Sequential Minimal Optimization for Support Vector Machines

Optimization Methods Course Project Report

Mattia Marilli
Università degli studi di Firenze
mattia.marilli@edu.unifi.it

Abstract

This report presents a Python implementation of the Sequential Minimal Optimization (SMO) algorithm for training Support Vector Machines with linear and RBF kernels. The implementation uses the Most Violating Pair rule and incremental error updates to efficiently solve the dual problem. Experiments on LIBSVM datasets show that the custom solver achieves classification accuracies comparable to scikit-learn's `SVC`, while highlighting differences in computational performance due to low-level optimizations.

1. Introduction

Support Vector Machines (SVMs) are widely used supervised learning models for binary classification tasks. Their effectiveness comes from the use of kernel functions, which enable the separation of non-linearly separable datasets in higher-dimensional feature spaces. However, the training process of SVMs requires solving a quadratic optimization problem, which may become computationally demanding as the dataset size increases.

Decomposition methods that address the dual formulation of the SVM optimization problem represent a standard approach to efficiently solve such problems. The dual problem is a quadratic programming problem with simple box constraints $0 \leq \alpha_i \leq C$ and a linear equality constraint $\sum_i \alpha_i y_i = 0$, which couples all variables together. In practice this would require manipulating the dense kernel matrix Q, which becomes computationally and memory-wise infeasible for medium and large datasets. The Sequential Minimal Optimization (SMO) algorithm addresses this issue by iteratively optimizing small subsets of variables, typically two at a time, which can be solved analytically. This approach reduces the computational burden and allows dynamically retrieving only the necessary parts of the kernel matrix, making training feasible on larger datasets where standard quadratic programming solvers would be impractical.

The goal of this project is to implement the SMO algorithm for the training of kernel SVMs using the Radial Basis Function (RBF) kernel. The implementation, developed in Python, relies on the `numpy` library for efficient numerical computation. The algorithm employs the *Most Violating Pairs* working set selection rule, and dynamically retrieves the required columns of the kernel matrix during optimization, rather than precomputing it entirely.

To validate the implementation, several binary classification problems from the LIBSVM dataset repository are used for experimentation. The correctness of the developed SMO implementation is verified by comparing the classification accuracy of the obtained models with those trained using the `SVC` class from `scikit-learn`. Furthermore, the experiments investigate the impact of varying the regularization parameter C.

2. Theoretical Background

2.1. Support Vector Machines

Support Vector Machines (SVMs) are supervised learning models for binary classification and regression problems.

Given a training set $\{(x_i, y_i)\}_{i=1}^{l}$, with $x_i \in \mathbb{R}^n$ and $y_i \in \{-1, 1\}$, the goal of a linear SVM is to determine a hyperplane

$$H(w, b) = \{x \in \mathbb{R}^n : w^\top x + b = 0\}$$

that maximally separates the two classes. Among all separating hyperplanes satisfying

$$y_i(w^\top x_i + b) \geq 1, \quad \forall i,$$

the optimal separating hyperplane is the one that maximizes the margin, i.e., the minimal distance between the hyperplane and the closest training points. This leads to the con-

vex quadratic optimization problem:

$$\min_{w,b} \quad \frac{1}{2}\|w\|^2 \tag{1}$$

$$\text{s.t.} \quad y_i(w^\top x_i + b) \geq 1, \quad i = 1, \ldots, l. \tag{2}$$

Hard-Margin SVM. In the hard-margin formulation, misclassification is **infinitely penalized**. Geometrically, the margin has width $\frac{2}{\|w\|}$, and every feasible solution defines a hyperplane such that all points are correctly classified and lie **outside the margin**. Among all such hyperplanes, the SVM hard-margin problem selects the one that **maximizes the margin**, i.e., the distance between the closest points of the two classes:

$$\max_{w,b} \frac{2}{\|w\|} \quad \text{s.t.} \quad y_i(w^\top x_i + b) \geq 1, \; i = 1, \ldots, l.$$

Points that lie exactly on the margin boundaries (i.e., satisfy the constraints with equality) are called **support vectors**. These points are the only ones that influence the position of the optimal hyperplane.

