A Parallel Multiclass Support Vector Machine

Author: Ovidiu Fritsch oafritsch@ucdavis.edu

1 Introduction

In this document, we break down Support Vector Machine (SVM) theory and walk through our implementation.

2 Theory

An SVM is a tool that classifies data into two or more classes. Consider first a binary, linear SVM. In this formulation, we assume that there are only 2 classes and that the training data is linearly separable. Suppose we have M training points in N dimensions: $\{(x_1, y_1), (x_2, y_2), ..., (x_M, y_M)\}$, $x_i \in \mathbf{R}^N$, and $y_i \in \{+1, -1\}$. This is the input to the SVM. The output is a hyperplane, defined by the normal vector $\mathbf{w}$ and a scalar b, that best separates the training data. To classify a new point, we look at which side of the hyperplane the point lies on. Let's examine this high level description in more detail.

2.1 Primal Problem

The optimal hyperplane is found by optimizing the following quadratic function subject to M linear constraints (See Appendix A). It is called the *Primal Problem*.

$$\min \mathcal{P}(\mathbf{w}, b) = \frac{1}{2}\mathbf{w}^2 \quad \text{subject to} \quad y_i(\mathbf{w} \cdot x_i - b) \geq 1 \; \forall i. \tag{1}$$

2.2 Dual Problem

Equation (1) can be solved using commercial quadratic programming algorithms. However, these algorithms are too slow for large datasets. Therefore, we use the method of Lagrange Multipliers to reformulate the objective. (See Appendix B).

$$\max \mathcal{L}(\alpha) = \sum_i \alpha_i - \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j x_i x_j y_i y_j \quad \text{subject to} \begin{cases} \forall i \; \alpha_i \geq 0, \\ \sum_i y_i \alpha_i = 0. \end{cases} \tag{2}$$

2.3 Linear Inseparability

When the input data is linearly inseparable, there is no solution to the above problem. We can either change the model to accommodate for linearly inseparable data (Soft Margin), or we can transform the data to make it linearly separable (Kernel Function).

2.3.1 Soft Margin

When data is not linearly separable, there are no such $\mathbf{w}$ and b that will satisfy the constraints in (1) $\forall i$. By introducing positive slack variables $\xi_i \forall i$, we allow each sample i to violate its hard margin by ξ_i. Equation 7 becomes:

$$\min_{\mathbf{w}, b} \frac{1}{2}\mathbf{w}^2 + C\sum_i \xi_i \quad \text{subject to} \quad y_i(\mathbf{w} \cdot x_i - b) \geq 1 - \xi_i. \tag{3}$$

where C controls the penalty of the slack terms. When the soft-margin primal form (3) is converted to the dual form, nothing changes except the inequality constraint in (2) becomes a box constraint:

$$0 \leq \alpha_i \leq C \; \forall i \tag{4}$$

2.3.2 Kernel Function

If the data is not linearly separable, then we can apply a function ϕ to each x_i that transforms x_i into a higher dimensional space such that the data is linearly separable. However, computing $\phi(x_i) \cdot \phi(x_j)$ is expensive if $\phi(x)$ has more dimensions than x. Kernel functions $K(x_i, x_j)$ allow us to sidestep this inefficiency because they compute $\phi(x_i) \cdot \phi(x_j)$ only in terms of x_i and x_j. The objective function in (2) can therefore be written as:

$$\max \mathcal{L}(\alpha) = \sum_i \alpha_i - \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j K(x_i, x_j) y_i y_j \tag{5}$$

2.4 Sequential Minimal Optimization

Sequential Minimal Optimization (SMO) is an algorithm that optimizes the dual objective function (5) by optimizing only two α's at each iteration. Suppose that in a given iteration, we choose to optimize α_1 and α_2 from an old set of feasible solutions α_1^{old}, α_2^{old}, ... , α_M. Since $\sum_i \alpha_i y_i = 0$,

$$\alpha_1 y_1 + \alpha_2 y_2 = \alpha_1^{old} y_1 + \alpha_2^{old} y_2 \tag{6}$$

Each iteration of SMO finds the constrained (Appendix C) optimum (Appendix D) of α_2 and then solves for α_1 using (6). There is a special case in which we cannot solve for the optimal α_2 because of division by zero. In this case, we evaluate (5) at the lower and upper bounds and choose the α_2 that maximizes the evaluation. We continue this process of choosing 2 α_i's until they stop changing within some threshold.

