac{1}{2}(\boldsymbol{\alpha}-\boldsymbol{\alpha}^*)^\top Q(\boldsymbol{\alpha}-\boldsymbol{\alpha}^*)+\varepsilon\sum_{i=1}^{l}(\alpha_i+\alpha_i^*)+\sum_{i=1}^{l}z_i(\alpha_i-\alpha_i^*) \\ \text{subject to} \quad & \boldsymbol{e}^\top(\boldsymbol{\alpha}-\boldsymbol{\alpha}^*)=0 \\ & 0\leqslant\alpha_i,\alpha_i^*\leqslant C, i=1,\cdots,l \end{aligned} \tag{26}$$

这里的 z_i 是对应数据的输出，如此设置是为了不和分类问题中的标签 y_i 混淆。

此处 $Q_{ij}=K(x_i,x_j)$。当我们求出该对偶问题后，也就能得到拟合函数

$$z(\boldsymbol{x})=\sum_{i=1}^{l}(-\alpha_i+\alpha_i^*)K(\boldsymbol{x}_i,\boldsymbol{x})+b \tag{27}$$

4.5.3 ν-SVC

ν-Support Vector Classification 在前面的 C-SVC 基础上引入了一个新参数 ν，用以控制训练误差和支持向量的数量：

$$\begin{aligned} \min_{\boldsymbol{w},b,\boldsymbol{\xi},\rho} \quad & \frac{1}{2}\|\boldsymbol{w}\|^2-\nu\rho+\frac{1}{l}\sum_{i=1}^{l}\xi_i \\ \text{subject to} \quad & y_i(\boldsymbol{w}^\top\phi(\boldsymbol{x}_i)+b)\geqslant\rho-\xi_i \\ & \xi_i\geqslant 0, i=1,\cdots,l \\ & \rho\geqslant 0 \end{aligned} \tag{28}$$

对应的对偶问题：

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & \frac{1}{2}\boldsymbol{\alpha}^\top Q\boldsymbol{\alpha} \\ \text{subject to} \quad & 0\leqslant\alpha_i\leqslant\frac{1}{l},, i=1,\cdots,l \\ & \boldsymbol{e}^\top\boldsymbol{\alpha}\geqslant\nu, \boldsymbol{y}^\top\boldsymbol{\alpha}=0 \end{aligned} \tag{29}$$

可以把它和 C-SVC 的对偶问题进行比较，发现框架大体相同，ν 其实是对 $\boldsymbol{e}^\top\boldsymbol{\alpha}$ 进行了限制。两种 SVC 问题的决策函数是相同的。

4.5.4 ν-SVR

类似的，ν-Support Vector Regression 是将 ν 引入 ε-SVR，它解决的是一个优化问题：

$$\begin{aligned}\min_{\boldsymbol{x},b,\boldsymbol{\xi},\boldsymbol{\xi}^*,\varepsilon} \quad & \frac{1}{2}\|\boldsymbol{w}\|^2 + C\big(\nu\varepsilon + \frac{1}{l}\sum_{i=1}^{l}(\xi_i + \xi_i^*)\big) \\ \text{subject to} \quad & (\boldsymbol{w}^\top\phi(\boldsymbol{x}_i) + b) - z_i \leqslant \varepsilon + \xi_i \\ & z_i - (\boldsymbol{w}^\top\phi(\boldsymbol{x}_i) + b) \leqslant \varepsilon + \xi_i^* \\ & \xi_i, \xi^* \geqslant 0, i = 1, \cdots, l \\ & \varepsilon \geqslant 0\end{aligned}$$

发现约束条件与 ε-SVR 差别不大，其对偶问题：

$$\begin{aligned}\min_{\boldsymbol{\alpha},\boldsymbol{\alpha}^*} \quad & \frac{1}{2}(\boldsymbol{\alpha} - \boldsymbol{\alpha}^*)^\top Q(\boldsymbol{\alpha} - \boldsymbol{\alpha}^*) + \boldsymbol{z}^\top(\boldsymbol{\alpha} - \boldsymbol{\alpha}^*) \\ \text{subject to} \quad & \boldsymbol{e}^\top(\boldsymbol{\alpha} - \boldsymbol{\alpha}^*) = 0, \boldsymbol{e}^\top(\boldsymbol{\alpha} + \boldsymbol{\alpha}_i^*) \leqslant C\nu \\ & 0 \leqslant \alpha_i, \alpha_i^* \leqslant C/l, i = 1, \cdots, l\end{aligned}$$

且其解出的近似函数和 ε-SVR 相同。

4.5.5 One-class SVM

现实中存在这样一个问题，在数据集 X 中判断某个数据 x_i 是不是异常数据，也就是说，对于 x_i，我们想判断 x_i 与 $X\backslash x_i$ 有多相似，如果相似度低，就将其剔除。可以发现这与分类任务并不完全相同，因为涉及到的类别只有一种。用于处理这一问题的支持向量机被称作 One-class SVM。这一过程也被称为分布估计（Distribution estimate）。

One-class SVM 的原问题：

$$\begin{aligned}\min_{\boldsymbol{w},\boldsymbol{\xi},\rho} \quad & \frac{1}{2}\|\boldsymbol{w}\|^2 - \rho + \frac{1}{\nu l}\sum_{i=1}^{l}\xi_i \\ \text{subject to} \quad & \boldsymbol{w}^\top\phi(\boldsymbol{x}_i) \geq \rho - \xi_i, \\ & \xi_i \geqslant 0, i = 1, \cdots, l.\end{aligned}$$

其对偶问题：

$$\begin{aligned}\min_{\boldsymbol{\alpha}} \quad & \frac{1}{2}\boldsymbol{\alpha}^\top Q\boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \leq \alpha_i \leq 1/(\nu l), i = 1, \cdots l \\ & \boldsymbol{e}^\top\boldsymbol{\alpha} = 1\end{aligned}$$

其中 $Q_{ij} = K(x_i, x_j)$，决策函数：

$$\operatorname{sgn}\left(\sum_{i=1}^{l}\alpha_i K(\boldsymbol{x}_i, \boldsymbol{x}) - \rho\right)$$

以上就是 libSVM 支持的五种 SVM 模型，其中最后一个模型，也就是 one-class SVM，在笔者学习过程中很少遇到，因此下一节专门讨论所谓的分布估计问题。

5 SVM 的分布估计

SVM 可以在不提供先验概率的情况下对标签数据（和分类任务中的目标值）进行分布估计，我们这里简单介绍这些问题的思路以及 libSVM 中相应的算法。

5.1 k 分类问题的概率估计

给定共 k 类数据，对于任意数据 $\boldsymbol{x}$，我们的目标是估计出

$$p_i = \Pr(y = i|\boldsymbol{x}), i = 1, \cdots, k \tag{30}$$

我们用“一对一”的方法，先估计成对概率：

$$r_{ij} \approx \Pr(y = i|y = i \text{ or } j, \boldsymbol{x}) \tag{31}$$

我们做出假设，r_{ij} 可以写成如下形式：

$$r_{ij} \approx \frac{1}{1 + \exp(A\hat{f} + B)} \tag{32}$$

其中：

$$\hat{f} = f(\boldsymbol{x}) = \sum_{i=1}^{l} \alpha_i y_i K(\boldsymbol{x}_i, \boldsymbol{x}) + b \tag{33}$$

也就是决策函数。然后我们利用基于训练数据的极大似然估计法，估计出参数 A 和 B。考虑到可能会过拟合，因此 libSVM 会先采用 5 折交叉验证去获取 $\hat{f}$。

在得到所有的 r_{ij} 后，我们便着手寻找一组能够最契合 r_{ij} 的概率分布 $[p_1, p_2, \cdots, p_k]$，这相当于求解下面的优化问题：

$$\begin{aligned} \min_{\boldsymbol{p}} \quad & \frac{1}{2} \sum_{i=1}^{k} \sum_{j:j \neq i} (r_{ji} p_i - r_{ij} p_j)^2 \\ \text{subject to} \quad & p_i \geq 0, \forall i \\ & \sum_{i=1}^{k} p_i = 1 \end{aligned} \tag{34}$$

该问题基于下面的概率等式（不难证明）：

$$\Pr(y = j|y = i \text{ or } y = j, \boldsymbol{x}) \cdot \Pr(y = i|\boldsymbol{x}) = \Pr(y = i|y = i \text{ or } y = j, \boldsymbol{x}) \cdot \Pr(y = j|\boldsymbol{x}) \tag{35}$$

接着将原问题重构成矩阵形式：

$$\begin{aligned} \min_{\boldsymbol{p}} \quad & \frac{1}{2} \boldsymbol{p}^\top Q \boldsymbol{p} \\ & Q_{ij} = \begin{cases} \sum_{s:s \neq i} r_{si}^2 & \text{if } i = j \\ -r_{ji} r_{ij} & \text{else} \end{cases} \end{aligned} \tag{36}$$

可以证明 p_i 非负的约束是冗余的。这样只剩下 $\sum p_i = 1$ 的约束，设其对应的拉格朗日乘子为 b，从而直接写出最优性条件：

$$\begin{bmatrix} Q & \boldsymbol{e} \\ \boldsymbol{e}^\top & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{p} \\ b \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ 1 \end{bmatrix} \tag{37}$$

其中 $\boldsymbol{e}$ 是 k 维全 1 向量。除了使用高斯消去法去解这个方程组，我们也可以使用普通的迭代法，以更好地在计算机上解决。由

$$\begin{aligned} Q\boldsymbol{p} + b\boldsymbol{e} &= 0 \\ -\boldsymbol{p}^\top Qp &= b\boldsymbol{p}^\top \boldsymbol{e} \\ &= b \end{aligned} \tag{38}$$

从而最优解 $\boldsymbol{p}$ 满足

$$(Q\boldsymbol{p})_t - \boldsymbol{p}^\top Q\boldsymbol{p} = Q_{tt}p_t + \sum_{j:j\neq t} Q_{tj}p_j - \boldsymbol{p}^\top Q\boldsymbol{p} \tag{39}$$

我们根据这个等式提出 $\boldsymbol{p}$ 的迭代算法：

Algorithm 1

Initialize $\boldsymbol{p}$ randomly with $\sum_{i=1}^{k} p_i = 1, p_i \geq 0, \forall i$
repeat
  $i \leftarrow 1$
  repeat
    $p_i \leftarrow p_i + \frac{1}{Q_{tt}}\big[-(Q\boldsymbol{p})_t + \boldsymbol{p}^\top Q\boldsymbol{p}\big]$
    $i \leftarrow i + 1$
  until $i > k$
until $\max_t |(Q\boldsymbol{p})_t - \boldsymbol{p}^\top Q\boldsymbol{p}| < \frac{0.005}{k}$

算法迭代 p_i 时是将下面两个步骤融合：

$$\begin{aligned} p_i &\leftarrow \frac{1}{Q_{tt}}\big[-\sum_{j:j\neq t} Q_{tj}p_j + \boldsymbol{p}^\top Q\boldsymbol{p}\big] \\ \boldsymbol{p} &\leftarrow \frac{1}{\sum p_i}\boldsymbol{p} \quad \text{(normalization)} \end{aligned}$$

此外，考虑到迭代终止条件（即满足线性方程 (37)）过于严苛，我们提出一个收敛阈值：

$$\|Q\boldsymbol{p} - \boldsymbol{p}^\top Q\boldsymbol{p}\boldsymbol{e}\|_\infty = \max_t |(Q\boldsymbol{p})_t - \boldsymbol{p}^\top Q\boldsymbol{p}| < \frac{0.005}{k}$$

利用 k 来控制收敛。

5.2 回归问题的噪声估计

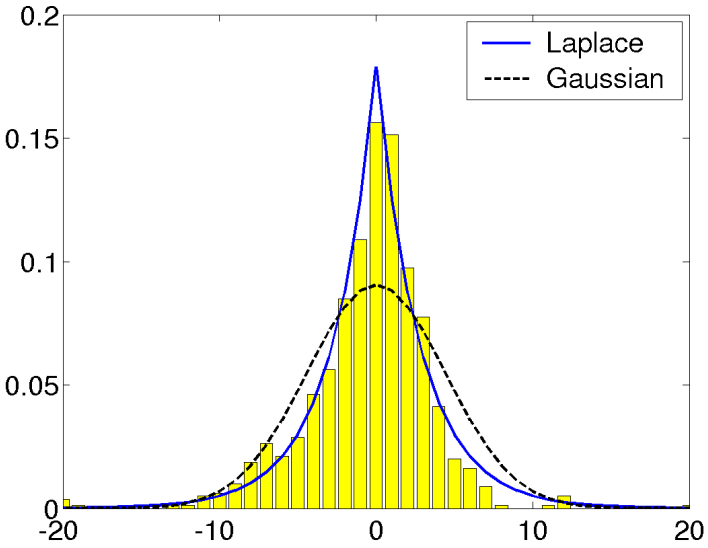
我们假定数据集 $\mathcal{D}$ 是从下面的模型采集得来：

$$y_i = f(\boldsymbol{x}_i) + \delta_i \tag{40}$$

其中 $f(\boldsymbol{x})$ 是潜在的未知函数，δ_i 来自一个独立同分布的随机噪声。给定测试数据 $\boldsymbol{x}$，我们希望估计出 $\Pr(y|\boldsymbol{x}, \mathcal{D})$，从而完成一些概率分布相关任务，比如区间估计：估计出

$$y \in [f(\boldsymbol{x}) - \Delta, f(\boldsymbol{x}) + \Delta]$$

的概率。我们设 $\hat{f}$ 为 SVR 对训练集 $\mathcal{D}$ 学习后得到的拟合函数，然后设 $\zeta = \zeta(\boldsymbol{x}) \equiv y - \hat{f}(\boldsymbol{x})$ 为预测误差。这里需要用交叉验证来减小偏差使得 ζ_i 更准确。根据实验得到下面的直方图：



libSVM 采用零均值的拉普拉斯分布来拟合误差：

$$p(z) = \frac{1}{2\sigma} \exp(-\frac{|z|}{\sigma}) \tag{41}$$

其中的参数 σ 可以利用极大似然法去估计：

$$\sigma = \frac{\sum_{i=1}^{l} |\zeta_i|}{l} \tag{42}$$

于是我们有

$$y = \hat{f}(\boldsymbol{x}) + z \tag{43}$$

其中 z 是满足参数为 σ 的拉普拉斯分布。

6 SMO 算法

SMO（Sequential Minimal Optimization）是求解 SVM 问题的高效算法之一，libSVM 采用的正是该算法。SMO 算法其实是一种启发式算法：先选择两个变量 α_i 和 α_j，然后固定其他