Soft-Margin SVM. In practice, datasets may not be linearly separable or may contain outliers, which would make the hard-margin problem infeasible. To handle this, the constraints are **relaxed** by introducing slack variables $\xi_i \geq 0$, allowing some points to lie inside the margin or be misclassified. The optimization problem becomes:

$$\min_{w,b,\xi} \quad \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{l} \xi_i \tag{3}$$

$$\text{s.t.} \quad y_i(w^\top x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0, \quad i = 1, \ldots, l. \tag{4}$$

Here, $C > 0$ controls the trade-off between maximizing the margin and minimizing the classification errors.

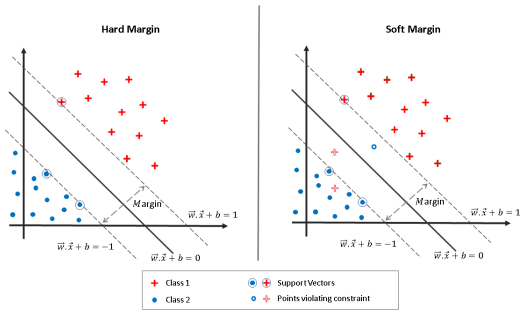

Figure 1. Illustration of the hard margin SVM compared to the soft margin SVM. Points inside the margin have $\xi_i > 0$.

Dual Problem. Using Wolfe's duality theory, the problem can be reformulated in its dual form:

$$\max_{\alpha} \quad \sum_{i=1}^{l} \alpha_i - \frac{1}{2}\sum_{i=1}^{l}\sum_{j=1}^{l} \alpha_i \alpha_j y_i y_j (x_i^\top x_j) \tag{5}$$

$$\text{s.t.} \quad 0 \leq \alpha_i \leq C, \quad \sum_{i=1}^{l} y_i \alpha_i = 0. \tag{6}$$

The optimal parameters are recovered as:

$$w^* = \sum_{i=1}^{l} \alpha_i^* y_i x_i, \quad b^* = y_k - w^{*\top} x_k \quad \text{for any } 0 < \alpha_k < C.$$

The resulting decision function is

$$f(x) = \text{sgn}\left(\sum_{i=1}^{l} \alpha_i^* y_i (x_i^\top x) + b^*\right).$$

Kernel Extension. In the nonlinear case, the input data are implicitly mapped into a higher-dimensional feature space $\mathcal{H}$ via a transformation $\phi : \mathbb{R}^n \to \mathcal{H}$. By introducing a kernel function $K(x_i, x_j) = \phi(x_i)^\top \phi(x_j)$, the dual problem becomes:

$$\max_{\alpha} \quad \sum_{i=1}^{l} \alpha_i - \frac{1}{2}\sum_{i=1}^{l}\sum_{j=1}^{l} \alpha_i \alpha_j y_i y_j K(x_i, x_j) \tag{7}$$

$$\text{s.t.} \quad 0 \leq \alpha_i \leq C, \quad \sum_{i=1}^{l} y_i \alpha_i = 0. \tag{8}$$

The decision function takes the form:

$$f(x) = \text{sgn}\left(\sum_{i=1}^{l} \alpha_i^* y_i K(x_i, x) + b^*\right).$$

Common kernel functions include the polynomial kernel $K(x, z) = (x^\top z + 1)^p$ and the RBF kernel $K(x, z) = \exp(-\|x - z\|^2/2\sigma^2)$, the latter being used in this project.

2.2. Sequential Minimal Optimization

The Sequential Minimal Optimization (SMO) algorithm, introduced by Platt (1998), is one of the most influential decomposition methods for solving the dual formulation of Support Vector Machines. It represents a special case of the general *decomposition approach*, where the original large-scale quadratic optimization problem is decomposed into a sequence of smaller subproblems that can be solved efficiently and exactly.

Decomposition Principle. In the SVM dual problem, all the Lagrange multipliers α_i are coupled by the linear equality constraint $\sum_i y_i \alpha_i = 0$, which prevents direct application of coordinate descent. Decomposition methods overcome this by partitioning the variable set $\alpha = (\alpha_1, \ldots, \alpha_l)$ into subsets, or *working sets*, which are optimized in turn while keeping the remaining variables fixed. The simplest possible subproblem arises when the working set size is two, since with two variables the equality constraint can be handled analytically.

