![UTS logo] UNIVERSITY OF TECHNOLOGY SYDNEY

A Study on Support Vector Machines

Keao Chen, 25584234

Table of contents

List of Figures

1 Introduction

Support Vector Machine (SVM), invented by Vladimir Vapnik in 1979, is a kind of machine learning algorithm with solid theoretical foundation (Platt, 1998). The classical implementation of SVM is to find the optimal separating hyperplane by solving a convex quadratic programming problem. Furthermore, by solving its Lagrangian dual problem, kernel methods can be naturally introduced, enabling SVM to handle not only linear classification problems but also complex non-linear ones. This approach primarily uses optimization theory as a tool and can theoretically find an exact analytical solution using Quadratic Programming (QP) solvers. In practice, however, numerical approaches like Sequential Minimal Optimization (SMO) is often employed instead of general QP solvers for better efficiency.

In addition to this perspective, SVM can also be understood from the general machine learning perspective, that is, optimizing a loss function to fit the data. This method typically employs stochastic gradient descent as a tool, gradually approaching the minimum of the loss function by calculating its gradient. The advantage of this method is its simplicity in implementation and lower computational cost, but it usually can only find an approximate solution and cannot utilize kernel methods. The essence of these two methods is the same, just using different viewpoints.

This report will first introduce the linear classification problem and the perceptron algorithm, which is background knowledge and motivation for SVM. Then, the hard margin maximum margin classifier will be introduced in detail, including its primal and lagrange dual form, and an implementation using a QP solver will be provided. The soft margin maximum margin classifier will be presented as an extension to handle noisy and non-linearly separable data. Two implementations of the soft margin classifier will be provided: one using SMO and the other using stochastic gradient descent. These implementations will be compared on both artificial and real-world datasets. Finally, multi-class classification will be discussed, along with a reflection and conclusion of the report.

1.1 Links

- GitHub Repository
- Colab Notebook
- This report is written with the help of AI. Here is the first conversation and the second conversation.

2 Background

2.1 Linear Classification

SVM is a solution to the linear classification problem. Linear classification refers to the task of classifying data points into different categories based on a linear decision boundary. If the input space X is a subset of $\mathbb{R}^n$, where $n \geq 1$ is the number of features in the dataset, a linear decision boundary can be defined as a flat affine subspace of dimension $n - 1$. For example, in a 2-dimensional space, the decision boundary is a line, while in a 3-dimensional space, it is a plane. In higher-dimensional spaces, it is referred to as a hyperplane.

A hyperplane can be defined by the equation:

$$\mathbf{w} \cdot \mathbf{x} + b = 0$$

If a point $\mathbf{x}_0$ does not lie on the hyperplane, it must either satisfy

$$\mathbf{w} \cdot \mathbf{x}_0 + b > 0$$

or

$$\mathbf{w} \cdot \mathbf{x}_0 + b < 0$$

The side that the point lies on can be determined by its sign:

$$sign(\mathbf{w} \cdot \mathbf{x}_0 + b)$$

The learning task of is to find a hyperplane that separates the data points into two classes correctly, providing the dataset is linearly separable. Therefore, the hypothesis set H for linear classification can be defined as the set of all possible hyperplanes in the input space X.

2.2 Perceptron

Perhaps the simplest algorithm for finding a hyperplane is the Perceptron Learning Algorithm (PLA). The PLA iteratively updates the parameters $\mathbf{w}$ and b of the hyperplane based on the misclassified data points. It has been proven that as long as the data is linearly separable, the perceptron algorithm is guaranteed to converge.

Let the class label be denoted by y, where $y = 1$ for the positive class and $y = -1$ for the negative class. The product

$$y(\mathbf{w} \cdot \mathbf{x} + b)$$

indicates the correctness of the prediction: a positive value implies a correct prediction, while a negative value indicates a misclassification.

Perceptron learns from misclassifications. When a data point is correctly classified, the algorithm proceeds to the next data point without updating the model, whereas when a data point is misclassified, the model parameters are updated according to the following rules:

$$\mathbf{w} \leftarrow \mathbf{w} + lr \cdot y \cdot \mathbf{x}$$
$$b \leftarrow b + lr \cdot y$$

where lr is the learning rate that pre-defined as a hyperparameter.

This update mechanism implies that if the true label is positive and the data point is misclassified, the normal vector $\mathbf{w}$ of the hyperplane is adjusted in the direction of $\mathbf{x}$, thereby increasing the likelihood of correctly classifying $\mathbf{x}$ as a positive instance. Conversely, if the true label is negative, the vector $\mathbf{w}$ is updated in the opposite direction of $\mathbf{x}$, making it more likely that $\mathbf{x}$ will be classified as negative.

The figure below shows the learning process of a perceptron on a synthetic dataset. The green arrow represents the normal vector before the update, while the black arrow shows the normal vector after the update. For example, in Update 6, when a positive sample (the yellow point) lies on negative side of the hyperplane, the green arrow shifts closer to this data point after the update and thereby the updated hyperplane classifies this point correctly.

Figure 2.1: Perceptron Learning Process

It is clear that after six updates the hyperplane that the perceptron found has separated all data points correctly, but is it the best one? Actually, it is almost the worst, since it is very close to one side. Intuitively, the optimal hyperplane should look like the one below, which maximizes the distance to the nearest data points from both classes. Therefore, the model has a bigger cushion to tolerate noise and outliers. This is the main idea and motivation of SVMs.

Figure 2.2: The Optimal Hyperplane

Furthermore, the hyperplane found by the perceptron is not unique. Different initializations of the model parameters or variations in the order of data presentation can lead to different hyperplanes (Bishop, 2006). The following figure illustrates six different hyperplanes obtained by perceptron with different shuffles of the training data. Obviously, they vary significantly.

Figure 2.3: Different Final Hyperplane With Different Data Order

3 Linear Support Vector Machines

3.1 Maximum Margin Classifier With Hard Margin

Maximum Margin Classifier is a solution to the problems of perceptron mentioned above. Unlike Perceptron, Maximum Margin Classifier is able to find the optimal decision boundary by maximizing its margin. Margin is defined to be the smallest distance between the decision boundary and any of the samples (Bishop, 2006).

3.1.1 Functional Distance and Geometric Distance

The distance from a point $\mathbf{x}_0$ to a hyperplane h can be calculated by substituting $\mathbf{x}$ into the equation of the hyperplane:

