Support Vector Machines (SVMs)

Geometry, Convex Optimization, Duality, Kernel Methods, and Extensions to SVC and SVR.

Contents

Sections are written to be read in any order. Each opens with its prerequisites, and the sidebar tracks where you are.

Notation

- Vectors

$x_i \in \mathbb{R}^d$ — input (feature) vector of the i-th data point

$w \in \mathbb{R}^d$ — normal vector to the separating hyperplane

$\phi(x) \in \mathcal{H}$ — feature-space representation of input x

- Scalars

$y_i \in \{-1, +1\}$ — class label of the i-th data point

$b \in \mathbb{R}$ — bias (offset) term of the hyperplane

$\gamma \in \mathbb{R}_+$ — geometric margin

$\xi_i \in \mathbb{R}_+$ — slack variable

$C \in \mathbb{R}_+$ — regularization parameter

$\epsilon \in \mathbb{R}_+$ — SVR tube radius

- Dual Variables (Scalars)

$\alpha_i \geq 0$ — Lagrange multipliers (classification)

$\alpha_i^* \geq 0$ — the *second* family of SVR multipliers, paired with α_i (Section 6). Here the $*$ is part of the name, **not** an optimality marker.

$\mu_i \geq 0$ — Lagrange multipliers for slack constraints

- Functions and Operators

$f(x) = \text{sign}(\langle w, x \rangle + b)$ — decision function

$\mathcal{K}(x, z)$ — kernel function

$\langle \cdot, \cdot \rangle$ — Euclidean inner product

$\|\cdot\|_2$ — Euclidean norm

- Sets and Indices

n — number of training samples

d — dimension of the input space

i, j — data point indices

- Optimal Quantities

w^*, b^* — optimal primal solution

α^* — optimal dual variables (classification only; see the warning above)

Quick Reference

Every formulation in this article, side by side. Each links to the section that derives it.

Hard-margin SVM — §2, §3

$$\begin{aligned}
\textbf{primal:} \quad & \min_{w,b} \ \tfrac{1}{2}\|w\|_2^2 \qquad \text{s.t.} \quad y_i\big(\langle w, x_i\rangle + b\big) \ \geq \ 1 \\
\textbf{dual:} \quad & \max_{\alpha} \ \sum_i \alpha_i \ - \ \tfrac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j \, \mathcal{K}(x_i, x_j) \\
\textbf{s.t.} \quad & \alpha_i \ \geq \ 0, \qquad \sum_i \alpha_i y_i \ = \ 0 \\
\textbf{decision:} \quad & f(x) \ = \ \operatorname{sign}\Big(\sum_i \alpha_i y_i \, \mathcal{K}(x_i, x) \ + \ b \Big) \\
\textbf{bias:} \quad & b \ = \ y_k \ - \ \sum_i \alpha_i y_i \, \mathcal{K}(x_i, x_k) \qquad \text{for any } \alpha_k > 0
\end{aligned}$$

Requires linear separability — fails outright otherwise.
scikit-learn — none directly; approximate with `SVC(C=1e6)`.

Soft-margin SVM — §5

$$\begin{aligned}
\textbf{primal:} \quad & \min_{w,b,\xi} \ \tfrac{1}{2}\|w\|_2^2 \ + \ C\sum_i \xi_i \\
& \qquad \text{s.t.} \quad y_i\big(\langle w, x_i\rangle + b\big) \ \geq \ 1 - \xi_i, \qquad \xi_i \ \geq \ 0 \\
\textbf{equivalently:} \quad & \min_{w,b} \ \tfrac{1}{2}\|w\|_2^2 \ + \ C\sum_i \max\big\{0, \ 1 - y_i f(x_i)\big\} \\
\textbf{dual:} \quad & \text{identical objective to hard-margin} \\
\textbf{s.t.} \quad & 0 \ \leq \ \alpha_i \ \leq \ C, \qquad \sum_i \alpha_i y_i \ = \ 0 \\
\textbf{decision:} \quad & f(x) \ = \ \operatorname{sign}\Big(\sum_i \alpha_i y_i \, \mathcal{K}(x_i, x) \ + \ b \Big) \\
\textbf{bias:} \quad & b \ = \ y_k \ - \ \sum_i \alpha_i y_i \, \mathcal{K}(x_i, x_k) \qquad \text{for any } \boldsymbol{0 < \alpha_k < C}
\end{aligned}$$

scikit-learn — `SVC`, `LinearSVC`.

ν-SVM — §5.6

$$\textbf{primal:} \quad \min_{w,b,\xi,\rho} \ \tfrac{1}{2}\|w\|_2^2 \ - \ \nu\rho \ + \ \tfrac{1}{n}\sum_i \xi_i$$
$$\text{s.t.} \quad y_i f(x_i) \ \geq \ \rho - \xi_i, \qquad \xi_i \ \geq \ 0, \quad \rho \ \geq \ 0$$

Meaning of ν — an upper bound on the fraction of margin errors, and a lower bound on the fraction of support vectors.

scikit-learn — `NuSVC` , `NuSVR` .

SVR — §6

$$\textbf{primal:} \quad \min_{w,b,\xi,\xi^*} \ \tfrac{1}{2}\|w\|_2^2 \ + \ C\sum_i \left(\xi_i + \xi_i^*\right)$$
$$\text{s.t.} \quad \left| y_i - f(x_i) \right| \ \leq \ \epsilon + \xi_i^{(*)}, \qquad \xi_i,\ \xi_i^* \ \geq \ 0$$
$$\textbf{equivalently:} \quad \min_{w,b} \ \tfrac{1}{2}\|w\|_2^2 \ + \ C\sum_i \max\left\{0,\ |y_i - f(x_i)| - \epsilon\right\}$$
$$\textbf{dual:} \quad \max_{\alpha,\alpha^*} \ -\tfrac{1}{2}\sum_i\sum_j (\alpha_i - \alpha_i^*)(\alpha_j - \alpha_j^*)\,\mathcal{K}(x_i, x_j)$$
$$- \ \epsilon\sum_i(\alpha_i + \alpha_i^*) \ + \ \sum_i y_i(\alpha_i - \alpha_i^*)$$
$$\textbf{s.t.} \quad 0 \ \leq \ \alpha_i,\ \alpha_i^* \ \leq \ C, \qquad \sum_i(\alpha_i - \alpha_i^*) \ = \ 0, \qquad \alpha_i\alpha_i^* = 0$$
$$\textbf{decision:} \quad f(x) \ = \ \sum_i (\alpha_i - \alpha_i^*)\,\mathcal{K}(x_i, x) \ + \ b \qquad (\text{no sign})$$

scikit-learn — `SVR` , `LinearSVR` , `NuSVR` .

Kernels — §4.4

kernel	$\mathcal{K}(x, z)$	parameters
linear	$\langle x, z\rangle$	—
polynomial	$(\langle x, z\rangle + c)^p$	$c \geq 0, p \in \mathbb{N}$
Gaussian (RBF)	$\exp\left(-\gamma\|x - z\|_2^2\right),\ \gamma = \frac{1}{2\sigma^2}$	$\gamma > 0$

Hyperparameters

knob	raising it means	see
C	less margin violation, more variance, more overfitting	§5.2, §11.3

knob	raising it means	see
γ	narrower RBF, more local influence, more overfitting	§11.3
ϵ	wider tube, **fewer** support vectors, flatter fit — in units of y	§6.7
ν	more margin errors and more support vectors allowed	§5.6

The three-case KKT table — §5.4

multiplier	margin	position
$\alpha_i = 0$	$y_i f(x_i) \geq 1$	outside the margin, not a support vector
$0 < \alpha_i < C$	$y_i f(x_i) = 1$	exactly on the margin boundary (*free* SV)
$\alpha_i = C$	$y_i f(x_i) \leq 1$	inside the margin, or misclassified (*bounded* SV)

1. The Classification(Binary) Problem as Geometry

We begin with the binary classification problem from a purely geometric perspective.
Let the training set be

$\{(x_i, y_i)\}_{i=1}^{n}$, where $x_i \in \mathbb{R}^d$ and $y_i \in \{-1, +1\}$.