参数，从而将问题转化成一个二变量的二次规划问题。求出能使此时目标值最优的一对 α_i 和 α_j 后，将它们固定，再选择两个变量，直到目标值收敛。这一部分以 C-SVC 作为研究对象，讨论 SMO 算法的过程。

6.1 朴素的 SMO 算法

这里的朴素是指变量选择方面，因为我们可以通过二重循环穷举选择，适用于变量少的情况。假设我们选择了变量 α_1 和 α_2，将其代入 (23) 式的目标函数：

$$\begin{aligned} W(\alpha_1,\alpha_2) &= \sum_{i=1}^{l}\alpha_i - \frac{1}{2}\sum_{i=1}^{l}\sum_{j=1}^{l}\alpha_i\alpha_j y_i y_j K(\boldsymbol{x}_i,\boldsymbol{x}_j) \\ &= \alpha_1+\alpha_2+\sum_{i=3}^{l}\alpha_i - \frac{1}{2}\big[\alpha_1\alpha_1 y_1 y_1 K(\boldsymbol{x}_1,\boldsymbol{x}_1) + 2\alpha_1 y_1\alpha_2 y_2 K(\boldsymbol{x}_1,\boldsymbol{x}_2 \\ &\quad + \alpha_2 y_2\alpha_2 y_2 K(\boldsymbol{x}_2,\boldsymbol{x}_2) + 2\alpha_1 y_1\sum_{i=3}^{l}\alpha_i y_i K(\boldsymbol{x}_1,\boldsymbol{x}_i) \\ &\quad + 2\alpha_2 y_2\sum_{i=3}^{l}\alpha_i y_i K(\boldsymbol{x}_2,\boldsymbol{x}_i) + \sum_{i=3}^{l}\sum_{j=3}^{l}\alpha_i\alpha_j y_i y_j K(\boldsymbol{x}_i,\boldsymbol{x}_j)\big] \end{aligned} \tag{44}$$

为了表达方便，我们将 $K(\boldsymbol{x}_i,\boldsymbol{x}_j)$ 简写成 K_{ij}。根据 $y_i^2=1$ 的性质，我们将函数进一步化简成

$$\begin{aligned} W(\alpha_1,\alpha_2) = \alpha_1+\alpha_2 - \frac{1}{2}\big[&\alpha_1^2K_{11} + 2\alpha_1 y_1\alpha_2 y_2 K_{12} + \alpha_2^2 K_{22} \\ &+ 2\alpha_1 y_1\sum_{i=3}^{l}\alpha_i y_i K_{1i} + 2\alpha_2 y_2\sum_{i=3}^{l}\alpha_i y_i K_{2i}\big] + \text{Const} \end{aligned} \tag{45}$$

其中 Const 是常数，在求极值点过程中可以忽略，即令 $f(\alpha_1,\alpha_2)\leftarrow f(\alpha_1,\alpha_2)-\text{Const}$。考虑后约束条件中后 $l-2$ 个变量被固定，引入常数 C：

$$\begin{aligned} \alpha_1 y_1+\alpha_2 y_2 &= -\sum_{i=3}^{m}\alpha_i y_i = C \\ \alpha_1 y_1 &= C-\alpha_2 y_2 \\ \alpha_1 &= y_1(C-\alpha_2 y_2) \end{aligned} \tag{46}$$

将 α_1 用 α_2 的函数代入，从而将 $W(\alpha_1,\alpha_2)$ 变成 $W(\alpha_2)$：

$$\begin{aligned} W(\alpha_2) = y_1(C-\alpha_2 y_2)+\alpha_2 - \frac{1}{2}\big[&(C-\alpha_2 y_2)^2K_{11} + 2(C-\alpha_2 y_2)\alpha_2 y_2 K_{12} + \alpha_2^2 K_{22} + \\ &2(C-\alpha_2 y_2)\sum_{i=3}^{m}\alpha_i y_i K_{1i} + 2\alpha_2 y_2\sum_{i=3}^{m}\alpha_i y_i K_{2i}\big] \end{aligned} \tag{47}$$

对 $f(\alpha_2)$ 求导并令其为 0：

$$
\begin{aligned}
W'(\alpha_2) &= -y_1y_2 + 1 - \frac{1}{2}\big[-2(C-\alpha_2y_2)y_2K_{11} + 2Cy_2K_{12} - 4\alpha_2K_{12} + 2\alpha_2K_{22} \\
&\quad - 2y_2\sum_{i=3}^{m}\alpha_iy_iK_{1i} + 2y_2\sum_{i=3}^{m}\alpha_iy_iK_{2i}\big] \\
&= 1 - y_1y_2 + Cy_2K_{11} - \alpha_2K_{11} - Cy_2K_{12} + 2\alpha_2K_{12} - \alpha_2K_{22} + y_2\sum_{i=3}^{m}\alpha_iy_iK_{1i} \\
&\quad - y_2\sum_{i=3}^{m}\alpha_iy_iK_{2i} \\
&= 0
\end{aligned} \tag{48}
$$

得到等式：

$$
\begin{aligned}
(K_{11} - 2K_{12} + K_{22})\alpha_2 &= 1 - y_1y_2 + Cy_2K_{11} - Cy_2K_{12} + y_2\sum_{i=3}^{m}\alpha_iy_iK_{1i} - y_2\sum_{i=3}^{m}\alpha_iy_iK_{2i} \\
&= y_2(y_2 - y_1 + CK_{11} - CK_{12} + \sum_{i=3}^{m}\alpha_iy_iK_{1i} - \sum_{i=3}^{m}\alpha_iy_iK_{2i})
\end{aligned} \tag{49}
$$

这里只要将 C 用 $-\sum_{i=1}^{3}\alpha_iy_i$ 替代，就可以得到 α_2 的解析解，但这样的计算代价是巨大的，同时对于计算机来说，迭代算法更加适合它的计算方法，因此我们将 C 设为 $\alpha_1^{\text{old}}y_1+\alpha_2^{\text{old}}y_2$，从而解出新的 α_2，也就是 α_2^{new}：

$$
\begin{aligned}
(K_{11} - 2K_{12} + K_{22})\alpha_2^{\text{new}} = y_2\Bigg(&y_2 - y_1 + (\alpha_1^{\text{old}}y_1 + \alpha_2^{\text{old}}y_2)K_{11} - \\
&(\alpha_1^{\text{old}}y_1 + \alpha_2^{\text{old}}y_2)K_{12} + \sum_{i=3}^{m}\alpha_iy_iK_{1i} - \sum_{i=3}^{m}\alpha_iy_iK_{2i}\Bigg)
\end{aligned} \tag{50}
$$

考虑支持向量机的表达式：

$$
f(\boldsymbol{x}) = \sum_{i=1}^{m}\alpha_iy_iK(\boldsymbol{x}_i, \boldsymbol{x}) + b \tag{51}
$$

所以我们就有

$$
\begin{aligned}
f(\boldsymbol{x}_1) &= \sum_{i=1}^{m}\alpha_iy_iK_{1i} + b \\
f(\boldsymbol{x}_2) &= \sum_{i=1}^{m}\alpha_iy_iK_{2i} + b
\end{aligned} \tag{52}
$$

用上式代替迭代式右端的两个求和部分：

$$
\begin{aligned}
(K_{11}-2K_{12}+K_{22})\alpha_2^{\text{new}} = y_2\Big(&y_2-y_1+(\alpha_1^{\text{old}}y_1+\alpha_2^{\text{old}}y_2)K_{11}- \\
&(\alpha_1^{\text{old}}y_1+\alpha_2^{\text{old}}y_2)K_{12}+f(x_1)-\alpha_1^{\text{old}}y_1K_{11}-\alpha_2^{\text{old}}y_2K_{12}-b \\
&-f(x_2)+\alpha_1^{\text{old}}y_1K_{12}+\alpha_2^{\text{old}}y_2K_{22}+b\Big) \\
=y_2\Big(&f(x_1)-y_1-\big(f(x_2)-y_2\big)+\alpha_2^{\text{old}}y_2(K_{11}-2K_{12}+K_{22})\Big)
\end{aligned}
\tag{53}
$$

这里对 $f(x_i)-y_i$ 有多种处理和理解方式，在《统计学习方法》中，它被设为 E_i，表示预测与实际的距离，理解为一种误差（在分类问题中没有误差这一概念）；但我更喜欢 libSVM 论文中的处理方式，也就是写成梯度的形式：

$$
\begin{aligned}
-y_i\nabla f(\alpha)_i+y_j\nabla f(\alpha)_j &= -y_i(\sum_{k=1}^{m}\alpha_k y_k y_i K_{ki}-1)+y_j(\sum_{k=1}^{m}\alpha_k y_k y_j K_{kj}-1) \\
&= \sum_{k=1}^{m}\alpha_k y_k K_{kj}+b-y_j-\sum_{k=1}^{m}\alpha_k y_k K_{ki}-b+y_i \\
&= -(f(\boldsymbol{x}_i)-y_i)+(f(\boldsymbol{x}_j)-y_j)
\end{aligned}
\tag{54}
$$

并将该值设为 b_{ij}。此外设

$$
a_{ij}=K_{ii}-2K_{ij}+K_{jj} \tag{55}
$$

从而我们能将 (53) 写成下面的形式：

$$
\alpha_2^{\text{new}}=\alpha_2^{\text{old}}+y_2\frac{b_{21}}{a_{21}}
$$

从而参数更新公式：

$$
\begin{cases}
\alpha_1^{\text{new}}=\alpha_2^{\text{old}}+y_1\dfrac{b_{12}}{a_{12}} \\
\alpha_2^{\text{new}}=\alpha_2^{\text{old}}-y_2\dfrac{b_{12}}{a_{12}}
\end{cases}
\tag{56}
$$

以上就是 SMO 迭代基本思路。但有一点值得注意，由于始终有 $0\le\alpha_i\le C$ 的限制存在，当我们迭代时，是有可能会跑出这个范围，此时应该及时停下来。

6.2 变量选择

一个显然的事实是，选择不同的变量进行迭代会影响目标函数值趋向于最优的速度，那么如何选择变量成为我们接下来要讨论的话题。这一过程也被称作“工作集选择”（Working set selection），$\{i,j\}$ 被称作工作集。

6.2.1 变量选择思路

《统计学习方法》中提到了变量选择的基本思想，为笔者在阅读 libSVM 论文中的大量公式前对这个问题有大致的认识。我们希望选取违反 KKT 条件最严重的变量作为我们的 α_i。由于在找到最优解之前，总存在一个 α_i 不满足下面的 KKT 条件：

$$\begin{aligned} \alpha_i = 0 &\Leftrightarrow y_i f(\boldsymbol{x}_i) \geq 1 \\ 0 < \alpha_i < C &\Leftrightarrow y_i f(\boldsymbol{x}_i) = 1 \\ \alpha_i = 0 &\Leftrightarrow y_i f(\boldsymbol{x}_i) \leq 1 \end{aligned} \tag{57}$$

我们希望 SMO 算法通过迭代使这样的 α_i 满足 KKT 条件，因此选择违法条件最严重的变量是好的选择。此外，对于上面三种情况，我们更偏向于选择第二种情况对应的变量，也就是没有到达边界（0 或 C），它会有更大的活动空间。

当已经选择好 α_i 后，我们希望找出更新时有最大变化的 α_j，也就是倾向于寻找绝对值最大的 $\frac{b_{ij}}{a_{ij}}$，这样迭代会更快。

6.2.2 基于一阶近似的变量选择

在前面选择第一个变量时，我们只提到了选择 “违反 KKT 条件最严重” 的样本点，那么我们需要找到一种方法来度量违反 KKT 条件的严重性。由此我们引入基于一阶近似（First order approximation）的变量选择法。

我们写出 C-SVC 的对偶问题的拉格朗日函数：

$$\mathcal{L}(\alpha, \lambda, \mu, \eta) = f(\alpha) - \sum_{i=1}^{m} \lambda_i \alpha_i + \sum_{i=1}^{m} \mu_i (\alpha_i - C) + \eta y^\top \alpha \tag{58}$$

其中 λ、μ、η 均为非负向量。如果 α 是原问题的解，那么它必然是 $\mathcal{L}$ 的有一个驻点，也就是梯度为零，整理后有：

$$\begin{aligned} \nabla f(\alpha) + \eta y &= \lambda - \mu \\ \lambda_i \alpha_i = 0, \mu_i (C - \alpha_i) = 0, &\alpha_i \geqslant 0, \mu_i \geqslant 0, i = 1, \cdots, m \end{aligned} \tag{59}$$

我们也不难求得 $f(\alpha)$ 的梯度为 $Q\alpha - e$。从而我们可以将上面的条件重写成：

$$\begin{aligned} \nabla f(\alpha)_i + \eta y_i &\geqslant 0 \qquad \text{if } \alpha_i < C \\ \nabla f(\alpha)_i + \eta y_i &\leqslant 0 \qquad \text{if } \alpha_i > 0 \end{aligned} \tag{60}$$

我们定义关于 α 的两个集合 I_{up} 和 I_{low}：

$$\begin{aligned} I_{\text{up}}(\alpha) &= \{t | \alpha_t < C, y_t = 1\} \cup \{t | \alpha_t > 0, y_t = -1\} \\ I_{\text{low}}(\alpha) &= \{t | \alpha_t < C, y_t = -1\} \cup \{t | \alpha_t > 0, y_t = 1\} \end{aligned} \tag{61}$$