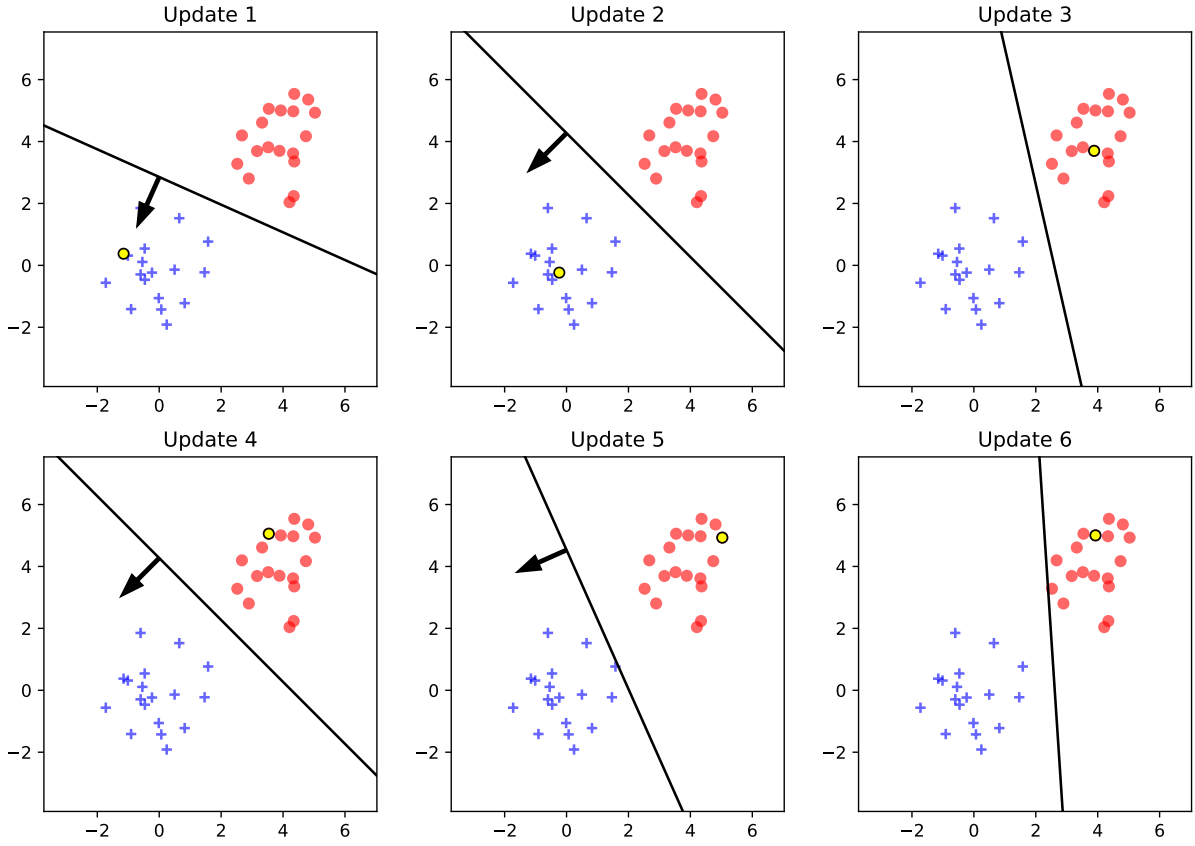$$distance(x_0, h) = |\mathbf{w} \cdot \mathbf{x}_0 + b|$$

if the hyperplane correctly classifies the point, the distance can also be expressed as:

$$distance(x_0, h) = y_0(\mathbf{w} \cdot \mathbf{x}_0 + b)$$

where y_0 is the class label of $\mathbf{x}_0$. This is called the functional distance.

In addition, this value can by manipulated arbitrarily. For example,

$$\mathbf{w} \cdot \mathbf{x} + b = 0 \quad \text{and} \quad 2\mathbf{w} \cdot \mathbf{x} + 2b = 0$$

represent the same hyperplane, but for the same input x, they yield different values. In fact, the latter is exactly twice the former. Therefore, we can pick a particular scalar to rescale the hyperplane, making it yield perferred output. We choose the scalar to be $1/\rho$, where

$$\rho = \min_{i=1,\cdots,n} y_i(\mathbf{w} \cdot \mathbf{x}_i + b)$$

so that:

$$\min_{i=1,\cdots,n} y_i(\frac{\mathbf{w}}{\rho} \cdot \mathbf{x}_i + \frac{b}{\rho}) = \frac{1}{\rho} \min_{i=1,\cdots,n} y_i(\mathbf{w} \cdot \mathbf{x}_i + b) = \frac{\rho}{\rho} = 1$$

Therefore, for any hyperplane, we can always rescale it to make the functional distance of the closest data points equal to 1.

Furthermore, the geometric distance from a point $\mathbf{x}_0$ to the hyperplane h_0 can be calculated as:

$$\frac{y_0(\mathbf{w} \cdot \mathbf{x}_0 + b)}{\|\mathbf{w}\|}$$

In other words, it is the functional distance divided by the norm of the normal vector $\mathbf{w}$. Geometric distance is also called Euclidean distance, which is invariant to the rescaling of the hyperplane.

The margin of a hyperplane is defined as the geometric distance from the hyperplane to the closest data points. The optimal hyperplane is the one that maximizes the margin. Since the hyperplane is determined by $\mathbf{w}$ and b, the optimization problem can be formulated as follows:

$$\arg\max_{\mathbf{w},b} \left\{ \frac{1}{\|\mathbf{w}\|} \min_{i=1,\cdots,n} [y_i(\mathbf{w} \cdot \mathbf{x}_i + b)] \right\}$$

Since we can always rescale the hyperplane so that

$$\min_{i=1,\cdots,n} y_i(\mathbf{w} \cdot \mathbf{x}_i + b) = 1$$

Thus, the optimization problem can be rewritten in a simpler form:

$$\begin{aligned} \arg\min_{\mathbf{w},b} \quad & \frac{1}{2}\|\mathbf{w}\|^2 \\ s.t. \quad & 1 - y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \le 0, i = 1, \cdots, n \end{aligned}$$

Here, the coefficient $\frac{1}{2}$ is introduced to simplify the result after differentiation.

3.1.2 Primal Problem

The lagrange function of this optimization problem is:

$$L(\mathbf{w}, b, \alpha) = \frac{1}{2}\|\mathbf{w}\|^2 + \sum_{i=1}^{n} \alpha_i \left[1 - y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \right]$$

where $\alpha_i \ge 0$ are the lagrange multipliers.

If we maximize L w.r.t. α:

$$\begin{aligned} \theta(\mathbf{w}, b) &= \max_{\alpha \ge 0} L(\mathbf{w}, b, \alpha) \\ &= \max_{\alpha \ge 0} \frac{1}{2}\|\mathbf{w}\|^2 + \sum_{i=1}^{n} \alpha_i [1 - y_i(\mathbf{w} \cdot \mathbf{x}_i + b)] \end{aligned}$$

It turns out that if there is an $\mathbf{x}_i$ that satisfies

$$1 - y_i(\mathbf{w} \cdot \mathbf{x}_i + b) > 0$$

which violates the constraint, L will go to $+\infty$, by making $\alpha = +\infty$, whereas if all $\mathbf{x}_i$ satisfy the constraint, the maximum value of L would be $\frac{1}{2}||\mathbf{w}||^2$ because we have to make all $\alpha_i = 0$ to avoid negative value.

Therefore, $\theta(\mathbf{w}, b)$ is actually defined by:

$$\theta(\mathbf{w}, b) = \begin{cases} \frac{1}{2}||\mathbf{w}||^2 & \text{if constraints are satisfied} \\ +\infty & \text{otherwise} \end{cases}$$

Minimize function $\theta(\mathbf{w}, b)$ w.r.t. $\mathbf{w}$ and b:

$$\min_{\mathbf{w},b} \theta(\mathbf{w}, b) = \min_{\mathbf{w},b} \max_{\alpha:\alpha_i \geq 0} L(\mathbf{w}, b, \alpha)$$

is equivalent to the original optimization problem and it's called the primal problem.

3.1.3 Dual Problem

If we instead minimize L w.r.t. $\mathbf{w}$ and b first and then maximize it w.r.t. α, it yields the dual problem:

$$\max_{\alpha:\alpha_i \geq 0} \min_{\mathbf{w},b} L(\mathbf{w}, b, \alpha)$$

There are several advantages of solving the dual problem instead of the primal one. First, the dual problem is always a convex optimization problem by nature (this is because the target function is linear w.r.t. α and the constraint sets are always convex sets), whereas the primal problem may not be convex, though in this case it is. Nevertheless, it's still a good practice to solve the dual problem, because it is usually simpler to solve than the primal one. It has been proven that for SVMs, solving the dual problem is equivalent to solving the primal one.