2.5 Heuristics for Choosing Lagrangian to Multiply

At each iteration of SMO, we would ideally like to choose the 2 α_i's whose optimization leads to the largest increase in (5). The outer loop selects the first α_i, then the inner loop selects the second α_i that maximizes $\mid E_2 - E_1 \mid$. 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 only the examples that violate the KKT conditions.

3 Python Implementation

3.1 Overall Structure

We have implemented 3 classes: SVC (Support Vector Classification), MPSVM (Multi Process SVM), and SVM (Support Vector Machine). SVM implements the binary classification task. MPSVM parallelizes the binary classification task by first partitioning the dataset. For each partition, MPSVM creates a child process that instantiates an SVM object, which is trained on the partitioned dataset. MPSVM also parallelizes the prediction task. Finally, the class visible to the user is SVC. Whereas SVM and MPSVM can only classify binary datasets, SVC implements multiclass classification by creating a One vs. Rest binary sub-classifier for each class. For each sub-classification task, SVC creates a process that instantiates an MPSVM object to train the sub-classifier. Note that (1) there are two levels of parallelism here, and (2) that each class has two public functions: `fit(x, y)` and `predict(x)`. In the following sections we explain the implementation of each class in more detail.

3.2 SVM

`fit(x, y)` finds the optimal parameters to the dual problem (2) by using SMO. Before running SMO, which makes many calls to $K(x_i, y_j)$, we precompute $K(x_i, y_j)$ for each pair x_i, y_i in the training set and store it in the Kernel Matrix $\mathbf{K}_{i,j}$ to speed up the computation.

Next we call `smo()` which returns the optimal Lagrange Multipliers `alphas` and the hyperplane intercept `b`. `smo()` implements the outer loop of the algorithm. At each iteration, we either loop again over *each* training example, or over each *non-boundary* training example, calling `examine_example(i2)` where `i2` is the index of the example. `examine_example(i2)` determines if `alphas[i2]` is eligible for optimization by checking if the `i2`th example violates the KKT conditions. If `alphas[i2]` is eligible for optimization, then we choose the second training example `i1` based on the heuristics in Section 2.5. Finally, we call `take_step(i1, i2, E2)`, which attempts to change `alphas[i2]` to its clipped optimal value. If this is successful, then we solve for α_1^{new} based on Equation (7) and set `alphas[i1]` $= \alpha_1^{new}$. If `alphas[i2]` is changed, then `smo()` increments `num_changed`, which is reset to `0` at the beginning of the outer loop. `smo()` terminates when either it has made a full pass through the training set and `num_changed` $= 0$ or when the outer loop has exceeded the maximum number of iterations, a tuning parameter set by the user.

`predict(x)` predicts the class of each example $x_i/$ by taking the sign of $\mathbf{w}^\top \cdot x_i - b$. Remember that $\mathbf{w} = \sum_i \alpha_i y_i x_i$, so we use the right side of this equation instead of directly computing $\mathbf{w}$. We optimize this summation by only including the is whose $\alpha_i > 0$. The corresponding x_is are called *support vectors* because the decision to classify a new sample depends only on them.

3.3 MPSVM

Both `fit(x, y)` and `predict(x)` are parallelized in MPSVM. The user may specify how many processes P to use, but we limit this to 5. Thus, $P = \min(5, \texttt{numProcesses})$.

`fit(x, y)` splits the dataset into P partitions. Each process creates its own `SVM` object and calls `fit(x', y')`, where `x'` and `y'` are the partitioned data. We facilitate this by using Python's `multiprocessing` module, which also provides a `Queue` class to facilitate IPC. After each process completes training, we append the trained `SVM` object to the queue. The main process waits for all the child processes to terminate before terminating itself.