可以推出这样的性质：$I_{\text{up}}(\alpha)$ 中的所有元素 i 都满足

$$-y_i \nabla f(\alpha)_i \leqslant \eta$$

而 $I_{\text{low}}(\alpha)$ 中的所有元素 i 都满足

$$-y_i \nabla f(\alpha)_i \geqslant \eta$$

α 是原问题的解当且仅当

$$m(\alpha) \leqslant M(\alpha) \tag{62}$$

其中

$$m(\alpha) = \max_{i \in I_{\text{up}}(\alpha)} -y_i \nabla f(\alpha)_i, M(\alpha) = \min_{i \in I_{\text{low}}(\alpha)} -y_i \nabla f(\alpha)_i \tag{63}$$

但我们前面提到，在求得解之前，这样的等式不会被满足，因此必然会存在这样一对 (i, j)，$i \in I_{\text{up}}(\alpha)$，$j \in I_{\text{low}}(\alpha)$ 但 $-y_i \nabla f(\alpha)_i > -y_j \nabla f(\alpha)_j$，那么我们称这对 (i, j) 为一个违反对（violating pair）。如果最大的一个违反对是 i 和 j，那么我们选择变量 α_i 和 α_j。当然我们也可以采用启发式方法，设置一个容忍值（tolerance）ε，如果在第 k 轮选择变量时有

$$m(\alpha^k) - M(\alpha^k) \leqslant \varepsilon \tag{64}$$

就停止算法。

我们重新审视 “一阶近似” 这个名称，在微积分中，一阶近似指的是用一阶导数（梯度）去近似函数值：

$$f(x + d) \approx f(x) + \nabla f(x)^\top d \tag{65}$$

而之所以称该算法基于一阶近似，是因为最大违反对 (i, j) 是一系列子问题

$$\begin{aligned} \text{Sub}(B) \equiv \min_{d_B} \quad & \nabla f(\alpha^k)_B^\top d_B \\ \text{subject to} \quad & y_B^\top d_B = 0 \\ & d_t \geqslant 0, \text{if } \alpha_t^k = 0, t \in B, \\ & d_t \leqslant 0, \text{if } \alpha_t^k = C, t \in B, \\ & -1 \leqslant d_t \leqslant 1, t \in B \end{aligned} \tag{66}$$

的最优解。这里的下标 B 指的是选定的两个变量对应的数据，比如 $B = \{1, 3\}$，那么 m 维向量 α 就变成 $[\alpha_1, \alpha_3]$。不难发现优化目标函数的由来：

$$\begin{aligned} f(\alpha^k + d) &\approx f(\alpha^k) + \nabla f(\alpha^k)^\top d \\ &= f(\alpha^k) + \nabla f(\alpha^k)_B^\top d_B \end{aligned}$$

正是一阶近似公式中的一阶近似项。

6.2.3 基于二阶近似的变量选择

libSVM 的 working set selection 是根据 second order information 来选择的，它在选择 i 采用的是前面提到的一阶近似方法，而在选择 j 时，不仅要求其与 i 构成违反对，还需要它能够最大程度减小目标函数。

函数的二阶近似：

$$f(x + d) = f(x) + \nabla f(x)^\top d + \frac{1}{2} d^\top \nabla^2 f(x) d \tag{67}$$

类似的，我们想寻找一系列优化问题

$$\begin{aligned}\text{Sub}(B) \equiv \min_{d_B} \quad & \frac{1}{2} d_B^\top \nabla^2 f(\alpha)_{BB} d_B + \nabla f(\alpha^k)_B^\top d_B \\ \text{subject to} \quad & y_B^\top d_B = 0 \\ & d_t \geqslant 0, \text{if } \alpha_t^k = 0, t \in B, \\ & d_t \leqslant 0, \text{if } \alpha_t^k = C, t \in B.\end{aligned} \tag{68}$$

的最优解，考虑到我们已经确定了 i，那么子问题共 $m-1$ 个。形式化 i 和 j 的选取：

$$\begin{aligned} & i \in \arg\max_t \{-y_t \nabla f(\alpha^k)_t | t \in I_{\text{up}}(\alpha^k)\} \\ & j \in \arg\min_t \{\text{Sub}(i,t) | t \in I_{\text{low}}(\alpha^k), -y_t \nabla f(\alpha^k)_t < -y_i \nabla f(\alpha^k)_i\}\end{aligned} \tag{69}$$

我们下面的任务就是尝试求解子问题 $\text{Sub}(i,j)$

$$\begin{aligned}\text{Sub}(B) &= \frac{1}{2} d_B^\top \nabla^2 f(\alpha)_{BB} d_B + \nabla f(\alpha^k)_B^\top d_B \\ &= \frac{1}{2} \begin{bmatrix} d_i & d_j \end{bmatrix} \begin{bmatrix} \frac{\nabla f(\alpha)_i}{\nabla \alpha_i} & \frac{\nabla f(\alpha)_i}{\nabla \alpha_j} \\ \frac{\nabla f(\alpha)_j}{\nabla \alpha_i} & \frac{\nabla f(\alpha)_j}{\nabla \alpha_j} \end{bmatrix} \begin{bmatrix} d_i \\ d_j \end{bmatrix} + \begin{bmatrix} \nabla f(\alpha)_i & \nabla f(\alpha)_j \end{bmatrix} \begin{bmatrix} d_i \\ d_j \end{bmatrix} \\ &= \frac{1}{2} \begin{bmatrix} d_i & d_j \end{bmatrix} \begin{bmatrix} y_i^2 K_{ii} & y_i y_j K_{ij} \\ y_j y_i K_{ji} & y_j^2 K_{jj} \end{bmatrix} \begin{bmatrix} d_i \\ d_j \end{bmatrix} + \begin{bmatrix} \nabla f(\alpha)_i & \nabla f(\alpha)_j \end{bmatrix} \begin{bmatrix} d_i \\ d_j \end{bmatrix}\end{aligned} \tag{70}$$

这里令 $\hat{d}_i = y_i d_i$，$\hat{d}_j = y_j d_j$，又由 $y_B^\top d_B = 0$，我们得到 $d_i = -d_j$，从而我们进一步化简：

$$\begin{aligned}\text{Sub}(B) &= \frac{1}{2}(K_{ii} - 2K_{ij} + K_{jj})\hat{d}_j^2 + [-y_i \nabla f(\alpha)_i + y_j \nabla f(\alpha)_j]\hat{d}_j \\ &= \frac{1}{2} a_{ij} \hat{d}_j^2 + b_{ij} \hat{d}_j \\ f(\hat{d}_j) &= \frac{1}{2} a_{ij} (\hat{d}_j + \frac{b_{ij}}{a_{ij}})^2 - \frac{b_{ij}^2}{2a_{ij}}\end{aligned} \tag{71}$$

从而对于每个 $\text{Sub}(B)$，对应的最优值为

$$-\frac{b_{ij}^2}{2a_{ij}} = -\frac{[-y_i \nabla f(\alpha)_i + y_j \nabla f(\alpha)_j]^2}{2(K_{ii} - 2K_{ij} + K_{jj})} \tag{72}$$

j 的选取就可以改写成

$$j \in \arg\min_t \{-\frac{b_{it}^2}{a_{it}} | t \in I_{\text{low}}(\alpha^k), -y_t \nabla f(\alpha^k)_t < -y_i \nabla f(\alpha^k)_i\} \tag{73}$$

这也就是 LIBSVM 中的变量选取方法。

6.3 α 的更新与剪辑

6.3.1 更新

我们将更新公式 (56) 中 b_{ij} 用其实际含义替换，发现需要分类讨论：

- 如果 $y_i \neq y_j$：

$$\begin{aligned} \alpha_i^{k+1} &= \alpha_i^k + y_i \frac{b_{ij}}{a_{ij}} \\ &= \alpha_i^k + \frac{-\nabla f(\alpha)_i - \nabla f(\alpha)_j}{a_{ij}} \\ \alpha_j^{k+1} &= \alpha_j^k - y_j \frac{b_{ij}}{a_{ij}} \\ &= \alpha_j^k + \frac{-\nabla f(\alpha)_i - \nabla f(\alpha)_j}{a_{ij}} \end{aligned} \tag{74}$$

定义

$$\delta_{y_i \neq y_j} = \frac{-\nabla f(\alpha)_i - \nabla f(\alpha)_j}{a_{ij}} \tag{75}$$

- 如果 $y_i = y_j$，类似的，我们有

$$\begin{aligned} \alpha_i^{k+1} &= \alpha_i^k - \frac{\nabla f(\alpha)_i - \nabla f(\alpha)_j}{a_{ij}} \\ \alpha_j^{k+1} &= \alpha_j^k + \frac{\nabla f(\alpha)_i - \nabla f(\alpha)_j}{a_{ij}} \end{aligned} \tag{76}$$

定义

$$\delta_{y_i = y_j} = \frac{\nabla f(\alpha)_i - \nabla f(\alpha)_j}{a_{ij}} \tag{77}$$

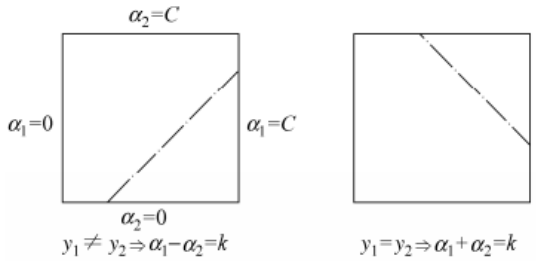
6.3.2 剪辑

我们在前面提到，由于 $\alpha_i \in [0, C]$，因此我们在更新时需要将 α 限制在该区间内，该步骤称为剪辑（Clipping），我们在6.1仅提到了这一步骤存在，这里进行一个深入讲解。

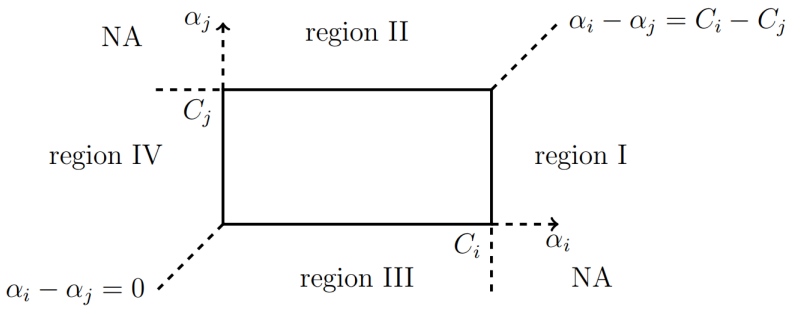
我们只讨论 $y_i \neq y_j$ 的情况，$y_i = y_j$ 的情况下讨论是类似的。由上面的更新公式，我们有：

$$\alpha_i^{k+1} - \alpha_i^k = \alpha_j^{k+1} - \alpha_j^k \tag{78}$$

此时两变量的状态 (α_i, α_j) 对应左图的虚线（图片摘自《统计学习方法》，图中是将 x_1 作为 x_i，x_2 作为 x_j）：



而更一般的情形如下图所示，$\alpha_i \in [0, C_i]$，每个变量的边界不一定相同，常常在类别不平衡的问题中使用：



显然 (α_i, α_j) 必须在图中的矩形中，而在更新的过程中，参数对存在跳出矩形区域的可能，也就是 Region I 到 Region IV 这四个不合理区域，需要将这些区域的点扳回矩形中；NA 则是不可达点，因为 $y_i \neq y_j$ 的情况下参数对只能向左下方或者右上方跳动。

以处于 Region I 的点为例，此时有

$$\begin{cases} \alpha_i > C_i \\ \alpha_i - \alpha_j > C_i - C_j \end{cases} \tag{79}$$

我们将 α_i 挪到矩形边缘中，同时保持 α_i 和 α_j 的关系：

$$\begin{cases} \alpha_i^{k+1} = C_i \\ \alpha_j^{k+1} = C_i - (\alpha_i^k - \alpha_j^k) \end{cases} \tag{80}$$

同理，对于 Region II，我们有

$$\begin{cases} \alpha_i^{k+1} = C_j + \alpha_i^k - \alpha_j^k \\ \alpha_j^{k+1} = C_j \end{cases} \tag{81}$$

Region III：

$$\begin{cases} \alpha_i^{k+1} = \alpha_i^k - \alpha_j^k \\ \alpha_j^{k+1} = 0 \end{cases} \tag{82}$$

Region IV:

$$\begin{cases} \alpha_i^{k+1} = 0 \\ \alpha_j^{k+1} = -(\alpha_i^k - \alpha_j^k) \end{cases} \tag{83}$$

对于 $y_i = y_j$ 的情况有类似的处理方式。

7 大规模数据下的 SVM

Thorsten Joachims，也是支持向量机软件包 SVM-Light 的作者，在《Making large-scale SVM learning practical》中提出在面对大规模数据时提高 SVM 训练效率的方案：

- 更有效和更高效的变量选择法；

- 不断地“收缩”问题规模；
- 计算上的改进：比如缓存机制的引入和梯度的增量式更新。

变量选择我们在前面已经提及；缓存机制涉及到操作系统的知识，我们放到后面讨论；因此这里主要探讨“收缩”和梯度更新技巧。

7.1 Shrink 方法

在解决支持向量机问题时，我们通常去解决其对偶问题，更具体地说，是去解一个长度为样本个数的 α 向量。但在大样本学习过程中，这显然会导致数据存储和运算量过大的问题。幸运的是，SVM 的解存在稀疏性，也就是最终模型仅与支持向量有关。Joachims 提出的 Shrinking 方法能够有效缩短 SVM 的训练时间，并将其应用到他开发的 SVM-light 中，该方法同时也被 libSVM 所采用。

7.1.1 问题引入

从 SVM 的对偶问题 OP1 的目标函数

$$W(\boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{\alpha}^\top Q\boldsymbol{\alpha} - \boldsymbol{e}^\top\boldsymbol{\alpha}$$

可以看出解决该问题至少要为矩阵 Q 和 $\boldsymbol{\alpha}$ 提供存储空间，而 $Q_{ij} \equiv y_i y_j K(x_i, x_j)$，假设样本数为 1000，那么我们至少需要为其提供 $1000 \times 1000 \times 4 + 1000 \times 4$ 个字节，也就是约 4 MB 的存储空间，显然是非常不合理的。

