Sequential Minimal Optimization for SVM

Contents

1 Introduction to Support Vector Machine (SVM) ........................................ 2
   1.1 Linear SVM .................................................................................. 2
   1.2 The dual problem .......................................................................... 3
   1.3 Non-linear SVM .......................................................................... 4
   1.4 Imperfect separation .................................................................... 5
   1.5 The KKT conditions ...................................................................... 6
   1.6 Checking KKT condition without using threshold \( b \) ................. 7

2 SMO Algorithm ...................................................................................... 9
   2.1 Optimize two \( \alpha_i \)'s .................................................................... 9
   2.2 SMO Algorithm: Updating after a successful optimization step ..... 13
   2.3 SMO Algorithm: Pick two \( \alpha_i \)'s for optimization .................. 14

3 C++ Implementation ............................................................................... 15
   3.1 The main routine ......................................................................... 15
   3.2 The examineExample routine ...................................................... 18
   3.3 The takeStep routine .................................................................... 20
   3.4 Evaluating classification function .............................................. 24
   3.5 Functions to compute dot product .............................................. 26
   3.6 Kernel functions .......................................................................... 28
   3.7 Input and output ........................................................................... 29
      3.7.1 Get parameters by command line ....................................... 29
      3.7.2 Read in data ........................................................................ 31
      3.7.3 Saving and loading model parameters .............................. 34
   3.8 Compute error rate ....................................................................... 36
   3.9 Multiclass ..................................................................................... 37
   3.10 Makefiles .................................................................................... 40

A The weight vectors of the parallel supporting planes ..................... 41

B The objective function of the dual problem ........................................ 42
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 NÜWEB 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 introduction 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 linear 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 \(\mathbf{x}_i \in \mathbb{R}^d\) and \(y_i \in \{\pm 1\}\). We would like to learn a linear separating hyperplane classifier:

\[
f(\mathbf{x}) = \text{sgn}(\mathbf{w} \cdot \mathbf{x} - b).
\]

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,

\[
H_1 : y = \mathbf{w} \cdot \mathbf{x} - b = +1,
\]
\[ H_2 : \ y = w \cdot x - b = -1, \]

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 \( w \) so that \( H_1 \) will be \( y = w \cdot x - b = +1 \), and \( H_2 \) will be \( y = w \cdot x - b = -1 \). 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 \textit{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 \( H : w \cdot x - b = 0 \) is \( \frac{|w \cdot x_i - b|}{\|w\|} = \frac{1}{\|w\|} \), and the distance between \( H_1 \) and \( H_2 \) is \( \frac{2}{\|w\|} \). So, in order to maximize the distance, we should minimize \( \|w\| = w^T w \) with the condition that there are no data points between \( H_1 \) and \( H_2 \):

\[ w \cdot x - b \geq +1, \quad \text{for positive examples} \quad y_i = +1, \]
\[ w \cdot x - b \leq -1, \quad \text{for negative examples} \quad y_i = -1. \]

These two conditions can be combined into
\[ y_i (w \cdot x_i - b) \geq 1. \]

So our problem can be formulated as
\[ \min_{w, b} \frac{1}{2} w^T w \quad \text{subject to} \quad y_i (w \cdot x_i - b) \geq 1. \]

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

Introducing Lagrange multipliers \( \alpha_1, \alpha_2, \ldots, \alpha_N \geq 0 \), we have the following Lagrangian:
\[ \mathcal{L}(w, b, \alpha) \equiv \frac{1}{2} w^T w - \sum_{i=1}^{N} \alpha_i y_i (w \cdot x_i - b) + \sum_{i=1}^{N} \alpha_i. \]

1.2 The dual problem

We can solve the Wolfe dual instead: maximize \( \mathcal{L}(w, b, \alpha) \) with respect to \( \alpha \), subject to the constraints that the gradient of \( \mathcal{L}(w, b, \alpha) \) with respect to the primal variables \( w \) and \( b \) vanish:
\[ \frac{\partial \mathcal{L}}{\partial w} = 0, \quad (1) \]
\[ \frac{\partial \mathcal{L}}{\partial b} = 0, \quad (2) \]
and that 
\[ \alpha \geq 0. \]

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 \( \mathcal{L}(\mathbf{w}, b, \alpha) \), we have 
\[ \mathcal{L}_D \equiv \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j, \]
in which the primal variables are eliminated.