`predict(x)` uses the classifiers on the queue to predict each x_i. Let us first consider the task of predicting a single x_i. `fit(x, y)` provides P classifiers. We think of each prediction by classifier P_j on x_i as a vote for which class x_i should belong to. The class we assign x_i to is the one with the most votes. To predict all x_i, we split x into P partitions x' and use parallelism similar to how we did in $fit(x, y)$. The difference is that instead of placing `SVM` objects on the queue, we place the prediction vector `y_hat'` for x'. We also include the start index of `x'` in `x` along with `y_hat'` because the main process must reorder each `y_hat'` so that the original order is preserved.

3.4 SVC

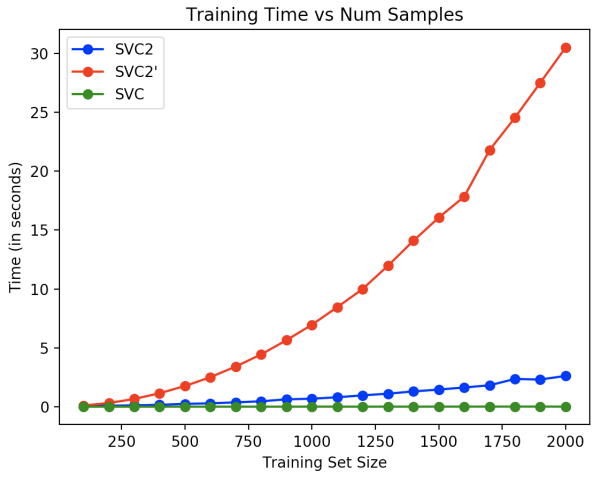
The SVC class is the interface to the user. It supports both binary and multiclass classification. To do multiclass classification on N classes, one popular method is to create N binary classifiers where each binary classifier C_i classifies a training example as either class i or not class i. We use another layer of parallelism here by spawning a new process for each binary classification task and placing the trained `MPSVM` classifier on the IPC queue. Since the maximum number of processes is 5 yet the data may belong to more than 5 classes, we start another sub-classification process (if there are any left) only when an existing one terminates. This ensures that no more than 5 processes are executing at once.

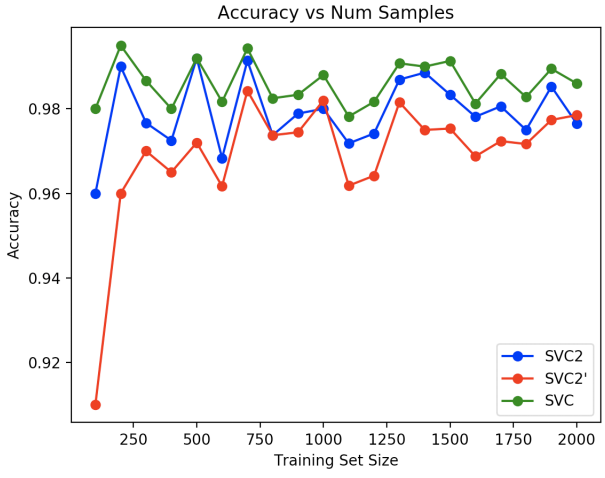
`predict(x)` uses each sub-classifier to predict each x_i. Consider again the task of predicting a single x_i. We use a similar voting technique as in MPSVM, except we now accumulate votes for each class and take the maximum. Each sub-classifier either votes for its own class (if it predicts +1) or for all the other classes (if it predicts -1). We use parallelism here by creating a process for each sub-classifier's prediction task and using the same logic as in `fit(x, y)` to ensure only 5 processes execute at a time.

4 Results

We evaluate the performance of our Support Vector Machine by comparing its accuracy and efficiency to scikit-learn's SVM library. We also investigate how the performance of our model changes when we vary the distribution of the data set as well as internal SVM parameters.