Joachims 从算法上基于以下事实提出了 Shrinking 方法来缓解这一问题：

- 支持向量（SVs）的数量要比训练样本少得多；
- 许多支持向量对应的 α_i 的值都是其上界 C。

在硬间隔 SVM 中，所有 $\alpha_i > 0$ 对应的样本点 x_i 都是支持向量，它们都位于分类间隔上，反之也成立；而在软间隔 SVM 中，支持向量不一定全部分布在分类间隔上，在分类间隔中甚至是分类错误的向量也被称作支持向量，这些向量的特征就是其对应的 $\alpha_i = C$。

我们将样本向量分为三类：

1. X 类：支持向量，但 $0 < \alpha_i < C$；
2. Y 类：支持向量，但 $\alpha_i = C$；
3. Z 类：非支持向量，也就是 $\alpha_i = 0$.

因此我们将数据重排：

$$\alpha = \begin{bmatrix}\alpha_X \\ \alpha_Y \\ \alpha_Z\end{bmatrix} = \begin{bmatrix}\alpha_X \\ C\mathbf{1} \\ \mathbf{0}\end{bmatrix}, y = \begin{bmatrix}y_X \\ y_Y \\ y_Z\end{bmatrix}, Q = \begin{bmatrix}Q_{XX} & Q_{XY} & Q_{XZ} \\ Q_{YX} & Q_{YY} & Q_{YZ} \\ Q_{ZX} & Q_{ZY} & Q_{ZZ}\end{bmatrix} \tag{84}$$

我们从而重写 $W(\alpha)$：

$$
\begin{aligned}
W(\alpha) &= \frac{1}{2}\alpha^\top Q\alpha - C\mathbf{1}^\top\alpha \\
&= \frac{1}{2}\sum_{m\in\{X,Y,Z\}}\sum_{n\in\{X,Y,Z\}}\alpha_m^\top Q_{mn}\alpha_n - C\mathbf{1}^\top\alpha_X - C\mathbf{1}^\top\alpha_Y - C\mathbf{1}^\top\alpha_Z \\
&= \frac{1}{2}\sum_{m\in\{X,Y\}}\sum_{n\in\{X,Y\}}\alpha_m^\top Q_{mn}\alpha_n - C\mathbf{1}^\top\alpha_X - C\mathbf{1}^\top\alpha_Y \\
&= \frac{1}{2}\alpha_X^\top Q_{XX}\alpha_X + C\alpha_X^\top Q_{XY}\mathbf{1} + \frac{1}{2}C^2\mathbf{1}^\top Q_{YY}\mathbf{1} - C\alpha_X^\top\mathbf{1} - |Y|C \\
&= \frac{1}{2}\alpha_X^\top Q_{XX}\alpha_X + C\alpha_X^\top\big(Q_{XY}\mathbf{1} - \mathbf{1}\big) + \frac{1}{2}C^2\mathbf{1}^\top Q_{YY}\mathbf{1} - |Y|C
\end{aligned}
\tag{85}
$$

考虑到后面两项为常数，我们重写对偶问题：

$$
\begin{aligned}
\min_{\alpha_X}\quad & \frac{1}{2}\alpha_X^\top Q_{XX}\alpha_X + C\alpha_X^\top\big(Q_{XY}\mathbf{1} - \mathbf{1}\big) \\
\text{subject to}\quad & \alpha_X^\top y_X + C\mathbf{1}^\top y_Y = 0 \\
& 0 \le \alpha_X \le C\mathbf{1}
\end{aligned}
\tag{86}
$$

可以发现，问题的规模大幅度减小，矩阵和向量的维数数量级只由支持向量个数决定。这一过程便称为收缩（Shrinking）。

7.1.2 启发式方法

虽然我们可以通过 Shrinking 来缩小问题规模，但它基于我们已知 α_i 是属于 α_X、α_Y 或 α_Z，这对于算法是很难判定的。到目前为止，还不清楚该算法如何识别哪些样本可以消除，也就是对应的 α_i 为 0 或 C 的样本。我们希望在优化过程的早期找到一些条件，这些条件表明某些变量最终会达到一个界限。由于充分条件未知，采用基于拉格朗日乘子估计的启发式方法。

设 A 是当前满足 $\alpha_i \in (0, C)$ 的集合：

$$
A = \{\alpha_i | 0 < \alpha < C, i = 1, \cdots, l\} \tag{87}
$$

然后设估计值 λ^{eq}：

$$
\lambda^{eq} = \frac{1}{|A|}\sum_{i\in A}\left[y_i - \sum_{j=1}^{l}\alpha_j y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)\right] \tag{88}
$$

注意到我们可以用 λ^{eq} 作为 SVM 决策函数中的 bias，也就是 b。以此设置拉格朗日乘子，也就是 α_i 的上下界：

$$
\begin{aligned}
\lambda_i^{lo} &= y_i\left(\left[\sum_{j=1}^{l}\alpha_j y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)\right] + \lambda^{eq}\right) - 1 \\
\lambda_i^{up} &= -y_i\left(\left[\sum_{j=1}^{l}\alpha_j y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)\right] + \lambda^{eq}\right) + 1
\end{aligned}
\tag{89}
$$

给定一正整数 h，此时考虑 SMO 迭代过程的前 h 个循环，在这 h 个循环中，对于某个 i，如果都有 $\lambda_i^{lo} > 0$ 且 $\lambda_i^{up} > 0$（也可以用一个极小阈值 ε 代替 0），那么我们就有信心将其删除。也就是说 α_i 已经是最优的，它们可以被固定，从而不需要对其梯度等值进行计算。

由于启发式算法没有定理去证明合理性，必然会存在删错了的情况。因此在 (86) 收敛后，对被删除变量的最优条件进行检查；如有必要，则会在这一次迭代中重新进行一次优化。

7.2 梯度重构策略

Joachims 称之为梯度的增量式更新（Incremental updates of the gradient），而在 libSVM 中被称为梯度重构（Gradient reconstruction）。

7.2.1 一个例子

在线性回归中，我们要解决的是下面的优化问题：

$$\min_{\boldsymbol{w}} \quad f(\boldsymbol{w}) = \frac{1}{2}\|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|^2 \tag{90}$$

该问题固然是有解析解的，但倘若我们用梯度下降法，给定步长 η，在每轮迭代中都有下面的操作：

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \frac{\partial f}{\partial \boldsymbol{w}} \tag{91}$$

也就是

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \boldsymbol{X}^\top (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) \tag{92}$$

如果不采用任何优化方法的话，我们每次都要计算一次 $\boldsymbol{X}^\top(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})$；而实际上相邻两次更新之间梯度变化量是一个很简单的值：

$$\begin{aligned} \Delta\nabla f(\boldsymbol{w}) &= \nabla f(\boldsymbol{w} + \Delta\boldsymbol{w}) - \nabla f(\boldsymbol{w}) \\ &= \boldsymbol{X}^\top\big(\boldsymbol{X}(\boldsymbol{w} + \Delta\boldsymbol{w}) - \boldsymbol{y}\big) - \boldsymbol{X}^\top\big(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\big) \\ &= \boldsymbol{X}^\top\boldsymbol{X}\Delta\boldsymbol{w} \end{aligned} \tag{93}$$

此外，要是我们能将 $\boldsymbol{X}^\top\boldsymbol{X}$ 存储，那么花费在梯度计算上的时间会更少。这就是梯度的增量式更新带来计算负担减少的一个例子。

7.2.2 libSVM 中的梯度重构

libSVM 继承了 Joachims 在 SVM-Light 中使用的增量更新思想，分别对工作集和非工作集的梯度进行更新。对工作集中变量对应的梯度的更新：

```
double delta_alpha_i = alpha[i] - old_alpha_i;
double delta_alpha_j = alpha[j] - old_alpha_j;
for (int k = 0; k < active_size; k++) {
   G[k] += Q_i[k] * delta_alpha_i + Q_j[k] * delta_alpha_j;
}
```

但我们仍然需要这些 inactive 的参数对应的梯度，也就是全部的 $\nabla f(\boldsymbol{x})$，为了减少梯度重构的开销，libSVM 选择在迭代中维护一个向量 $\bar{G} \in \mathbb{R}^l$：

$$\bar{G}_i = C \sum_{j:\alpha_j=C} Q_{ij}, i = 1, \cdots, l \tag{94}$$

从而对于不属于 active 集合的变量 α_i，我们有

$$\begin{aligned} \nabla f(\boldsymbol{\alpha})_i &= \sum_{j=1}^{l} Q_{ij}\alpha_j - 1 \\ &= \sum_{\alpha_j=0} Q_{ij} \cdot 0 + \sum_{\alpha_j=C} CQ_{ij} + \sum_{0<\alpha_j<C} Q_{ij}\alpha_j - 1 \\ &= C \sum_{\alpha_j=C} Q_{ij} + \sum_{0<\alpha_j<C} Q_{ij}\alpha_j - 1 \\ &= \bar{G}_i + \sum_{0<\alpha_j<C} Q_{ij}\alpha_j - 1 \end{aligned} \tag{95}$$

实现通过提前计算 $\bar{G}$ 加快计算梯度：

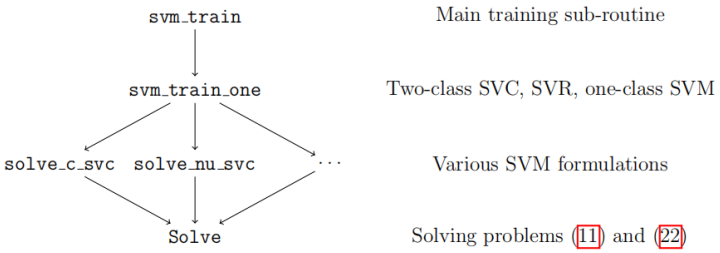
```
{
  bool ui = is_upper_bound(i);
  bool uj = is_upper_bound(j);
  update_alpha_status(i);
  update_alpha_status(j);
  int k;
  if (ui != is_upper_bound(i)) {
    Q_i = Q.get_Q(i, l);
    if (ui)
      for (k = 0; k < l; k++)
        G_bar[k] -= C_i * Q_i[k];
    else
      for (k = 0; k < l; k++)
        G_bar[k] += C_i * Q_i[k];
  }

  if (uj != is_upper_bound(j)) {
    Q_j = Q.get_Q(j, l);
    if (uj)
      for (k = 0; k < l; k++)
        G_bar[k] -= C_j * Q_j[k];
    else
      for (k = 0; k < l; k++)
        G_bar[k] += C_j * Q_j[k];
  }
}
```

这里对更新进行限制：只有状态发生变换的变量，才能参与 $\bar{G}$ 的更新，这是为了减少循环次数。如果 ui 是 false，也就是 α_i 从 active 变成 inactive，对应的就是将对应的增量加入 $\bar{G}$，反之就是从 $\bar{G}$ 中对应 α_i 的增量删除。

8 libSVM 框架简述

libSVM 总体由头文件 svm.h 和源文件 svm.cpp 构成，我们在这里对 libSVM 的总体结构进行剖析。



如图是 libSVM 求解问题的逻辑结构：训练一个复杂的 SVM 模型，也就是 svm_train，实际上以训练一个支持向量机为基础的 (svm_train)，我们提到过 libSVM 中可以求解 5 种 SVM 模型。所以 svm_train 会选择一个 SVM 进行求解，比如 solve_c_svc 便是求解 SVM 分类问题；但所有的支持向量机最终都是解决一个线性约束的二次规划问题，因此万流归宗，所有问题都要用 Solve 函数求解。

9 svm.h

svm.h 对 libSVM 中必要的数据结构和接口函数进行了声明和定义。

9.1 数据结构

svm.h 定义的数据结构有四种：

- svm_node：数据节点；
- svm_problem：数据集；
- svm_parameter：SVM 参数；
- svm_model：SVM 模型.

```
struct svm_node {
    int index;
    double value;
};
```

这个结构体表示一个数据单位，比如我们用下面的代码

```
svm_node data[] = {
    {1, 0.12};
    {2, 1.40};
    {3, 4.00};
```

```
    {-1};
}
```

就构建了下面这条数据：

1	2	3	-1
0.12	1.40	4.00	空

也就是 $\boldsymbol{x}_i = (0.12, 1.40, 4.00)$。之所以使用了 'index'，是为了在大量数据为 0 时节省存储空间，比如下面的代码

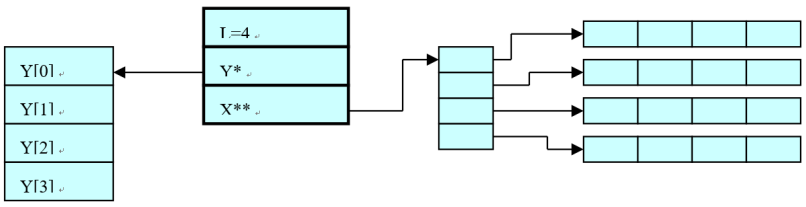
```
svm_node data[] = {
    {1, 0.12};
    {2, 1.40};
    {4, 4.00};
    {-1};
}
```

创建的数据为 $\boldsymbol{x}_i = (0.12, 1.40, 0, 4.00)$；这样设计还有一个好处就是在做点乘的时候可以加快运算速度。

第二个 structure：

```
struct svm_problem {
    int l;
    double *y;
    struct svm_node **x;
};
```

这个结构体表示数据集：l 为数据集的数据量；y 对应分类问题中的标签，或者是回归问题中的输出；而 x 就是特征向量，一个指向指针的指针，存储的是二维表格数据：x[0] 就是数据集第一条数据，x[0][0].value 就是第一条数据的第 x[0][0].index 个数据。示意图如下



对应样本数为 4 的数据集。

第三个 structure：

```
enum { C_SVC, NU_SVC, ONE_CLASS, EPSILON_SVR, NU_SVR }; /* svm_type */
enum { LINEAR, POLY, RBF, SIGMOID, PRECOMPUTED }; /* kernel_type */

struct svm_parameter {
```

```
    int svm_type;
    int kernel_type;
    int degree; /* for poly */
    double gamma; /* for poly/rbf/sigmoid */
    double coef0; /* for poly/sigmoid */

    /* these are for training only */
    double cache_size; /* in MB */
    double eps;     /* stopping criteria */
    double C;       /* for C_SVC, EPSILON_SVR and NU_SVR */
    int nr_weight;  /* for C_SVC */
    int *weight_label; /* for C_SVC */
    double *weight; /* for C_SVC */
    double nu;      /* for NU_SVC, ONE_CLASS, and NU_SVR */
    double p;       /* for EPSILON_SVR */
    int shrinking;  /* use the shrinking heuristics */
    int probability; /* do probability estimates */
};
```

这是一个 SVM 模型的参数，从变量命名和注释不难看出各个成员变量的含义：

- svm_type：支持向量机类别，对应不同的学习任务；
- kernel_type：使用哪种核方法，有线性核、多项式核、RBF 核（高斯核）、Sigmoid 核等；对应的核函数公式：

$$
\begin{aligned}
K_{\text{linear}}(\boldsymbol{x}_i, \boldsymbol{x}_j) &= \boldsymbol{x}_i^\top \boldsymbol{x}_j \\
K_{\text{poly}}(\boldsymbol{x}_i, \boldsymbol{x}_j) &= (\gamma \boldsymbol{x}_i^\top \boldsymbol{x}_j + r)^d \\
K_{\text{RBF}}(\boldsymbol{x}_i, \boldsymbol{x}_j) &= \exp(-\gamma \|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2) \\
K_{\text{Sigmoid}}(\boldsymbol{x}_i, \boldsymbol{x}_j) &= \tanh(\gamma \boldsymbol{x}_i^\top \boldsymbol{x}_j + r)
\end{aligned}
$$

- degree：对应多项式核的 d；
- gamma：对应上面核函数公式的 γ；
- coef0：应公式中的 r.