Formally, at iteration k, the variable vector is split into two groups:

$$B^{(k)} = \{\alpha_i, \alpha_j\}, \quad N^{(k)} = \alpha \setminus B^{(k)}.$$

The subproblem over $B^{(k)}$ can be expressed as:

$$\min_{\alpha_i, \alpha_j} \quad \frac{1}{2} \begin{bmatrix} \alpha_i \\ \alpha_j \end{bmatrix}^\top \begin{bmatrix} K_{ii} & K_{ij} \\ K_{ij} & K_{jj} \end{bmatrix} \begin{bmatrix} \alpha_i \\ \alpha_j \end{bmatrix} + (y_i E_i + y_j E_j) \begin{bmatrix} \alpha_i \\ \alpha_j \end{bmatrix} \tag{9}$$

$$\text{s.t.} \quad 0 \le \alpha_i, \alpha_j \le C, \quad y_i \alpha_i + y_j \alpha_j = \zeta, \tag{10}$$

where $E_i = f(x_i) - y_i$ and ζ is a constant ensuring feasibility with respect to the coupling constraint.

Analytical Update. Because this subproblem involves only two variables, it admits a closed-form solution. Denoting $\eta = K_{ii} + K_{jj} - 2K_{ij}$, the optimal value of α_j is computed as:

$$\alpha_j^{\text{new}} = \alpha_j^{\text{old}} + \frac{y_j (E_i - E_j)}{\eta}.$$

The updated value is then projected onto its feasible interval $[L, H]$, where

$$L = \begin{cases} \max(0, \alpha_j^{\text{old}} + \alpha_i^{\text{old}} - C), & \text{if } y_i \neq y_j, \\ \max(0, \alpha_j^{\text{old}} - \alpha_i^{\text{old}}), & \text{if } y_i = y_j, \end{cases}$$

$$H = \begin{cases} \min(C, \alpha_j^{\text{old}} + \alpha_i^{\text{old}}), & \text{if } y_i \neq y_j, \\ \min(C, C + \alpha_j^{\text{old}} - \alpha_i^{\text{old}}), & \text{if } y_i = y_j. \end{cases}$$

Finally, α_i is updated to maintain the equality constraint:

$$\alpha_i^{\text{new}} = \alpha_i^{\text{old}} + y_i y_j (\alpha_j^{\text{old}} - \alpha_j^{\text{new}}).$$

The bias term is recomputed after each iteration using:

$$b^{\text{new}} = \begin{cases} b_1 = b^{\text{old}} - E_i - y_i (\alpha_i^{\text{new}} - \alpha_i^{\text{old}}) K_{ii} \\ \qquad - y_j (\alpha_j^{\text{new}} - \alpha_j^{\text{old}}) K_{ij}, \\ b_2 = b^{\text{old}} - E_j - y_i (\alpha_i^{\text{new}} - \alpha_i^{\text{old}}) K_{ij} \\ \qquad - y_j (\alpha_j^{\text{new}} - \alpha_j^{\text{old}}) K_{jj}, \end{cases} \tag{11}$$

The final bias value b^{new} is then selected according to the updated Lagrange multipliers. Specifically, if α_i^{new} lies strictly within $(0, C)$, then $b^{\text{new}} = b_1$; if α_j^{new} lies strictly within $(0, C)$, then $b^{\text{new}} = b_2$; and if both α_i^{new} and α_j^{new} are within $(0, C)$, the bias is set to the average

$$b^{\text{new}} = \frac{b_1 + b_2}{2}.$$

Working Set Selection. The efficiency of the SMO algorithm crucially depends on the strategy adopted to select the pair of variables (i, j) to be updated at each iteration. A naïve random or cyclic selection often leads to slow convergence, whereas more informed strategies can dramatically improve performance.

Among the most effective is the *Most Violating Pair* (MVP) rule, which originates from the theory of decomposition methods. The idea is to identify, at each iteration, the pair of multipliers (i, j) that maximally violates the Karush–Kuhn–Tucker (KKT) optimality conditions. The KKT conditions for the SVM dual problem can be expressed in terms of the gradient of the dual objective function:

$$\nabla_i f(\alpha) = y_i f(x_i) - 1,$$

where $f(x_i)$ is the current SVM decision function. For optimality, all variables must satisfy:

$$\begin{cases} \nabla_i f(\alpha) \ge 0, & \text{if } \alpha_i = 0, \\ \nabla_i f(\alpha) = 0, & \text{if } 0 < \alpha_i < C, \\ \nabla_i f(\alpha) \le 0, & \text{if } \alpha_i = C. \end{cases}$$