The goal is to construct a decision rule that separates points of different labels with maximal robustness(maximize the minimum distance from a data point to our hyperplane, that gives = simpler model = better generalization). SVMs approach this problem by explicitly reasoning about geometry in the input space, rather than by introducing a loss function from the outset.

1.1. Linear Separability and Hyperplanes

A hyperplane in $\mathbb{R}^d$ is defined as the set

$$H = \{x \in \mathbb{R}^d \mid \langle w, x \rangle + b = 0\},$$

where $w \in \mathbb{R}^d$ is the normal vector to the hyperplane and $b \in \mathbb{R}$ is the bias term

We are building a model for a classification problem(let it be binary) .The hyperplane induces a classifier of the form

$$f(x) = \text{sign}(\langle w, x \rangle + b).$$

A small definition

$$\text{sign}(f(\cdot)) = \begin{cases} -1, & \text{if } f(\cdot) < 0 \\ 0, & \text{if } f(\cdot) = 0 \\ 1, & \text{if } f(\cdot) > 0 \end{cases}$$

The space is partitioned into two half-spaces:

$\langle w, x \rangle + b > 0$ and $\langle w, x \rangle + b < 0$,

which correspond to the two class labels $\{-1, +1\}$

Linear Separability

The dataset is said to be linearly separable if there exists at least one pair (w, b) such that

$$y_i(\langle w, x_i \rangle + b) > 0.$$

for all

$$i = 1, 2, ..., n.$$

Proof:

A point is classified correctly exactly when the sign of $\langle w, x_i \rangle + b$ matches its label. We check the two cases separately.

- If $y_i = +1$, correct classification means $\langle w, x_i \rangle + b > 0$, hence $y_i(\langle w, x_i \rangle + b) = \langle w, x_i \rangle + b > 0$.
- If $y_i = -1$, correct classification means $\langle w, x_i \rangle + b < 0$, hence $y_i(\langle w, x_i \rangle + b) = -(\langle w, x_i \rangle + b) > 0$.

In both cases

$$y_i(\langle w, x_i \rangle + b) > 0.$$

Both inequalities have to be strict: a point with $\langle w, x_i \rangle + b = 0$ lies *on* the hyperplane, and by the definition of sign above it receives no label at all.

This condition guarantees that a hyperplane exists which correctly classifies all training points.

At this stage, infinitely many separating hyperplanes may exist.But the central question becomes :

- Which separating hyperplane should be chosen?

And tadaa SVMs answer this by introducing the notion of margin!!

1.2. Functional and Geometric Margins

Functional Margin

Given a classifier (w, b), the functional margin of a labeled point (x_i, y_i) is defined as

$$\hat{\gamma}_i = y_i(\langle w, x_i \rangle + b).$$

The functional margin of the dataset is

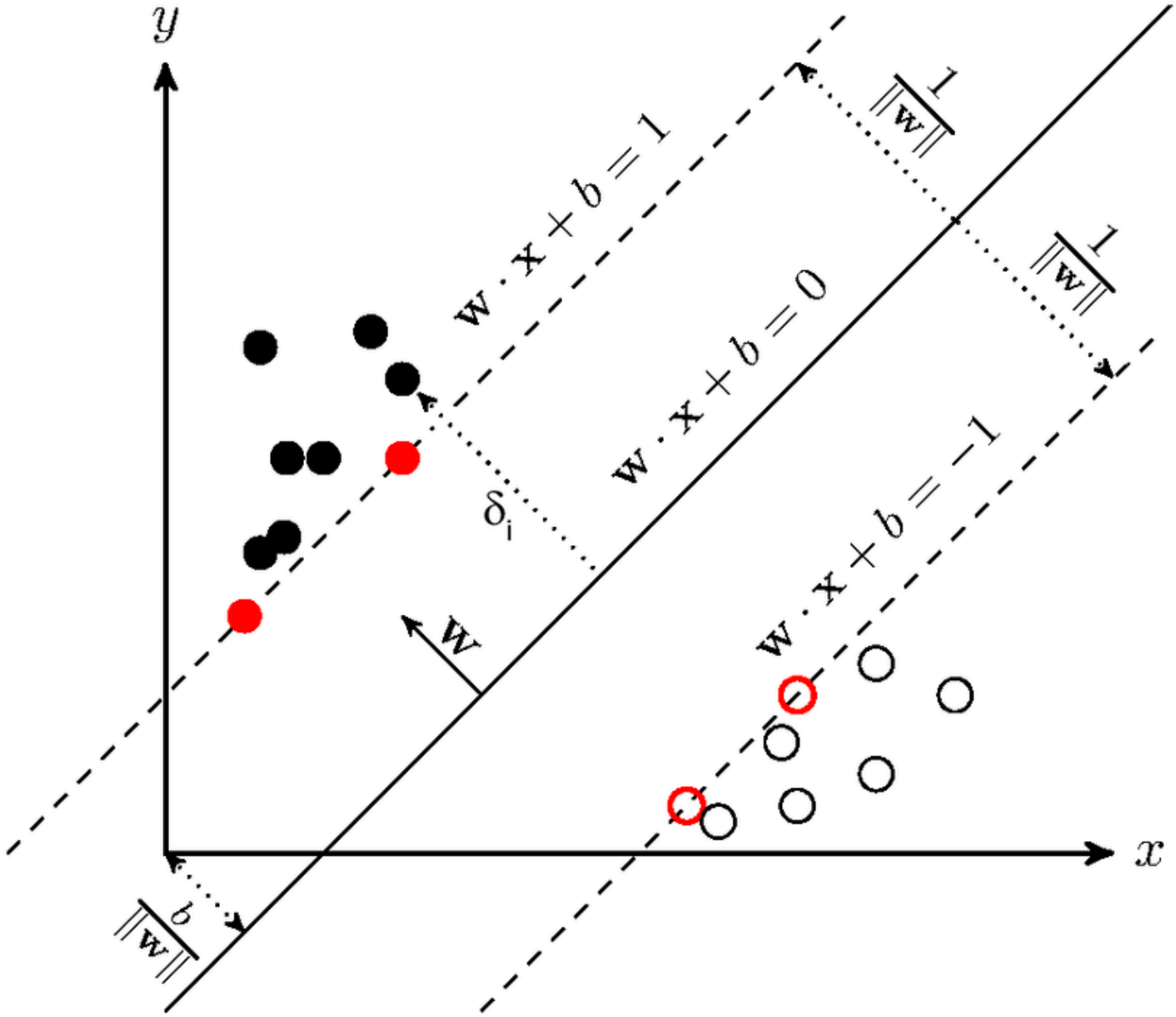
$$\hat{\gamma} = \min_i \hat{\gamma}_i.$$

While convenient algebraically, the functional margin depends on the scale of w and b. Multiplying (w, b) by any positive constant scales the functional margin arbitrarily, without changing the decision boundary.

This makes the functional margin unsuitable as a geometric notion of confidence.