下面则是训练过程中需要的参数：

- cache_size：用于存放数据所需的内存空间，以 MB 为单位；
- eps：对应工作集选取时用到的容忍值 ε：

$$
m(\boldsymbol{\alpha}^k) - M(\boldsymbol{\alpha}^k) \leq \varepsilon
$$

则认为 $\boldsymbol{\alpha}^k$ 已经是最优解；

- C：惩罚因子 C；

- nr_weight：直译是“权重的数目”，用于正负样本数不平衡时，对不同样本设置不同权重；
- weight_label：同上；
- nu：对应 ν-SVC 和 ν-SVR 中的参数 ν；
- p：不明；
- shrinking：决定训练过程中是否使用 Shrinking 技巧；
- probability：指明训练过程是否需要预报概率，对应前面分布估计，因为 SVM 中的距离没有概率含义.

最后一个 structure 是关于 SVM 的模型参数：

```
struct svm_model {
    struct svm_parameter param; /* parameter */
    int nr_class; /* number of classes, = 2 in regression/one class svm */
    int l;      /* total #SV */
    struct svm_node **SV; /* SVs (SV[l]) */
    double **sv_coef;  /* coefficients for SVs in decision functions
                          (sv_coef[k-1][l]) */
    double *rho;       /* constants in decision functions (rho[k*(k-1)/2]) */
    double *probA;     /* pariwise probability information */
    double *probB;
    int *sv_indices; /* sv_indices[0,...,nSV-1] are values in
                    [1,...,num_traning_data] to indicate SVs in the training
                    set */

    /* for classification only */

    int *label; /* label of each class (label[k]) */
    int *nSV; /* number of SVs for each class (nSV[k]) */
              /* nSV[0] + nSV[1] + ... + nSV[k-1] = l */
    /* XXX */
    int free_sv; /* 1 if svm_model is created by svm_load_model*/
    /* 0 if svm_model is created by svm_train */
};
```

大部分参数含义不难理解：

- param：模型参数，有趣的是它并不是指针形式，一个可能的原因是防止下次训练将参数冲掉；
- nr_class：分类问题的类别数；
- l：支持向量个数；
- SV：所有的支持向量；

- sv_coef：支持向量在决策函数的系数，也就是 $\boldsymbol{\alpha}$；
- rho：对应 ν-SVC 和 One-class SVM 中的 ρ，它和其他问题的决策函数中的 b 互为相反数：$b = -\rho$；
- probA, probB：应该对应的是 r_{ij}，具体不明；
- sv_indices：α_i 对应的数据 x_i 构成的数组，共 l 个；

以下参数仅用于分类任务中：

- label：标签数组；
- nSV：每一类样本中国的支持向量个数，共 nr_class 个元素；
- free_SV：果该模型是由 svm_load_model 导入，free_SV 为 1；如果是训练出来的，free_SV 为 0.

9.2 API 函数

svm.h 中为用户提供的接口函数共有 19 个，但功能总结下来只有 6 种：

- 机器学习相关：svm_train、svm_cross_validation；
- 模型的存取：svm_save_model 和 svm_load_model；
- 获取模型参数：svm_get... 都是这类；
- 利用模型预测：svm_predict_...；
- 模型删除和内存释放：svm_free_model_content、svm_free_and_destroy_model 和 svm_destroy_param
- 调试功能接口：svm_check... 和 svm_set_print_string_function.

svm.h 的内容就是上面这些，svm.h 为用户展现了 libSVM 的基本框架，同时提供主要的 SVM 功能相关接口，但对我们来说这是不够的，我们还是需要深入 svm.cpp 中去理解算法细节。

10 关于 C++ 编程

我们假定读者已经有 C/C++ 简单编程的基础，libSVM 中有很多程序设计手法值得我们去学习，会为我们后面的程序设计工作带来帮助，在此进行收集和总结。

10.1 模板编程

libSVM 使用了模板（template）去编写基础的函数，先是求极大值和极小值的函数：

```
#ifndef min
template <class T>
static inline T min(T x, T y) {
   return (x < y) ? x : y;
}
#endif

#ifndef max
template <class T>
static inline T max(T x, T y) {
   return (x > y) ? x : y;
}
#endif
```

注意到这里的预编译语句：只有标准库没有 min 和 max 函数的情况下才会调用模板，因为不是所有编译器都十分支持泛型编程（比如 Microsoft VC++）。还注意到函数用 static inline 装饰，有效减少调用函数时的开销。包括下面的 swap 函数：

```
template <class T>
static inline void swap(T &x, T &y) {
   T t = x;
   x = y;
   y = t;
}
```

事实上 C++ 的标准库中已经支持泛型的 swap 函数，编写者将这些基本函数自己重编写，估计是为了减少对库函数的依赖，从而支持多平台。

最后一个模板函数是 clone：

```
template <class S, class T>
static inline void clone(T *&dst, S *src, int n) {
   dst = new T[n];
   memcpy((void *)dst, (void *)src, sizeof(T) * n);
}
```

属于深拷贝，‘dst‘和 ‘src‘只是内容相同，此外没有任何联系。

10.2 宏定义

```
#define Malloc(type, n) (type *)malloc((n) * sizeof(type))
```

我们知道在 C++ 中申请空间标准是用 new，C 中才是用 malloc 函数族。作者把 malloc 封装起来，我认为是让 C++ 程序更加规范。

10.3 数学函数

libSVM 中幂次函数的设计颇为巧妙：

```
static inline double powi(double base, int times) {
    double tmp = base, ret = 1.0;

    for (int t = times; t > 0; t /= 2) {
        if (t % 2 == 1) ret *= tmp;
        tmp = tmp * tmp;
    }
    return ret;
}
```

我们以求解 a^5 为例：

ret	tmp	t
1	a	5
a	a^2	2
a	a^4	1
a^5	a^4	0

从上往下为循环过程。实际上就是分治算法：

$$a^{2k+1} = a \cdot (a^2)^k$$
$$a^{2k} = (a^2)^k$$

直接相乘的复杂度为 $O(n)$，n 为幂次，该算法则是 $O(\log n)$。

10.4 虚函数

虚函数是 C++ 面向对象中多态的一个著名特性。C++ 基类定义的虚函数，在其派生类中如果进行了改写，但基类指针指向派生类对象，或基类引用绑定了派生类对象时，调用的虚函数是派生类中改写的函数：

```
class Base {
  public:
    virtual void virfunc() {}
};

class Derived : public Base {
  public:
   virtual void virfunc() {
       printf("Hello world");
    }
};
```

```
Base* a = new Derived();
a->virfunc(); // 输出Hello world
```

而如果无法对基类中的虚函数进行定义（抽象的事物往往难以定义），我们将其进行抽象成“纯虚函数”：

```
class Base {
  public:
    virtual void virfunc() = 0;
};
```

这有点类似于 Java 中的接口（Interface），无法独自调用该函数，除非被派生类改写。这一部分还有可以深挖的地方，比如虚函数表和虚函数指针是 C++ 面试中的常见问题。

11 libSVM 的数据存取

SVM 问题中，空间开销最大的两种数据，一个是训练数据，另一个则是核函数矩阵。由于两者都属于 Tabular（表格）数据，因此我们没有必要将两者的实现分离。libSVM 中用一个虚基类 QMatrix 表示数据矩阵，而其派生类 Kernel 则用来存储核函数；此外，libSVM 采用了 Joachims 所说的缓存机制，加快了数据的存取，我们在这一部分对以上内容进行讲解。

11.1 QMatrix 类

```
class QMatrix {
  public:
   virtual Qfloat *get_Q(int column, int len) const = 0;
   virtual double *get_QD() const = 0;
   virtual void swap_index(int i, int j) const = 0;
   virtual ~QMatrix() {}
};
```

里面的函数都是虚函数，因此 QMatrix 是一个虚基类，无法单独使用。我们来看看成员函数的作用：

- get_Q：顺序获取 SVM 问题的 Q 指定**列**的指定个元素；
- get_QD：看到后面的代码有这样一句：

  ```
  double quad_coef = QD[i] + QD[j] + 2 * Q_i[j];
  ```

  对应求 $K_{ii} - 2K_{ij} + K_{jj}$，就明白”QD” 的意思是 QMatrix Diagonal，为了减少存取的时间消耗，特地用一个浮点数组存储 Q 的对角线元素；
- swap_index：交换指定的数据列。

11.2 Kernel 类

Kernel 类是 QMatrix 类的继承，用来存储和处理核函数矩阵：

```
class Kernel : public QMatrix {
  public:
   Kernel(int l, svm_node *const *x, const svm_parameter &param);
   virtual ~Kernel();

   static double k_function(const svm_node *x, const svm_node *y,
                            const svm_parameter &param);
   virtual Qfloat *get_Q(int column, int len) const = 0;
   virtual double *get_QD() const = 0;
   virtual void swap_index(int i, int j) const // no so const...
   {
       swap(x[i], x[j]);
       if (x_square) swap(x_square[i], x_square[j]);
   }

  protected:
   double (Kernel::*kernel_function)(int i, int j) const;

  private:
   const svm_node **x;
   double *x_square;

   // svm_parameter
   const int kernel_type;
   const int degree;
   const double gamma;
   const double coef0;

   static double dot(const svm_node *px, const svm_node *py);
   double kernel_linear(int i, int j) const { return dot(x[i], x[j]); }
   double kernel_poly(int i, int j) const {
       return powi(gamma * dot(x[i], x[j]) + coef0, degree);
   }
   double kernel_rbf(int i, int j) const {
       return exp(-gamma * (x_square[i] + x_square[j] - 2 * dot(x[i], x[j])));
   }
   double kernel_sigmoid(int i, int j) const {
       return tanh(gamma * dot(x[i], x[j]) + coef0);
   }
   double kernel_precomputed(int i, int j) const {
       return x[i][(int)(x[j][0].value)].value;
   }
};
```

先从成员变量开始：

- x：指向样本数据的指针；
- x_square：只在使用 RBF 核时用到；
- kernel_type：在 svm.h 中提过，略；
- degree：同上；
- gamma：同上；
- coef0：同上.
- kernel_function：函数指针，指向核函数.

然后是类函数：

- 构造函数：

```
Kernel::Kernel(int l, svm_node *const *x_, const svm_parameter &param)
: kernel_type(param.kernel_type),
  degree(param.degree),
  gamma(param.gamma),
  coef0(param.coef0) {
  switch (kernel_type) {
    case LINEAR:
      kernel_function = &Kernel::kernel_linear;
      break;
    case POLY:
      kernel_function = &Kernel::kernel_poly;
      break;
    case RBF:
      kernel_function = &Kernel::kernel_rbf;
      break;
    case SIGMOID:
      kernel_function = &Kernel::kernel_sigmoid;
      break;
    case PRECOMPUTED:
      kernel_function = &Kernel::kernel_precomputed;
      break;
  }

  clone(x, x_, l);

  if (kernel_type == RBF) {
    x_square = new double[l];
    for (int i = 0; i < l; i++)
      x_square[i] = dot(x[i], x[i]);
  } else
```

```
        x_square = 0;
    }
```

构造函数做了下面几件事情：

(1) 对成员变量中的 SVM 参数赋值；

(2) 确定核函数；

(3) 复制训练集；

(4) 如果选择 RBF 核，则令 x_square[i] 为 $\boldsymbol{x}_i^\top \boldsymbol{x}_j$。

- 析构函数 Kernel：释放空间
- dot：实现两个向量的点积；注意到 svm_node 的设计在节约空间的同时优化了计算；
- k_function：用来计算 Q，给定两个向量 $\boldsymbol{x}$ 和 $\boldsymbol{y}$ 和核函数 K，计算 $K(\boldsymbol{x},\boldsymbol{y})$；
- kernel_linear，kernel_poly，kernel_rbf，kernel_sigmoid：也是计算对应的核函数，“用于预测步骤”，和 k_function 不同的是参数：k_function 接受的向量格式是 svm_node*，kernel_* 这些函数接受的是给定数据集的序列数：int i 和 int j。

11.3 SVC_Q 等类

Kernel 类只是抽象的核函数存储类，实际上不同的问题有不同的核函数矩阵，比如分类问题下

$$Q_{ij} \equiv y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

而在 One-class SVM 中则是

$$Q_{ij} \equiv K(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

因此，根据不同问题，libSVM 以 Kernel 为基类又派生出三个类：SVC_Q, ONE_CLASS_Q 和 SVR_Q，分别对应分类、区间估计和回归问题。

