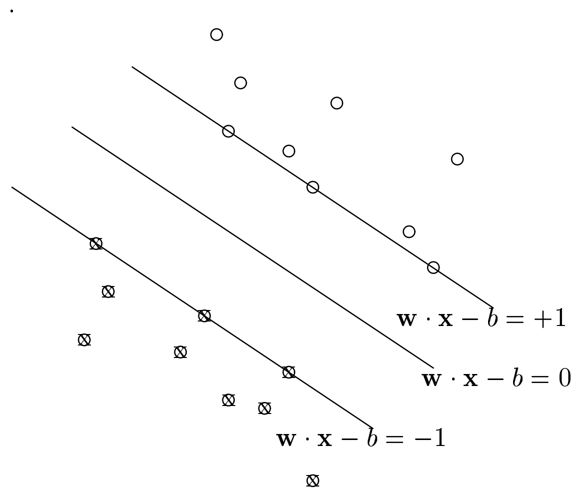
Sequential Minimal Optimization for SVM

Contents



Abstract

This is a C++ implementation of John C. Platt's sequential minimal optimization (SMO) for training a support vector machine (SVM). This program is based on the pseudocode in Platt (1998).

This is both the documentation and the C++ code. It is a `NUWEB` document from which both the LaTeX file and the C++ file can be generated. The documentation is essentially my notes when reading the papers (most of them being *cut-and-paste* from the papers).

1 Introduction to Support Vector Machine (SVM)

This introductio to Support Vector Machine for binary classification is based on Burges (1998).

1.1 Linear SVM

First let us look at the linear support vector machine. It is based on the idea of hyperplane classifier, or linearly separability.

Suppose we have N training data points $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_N, y_N)\}$ where $\mathbf{x}_i \in \mathcal{R}^d$ and $y_i \in \{\pm 1\}$. We would like to learn a linear separating hyperplane classifier:

$$f(\mathbf{x}) = sgn(\mathbf{w} \cdot \mathbf{x} - b).$$

Furthermore, we want this hyperplane to have the maximum separating margin with respect to the two classes. Specifically, we want to find this hyperplane $H: \; y = \mathbf{w} \cdot \mathbf{x} - b = 0$ and two hyperplanes parallel to it and with equal distances to it,

$$H_1: \; y = \mathbf{w} \cdot \mathbf{x} - b = +1,$$

$$H_2: \; y = \mathbf{w} \cdot \mathbf{x} - b = -1,$$

with the condition that there are no data points between H_1 and H_2, and the distance between H_1 and H_2 is maximized.

For any separating plane H and the corresponding H_1 and H_2, we can always "normalize" the coefficients vector $\mathbf{w}$ so that H_1 will be $y = \mathbf{w} \cdot \mathbf{x} - b = +1$, and H_2 will be $y = \mathbf{w} \cdot \mathbf{x} - b = -1$. See Appendix A for details.

We want to maximize the distance between H_1 and H_2. So there will be some positive examples on H_1 and some negative examples on H_2. These examples are called *support vectors* because only they participate in the definition of the separating hyperplane, and other examples can be removed and/or moved around as long as they do not cross the planes H_1 and H_2.

Recall that in 2-D, the distance from a point (x_0, y_0) to a line $Ax+By+C=0$ is $\frac{|Ax_0+By_0+C|}{\sqrt{A^2+B^2}}$. Similarly, the distance of a point on H_1 to $H: \; \mathbf{w} \cdot \mathbf{x} - b = 0$ is $\frac{|\mathbf{w}\cdot\mathbf{x}-b|}{\|\mathbf{w}\|} = \frac{1}{\|\mathbf{w}\|}$, and the distance between H_1 and H_2 is $\frac{2}{\|\mathbf{w}\|}$. So, in order to maximize the distance, we should minimize $\|\mathbf{w}\| = \mathbf{w}^T\mathbf{w}$ with the condition that there are no data points between H_1 and H_2:

$$\mathbf{w} \cdot \mathbf{x} - b \geq +1, \quad \text{for positive examples } y_i = +1,$$

$$\mathbf{w} \cdot \mathbf{x} - b \leq -1, \quad \text{for negative examples } y_i = -1.$$

These two conditions can be combined into

$$y_i(\mathbf{w} \cdot \mathbf{x}_i - b) \geq 1.$$

So our problem can be formulated as

$$\min_{\mathbf{w},b} \frac{1}{2}\mathbf{w}^T\mathbf{w} \quad \text{subject to } y_i(\mathbf{w} \cdot \mathbf{x}_i - b) \geq 1.$$

This is a convex, quadratic programming problem (in $\mathbf{w}, b$), in a convex set.

Introducing Lagrange multipliers $\alpha_1, \alpha_2, \ldots, \alpha_N \geq 0$, we have the following Lagrangian:

$$\mathscr{L}(\mathbf{w}, b, \boldsymbol{\alpha}) \equiv \frac{1}{2}\mathbf{w}^T\mathbf{w} - \sum_{i=1}^{N} \alpha_i y_i(\mathbf{w} \cdot \mathbf{x}_i - b) + \sum_{i=1}^{N} \alpha_i.$$

1.2 The dual problem

We can solve the Wolfe dual instead: *maximize* $\mathscr{L}(\mathbf{w}, b, \boldsymbol{\alpha})$ with respect to $\boldsymbol{\alpha}$, subject to the constraints that the gradient of $\mathscr{L}(\mathbf{w}, b, \boldsymbol{\alpha})$ with respect to the primal variables $\mathbf{w}$ and b vanish:

$$\frac{\partial \mathscr{L}}{\partial \mathbf{w}} = \mathbf{0}, \tag{1}$$

$$\frac{\partial \mathscr{L}}{\partial b} = 0, \tag{2}$$

and that

$$\boldsymbol{\alpha} \geq \mathbf{0}.$$

From Equations 1 and 2, we have

$$\mathbf{w} = \sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i,$$

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

Substitute them into $\mathscr{L}(\mathbf{w}, b, \boldsymbol{\alpha})$, we have

$$\mathscr{L}_D \equiv \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j,$$

in which the primal variables are eliminated.

When we solve α_i, we can get $\mathbf{w} = \sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i$, (we will later show how to compute the threshold b), and we can classify a new object $\mathbf{x}$ with

$$\begin{aligned} f(\mathbf{x}) &= sgn(\mathbf{w} \cdot \mathbf{x} + b) \\ &= sgn((\sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i) \cdot \mathbf{x} + b) \\ &= sgn(\sum_{i=1}^{N} \alpha_i y_i (\mathbf{x}_i \cdot \mathbf{x}) + b). \end{aligned}$$

Please note that in the objective function and the solution, the training vectors $\mathbf{x}_i$ occur only in the form of dot product.

Before going into the details to how to solve this quadratic programming problem, let's extend it in two directions.

1.3 Non-linear SVM

What if the surface separating the two classes are not linear? Well, we can transform the data points to another high dimensional space such that the data points will be linearly separable. Let the transformation be $\Phi(\cdot)$. In the high dimensional space, we solve

$$\mathscr{L}_D \equiv \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)$$

Suppose, in addition, $\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j) = k(\mathbf{x}_i, \mathbf{x}_j)$. That is, the dot product in that high dimensional space is equivalent to a *kernel* function of the input space. So we need not be explicit about the transformation $\Phi(\cdot)$ as long as we know that the kernel function $k(\mathbf{x}_i, \mathbf{x}_j)$ is equivalent to the dot product of some other high dimensional space. There are many kernel functions that can be used this way, for example, the radial basis function (Gaussian kernel)

$$K(\mathbf{x}_i, \mathbf{x}_j) = e^{-\|\mathbf{x}_i - \mathbf{x}_j\|^2/2\sigma^2}.$$

The Mercer's condition can be used to determine if a function can be used as a kernel function:

There exists a mapping Φ and an expansion

$$K(\mathbf{x}, \mathbf{y}) = \sum_i \Phi(\mathbf{x})_i \Phi(\mathbf{y})_i,$$

if and only if, for any $g(\mathbf{x})$ such that

$$\int g(\mathbf{x})^2 d\mathbf{x} \text{ is finite,}$$

then

$$\int K(\mathbf{x}, \mathbf{y}) g(\mathbf{x}) g(\mathbf{y}) d\mathbf{x} d\mathbf{y} \geq 0.$$

1.4 Imperfect separation

The other direction to extend SVM is to allow for noise, or imperfect separation. That is, we do not strictly enforce that there be no data points between H_1 and H_2, but we definitely want to penalize the data points that cross the boundaries. The penalty C will be finite. (If $C = \infty$, we come back to the original perfect separating case.)

We introduce non-negative slack variables $\xi_i \geq 0$, so that

$$\mathbf{w} \cdot \mathbf{x}_i - b \geq +1 - \xi_i \quad \text{for } y_i = +1,$$

$$\mathbf{w} \cdot \mathbf{x}_i - b \leq -1 + \xi_i \quad \text{for } y_i = -1,$$

$$\xi_i \geq 0, \quad \forall i.$$

and we add to the objective function a penalizing term:

$$\underset{\mathbf{w}, b, \boldsymbol{\xi}}{\text{minimize}} \frac{1}{2} \mathbf{w}^T \mathbf{w} + C (\sum_i \xi_i)^m$$

where m is usually set to 1, which gives us

$$\begin{array}{lll} \underset{\mathbf{w}, b, \xi_i}{\text{minimize}} & \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^N \xi_i & \\ \text{subject to} & y_i (\mathbf{w}^T \mathbf{x}_i - b) + \xi_i - 1 \geq 0, & 1 \leq i \leq N \\ & \xi_i \geq 0, & 1 \leq i \leq N \end{array}$$

Introducing Lagrange multipliers $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, the Lagrangian is

$$\begin{aligned}\mathscr{L}(\mathbf{w}, b, \xi_i; \boldsymbol{\alpha}, \boldsymbol{\beta}) &= \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{N}\xi_i \\ &\quad - \sum_{i=1}^{N}\alpha_i\left[y_i(\mathbf{w}^T\mathbf{x}_i - b) + \xi_i - 1\right] - \sum_{i=1}^{N}\mu_i\xi_i \\ &= \frac{1}{2}\mathbf{w}^T\mathbf{w} + \sum_{i=1}^{N}(C - \alpha_i - \mu_i)\,\xi_i \\ &\quad - \left(\sum_{i=1}^{N}\alpha_i y_i \mathbf{x}_i^T\right)\mathbf{w} - \left(\sum_{i=1}^{N}\alpha_i y_i\right)b + \sum_{i=1}^{N}\alpha_i\end{aligned}$$

Neither the ξ_i's, nor their Lagrange multipliers, appear in the Wolfe dual problem:

$$\underset{\boldsymbol{\alpha}}{\text{maximize}}\, \mathscr{L}_D \equiv \sum_i \alpha_i - \frac{1}{2}\sum_{i,j}\alpha_i\alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j$$

subject to:

$$0 \leq \alpha_i \leq C,$$

$$\sum_i \alpha_i y_i = 0.$$

The only difference from the perfectly separating case is that α_i is now bounded above by C instead of ∞. For details, see Appendiex B.

