Hard Margin SVM: Complete Mathematical Derivation

I. Problem Setup: Binary Classification

The Scenario: We want to classify data into two categories. For example, predicting whether a student gets into tech school ($y = +1$) or does not ($y = -1$).

The Data:

- n total students
- For each student i, we have:
  - A feature vector $\mathbf{x}_i$ (e.g., GPA, SAT scores, coding projects)
  - A label $\mathbf{y}_i \in \{-1, +1\}$

The Goal: Build an SVM model that creates a **maximum margin classifier**—a decision boundary with the largest possible "safety zone" between the two classes. This breathing room helps the model generalize better to new, unseen data.

II. Defining the Hyperplanes

To find the best separating boundary, we define three parallel hyperplanes:

1. Decision Boundary (The Separator):

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

- **w** is the weight vector (normal vector perpendicular to the hyperplane)
- **b** is the bias term (shifts the plane)
- This line separates the two classes

2. Positive Margin Boundary (Upper Boundary):

$\mathbf{w \cdot x - b = +1}$

- All points of class +1 should lie on or above this line
- The closest +1 points touch this boundary (these become support vectors)

3. Negative Margin Boundary (Lower Boundary):

$\mathbf{w \cdot x - b = -1}$

- All points of class -1 should lie on or below this line

- The closest -1 points touch this boundary (these become support vectors)

Key Geometric Insight: The vector **w** points in the direction perpendicular to all three hyperplanes. Think of w as an arrow pointing "upward" from the negative class toward the positive class.

III. Calculating the Margin Width (Step-by-Step)

Now comes the key question: **What is the distance between the two margin boundaries?**

Step 1: Start at a point on the decision boundary

Let $\mathbf{x}_0$ be any point on the decision boundary, so: $\mathbf{w}\cdot\mathbf{x}_0 - \mathbf{b} = \mathbf{0}$

Step 2: Walk perpendicular toward the positive margin

We want to walk from x_0 in the direction of **w** (the perpendicular direction) until we reach the positive margin boundary.

Let's walk a distance of **k** units. Since we need a unit direction vector, we walk in the direction $\mathbf{w}/||\mathbf{w}||$ (the normalized version of w).

Our new position after walking k units: $\mathbf{x}_1 = \mathbf{x}_0 + \mathbf{k}(\mathbf{w}/||\mathbf{w}||)$

Step 3: This new point must satisfy the positive margin equation

Since x_1 is on the positive margin boundary: $\mathbf{w}\cdot\mathbf{x}_1 - \mathbf{b} = \mathbf{1}$

Substitute x_1: $\mathbf{w}\cdot(\mathbf{x}_0 + \mathbf{k}(\mathbf{w}/||\mathbf{w}||)) - \mathbf{b} = \mathbf{1}$

Expand using the dot product: $\mathbf{w}\cdot\mathbf{x}_0 + \mathbf{k}(\mathbf{w}\cdot\mathbf{w})/||\mathbf{w}|| - \mathbf{b} = \mathbf{1}$

Step 4: Simplify using what we know

We know that $\mathbf{w}\cdot\mathbf{x}_0 - \mathbf{b} = \mathbf{0}$ (since x_0 is on the decision boundary), so: $\mathbf{0} + \mathbf{k}(\mathbf{w}\cdot\mathbf{w})/||\mathbf{w}|| = \mathbf{1}$

Also, $\mathbf{w}\cdot\mathbf{w} = ||\mathbf{w}||^2$ (the dot product of a vector with itself), so: $\mathbf{k}(||\mathbf{w}||^2)/||\mathbf{w}|| = \mathbf{1}$

$\mathbf{k}\cdot||\mathbf{w}|| = \mathbf{1}$

$\mathbf{k} = \mathbf{1}/||\mathbf{w}||$

Step 5: Calculate the total margin

- Distance from decision boundary to positive margin: $\mathbf{1}/||\mathbf{w}||$
- Distance from decision boundary to negative margin: $\mathbf{1}/||\mathbf{w}||$ (by symmetry)
- **Total margin width** $= \mathbf{1}/||\mathbf{w}|| + \mathbf{1}/||\mathbf{w}|| = \mathbf{2}/||\mathbf{w}||$

Intuition: The smaller $||w||$ is, the larger the margin. So to maximize the margin, we need to minimize $||w||$.