Geometric Margin

The geometric margin is the distance between the hyperplane(decision boundary) and the support vectors(closest data points to the hyperplane).



source

The distance from a point x to the hyperplane H is

$$\text{dist}(x, H) = d = \frac{|\langle w, x\rangle + b|}{||w||_2}.$$

Proof:

let's assume we have an arbitrary point x_0 and a hyperplane defined by the equation $\langle w, x\rangle + b = 0$. We need to find the perpendicular distance d from x_0 to this hyperplane

Let x_p be the orthogonal projection of x_0 onto the hyperplane. The vector from x_p to x_0 , i.e. , $(x_0 - x_p)$ is parallel to the normal vector of the hyperplane w . Thus, the distance d is the length of the vector $(x_0 - x_p)$. The projection of the vector $(x_0 - x_p)$ onto the unit normal vector ($\frac{w}{\|w\|_2}$) will give us the desired distance d :

$$d = |(x_0 - x_p) \cdot \frac{w}{\|w\|_2}|$$

Since point x_p lies on the hyperplane, its coordinates satisfy the equation

$$\langle w, x_p \rangle + b = 0$$

or

$$\langle w, x_p \rangle = -b$$

Let's expand the dot product in the distance formula:

$$d = \frac{|\langle w, x_0 \rangle - \langle w, x_p \rangle|}{\|w\|_2}$$

Substitute $\langle w, x_p \rangle = -b$:

$$d = \frac{|\langle w, x_0 \rangle - (-b)|}{\|w\|_2}$$

$$d = \frac{|\langle w, x_0 \rangle + b|}{\|w\|_2}$$

Accordingly, the geometric margin of a labeled point (x_i, y_i) is

$$\gamma_i = \frac{y_i(\langle w, x_i \rangle + b)}{\|w\|_2}.$$

The geometric margin of the dataset is

$$\gamma = \min_i \gamma_i.$$

Unlike the functional margin, the geometric margin is invariant under rescaling of (w, b) and has a direct geometric interpretation.

1.3. Margin Maximization Principle

Among all separating hyperplanes, Support Vector Machines choose the one that maximizes the geometric margin :

$$\max_{w,b} \gamma.$$

Geometrically, this corresponds to selecting the hyperplane that is as far as possible from the closest data points of either class.

Why Margin Matters ?

Maximizing the margin has several fundamental consequences:

1. Robustness to perturbations
   A larger margin implies that small perturbations of the input vectors are less likely to change the classification outcome.
2. Capacity control
   From statistical learning theory, classifiers with larger margins have lower effective capacity, leading to better generalization guarantees. This is the load-bearing claim of the whole method, so Section 1.4 makes it precise.
3. Uniqueness of solution
   While many separating hyperplanes may exist, the maximum-margin hyperplane is unique whenever the data are linearly separable — both w^* and b^* are pinned down. (Uniqueness of b can fail once we relax to the soft-margin problem; see Section 5.)

This principle provides the conceptual bridge between geometry and generalization, which will later be formalized through convex optimization and duality.

1.4. Why a Large Margin Actually Generalizes

It is easy to assert that a wide margin is "more robust" and leave it there. But there is a real theorem underneath, and it is worth stating, because it is also the reason the kernel trick of Section 4 does not immediately destroy everything.

The problem with counting dimensions

The classical way to bound the capacity of a hypothesis class is the **VC dimension** — the largest number of points the class can shatter (label in all possible ways). For unrestricted hyperplanes in $\mathbb{R}^d$,

$$\mathrm{VC}(\{\text{hyperplanes in } \mathbb{R}^d\}) = d + 1.$$

The standard VC generalization bound then says that with probability at least $1-\eta$ over the draw of the training set, the true risk $R(f)$ and the empirical risk $\hat{R}(f)$ satisfy

$$R(f) \;\le\; \hat{R}(f) + \sqrt{\frac{h\left(\log\frac{2n}{h}+1\right) - \log\frac{\eta}{4}}{n}},$$

where h is the VC dimension. Read literally this is a disaster for kernel methods: the Gaussian kernel maps into an *infinite*-dimensional space, so $h=\infty$ and the bound is vacuous. If dimension were really what controlled generalization, an RBF SVM could never work at all. And yet it does.

The resolution: margin, not dimension

The escape is that we are not using *all* hyperplanes. We are only using those that separate the data with a margin of at least γ, and that is a much smaller class.

> **Theorem (Vapnik).** Consider the class of hyperplanes with geometric margin at least γ on data contained in a ball of radius R. Its VC dimension satisfies
>
> $$h \;\le\; \min\left(\left\lceil \frac{R^2}{\gamma^2} \right\rceil,\; d\right) + 1.$$

The important part is the $\frac{R^2}{\gamma^2}$ term, because **it does not mention d at all**. Once the margin is wide enough that $R^2/\gamma^2 < d$, the ambient dimension stops mattering entirely and only the ratio of data radius to margin does.

This is precisely the licence the kernel trick needs. We may map into a space of infinite dimension, because the bound that governs us is controlled by R^2/γ^2, and both R and γ are measured *in the feature space* where they remain finite. Maximizing the margin is therefore not a heuristic for robustness — it is direct minimization of the quantity that appears in the capacity bound.

Caveats

First, these VC bounds are notoriously **loose**. They are existence results about worst-case distributions, and the numerical value they produce for a real dataset is often greater than 1 (i.e. it tells you the error rate is at most 130%, which is true and useless). Their value is structural: they tell you *which quantity* to optimize, not what your test error will be. You still need a validation set for that.

Second, the modern treatment usually replaces VC dimension with **Rademacher complexity**, which gives a sharper and cleaner margin bound of the rough form

$$R(f) \;\lesssim\; \hat{R}_\gamma(f) + \frac{2R}{\gamma\sqrt{n}} + \sqrt{\frac{\log(1/\eta)}{2n}},$$

where $\hat{R}_\gamma$ is the fraction of training points with margin below γ. Same moral, better constants, and no shattering arguments required. Either way the conclusion is the one that matters here: **the margin is the capacity control, and the dimension is not.**

1.5. Scale Normalization

Because the decision boundary defined by (w, b) is invariant under positive rescaling, we may impose a normalization constraint without loss of generality: we fix the functional margin **of the dataset** to be $\hat{\gamma} = 1$. That is, we rescale (w, b) so that the *closest* point has functional margin exactly one:

$$\min_i \; y_i(\langle w, x_i \rangle + b) = 1$$

Proof :

We can always rescale (w, b) so that $\min_i |\langle w, x_i \rangle + b| = 1$.

Let

$$c = \min_i |\langle w, x_i \rangle + b|.$$

Note that $c > 0$: if $c = 0$ then some training point would lie exactly on the hyperplane, contradicting separability. So we are allowed to divide by c.

Now substitute $w_{\text{new}} = \frac{w}{c}$ and $b_{\text{new}} = \frac{b}{c}$:

$$\min_i |\langle w_{\text{new}}, x_i \rangle + b_{\text{new}}| = \min_i \left| \left\langle \tfrac{w}{c}, x_i \right\rangle + \tfrac{b}{c} \right|$$

$$= \min_i \frac{|\langle w, x_i \rangle + b|}{c} = \frac{1}{c} \min_i |\langle w, x_i \rangle + b| = \frac{c}{c} = 1.$$

Since $c > 0$, the pair $(w_{\text{new}}, b_{\text{new}})$ describes the **same** hyperplane and the same classifier as (w, b) — only the scale has changed.

Now let's maximize the margin keeping in mind those derivations

$$\max_{w,b} \; \frac{\min_i y_i(\langle w, x_i \rangle + b)}{\|w\|_2}$$

$$\max_{w,b} \; \frac{\min_i \hat{\gamma}_i}{\|w\|_2}$$

$$\max_{w,b} \frac{1}{\|w\|_2}$$

Under this normalization, maximizing the geometric margin is equivalent to minimizing $\|w\|_2$.

2. Hard-Margin Support Vector Machine

Prerequisites: §1 — margins, scale normalization.
Establishes: the primal quadratic program, support vectors, and why the primal alone is not enough.