When we solve \( \alpha_i \), 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 
\[ f(\mathbf{x}) = \text{sgn}(\mathbf{w} \cdot \mathbf{x} + b) = \text{sgn}(\sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i \cdot \mathbf{x} + b) = \text{sgn}(\sum_{i=1}^{N} \alpha_i y_i (\mathbf{x}_i \cdot \mathbf{x}) + b). \]

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

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 \( k(x_i, x_j) \) 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)

\[
K(x_i, x_j) = e^{-\|x_i - x_j\|^2 / 2\sigma^2}.
\]

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

There exists a mapping \( \Phi \) and an expansion

\[
K(x, y) = \sum_i \Phi(x)_i \Phi(y)_i,
\]

if and only if, for any \( g(x) \) such that

\[
\int g(x)^2 dx \text{ is finite,}
\]

then

\[
\int K(x, y)g(x)g(y) dx dy \geq 0.
\]

### 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

\[
\begin{align*}
    w \cdot x_i - b & \geq +1 - \xi_i \quad \text{for } y_i = +1, \\
    w \cdot x_i - b & \leq -1 + \xi_i \quad \text{for } y_i = -1, \\
    \xi_i & \geq 0, \quad \forall i.
\end{align*}
\]

and we add to the objective function a penalizing term:

\[
\minimize_{w, b, \xi} \frac{1}{2} w^T w + C \left( \sum_i \xi_i \right)^m
\]

where \( m \) is usually set to 1, which gives us

\[
\begin{align*}
    & \minimize_{w, b, \xi} \quad \frac{1}{2} w^T w + C \sum_{i=1}^N \xi_i \\
    \text{subject to} & \quad y_i (w^T x_i - b) + \xi_i - 1 \geq 0, \quad 1 \leq i \leq N \\
    & \quad \xi_i \geq 0, \quad 1 \leq i \leq N
\end{align*}
\]
Introducing Lagrange multipliers $\alpha$, $\beta$, the Lagrangian is

$$L(w, b, \xi; \alpha, \beta) = \frac{1}{2}w^Tw + \sum_{i=1}^{N} \xi_i - \sum_{i=1}^{N} \alpha_i [y_i(w^T x_i - b) + \xi_i - 1] - \sum_{i=1}^{N} \mu_i \xi_i$$

$$= \frac{1}{2}w^Tw + \sum_{i=1}^{N} (C - \alpha_i - \mu_i) \xi_i$$

Neither the $\xi_i$’s, nor their Lagrange multipliers, appear in the Wolfe dual problem:

$$\text{maximize} \; \mathcal{L}_D(\alpha) = \sum_{i} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j x_i \cdot x_j$$

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 $\alpha_i$ is now bounded above by $C$ instead of $\infty$. For details, see Appendix B.

The solution is again given by

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

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 $\mathcal{L}(w, b, \alpha, \beta)$ with respect to the primal variables $w, b, \xi$ vanishes (this must always be satisfied by the dual problem), and that for $1 \leq i \leq N,$

$$\alpha_i (y_i (w^T x_i - b) + \xi_i - 1) = 0,$$

$$\mu_i \xi_i = 0.$$

Depending on the value of $\alpha_i$, we have three cases to consider:
1. If $\alpha_i = 0$, then $\mu_i = C - \alpha_i = C > 0$. From Equation 4, $\xi_i = 0$. so we have
\[ y_i(w^T x_i - b) - 1 \geq 0. \]

2. If $0 < \alpha_i < C$, from Equation 3, we have
\[ y_i(w^T x_i - b) + \xi_i - 1 = 0 \]  
Equation 5

Note that $\mu_i = C - \alpha_i > 0$, so $\xi_i = 0$ (Equation 4). Substituting into Equation 5, we have
\[ y_i(w^T x_i - b) - 1 = 0. \]

3. If $\alpha_i = C$, then from Equation 3, we have
\[ y_i(w^T x_i - b) + \xi_i - 1 = 0 \]  
Equation 6

Note that $\mu_i = C - \alpha_i = 0$, we have $\xi_i \geq 0$. So
\[ y_i(w^T x_i - b) - 1 \leq 0. \]

The quantity $y_i(w^T x_i - b) - 1$ can be computed as
\[ R_i = y_i(w^T x_i - b) - y_i^2 = y_i(w^T x_i - b - y_i) = y_i E_i \]
where $E_i = w^T x_i - b - y_i = u_i - y_i$ is the prediction error.