To solve the dual problem, first set the derivatives of L w.r.t. $\mathbf{w}$ and b to zero:

$$\nabla_{\mathbf{w}} L(\mathbf{w}, b, \alpha) = \mathbf{w} - \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i = 0$$

$$\nabla_{b} L(\mathbf{w}, b, \alpha) = -\sum_{i=1}^{n} \alpha_i y_i = 0$$

we obtain:

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

$$\sum_{i=1}^{n} \alpha_i y_i = 0 \tag{3.1}$$

substituting them back into L:

$$\min_{\mathbf{w},b} L(\mathbf{w}, b, \alpha) = -\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) + \sum_{i=1}^{n} \alpha_i$$

Having done that, maximizing w.r.t. α, dual optimization problem can then be summarized as:

$$
\begin{aligned}
\max_{\alpha} \quad & \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \\
s.t. \quad & \alpha_i \geq 0, i = 1, \cdots, n \\
& \sum_{i=1}^{n} \alpha_i y_i = 0
\end{aligned}
$$

3.1.4 Solving the Dual Problem Using a QP Solver

Equation 3.2 is a simpler quadratic programming problem, which can be solved by standard optimization tools. In this case, cvxopt is used and the stardard form of a quadratic programming problem that cvxopt uses is:

$$
\begin{aligned}
\min_{x} \quad & \frac{1}{2} x^\top P x + q^\top x \\
s.t. \quad & Gx \leq h, Ax = b
\end{aligned}
$$

To write the dual problem in that form, note that

$$\sum_{i,j} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$

can be expressed as a quadratic form: $\alpha^\top P \alpha$, where

$$P_{ij} = y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$

Let

$$q = -\mathbf{1}, G = -I, h = \mathbf{0}, A = \mathbf{y}, b = 0$$

where I is the $n \times n$ identity matrix. In this way, the dual problem can be written in the standard quadratic programming form:

$$\min_{\alpha} \quad \frac{1}{2}\alpha^{\top}P\alpha + (-\mathbf{1}^{\top}\alpha)$$
$$s.t. \quad -I\alpha \leq \mathbf{0}$$
$$\mathbf{y}^{\top}\alpha = 0$$

The corresponding Python code is as follows:

```python
P = matrix(np.outer(y, y) * (X @ X.T))
q = matrix(-np.ones(n_samples))
G = matrix(-np.eye(n_samples))
h = matrix(np.zeros(n_samples))
A = matrix(y.reshape(1, -1).astype(float))
b_eq = matrix(0.0)
```

After solving for the optimal solution α^*, we can obtain the optimal solution $\mathbf{w}^*$ and b^* of the primal problem according to Equation 3.1 and the Karush-Kuhn-Tucker (KKT) conditions, which are necessary and sufficient conditions for optimality (Platt, 1998):

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

$$b^* = y_j - \sum_{i=1}^{n} \alpha_i^* y_i (\mathbf{x}_i \cdot \mathbf{x}_j)$$

where $\mathbf{x}_j$ is any of the support vectors that satisfy $\alpha_j^* > 0$. Here, according to complementary slackness in KKT, $\alpha_j^* > 0$ indicates that $\mathbf{x}_j$ is a support vector, which lies on the margin boundary and satisfies:

$$y_i(\mathbf{w}^* \cdot \mathbf{x}_i + b^*) = 1$$

The other data points with $\alpha_i^* = 0$ do not contribute to the decision function, which reveals that the hyperplane depends only on the support vectors.

A better approach to compute b is to use all support vectors, as this makes it numerically more stable (Bishop, 2006):

$$b = \frac{1}{N} \sum_{i \in S} \left(y_i - \sum_{j \in S}^{n} \alpha_j^* y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \right)$$

where S is the set of indexes of all support vectors and N is the number of support vectors.

The Python code for computing $\mathbf{w}$ and b based on α is as follows:

```
w = np.sum((alphas * labels).reshape(-1, 1) * support_vectors,axis=0)
b = labels[0] - (support_vectors[0] @ w) # use one support vector
b = np.mean([y_i - np.sum(alphas * labels * (X @ x_i)) for i, x_i, y_i in
  ↪  zip(sv_indices, support_vectors, support_vector_labels)]) # use all
  ↪  support vectors


def predict(self, X):
    return np.sign(X @ w + b)
```

The decision function of the maximum margin classifier is:

$$f(\mathbf{x}) = \mathbf{w}^* \cdot \mathbf{x} + b^* = \sum_{i=1}^{n} \alpha_i^* y_i (\mathbf{x}_i \cdot \mathbf{x}) + b^*$$

or equivalently:

$$f(\mathbf{x}) = \sum_{i \in S} \alpha_i^* y_i (\mathbf{x}_i \cdot \mathbf{x}) + b^*$$

This is because only support vectors have non-zero α_i^*.

The figure below shows the decision boundary found by the maximum margin classifier on the same synthetic dataset as perceptron used. It is clear that the hyperplane maximizes the margin and classifies all data points correctly.
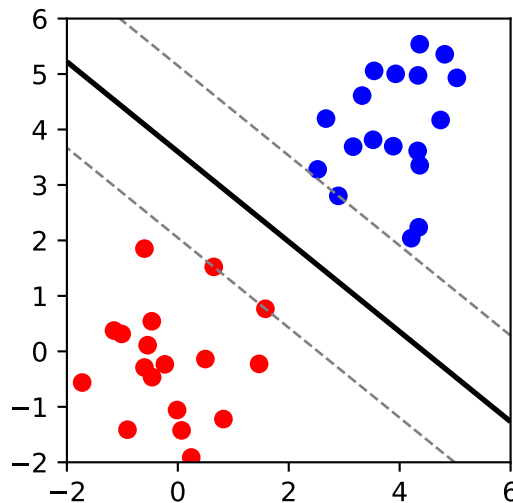

Figure 3.1: The Classification Boundary Found by MMC

3.2 Maximum Margin Classifier With Soft Margin

Even though a maximum margin classifier is able to find the optimal hyperplane for linearly separable data already, it is quite sensitive to noise and outliers. In addtion, real-world

data is often not linearly separable, in which case the maximum margin classifier fails to find a decision boundary.

A solution to these problems is to introduce slack variables $\xi = \{\xi_1, \cdots, \xi_n\}$ for each data point that allow some points to violate the margin or even the decision boundary. In this way, the model gives up classifying all training data correctly, but it allows some data points, which are probably noises or outliers, to be very close to the decision boundary or even on the wrong side of it.

This will increase the bias of the model but reduce its variance, leading to a worse performance on training set, but probably a better performance on testing data. In other words, it generalizes better. Moreover, to trade off the margin size and the number of misclassifications, a regularization parameter C is introduced. Accordingly, the optimization problem becomes:

$$\min_{\mathbf{w}, b, \xi} \quad \frac{1}{2}||\mathbf{w}||^2 + C \sum_{i=1}^{n} \xi_i$$
$$s.t. \quad y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i, i = 1, \cdots, n$$
$$\xi_i \geq 0, i = 1, \cdots, n$$

and its dual problem becomes:

$$\max_{\alpha} \quad \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$
$$s.t. \quad 0 \leq \alpha_i \leq C, i = 1, \cdots, n$$
$$\sum_{i=1}^{n} \alpha_i y_i = 0$$

It turns out that the only difference from the hard margin SVM is that α_i now has an upper bound C. In addition, the support vectors can be categorized into two types: the first type are those that lie exactly on the margin boundary, which satisfy $0 < \alpha_i < C$; the second type are those that violate the margin or even the decision boundary, which satisfy $\alpha_i = C$. The other data points with $\alpha_i = 0$ still do not contribute to the decision function.