The solution is again given by

$$\mathbf{w} = \sum_{i=1}^{N}\alpha_i y_i \mathbf{x}_i$$

To train the SVM, we search through the feasible region of the dual problem and maximize the objective function. The optimal solution can be checked using the KKT conditions.

1.5 The KKT conditions

The KKT optimality conditions of the primal problem are, the gradient of $\mathscr{L}(\mathbf{w}, b, \boldsymbol{\alpha}, \boldsymbol{\beta})$ with respect to the primal variables $\mathbf{w}$, b, $\boldsymbol{\xi}$ vanishes (this must always be satisfied by the dual problem), and that for $1 \leq i \leq N$,

$$\alpha_i(y_i(\mathbf{w}^T\mathbf{x}_i - b) + \xi_i - 1) = 0, \tag{3}$$

$$\mu_i\xi_i = 0. \tag{4}$$

Depending on the value of α_i, we have three cases to consier:

1. If $\alpha_i = 0$, then $\mu_i = C - \alpha_i = C > 0$. From Equation 4, $\xi_i = 0$. so we have

$$y_i(\mathbf{w}^T \mathbf{x}_i - b) - 1 \geq 0.$$

2. If $0 < \alpha_i < C$, from Equation 3, we have

$$y_i(\mathbf{w}^T \mathbf{x}_i - b) + \xi_i - 1 = 0 \tag{5}$$

Note that $\mu_i = C - \alpha_i > 0$, so $\xi_i = 0$ (Equation 4). Substituting into Equation 5, we have

$$y_i(\mathbf{w}^T \mathbf{x}_i - b) - 1 = 0.$$

3. If $\alpha_i = C$, then from Equation 3, we have

$$y_i(\mathbf{w}^T \mathbf{x}_i - b) + \xi_i - 1 = 0 \tag{6}$$

Note that $\mu_i = C - \alpha_i = 0$, we have $\xi_i \geq 0$. So

$$y_i(\mathbf{w}^T \mathbf{x}_i - b) - 1 \leq 0.$$

The quantity $y_i(\mathbf{w}^T \mathbf{x}_i - b) - 1$ can be computed as

$$R_i = y_i(\mathbf{w}^T \mathbf{x}_i - b) - y_i^2 = y_i(\mathbf{w}^T \mathbf{x}_i - b - y_i) = y_i E_i$$

where $E_i = \mathbf{w}^T \mathbf{x}_i - b - y_i = u_i - y_i$ is the prediction error.

To summarize, the KKT condition implies:

$$\begin{aligned} \alpha_i = 0 \quad &\Rightarrow \quad R_i \geq 0, \\ 0 < \alpha_i < C \quad &\Rightarrow \quad R_i \approx 0, \\ \alpha_i = C \quad &\Rightarrow \quad R_i \leq 0. \end{aligned}$$

In the following two cases, the KKT condition is violated:

- $\alpha_i < C$ and $R_i < 0$,
- $\alpha_i > 0$ and $R_i > 0$.

1.6 Checking KKT condition without using threshold b

As the dual problem does not solve for the threshold b directly, it would be beneficial to check the KKT condition without using threshold b. This technique is due to Keerthi et al. (2001).

The quantity $y_i(\mathbf{w}^T \mathbf{x}_i - b) - 1$ (which must ≥ 0 for all i if the KKT condition is satisfied) can also be written as

$$\begin{aligned} & y_i(\mathbf{w}^T \mathbf{x}_i - b) - 1 \\ = \quad & y_i(\mathbf{w}^T \mathbf{x}_i - b) - y_i^2 \\ - \quad & y_i(\mathbf{w}^T \mathbf{x}_i - y_i - b) \\ = \quad & y_i(F_i - b), \end{aligned}$$

where $F_i \equiv \mathbf{w}^T\mathbf{x}_i - y_i$.

Note for $E_i = F_i - b$, we have $E_i - E_j = F_i - F_j$. (This equality is useful, as Platt's SMO algorithm uses $E_i - E_j$ when optimization the two Lagrange multipliers α_i, α_j.)

This notation is useful because the KKT conditions

$$\begin{aligned} \alpha_i = 0 &\Rightarrow y_i(F_i - b) \geq 0 \\ 0 < \alpha_i < C &\Rightarrow y_i(F_i - b) \approx 0 \\ \alpha_i = C &\Rightarrow y_i(F_i - b) \leq 0 \end{aligned}$$

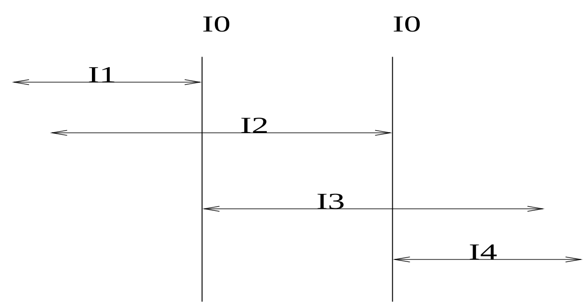
can be written as

$$\begin{aligned} i \in I_0 \cup I_1 \cup I_2 &\Rightarrow F_i \geq b \\ i \in I_0 \cup I_3 \cup I_4 &\Rightarrow F_i \leq b, \end{aligned}$$

where

$$\begin{aligned} I_0 &\equiv \{i : 0 < \alpha_i < C\} \\ I_1 &\equiv \{i : y_i = +1, \alpha_i = 0\} \\ I_2 &\equiv \{i : y_i = -1, \alpha_i = C\} \\ I_3 &\equiv \{i : y_i = +1, \alpha_i = C\} \\ I_4 &\equiv \{i : y_i = -1, \alpha_i = 0\}. \end{aligned}$$

So that $\forall i \in I_0 \cup I_1 \cup I_2$, and $\forall j \in I_0 \cup I_3 \cup I_4$, we should have $F_i \geq F_j$, if KKT condition is satisfied.



To check if this condition holds, we define

$$\begin{aligned} b_{\text{up}} &= \min\{F_i : i \in I_0 \cup I_1 \cup I_2\}, \\ b_{\text{low}} &= \max\{F_i : i \in I_0 \cup I_3 \cup I_4\}. \end{aligned}$$

The KKT condition implies $b_{\text{up}} \geq b_{\text{low}}$, and similarly, $\forall i \in I_0 \cup I_1 \cup I_2$, $F_i \geq b_{\text{low}}$, and $\forall i \in I_0 \cup I_3 \cup I_4$, $F_i \leq b_{\text{up}}$.

These comparisons do not use the threshold b.

As an added benefit, given the first α_i, these comparisons automatically finds the second α_i for joint optimization in SMO.

2 SMO Algorithm

2.1 Optimize two α_i's

The SMO algorithm searches through the feasible region of the dual problem and maxmizes the objective function

$$\mathscr{L}_D \equiv \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j,$$

$$0 \le \alpha_i \le C, \quad \forall i.$$

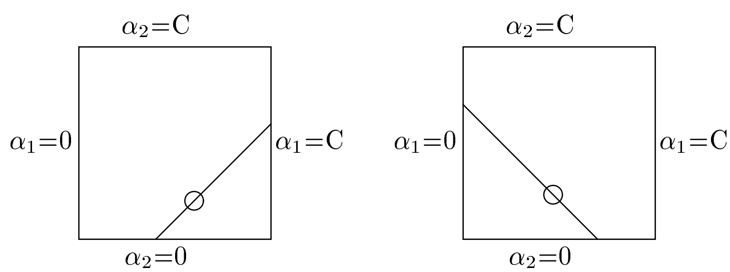
It works by optimzing two α_i's at a time (with the other α_i's fixed). It uses heuristics to choose the two α_i' for optimization. This is essentially a hill-climbing.

Without loss of generality, suppose we are optimizing α_1, α_2, from an old set of feasible solution: α_1^{old}, α_2^{old}, α_3, …, α_N. (For initialization, we can set $\boldsymbol{\alpha}^{\text{old}} = \mathbf{0}$.)

Because $\sum_{i=1}^{N} y_i \alpha_i = 0$, we have

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

This confines the optimization to be on a line, as shown in the following figure:



$y_1 \ne y_2 \Rightarrow \alpha_1 - \alpha_2 = \gamma$ $\qquad$ $y_1 = y_2 \Rightarrow \alpha_1 + \alpha_2 = \gamma$

Let $s = y_1 y_2$. Multiply

$$y_1 \alpha_1 + y_2 \alpha_2 = \text{Const.}$$

by y_1, and we have

$$\alpha_1 = \gamma - s\alpha_2.$$

where $\gamma \equiv \alpha_1 + s\alpha_2 = \alpha_1^{\text{old}} + s\alpha_2^{\text{old}}$.