Let us define the index sets:

$$R = \{i \,|\, \alpha_i < C,\, y_i = 1\} \cup \{i \,|\, \alpha_i > 0,\, y_i = -1\},$$

$$S = \{i \,|\, \alpha_i < C,\, y_i = -1\} \cup \{i \,|\, \alpha_i > 0,\, y_i = 1\}.$$

Then, the MVP rule selects the pair:

$$(i^*, j^*) = \arg \max_{i \in R,\, j \in S} \left(-\frac{\nabla_i f(\alpha) - \nabla_j f(\alpha)}{y_i y_j (K_{ii} + K_{jj} - 2K_{ij})} \right),$$

that is, the two variables corresponding respectively to the maximum and minimum gradient components among feasible indices. Intuitively, the pair (i^*, j^*) represents the two points that contribute the most to the current violation of optimality — hence "most violating pair."

A key advantage of the MVP rule is its computational efficiency. The search for the most violating pair does not require examining all possible pairs, which would be $O(n^2)$. Instead, it can be performed in $O(n)$ time by scanning the gradient vector once to find:

$$i^* = \arg \max_{i \in R} \nabla_i f(\alpha), \qquad j^* = \arg \min_{j \in S} \nabla_j f(\alpha).$$

This simple but effective mechanism ensures that only the two most critical multipliers are selected at each step. In practice, this dramatically reduces computational overhead while preserving fast convergence, making the MVP rule the standard in modern SMO implementations.

Moreover, maintaining an updated gradient vector throughout the iterations allows the algorithm to efficiently monitor KKT violations and update gradient entries incrementally after each pair optimization.

Convergence Properties. From a theoretical standpoint, SMO is guaranteed to converge to the optimal solution of the SVM dual problem because:

- The objective function is convex and bounded below.
- Each iteration produces a feasible point with non-increasing objective value.
- The number of feasible extreme points is finite.

Hence, convergence to a KKT-satisfying solution is ensured, as formally established in the general framework of decomposition methods.

3. Implementation

3.1. Algorithm Overview

The `SVM` class implements the Sequential Minimal Optimization (SMO) algorithm for training Support Vector Machines with both linear and RBF kernels. It encapsulates all the model parameters, kernel definitions, and optimization routines, ensuring that training proceeds efficiently and consistently with the theoretical formulation of SVMs.

3.2. Code Structure

The implementation is organized as follows:

- **Initialization:** Hyperparameters such as the regularization constant C, kernel type, RBF gamma, and the maximum number of iterations are set. Model variables (α, b, support vectors, and the error cache) are initialized.
- **Training (`fit` method):** Executes the SMO loop that iteratively optimizes pairs of Lagrange multipliers to minimize the dual problem.
- **Prediction (`predict` method):** Uses the optimized support vectors to compute decision scores for new samples.

3.3. Initialization in `fit()`

Before starting the iterative SMO optimization, the `fit()` method performs several critical initializations:

- **Alpha Initialization:** All Lagrange multipliers α are set to zero. This corresponds to starting from a model that initially predicts all samples as belonging to the negative class:

```
self.alpha = np.zeros(N)
```

- **Support Vectors and Labels:** Initially, all training samples are considered candidate support vectors. The SMO algorithm will dynamically identify the true support vectors during training:

```
self.support_vectors = x_train
self.support_labels = y_train
```

- **Bias Initialization:** The bias term b is initialized to zero and will be updated after each pair-wise α update to maintain correct decision boundaries.
- **Error Cache Initialization:** The prediction errors for all training samples are precomputed using the initial α and b. This cache is crucial for efficient selection of the Most Violating Pair (MVP) and for incremental updates:

```
error_cache = (self.alpha * y_train) @ K
     + self.b - y_train
```

  Using a cached error vector avoids recomputing predictions from scratch at every iteration, significantly speeding up the SMO routine. A correct initialization ensures that the cached errors reflect the true model state given the current α and b, enabling consistent selection of Lagrange pairs. An arbitrary initialization (e.g., all -1) would introduce fictitious errors, slowing convergence.