3.2.1 Sequential Minimal Optimization

Sequential Minimal Optimization (SMO) is a popular algorithm for solving the dual problem of the soft margin SVM. The main idea behind SMO is coordinate descent optimized specifically for the SVM dual problem. Coordinate descent refers to optimizing one variable at a time while keeping the others fixed. Therefore, for convex optimization problems, the optimization problem is reduced to finding the minimum of a single variable convex function. However, in the case of SVM dual problem, there is an equality constraint:

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

for which if we optimize one α_i at a time, it is impossible to satisfy the constraint. To address this issue, SMO optimizes two variables at a time, which allows to adjust one variable while compensating with the other to satisfy the equality constraint.

The following conclusions are from Platt (1998).

3.2.1.1 Two-variable Optimization

According to Equation 3.1, we have:

$$f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b = \sum_{i=1}^{n} \alpha_i y_i (\mathbf{x}_i \cdot \mathbf{x}) + b$$

Let $E_i = f(\mathbf{x}_i) - y_i$ and $K_{ij} = \mathbf{x}_i \cdot \mathbf{x}_j$.

Assuming we select α_1 and α_2 to optimize, the unclipped solution is:

$$\alpha_2^{new} = \alpha_2 + \frac{y_2(E_1 - E_2)}{\eta}$$
$$\alpha_1^{new} = \alpha_1 + y_1 y_2 (\alpha_2 - \alpha_2^{new})$$

where $\eta = 2K_{12} - K_{11} - K_{22}$.

To ensure that the new α_1 and α_2 satisfy the constraints, we need to clip them within the feasible region. The clipping bounds depend on whether y_1 and y_2 are the same or different:

$$\begin{aligned} \text{if} \quad & y_1 \neq y_2: \\ & L = \max(0, \alpha_2 - \alpha_1) \\ & H = \min(C, C + \alpha_2 - \alpha_1) \end{aligned}$$

$$\begin{aligned} \text{if} \quad & y_1 = y_2: \\ & L = \max(0, \alpha_1 + \alpha_2 - C) \\ & H = \min(C, \alpha_1 + \alpha_2) \end{aligned}$$

The pseudo Python code below demonstrates the two-variable optimization process. `i1, i2` are the indexes of the two variables being optimized. The array `alphas` stores the values of all α_i, `y` stores the class labels, and `K` is the kernel matrix (Since I do not introduce kernel trick, it is just inner product). The function `compute_error` computes the error E_i, which is the difference between the predicted and true labels.

```python
if i1 == i2: return 0
alpha1, alpha2 = alphas[i1], alphas[i2]
y1, y2 = y[i1], y[i2]
E1, E2 = compute_error(i1), compute_error(i2)
s = y1 * y2
if y1 != y2:
```

```
    L = max(0, alpha2 - alpha1)
    H = min(self.C, self.C + alpha2 - alpha1)
else:
    L = max(0, alpha1 + alpha2 - self.C)
    H = min(self.C, alpha1 + alpha2)
if L == H: return 0
k11, k12, k22 = K[i1, i1], K[i1, i2], K[i2, i2]
eta = k11 + k22 - 2 * k12
if eta > 0:
    alpha2_new = alpha2 + y2 * (E1 - E2) / eta
    if alpha2_new >= H: alpha2_new = H
    elif alpha2_new <= L: alpha2_new = L
else:
    alpha2_new = L if abs(alpha2 - L) < abs(alpha2 - H) else H
alpha1_new = alpha1 + s * (alpha2 - alpha2_new)
```

3.2.1.2 Choosing Variables

In fact, as long as SMO always selects two variables to optimize, it will eventually converge (Platt, 1998). To speed up this process, SMO uses heuristics to choose variables. The selections of the first and the second variable are called the outer and inner loop, respectively.

According to Platt (1998), the algorithm for selecting two variables is as follows:

Outer Loop:

1. Iterates over all data points and examines whether they violate the KKT conditions. If a data point violates the KKT conditions, it is eligible for selection.
2. Iterate only over non-bound examples ($0 < \alpha < C$)
3. Repeat step 2 until all of the non-bound examples obey the KKT conditions within a specified tolerance, typically 10^{-3}.
4. Alternate between "entire training set pass" and "repeated non-bound subset passes" until entire training set satisfies KKT conditions.

Inner Loop:

1. Approximate step size by $|E_1 - E_2|$. If $E_i > 0$, choose example with minimum E_2, whereas if $E_1 < 0$: choose example with maximum E_2.

If the choice of the second variable makes no positive progress, then iterate through non-bound examples, and find one that makes positive progress; If still no progress, iterate through entire training set. If nothing works, skip this pair and return to the outer loop.

3.2.1.3 Computing b

1. If α_1 is non-bound: use b_1
2. If α_2 is non-bound: use b_2
3. If both non-bound: use either (they should be equal, or average for numerical stability)
4. If both bound: use $\frac{(b_1 + b_2)}{2}$

The figure below shows the decision boundary found by SMO SVM on a noisy dataset. It can be seen that the hyperplane allows some points to violate the margin and even be misclassified.
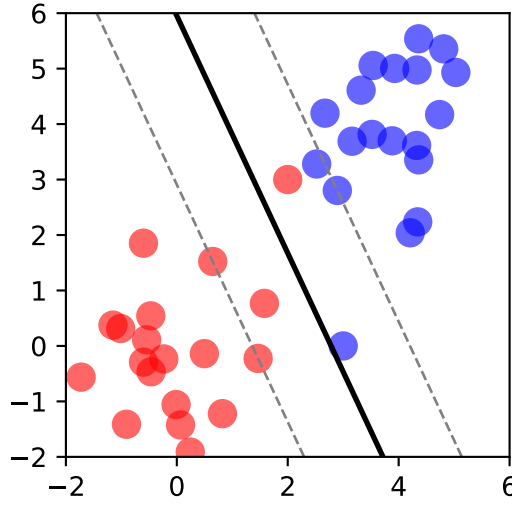

Figure 3.2: The Decision Boundary Found by SMO SVM on Noisy Data

3.2.2 Hinge Loss With L2 Regularization

In addition to solving the dual problem, the soft margin SVM can also be addressed from another perspective: optimizing the regularized hinge loss. It can be shown that the original optimization problem of the soft margin SVM is equivalent to the following optimization problem:

$$\min_{\mathbf{w}, b} \quad \lambda ||\mathbf{w}||^2 + \sum_{i=1}^{n} \max(0, 1 - y_i(\mathbf{w} \cdot \mathbf{x}_i + b))$$

Here, the first term is the regularization term, and λ is the regularization parameter, which controls the trade-off between the two terms and is inversely proportional to C above. When λ approaches 0, the penalty from regularization decreases, and the model tends to classify the data correctly. When $\lambda = 0$, the model essentially becomes the hard margin SVM.

In this case, if the dataset is linearly separable, the model can find a hyperplane that perfectly classifies the data, but it will be extremely sensitive to noise and outliers, leading to poor generalization. To balance perfect classification with generalization ability, λ

is usually set to a value greater than 0. In this way, while the model pursues perfect classification, it also minimizes the norm of the weights as much as possible, thereby enlarging the margin (recall that the margin is the inverse of $||\mathbf{w}||$), and improving generalization.

From the perspective of model complexity, regularization constrains the parameter space by eliminating large values of weights, thus limiting the number of hypotheses in the hypothesis set $\mathbb{H}$, reducing model complexity, and lowering the risk of overfitting. This reflects the idea of the bias-variance trade-off.