2.1. Primal Optimization Problem

Under the normalization constraint, the problem of finding the maximum-margin separating hyperplane can be written as the following optimization problem:

$\min_{w,b} \frac{1}{2} \|w\|_2^2$ s.t. $y_i(\langle w, x_i \rangle + b) \geq 1, i = 1, 2, ..., n.$

This is known as the hard-margin Support Vector Machine. The fraction $\frac{1}{2}$ and a term $\|w\|_2^2$ are added for mathematical convenience(for smoothness and differentiability when optimizing) . Minimizing $\|w\|_2$ is equivalent to minimizing $\|w\|_2^2$ cause it yields the same optimal result.

Remarks

1. Quadratic objective
   The objective function $\frac{1}{2} \|w\|_2^2$ is a strictly convex quadratic function in w.
2. Linear Constraints
   The constraints are affine functions of (w, b), defining a convex feasible region.
3. Convexity
   Since the objective is convex and the feasible set is convex, this is a convex optimization problem. Any local minimum is also a global minimum.
4. Uniqueness
   Because the objective is **strictly** convex in w, the solution w^* is unique whenever the data are linearly separable. The bias b^* is then unique as well: at the optimum at least one constraint is active in *each* class, and those two equalities pin b down. (In the soft-margin problem of Section 5 this can fail, and b may range over an interval.)

2.2. Geometric Interpretation of the Constraints

Each constraint

$$y_i(\langle w, x_i \rangle + b) \geq 1$$

enforces that the point x_i lies on the correct side of the hyperplane and at least a distance $\frac{1}{\|w\|_2}$ from it.

The points that satisfy

$$y_i(\langle w, x_i \rangle + b) = 1$$

lie exactly on the margin boundaries. These points are the support vectors. All other points satisfy the constraints strictly and do not influence the optimal solution.

2.3. Support Vectors and Sparsity

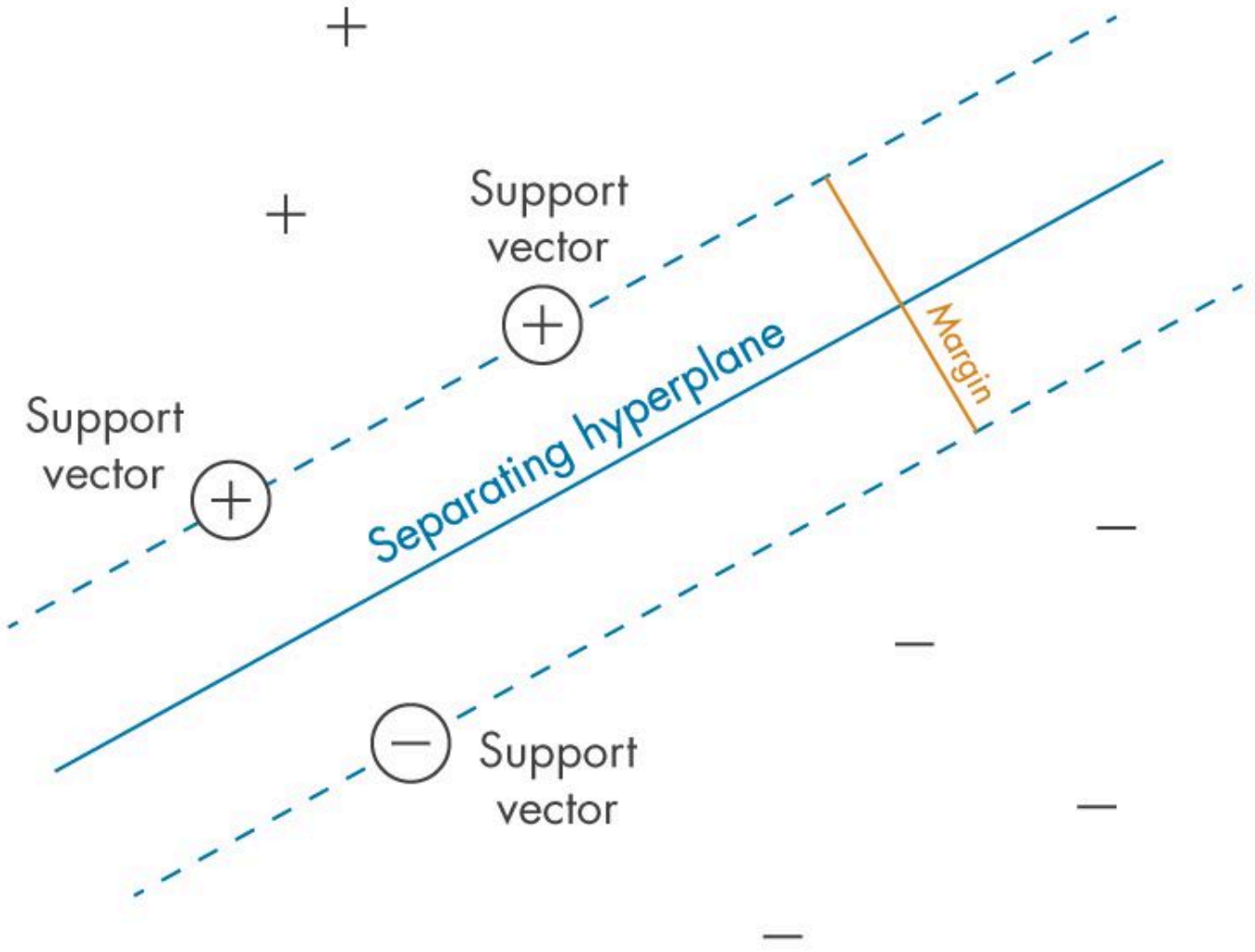
A fundamental consequence of the margin maximization formulation is that only a subset of the training points determines the solution!!

Definition.

A data point (x_i, y_i) is called a support vector if

$$y_i(\langle w^*, x_i \rangle + b^*) = 1,$$

where (w^*, b^*) denotes the optimal solution of the primal problem.



source

Geometrically, support vectors are the points closest to the decision boundary. Removing any non-support vector does not change the optimal hyperplane.

This sparsity property will later emerge naturally from the DUAL formulation.

You may ask "Why the Primal Formulation Is Not Enough?"

While the primal problem provides clear geometric intuition, it has two important limitations:

1.It relies explicitly on the assumption of linear separability.

2.It depends directly on the inner product $\langle w, x \rangle$, which restricts us to linear decision boundaries. Both limitations will be addressed by:

- Introducing Lagrangian duality

- Reformulating the problem in terms of inner products between data points.

3. Lagrangian Duality and The Dual Problem

Prerequisites: §2 — the primal problem.
Establishes: KKT conditions, Slater's condition and strong duality, the dual QP, where sparsity comes from, and how to recover b.

The primal formulation of the hard-margin Support Vector Machine is a convex optimization problem with inequality constraints. To analyze its structure and derive a more flexible representation, we apply Lagrangian duality. Before we start, let me firstly introduce several concepts.

What is a convex optimization problem?

A constrained optimization problem is called a convex optimization problem if

$\min_x f(x)$ <- $f(x)$ is a convex function

s.t.

$g_i(x) \leq 0$ for all $i = 1, ..., m$ <- $g_i(x)$ is also a convex function

and $h_j(x) = 0$ for all $j = 1, ..., p$ <- $h_j(x) = 0$ is an affine equality (thus, a convex set)

The KKT conditions are the workhorse from here on, but the exact logic is worth stating carefully, because it is easy to get backwards:

- For a **convex** problem, any point satisfying the KKT conditions **is** optimal. Sufficiency comes for free.
- **Necessity** — that every optimal point satisfies KKT — is *not* automatic. It requires a constraint qualification, most commonly **Slater's condition**: there exists a strictly feasible point, i.e. some x with $g_i(x) < 0$ for all i.

For a linearly separable training set the hard-margin SVM does satisfy Slater's condition (take any separating (w, b) and scale it up until every constraint is strict), so KKT is here both necessary and sufficient. Slater also buys us **strong duality**: the optimal dual value equals the optimal primal value,

with no duality gap. That is precisely what licenses us to solve the dual instead of the primal — the two problems have the same answer.