To summarize, the KKT condition implies:
- $\alpha_i = 0 \Rightarrow R_i \geq 0,$
- $0 < \alpha_i < C \Rightarrow R_i \approx 0,$
- $\alpha_i = C \Rightarrow R_i \leq 0.$

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 $y_i(w^T x_i - b) - 1$ (which must $\geq 0$ for all $i$ if the KKT condition is satisfied) can also be written as
\[ y_i(w^T x_i - b) - 1 \]
\[ = y_i(w^T x_i - b) - y_i^2 \]
\[ - y_i(w^T x_i - y_i - b) \]
\[ = y_i(F_i - b), \]
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 optimizing the two Lagrange multipliers $\alpha_i, \alpha_j$.)

This notation is useful because the KKT conditions

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

can be written as

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

where

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

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

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

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 $\alpha_i$, these comparisons automatically finds the second $\alpha_i$ for joint optimization in SMO.
2 SMO Algorithm

2.1 Optimize two $\alpha_i$'s

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

$$L_D = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j x_i \cdot x_j,$$

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

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

Without loss of generality, suppose we are optimizing $\alpha_1$, $\alpha_2$, from an old set of feasible solution: $\alpha_1^{old}$, $\alpha_2^{old}$, $\alpha_3$, ..., $\alpha_N$. (For initialization, we can set $\alpha_i^{old} = 0$.)

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

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

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

\[ \begin{align*}
\alpha_1 & = 0 \\
\alpha_2 & = C
\end{align*} \quad \begin{align*}
\alpha_1 & = C \\
\alpha_2 & = 0
\end{align*} \]

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

Fixing the other $\alpha_i$'s, the objective function can be written as

$$L_D = \alpha_1 + \alpha_2 + \text{Const}.$$

\[ -\frac{1}{2} \left( y_1 y_1 x_1^T x_1 \alpha_1^2 + y_2 y_2 x_2^T x_2 \alpha_2^2 + 2 y_1 y_2 x_1^T x_2 \alpha_1 \alpha_2 \right) + 2 \left( \sum_{i=3}^{N} \alpha_i y_i x_i^T \right) (y_1 x_1 \alpha_1 + y_2 x_2 \alpha_2) + \text{Const}. \]
Let $K_{11} = x_1^T x_1$, $K_{22} = x_2^T x_2$, $K_{12} = x_1^T x_2$, and

$$v_j = \sum_{i=3}^N \alpha_i y_i x_i^T x_j$$

$$= x_j^T w_1 - \alpha_1 y_1 x_1^T x_j - \alpha_2 y_2 x_2^T x_j$$

$$= (x_j^T w_1 - b_1) + b_2 - \alpha_1 y_1 x_1^T x_j - \alpha_2 y_2 x_2^T x_j$$

$$= v_j^{old} + b^{old} - \alpha_1 y_1 x_1^T x_j - \alpha_2 y_2 x_2^T x_j,$$

where $v_j^{old} = x_j^T w_1^{old} - b_1$ is the output of $x_j$ under old parameters.

$$\mathcal{L}_D = \alpha_1 + \alpha_2 - \frac{1}{2} \left( K_{11} \alpha_1^2 + K_{22} \alpha_2^2 + 2sK_{12} \alpha_1 \alpha_2 \right) + 2y_1 y_1 + 2y_2 y_2$$

$$= \gamma - \alpha_2 \left( K_{11} (\gamma - \alpha_2)^2 + K_{22} \alpha_2^2 \right) + 2y_1 y_1 + 2y_2 y_2$$

$$= (1-s) \alpha_2 \left( K_{11} (\gamma - \alpha_2)^2 - \frac{1}{2} K_{22} \alpha_2^2 - sK_{12} (\gamma - \alpha_2) \alpha_2 \right)$$

$$= (1-s) \alpha_2 \left( K_{11} (\gamma - \alpha_2)^2 + sK_{11} \gamma \alpha_2 - \frac{1}{2} K_{11} s^2 \alpha_2^2 - \frac{1}{2} K_{22} \alpha_2^2 \right)$$