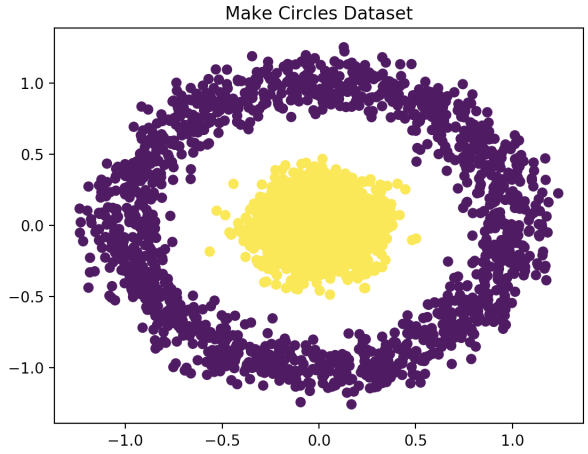
Our first dataset consists of 5 dimensional points drawn from one of two normal distributions. We vary the number of samples and report the time and accuracy of Sklearn's model (**SVC**), our model (**SVC2**), and our model *without* parallelism at the MPSVM layer(**SVC2'**).

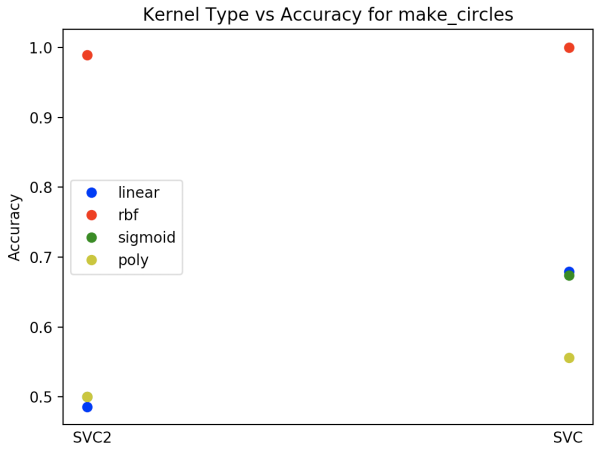




scikit-learn's **SVC** outperforms our model by less than 5 seconds for data with less than 2000 rows. However, our model significantly outperforms our model without parallelism. In the Figure on the right, we see that our test set accuracy is practically as good as scikit-learn's **SVC** accuracy.

Our second dataset is the makeCircles linearly inseparable dataset from Sklearn. It's purpose is to show that we can accurately classify such data by using Kernel functions. We fix the training size at 1000, vary the kernel functions, and report the accuracy for SVC and SVC2.

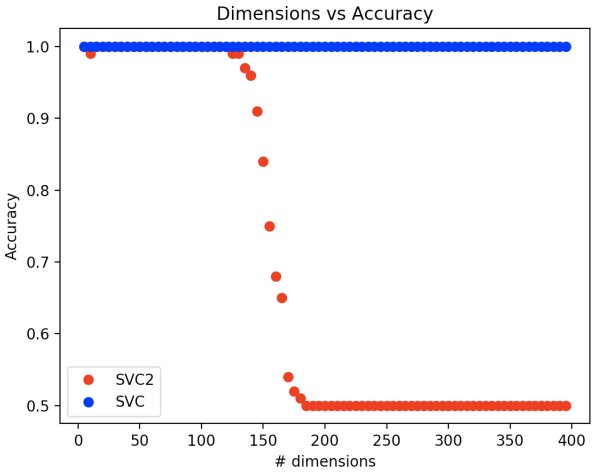


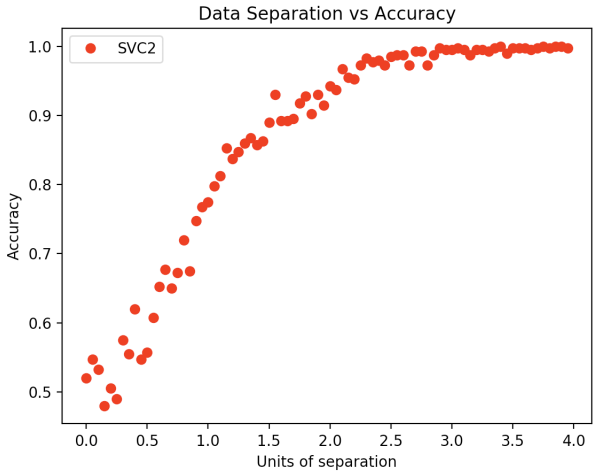


As expected, the linear classifier cannot separate the data so the accuracy is low at about 50%. However, the Radial Basis Function Kernel (rbf) was able to transform the data so to make it linearly separable and thus the accuracy is high (near 100%).