Karush-Kuhn-Tucker (KKT) Conditions

1. Primal Feasibility

The solution must satisfy the constraints :

$$g_i(x) \leq 0, h_j(x) = 0$$

2. Dual Feasibility

Lagrange multipliers must satisfy

$$\alpha_i \geq 0$$

3. Stationarity

Gradient of the Lagrangian with respect to primal variables is zero :

$$\nabla_x \mathcal{L}(x, \alpha, v) = 0$$

Stationarity connects primal variables to dual variables

4. Complementary Slackness

For each component, either the constraint is active or its multiplier is zero, never both nonzero :

$$\alpha_i g_i(x) = 0$$

3.1. The Primal Optimization Problem, The Lagrangian

We introduce a non-negative Lagrange multiplier $\alpha_i \geq 0$ for each constraint

$$y_i(\langle w, x_i \rangle + b) \geq 1.$$

The Lagrangian associated with the primal problem is

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2} \|w\|_2^2 - \sum_{i=1}^{n} \alpha_i [y_i(\langle w, x_i \rangle + b) - 1],$$

where $\alpha = (\alpha_1, ..., \alpha_n)$ - vector of lagrange multipliers

The primal problem corresponds to minimizing $\mathcal{L}$ with respect to (w, b) and maximizing with respect to $\alpha \geq 0$

To obtain the dual formulation, we minimize the Lagrangian with respect to the primal variables w and b

Let's compute the derivatives with respect to primal variables and apply stationarity conditions

Derivative with respect to w

$$\frac{\partial \mathcal{L}}{\partial w} = w - \sum_{i=1}^{n} \alpha_i y_i x_i.$$

Setting the derivative to zero yields

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

This equation already reveals an important fact : the normal vector w lies in the span of the training points

Derivative with respect to b

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{i=1}^{n} \alpha_i y_i$$

Setting it to zero gives

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

3.2. The Dual Optimization Problem

Substituting the stationarity conditions back into the Lagrangian eliminates the primal variables (w, b). The resulting dual objective depends only on the dual variables α_i.

Usually it's convenient to maximize the dual while minimizing the primal so let's simplify and derive the dual problem. Firstly,we expand our primal lagrangian(hard margin)

$$\mathcal{L}_{primal}(w, b, \alpha) = \frac{1}{2} \|w\|_2^2 - \sum_i \alpha_i y_i \langle w, x_i \rangle - b \sum_i \alpha_i y_i + \sum_i \alpha_i$$

Now let's substitute the derivatives w.r.t. primal variables

$$\mathcal{L}(w,b,\alpha) = \frac{1}{2}\left\|\sum_i \alpha_i y_i x_i\right\|_2^2 - \langle w, w\rangle - b * 0 + \sum_i \alpha_i$$

Keep in mind that

$$\left\|\sum_i \alpha_i y_i x_i\right\|_2^2 = \langle \sum_i \alpha_i y_i x_i, \sum_j \alpha_j y_j x_j\rangle = \sum_i \sum_j \alpha_i \alpha_j y_i y_j \langle x_i, x_j\rangle$$

Thus,

$$\mathcal{L}(w,b,\alpha) = \frac{1}{2}[\sum_i \sum_j \alpha_i \alpha_j y_i y_j \langle x_i, x_j\rangle] - \sum_i \sum_j \alpha_i \alpha_j y_i y_j \langle x_i, x_j\rangle + \sum_i \alpha_i$$

Our Dual

$$\mathcal{L}_{dual} = \sum_i \alpha_i - \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j \langle x_i, x_j\rangle$$

The dual optimization problem becomes

$$\max_\alpha \sum_i \alpha_i - \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j \langle x_i, x_j\rangle$$

subject to

$$\alpha_i \geq 0, \sum \alpha_i y_i = 0$$

This is a *quadratic programming* problem in the variables α_i

Remarks

1.Dependence on inner products only

The data appear exclusively through inner products $\langle x_i, x_j\rangle$. This observation is the key to kernel methods.

2.Dimensionality

The optimization is performed in $\mathbb{R}^n$, independent of the ambient dimension d.

3.Convexity

The dual objective is concave and the feasible region is convex, so every local maximum is a global maximum.

This does **not**, however, give uniqueness. The Hessian of the dual objective is $-Q$ where $Q_{ij} = y_i y_j \langle x_i, x_j \rangle$, and Q is only positive *semi*definite. Whenever Q is singular — for instance when $n > d$ with a linear kernel, or when the training set contains duplicated points — the set of maximizers is a convex set rather than a single point. The primal solution w^* is unique; the dual solution α^* need not be.

3.3. Connecting the Dual: Sparsity and the Bias Term

Using the stationarity condition $w^* = \sum_i \alpha_i y_i x_i$, the classifier can be written entirely in terms of the dual variables:

$$f(x) = \operatorname{sign}\left(\sum_i \alpha_i y_i \langle x_i, x \rangle + b \right).$$

Where the sparsity comes from

This is the promise made back in Section 2.3, and complementary slackness is what delivers it. For every i,

$$\alpha_i \left[y_i (\langle w^*, x_i \rangle + b^*) - 1 \right] = 0.$$

A product is zero only if one of its factors is. So for each training point exactly one of two things holds:

- $y_i(\langle w^*, x_i \rangle + b^*) > 1$ — the point sits strictly outside the margin, its constraint is inactive, and therefore $\alpha_i = 0$;
- $y_i(\langle w^*, x_i \rangle + b^*) = 1$ — the point sits exactly on the margin boundary, and α_i is free to be positive.

Since $w^* = \sum_i \alpha_i y_i x_i$, only the second group contributes anything at all to w^*. Those are precisely the support vectors. This is also why deleting a non-support vector leaves the solution untouched: its multiplier was already zero, so it was never part of the sum.

Recovering the bias

The dual hands us w^* but not b^*. We recover it from any support vector x_k (any k with $\alpha_k > 0$), whose constraint holds with equality:

$$y_k \left(\sum_i \alpha_i y_i \langle x_i, x_k \rangle + b^* \right) = 1.$$

Multiplying through by y_k and using $y_k^2 = 1$:

$$b^* = y_k - \sum_i \alpha_i y_i \langle x_i, x_k \rangle.$$

In practice one averages this expression over all support vectors, which is numerically more stable than relying on a single point.

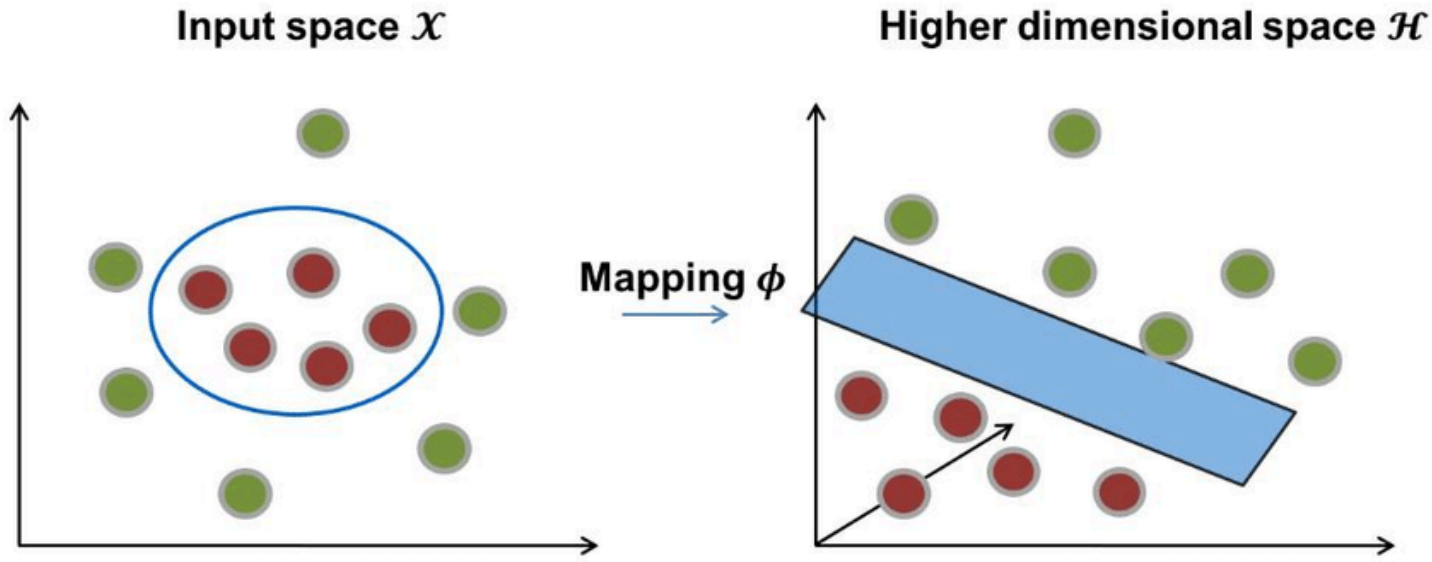
4. Kernel Methods: From Inner Products to Nonlinear Decision Boundaries