Fixing the other α_i's, the objective function can be written as

$$\begin{aligned}\mathscr{L}_D \quad = \quad & \alpha_1 + \alpha_2 + \text{Const.} \\ & -\frac{1}{2}\Big(y_1 y_1 \mathbf{x}_1^T \mathbf{x}_1 \alpha_1^2 + y_2 y_2 \mathbf{x}_2^T \mathbf{x}_2 \alpha_2^2 + 2 y_1 y_2 \mathbf{x}_1^T \mathbf{x}_2 \alpha_1 \alpha_2 \\ & \quad +2 \left(\sum_{i=3}^{N} \alpha_i y_i \mathbf{x}_i^T \right) (y_1 \mathbf{x}_1 \alpha_1 + y_2 \mathbf{x}_2 \alpha_2) + \text{Const.} \Big)\end{aligned}$$

Let $K_{11} = \mathbf{x}_1^T \mathbf{x}_1$, $K_{22} = \mathbf{x}_2^T \mathbf{x}_2$, $K_{12} = \mathbf{x}_1^T \mathbf{x}_2$, and

$$\begin{aligned}
v_j &\equiv \sum_{i=3}^{N} \alpha_i y_i \mathbf{x}_i^T \mathbf{x}_j \\
&= \mathbf{x}_j^T \mathbf{w}^{\text{old}} - \alpha_1^{\text{old}} y_1 \mathbf{x}_1^T \mathbf{x}_j - \alpha_2^{\text{old}} y_2 \mathbf{x}_2^T \mathbf{x}_j \\
&= (\mathbf{x}_j^T \mathbf{w}^{\text{old}} - b^{\text{old}}) + b^{\text{old}} - \alpha_1^{\text{old}} y_1 \mathbf{x}_1^T \mathbf{x}_j - \alpha_2^{\text{old}} y_2 \mathbf{x}_2^T \mathbf{x}_j \\
&= u_j^{\text{old}} + b^{\text{old}} - \alpha_1^{\text{old}} y_1 \mathbf{x}_1^T \mathbf{x}_j - \alpha_2^{\text{old}} y_2 \mathbf{x}_2^T \mathbf{x}_j,
\end{aligned}$$

where $u_j^{\text{old}} = \mathbf{x}_j^T \mathbf{w}^{\text{old}} - b^{\text{old}}$ is the output of $\mathbf{x}_j$ under old parameters.

$$\begin{aligned}
\mathscr{L}_D &= \alpha_1 + \alpha_2 - \frac{1}{2}\Big(K_{11}\alpha_1^2 + K_{22}\alpha_2^2 + 2sK_{12}\alpha_1\alpha_2 \\
&\qquad +2y_1 v_1 \alpha_1 + 2y_2 v_2 \alpha_2\Big) + \text{Const.} \\
&= \gamma - s\alpha_2 + \alpha_2 - \frac{1}{2}\Big(K_{11}(\gamma - s\alpha_2)^2 + K_{22}\alpha_2^2 \\
&\qquad +2sK_{12}(\gamma - s\alpha_2)\alpha_2 \\
&\qquad +2y_1 v_1(\gamma - s\alpha_2) + 2y_2 v_2 \alpha_2\Big) + \text{Const.} \\
&= (1-s)\alpha_2 - \frac{1}{2}K_{11}(\gamma - s\alpha_2)^2 - \frac{1}{2}K_{22}\alpha_2^2 - sK_{12}(\gamma - s\alpha_2)\alpha_2 \\
&\qquad -y_1 v_1(\gamma - s\alpha_2) - y_2 v_2 \alpha_2 + \text{Const.} \\
&= (1-s)\alpha_2 - \frac{1}{2}K_{11}\gamma^2 + sK_{11}\gamma\alpha_2 - \frac{1}{2}K_{11}s^2\alpha_2^2 - \frac{1}{2}K_{22}\alpha_2^2 \\
&\qquad -sK_{12}\gamma\alpha_2 + s^2K_{12}\alpha_2^2 - y_1 v_1 \gamma + s y_1 v_1 \alpha_2 - y_2 v_2 \alpha_2 \\
&\qquad +\text{Const.} \\
&= (1-s)\alpha_2 + sK_{11}\gamma\alpha_2 - \frac{1}{2}K_{11}\alpha_2^2 - \frac{1}{2}K_{22}\alpha_2^2 \\
&\qquad -sK_{12}\gamma\alpha_2 + K_{12}\alpha_2^2 + y_2 v_1 \alpha_2 - y_2 v_2 \alpha_2 \\
&\qquad +\text{Const.} \\
&= \left(-\frac{1}{2}K_{11} - \frac{1}{2}K_{22} + K_{12}\right)\alpha_2^2 \\
&\qquad + \left(1 - s + sK_{11}\gamma - sK_{12}\gamma + y_2 v_1 - y_2 v_2\right)\alpha_2 \\
&\qquad +\text{Const.} \\
&= \frac{1}{2}\left(2K_{12} - K_{11} - K_{22}\right)\alpha_2^2 \\
&\qquad + \left(1 - s + sK_{11}\gamma - sK_{12}\gamma + y_2 v_1 - y_2 v_2\right)\alpha_2 \\
&\qquad +\text{Const.}
\end{aligned}$$

Let $\eta \equiv 2K_{12} - K_{11} - K_{12}$. The coefficient of α_2 is

$$\begin{aligned}
& 1 - s + sK_{11}\gamma - sK_{12}\gamma + y_2 v_1 - y_2 v_2 \\
= \;& 1 - s + sK_{11}(\alpha_1^{\text{old}} + s\alpha_2^{\text{old}}) - sK_{12}(\alpha_1^{\text{old}} + s\alpha_2^{\text{old}}) \\
& + y_2(u_1^{\text{old}} + b^{\text{old}} - \alpha_1^{\text{old}} y_1 K_{11} - \alpha_2^{\text{old}} y_2 K_{12})
\end{aligned}$$

$$
\begin{aligned}
& -y_2(u_2^{\mathrm{old}} + b^{\mathrm{old}} - \alpha_1^{\mathrm{old}} y_1 K_{12} - \alpha_2^{\mathrm{old}} y_2 K_{22}) \\
= \; & 1 - s + sK_{11}\alpha_1^{\mathrm{old}} + K_{11}\alpha_2^{\mathrm{old}} - sK_{12}\alpha_1^{\mathrm{old}} - K_{12}\alpha_2^{\mathrm{old}} \\
& + y_2 u_1^{\mathrm{old}} + y_2 b^{\mathrm{old}} - sK_{11}\alpha_1^{\mathrm{old}} - K_{12}\alpha_2^{\mathrm{old}} \\
& - y_2 u_2^{\mathrm{old}} - y_2 b^{\mathrm{old}} + sK_{12}\alpha_1^{\mathrm{old}} + K_{22}\alpha_2^{\mathrm{old}} \\
= \; & 1 - s + (sK_{11} - sK_{12} - sK_{11} + sK_{12})\alpha_1^{\mathrm{old}} \\
& + (K_{11} - 2K_{12} + K_{22})\alpha_2^{\mathrm{old}} \\
& + y_2(u_1^{\mathrm{old}} - u_2^{\mathrm{old}}) \\
= \; & y_2^2 - y_1 y_2 + (K_{11} - 2K_{12} + K_{22})\alpha_2^{\mathrm{old}} + y_2(u_1^{\mathrm{old}} - u_2^{\mathrm{old}}) \\
= \; & y_2(y_2 - y_1 + u_1^{\mathrm{old}} - u_2^{\mathrm{old}}) - \eta\alpha_2^{\mathrm{old}} \\
= \; & y_2((u_1^{\mathrm{old}} - y_1) - (u_2^{\mathrm{old}} - y_2)) - \eta\alpha_2^{\mathrm{old}} \\
= \; & y_2(E_1^{\mathrm{old}} - E_2^{\mathrm{old}}) - \eta\alpha_2^{\mathrm{old}}.
\end{aligned}
$$

So the objective function is

$$\mathscr{L}_D = \frac{1}{2}\eta\alpha_2^2 + (y_2(E_1^{\mathrm{old}} - E_2^{\mathrm{old}}) - \eta\alpha_2^{\mathrm{old}})\alpha_2 + \mathrm{Const.}$$

The first and second derivatives are

$$\frac{\mathrm{d}\mathscr{L}_D}{\mathrm{d}\alpha_2} = \eta\alpha_2 + (y_2(E_1^{\mathrm{old}} - E_2^{\mathrm{old}}) - \eta\alpha_2^{\mathrm{old}}),$$

$$\frac{\mathrm{d}^2\mathscr{L}_D}{\mathrm{d}\alpha_2^2} = \eta.$$

Note that $\eta = 2K_{12} - K_{11} - K_{22} \leq 0$. Proof: Let $K_{11} = \mathbf{x}_1^T\mathbf{x}_1$, $K_{12} = \mathbf{x}_1^T\mathbf{x}_2$, $K_{22} = \mathbf{x}_2^T\mathbf{x}_2$. Then $\eta = -(\mathbf{x}_2 - \mathbf{x}_1)^T(\mathbf{x}_2 - \mathbf{x}_1) = -\|\mathbf{x}_2 - \mathbf{x}_1\|^2 \leq 0$.

Let $\frac{\mathrm{d}\mathscr{L}_D}{\mathrm{d}\alpha_2} = 0$, and we have

$$
\begin{aligned}
\alpha_2^{\mathrm{new}} &= -\frac{y_2(E_1^{\mathrm{old}} - E_2^{\mathrm{old}}) - \eta\alpha_2^{\mathrm{old}}}{\eta} \\
&= \alpha_2^{\mathrm{old}} + \frac{y_2(E_2^{\mathrm{old}} - E_1^{\mathrm{old}})}{\eta}
\end{aligned}
$$

If $\eta < 0$, the above equation gives us the unconstrained maximum point α_2^{new}. It must be checked against the feasible range. Let $s = y_1 y_2$, and $\gamma = \alpha_1^{\mathrm{old}} + s\alpha_2^{\mathrm{old}}$. The range of α_2 is determined as follows:

- If $s = 1$, then $\alpha_1 + \alpha_2 = \gamma$.
  - If $\gamma > C$, then $\max \alpha_2 = C$, and $\min \alpha_2 = \gamma - C$.
  - If $\gamma < C$, then $\min \alpha_2 = 0$, and $\max \alpha_2 = \gamma$.
- If $s = -1$, then $\alpha_1 - \alpha_2 = \gamma$.
  - If $\gamma > 0$, then $\min \alpha_2 = 0$, and $\min \alpha_2 = C - \gamma$.

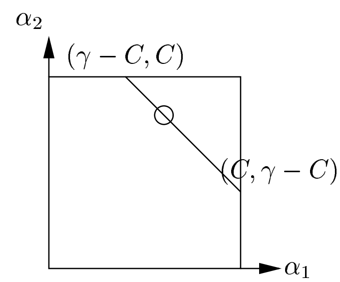


Figure 1: $\alpha_1 + \alpha_2 = \gamma$, and $\gamma > C$.

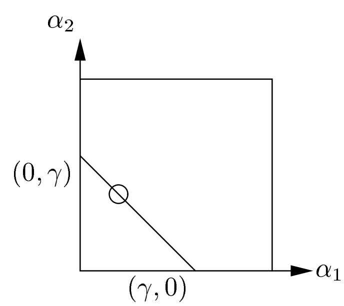


Figure 2: $\alpha_1 + \alpha_2 = \gamma$, and $\gamma < C$.

- If $\gamma < 0$, then $\min \alpha_2 = -\gamma$, and $\max \alpha_2 = C$.