$$= (1-s) \alpha_2 \left( K_{11} (\gamma - \alpha_2)^2 + sK_{12} \gamma \alpha_2 + y_1 y_1 \gamma + y_1 y_1 + \alpha_2 \right) + 2y_2 y_2$$

$$= \left( \frac{1}{2} K_{11} - \frac{1}{2} K_{22} + K_{12} \right) \alpha_2^2$$

$$+ (1-s + sK_{11} \gamma - K_{12} \gamma + y_2 y_1 - y_2 y_2) \alpha_2$$

$$= \left( \frac{1}{2} (2K_{12} - K_{11} - K_{22}) \alpha_2^2 \right)$$

Let $\eta \equiv 2K_{12} - K_{11} - K_{22}$. The coefficient of $\alpha_2$ is

$$1 - s + sK_{11} \gamma - sK_{12} \gamma + y_2 y_1 - y_2 y_2$$

$$= 1 - s + sK_{11} (\alpha_1^{old} + \alpha_2^{old}) - sK_{12} (\alpha_1^{old} + \alpha_2^{old})$$

$$+ y_2 (\gamma_1^{old} + \gamma_2^{old} y_1 K_{11} - \alpha_2^{old} y_2 K_{12})$$
So the objective function is

\[ \mathcal{L}_D = \frac{1}{2} \eta \alpha_2^2 + (y_2(E_1^{\text{old}} - E_2^{\text{old}}) - \eta \alpha_2^{\text{old}})^2 + \text{Const.} \]

The first and second derivatives are

\[
\frac{d\mathcal{L}_D}{d\alpha_2} = \eta \alpha_2 + (y_2(E_1^{\text{old}} - E_2^{\text{old}}) - \eta \alpha_2^{\text{old}}),
\]

\[
\frac{d^2\mathcal{L}_D}{d\alpha_2^2} = \eta.
\]

Note that \( \eta = 2K_{12} - K_{11} - K_{22} \leq 0 \). Proof: Let \( K_{11} = x_1^T x_1, K_{12} = x_1^T x_2, K_{22} = x_2^T x_2 \). Then \( \eta = -(x_2 - x_1)^T (x_2 - x_1) = -\|x_2 - x_1\|^2 \leq 0 \).

Let \( \frac{d\mathcal{L}_D}{d\alpha_2} = 0 \), and we have

\[
\alpha_2^{\text{new}} = \frac{y_2(E_1^{\text{old}} - E_2^{\text{old}}) - \eta \alpha_2^{\text{old}}}{\eta} = \alpha_2^{\text{old}} + \frac{y_2(E_2^{\text{old}} - E_1^{\text{old}})}{\eta}.
\]

If \( \eta < 0 \), the above equation gives us the unconstrained maximum point \( \alpha_2^{\text{new}} \). It must be checked against the feasible range. Let \( s = y_1 y_2 \), and \( \gamma = \alpha_1^{\text{old}} + s \alpha_2^{\text{old}} \). The range of \( \alpha_2 \) 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 $\alpha_2$ be $L$, maximum be $H$. Then

$$\alpha_{2,\text{new,clipped}} = \begin{cases} H, & \text{if } H < \alpha_{2,\text{new}}, \\ \alpha_{2,\text{new}}, & \text{if } L \leq \alpha_{2,\text{new}} \leq H \\ L, & \text{if } \alpha_{2,\text{new}} < L. \end{cases}$$

To summarize, given $\alpha_1$, $\alpha_2$ (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 $\alpha$’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 $\alpha_i^{\text{new}}$ to be the one with larger objective function value. The objective function is

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

2.2 SMO Algorithm: Updating after a successful optimization step

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

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

The change in $E$ is

$$
\Delta E(x, y) = \Delta \alpha_1 y_1 x_1^T x + \Delta \alpha_2 y_2 x_2^T x - \Delta b. \ (8)
$$

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

$$
0 = E(x, y)^{\text{new}}$$
\[ E(x, y)^{\text{old}} + \Delta E(x, y) = E(x, y)^{\text{old}} + \Delta \alpha_1 y_1 x_1^T x + \Delta \alpha_2 y_2 x_2^T x - \Delta b \]

we have

\[ \Delta b = E(x, y)^{\text{old}} + \Delta \alpha_1 y_1 x_1^T x + \Delta \alpha_2 y_2 x_2^T x. \] (9)

If \( \alpha_1 = 0 \), we can only say \( y_1 E_1^{\text{new}} \geq 0 \); similarly, if \( \alpha_1 = C \), we have \( y_1 E_1^{\text{new}} \leq 0 \). If both \( \alpha_1 \) and \( \alpha_2 \) take values 0 or \( C \), the original SMO algorithm computes two values of the new \( b \) for \( \alpha_1 \) and \( \alpha_2 \) using Equation 9, and takes the average. This is regarded as problematic by Keerthi et al. (2001).