3.4. Pair Selection and Threshold Handling

The algorithm selects pairs (α_i, α_j) based on the *Most Violating Pair (MVP)* rule. To handle numerical precision issues, very small tolerances ($1e-10$) are used:

```
L_indices = (alpha <= 1e-10)
U_indices = (alpha >= C - 1e-10)
Free_indices = (alpha > 1e-10) & (alpha < C
     - 1e-10)
```

Without this safeguard, the algorithm could mistakenly consider multipliers as strictly zero or C due to floating-point errors, causing the SMO routine to stall.

3.5. Stopping Criterion

The SMO training loop continues until either:

- The maximum number of iterations (`max_iter`) is reached.
- No pairs violate the Karush-Kuhn-Tucker (KKT) conditions beyond the specified tolerance (`kkt_thr`).

Formally, the stopping condition ensures that for all candidate multipliers:

$$|E_i - E_j| < \epsilon \quad \text{and} \quad 0 \leq \alpha_i, \alpha_j \leq C$$

If the MVP heuristic fails to find a valid pair, it returns $(-1, -1)$ and the loop terminates. This guarantees that the solution is within an acceptable margin of optimality.

3.6. Core SMO Updates

Once a pair of multipliers is selected, the algorithm:

- Computes the corresponding kernel columns to avoid redundant calculations.
- Calculates feasible bounds L, H and the second derivative term η.
- Updates α_i, α_j while respecting box constraints.
- Recomputes the bias b using the updated multipliers.
- Incrementally updates the error cache for all samples.

This vectorized approach ensures high computational efficiency, reducing Python-level loops and taking advantage of NumPy's low-level linear algebra optimizations.

3.7. Support Vector Extraction

After convergence, only non-zero α values are retained as support vectors:

```
sv_idx = (self.alpha <= 1e-10)
self.alpha = self.alpha[sv_idx]
self.support_vectors = self.support_vectors
    [sv_idx]
self.support_labels = self.support_labels[
    sv_idx]
```

This step reduces model size and speeds up predictions for new samples.

4. Experiments and Results

4.1. Experimental Setup and Expectations

All experiments were conducted using the six binary classification datasets `a1a`–`a6a` from the LIBSVM repository, which are derived from the Adult dataset for income prediction. Each dataset increases in size, from `a1a` (1,605 samples) to `a6a` (11,220 samples), while maintaining a consistent number of features (123).

For each dataset, the data were randomly shuffled and split into 80% training and 20% testing subsets. The experiments compared the proposed custom SMO-based SVM implementation (`SVM`) against the optimized `SVC` class from the `scikit-learn` library.

Both solvers used an RBF kernel with parameter $\gamma = 1/n_{features}$, and the regularization parameter C was varied in $\{0.1, 1, 10, 100\}$. The custom SVM was limited to 2000 optimization iterations with a KKT tolerance threshold of 10^{-2}.

We expected both implementations to achieve similar accuracies, as they minimize the same dual optimization objective. However, we anticipated that the `scikit-learn` implementation would be substantially faster due to its low-level optimizations (vectorized C/C++ backend, kernel caching, and efficient working set selection) compared to the Python version of the custom SMO solver.

4.2. Performance Results

Table 1 reports the training time and test accuracy obtained on all datasets. Both implementations achieve nearly identical accuracies for moderate values of C, confirming the correctness of the custom SMO implementation. However, the training times of the proposed solver are higher due to the absence of specialized computational optimizations.

The results demonstrate that the custom SMO SVM achieves decision boundaries consistent with those of `scikit-learn`'s `SVC`, confirming its algorithmic correctness. However, the training performance differs drastically due to the computational optimizations employed in `scikit-learn`, which is based on the `LIBSVM` library written in C++. The latter benefits from:

- **Kernel caching and precomputation:** partial kernel matrices are stored and reused across iterations, reducing redundant kernel evaluations;
- **Vectorized linear algebra routines:** operations are implemented in optimized BLAS/LAPACK calls, fully leveraging CPU vectorization;

- **Adaptive stopping criteria:** the solver dynamically adjusts precision thresholds depending on optimization progress.

In contrast, the custom SMO implementation performs its computations in Python using the `NumPy` library, which provides basic vectorization capabilities for array operations. This allows efficient element-wise arithmetic and matrix manipulation at the Python level, but it is still significantly slower than the low-level, highly optimized routines employed by `LIBSVM`. The latter executes most of its core computations—such as kernel evaluations, gradient updates, and dot products—in compiled C code, directly leveraging CPU cache locality and instruction-level parallelism.