Let the minimum feasible value of α_2 be L, maximum be H. Then

$$\alpha_2^{\text{new,clipped}} = \begin{cases} H, & \text{if } H < \alpha_2^{\text{new}}, \\ \alpha_2^{\text{new}}, & \text{if } L \le \alpha_2^{\text{new}} \le H \\ L, & \text{if } \alpha_2^{\text{new}} < L. \end{cases}$$

To summarize, given α_1, α_2 (and the corresponding y_1, y_2, K_{11}, K_{12}, K_{22}, $E_2^{\text{old}} - E_1^{\text{old}}$), we can optimize the two α's by the following procedure:

1. $\eta = 2K_{12} - K_{11} - K_{22}$.
2. If $\eta < 0$,

   $$\Delta\alpha_2 = \frac{y_2(E_2^{\text{old}} - E_1^{\text{old}})}{\eta},$$

   and clip the solution within the feasible region. Then

   $$\Delta\alpha_1 = -s\Delta\alpha_2.$$

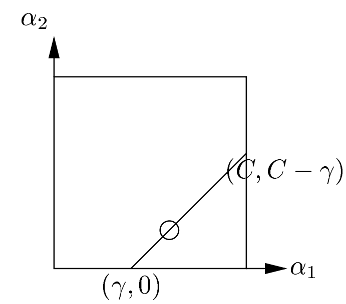


Figure 3: $\alpha_1 - \alpha_2 = \gamma$, and $\gamma > 0$.

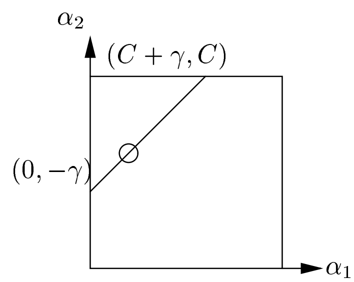


Figure 4: $\alpha_1 - \alpha_2 = \gamma$, and $\gamma < 0$.

3. If $\eta = 0$, we need to evaluate the objective function at the two endpoints, and set α_2^{new} to be the one with larger objective function value. The objective function is

$$\mathscr{L}_D = \frac{1}{2}\eta\alpha_2^2 + (y_2(E_1^{\text{old}} - E_2^{\text{old}}) - \eta\alpha_2^{\text{old}})\alpha_2 + \text{Const.} \tag{7}$$

2.2 SMO Algorithm: Updating after a successful optimization step

When α_1, α_2 are changed by $\Delta\alpha_1$, $\Delta\alpha_2$, we can update E_i's, F_i's, $\mathbf{w}$ (for linear kernel), and b. Let $E(\mathbf{x}, y)$ be the prediction error on $(\mathbf{x}, y)$:

$$E(\mathbf{x}, y) = \sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i^T \mathbf{x} - b - y,$$

The change in E is

$$\Delta E(\mathbf{x}, y) = \Delta\alpha_1 y_1 \mathbf{x}_1^T \mathbf{x} + \Delta\alpha_2 y_2 \mathbf{x}_2^T \mathbf{x} - \Delta b. \tag{8}$$

The change in the threshold can be computed by forcing $E_1^{\text{new}} = 0$ if $0 < \alpha_1^{\text{new}} < C$ (or $E_2^{\text{new}} = 0$ if $0 < \alpha_2^{\text{new}} < C$). From

$$0 \quad = \quad E(\mathbf{x}, y)^{\text{new}}$$

$$
\begin{aligned}
&= E(\mathbf{x}, y)^{\text{old}} + \Delta E(\mathbf{x}, y) \\
&= E(\mathbf{x}, y)^{\text{old}} + \Delta\alpha_1 y_1 \mathbf{x}_1^T \mathbf{x} + \Delta\alpha_2 y_2 \mathbf{x}_2^T \mathbf{x} - \Delta b
\end{aligned}
$$

we have

$$\Delta b = E(\mathbf{x}, y)^{\text{old}} + \Delta\alpha_1 y_1 \mathbf{x}_1^T \mathbf{x} + \Delta\alpha_2 y_2 \mathbf{x}_2^T \mathbf{x}. \tag{9}$$

If $\alpha_1 = 0$, we can only say $y_1 E_1^{\text{new}} \geq 0$; similarly, if $\alpha_1 = C$, we have $y_1 E_2^{\text{new}} \leq 0$. If both α_1 and α_2 take values 0 or C, the original SMO algorithm computes two values of the new b for α_1 and α_2 using Equation 9, and takes the average. This is regarded as problematic by Keerthi et al. (2001).

Similarly, from

$$F(\mathbf{x}, y) = \sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i^T \mathbf{x} - y$$

we have

$$\Delta F(\mathbf{x}, y) = \Delta\alpha_1 y_1 \mathbf{x}_1^T \mathbf{x} + \Delta\alpha_2 y_2 \mathbf{x}_2^T \mathbf{x}. \tag{10}$$

For the weight vector of linear kernels,

$$
\begin{aligned}
\mathbf{w} &= \sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i, \\
\Delta \mathbf{w} &= \Delta\alpha_1 y_1 \mathbf{x}_1 + \Delta\alpha_2 y_2 \mathbf{x}_2. \qquad (11)
\end{aligned}
$$

2.3 SMO Algorithm: Pick two α_i's for optimization

The heuristics for picking two α_i's for optimization in the original SMO paper are as follows:

- The outer loop selects the first α_i, the inner loop selects the second α_i that maximizes $|E_2 - E_1|$.
- The outer loop alternates between one sweep through all examples and as many sweeps as possible through the non-boundary examples (those with $0 < \alpha_i < C$), selecting the example that violates the KKT condition.
- Given the first α_i, the inner loop looks for a non-boundary that maximizes $|E_2 - E_1|$. If this does not make progress, it starts a sequential scan through the non-boundary examples, starting at a random position; if this fails too, it startis a sequential scan through all the examples, also starting at a random postion.

Because the algorithm spends most of the time adjusting the non-boundary examples, the E_i's of these examples are cached.

The improvement proposed in Keerthi et al. (2001) avoids the use of the threshold b in checking KKT condition, and compares two F_i's, which also automatically selects the second α_i for joint optimization. There are two variantions when the outer loop deals only with the non-boundary examples:

- The first α is selected sequentially from all the non-boundary examples. If the first α_i violates the KKT condition when compared with α_j with $F_j = b_{\text{low}}$, or $F_j = b_{\text{up}}$, then select α_j as the second α.
- The two α's are also those with $F_i = b_{\text{low}}$ or $F_i = b_{\text{up}}$.

After a successful step using a pair of indices, (i_2, i_1), let $\tilde{I} = I_0 \cup \{i_1, i_2\}$. We claim that each of the two sets, $\tilde{I} \cap (I_0 \cup I_1 \cup I_2)$ and $\tilde{I} \cap (I_0 \cup I_3 \cup I_4)$, is non-empty, hence we can compute partial b_{low} and b_{up} from the two sets.

> Proof: The two sets are non-empty if $I_0 \neq \emptyset$. If $I_0 = \emptyset$, then α_1, α_2 can only take values from 0 or C. They cannot take the same value and $y_1 = y_2$ at the same time, otherwise we have $0 + 0 = \gamma$ or $C + C = \gamma$ for $\alpha_1 + s\alpha_2 = \gamma$, in which α_1 and α_2 cannot be changed, which contradicts the fact that we just had a successful step. So if they take the same value, with different y_i's, they will belong to two different sets. If they take different values, with the same y_i's, they will also belong to two different sets.

3 C++ Implementation

Now let's write the C++ code, based on the pseudocode in Platt (1998).

"c/smo.cc" 15a ≡

```
⟨Header files to include 17, ... ⟩
using namespace std;
⟨Global variables 16, ... ⟩
⟨Functions 18a, ... ⟩
⟨Main routine 15b⟩
◇
```

3.1 The main routine

The `main` routine implements the outer loop that selects the first α_i for optimization. It alternates between a sweep through all the examples (`examineAll==1`) and as many sweeps as possible through the non-boundary examples (`examineAll==0`). If an example `k` violates the KKT condition more than `eps`$=\epsilon$, it is selected as the first α_i, and `examineExample(k)` is called, which returns 1 if positive progress is made to improve the objective function (with two changed α_i's).

⟨Main routine 15b⟩ ≡

```
int main(int argc, char *argv[]) {
  ⟨Variables local to main 31a⟩
  int numChanged;
  int examineAll;

  ⟨Get in parameters 29d⟩
  ⟨Read in data 31c⟩
```

```
    if (!is_test_only) {
        alph.resize(end_support_i, 0.);

        /* initialize threshold to zero */
        b = 0.;

        /* E_i = u_i - y_i = 0 - y_i = -y_i */
        error_cache.resize(N);

        if (is_linear_kernel)
           w.resize(d,0.);
    }

    ⟨Initialization 26a, ... ⟩

    if (!is_test_only) {
        numChanged = 0;
        examineAll = 1;
        while (numChanged > 0 || examineAll) {
            numChanged = 0;
            if (examineAll) {
                for (int k = 0; k < N; k++)
                  numChanged += examineExample (k);
              }
            else {
                for (int k = 0; k < N; k++)
                  if (alph[k] != 0 && alph[k] != C)
                    numChanged += examineExample (k);
              }
            if (examineAll == 1)
              examineAll = 0;
            else if (numChanged == 0)
              examineAll = 1;

            //cerr << error_rate() << endl;
            ⟨Diagnostic info 36d⟩
        }
        ⟨Write model parameters 36a⟩
        cerr << "threshold=" << b << endl;
    }
    cout << error_rate() << endl;
    ⟨Write classification output 36c⟩
  }
  ◇
```

Macro referenced in scrap 15a.

Let's define the global variables.

⟨Global variables 16⟩ ≡

```
int N = 0;                    /* N points(rows) */
int d = -1;                   /* d variables */
float C=0.05;
float tolerance=0.001;
float eps=0.001;
float two_sigma_squared=2;

vector<float> alph;           /* Lagrange multipliers */
float b;                      /* threshold */
vector<float> w;              /* weight vector: only for linear kernel */

vector<float> error_cache;

struct sparse_binary_vector {
    vector<int> id;
  };
struct sparse_vector {
    vector<int> id;
    vector<float> val;
  };
typedef vector<float> dense_vector;

bool is_sparse_data = false;
bool is_binary = false;
/* use only one of these */
vector<sparse_binary_vector> sparse_binary_points;
vector<sparse_vector> sparse_points;
vector<dense_vector> dense_points;

vector<int> target;           /* class labels of training data points */
bool is_test_only = false;
bool is_linear_kernel = false;

/* data points with index in [first_test_i .. N)
 * will be tested to compute error rate
 */
int first_test_i = 0;

/*
 * support vectors are within [0..end_support_i)
 */
int end_support_i = -1;
```
◇

Macro defined by scraps 16, 19a, 21b, 22c, 23b, 26b, 29c.
Macro referenced in scrap 15a.

⟨Header files to include 17⟩ ≡

```
#include <vector>
#include <algorithm>
```

```
#include <functional>
◇
```

Macro defined by scraps 17, 29a, 31b, 33.
Macro referenced in scrap 15a.