The second term is the hinge loss, which is named after its shape, as shown in the figure below. When the functional margin is greater than or equal to 1

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

it means the point lies outside the margin or exactly on it, so no penalty is applied. When the margin is less than 1, the penalty equals the distance by which the point falls short of the margin. The closer the point is to the hyperplane, the larger the violation of the margin, and thus the heavier the penalty.
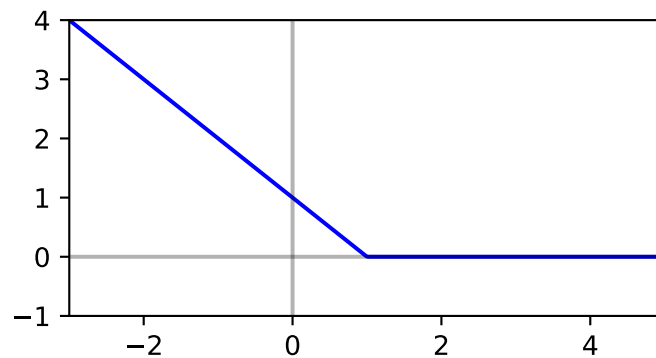

Figure 3.3: The Illustration of Hinge Loss

The following code demonstrates how to compute hinge loss with L2 regularization in Python. First, the distance of each point to the hyperplane is calculated. Points with distance greater than 1 are set to 0, and then the average is taken to obtain the hinge loss. The regularization term is computed according to the formula, and finally the two terms are added together to obtain the final loss value.

```python
def loss(X, y, w, b, lambda_):
    distances = y * (X @ w + b)
    hinge_loss = np.mean(np.maximum(0, 1 - distances))
    reg_loss = lambda_ * (w @ w)
    return hinge_loss + reg_loss
```

3.2.3 Mini-batch SGD

Since the hinge loss is almost everywhere differentiable and is a convex function, in addition to analytical methods, one can also use stochastic gradient descent (SGD) to find the minimum. SGD refers to updating the parameters in the direction determined by the gradient of the function. At non-differentiable points, a sub-gradient is chosen to replace the gradient. For example, for the hinge loss function, at non-differentiable points, any element from

$$\{-\alpha y\mathbf{x} \mid \alpha \in [0, 1]\}$$

can be chosen as the sub-gradient. Since the gradient always points in the direction of steepest ascent, its negative points to the steepest descent direction. Because the objective function is convex, descending along the steepest slope at each step will gradually lead us closer to the minimum of the function, i.e., the point where the hinge loss is minimized.

The stochastic nature of SGD comes from computing the gradient on a small randomly sampled batch of data from the dataset each time. This avoids the expensive computation cost of calculating gradients over the entire dataset, while also avoiding the instability of using just a single data point, thus achieving a balance.

3.2.3.1 Gradient Calculation

To compute the gradient of the hinge loss, first write hinge loss in a piecewise form:

$$L_i(\mathbf{w}, b) = \begin{cases} 1 - y_i(\mathbf{w} \cdot \mathbf{x}_i + b) + \lambda||\mathbf{w}||^2 & \text{if } 1 - y_i(\mathbf{w} \cdot \mathbf{x}_i + b) > 0 \\ \lambda||\mathbf{w}||^2 & \text{otherwise} \end{cases}$$

This function is non-differentiable at the piecewise boundary, but differentiable elsewhere. When $1 - y_i(\mathbf{w} \cdot \mathbf{x}_i + b) > 0$,

$$\nabla_{\mathbf{w}} L_i(\mathbf{w}, b) = -y_i\mathbf{x}_i + 2\lambda\mathbf{w}$$

$$\nabla_b L_i(\mathbf{w}, b) = -y_i$$

When $1 - y_i(\mathbf{w} \cdot \mathbf{x}_i + b) < 0$, the hinge loss is 0, and only the regularization term is considered:

$$\nabla_{\mathbf{w}} L_i(\mathbf{w}, b) = 2\lambda\mathbf{w}$$

$$\nabla_b L_i(\mathbf{w}, b) = 0$$

The following code demonstrates how to compute the gradient. First, the distance of each point to the hyperplane is calculated. Then, depending on whether the distance is greater than 1, the gradient is computed differently. For points with distance less

than 1, the gradient is the sum of the hinge loss gradient and the regularization term gradient. For points with distance greater than 1, the hinge loss gradient is 0, and only the regularization term gradient is computed. Here, for points with distance exactly equal to 1, the sub-gradient is chosen to be 0.

```python
def calc_gradient(X, y, w, b, lambda_):
    distances = y * (X @ w + b)
    mask = distances < 1
    dw = 2 * lambda_ * w
    if np.any(mask):
        dw += -np.mean((y[mask]).reshape(-1, 1) * X[mask], axis=0)
    db = 0.0
    if np.any(mask):
        db = -np.mean(y[mask])
    return dw, db
```

3.2.3.2 Learning Rate

The gradient determines the update direction at each step, while the learning rate controls the step size. If the learning rate is too large, the algorithm may overshoot the minimum, or even cause the loss to increase. If the learning rate is too small, convergence will be very slow, and in the case of non-convex optimization, it may increase the risk of getting stuck in a local optimum. Therefore, choosing an appropriate learning rate is also crucial. The two cases described above are illustrated in the figure below.
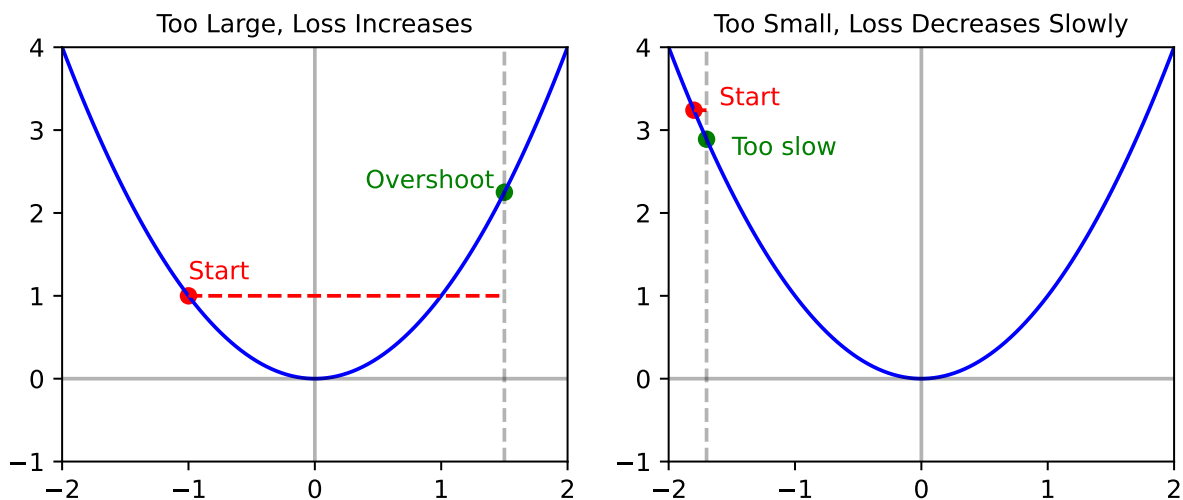

Figure 3.4: Effects of Different Learning Rates on Convergence

Apart from using a fixed learning rate, a dynamically adjusted learning rate can be used. For example, the learning rate can gradually decrease as the number of iterations increases. The advantage of this approach is that in the initial stage, a larger learning rate allows the algorithm to quickly approach the optimal solution; while in the later stage, a smaller learning rate helps avoid overshooting the optimum and allows more stable convergence.

3.2.3.3 Update Strategy

Having calculated the gradient and chosen a appropriate learning rate, the parameters can be updated as follows:

$$\mathbf{w} \leftarrow \mathbf{w} - lr\nabla_{\mathbf{w}}L_i(\mathbf{w}, b)$$
$$b \leftarrow b - lr\nabla_b L_i(\mathbf{w}, b)$$

where lr is the learning rate.

The complete learning process is as follows:

1. First, randomly initialize the model parameters $\mathbf{w}$ and b.
2. Then, in each iteration, split all the data into multiple mini-batches.
3. For each mini-batch, compute the gradient for that batch and update the model parameters.
4. Repeat steps 2 and 3 until the preset number of iterations is reached.