Now, we will see how time and accuracy are affected by increased dimensionality of the datasets while keeping the training size fixed at 500 samples (Left Figure). We also set the maximum number of iterations parameter to 1000; if we don't do this, our model will take too much time on high dimensional data. The data for this example is drawn from two normal distribution in N dimensions.

In the Right Figure, we vary the separation of the training data and plot the accuracy.

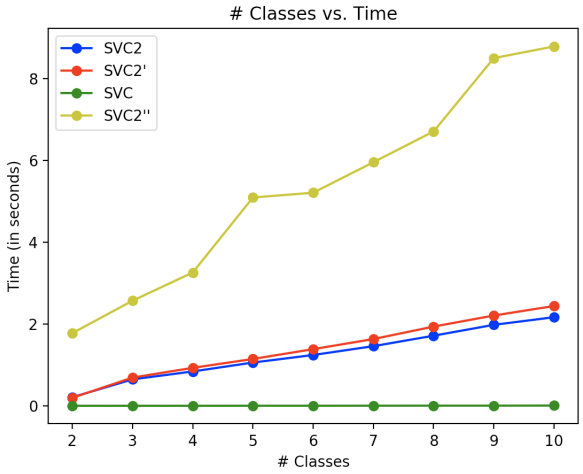


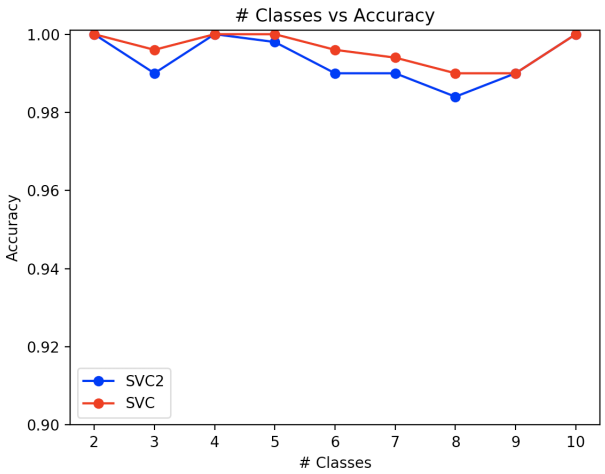


In the figure on the left, we can see the limitations of our model. It performs poorly when the dimensionality of the problem exceeds 150. This is because the number of iterations exceeds 1000, in which case we terminate the process. This results in a sub-optimal model and therefore, a weaker accuracy on the test set.

In the figure on the right, we get what we expect. The closer the data from different classes is clustered together, the harder it is to draw a decision boundary so the lower the accuracy of the prediction.

Finally, we show the efficiency and accuracy of our multiclass classification as a function of the number of classes. We fix the number of samples at 500, the number of dimensions at 3, and the data separation at 4. We compare 4 classes. **SVC** and **SVC2** are the same as before. **SVC2'** is our model without parallelism at the *SVC* layer, and **SVC2"** is out model without parallelism at the *MPSVM* layer.





In the figure on the left, we can see that parallelism at the SVC layer has a relatively small effect on the training time compared to parallelism at the MPSVM layer. This is because MPSVM creates exponentially more processes then SVC. For every process that SVC spawns, MPSVM spawns multiple processes. This means that the majority of the multiprocessing efficiency comes from MPSVM, which is why we are seeing very poor performance when MPSVM parallelism is removed (yellow line), and marginally poorer performance when only SVC parallelism is removed (red line). Sklearn (green line) outperforms our model (blue line) as usual.

On the right, we can see that our SVC2 classifier accurately classifies the data regardless of the number of classes. For this dataset, Sklearn's SVC slightly outperforms our SVC2 model.