3.2 The `examineExample` routine

Given the first α_i (with index `i1`), `examineExample(i1)` first checks if it violates the KKT condition by more than `tolerance`, if it does, then looks for the second α_i (with index `i2`) and jointly optimize the two α_i's by calling `takeStep(i1,i2)`.

⟨Functions 18a⟩ ≡

```
int examineExample(int i1)
{
  float y1, alph1, E1, r1;

  y1 = target[i1];
  alph1 = alph[i1];

  if (alph1 > 0 && alph1 < C)
     E1 = error_cache[i1];
  else
     E1 = learned_func(i1) - y1;

  r1 = y1 * E1;
  if ((r1 < -tolerance && alph1 < C)
     || (r1 > tolerance && alph1 > 0))
    {
      /* Try i2 by three ways; if successful, then immediately return 1; */
      ⟨Try argmax E1 - E2 18b⟩
      ⟨Try iterating through the non-bound examples 19b⟩
      ⟨Try iterating through the entire training set 19c⟩
    }

  return 0;
}◇
```

Macro defined by scraps 18a, 20, 24b, 25abc, 26d, 27ab, 28b, 32, 34, 35, 36b.
Macro referenced in scrap 15a.

Use the heuristic to choose the second example from non-bound examples, so that `E1-E2` is maximized.

⟨Try argmax `E1 - E2` 18b⟩ ≡

```
{
  int k, i2;
  float tmax;

  for (i2 = (-1), tmax = 0, k = 0; k < end_support_i; k++)
    if (alph[k] > 0 && alph[k] < C) {
```

```
          float E2, temp;

          E2 = error_cache[k];
          temp = fabs(E1 - E2);
          if (temp > tmax)
            {
              tmax = temp;
              i2 = k;
            }
        }

    if (i2 >= 0) {
        if (takeStep (i1, i2))
          return 1;
      }
  }◇
```

Macro referenced in scrap 18a.

⟨Global variables 19a⟩ ≡

```
  int takeStep(int i1, int i2);
  ◇
```

Macro defined by scraps 16, 19a, 21b, 22c, 23b, 26b, 29c.
Macro referenced in scrap 15a.

If we cannot make progress with the best non-bound example, then try any non-bound examples.

⟨Try iterating through the non-bound examples 19b⟩ ≡

```
  {
    int k, k0;
    int i2;

    for (k0 = (int) (drand48 () * end_support_i), k = k0; k < end_support_i + k0; k++) {
        i2 = k % end_support_i;
        if (alph[i2] > 0 && alph[i2] < C) {
            if (takeStep(i1, i2))
                return 1;
          }
      }
  }◇
```

Macro referenced in scrap 18a.

If we cannot make progress with the non-bound examples, then try any example.

⟨Try iterating through the entire training set 19c⟩ ≡

```
  {
    int k0, k, i2;

    for (k0 = (int)(drand48 () * end_support_i), k = k0; k < end_support_i + k0; k++) {
        i2 = k % end_support_i;
        if (takeStep(i1, i2))
```

```
            return 1;
        }
    }◇
```

Macro referenced in scrap 18a.

3.3 The `takeStep` routine

Now let's write `takeStep` which optimizes two Lagrange multipliers. If successful, return 1, else return 0.

⟨Functions 20⟩ ≡

```
int takeStep(int i1, int i2) {
  int y1, y2, s;
  float alph1, alph2; /* old_values of alpha_1, alpha_2 */
  float a1, a2;       /* new values of alpha_1, alpha_2 */
  float E1, E2, L, H, k11, k22, k12, eta, Lobj, Hobj;

  if (i1 == i2) return 0;

  ⟨Look up alph1, y1, E1, alph2, y2, E2 21a⟩
  s = y1 * y2;

  ⟨Compute L, H 22a⟩
  if (L == H)
    return 0;

  ⟨Compute eta 22b⟩
  if (eta < 0) {
      a2 = alph2 + y2 * (E2 - E1) / eta;
      if (a2 < L)
        a2 = L;
      else if (a2 > H)
        a2 = H;
    }
  else {
    ⟨Compute Lobj, Hobj: objective function at a2=L, a2=H 22d⟩
    if (Lobj > Hobj+eps)
      a2 = L;
    else if (Lobj < Hobj-eps)
      a2 = H;
    else
      a2 = alph2;
  }

  if (fabs(a2-alph2) < eps*(a2+alph2+eps))
    return 0;

  a1 = alph1 - s * (a2 - alph2);
  if (a1 < 0) {
        a2 += s * a1;
```

```
            a1 = 0;
      }
      else if (a1 > C) {
            float t = a1-C;
            a2 += s * t;
            a1 = C;
      }
```

⟨Update threshold to reflect change in Lagrange multipliers 23a⟩
⟨Update weight vector to reflect change in a1 and a2, if linear SVM 23c⟩
⟨Update error cache using new Lagrange multipliers 24a⟩

```
      alph[i1] = a1;  /* Store a1 in the alpha array.*/
      alph[i2] = a2;  /* Store a2 in the alpha array.*/

      return 1;
    }◇
```

Macro defined by scraps 18a, 20, 24b, 25abc, 26d, 27ab, 28b, 32, 34, 35, 36b.
Macro referenced in scrap 15a.

As the SMO algorithm spends most of its time on adjusting the α_i's of the non-boundary examples, an error cache is maintained for them. Each time after a successful optimization step, for the two α_i's, if $0 < \alpha_i < C$, the corresponding E_i is set zero. The E_i's for other α_i's (that have been kept fixed during the optimization step) is updated using Equation 8.

⟨Look up `alph1, y1, E1, alph2, y2, E2` 21a⟩ ≡

```
      alph1 = alph[i1];
      y1 = target[i1];
      if (alph1 > 0 && alph1 < C)
         E1 = error_cache[i1];
      else
         E1 = learned_func(i1) - y1;

      alph2 = alph[i2];
      y2 = target[i2];
      if (alph2 > 0 && alph2 < C)
         E2 = error_cache[i2];
      else
         E2 = learned_func(i2) - y2;
    ◇
```

Macro referenced in scrap 20.

⟨Global variables 21b⟩ ≡

```
     float (*learned_func)(int) = NULL;
    ◇
```

Macro defined by scraps 16, 19a, 21b, 22c, 23b, 26b, 29c.
Macro referenced in scrap 15a.

Compute the feasible range of α_2^{new}. See the graphs on Page 9.

⟨Compute L, H 22a⟩ ≡

```
if (y1 == y2) {
  float gamma = alph1 + alph2;
  if (gamma > C) {
      L = gamma-C;
      H = C;
  }
  else {
      L = 0;
      H = gamma;
  }
}
else {
    float gamma = alph1 - alph2;
    if (gamma > 0) {
        L = 0;
        H = C - gamma;
    }
    else {
        L = -gamma;
        H = C;
    }
}
```
◇

Macro referenced in scrap 20.

⟨Compute `eta` 22b⟩ ≡

```
k11 = kernel_func(i1, i1);
k12 = kernel_func(i1, i2);
k22 = kernel_func(i2, i2);
eta = 2 * k12 - k11 - k22;
```
◇

Macro referenced in scrap 20.

⟨Global variables 22c⟩ ≡

```
float (*kernel_func)(int,int)=NULL;
```
◇

Macro defined by scraps 16, 19a, 21b, 22c, 23b, 26b, 29c.
Macro referenced in scrap 15a.

See Equation 7 on Page 13 for evaluating $\mathscr{L}_D$ at α_2.

⟨Compute Lobj, Hobj: objective function at a2=L, a2=H 22d⟩ ≡

```
{
    float c1 = eta/2;
    float c2 = y2 * (E1-E2)- eta * alph2;
    Lobj = c1 * L * L + c2 * L;
    Hobj = c1 * H * H + c2 * H;
}
```
◇

Macro referenced in scrap 20.

See Equation 9 on Page 14 for updating the threshold b when either of α_1 and α_2 are non-boundary.

⟨Update threshold to reflect change in Lagrange multipliers 23a⟩ ≡

```
{
  float b1, b2, bnew;

  if (a1 > 0 && a1 < C)
    bnew = b + E1 + y1 * (a1 - alph1) * k11 + y2 * (a2 - alph2) * k12;
  else {
      if (a2 > 0 && a2 < C)
          bnew = b + E2 + y1 * (a1 - alph1) * k12 + y2 * (a2 - alph2) * k22;
      else {
          b1 = b + E1 + y1 * (a1 - alph1) * k11 + y2 * (a2 - alph2) * k12;
          b2 = b + E2 + y1 * (a1 - alph1) * k12 + y2 * (a2 - alph2) * k22;
          bnew = (b1 + b2) / 2;
      }
  }

  delta_b = bnew - b;
  b = bnew;
}◇
```

Macro referenced in scrap 20.

⟨Global variables 23b⟩ ≡

```
float delta_b;
◇
```

Macro defined by scraps 16, 19a, 21b, 22c, 23b, 26b, 29c.
Macro referenced in scrap 15a.

A linear SVM can be sped up by only using the weight vector (rather than all of the training examples that correspond to non-zero Lagrange multipliers) when evaluating the learned classification function.

If the joint optimization succeeds, this stored weight vector must be updated to reflect the new Lagrange multiplier values. See Equation 11 on Page 14.

⟨Update weight vector to reflect change in a1 and a2, if linear SVM 23c⟩ ≡

```

  if (is_linear_kernel) {
    float t1 = y1 * (a1 - alph1);
    float t2 = y2 * (a2 - alph2);

    if (is_sparse_data && is_binary) {
        int p1,num1,p2,num2;

        num1 = sparse_binary_points[i1].id.size();
        for (p1=0; p1<num1; p1++)
            w[sparse_binary_points[i1].id[p1]] += t1;
```

```
        num2 = sparse_binary_points[i2].id.size();
        for (p2=0; p2<num2; p2++)
            w[sparse_binary_points[i2].id[p2]] += t2;
    }
    else if (is_sparse_data && !is_binary) {
        int p1,num1,p2,num2;

        num1 = sparse_points[i1].id.size();
        for (p1=0; p1<num1; p1++)
            w[sparse_points[i1].id[p1]] +=
                t1 * sparse_points[i1].val[p1];

        num2 = sparse_points[i2].id.size();
        for (p2=0; p2<num2; p2++)
            w[sparse_points[i2].id[p2]] +=
                t2 * sparse_points[i2].val[p2];
    }
    else
        for (int i=0; i<d; i++)
            w[i] += dense_points[i1][i] * t1 + dense_points[i2][i] * t2;
}◇
```

Macro referenced in scrap 20.

See Equation 8 on Page 13.