Similarly, from

\[ F(x, y) = \sum_{i=1}^{N} \alpha_i y_i x_i^T x - y \]

we have

\[ \Delta F(x, y) = \Delta \alpha_1 y_1 x_1^T x + \Delta \alpha_2 y_2 x_2^T x. \] (10)

For the weight vector of linear kernels,

\[ w = \sum_{i=1}^{N} \alpha_i y_i x_i, \]

\[ \Delta w = \Delta \alpha_1 y_1 x_1 + \Delta \alpha_2 y_2 x_2. \] (11)

### 2.3 SMO Algorithm: Pick two \( \alpha_i \)’s for optimization

The heuristics for picking two \( \alpha_i \)’s for optimization in the original SMO paper are as follows:

- The outer loop selects the first \( \alpha_i \), the inner loop selects the second \( \alpha_i \) 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 \( \alpha_i \), 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 starts a sequential scan through all the examples, also starting at a random position.

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 \( \alpha_i \) for joint optimization. There are two variations when the outer loop deals only with the non-boundary examples.
• The first \(\alpha\) is selected sequentially from all the non-boundary examples. If the first \(\alpha_i\) violates the KKT condition when compared with \(\alpha_j\) with \(F_j = b_{\text{low}}\), or \(F_j = b_{\text{up}}\), then select \(\alpha_j\) as the second \(\alpha\).

• The two \(\alpha\)'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 \(b_{\text{low}}\) and \(b_{\text{up}}\) from the two sets.

Proof: The two sets are non-empty if \(I_0 \neq \emptyset\). If \(I_0 = \emptyset\), then \(\alpha_1, \alpha_2\) 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 \(\alpha_1\) and \(\alpha_2\) 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 \(\alpha_i\) for optimization. It alternates between a sweep through all the examples (\(\text{examineAll}=1\)) and as many sweeps as possible through the non-boundary examples (\(\text{examineAll}=0\)). If an example \(k\) violates the KKT condition more than \(\epsilon\), it is selected as the first \(\alpha_i\), and \(\text{examineExample}(k)\) is called, which returns 1 if positive progress is made to improve the objective function (with two changed \(\alpha_i\)'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 */
    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;
    ⟨ Write classification output 36c ⟩
} 

Macro referenced in scrap 15a.

Let’s define the global variables.
⟨ Global variables 16 ⟩
```c
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>
```

17
3.2 The examineExample routine

Given the first $\alpha_i$ (with index $i_1$), examineExample($i_1$) first checks if it violates the KKT condition by more than tolerance, if it does, then looks for the second $\alpha_i$ (with index $i_2$) and jointly optimize the two $\alpha_i$’s by calling takeStep($i_1$,$i_2$).

\[
\langle \text{Functions 18a} \rangle \equiv \int \text{examineExample}(\text{int } i_1) \\
\{ \text{float } y_1, \text{alpha}_1, E_1, r_1; \}
\]

\[
\text{y}_1 = \text{target}[i_1]; \\
\text{alpha}_1 = \text{alpha}[i_1]; \\
\text{if } (\text{alpha}_1 > 0 \text{ && alpha}_1 < C) \\
\text{E}_1 = \text{error_cache}[i_1]; \\
\text{else} \\
\text{E}_1 = \text{learned_func}(i_1) - y_1; \\
\text{r}_1 = y_1 * E_1; \\
\text{if } ((r_1 < -\text{tolerance }\text{ && alpha}_1 < C) \\
|| (r_1 > \text{tolerance }\text{ && alpha}_1 > 0)) \\
\{ \\
\text{/* Try i2 by three ways; if successful, then immediately return 1; */} \\
\text{(Try argmax E1 - EZ 18b)} \\
\text{(Try iterating through the non-bound examples 19b)} \\
\text{(Try iterating through the entire training set 19c)} \\
\}
\]

\[
\text{return } 0; \}
\]