The code is shown below:

```python
w = np.random.normal(0, 0.01, n_features)
b = 0.0

for epoch in range(n_epochs):
    batches = get_batches(X, y, batch_size)
    for batch in batches:
        dw, db = gradient(X_batch, y_batch)
        w -= lr * dw
        b -= lr * db
```

3.2.4 Comparison With Hard Margin SVM

As shown in the figure below, after introducing two outliers into the original dataset, the decision boundary of the hard margin classifier is largely shifted due to the influence of the noise, producing a hyperplane with very narrow margins, whereas the soft margin classifier is less affected by outliers, because it allows some points to violate the margin or even be misclassified.
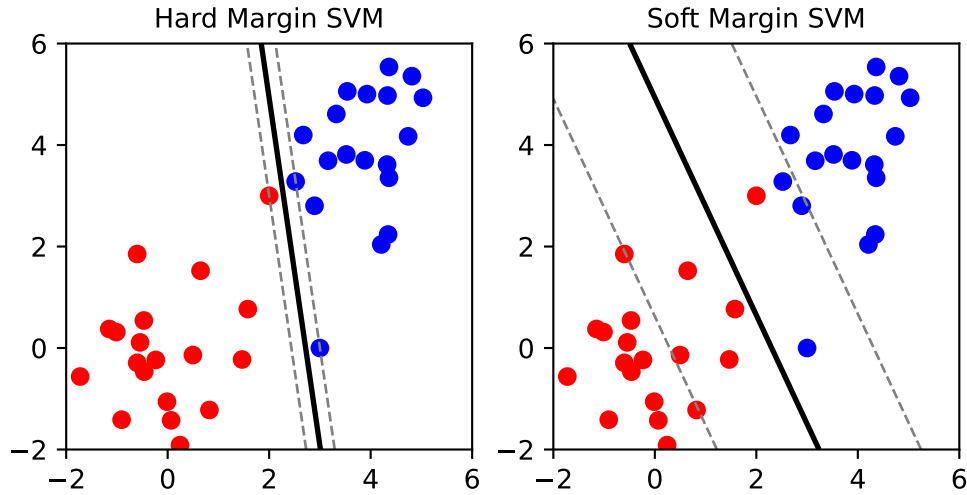
Figure 3.5: Comparison Between Hard Margin SVM and Soft Margin SVM on Noisy Data

The figure below shows different decision boundaries obtained by the soft margin SVM with different values of the regularization parameter λ. A smaller λ (equivalent to a larger C) means a smaller penalty for misclassification, leading to a narrower margin and a more complex model that fits the training data better but may overfit. Conversely, a larger λ (equivalent to a smaller C) results in a wider margin and a simpler model that may underfit the training data but generalizes better.
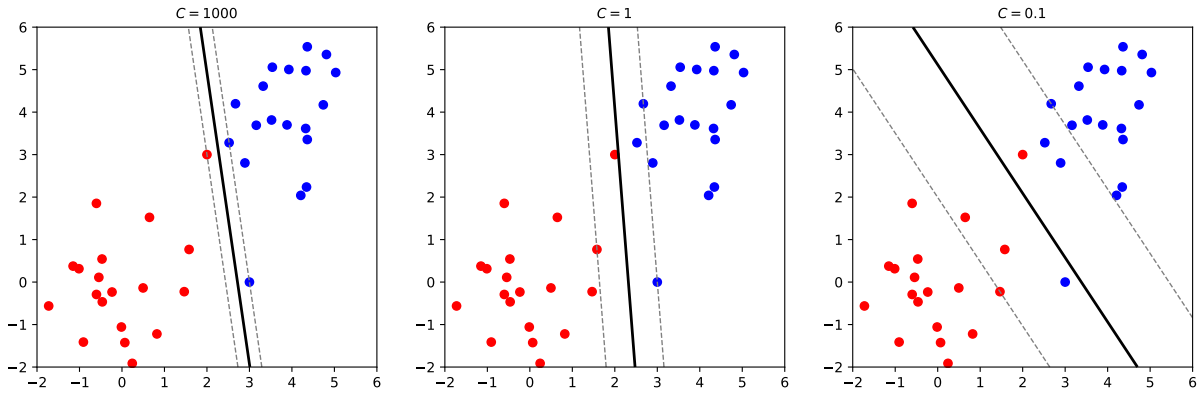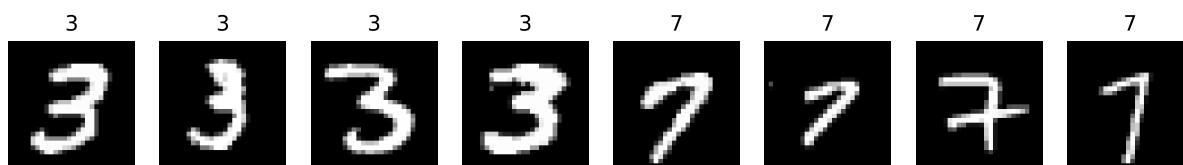

Figure 3.6: Effect of Regularization Parameter on Decision Boundary of Soft Margin SVM

4 Experiments On MNIST Dataset

4.1 Dataset Description

The input dataset X consists of 28×28 pixel grayscale images of handwritten digits. The pixel values range from 0 to 255 where 0 represents pure white and 255 represents pure black. The pixel values are normalized to the range $[0, 1]$ by dividing by 255. The training set contains 60,000 images and the test set contains 10,000 images. In this experiment, only the digits 3 and 7 are used to form a binary classification problem. Each image is flattened into a 784-dimensional vector. The output is a binary label indicating whether the digit is a 3 or a 7.

The figure below shows some samples of the digits 3 and 7 from the training set.

4.2 Training with SGD SVM

The code below trains a soft margin SGD based SVM.

```
sgd_svm = SGDSVM(epochs=200, lr=0.0001, lambda_=0.001)
sgd_svm.fit(X, y)
```

The figure below shows the training loss curve. It can be seen that the loss decreases rapidly in the initial stage and then gradually stabilizes.
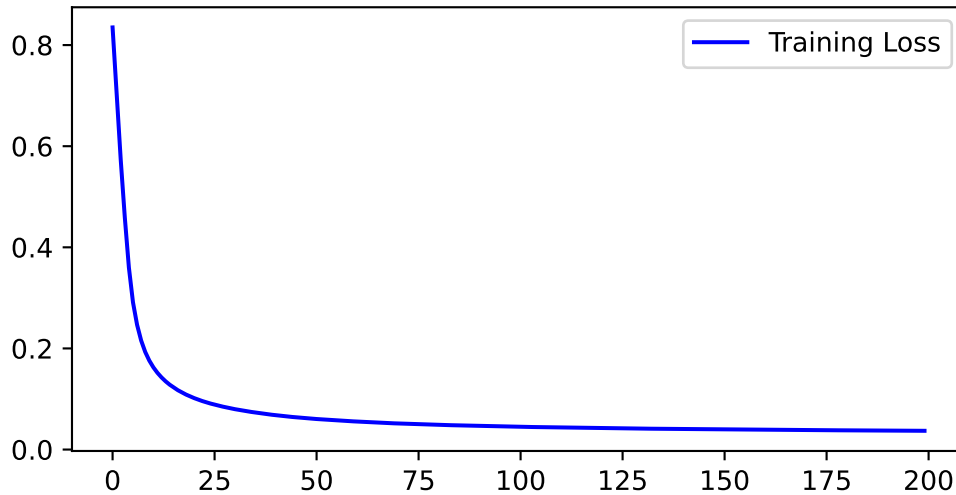
Figure 4.1: Training Loss Curve of SGD SVM on MNIST