⟨Update error cache using new Lagrange multipliers 24a⟩ ≡

```
{
    float t1 = y1 * (a1-alph1);
    float t2 = y2 * (a2-alph2);

    for (int i=0; i<end_support_i; i++)
    if (0 < alph[i] && alph[i] < C)
        error_cache[i] +=  t1 * kernel_func(i1,i) + t2 * kernel_func(i2,i)
                         - delta_b;
    error_cache[i1] = 0.;
    error_cache[i2] = 0.;
}◇
```

Macro referenced in scrap 20.

3.4 Evaluating classification function

We use a function pointer `learned_func` to represent the learned function $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} - b$. It takes the index of the data point k, and computes $f(\mathbf{x}_k)$.

According to the kernel type and input data type, we define the following functions for evaluating the learned classification function.

⟨Functions 24b⟩ ≡

```
float learned_func_linear_sparse_binary(int k) {
  float s = 0.;

  for (int i=0; i<sparse_binary_points[k].id.size(); i++)
    s += w[sparse_binary_points[k].id[i]];

  s -= b;
  return s;
}◇
```

Macro defined by scraps 18a, 20, 24b, 25abc, 26d, 27ab, 28b, 32, 34, 35, 36b.
Macro referenced in scrap 15a.

⟨Functions 25a⟩ ≡

```
float learned_func_linear_sparse_nonbinary(int k) {
  float s = 0.;

  for (int i=0; i<sparse_points[k].id.size(); i++)
  {
      int j = sparse_points[k].id[i];
      float v = sparse_points[k].val[i];
      s += w[j] * v;
  }
  s -= b;
  return s;
}◇
```

Macro defined by scraps 18a, 20, 24b, 25abc, 26d, 27ab, 28b, 32, 34, 35, 36b.
Macro referenced in scrap 15a.

⟨Functions 25b⟩ ≡

```
float learned_func_linear_dense(int k) {
  float s = 0.;

  for (int i=0; i<d; i++)
    s += w[i] * dense_points[k][i];

  s -= b;
  return s;
}◇
```

Macro defined by scraps 18a, 20, 24b, 25abc, 26d, 27ab, 28b, 32, 34, 35, 36b.
Macro referenced in scrap 15a.

⟨Functions 25c⟩ ≡

```
float learned_func_nonlinear(int k) {
  float s = 0.;
  for (int i=0; i<end_support_i; i++)
      if (alph[i] > 0)
          s += alph[i]*target[i]*kernel_func(i,k);
  s -= b;
  return s;
}◇
```

Macro defined by scraps 18a, 20, 24b, 25abc, 26d, 27ab, 28b, 32, 34, 35, 36b.

Macro referenced in scrap 15a.

During initialization, we point `learned_func` one of functions below.

⟨Initialization 26a⟩ ≡

```
        if (is_linear_kernel && is_sparse_data && is_binary)
            learned_func = learned_func_linear_sparse_binary;
        if (is_linear_kernel && is_sparse_data && !is_binary)
            learned_func = learned_func_linear_sparse_nonbinary;
        if (is_linear_kernel && !is_sparse_data)
            learned_func = learned_func_linear_dense;
        if (!is_linear_kernel)
            learned_func = learned_func_nonlinear;
```
◇

Macro defined by scraps 26ac, 28a, 29b.
Macro referenced in scrap 15b.

3.5 Functions to compute dot product

Accroding to the input data type, we have different functions to compute the dot product of two data points.

⟨Global variables 26b⟩ ≡

```
     float (*dot_product_func)(int,int)=NULL;
```
◇

Macro defined by scraps 16, 19a, 21b, 22c, 23b, 26b, 29c.
Macro referenced in scrap 15a.

⟨Initialization 26c⟩ ≡

```
        if (is_sparse_data && is_binary)
            dot_product_func = dot_product_sparse_binary;
        if (is_sparse_data && !is_binary)
            dot_product_func = dot_product_sparse_nonbinary;
        if (!is_sparse_data)
            dot_product_func = dot_product_dense;
```
◇

Macro defined by scraps 26ac, 28a, 29b.
Macro referenced in scrap 15b.

⟨Functions 26d⟩ ≡

```
    float dot_product_sparse_binary(int i1, int i2)
    {
      int p1=0, p2=0, dot=0;
      int num1 = sparse_binary_points[i1].id.size();
      int num2 = sparse_binary_points[i2].id.size();

      while (p1 < num1 && p2 < num2) {
```

```
        int a1 = sparse_binary_points[i1].id[p1];
        int a2 = sparse_binary_points[i2].id[p2];
        if (a1 == a2) {
            dot++;
            p1++;
            p2++;
          }
        else if (a1 > a2)
          p2++;
        else
          p1++;
      }
    return (float)dot;
  }◇
```

Macro defined by scraps 18a, 20, 24b, 25abc, 26d, 27ab, 28b, 32, 34, 35, 36b.
Macro referenced in scrap 15a.

⟨Functions 27a⟩ ≡

```
  float dot_product_sparse_nonbinary(int i1, int i2)
  {
    int p1=0, p2=0;
    float dot = 0.;
    int num1 = sparse_points[i1].id.size();
    int num2 = sparse_points[i2].id.size();

    while (p1 < num1 && p2 < num2) {
        int a1 = sparse_points[i1].id[p1];
        int a2 = sparse_points[i2].id[p2];
        if (a1 == a2) {
            dot += sparse_points[i1].val[p1] * sparse_points[i2].val[p2];
            p1++;
            p2++;
          }
        else if (a1 > a2)
          p2++;
        else
          p1++;
      }
    return (float)dot;
  }◇
```

Macro defined by scraps 18a, 20, 24b, 25abc, 26d, 27ab, 28b, 32, 34, 35, 36b.
Macro referenced in scrap 15a.

⟨Functions 27b⟩ ≡

```
  float dot_product_dense(int i1, int i2)
  {
    float dot = 0.;
    for (int i=0; i<d; i++)
      dot += dense_points[i1][i] * dense_points[i2][i];
```

```
  return dot;
}
```
◇

Macro defined by scraps 18a, 20, 24b, 25abc, 26d, 27ab, 28b, 32, 34, 35, 36b.
Macro referenced in scrap 15a.

3.6 Kernel functions

The linear kernel is simply the dot product. Currently, we have only one non-linear kernel: radial basis function kernel.

⟨Initialization 28a⟩ ≡

```
    if (is_linear_kernel)
        kernel_func = dot_product_func;
    if (!is_linear_kernel)
        kernel_func = rbf_kernel;
```
◇

Macro defined by scraps 26ac, 28a, 29b.
Macro referenced in scrap 15b.

The calculation of $\|\mathbf{x}_1 - \mathbf{x}_2\|^2$ in a Gaussian kernel can be sped up using the following equation:

$$\begin{aligned}\|\mathbf{x}_1 - \mathbf{x}_2\|^2 &= (\mathbf{x}_1 - \mathbf{x}_2)^T(\mathbf{x}_1 - \mathbf{x}_2) \\ &= \mathbf{x}_1^T\mathbf{x}_1 + \mathbf{x}_2^T\mathbf{x}_2 - 2\mathbf{x}_1^T\mathbf{x}_2,\end{aligned}$$

where $\mathbf{x}_i^T\mathbf{x}_i$ can be pre-computed. For each of the d dimensions, directly computing $\|\mathbf{x}_1 - \mathbf{x}_2\|^2$ needs 3 operations:

1. $a = x_{1j} - x_{2j}$
2. $b = a \times a$
3. $s = s + b$.

In comparison, the new method needs only 2 operations:

1. $a = x_{1j} \times x_{2j}$
2. $s = s + a$.

⟨Functions 28b⟩ ≡

```
float rbf_kernel(int i1, int i2)
{
    float s = dot_product_func(i1,i2);
    s *= -2;
    s += precomputed_self_dot_product[i1] + precomputed_self_dot_product[i2];
    return exp(-s/two_sigma_squared);
}
```
◇

Macro defined by scraps 18a, 20, 24b, 25abc, 26d, 27ab, 28b, 32, 34, 35, 36b.
Macro referenced in scrap 15a.

⟨Header files to include 29a⟩ ≡

```
#include <cmath>
◇
```

Macro defined by scraps 17, 29a, 31b, 33.
Macro referenced in scrap 15a.

⟨Initialization 29b⟩ ≡

```

    if (!is_linear_kernel) {
        precomputed_self_dot_product.resize(N);
        for (int i=0; i<N; i++)
            precomputed_self_dot_product[i] = dot_product_func(i,i);
    }
◇
```

Macro defined by scraps 26ac, 28a, 29b.
Macro referenced in scrap 15b.

⟨Global variables 29c⟩ ≡

```

    vector<float> precomputed_self_dot_product;
◇
```

Macro defined by scraps 16, 19a, 21b, 22c, 23b, 26b, 29c.
Macro referenced in scrap 15a.

3.7 Input and output

3.7.1 Get parameters by command line

Finally, we have to get the parameters `C`, `eps`, `tolerance`, etc. (And the input/output file names.) We use the `getopt()` routine to handle this.

⟨Get in parameters 29d⟩ ≡

```
{
  extern char *optarg;
  extern int optind;
  int c;
  int errflg = 0;

  while ((c = getopt (argc, argv, "n:d:c:t:e:p:f:m:o:r:lsba")) != EOF)
    switch (c)
      {
      case 'n':
        N = atoi(optarg);
        break;
      case 'd':
        d = atoi(optarg);
        break;
      case 'c':
        C = atof (optarg);
```

```
      break;
    case 't':
      tolerance = atof(optarg);
      break;
    case 'e':
      eps = atof (optarg);
      break;
    case 'p':
      two_sigma_squared = atof (optarg);
      break;
    case 'f':
      data_file_name = optarg;
      break;
    case 'm':
      svm_file_name = optarg;
      break;
    case 'o':
      output_file_name = optarg;
      break;
    case 'r':
      srand48 (atoi (optarg));
      break;
    case 'l':
      is_linear_kernel = true;
      break;
    case 's':
      is_sparse_data = true;
      break;
    case 'b':
      is_binary = true;
      break;
    case 'a':
      is_test_only = true;
      break;
    case '?':
      errflg++;
    }

if (errflg || optind < argc)
  {
    cerr << "usage: " << argv[0] << " " <<
              "-f  data_file_name\n"
              "-m  svm_file_name\n"
              "-o  output_file_name\n"
              "-n  N\n"
              "-d  d\n"
              "-c  C\n"
              "-t  tolerance\n"
              "-e  epsilon\n"
              "-p  two_sigma_squared\n"
```

```
                    "-r  random_seed\n"
                    "-l  (is_linear_kernel)\n"
                    "-s  (is_sparse_data)\n"
                    "-b  (is_binary)\n"
                    "-a  (is_test_only)\n"
                    ;
          exit (2);
        }
    }◇
```

Macro referenced in scrap 15b.