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

\[
\langle \text{Try argmax E1 - EZ 18b} \rangle \equiv \int \\
\{ \text{int } k, i_2; \text{float } t_{\text{max}}; \}
\]

\[
\text{for } (i_2 = (-1), t_{\text{max}} = 0, k = 0; k < \text{end_support_i}; k++) \\
\text{if } (\text{alpha}[k] > 0 \text{ && alpha}[k] < C) \}
\]
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))
3.3 The takeStep routine

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

```c
⟨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;
    }
    return 1;
}¨
```

Macro referenced in scrap 18a.
a1 = 0;
}
else if (a1 > C) {
    float t = a1-C;
    a2 += s * t;
    a1 = C;
}

\[
\text{(Update threshold to reflect change in Lagrange multipliers 23a)}
\]
\[
\text{(Update weight vector to reflect change in a1 and a2, if linear SVM 23c)}
\]
\[
\text{(Update error cache using new Lagrange multipliers 24a)}
\]

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

return 1;
}

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

\[
\text{(Look up alph1, y1, E1, alph2, y2, E2 21a) } \equiv \\
\text{ alph1 = alph[i1];} \\
\text{ y1 = target[i1];} \\
\text{ if (alph1 > 0 && alph1 < C) } \\
\text{ E1 = error_cache[i1];} \\
\text{ else } \\
\text{ E1 = learned_func(i1) - y1;}
\]

\[
\text{ alph2 = alph[i2];} \\
\text{ y2 = target[i2];} \\
\text{ if (alph2 > 0 && alph2 < C) } \\
\text{ E2 = error_cache[i2];} \\
\text{ else } \\
\text{ E2 = learned_func(i2) - y2;}
\]

\[
\text{Macro referenced in scrap 20.}
\]

\[
\text{(Global variables 21b) } \equiv \\
\text{ float (*learned_func)(int) = NULL;}
\]

Compute the feasible range of $\alpha_2^{\text{new}}$. See the graphs on Page 11.
(Compute $L$, $H$) $\equiv$

```c
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 $\eta$) $\equiv$

```c
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) $\equiv$

```c
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 $\mathcal{L}$ at $\alpha_2$.

(Compute $L_{obj}$, $H_{obj}$: objective function at $a_2=L$, $a_2=H$) $\equiv$

```c
{ 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 $\alpha_1$ and $\alpha_2$ are non-boundary.

(Update threshold to reflect change in Lagrange multipliers 23a) $\equiv$

\{
  float $b_1$, $b_2$, $b_{\text{new}}$;
  if ($a_1 > 0$ && $a_1 < C$)
    $b_{\text{new}} = b + E_1 + y_1 \times (a_1 - \alpha_1) \times k_{11} + y_2 \times (a_2 - \alpha_2) \times k_{12}$;
  else {
    if ($a_2 > 0$ && $a_2 < C$)
      $b_{\text{new}} = b + E_2 + y_1 \times (a_1 - \alpha_1) \times k_{12} + y_2 \times (a_2 - \alpha_2) \times k_{22}$;
    else {
      $b_1 = b + E_1 + y_1 \times (a_1 - \alpha_1) \times k_{11} + y_2 \times (a_2 - \alpha_2) \times k_{12}$;
      $b_2 = b + E_2 + y_1 \times (a_1 - \alpha_1) \times k_{12} + y_2 \times (a_2 - \alpha_2) \times k_{22}$;
      $b_{\text{new}} = (b_1 + b_2) / 2$;
    }
  }
  $\delta_b = b_{\text{new}} - b$;
  $b = b_{\text{new}}$;
\}

Macro referenced in scrap 20.

(Update weight vector to reflect change in $a_1$ and $a_2$, if linear SVM 23c) $\equiv$

if (is_linear_kernel) {
  float $t_1 = y_1 \times (a_1 - \alpha_1)$;
  float $t_2 = y_2 \times (a_2 - \alpha_2)$;
  if (isSparseData && isBinary) {
    int $p_1$, $num_1$, $p_2$, $num_2$;
    $num_1 = \text{sparse_binary_points[i1].id.size}()$;
    for ($p_1 = 0; p_1 < \text{num}_1; p_1++$)
      $w[\text{sparse_binary_points[i1].id[p1]]} += t_1$;

  }
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;
}

See Equation 8 on Page 13.