Moreover, the custom solver recomputes kernel values on demand and applies a simple sequential working set update strategy without advanced heuristics or caching. As a result, although the algorithmic complexity of the implementation scales correctly with data size, its computational throughput remains much lower. This makes it considerably less efficient on large datasets or for high values of C, where the number of required optimization steps and kernel evaluations increases sharply.

Effect of the regularization parameter C. The parameter C controls the trade-off between maximizing the margin and minimizing classification errors. For small C, the model allows soft violations of the margin constraints, resulting in a smoother, more regularized decision boundary.

As C increases, the optimization problem approaches the *hard-margin* formulation, in which the SVM enforces strict separation of the training data and the optimization surface becomes less smooth. This makes the problem numerically harder to solve and more sensitive to convergence thresholds. Higher values of C typically lead to longer training times, as the solver must handle a more constrained and complex optimization problem to satisfy the stricter margin requirements. This means that increasing C improves training accuracy but at the cost of higher computational effort and a potentially higher risk of overfitting.

This phenomenon is evident in the results: for $C = 100$, the accuracy of the custom solver drops significantly across several datasets (e.g., `a6a`, where it falls to 67.4%). This behavior is not due to model bias, but rather to incomplete convergence: the maximum of 2000 iterations is insufficient for the optimizer to satisfy the KKT conditions below the defined tolerance. In contrast, `scikit-learn`'s solver dynamically adapts its iteration schedule and tolerance, continuing the optimization until convergence, thus achieving consistently higher accuracy for large C.

Dataset	C	SVM Time	SVM Acc	SVC Time	SVC Acc
a1a	0.1	0.30 s	75.08 %	0.03 s	75.08 %
a1a	1	0.40 s	80.69 %	0.03 s	80.69 %
a1a	10	1.10 s	83.18 %	0.03 s	83.18 %
a1a	100	2.40 s	81.00 %	0.04 s	81.93 %
a2a	0.1	0.82 s	75.50 %	0.06 s	75.50 %
a2a	1	0.90 s	82.34 %	0.06 s	82.34 %
a2a	10	2.30 s	82.12 %	0.06 s	82.12 %
a2a	100	3.80 s	71.74 %	0.08 s	82.56 %
a3a	0.1	1.65 s	76.92 %	0.12 s	76.92 %
a3a	1	1.85 s	83.36 %	0.11 s	83.36 %
a3a	10	4.90 s	84.14 %	0.10 s	84.14 %
a3a	100	5.25 s	79.75 %	0.13 s	85.09 %
a4a	0.1	3.60 s	76.38 %	0.25 s	76.38 %
a4a	1	3.70 s	84.85 %	0.23 s	84.85 %
a4a	10	6.80 s	85.27 %	0.21 s	85.06 %
a4a	100	7.60 s	73.04 %	0.29 s	84.43 %
a5a	0.1	6.40 s	75.06 %	0.44 s	75.06 %
a5a	1	6.50 s	84.80 %	0.41 s	84.72 %
a5a	10	9.80 s	84.41 %	0.39 s	84.18 %
a5a	100	10.90 s	75.99 %	0.51 s	84.49 %
a6a	0.1	17.50 s	83.11 %	1.12 s	82.09 %
a6a	1	17.90 s	84.27 %	1.15 s	84.14 %
a6a	10	18.20 s	85.07 %	1.22 s	85.03 %
a6a	100	19.10 s	67.38 %	1.85 s	84.71 %

Table 1. Comparison between custom SMO SVM and scikit-learn SVC on datasets a1a–a6a (train/test split).

5. Conclusion

This project presented a complete Python implementation of the Sequential Minimal Optimization (SMO) algorithm for training Support Vector Machines with linear and RBF kernels. Theoretical derivations and algorithmic details were revisited to highlight the fundamental mechanisms of decomposition-based quadratic programming.

Experimental results on benchmark datasets confirmed the correctness of the implementation, as classification accuracies were nearly identical to those obtained with the `scikit-learn SVC` solver. The custom solver's execution time was higher due to the lack of low-level optimizations. Despite this, the implementation successfully demonstrates how the SMO algorithm can be developed from first principles and extended with the Most Violating Pair rule for improved convergence.