Prerequisites: §3 — the dual depends on data only through inner products.
Establishes: the kernel trick, Mercer's condition, closure rules, explicit feature maps, and the representer theorem.

4.1. From Feature Maps to Kernels

Suppose we map inputs into a (possibly high-dimensional) feature space:

$$\phi : \mathbb{R}^d \to \mathcal{H},$$



source

Where $\mathcal{H}$ is a Hilbert space(Inner-product space) . A linear classifier in feature space has the form

$$f(x) = \text{sign}(\langle w, \phi(x) \rangle_{\mathcal{H}} + b).$$

If we try to directly work with the feature mapping $\phi(x)$, we may need to handle extremely high-dimensional vectors. The kernel method avoids explicit computation of those mappings.

4.2. Kernel Trick

Let's firstly introduce the concept of a kernel

What is a kernel $\mathcal{K}$?

A kernel is a function

$$\mathcal{K} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$$

such that

$$\mathcal{K}(x, z) = \langle \phi(x), \phi(z) \rangle_{\mathcal{H}}$$

for some feature mapping ϕ into some inner-product space $\mathcal{H}$

In the SVM dual , we replace every $\langle x_i, x_j \rangle$ by $\mathcal{K}(x_i, x_j)$. The hard-margin dual becomes:

$$\max_{\alpha} \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathcal{K}(x_i, x_j)$$

subject to

$$\alpha_i \geq 0, \sum_i \alpha_i y_i = 0.$$

The corresponding decision function is

$$f(x) = \text{sign}(\sum_i \alpha_i y_i \mathcal{K}(x_i, x) + b).$$

Only support vectors (those with $\alpha_i > 0$) contribute to the sum.

4.3. Which Functions Are Valid Kernels?

Not every symmetric function $\mathcal{K}(x, z)$ is a valid kernel. The key requirement is that the kernel corresponds to an inner product in some (possibly infinite-dimensional) feature space.

Let me formalize it by introducing the definition of the *Gram matrix* and a simple version of *Mercer's Theorem*.

Definition

Given points $x_1, ..., x_n$, we define the Gram matrix $G \in \mathbb{R}^{n \times n}$ by

$$G_{ij} = \mathcal{K}(x_i, x_j).$$

A function $\mathcal{K}$ is a valid (Mercer) kernel if, for any finite set $\{x_1, ..., x_n\}$, the Gram matrix G is positive semidefinite (PSD), i.e.

$$\text{spectrum}(G) \geq 0$$

or
$c^\intercal G c \geq 0$ for all $c \in \mathbb{R}^n$.
This PSD condition is what ensures that $\mathcal{K}$ behaves like an inner product.

In general ,a Gram matrix G of vectors $x_1, x_2, ..., x_n$ is the matrix of all possible dot products of those vectors :

$$G(x_1, x_2, ..., x_n) = \begin{bmatrix} \langle x_1, x_1 \rangle & \langle x_1, x_2 \rangle & ... & \langle x_1, x_n \rangle \\ \langle x_2, x_1 \rangle & \langle x_2, x_2 \rangle & ... & \langle x_2, x_n \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle x_n, x_1 \rangle & \langle x_n, x_2 \rangle & ... & \langle x_n, x_n \rangle \end{bmatrix}$$

Mercer's Theorem in a nutshell

If $\mathcal{K}$ is symmetric and positive semidefinite (under mild regularity assumptions), then there exists a feature mapping ϕ into a Hilbert space $\mathcal{H}$ such that

$$\mathcal{K}(x, z) = \langle \phi(x), \phi(z) \rangle_{\mathcal{H}}.$$

Thus,using $\mathcal{K}$ in the dual is just the same as running a linear SVM in the feature space $\mathcal{H}$, without explicitly knowing the mapping ϕ.

4.4. Examples of Most Common Kernels

1.Linear Kernel

$$\mathcal{K}(x, z) = \langle x, z \rangle$$

2.Polynomial Kernel

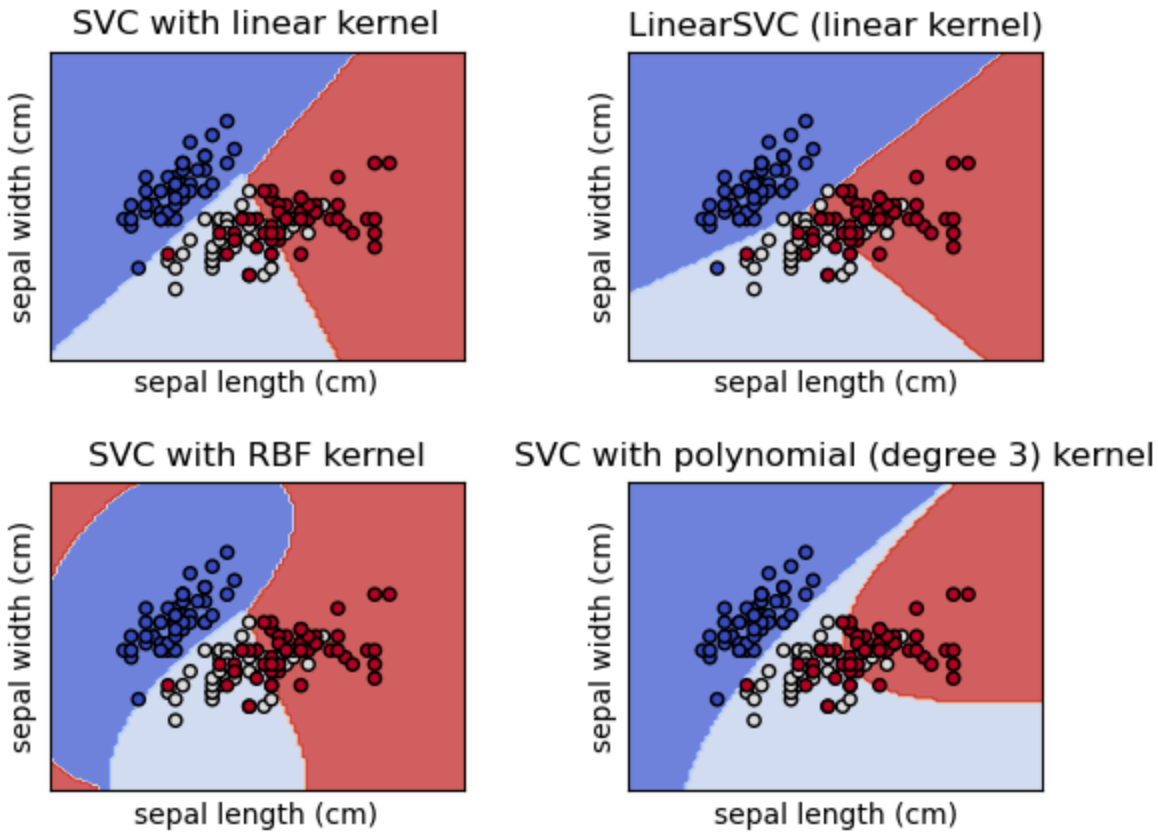
$$\mathcal{K}(x, z) = (\langle x, z \rangle + c)^p$$

$c \geq 0, p \in \mathbb{N}$.