⟨Variables local to main 31a⟩ ≡

```
    char *data_file_name = "svm.data";
    char *svm_file_name = "svm.model";
    char *output_file_name = "svm.output";
    ◇
```

Macro referenced in scrap 15b.

⟨Header files to include 31b⟩ ≡

```
    #include <iostream>
    #include <cstdlib>
    #include <unistd.h>
    ◇
```

Macro defined by scraps 17, 29a, 31b, 33.
Macro referenced in scrap 15a.

3.7.2 Read in data

The data file is a flat text file, each data point occupies one line in which the class label (+1 or −1) follows the attribute values. Ordinarily, a line will be

`attribute_1_value attribute_2_value ... attribute_d_value target_value`

For sparse format, a line will be

`id_1 val_1 id_2 val_2 ... id_m val_m target_value`

where `id_j` should be between 1 and d. For sparse binary format, a line will be

`id_1 id_2 ... id_m target_value`

here, too, `id_j` should be between 1 and d.

The data is read into `dense_points`, or `sparse_points`, or `sparse_binary_points`, according to the input format.

⟨Read in data 31c⟩ ≡

```
{
  int n;
  if (is_test_only) {
      ifstream svm_file(svm_file_name);
      end_support_i = first_test_i = n = read_svm(svm_file);
      N += n;
  }
  if (N > 0) {
    target.reserve(N);
    if (is_sparse_data && is_binary)
        sparse_binary_points.reserve(N);
    else if (is_sparse_data && !is_binary)
        sparse_points.reserve(N);
    else
        dense_points.reserve(N);
  }
  ifstream data_file(data_file_name);
  n = read_data(data_file);
  if (is_test_only) {
      N = first_test_i + n;
  }
  else {
      N = n;
      first_test_i = 0;
      end_support_i = N;
  }
}◇
```

Macro referenced in scrap 15b.

The actually reading of data is handled by `read_data(istream&)`; it appends data points from the input stream to `dense_points` (or `sparse_points`, or `sparse_binary_points`, depending on input format).

The function `read_data(istream&)` may be called by `read_svm()` to read in the support vectors of previous trained model (if non-linear kernel is used). These support vectors are read into `dense_points` (or `sparse_points`, or `sparse_binary_points`, depending on input format) before the data points in the data file. The starting index of the data points in the data file is `first_test_i`.

⟨Functions 32⟩ ≡

```
int read_data(istream& is)
{
  string s;
  int n_lines;

  for (n_lines = 0; getline(is, s, '\n'); n_lines++) {
    istrstream line(s.c_str());
    vector<float> v;
    float t;
    while (line >> t)
```

```
            v.push_back(t);
         target.push_back(v.back());
         v.pop_back();
         int n = v.size();
         if (is_sparse_data && is_binary) {
             sparse_binary_vector x;
             for (int i=0; i<n; i++) {
                 if (v[i] < 1 || v[i] > d) {
                     cerr << "error: line " << n_lines+1
                         << ": attribute index " << int(v[i]) << " out of range."<<endl;
                     exit(1);
                 }
                 x.id.push_back(int(v[i])-1);
             }
             sparse_binary_points.push_back(x);
         }
         else if (is_sparse_data && !is_binary) {
             sparse_vector x;
             for (int i=0; i<n; i+=2) {
                 if (v[i] < 1 || v[i] > d) {
                     cerr << "data file error: line " << n_lines+1
                          << ": attribute index " << int(v[i]) << " out of range."
                          << endl;
                     exit(1);
                 }
                 x.id.push_back(int(v[i])-1);
                 x.val.push_back(v[i+1]);
             }
             sparse_points.push_back(x);
         }
         else {
             if (v.size() != d) {
                     cerr << "data file error: line " << n_lines+1
                          << " has " << v.size() << " attributes; should be d=" << d
                          <<endl;
                     exit(1);
             }
             dense_points.push_back(v);
         }
       }
       return n_lines;
     }◇
```

Macro defined by scraps 18a, 20, 24b, 25abc, 26d, 27ab, 28b, 32, 34, 35, 36b.
Macro referenced in scrap 15a.

⟨Header files to include 33⟩ ≡

```
#include <string>
#include <vector>
#include <iostream>
#include <fstream>
```

```
#include <strstream>
◇
```

Macro defined by scraps 17, 29a, 31b, 33.
Macro referenced in scrap 15a.

3.7.3 Saving and loading model parameters

The output order of the model paramters will be

1. The number of attributes d.
2. The flag `is_sparse_data`
3. The flag `is_binary`
4. The flag `is_linear_kernel`
5. The threshold b
6. If the linear kernel is used:
   (a) The weight vector $\mathbf{w}$
7. If non-linear kernel is used
   (a) Kernel paramters (e.g., $2\sigma^2$ for radial basis function kernel)
   (b) The number of support vectors
   (c) The Lagrange multipliers of the support vectors
   (d) The support vectors, one per line

⟨Functions 34⟩ ≡

```
void write_svm(ostream& os) {
    os << d << endl;
    os << is_sparse_data << endl;
    os << is_binary << endl;
    os << is_linear_kernel << endl;
    os << b << endl;
    if (is_linear_kernel) {
        for (int i=0; i<d; i++)
            os << w[i] << endl;
    }
    else {
        os << two_sigma_squared << endl;
        int n_support_vectors=0;
        for (int i=0; i<end_support_i; i++)
            if (alph[i] > 0)
                n_support_vectors++;
        os << n_support_vectors << endl;
        for (int i=0; i<end_support_i; i++)
            if (alph[i] > 0)
```

```
                os << alph[i] << endl;
        for (int i=0; i<end_support_i; i++)
            if (alph[i] > 0) {
                if (is_sparse_data && is_binary) {
                    for (int j=0; j<sparse_binary_points[i].id.size(); j++)
                        os << (sparse_binary_points[i].id[j]+1) << ' ';
                }
                else if (is_sparse_data && !is_binary) {
                    for (int j=0; j<sparse_points[i].id.size(); j++)
                        os << (sparse_points[i].id[j]+1) << ' '
                           << sparse_points[i].val[j] << ' ';
                }
                else {
                    for (int j=0; j<d; j++)
                        os << dense_points[i][j] << ' ';
                }
                os << target[i];
                os << endl;
            }
    }
}◇
```

Macro defined by scraps 18a, 20, 24b, 25abc, 26d, 27ab, 28b, 32, 34, 35, 36b.
Macro referenced in scrap 15a.

⟨Functions 35⟩ ≡

```
int read_svm(istream& is) {
    is >> d;
    is >> is_sparse_data;
    is >> is_binary;
    is >> is_linear_kernel;
    is >> b;
    if (is_linear_kernel) {
        w.resize(d);
        for (int i=0; i<d; i++)
            is >> w[i];
    }
    else {
        is >> two_sigma_squared;
        int n_support_vectors;
        is >> n_support_vectors;
        alph.resize(n_support_vectors, 0.);
        for (int i=0; i<n_support_vectors; i++)
                is >> alph[i];
        string dummy_line_to_skip_newline;
        getline(is, dummy_line_to_skip_newline, '\n');
        return read_data(is);
    }
    return 0;
}◇
```

Macro defined by scraps 18a, 20, 24b, 25abc, 26d, 27ab, 28b, 32, 34, 35, 36b.
Macro referenced in scrap 15a.

⟨Write model parameters 36a⟩ ≡

```
{
if (!is_test_only && svm_file_name != NULL) {
    ofstream svm_file(svm_file_name);
    write_svm(svm_file);
}
}◇
```

Macro referenced in scrap 15b.

3.8 Compute error rate

⟨Functions 36b⟩ ≡

```
float
error_rate()
{
    int n_total = 0;
    int n_error = 0;
    for (int i=first_test_i; i<N; i++) {
       if (learned_func(i) > 0 != target[i] > 0)
           n_error++;
       n_total++;
    }
    return float(n_error)/float(n_total);
}
◇
```

Macro defined by scraps 18a, 20, 24b, 25abc, 26d, 27ab, 28b, 32, 34, 35, 36b.
Macro referenced in scrap 15a.

The classification output is $\mathbf{w}\mathbf{x}_i - b$ for each data point $\mathbf{x}_i$, one per line.

⟨Write classification output 36c⟩ ≡

```
{
    ofstream output_file(output_file_name);
    for (int i=first_test_i; i<N; i++)
       output_file << learned_func(i) << endl;
}◇
```

Macro referenced in scrap 15b.

⟨Diagnostic info 36d⟩ ≡

```
/* L_D */
{
#if 0
  float s = 0.;
  for (int i=0; i<N; i++)
     s += alph[i];
  float t = 0.;
```

```
  for (int i=0; i<N; i++)
      for (int j=0; j<N; j++)
         t += alph[i]*alph[j]*target[i]*target[j]*kernel_func(i,j);
  cerr << "Objective function=" << (s - t/2.) << endl;
  for (int i=0; i<N; i++)
      if (alph[i] < 0)
         cerr << "alph[" << i << "]=" << alph[i] << " < 0" << endl;
  s = 0.;
  for (int i=0; i<N; i++)
    s += alph[i] * target[i];
  cerr << "s=" << s << endl;
  cerr << "error_rate=" << error_rate() << '\t';
#endif
  int non_bound_support =0;
  int bound_support =0;
  for (int i=0; i<N; i++)
    if (alph[i] > 0) {
       if (alph[i] < C)
          non_bound_support++;
       else
          bound_support++;
    }
  cerr << "non_bound=" << non_bound_support << '\t';
  cerr << "bound_support=" << bound_support << endl;
}
◇
```

Macro referenced in scrap 15b.

⟨Is the objective function increasing? 37⟩ ≡

```
{
    float c1 = eta/2;
    float c2 = y2 * (E1-E2)- eta * alph2;
    float t1 = c1 * alph2 * alph2 + c2 * alph2;
    float t2 = c1 * a2 * a2 + c2 * a2;
    if (t2-t1 < 0)
        cerr <<  "change=" << t2 - t1 << endl;
}◇
```

Macro never referenced.

3.9 Multiclass

The SMO code handles only binary classification. To handle the multiclass case, we use the following script. The input data format is similar to that of SMO, except the class labels with be 0, 1, ..., $n-1$, if there are n classes. The script `smo_multi_class` builds n binary classifiers, $f_c(\mathbf{x}) = sgn(\mathbf{w}_c\mathbf{x} - b_c)$, one for each of the c classes. The classification rule of the multiclass classifier is

$$\hat{c} = \arg\max_c \mathbf{w}_c\mathbf{x} - b_c.$$

The models are saved in ${svm_file_name_prefix}.$c$, where $c = 0, 1, \ldots, n-1$.