IV. The Optimization Problem

The Objective: Maximize the Margin

We want to: **Maximize: $2/||w||$**

This is equivalent to: **Minimize: $||w||$**

For mathematical convenience (makes derivatives cleaner), we minimize: **Minimize: $(1/2)||w||^2$**

Why $(1/2)||w||^2$?

- Minimizing $||w||$ is the same as minimizing $||w||^2$
- The factor 1/2 cancels out when we take the derivative
- $||w||^2$ is smooth and differentiable everywhere

The Constraints: Keep Points on Correct Sides

We need every data point to be:

1. Correctly classified
2. Outside or on the margin boundaries

For positive class ($y_i = +1$): Points must be on or above the positive margin: **$w \cdot x_i - b \geq +1$**

For negative class ($y_i = -1$): Points must be on or below the negative margin: **$w \cdot x_i - b \leq -1$**

Unified Constraint Form

We can combine both constraints elegantly by multiplying by y_i:

$y_i(w \cdot x_i - b) \geq 1$ for all $i = 1, 2, ..., n$

Why this works:

- If $y_i = +1$ and $w \cdot x_i - b \geq 1$, then $y_i(w \cdot x_i - b) \geq 1$ ✓
- If $y_i = -1$ and $w \cdot x_i - b \leq -1$, then $y_i(w \cdot x_i - b) = (-1)(\leq -1) \geq 1$ ✓

Final Hard Margin SVM Formulation

Minimize: $(1/2)||w||^2$

Subject to: $y_i(w \cdot x_i - b) \geq 1$, for all $i = 1, 2, ..., n$

This is a **Convex Quadratic Programming Problem** with:

- Quadratic objective function
- Linear inequality constraints
- Guaranteed unique global minimum

V. Understanding Support Vectors

Definition

Support vectors are the training points that lie exactly on the margin boundaries (not inside, not outside—right on the edge).

Mathematical Identification

For support vectors, the inequality constraint becomes an equality: $\mathbf{y_i(w \cdot x_i - b) = 1}$

The Remarkable Property

Only support vectors determine the decision boundary.

Why?

- Points far from the margin (where $y_i(w \cdot x_i - b) > 1$) have "slack" in the constraint
- Moving these points (as long as they stay on the correct side) doesn't change the optimal w and b
- Only the support vectors—the critical points touching the margin—define the solution

Practical Implication: If you have 10,000 training points but only 50 support vectors, the other 9,950 points are redundant for defining the classifier. This makes SVMs memory-efficient and robust.

Visual Intuition

Imagine balancing a ruler (the decision boundary) on top of marbles (data points):

- The ruler only touches a few marbles (support vectors)
- Moving marbles that aren't touching the ruler doesn't affect its position
- Only the touching marbles matter for the balance

Unified Constraint Form - Detailed Explanation

The Problem: Two Separate Constraints

Originally, we have TWO different constraints depending on the class:

For positive class ($y_i = +1$): $w \cdot x_i - b \geq +1$ **For negative class ($y_i = -1$):** $w \cdot x_i - b \leq -1$

Writing separate constraints is cumbersome. Can we combine them into ONE elegant constraint?

The Solution: Multiply by y_i

The trick is to multiply both sides by $\mathbf{y_i}$. Here's why this works:

Case 1: Positive Class ($y_i = +1$)

Original constraint: $w \cdot x_i - b \geq 1$

Multiply both sides by $y_i = +1$: $y_i(w \cdot x_i - b) \geq y_i \times 1$ $(+1)(w \cdot x_i - b) \geq (+1) \times 1$ $w \cdot x_i - b \geq 1$

Result: $y_i(w \cdot x_i - b) \geq 1$ ✓

The constraint remains the same because multiplying by +1 doesn't change anything.

Case 2: Negative Class ($y_i = -1$)

Original constraint: $w \cdot x_i - b \leq -1$

Multiply both sides by $y_i = -1$:

Here's the KEY insight: When you multiply an inequality by a **negative number**, the inequality sign **flips**.

$y_i(w \cdot x_i - b) \geq y_i \times (-1)$ ← Notice the flip! $(-1)(w \cdot x_i - b) \geq (-1) \times (-1)$ $-(w \cdot x_i - b) \geq +1$ $-w \cdot x_i + b \geq 1$

Let's verify this is correct. Starting from our original constraint: $w \cdot x_i - b \leq -1$ $-w \cdot x_i + b \geq +1$ ← Same thing! (multiply by -1 and flip)

Result: $y_i(w \cdot x_i - b) \geq 1$ ✓

Concrete Numerical Example

Let's say we have a student with features x_i, and our model gives $w \cdot x_i - b = -5$.