The code below calculates the accuracy on the validation set.

```
print(
    f"The accuracy on the validation set is: {(sgd_svm.predict(X_valid)
    ↪   == y_valid).mean()*100:.2f}%"
)
```

The accuracy on the validation set is: 98.28%

The figure below visualizes the learned weights. The red areas indicate positive weights, while the blue areas indicate negative weights. It can be observed that the model focuses on the regions that are most discriminative between the digits 3 and 7.
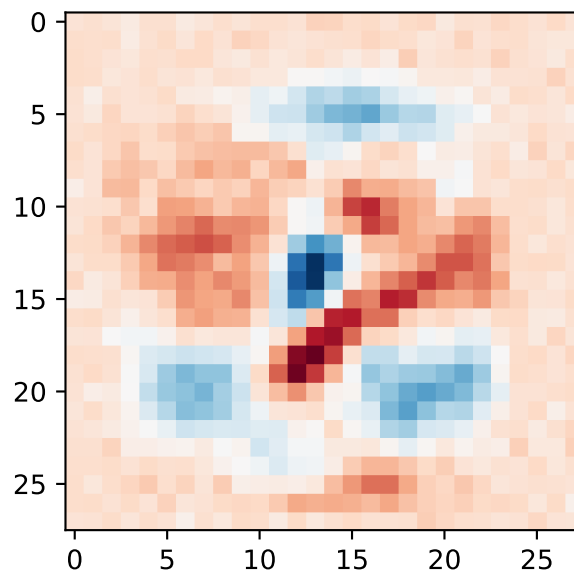

Figure 4.2: Visualization of Learned Weights by SGD SVM

The figure below shows the confusion matrix of the predictions on the validation set. It can be seen that most samples are correctly classified, with only a few misclassifications.
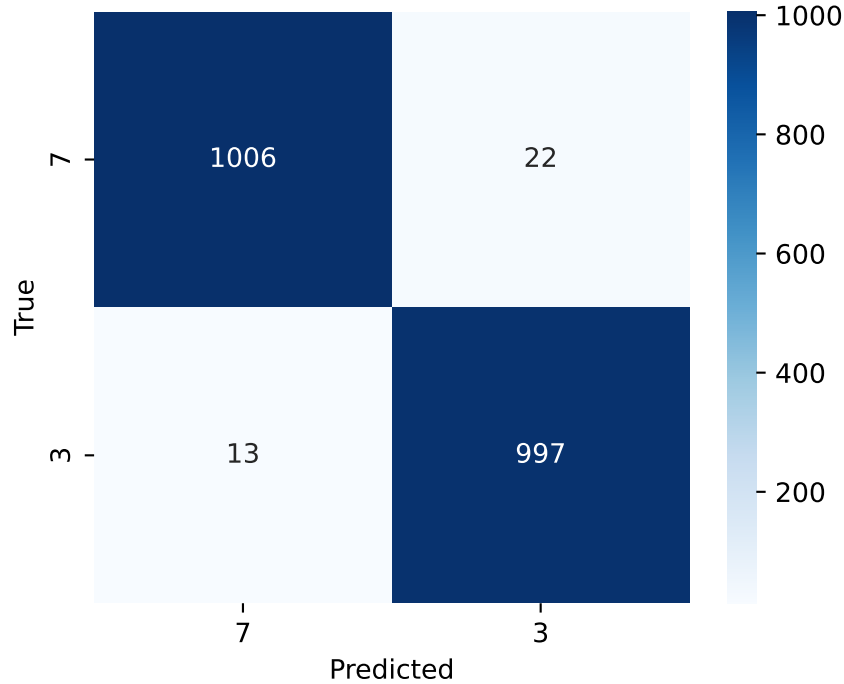

Figure 4.3: Confusion Matrix and Classification Report of SGD SVM on MNIST

The table below shows the precision, recall, and F1-score for each class.

```
print(classification_report(y_valid, y_pred, target_names=["7", "3"]))
```

```
              precision    recall  f1-score   support

           7       0.99      0.98      0.98      1028
           3       0.98      0.99      0.98      1010

    accuracy                           0.98      2038
   macro avg       0.98      0.98      0.98      2038
weighted avg       0.98      0.98      0.98      2038
```

4.3 Comparison with SMO Algorithm

SMO algorithm is expected to be able to find a more accurate solution to the soft margin SVM problem because it directly solves the dual problem. The code below randomly selects 500 samples from the training set to form a smaller training set and trains an SMO SVM on it.

```
n_train_samples = 500
indices = np.random.choice(len(X), n_train_samples, replace=False)
X_train_small = X[indices]
y_train_small = y[indices]

smo_mnist = SMOSVM(C=100, tol=1e-3, max_iter=50)
smo_mnist.fit(X_train_small, y_train_small)
```

The code below evaluates the SMO SVM on the validation set and compares it with the SGD SVM trained on the same small subset. SMO SVM and SGD SVM have almost the same performance.

```
SMO SVM accuracy on validation set: 97.15%
SGD SVM accuracy on validation set: 97.25%
```
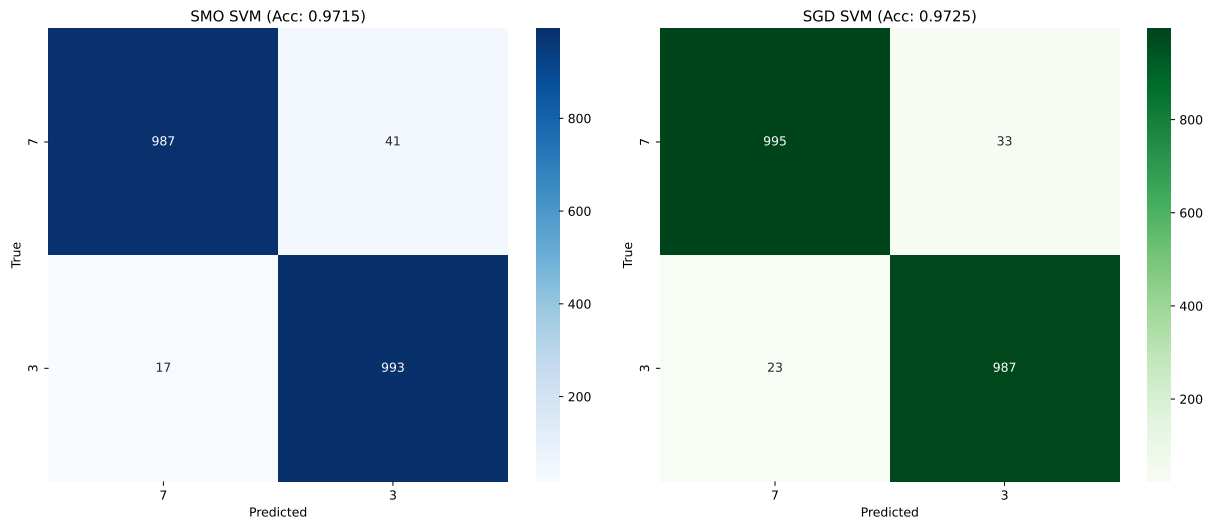

Figure 4.4: Confusion Matrices of SMO SVM and SGD SVM on MNIST Validation Set

In conclusion, the comparison shows that both SMO and SGD based approaches can effectively solve the soft margin SVM problem.

5 Multi-class SVM

SVM is not able to handle multi-class classification problems directly due to its binary nature. However, there are two common strategies to extend SVM to multi-class problems: one-versus-the-rest (OvR) and one-versus-one (OvO), both of which involve training multiple binary classifiers.