5 Appendix

A. Derivation of Primal Form

Given $\mathbf{w} \in \mathbf{R}^N$ and $b \in \mathbf{R}$, all $\mathbf{x}$ that satisfy the following equation

$$\mathbf{w} \cdot \mathbf{x} - b = 0 \tag{7}$$

define a hyperplane H in $\mathbf{E}^N$. However, we want to separate our training data. So for each x_i we impose the following constraints:

$$\mathbf{w} \cdot x_i^+ - b \geq +1 \tag{8}$$

$$\mathbf{w} \cdot x_i^- - b \leq -1 \tag{9}$$

where x_i^+ has $y_i = +1$ and x_i^- has $y_i = -1$. To make these equations more compact, we include $y_i \in +1, -1$:

$$y_i(\mathbf{w} \cdot x_i - b) \geq 1 \tag{10}$$

Points x_i such that

$$y_i(\mathbf{w} \cdot x_i - b) = 1 \tag{11}$$

are called *support vectors* and they define hyperplanes H^+ and H^- that are parallel to H. The optimal hyperplane H maximizes the distance D between H^+ and H^-. To obtain an expression for D, we consider two points x^+ and x^- that lie on H^+ and H^-, respectively.

$$\begin{aligned} D &= (x^+ - x^-) \cdot \frac{\mathbf{w}}{|| \mathbf{w} ||} \\ &= \frac{2}{|| \mathbf{w} ||} \end{aligned} \tag{12}$$

Since $|| \mathbf{w} || = \mathbf{w}^\top \mathbf{w} = \mathbf{w}^2$ and maximizing $\frac{2}{||\mathbf{w}||}$ is the same as minimizing $\frac{1}{2} || \mathbf{w} ||$, our objective is:

$$\min_{\mathbf{w},b} \frac{1}{2}\mathbf{w}^2 \quad \text{subject to} \quad y_i(\mathbf{w} \cdot x_i - b) \geq 1 \; \forall i. \tag{13}$$

B. Derivation of Dual Form

Recall that this is the Primal Form:

$$\min \mathcal{P} = \frac{1}{2}\mathbf{w}^2 \quad \text{subject to} \quad y_i(\mathbf{w} \cdot x_i - b) \geq 1 \; \forall i. \tag{14}$$

The Lagrangian method allows us to combine the constraints of an optimization problem into a new objective function $\mathcal{L}$ such that if $p*$ is the optimal value of minimizing $\mathcal{P}$, then $\mathcal{L} \leq p*$. Therefore, we want to maximize $\mathcal{L}$. $\mathcal{L}$ is given by:

$$\mathcal{L}(\alpha, \mathbf{w}, b) = \frac{1}{2}\mathbf{w}^2 - \sum_i \alpha_i (y_i(\mathbf{w} \cdot x_i + b) - 1) \tag{15}$$

Now we take the partial derivatives of $\mathcal{L}$ with respect to $\mathbf{w}$ and b:

$$\begin{aligned} &\frac{d\mathcal{L}}{d\mathbf{w}} = \mathbf{w} - \sum_i^M \alpha_i y_j x_j && \frac{d\mathcal{L}}{db} = -\sum_j^M \alpha_i y_i \\ &\mathbf{w} - \sum_j^M \alpha_j y_j x_j = 0 && \sum_i^M \alpha_i y_i = 0 \\ &\mathbf{w} = \sum_j^M \alpha_j y_j x_j \end{aligned} \tag{16}$$

Now by substituting $\mathbf{w}$ back into $\mathcal{L}$, we get the dual form of the optimization problem:

$$\begin{aligned} &= \frac{1}{2}(\sum_j \alpha_j y_j x_j)^2 - \sum_i^M \alpha_i (y_i((\sum_j \alpha_j y_j x_j) \cdot x_i + b) - 1) \\ &= \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j x_i x_j y_i y_j - \sum_i \sum_j \alpha_i \alpha_j x_i x_j y_i y_j - \sum_i \alpha_i y_i b + \sum_i \alpha_i \\ &= \sum_i \alpha_i - \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j x_i x_j \qquad (\text{Remember} \sum_i y_i \alpha_i = 0.) \end{aligned} \tag{17}$$

C. Lower and Upper Bounds for α_i

Without loss of generality, suppose that we are optimizing α_1 and α_2, and that y_1 and y_2 are the classes of x_1 and x_2. Let $s = y_1 y_2$ and $\gamma = \alpha_1^{old} y_1 + \alpha_2^{old} y_2$.