3.Gaussian(RBF) kernel

$$\mathcal{K}(x, z) = \exp(-\frac{\|x - z\|_2^2}{2\sigma^2}),$$

$\sigma > 0$.



source

4.5. Building Kernels: Closure Rules

Listing three kernels and asserting they are valid is unsatisfying. In practice one rarely verifies Mercer's condition from scratch — instead one builds new kernels out of old ones using a small set of closure rules. Each has a one-line proof, and together they cover almost everything you will meet.

Let $\mathcal{K}_1, \mathcal{K}_2$ be valid kernels with Gram matrices G_1, G_2, and let $c \in \mathbb{R}^n$ be arbitrary.

1. Non-negative scaling. $\mathcal{K} = a\mathcal{K}_1$ with $a \geq 0$ is valid, since $c^\top (aG_1)c = a\,(c^\top G_1 c) \geq 0$.

2. Sums. $\mathcal{K} = \mathcal{K}_1 + \mathcal{K}_2$ is valid, since

$$c^\top (G_1 + G_2)c = c^\top G_1 c + c^\top G_2 c \geq 0.$$

3. Rank-one kernels. For any function $f : \mathbb{R}^d \to \mathbb{R}$, the kernel $\mathcal{K}(x, z) = f(x)f(z)$ is valid. Its Gram matrix is $vv^\top$ with $v_i = f(x_i)$, so

$$c^\top vv^\top c = (v^\top c)^2 \geq 0.$$

4. Products. $\mathcal{K} = \mathcal{K}_1\mathcal{K}_2$ is valid. The Gram matrix is the elementwise (Hadamard) product $G_1 \circ G_2$, and the **Schur product theorem** says the Hadamard product of two PSD matrices is PSD.

5. Conjugation. $\mathcal{K}(x, z) = f(x)\mathcal{K}_1(x, z)f(z)$ is valid — combine rules 3 and 4.

6. Limits. A pointwise limit of valid kernels is valid, since the PSD condition $c^\top G c \geq 0$ is preserved under limits.

From these, polynomials with non-negative coefficients of a valid kernel are valid, and so is $\exp(\mathcal{K}_1)$ — which is exactly what we need next.

4.6. What the Feature Maps Actually Look Like

Mercer's theorem promises a ϕ exists but does not hand it to you. For the two nonlinear kernels above we can write it down explicitly, and doing so makes the whole "implicit high-dimensional space" story concrete rather than mystical.

The polynomial kernel

Take $d = 2, p = 2$, and expand directly:

$$\mathcal{K}(x, z) = (\langle x, z \rangle + c)^2 = (x_1 z_1 + x_2 z_2 + c)^2$$

$$= x_1^2 z_1^2 + x_2^2 z_2^2 + 2x_1 x_2 z_1 z_2 + 2c x_1 z_1 + 2c x_2 z_2 + c^2.$$

Every term is a product of something depending only on x with something depending only on z, so we can simply read off

$$\phi(x) = \left(x_1^2,\ x_2^2,\ \sqrt{2}\,x_1x_2,\ \sqrt{2c}\,x_1,\ \sqrt{2c}\,x_2,\ c\right) \in \mathbb{R}^6.$$

Two input dimensions became six feature dimensions, containing every monomial up to degree 2. In general $(\langle x, z\rangle + c)^p$ corresponds to all monomials up to degree p, of which there are $\binom{d+p}{p}$ — for $d = 100, p = 5$ that is over 96 million coordinates. Computing ϕ explicitly is hopeless; computing $\mathcal{K}$ costs one dot product and one power. That gap **is** the kernel trick.

The Gaussian kernel is genuinely infinite-dimensional

Take $d = 1$ and $\sigma = 1$ for readability. Expand the square in the exponent:

$$\mathcal{K}(x, z) = e^{-\frac{(x-z)^2}{2}} = e^{-\frac{x^2}{2}}\, e^{-\frac{z^2}{2}}\, e^{xz}.$$

Now expand the last factor as its power series:

$$e^{xz} = \sum_{k=0}^{\infty} \frac{(xz)^k}{k!} = \sum_{k=0}^{\infty} \frac{x^k}{\sqrt{k!}} \cdot \frac{z^k}{\sqrt{k!}}.$$

Substituting back and reading off the coordinates:

$$\phi(x) = e^{-\frac{x^2}{2}}\, e^{-\frac{z^2}{2}} \left(1 + \left(\frac{x}{\sqrt{1!}} \cdot \frac{z}{\sqrt{1!}}\right) + \left(\frac{x^2}{\sqrt{2!}} \cdot \frac{z^2}{\sqrt{2!}}\right) + \left(\frac{x^3}{\sqrt{3!}} \cdot \frac{z^3}{\sqrt{3!}}\right) + \ \dots\right).$$

This is an honest infinite sequence — the RBF kernel corresponds to a feature space containing *every* polynomial degree at once, damped by a Gaussian envelope. You could never write this vector down, yet $\mathcal{K}(x, z)$ costs one subtraction, one norm and one $\exp$.

This also **proves** the Gaussian kernel is PSD, using the closure rules: $\langle x, z\rangle^k$ is a product of k linear kernels (rule 4), the coefficients $1/k!$ are non-negative (rule 1), the series is a limit of finite sums (rules 2 and 6), and the $e^{-x^2/2}e^{-z^2/2}$ prefactor is rule 5. No integral operators required.

4.7. The Representer Theorem

In Section 3.1 the stationarity condition handed us $w = \sum_i \alpha_i y_i x_i$, and we noted in passing that w lies in the span of the training points. That was not an accident of the SVM. It is an instance of a general principle covering essentially every regularized kernel method — kernel ridge regression, kernel logistic regression, SVR, and the SVM alike.

> **Representer Theorem.** Let $\mathcal{H}$ be the reproducing kernel Hilbert space of a kernel $\mathcal{K}$, let L be any loss function, and let Ω be strictly increasing. Then any minimizer of
>
> $$> \min_{f \in \mathcal{H}} \sum_{i=1}^{n} L\big(y_i, f(x_i)\big) + \Omega\big(\|f\|_{\mathcal{H}}\big) >$$
>
> admits a representation
>
> $$> f^*(\cdot) = \sum_{i=1}^{n} \alpha_i \, \mathcal{K}(x_i, \cdot). >$$

Why this matters. We are optimizing over $\mathcal{H}$, which may be infinite-dimensional — an intractable search over functions. The theorem says the solution always lives in the n-dimensional subspace spanned by the training points. An infinite-dimensional problem collapses to finding n numbers. This, not Mercer's theorem, is what makes kernel methods computable.