11.3.1 SVC_Q 类

```
class SVC_Q : public Kernel {
  public:
   SVC_Q(const svm_problem &prob, const svm_parameter &param, const schar *y_)
      : Kernel(prob.l, prob.x, param) {
      clone(y, y_, prob.l);
      cache = new Cache(prob.l, (long int)(param.cache_size * (1 << 20)));
      QD = new double[prob.l];
      for (int i = 0; i < prob.l; i++)
         QD[i] = (this->*kernel_function)(i, i);
   }
```

```
    Qfloat *get_Q(int i, int len) const {
        Qfloat *data;
        int start, j;
        if ((start = cache->get_data(i, &data, len)) < len) {
            for (j = start; j < len; j++)
                data[j] =
                    (Qfloat)(y[i] * y[j] * (this->*kernel_function)(i, j));
        }
        return data;
    }

    double *get_QD() const { return QD; }

    void swap_index(int i, int j) const {
        cache->swap_index(i, j);
        Kernel::swap_index(i, j);
        swap(y[i], y[j]);
        swap(QD[i], QD[j]);
    }

    ~SVC_Q() {
        delete[] y;
        delete cache;
        delete[] QD;
    }

  private:
    schar *y;
    Cache *cache;
    double *QD;
};
```

由于每个函数实现语句不多，因此把实现放进类中。我们先看成员变量：

- `y`：是一个 `signed char` 指针，因为是分类问题，$y_i \in \{-1, +1\}$，因此选择最小的带符号类型 `signed char`；
- `cache`：缓存机制的实现，后面会提到；
- `QD`：前面也解释过，Q 矩阵的对角线元素；

成员函数大部分是对基类虚函数的改写：

- 构造函数：主要任务有
  1. 参数赋值；
  2. 数据集（包括特征向量和标签）的复制；

3. 初始化缓存和 QD 数组；

- 析构函数：释放占用的空间；
- get_Q：获取 Q 矩阵指定行（考虑到对称性，列也可以）的多个元素，注意到分类问题中

$$Q_{ij} = y_i y_j K(x_i, x_j) \tag{96}$$

这里的 for 循环其实是由于 Cache 的未命中而做出的弥补；

- get_QD：获取对角线元素数组；
- swap_index：将第 i 个数据和第 j 个数据进行调换，对很多变量都要做出调整；

11.3.2 ONE_CLASS_Q 类

和 SVC_Q 类大同小异，由于没有标签，在 swap_index 也就不需要对 y 进行调换，此外这里的 Q_{ij}：

$$Q_{ij} = K(x_i, x_j) \tag{97}$$

11.3.3 SVR_Q 类

```
class SVR_Q : public Kernel {
  public:
   SVR_Q(const svm_problem &prob, const svm_parameter &param)
       : Kernel(prob.l, prob.x, param) {
       l = prob.l;
       cache = new Cache(l, (long int)(param.cache_size * (1 << 20)));
       QD = new double[2 * l];
       sign = new schar[2 * l];
       index = new int[2 * l];
       for (int k = 0; k < l; k++) {
           sign[k] = 1;
           sign[k + l] = -1;
           index[k] = k;
           index[k + l] = k;
           QD[k] = (this->*kernel_function)(k, k);
           QD[k + l] = QD[k];
       }
       buffer[0] = new Qfloat[2 * l];
       buffer[1] = new Qfloat[2 * l];
       next_buffer = 0;
   }

   void swap_index(int i, int j) const {
       swap(sign[i], sign[j]);
       swap(index[i], index[j]);
```

```
        swap(QD[i], QD[j]);
    }

    Qfloat *get_Q(int i, int len) const {
        Qfloat *data;
        int j, real_i = index[i];
        if (cache->get_data(real_i, &data, l) < l) {
            for (j = 0; j < l; j++)
                data[j] = (Qfloat)(this->*kernel_function)(real_i, j);
        }

        // reorder and copy
        Qfloat *buf = buffer[next_buffer];
        next_buffer = 1 - next_buffer;
        schar si = sign[i];
        for (j = 0; j < len; j++)
            buf[j] = (Qfloat)si * (Qfloat)sign[j] * data[index[j]];
        return buf;
    }

    double *get_QD() const { return QD; }

    ~SVR_Q() {
        delete cache;
        delete[] sign;
        delete[] index;
        delete[] buffer[0];
        delete[] buffer[1];
        delete[] QD;
    }

  private:
    int l;
    Cache *cache;
    schar *sign;
    int *index;
    mutable int next_buffer;
    Qfloat *buffer[2];
    double *QD;
};
```

和分类问题不同，SVR 问题中每个样本向量对应的是两个参数 α_i 和 α^*，从而我们可以在其构造函数中看到它申请了不少长度为两倍样本数的数组。同时它申请了缓冲区，我暂时不明白这些数组的用处，只能从注释中知道它们是为了所谓“重排和复制”。注意到回归问题中的 Q_{ij} 和 ONE_CLASS_SVM 中相同。

至此我们大致分析完 libSVM 中的数据存储相关部分，接下来就是算法代码了。

12 缓存机制

为了提高 SVM 的训练速度，从算法上我们可以收缩矩阵维数。我们也可以通过计算机系统中的缓存机制（Caching），从数据存取的角度来减少训练时间。从现在最流行的 SVM 工具包 libSVM 的 C++ 源码中，我们可以见识到这种机制的巧妙设计。

12.1 核函数缓存

libSVM 中所采用的缓存机制是为了对核函数矩阵，也是消耗空间最大的数据结构进行存储：

$$K_{l\times l}=\begin{bmatrix}\phi(\boldsymbol{x}_1)^2 & \phi(\boldsymbol{x}_1)\phi(\boldsymbol{x}_2) & \cdots & \phi(\boldsymbol{x}_1)\phi(\boldsymbol{x}_l)\\ \phi(\boldsymbol{x}_2)\phi(\boldsymbol{x}_1) & \ddots & & \vdots\\ \vdots & & \ddots & \\ \phi(\boldsymbol{x}_l)\phi(\boldsymbol{x}_1) & \cdots & & \phi(\boldsymbol{x}_l)^2\end{bmatrix}$$

由于训练过程需要频繁使用核函数，所以相比于用一次计算一次，我们当然会选择对有限的核函数的值存储起来供以后使用，这就会带来存储空间不够用的问题，考虑 K 是对称矩阵，那我们也需要存储 $\frac{l(l-1)}{2}$ 个数据，当数据量到达 10000 时，我们需要约 200MB 去存储它们（一个浮点数最少 4 字节）我们希望能从软硬件上加快速度，因此引入缓存机制：用一段不大的内存空间去存储使用最频繁的核函数，借助程序的局部性原理让数据存取速度提升。

12.2 代码概览

libSVM 用一个缓存类（Cache）实现内存管理，包括申请、释放等。类定义如下（svm.cpp）：

```
//
// Kernel Cache
//
// l is the number of total data items
// size is the cache size limit in bytes
//
class Cache {
  public:
   Cache(int l, long int size);
   ~Cache();

   // request data [0,len)
   // return some position p where [p,len) need to be filled
   // (p >= len if nothing needs to be filled)
   int get_data(const int index, Qfloat **data, int len);
   void swap_index(int i, int j);

  private:
   int l;
   long int size;
```

```
    struct head_t {
        head_t *prev, *next; // a circular list
        Qfloat *data;
        int len; // data[0,len) is cached in this entry
    };

    head_t *head;
    head_t lru_head;
    void lru_delete(head_t *h);
    void lru_insert(head_t *h);
};
```

注：*QFloat* 等价于 *float*：

观察其中的数据结构和函数变量命名，可以发现 libSVM 使用一个遵循 LRU（最近最少用）缓存机制的循环双向链表来管理数据。

12.3 成员变量及函数

在 Cache 类中，l 和 size 不难理解，分别表示样本数和用户指定分配的内存。值得注意的是，在 svm.cpp 中有这样的语句：

```
class SVC_Q {
    SVC_Q(const svm_problem &prob, const svm_parameter &param, const schar *y_)
        : Kernel(prob.l, prob.x, param) {
        ...
        cache = new Cache(prob.l, (long int)(param.cache_size * (1 << 20)));
        ...
    }
    ...
};
```

也就是说用户输入的存储空间参数是以 MB 为单位的。

此处的 head_t 结构除了双向链表必备的头尾指针外，还有一个 float 类型的指针和长度参数 len，说明一个 head_t 节点指向一列长度为 len 的数组数据。

在构造函数中，我们可以看到 head 和 lru_head 的区别和联系：

```
Cache::Cache(int l_, long int size_) : l(l_), size(size_) {
    head = (head_t *)calloc(l, sizeof(head_t)); // initialized to 0
    size /= sizeof(Qfloat);
    size -= l * sizeof(head_t) / sizeof(Qfloat);
    size = max(size, 2 * (long int)l); // cache must be large enough for two columns
    lru_head.next = lru_head.prev = &lru_head;
}
```

可以发现 head_t 指向的是长度为 l 的数组，也就是存储全部数据的地方，形成一个 l 行 len 列的表格。此时的 size 单位是字节，因为一个 QFloat 是四字节，上面代码的第 3、4 行分别将其

改为可存储的数据个数（第 3 行），减去 head_t 所占空间（第四行）；最后要保证数据量至少要容纳 $2l$ 条数据（第 5 行），因为支持向量回归问题中，每个样本 x_i 对应两个拉格朗日乘子 α_i 和 α_i^*。从第六行可以看出 lru_head 是用来实现 LRU 算法的双向循环链表，由于初始无数据，所以自己指向自己。

析构函数就是简单的链表和指针的空间释放：

```
Cache::~Cache() {
    for (head_t *h = lru_head.next; h != &lru_head; h = h->next) {
        free(h->data);
    }
    free(head);
}
```

然后是实现双向链表基本功能：删除和插入节点的函数：

```
void Cache::lru_delete(head_t *h) {
    // delete from current location
    h->prev->next = h->next;
    h->next->prev = h->prev;
}

void Cache::lru_insert(head_t *h) {
    // insert to last position
    h->next = &lru_head;
    h->prev = lru_head.prev;
    h->prev->next = h;
    h->next->prev = h;
}
```

要注意的是 LRU 算法模拟的是 FIFO 队列，删除的是队首元素，所以新插入的元素应该从尾部插入。

接下来是共有接口 ‘get_data‘，用于从数据集里取数据：

```
int Cache::get_data(const int index, Qfloat **data, int len) {
    head_t *h = &head[index];
    if (h->len) {
        lru_delete(h);
    }
    int more = len - h->len;

    if (more > 0) {
        // free old space
        while (size < more) {
            head_t *old = lru_head.next;
            lru_delete(old);
            free(old->data);
            size += old->len;
```

```
            old->data = 0;
            old->len = 0;
        }

        // allocate new space
        h->data = (Qfloat *)realloc(h->data, sizeof(Qfloat) * len);
        size -= more;
        swap(h->len, len);
    }

    lru_insert(h);
    *data = h->data;
    return len;
}
```

函数参数不难理解：从第 l 个样本取 len 个数据，并将其赋到 data 上。如果对应的 head[index] 被分配了内存，我们将其从链表中断开；如果内存不够用，就将其内存释放，等到内存够用了再重新分配。此时的缓存属于“未命中”，因此我们得从“内存”中将所需数据移入缓存中；由于我们刚对数据进行存取，因此我们将其从队尾插入，符合 LRU 规则；最后是将数据付给 data，供用户使用。

最后一个函数接口，swap_index(int i, int j)，用于将 head[i] 和 head[j] 的内容对换，个人理解用处是为了在存取旧数据后改变旧数据的 LRU 优先级。

```
void Cache::swap_index(int i, int j) {
    if (i == j) return;

    if (head[i].len)
        lru_delete(&head[i]);
    if (head[j].len)
        lru_delete(&head[j]);
    swap(head[i].data, head[j].data);
    swap(head[i].len, head[j].len);
    if (head[i].len)
        lru_insert(&head[i]);
    if (head[j].len)
        lru_insert(&head[j]);

    if (i > j) swap(i, j);
    for (head_t *h = lru_head.next; h != &lru_head; h = h->next) {
        if (h->len > i) {
            if (h->len > j)
                swap(h->data[i], h->data[j]);
            else {
                // give up
                lru_delete(h);
                free(h->data);
```

```
                size += h->len;
                h->data = 0;
                h->len = 0;
            }
        }
    }
}
```

swap_index 将待交换的两个节点从链表中分离，然后交换并回到链表。但笔者并不理解后面遍历链表的操作目的，尤其是将数组的 index 和数据的长度 len 进行比较。

据说 Cache 的存在使得 libSVM 的训练速度提升了 20 倍，说明编程技术对算法实现的重要性。我们这里只提到了数据在 Cache 的存取，但并没有涉及核函数缓存，是因为笔者也在探索中。只看到有人提到调用 get_data 后，程序会计算

$$Q_{ij} = y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

并将其填入 QFloat **data 中。

13 Solver 类

我们这里会研究 libSVM 中的 Solver 类，它用于求解一个带线性约束的二次规划问题。

```
class Solver {
  public:
   Solver(){};
   virtual ~Solver(){};

   struct SolutionInfo {
      double obj;
      double rho;
      double upper_bound_p;
      double upper_bound_n;
      double r; // for Solver_NU
   };

   void Solve(int l, const QMatrix &Q, const double *p_, const schar *y_,
              double *alpha_, double Cp, double Cn, double eps,
              SolutionInfo *si, int shrinking);

  protected:
   int active_size;
   schar *y;
   double *G; // gradient of objective function
   enum { LOWER_BOUND, UPPER_BOUND, FREE };
   char *alpha_status; // LOWER_BOUND, UPPER_BOUND, FREE
   double *alpha;
   const QMatrix *Q;
```

```
    const double *QD;
    double eps;
    double Cp, Cn;
    double *p;
    int *active_set;
    double *G_bar; // gradient, if we treat free variables as 0
    int l;
    bool unshrink; // XXX

    double get_C(int i) { return (y[i] > 0) ? Cp : Cn; }
    void update_alpha_status(int i) {
        if (alpha[i] >= get_C(i))
            alpha_status[i] = UPPER_BOUND;
        else if (alpha[i] <= 0)
            alpha_status[i] = LOWER_BOUND;
        else
            alpha_status[i] = FREE;
    }
    bool is_upper_bound(int i) { return alpha_status[i] == UPPER_BOUND; }
    bool is_lower_bound(int i) { return alpha_status[i] == LOWER_BOUND; }
    bool is_free(int i) { return alpha_status[i] == FREE; }
    void swap_index(int i, int j);
    void reconstruct_gradient();
    virtual int select_working_set(int &i, int &j);
    virtual double calculate_rho();
    virtual void do_shrinking();

  private:
    bool be_shrunk(int i, double Gmax1, double Gmax2);
};
```