The ith line of the classification output file contains the n values of $\mathbf{w}_c\mathbf{x}_i - b_c$ of the data point $\mathbf{x}_i$.

"scripts/smo_multi_class" 38 ≡

```
#!/bin/sh
## smo_multi_class: multi-class wrapper for SMO
##   Usage: smo_multi_class options -- smo-options
##      options must include:
##           -c number-of-classes
##           -f data-file-name
##           -o output-file-name
##           -m svm-file-name-prefix
##      The 'smo-options' after '--' are passed to smo.

if [ $# -lt 8 ]
then
    sed -n '/^##/s/^## //p' $0 >&2
    exit 1
fi

number_of_classes=0
data_file_name=NULL
output_file_name=NULL
svm_file_name_prefix=NULL

while getopts c:f:o:m: c
do
    case $c in
        c) number_of_classes=$OPTARG;;
        f) data_file_name=$OPTARG;;
        o) output_file_name=$OPTARG;;
        m) svm_file_name_prefix=$OPTARG;;
       \?) sed -n '/^##/s/^## //p' $0 >&2
           exit 1;;
    esac
done
shift `expr $OPTIND - 1`

if [ $output_file_name = NULL ] || [ $svm_file_name_prefix = NULL ]
then
       sed -n '/^##/s/^## //p' $0 >&2
       exit 1
fi

if [ $number_of_classes -ge 2 ]
then
    :
else
```

```
    echo "error: invalid number of classes ($number_of_classes); should be >= 2" >&2
    exit 1
fi

if [ ! -f $data_file_name ]
then
    echo "error: cannot open data file: $data_file_name" >&2
    exit 1
fi

tmp_data_file_name=../tmp/multiclasstmpsvm.data
tmp_output_file_name=../tmp/multiclasstmpsvm.output
all_target_file_name=../tmp/multiclasstmpsvm.all_target
cat $data_file_name | awk '{ print $NF }' > $all_target_file_name
printf "" > $output_file_name
i=0
while [ $i -lt $number_of_classes ]
do
    printf "class $i: "
    individual_svm_file_name=${svm_file_name_prefix}.$i
    cat $data_file_name |
    awk '{ for (i=1; i<NF; i++)
             printf("%s ",$i);
           if ($NF == '$i')
             printf("1\n");
           else
             printf("-1\n");
         }' > $tmp_data_file_name
    ../c/smo "$@" \
         -f $tmp_data_file_name \
         -o $tmp_output_file_name \
         -m $individual_svm_file_name
    paste $output_file_name  $tmp_output_file_name > ${output_file_name}.tmp
    mv ${output_file_name}.tmp $output_file_name
    rm $tmp_data_file_name $tmp_output_file_name
    i=`expr $i + 1`
done
printf "multi-class: "
paste $output_file_name $all_target_file_name |
awk 'BEGIN { n_total = 0.
             n_error = 0.
           }
     {
        best_val = $1
        best_i = 1
        for (i=2; i<NF; i++)
            if ($i > best_val) {
                best_val = $i
                best_i = i
            }
```

```
            best_i--
            if (best_i != $NF)
                n_error++
            n_total++
        }
        END { print n_error/n_total }'
rm $all_target_file_name
◇
```

3.10 Makefiles

"Makefile" 40a ≡

```
all: smo.tex smo.dvi smo.ps ccode
ccode:
^Icd c; make
smo.dvi:        smo.tex \
                pic/fig1.eps pic/fig2.eps pic/fig3.eps \
                pic/fig4.eps pic/fig5.eps pic/fig6.eps \
                pic/I0-I4.eps
pic/fig1.eps:   pic/fig1.pic
pic/fig2.eps:   pic/fig2.pic
pic/fig3.eps:   pic/fig3.pic
pic/fig4.eps:   pic/fig4.pic
pic/fig5.eps:   pic/fig5.pic
pic/fig6.eps:   pic/fig6.pic

include $(HOME)/doc/rules.mk
◇
```

"c/Makefile" 40b ≡

```
all: smo

# CXXFLAGS=-g
CXXFLAGS=-O3
◇
```

References

Christopher J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998. URL `citeseer.nj.nec.com/burges98tutorial.html`.

S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy. Improvements to platt's SMO algorithm for SVM classifier design. *Neural Computation*, 13(3):637–649, March 2001.

J. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Scholkopf, C. Burges, and A. Smola, editors, *Advances in kernel methods: support vector learning*. MIT Press, 1998.

A The weight vectors of the parallel supporting planes

Suppose $H_1 : \mathbf{a} \cdot \mathbf{x} - b_1 = 0$, and $H_2 : \mathbf{a} \cdot \mathbf{x} - b_2 = 0$ are the two parallel planes. Because they are parallel, they can have the same weight vector $\mathbf{a}$. Let $b' = \frac{b_1+b_2}{2}$, and $\delta = b_1 - b'$. So $b_1 = b' + \delta$ and $b_2 = b' - \delta$. We can rewrite the equations as

$$\begin{aligned} H_1 : &\quad \mathbf{a} \cdot \mathbf{x} - (b' + \delta) = 0 \\ H_2 : &\quad \mathbf{a} \cdot \mathbf{x} - (b' - \delta) = 0 \end{aligned}$$

or

$$\begin{aligned} H_1 : &\quad \mathbf{a} \cdot \mathbf{x} - b' = \delta \\ H_2 : &\quad \mathbf{a} \cdot \mathbf{x} - b' = -\delta \end{aligned}$$

Divide the equations by δ, we have

$$\begin{aligned} H_1 : &\quad \frac{1}{\delta}\mathbf{a} \cdot \mathbf{x} - \frac{b'}{\delta} = 1 \\ H_2 : &\quad \frac{1}{\delta}\mathbf{a} \cdot \mathbf{x} - \frac{b'}{\delta} = -1 \end{aligned}$$

Let $\mathbf{w}' = \frac{1}{\delta}\mathbf{a}$ and $b = \frac{b'}{\delta}$, we have

$$\begin{aligned} H_1 : &\quad \mathbf{w} \cdot \mathbf{x} - b = +1 \\ H_2 : &\quad \mathbf{w} \cdot \mathbf{x} - b = -1 \end{aligned}$$

B The objective function of the dual problem

For the convex quadratic primal problem

$$\begin{aligned} \underset{\mathbf{w},b,\xi_i}{\text{minimize}} &\quad \tfrac{1}{2}\mathbf{w}^T\mathbf{w} + C\textstyle\sum_{i=1}^{N} \xi_i \\ \text{subject to} &\quad y_i(\mathbf{w}^T\mathbf{x}_i - b) + \xi_i - 1 \geq 0, \quad 1 \leq i \leq N \\ &\quad \xi_i \geq 0, \qquad\qquad\qquad\qquad 1 \leq i \leq N, \end{aligned}$$

the Lagrangian is

$$\begin{aligned} \mathscr{L}(\mathbf{w}, b, \xi_i; \boldsymbol{\alpha}, \boldsymbol{\beta}) &= \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{N} \xi_i \\ &\quad - \sum_{i=1}^{N} \alpha_i \left[y_i(\mathbf{w}^T\mathbf{x}_i - b) + \xi_i - 1\right] - \sum_{i=1}^{N} \mu_i\xi_i \\ &= \frac{1}{2}\mathbf{w}^T\mathbf{w} + \sum_{i=1}^{N} (C - \alpha_i - \mu_i)\,\xi_i \\ &\quad - \left(\sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i^T\right)\mathbf{w} - \left(\sum_{i=1}^{N} \alpha_i y_i\right) b + \sum_{i=1}^{N} \alpha_i, \end{aligned}$$

where $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ are the Lagrange multipliers, the Wolfe dual problem is

$$\begin{array}{lll} \underset{\boldsymbol{\alpha},\boldsymbol{\beta}}{\text{maximize}} & \mathscr{L}(\mathbf{w}, b, \xi_i; \boldsymbol{\alpha}, \boldsymbol{\beta}) & \\ \text{subject to} & \frac{\partial \mathscr{L}}{\partial \mathbf{w}} = \mathbf{0} & \\ & \frac{\partial \mathscr{L}}{\partial b} = 0 & \\ & \frac{\partial \mathscr{L}}{\partial \xi_i} = 0 & 1 \leq i \leq N \\ & \boldsymbol{\alpha} \geq \mathbf{0} & \\ & \boldsymbol{\beta} \geq \mathbf{0}. & \end{array}$$

The constraint $\frac{\partial \mathscr{L}}{\partial \mathbf{w}} = \mathbf{0}$ implies

$$\mathbf{w}^T - \sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i^T = \mathbf{0},$$

or, equivalently,

$$\mathbf{w} = \sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i.$$

The constraint $\frac{\partial \mathscr{L}}{\partial b} = 0$ implies

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

The constraints $\frac{\partial \mathscr{L}}{\partial \xi_i} = 0$ imply

$$C - \alpha_i - \mu_i = 0, \qquad 1 \leq i \leq N.$$

Note that $\boldsymbol{\alpha} \geq \mathbf{0}$, $\boldsymbol{\beta} \geq \mathbf{0}$, and we have

$$0 \leq \alpha_i \leq C.$$

Substituting these results into $\mathscr{L}(\mathbf{w}, b, \xi_i; \boldsymbol{\alpha}, \boldsymbol{\beta})$:

$$\begin{aligned} \mathscr{L}(\mathbf{w}, b, \xi_i; \boldsymbol{\alpha}, \boldsymbol{\beta}) &= \frac{1}{2}\mathbf{w}^T\mathbf{w} + \sum_{i=1}^{N} 0 \times \xi_i - \mathbf{w}^T\mathbf{w} - 0 \times b + \sum_{i=1}^{N} \alpha_i \\ &= -\frac{1}{2}\mathbf{w}^T\mathbf{w} + \sum_{i=1}^{N} \alpha_i \\ &= \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} y_i y_j \mathbf{x}_i^T \mathbf{x}_j \alpha_i \alpha_j \end{aligned}$$

To summarize, the dual problem is

$$\begin{array}{lll} \underset{\boldsymbol{\alpha}}{\text{maximize}} & \mathscr{L}_D = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} y_i y_j \mathbf{x}_i^T \mathbf{x}_j \alpha_i \alpha_j & \\ \text{subject to} & \sum_{i=1}^{N} y_i \alpha_i = 0 & \\ & 0 \leq \alpha_i \leq C & 1 \leq i \leq N. \end{array}$$