- If $s = 1$, then $\gamma = \pm(\alpha_1^{old} + \alpha_2^{old})$
  - If $\mid \gamma \mid > C$, then $\max \alpha_2 = C$, and $\min \alpha_2 = \gamma - C$.
  - If $\mid \gamma \mid < C$, then $\min \alpha_2 = 0$, and $\max \alpha_2 = \gamma$.
- If $s = -1$, then $\gamma = \pm(\alpha_1^{old} - \alpha_2^{old})$
  - If $\gamma > 0$, then $\min \alpha_2 = 0$ and $\max \alpha_2 = C - \gamma$.
  - If $\gamma < 0$, then $\min \alpha_2 = -\gamma$, and $\max \alpha_2 = C$.

Therefore, the Lower and Upper bounds on α_2 are as follows:

- If $s = 1$, then

$$L = \max(0, \gamma - C) \quad U = \min(\gamma, C) \tag{18}$$

- If $s = -1$, then

$$L = \max(0, \gamma) \quad U = \min(C, C + \gamma) \tag{19}$$

D. Partial Derivative of $\mathcal{L}$ w.r.t α_i

Recall the dual objective function in Equation (5):

$$\max \mathcal{L}(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j K(x_i, x_j) y_i y_j \tag{20}$$

If we make all α_is other than α_1 and α_2 constant (we can do this because they are constant when we take the $\frac{d\mathcal{L}}{d\alpha_2}$), (20) becomes:

$$\mathcal{L} = \alpha_1 + \alpha_2 - \frac{1}{2}(y_1 y_2 x_1^\top x_1 \alpha_1^2 + y_2 y_2 x_2^\top x_2 \alpha_2^2 + 2 y_1 y_2 x_1^\top x_2 \alpha_1 \alpha_2 + 2(\sum_{i=3}^{M} \alpha_i y_i x + i^\top)(t_1 x_1 \alpha_1 + y_2 x_2 \alpha_2)) \tag{21}$$

Let $K_{11} = x_1^\top x_1$, $K_{22} = x_2^\top x_2$, $K_{12} = x_1^\top x_2$, and

$$\begin{aligned} v_j &= \sum_{i=3}^{M} \alpha_i y_i x_i^\top x_j \\ &= x_j^\top \mathbf{w}^{old} - \alpha_1^{old} y_1 x_1^\top x_j - \alpha_2^{old} y_2 x_2^\top x_j \\ &= (x_j^\top \mathbf{w}^{old}) + b^{old} - \alpha_1^{old} y_1 x_1^\top x_j - \alpha_2^{old} y_2 x_2^\top x_j \\ &= u_j^{old} = b^{old} - \alpha_1^{old} y_1 x_1^\top x_j - \alpha_2^{old} y_2 x_2^\top x_j \end{aligned} \tag{22}$$

where $u_j^{old} = x_j^\top \mathbf{w}^{old} - b^{old}$ is the output of the decision function under old parameters.

$$\begin{aligned} \mathcal{L} &= \alpha_1 + \alpha_2 - \frac{1}{2}(K_{11}\alpha_1^2 + K_{22}\alpha_2^2 + 2sK_{12}\alpha_1\alpha_2 + 2y_1 v_1 \alpha_1 + 2y_2 v_2 \alpha_2) \\ &= \text{... skipping some steps ...} \\ &= \frac{1}{2}(2K_{12} - K_{11} - K_{22})\alpha_2^2 + (1 - s + sK_{11}\gamma - sK_{12}\gamma + y_2 v_1 - y_2 v_2)\alpha_2 \end{aligned} \tag{23}$$

After some steps, the coefficient of α_2 can be expressed as:

$$\alpha_2 = y_2(E_1^{old} - E_2^{old}) - \eta \alpha_2^{old} \tag{24}$$

where $\eta = K_{11} + K_{22} - 2K_{12}$. Plugging (24) back into (23), we get:

$$\mathcal{L} = \frac{1}{2}\eta\alpha_2^2 + (y_2(E_1^{old} - E_2^{old}) - \eta\alpha_2^{old})\alpha_2 \tag{25}$$

Finally, the first derivative is:

$$\frac{d\mathcal{L}}{d\alpha_2} = \eta\alpha_2 + (y_2(E_1^{old} - E_2^{old}) - \eta\alpha_2^{old}) \tag{26}$$