我们先来看成员变量：

- `SolutionInfo`：一个存储优化问题的解对应参数的结构体：

```
struct SolutionInfo {
   double obj; // 目标函数最优值
   double rho; // 对应
   double upper_bound_p;
   double upper_bound_n;
   double r; // for Solver_NU
   };
```

- `active_size`：实际参加矩阵运算的样本数，我们在这里详细介绍了关于 `active_size` 变量的含义；
- `y`：类别向量，$y \in \{-1, +1\}$；

- `G`：优化函数的梯度向量；
- 枚举变量 `enum { LOWER_BOUND, UPPER_BOUND, FREE }`：分别表示对应的 $\alpha_i = 0$，$\alpha_i = C$ 和 $\alpha_i \in (0, C)$；
- `alpha_status`：上面对应枚举值的数组；
- `alpha`：也就是 α 向量；
- `Q`：也就是前面提到的 Q 矩阵，核函数和 Solver 相互结合，可以产生多种 SVC,SVR；
- `QD`：`Q` 的对角元素构成的数组；
- `eps`：判断迭代是否终止的阈值 ϵ；
- `Cp, Cn`，表示 Positive Class 和 Negative Class，用于多分类问题；
- `p`：对应优化目标函数 $f = \frac{1}{2}\alpha^\top Q\alpha + p^\top\alpha$ 中的向量 p；
- `active_set`：也就是参加矩阵运算的样本数的序列集合，长度为 `active_size`；
- `G_bar`：对应论文中的 $\bar{G}$，详细解析笔者写在这里；
- `l`：样本数；
- `unshrinked`：表示是否还没使用 Shrinking 技巧.

然后是成员函数，构造和析构函数跳过：

- `get_C`：返回对应样本的种类；设置不同的 Cp 和 Cn 是为了处理数据的不平衡，类似于“再缩放”（?）；
- `update_alpha_status`：用于在每轮迭代后对 α_i 的状态更新，状态对应前面的枚举值；
- 三个 `is_*` 函数，用于判断 α_i 状态的封装函数；
- `swap_index`：交换两样本产生的内容；
- `reconstruct_gradient`：梯度的增量式更新，数学推导和代码讲解放在这里；
- `select_working_set`：选择工作集，数学推导和代码讲解放在这里；
- `calculate_rho`：计算参数 ρ，相应的也就是在计算 b；
- `do_shrinking`：采取收缩算法，数学推导放在这里；
- `be_shrunk`：判断参数 α_i 是否收敛到最优；
- `Solve`：核心函数，用于求解问题。

我们接下来对上面一些复杂或没有提到的函数进行解读。

13.1 ρ 的计算

可以证明，$C-$SVC 和 ε-SVR 中的 b 与 One-class SVM 中的 ρ 等价，且满足

$$b+\rho=0 \tag{98}$$

对于一个支持向量 $\boldsymbol{x}_i$，当我们得到最优的 $\boldsymbol{\alpha}$ 后，有

$$b=y_i-\sum_{j=1}^{l}y_i\alpha_jK(\boldsymbol{x}_j,\boldsymbol{x}_i) \tag{99}$$

采用更鲁棒的方法，对所有支持向量对应的 b 取平均值：

$$\begin{aligned} b^*&=\frac{1}{|S|}\sum_{i\in S}(y_i-\sum_{j=1}^{l}y_i\alpha_jK(\boldsymbol{x}_j,\boldsymbol{x}_i))\\ &=\frac{1}{|S|}\sum_{i\in S}\big(-y_i\nabla f(\alpha)_i\big) \end{aligned} \tag{100}$$

其中 S 为支持向量的下标集。由 b 与 ρ 的关系，我们也可得到

$$\rho=\frac{1}{|S|}\sum_{i\in S}y_i\nabla f(\alpha)_i \tag{101}$$

此时应该有

$$-M(\alpha)\leqslant\rho\leqslant-m(\alpha) \tag{102}$$

而不幸的是如果 $S=\emptyset$，那么我们令

$$\begin{aligned} r_1&=\frac{1}{2}\left(\max_{\alpha_i=C,y_i=1}\nabla f(\alpha)_i+\min_{\alpha_i=0,y_i=1}\nabla f(\alpha)_i\right)\\ r_2&=\frac{1}{2}\left(\max_{\alpha_i=C,y_i=-1}\nabla f(\alpha)_i+\min_{\alpha_i=0,y_i=-1}\nabla f(\alpha)_i\right) \end{aligned} \tag{103}$$

从而设

$$\rho=\frac{r_1+r_2}{2},-b=\frac{r_1-r_2}{2} \tag{104}$$

我们来看 `calculate_rho` 的代码：

```
double Solver::calculate_rho() {
    double r;
    int nr_free = 0;
    double ub = INF, lb = -INF, sum_free = 0;
    for (int i = 0; i < active_size; i++) {
        double yG = y[i] * G[i];

        if (is_upper_bound(i)) {
            if (y[i] == -1)
                ub = min(ub, yG);
```

```
            else
                lb = max(lb, yG);
        } else if (is_lower_bound(i)) {
            if (y[i] == +1)
                ub = min(ub, yG);
            else
                lb = max(lb, yG);
        } else {
            ++nr_free;
            sum_free += yG;
        }
    }

    if (nr_free > 0)
        r = sum_free / nr_free;
    else
        r = (ub + lb) / 2;

    return r;
}
```

代码的行为和算法一致：遍历 active_set，同时计算 sum_free 和约束上下界，为两种情况做准备。

13.2 Shrinking 相关

我们先看源码：

```
void Solver::do_shrinking() {
    int i;
    double Gmax1 = -INF; // max { -y_i * grad(f)_i | i in I_up(\alpha) }
    double Gmax2 = -INF; // max { y_i * grad(f)_i | i in I_low(\alpha) }

    // find maximal violating pair first
    for (i = 0; i < active_size; i++) {
        if (y[i] == +1) {
            if (!is_upper_bound(i)) {
                if (-G[i] >= Gmax1) Gmax1 = -G[i];
            }
            if (!is_lower_bound(i)) {
                if (G[i] >= Gmax2) Gmax2 = G[i];
            }
        } else {
            if (!is_upper_bound(i)) {
                if (-G[i] >= Gmax2) Gmax2 = -G[i];
            }
            if (!is_lower_bound(i)) {
```

```
            if (G[i] >= Gmax1) Gmax1 = G[i];
        }
    }
}
...
}
```

注意到这里是找最大违反对，也就是求解 $m(\alpha)$ 和 $M(\alpha)$。

```
...
if (unshrink == false && Gmax1 + Gmax2 <= eps * 10) {
    unshrink = true;
    reconstruct_gradient();
    active_size = l;
    info("*");
}
...
```

这一部分在论文里也有提及：为了防止收缩策略幅度太大，于是利用收敛阈值 ε 对其限制：如果

$$m(\alpha) \leqslant M(\alpha^k) + 10\varepsilon$$

就进行收缩。最后一个 `for` 循环则是将所有已经被收缩的变量放到后面，为访问提供便利，在论文中这一操作被称为 “Index Rearrangement”：

```
{
    ...
    for (i = 0; i < active_size; i++)
        if (be_shrunk(i, Gmax1, Gmax2)) {
            active_size--;
            while (active_size > i) {
                if (!be_shrunk(active_size, Gmax1, Gmax2)) {
                    swap_index(i, active_size);
                    break;
                }
                active_size--;
            }
        }
}
```

13.3 Solve 函数

Solve 函数是 Solver 类甚至是整个 libSVM 的核心函数，负责求解二次规划问题。我们先来看函数声明：

```
void Solver::Solve(int l, const QMatrix &Q, const double *p_, const schar *y_,
```

```
                double *alpha_, double Cp, double Cn, double eps,
                SolutionInfo *si, int shrinking);
```

函数参数都是我们前面提到过的，它们可以刻画出一个二次规划问题。我们将它的实现拆开来讲：

13.3.1 初始化

先是一些数据和参数的拷贝和赋值：

```
this->l = l;
this->Q = &Q;
QD = Q.get_QD();
clone(p, p_, l);
clone(y, y_, l);
clone(alpha, alpha_, l);
this->Cp = Cp;
this->Cn = Cn;
this->eps = eps;
unshrink = false;
```

然后初始化所有的 α_i 都是 active 的，在之后的迭代中随着 shrink 而减少。

```
{
   active_set = new int[l];
   for (int i = 0; i < l; i++)
       active_set[i] = i;
   active_size = l;
}
```

初始化梯度，包括 G 和 G_bar：

```
{
    G = new double[l];
    G_bar = new double[l];
    int i;
    for (i = 0; i < l; i++) {
        G[i] = p[i];
        G_bar[i] = 0;
    }
    for (i = 0; i < l; i++)
        if (!is_lower_bound(i)) {
            const Qfloat *Q_i = Q.get_Q(i, l);
            double alpha_i = alpha[i];
            int j;
            for (j = 0; j < l; j++)
                G[j] += alpha_i * Q_i[j];
            if (is_upper_bound(i))
```

```
                for (j = 0; j < l; j++)
                    G_bar[j] += get_C(i) * Q_i[j];
        }
}
```

由

$$f(\alpha) = \frac{1}{2}\alpha^\top Q\alpha - p^\top \alpha$$
$$\nabla f(\alpha)_i = \sum_{j=1}^{l} Q_{ij}\alpha_i - p_i$$

在第一个循环中，对每个梯度分量先加上 p_i；此外，由于所有变量都是 active 的，G_bar 初始化为 0。第二个循环显然是上式的求和过程。注意到为了减少无意义的循环，当 α_i 为 0 时就不再计算；如果发现存在变成 inactive 的 α_i，就对 G_bar 进行更新；

13.3.2 优化

初始化完成后就可以对问题进行优化，为了防止不收敛，我们得确定一个最大循环次数：

```
int max_iter = max(10000000, l > INT_MAX / 100 ? INT_MAX : 100 * l);
```

但我目前并没有找到文献资料能说明这样的设计的原因是什么，姑且理解为一种启发式方法。

正式优化之前，libSVM 会每隔一定的循环次数就进行 shrink 操作：

```
if (--counter == 0) {
   counter = min(l, 1000);
   if (shrinking)
       do_shrinking();
   info(".");
}
```

然后是工作集选取：

```
int i, j;
if (select_working_set(i, j) != 0) {
   // reconstruct the whole gradient
   reconstruct_gradient();
   // reset active set size and check
    active_size = l;
    info("*");
   if (select_working_set(i, j) != 0)
      break;
   else
      counter = 1; // do shrinking next iteration
}
```

如果选取工作集失败则会进入这个 if 语句，此时会重构梯度，将所有的 α 都设置为 active，然后重新选择，如果再一次选不出来，说明这个问题已经收敛了（?），退出迭代；否则就继续迭代，而且必然会进行一次 Shrink，因为我们将所有拉格朗日乘子都激活了。

接着是对选定的 α_i 和 α_j 进行更新，这里涉及到更新和剪辑操作，要对 $y_i = y_j$ 和 $y_i \neq y_j$ 进行分类讨论：

```
const Qfloat *Q_i = Q.get_Q(i, active_size);
const Qfloat *Q_j = Q.get_Q(j, active_size);

double C_i = get_C(i);
double C_j = get_C(j);

double old_alpha_i = alpha[i];
double old_alpha_j = alpha[j];

if (y[i] != y[j]) {
    double quad_coef = QD[i] + QD[j] + 2 * Q_i[j];
    if (quad_coef <= 0) quad_coef = TAU;
    double delta = (-G[i] - G[j]) / quad_coef;
    double diff = alpha[i] - alpha[j];
    alpha[i] += delta;
    alpha[j] += delta;

    if (diff > 0) {
        if (alpha[j] < 0) {
            alpha[j] = 0;
            alpha[i] = diff;
        }
    } else {
        if (alpha[i] < 0) {
            alpha[i] = 0;
            alpha[j] = -diff;
        }
    }
    if (diff > C_i - C_j) {
        if (alpha[i] > C_i) {
            alpha[i] = C_i;
            alpha[j] = C_i - diff;
        }
    } else {
        if (alpha[j] > C_j) {
            alpha[j] = C_j;
            alpha[i] = C_j + diff;
        }
    }
} else {
    double quad_coef = QD[i] + QD[j] - 2 * Q_i[j];
```

```
    if (quad_coef <= 0) quad_coef = TAU;
    double delta = (G[i] - G[j]) / quad_coef;
    double sum = alpha[i] + alpha[j];
    alpha[i] -= delta;
    alpha[j] += delta;

    if (sum > C_i) {
        if (alpha[i] > C_i) {
            alpha[i] = C_i;
            alpha[j] = sum - C_i;
        }
    } else {
        if (alpha[j] < 0) {
            alpha[j] = 0;
            alpha[i] = sum;
        }
    }
    if (sum > C_j) {
        if (alpha[j] > C_j) {
            alpha[j] = C_j;
            alpha[i] = sum - C_j;
        }
    } else {
        if (alpha[i] < 0) {
            alpha[i] = 0;
            alpha[j] = sum;
        }
    }
}
```

具体的算法细节被放置在这里。

13.3.3 辅助变量的更新

优化完成后需要对辅助变量 G 和 G_bar 进行更新，准备下一次迭代，我们在梯度重构中讲解了这部分代码。

13.3.4 迭代超限的处理

如果到了指定迭代次数但仍未收敛，我们会重构一次梯度以计算此时的优化目标函数值，同时输出警告信息：

```
if (iter >= max_iter) {
  if (active_size < l) {
  // reconstruct the whole gradient to calculate objective value
  reconstruct_gradient();
  active_size = l;
```

```
    info("*");
    }
    fprintf(stderr, "\nWARNING: reaching max number of iterations\n");
}
```