Scenario A: Student didn't get in ($y_i = -1$)

Check constraint: $w \cdot x_i - b \leq -1$? Is $-5 \leq -1$? **YES** ✓ (constraint satisfied)

Using unified form: $y_i(w \cdot x_i - b) \geq 1$? $(-1) \times (-5) = +5$ Is $+5 \geq 1$? **YES** ✓ (constraint satisfied)

Scenario B: Student got in ($y_i = +1$)

Check constraint: $w \cdot x_i - b \geq +1$? Is $-5 \geq +1$? **NO** ✗ (constraint violated - this would be a misclassification)

Using unified form: $y_i(w \cdot x_i - b) \geq 1$? $(+1) \times (-5) = -5$ Is $-5 \geq 1$? **NO** ✗ (constraint violated)

Both forms agree!

Another Example

Student with $w \cdot x_i - b = +3$

Scenario C: Student got in ($y_i = +1$)

Original: $w \cdot x_i - b \geq +1$? Is $+3 \geq +1$? **YES** ✓

Unified: $y_i(w \cdot x_i - b) \geq 1$? $(+1) \times (+3) = +3$ Is $+3 \geq 1$? **YES** ✓

Scenario D: Student didn't get in ($y_i = -1$)

Original: $w \cdot x_i - b \leq -1$? Is $+3 \leq -1$? **NO** ✗ (misclassification)

Unified: $y_i(w \cdot x_i - b) \geq 1$? $(-1) \times (+3) = -3$ Is $-3 \geq 1$? **NO** ✗ (misclassification)

Again, both forms agree!

Why This is Beautiful

Instead of writing:

if $y_i = +1$: check $w \cdot x_i - b \geq +1$

if $y_i = -1$: check $w \cdot x_i - b \leq -1$

We write one elegant constraint:

$y_i(w \cdot x_i - b) \geq 1$ for ALL i

This makes:

- The math cleaner
- Optimization algorithms simpler
- Code implementation easier

The Key Mathematical Insight

The "magic" happens because:

- Multiplying by +1 keeps the inequality direction: $\geq$ stays $\geq$
- Multiplying by -1 flips the inequality direction: $\leq$ becomes $\geq$
- The value -1 in the original negative constraint becomes +1 after multiplying by $y_i = -1$

So both cases end up as "≥ 1" after multiplication by y_i!

Understanding Time Complexity of Hard Margin SVM

What Are We Solving?

Recall our optimization problem:

Minimize: $(1/2)||w||^2$

Subject to: $y_i(w \cdot x_i - b) \geq 1$, for all $i = 1, 2, ..., n$

This is a **Quadratic Programming (QP)** problem. We need to find the optimal values of **w** and **b** that satisfy all n constraints.

Why Is It Computationally Expensive?

The Core Challenge: Interdependent Constraints

Unlike simple linear regression where you can solve directly with a formula, SVM requires:

1. Checking all n data points
2. Ensuring all constraints are satisfied simultaneously
3. Finding the maximum margin (optimal solution)

The constraints are coupled - changing w to satisfy one constraint affects all other constraints.

Method 1: Generic Quadratic Programming Solvers

Time Complexity: $O(n^3)$

How It Works:

These solvers use techniques like:

- **Interior Point Methods**
- **Active Set Methods**

Why $O(n^3)$?

Think of it as solving a system where:

1. We have n variables (related to n data points in the dual formulation)
2. At each iteration, we need to:
    - Compute an $n \times n$ matrix (Hessian matrix) → $O(n^2)$ operations
    - Invert or factor this matrix → $O(n^3)$ operations
    - Update all variables → $O(n^2)$ operations

Simple analogy: Imagine solving a puzzle where:

- You have n pieces
- Every piece affects every other piece
- You need to check all n^2 pairwise interactions
- And do this iteratively

Example with numbers:

- n = 1,000 training points
- $O(n^3) = 1{,}000^3 = 1{,}000{,}000{,}000$ operations (1 billion!)
- n = 10,000 points
- $O(n^3) = 10{,}000^3 = 1{,}000{,}000{,}000{,}000$ operations (1 trillion!)

This is why it's slow for large datasets!