5.1 One Versus The Rest

One versus rest (OvR) refers to training k binary classifiers for a k-class classification problem. Each classifier is trained to distinguish one class from all other classes. During training, the kth classifier is trained using the samples from the kth class as positive examples and samples from all other classes as negative examples. During prediction, the classifier that outputs the highest decision function value determines the predicted class. One disadvantage of this approach is that one data point may be classified into multiple classes. Another disadvantage is that the classifiers are very likely to be imbalanced, as the positive data from one class while the negative data from all other classes.

5.2 One Versus One

One versus one (OvO) refers to training $\binom{k}{2}$ binary classifiers for a k-class classification problem. Each classifier is trained to distinguish between a pair of classes. The advantage of this approach is that each classifier is trained on a balanced dataset, as each classifier only uses data from two classes. During prediction, each classifier votes for one of the two classes it was trained on, and the class with the most votes is chosen as the final prediction. One disadvantage of this approach is that it requires training a large number of classifiers, which can be computationally expensive for problems with many classes. Another disadvantage is that the classifiers may be inconsistent, as different classifiers may produce conflicting predictions.

Bishop (2006) lists several methods for addressing aforementioned issues like Direct Acyclic Graph SVM (DAGSVM) and concludes that in practice, OvR is widely used. Therefore, in this section, I will implement the OvR strategy using the SGD SVM and test it on a three-class classification problem using digits 3, 5, and 7 from the MNIST dataset.

Figure 5.1: Some Samples of Digits 3, 5, and 7 from the MNIST Dataset

The code below implements the OvR strategy using the previously defined `SGDSVM` class. The `OvR_SGD_SVM` class contains a list of `SGDSVM` models, one for each class. During training, each model is trained to distinguish one class from all other classes. During prediction, the model that outputs the highest decision function value determines the predicted class.

```python
class OvR_SGD_SVM:
    def __init__(self, n_classes, lambda_=0.01, lr=0.01, epochs=10,
    ↪  batch_size=64):
        self.n_classes = n_classes
        self.models = [
            SGDSVM(lambda_=lambda_, lr=lr, epochs=epochs,
    ↪  batch_size=batch_size)
            for _ in range(n_classes)
        ]

    def fit(self, X, y):
        for i in range(self.n_classes):
            y_binary = np.where(y == i, 1, -1)
            self.models[i].fit(X, y_binary)

    def predict(self, X):
        decision_values = np.array([model.predict(X) for model in
    ↪  self.models])
        return np.argmax(decision_values, axis=0)
```

The code below trains the OvR SGD SVM on the three-class dataset and evaluates it on the validation set in terms of accuracy.

```python
ovr_svm = OvR_SGD_SVM(n_classes=3, epochs=200, lr=0.0001, lambda_=0.001)
ovr_svm.fit(X, y)
ovr_svm_pred = ovr_svm.predict(X_valid)
print(f"The accuracy on validation set is {(ovr_svm_pred ==
↪  y_valid).mean()*100:.2f}%")
```

```
The accuracy on validation set is 94.81%
```

The figure below shows the confusion matrix of the predictions on the validation set. It can be seen that the model often confuses the digits 3 and 5, while the digit 7 is less likely to be misclassified. This makes sense because the digits 3 and 5 look more similar to each other compared to the digit 7.
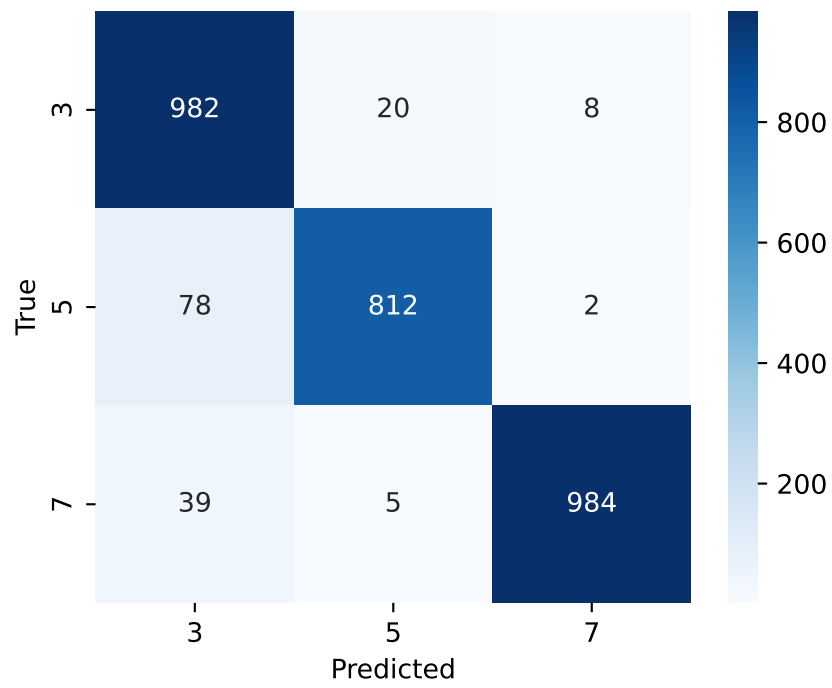
Figure 5.2: Confusion Matrix of OvR SGD SVM on MNIST Validation Set

6 Reflection and Conclusion

The motivation for me choosing the SVM as the topic of this report is that unlike other gradient based models that highly rely on numerical optimization and experience in tuning hyperparameters, SVM has a solid theoretical foundation and can be solved analytically. This makes it a little bit harder to understand than other models, but also more interesting. It intimitated me in the first semester when I learned it in the foundimental data analytics course. The concepts and terminologies from optimization theory like convex optimization, lagrange duality, and so forth stopped me from going through the underlying principles of SVM. Therefore, I decided to take a deep dive into SVM in this report.

Along the way, I learned basic knowledge about convex optimization and reviewed some rusty concepts from linear algebra. The biggest challenge for me is to understand the mathematical derivations from the primal problem to the dual problem, because it involves a lot of summation operations. Another challenge is to derive and implement the SMO algorithm, which also involves a lot of calculations, and the implementation of SMO requires handling many details, such as dealing with edge cases. In this process, Bishop (2006), Platt (1998) and James et al. (2023) are very helpful to me. The conclusions and inferences used in this report are mainly based on these references. AI is also employed to answer some specific questions, assist in writing some code and translation. The link of the conversation with AI can be found in appendix.

The limitation of this report is that it only covers the linear SVM, while in practice, kernel SVM is commonly used. It is also worth mentioning that the implementation of SMO in this report is not optimized for efficiency. During training, it takes forever to converge on the whole MNIST dataset. In practice, libraries like LIBSVM and scikit-learn have highly optimized implementations of SVM that can handle large datasets efficiently.

In conclusion, this report provides a comprehensive overview of the principles and implementation of SVM. It covers both hard margin and soft margin SVM, as well as multi-class classification using the OvR strategy. Through theoretical analysis and practical experiments, it demonstrates the effectiveness of SVM in handling classification tasks.

7 References

Bishop, C. M. (2006). *Pattern recognition and machine learning.* Springer.

James, G., Witten, D., Hastie, T., Tibshirani, R., & Taylor, J. (2023). Support Vector Machines. In *An Introduction to Statistical Learning* (pp. 367–398). Springer International Publishing. https://doi.org/10.1007/978-3-031-38747-0_9

Platt, J. (1998). *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines* (MSR-TR-98-14). Microsoft. https://www.microsoft.com/en-us/research/publication/sequential-minimal-optimization-a-fast-algorithm-for-training-support-vector-machines/

8 Appendix

1. The conversation with AI can be found at here and here.
2. The jupyter notebook for the visualizations and the implementations of the models in this report can be found at here.
3. This report is written in Quarto markdown. The source code of this report can be found at here.
4. This report uses `quarto titlepages` theme, which can be found here.
5. The picture on the cover page is from here.