13.3.5 收尾工作

在迭代结束后，便是最优解和参数的处理工作：

```
// calculate rho
si->rho = calculate_rho();

// calculate objective value
{
    double v = 0;
    int i;
    for (i = 0; i < l; i++)
        v += alpha[i] * (G[i] + p[i]);

    si->obj = v / 2;
}

// put back the solution
{
    for (int i = 0; i < l; i++)
        alpha_[active_set[i]] = alpha[i];
}
```

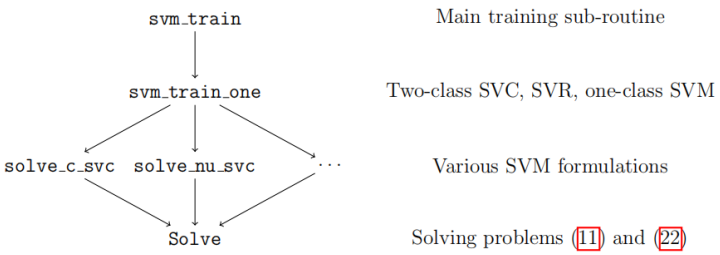
括 ρ，目标函数值的代入，以及参数解的赋值。别忘了申请空间的释放：

```
delete[] p;
delete[] y;
delete[] alpha;
delete[] alpha_status;
delete[] active_set;
delete[] G;
delete[] G_bar;
```

至此，我们对完成了 libSVM 的求解器的分析。

14 非接口函数

我们这里介绍 libSVM 中的非接口函数，还是下面的结构层次图：



我们已经在前面介绍了 Solve 类，现在介绍 Solve 类上层的函数。

14.1 求解具体问题

libSVM 中有五个求解具体 SVM 问题的函数：

- solve_c_svc;
- solve_nu_svc;
- solve_one_class;
- solve_epsilon_svr;
- solve_nu_svr;

分别与前面提到的5 种 SVM对应，都需要借助 Solver 类去实现。这些函数都用 static 关键字修饰，因此无法被模块外直接调用。先看 solve_c_svc：

```
static void solve_c_svc(const svm_problem *prob, const svm_parameter *param,
                        double *alpha, Solver::SolutionInfo *si, double Cp,
                        double Cn) {
    int l = prob->l;
    double *minus_ones = new double[l];
    schar *y = new schar[l];

    int i;

    for (i = 0; i < l; i++) {
        alpha[i] = 0;
        minus_ones[i] = -1;
        if (prob->y[i] > 0)
            y[i] = +1;
        else
            y[i] = -1;
    }

    Solver s;
    s.Solve(l, SVC_Q(*prob, *param, y), minus_ones, y, alpha, Cp, Cn,
            param->eps, si, param->shrinking);
```

```
    double sum_alpha = 0;
    for (i = 0; i < l; i++)
        sum_alpha += alpha[i];

    if (Cp == Cn)
        info("nu = %f\n", sum_alpha / (Cp * prob->l));

    for (i = 0; i < l; i++)
        alpha[i] *= y[i];

    delete[] minus_ones;
    delete[] y;
}
```

在正式求解之前，函数会初始化参数和规范化数据，比如将 α 设置为零向量，将 p 设置为全-1向量（就是 minus_one）；而对于数据中的标签 y，由于外部数据有时会用 0 和 1 区分正负类，因此将其映射到
$-1,+1$。在求解完成后，实际上输出的 α 是 $\left[\alpha_i y_i\right]_{1\times l}$，这样设计是为了方便决策函数的表达：

```
struct decision_function {
    double *alpha;
    double rho;
};
```

这样就不需要多出一个成员 schar *y 来存储 y_i，同时减少计算量。solve_nu_svc、solve_one_class 的流程与 solve_c_svc 大致相同，区别就在于初始条件的设置和多出一些参数。还剩下两个回归问题，我们来看 solve_epsilon_svr：

```
static void solve_epsilon_svr(const svm_problem *prob,
                              const svm_parameter *param, double *alpha,
                              Solver::SolutionInfo *si) {
    int l = prob->l;
    double *alpha2 = new double[2 * l];
    double *linear_term = new double[2 * l];
    schar *y = new schar[2 * l];
    int i;

    for (i = 0; i < l; i++) {
        alpha2[i] = 0;
        linear_term[i] = param->p - prob->y[i];
        y[i] = 1;

        alpha2[i + l] = 0;
        linear_term[i + l] = param->p + prob->y[i];
        y[i + l] = -1;
    }
```

```
    Solver s;
    s.Solve(2 * l, SVR_Q(*prob, *param), linear_term, y, alpha2, param->C,
            param->C, param->eps, si, param->shrinking);

    double sum_alpha = 0;
    for (i = 0; i < l; i++) {
        alpha[i] = alpha2[i] - alpha2[i + l];
        sum_alpha += fabs(alpha[i]);
    }
    info("nu = %f\n", sum_alpha / (param->C * l));

    delete[] alpha2;
    delete[] linear_term;
    delete[] y;
}
```

注意到这里我们需要长度为 $2l$ 的数组存储 α 和 $\alpha^\star$，然后在长度为 l 的向量中填入的是 $\alpha - \alpha^\star$。solve_nu_svr 与之类似，因此不过多赘述。

14.2 单次训练

svm_train_one 负责训练一次支持向量机并输出结果（一个 decision_function），它根据传入参数指定的 SVM 种类进行指定的求解：

```
switch (param->svm_type) {
    case C_SVC:
        solve_c_svc(prob, param, alpha, &si, Cp, Cn);
        break;
    case NU_SVC:
        solve_nu_svc(prob, param, alpha, &si);
        break;
    case ONE_CLASS:
        solve_one_class(prob, param, alpha, &si);
        break;
    case EPSILON_SVR:
        solve_epsilon_svr(prob, param, alpha, &si);
        break;
    case NU_SVR:
        solve_nu_svr(prob, param, alpha, &si);
        break;
}
```

然后计算出支持向量个数和到达边界的支持向量个数：

```
int nSV = 0;
int nBSV = 0;
```

```
for (int i = 0; i < prob->l; i++) {
    if (fabs(alpha[i]) > 0) {
        ++nSV;
        if (prob->y[i] > 0) {
            if (fabs(alpha[i]) >= si.upper_bound_p) ++nBSV;
        } else {
            if (fabs(alpha[i]) >= si.upper_bound_n) ++nBSV;
        }
    }
}
```

最后将计算结果返回：

```
decision_function f;
f.alpha = alpha;
f.rho = si.rho;
return f
```

可以说流程十分简单。

14.3 多分类下的分布估计

libSVM 提供了 sigmoid_train，sigmoid_predoct 和 multiclass_probablity 以及 svm_binary_svc_probability 负责分类问题的分布估计任务。从上面可以了解到我们要利用已知样本对下式极大似然估计出参数 A 和 B：

$$\frac{1}{1+\exp(Af+B)} \tag{105}$$

libSVM 在 sigmoid_train 中采用“带回溯的牛顿法”去求解这个优化问题（留坑）。而 sigmoid_predict 就是在已求出参数的情况下对 r_{ij} 进行估计：

```
static double sigmoid_predict(double decision_value, double A, double B) {
    double fApB = decision_value * A + B;
    // 1-p used later; avoid catastrophic cancellation
    if (fApB >= 0)
        return exp(-fApB) / (1.0 + exp(-fApB));
    else
        return 1.0 / (1 + exp(fApB));
}
```

这里一个有趣的手法就是对 $Af+B$ 的值进行讨论，如果非负，则返回

$$\frac{\exp(-Af-B)}{1+\exp(-Af-B)}$$

否则按原式计算，这样设计的目的应该是防止指数爆炸带来的数据溢出。在将所有的 r_{ij} 求出来后，便是对类别分布 $[p_i]$ 的估计，也就是论文中提到的概率估计迭代算法，这一算法由 mul-

ticlass_probability 函数来计算。而 svm_binary_svc_probability 先做交叉验证，然后用决策值来做概率估计，需要调用 sigmoid_train 函数。

14.4 回归问题的噪声估计

svm_svr_probablity 函数先用五折交叉验证校准误差，然后以拉普拉斯分布为先验分布，对误差进行最大似然估计。

14.5 多分类数据整理

svm_group_classes 函数允许多达 16 类的数据输入，然后对这些数据进行整理和排序：按不同类数据在数据集中第一次出现的顺序作为类别顺序；将同类数据排在一起，并给出其在数据集中的起始点和数量。这样处理显然是为了后续训练的方便。

15 接口函数

我们在这里结束对 libSVM 代码级的讨论（数学上还有很多值得深挖的地方），我们这里讨论 libSVM 的接口函数，也是用户直接调用的方法。

15.1 模型训练

给定数据集进行训练，在 libSVM 中由 svm_train 完成。其逻辑用伪代码书写如下：

```
def svm_train(problem):
  if problem is distribution estimate or regression:
    Initialization
    if problem is regression and need estimating:
      call svm_svr_probability /* 噪声估计 */
    call svm_train_one       /* 单次训练 */
    process the parameters
  else:                        /* 回归问题 */
    call svm_group_classes   /* 整理多类别数据 */
    calculate parameter 'C'
    for i <- 1 to nr_class:
      for j <- i+1 to nr_class:
      /* 用一对一策略处理多分类问题 */
      if need estimating:
        call svm_binary_svc_probability
      call svm_train_one
    build output
  free the sapce we allocated before
  return model we trained
```

svm_train 工作量最大的地方就是多分类问题，包括训练好 $\frac{n(n-1)}{2}$ 个模型后如何处理输出的问题：用矩阵来存储所有分类模型中的 α 向量。

15.2 模型预测

在 svm_predict_values 函数中，我们用训练好的模型对测试数据进行预测：

- 如果任务是回归，我们直接计算拟合函数值 $\sum_j \alpha_j K(x_i, x) - \rho$ 并返回；
- 如果任务是单类 SVM，如果 $\sum_j \alpha_j K(x_j, x) - \rho > 0$，则认为测试数据可以被归为训练数据集中，返回 1，否则返回-1；
- 如果任务是回归，使用投票法选取票数最多的类作为输出。

函数 svm_predict 是对 svm_predict_values 的封装。svm_predict_probablity 也是面向分类问题的预测函数，但基于的是概率估计最大原则。注意到函数中调用了 sigmoid_train，sigmoid_predict 以及 multiclass_probability。如果不小心将回归问题输入，则不做处理，直接返回预测值。

15.3 交叉验证

svm_cross_validation 将数据打乱，分成指定的份数，依次训练子模型，并将结果存储到 target 中。值得注意的是，在分类问题中，如果随机划分就会存在某类数据在某一份数据中不存在的情况。libSVM 采用的是将每类数据平均随机分到每一份数据中的方法解决这一问题。

15.4 模型存取

通过将模型以文件的形式进行存取，我们可以节约重复训练的时间。我们先来看模型保存，libSVM 通过 svm_save_model 函数来保存模型：

```
int svm_save_model(const char *model_file_name, const svm_model *model) {
   ...
    char *old_locale = setlocale(LC_ALL, NULL);
    if (old_locale) {
        old_locale = strdup(old_locale);
    }
    setlocale(LC_ALL, "C");
    ...
}
```

在正式写入模型前，上面的代码先对语言环境（字符集）进行统一，这里是取消用户计算机上的设定，选取默认值 “C”。然后就是写入模型的参数，比如

```
fprintf(fp, "svm_type %s\n", svm_type_table[param.svm_type]);
fprintf(fp, "kernel_type %s\n", kernel_type_table[param.kernel_type]);
```

这些必备参数，还有一些选择性参数，如果对应指针不为空则写入：

```
if (model->nSV) {
   fprintf(fp, "nr_sv");
   for (int i = 0; i < nr_class; i++) fprintf(fp, " %d", model->nSV[i]);
```

```
    fprintf(fp, "\n");
}
```

最重要的是将模型训练好的支持向量和对应的 α_i 写入：

```
fprintf(fp, "SV\n");
const double *const *sv_coef = model->sv_coef;
const svm_node *const *SV = model->SV;

for (int i = 0; i < l; i++) {
    for (int j = 0; j < nr_class - 1; j++) fprintf(fp, "%.17g ", sv_coef[j][i]);

    const svm_node *p = SV[i];

    if (param.kernel_type == PRECOMPUTED)
        fprintf(fp, "0:%d ", (int)(p->value));
    else
        while (p->index != -1) {
            fprintf(fp, "%d:%.8g ", p->index, p->value);
            p++;
        }
    fprintf(fp, "\n");
}
```

最后是还原用户的语言环境：

```
setlocale(LC_ALL, old_locale);
free(old_locale);
```

可以看出，libSVM 将文件模型分为两部分：模型的基本参数，也就是超参数（被称作 model header），以及模型参数。libSVM 会先用 read_model_header 去读取模型超参数，然后再读取全部模型。read_model_header 的基本逻辑就是一行一行匹配，如果标签匹配则读取内容，给新模型赋值。这里用到一个宏：

```
#define FSCANF(_stream, _format, _var)                \
    do {                                              \
        if (fscanf(_stream, _format, _var) != 1) return false; \
    } while (0)
```

是为了“解决 scanf 失败”等问题。在 svm_load_model 中，调用 read_model_header 后就是对模型参数的读取，这里略去。

如果模型格式出现问题，libSVM 也会去检查：svm_check_parameter 和 svm_check_probality_model 都是用来检查模型参数是否正确的函数，保证了存取模型的安全性。

libSVM 还剩下用于获取模型参数的接口和删除模型的函数，比较简单，我们在这里略去。

16 结语

至此，我们已经大致分析完整个 libSVM。随着阅读的深入，便越来越觉得自己能力的卑微，而保持这种卑微或许才是不断求知的动力吧。

笔者其实对 libSVM 并没有完全读懂，对于其中一小部分操作，也无法从说出其作用。后面或许还会继续研究相关内容。