Method 2: Sequential Minimal Optimization (SMO)

Time Complexity: $O(n^2)$ to $O(n^3)$, typically closer to $O(n^2)$

The Key Idea:

Instead of solving for all variables at once, SMO:

1. Breaks the problem into the smallest possible pieces
2. Optimizes just 2 variables at a time
3. Iterates until convergence

Why Is This Faster?

The trick: By fixing all variables except 2, the sub-problem becomes trivial (can be solved analytically in O(1)).

Step-by-Step Process:

Iteration structure:

1. Pick 2 variables (data points) that violate optimality conditions
2. Fix all other n-2 variables
3. Solve the 2-variable sub-problem analytically → O(1)
4. Update the solution
5. Repeat until convergence

How many iterations?

- Best case: O(n) iterations → Total: $O(n) \times O(n) = O(n^2)$
- Each iteration examines all n points → O(n)
- Worst case: Many passes needed → $O(n^3)$

Why $O(n^2)$ in Each Iteration?

Even within one iteration:

1. Evaluate which 2 variables to optimize → O(n)
2. For each pair, compute kernel values → O(n)
3. Update decision function for all points → O(n)
4. Total per iteration → O(n)

With convergence in ~n iterations → O(n²) total

Method 3: Best Case Scenario

Time Complexity: O(n × s)

Where s = number of support vectors

When Does This Happen?

This occurs when:

- Data is well-separated
- Very few points become support vectors (s << n)
- Algorithm converges quickly

Why So Fast?

Once you identify the support vectors:

- Only these s points matter
- Other n-s points don't affect the solution
- Computations only involve s points, not all n points

Example:

- n = 10,000 training points
- s = 50 support vectors (only 0.5%!)
- Complexity ≈ O(10,000 × 50) = O(500,000) instead of O(n²) = O(100,000,000)
- **That's 200× faster!**

Real-world observation: Many datasets have s ≈ 5-20% of n, making SVMs practical.

Space Complexity: O(n²)

Why Do We Need O(n²) Space?

In the dual formulation (Lagrangian), SVM needs to store the **kernel matrix K**:

$K[i,j] = kernel(x_i, x_j)$

This is an n×n matrix containing similarity between every pair of points.

Example:

- $n = 10{,}000$ points
- Kernel matrix = $10{,}000 \times 10{,}000$ = 100 million entries
- If each entry is 8 bytes (double precision) = 800 MB!

Practical Optimization:

Most implementations use:

- **Caching:** Store only frequently used rows
- **Chunking:** Process the matrix in blocks
- **After training:** Only store s support vectors → $O(s \times d)$ space

---

Prediction Time: $O(s \times d)$

How Prediction Works:

For a new point **x_new**, we compute:

prediction = sign(Σ ($\alpha_i \times y_i \times$ kernel(x_i, x_new)) + b)

Where the sum is only over **support vectors** (s of them).

Breaking It Down:

1. For each of the s support vectors:
   - Compute kernel(x_i, x_new) → $O(d)$ operations (d = feature dimension)
   - Multiply by α_i and y_i → $O(1)$
2. Sum all s terms → $O(s)$
3. Add bias b and take sign → $O(1)$

Total: $O(s \times d)$

Example:

- s = 100 support vectors
- d = 50 features
- Operations per prediction = $100 \times 50 = 5{,}000$
- **Very fast!** Can predict thousands of samples per second

Visual Comparison

Dataset size (n)	QP Solver	SMO	Best Case
100	1 million	10,000	1,000
1,000	1 billion	1 million	10,000
10,000	1 trillion	100 million	100,000

100,000 too slow! 10 billion 1 million

Why This Matters in Practice

When SVM Is Fast:

✓ Small to medium datasets ($n < 10,000$) ✓ Few support vectors ($s << n$) ✓ Well-separated classes

When SVM Is Slow:

✗ Very large datasets ($n > 100,000$) ✗ Many support vectors ($s \approx n$) ✗ High-dimensional sparse features (text data)

Alternatives for Large Data:

- **Linear SVM:** $O(n \times d)$ using stochastic gradient descent
- **Neural Networks:** Can use mini-batches
- **Random Forests:** $O(n \log n)$ per tree

Summary in Simple Terms

Training SVM is like: Organizing n students into two groups where you need to:

1. Check every student against every other student → n^2 comparisons
2. Find the widest "gap" between groups
3. Iterate until you find the perfect arrangement

The more students (n), the more comparisons (n^2), the slower it gets!

But once you find the "boundary students" (support vectors), you only need to remember them for future predictions!