\langle\text{Update error cache using new Lagrange multipliers 24a}\rangle \equiv \{
    \text{float } t1 = y1 * (a1-alph1);
    \text{float } t2 = y2 * (a2-alph2);
    \text{for (int } i=0; i<end\_support\_i; i++)
        \text{if (0 < alph[i] && alph[i] < C)}
            \text{error_cache[i]} += t1 * kernel\_func(i1,i) + t2 * kernel\_func(i2,i)
                - \text{delta}_b;
    \text{error_cache[i1]} = 0.0;
    \text{error_cache[i2]} = 0.0;
\}\}

3.4 Evaluating classification function

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

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

\langle\text{Functions 24b}\rangle \equiv
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.
During initialization, we point \texttt{learned\_func} one of functions below.

\begin{verbatim}
(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;
\end{verbatim}

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

\subsection*{3.5 Functions to compute dot product}

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

\begin{verbatim}
(Global variables 26b) ≡

 float (*dot_product_func)(int,int)=NULL;
\end{verbatim}

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

\begin{verbatim}
(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;
\end{verbatim}

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

\begin{verbatim}
(Funtions 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) {
     \ldots
   }

   \ldots
 }
\end{verbatim}
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];

27
3.6 Kernel functions

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

\[
\text{(Initialization 28a) } \equiv 
\]
\[
\begin{cases}
\text{if (is_linear_kernel)} \\
\quad \text{kernel_func} = \text{dot_product_func}; \\
\text{if (!is_linear_kernel)} \\
\quad \text{kernel_func} = \text{rbf_kernel};
\end{cases}
\]

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. \( s = s + a. \)

\[
\text{(Functions 28b) } \equiv 
\]
\[
\text{float rbf_kernel(int i1, int i2)} \\
\{ \\
\quad \text{float s = dot_product_func(i1,i2);} \\
\quad s **= -2; \\
\quad s += \text{precomputed_self_dot_product[i1]} + \text{precomputed_self_dot_product[i2]}; \\
\quad \text{return exp(-s/two_sigma_squared);} \\
\}
\]
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;

       switch (c)
       {
       case 'n':
          N = atoi(optarg);
          break;
       case 'd':
          d = atoi(optarg);
          break;
       case 'c':
          C = atof (optarg);
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);
}
}

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.
{  
  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++) {
    istringstream 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>
3.7.3 Saving and loading model parameters

The output order of the model parameters 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 \( w \)
7. If non-linear kernel is used
   (a) Kernel parameters (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) ≡

```cpp
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 \( wx_i - b \) for each data point \( x_i \), 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.;
}

36
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, \( f_c(x) = \text{sgn}(w_c x - b_c) \), one for each of the c classes. The classification rule of the multiclass classifier is
\[
\hat{c} = \arg \max_c w_c x - b_c.
\]
The models are saved in \( ${svm_file_name_prefix}c \), where \( c = 0, 1, \ldots, n - 1 \).

The \( i \)th line of the classification output file contains the \( n \) values of \( w_c x_i - b_c \) of the data point \( x_i \).

"scripts/smo_multi_class" 38

```bash
#!/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

38
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  n_error = 0. \n} \n{ \n  best_val = $1 \n  best_i = 1 \n  for (i=2; i<NF; i++) \n    if ($i > best_val) { \n      best_val = $i \n      best_i = i \n    } \n  } \n  printf("\n" "$\n  printf("\n\n\n39


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:
    '/cd 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

pic/I0-I4.pdf: pic/I0-I4.eps
pic/fig1.pdf: pic/fig1.eps
pic/fig2.pdf: pic/fig2.eps
pic/fig3.pdf: pic/fig3.eps
pic/fig4.pdf: pic/fig4.eps
pic/fig5.pdf: pic/fig5.eps
pic/fig6.pdf: pic/fig6.eps

include $(HOME)/doc/rules.mk

"c/Makefile" 40b ≡

all: smo

# CXXFLAGS=-g
CXXFLAGS=-O3

40
References


A The weight vectors of the parallel supporting planes

Suppose $H_1 : a \cdot x - b_1 = 0$, and $H_2 : a \cdot x - b_2 = 0$ are the two parallel planes. Because they are parallel, they can have the same weight vector $a$. 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

$$H_1 : a \cdot x - (b' + \delta) = 0$$
$$H_2 : a \cdot x - (b' - \delta) = 0$$

or

$$H_1 : a \cdot x - b' = \delta$$
$$H_2 : a \cdot x - b' = -\delta$$

Divide the equations by $\delta$, we have

$$H_1 : \frac{1}{\delta} a \cdot x - \frac{b'}{\delta} = 1$$
$$H_2 : \frac{1}{\delta} a \cdot x - \frac{b'}{\delta} = -1$$

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

$$H_1 : w' \cdot x - b = +1$$
$$H_2 : w' \cdot x - b = -1$$
The objective function of the dual problem

For the convex quadratic primal problem

\[
\begin{align*}
\text{minimize} & \quad \frac{1}{2} w^T w + C \sum_{i=1}^{N} \xi_i \\
\text{subject to} & \quad y_i (w^T x_i - b) + \xi_i - 1 \geq 0, \quad 1 \leq i \leq N \\
& \quad \xi_i \geq 0, \quad 1 \leq i \leq N,
\end{align*}
\]

the Lagrangian is

\[
\mathcal{L}(w, b, \xi; \alpha, \beta) = \frac{1}{2} w^T w + C \sum_{i=1}^{N} \xi_i - \sum_{i=1}^{N} \alpha_i [y_i (w^T x_i - b) + \xi_i - 1] - \sum_{i=1}^{N} \mu_i \xi_i
\]

\[
= \frac{1}{2} w^T w + \sum_{i=1}^{N} (C - \alpha_i - \mu_i) \xi_i - \left( \sum_{i=1}^{N} \alpha_i y_i x_i^T \right) w - \left( \sum_{i=1}^{N} \alpha_i y_i \right) b + \sum_{i=1}^{N} \alpha_i,
\]

where \( \alpha, \beta \) are the Lagrange multipliers, the Wolfe dual problem is

\[
\begin{align*}
\text{maximize} \quad & \mathcal{L}(w, b, \xi; \alpha, \beta) \\
\text{subject to} \quad & \frac{\partial \mathcal{L}}{\partial w} = 0 \\
& \frac{\partial \mathcal{L}}{\partial b} = 0 \\
& \frac{\partial \mathcal{L}}{\partial \xi_i} = 0, \quad 1 \leq i \leq N \\
& \alpha \geq 0 \\
& \beta \geq 0.
\end{align*}
\]

The constraint \( \frac{\partial \mathcal{L}}{\partial w} = 0 \) implies

\[w^T - \sum_{i=1}^{N} \alpha_i y_i x_i^T = 0,\]

or, equivalently,

\[w = \sum_{i=1}^{N} \alpha_i y_i x_i.\]

The constraint \( \frac{\partial \mathcal{L}}{\partial b} = 0 \) implies

\[\sum_{i=1}^{N} \alpha_i y_i = 0.\]
The constraints \( \frac{\partial \mathcal{L}}{\partial \xi_i} = 0 \) imply

\[ C - \alpha_i - \mu_i = 0, \quad 1 \leq i \leq N. \]

Note that \( \alpha \geq 0, \beta \geq 0 \), and we have

\[ 0 \leq \alpha_i \leq C. \]

Substituting these results into \( \mathcal{L}(w, b, \xi; \alpha, \beta) \):

\[
\mathcal{L}(w, b, \xi; \alpha, \beta) = \frac{1}{2} w^T w + \sum_{i=1}^{N} 0 \times \xi_i - w^T w - 0 \times b + \sum_{i=1}^{N} \alpha_i
\]

\[
= \frac{1}{2} w^T w + \sum_{i=1}^{N} \alpha_i
\]

\[
= \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j x_i^T x_j \alpha_i \alpha_j
\]

To summarize, the dual problem is

\[
\text{maximize } \mathcal{L}_D = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j x_i^T x_j \alpha_i \alpha_j
\]

subject to

\[ \sum_{i=1}^{N} y_i \alpha_i = 0 \]

\[ 0 \leq \alpha_i \leq C \quad 1 \leq i \leq N. \]