Proof:

The key property of an RKHS is **reproducing**: for every $f \in \mathcal{H}$ and every point x,

$$f(x) = \langle f, \mathcal{K}(x, \cdot) \rangle_{\mathcal{H}}.$$

Let $S = \operatorname{span}\{\mathcal{K}(x_1, \cdot), \ldots, \mathcal{K}(x_n, \cdot)\}$ and decompose any $f \in \mathcal{H}$ orthogonally as

$$f = f_{\parallel} + f_{\perp}, \qquad f_{\parallel} \in S, \quad f_{\perp} \perp S.$$

Now evaluate f at a training point. By the reproducing property and because $f_{\perp}$ is orthogonal to every $\mathcal{K}(x_j, \cdot)$,

$$f(x_j) = \langle f_{\parallel} + f_{\perp}, \mathcal{K}(x_j, \cdot) \rangle = \langle f_{\parallel}, \mathcal{K}(x_j, \cdot) \rangle + 0 = f_{\parallel}(x_j).$$

So **the loss term cannot see $f_{\perp}$ at all** — the predictions on the training data are identical whether $f_{\perp}$ is there or not.

The penalty, however, does see it. By orthogonality (Pythagoras in $\mathcal{H}$),

$$\|f\|_{\mathcal{H}}^2 = \|f_{\parallel}\|_{\mathcal{H}}^2 + \|f_{\perp}\|_{\mathcal{H}}^2 \;\geq\; \|f_{\parallel}\|_{\mathcal{H}}^2,$$

with equality if and only if $f_{\perp} = 0$. Since Ω is strictly increasing, dropping $f_{\perp}$ leaves the loss unchanged and *strictly* decreases the penalty unless $f_{\perp}$ was already zero.

Therefore any minimizer must have $f_{\perp} = 0$, i.e. $f^* \in S$, which is exactly the claimed representation. ■

Setting L to the hinge loss and $\Omega(t) = \frac{1}{2}t^2$ recovers the SVM, and α_i absorbs the y_i — so $w^* = \sum_i \alpha_i y_i \phi(x_i)$ is just this theorem in disguise.

5. Soft-Margin Support Vector Machines

Prerequisites: §2 and §3. §4 if you want the kernelized form.
Establishes: slack variables, the hinge loss, the box-constrained dual, the three-case KKT table, and ν-SVM. This is the formulation you will actually use.

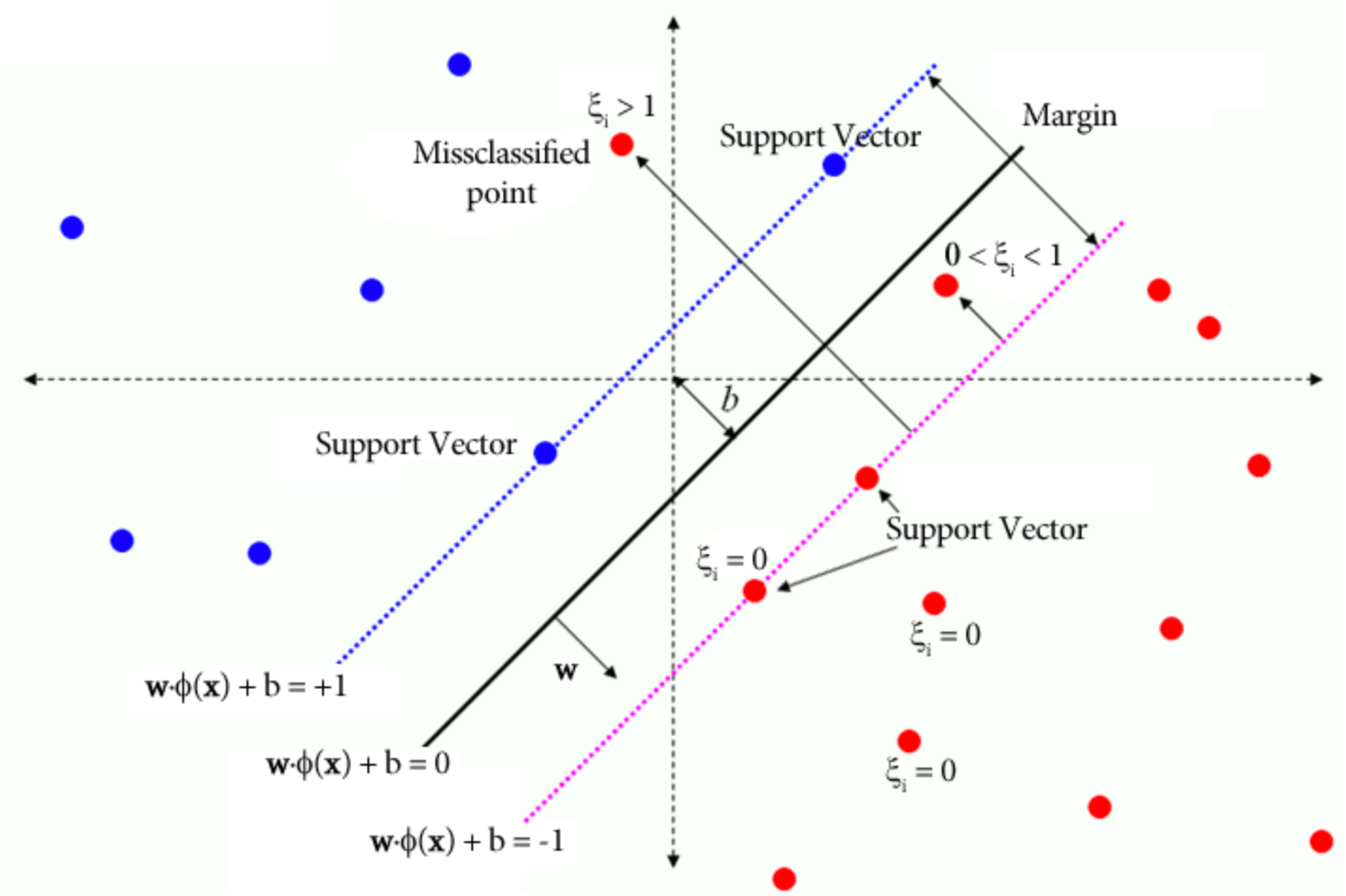
5.1. Slack Variables and Constraint Relaxation

For each training point , we introduce a slack variable $\xi_i \geq 0$ that measures the degree of constraint violation. The margin constraints become

$$y_i(\langle w, x_i \rangle + b) \geq 1 - \xi_i, i = 1, ..., n.$$

Interpretation of ξ_i :

- $\xi_i = 0$: point lies on or outside the margin and correctly classified.
- $0 < \xi_i < 1$: point is correctly classified but violates the margin.
- $\xi_i = 1$: point lies exactly *on* the decision boundary.
- $\xi_i > 1$: point is misclassified.



source

5.2. Primal Soft-Margin Optimization Problem

The soft-margin SVM trades off margin maximization against constraint violations:

$$\min_{w,b,\xi} \frac{1}{2} \|w\|_2^2 + C \sum_i \xi_i$$

subject to

$$y_i(\langle w, x_i \rangle + b) \geq 1 - \xi_i$$

$i = 1, \ldots, n$ and $\xi_i \geq 0$

Here the hyperparameter $C > 0$ controls the bias-variance tradeoff like :

- Large C : heavy penalty to violations = low bias, high variance
- Small C : allows violations for wider margin = higher bias , lower variance

Now, before we continue ,I will show you another, BUT FOUNDATIONAL concept for SVMs - the *hinge loss*

Definition

$$\ell_{hinge}(y, f(x)) = \max\{0, 1 - yf(x)\}.$$

5.3. Connection to Hinge Loss

The constrained soft-margin problem can be rewritten as an unconstrained regularized risk minimization problem.Then the soft-margin SVM is equivalent to:

$$\min_{w,b} \frac{1}{2}\|w\|_2^2 + C\sum_i \max\{0, 1 - y_i(\langle w, x_i\rangle + b)\}.$$

The equivalence is easy to see. At the optimum each ξ_i is as small as the constraints permit: it must satisfy both $\xi_i \geq 1 - y_i f(x_i)$ and $\xi_i \geq 0$, and since it is penalized it will sit exactly at the larger of the two,

$$\xi_i = \max\{0,\ 1 - y_i f(x_i)\},$$

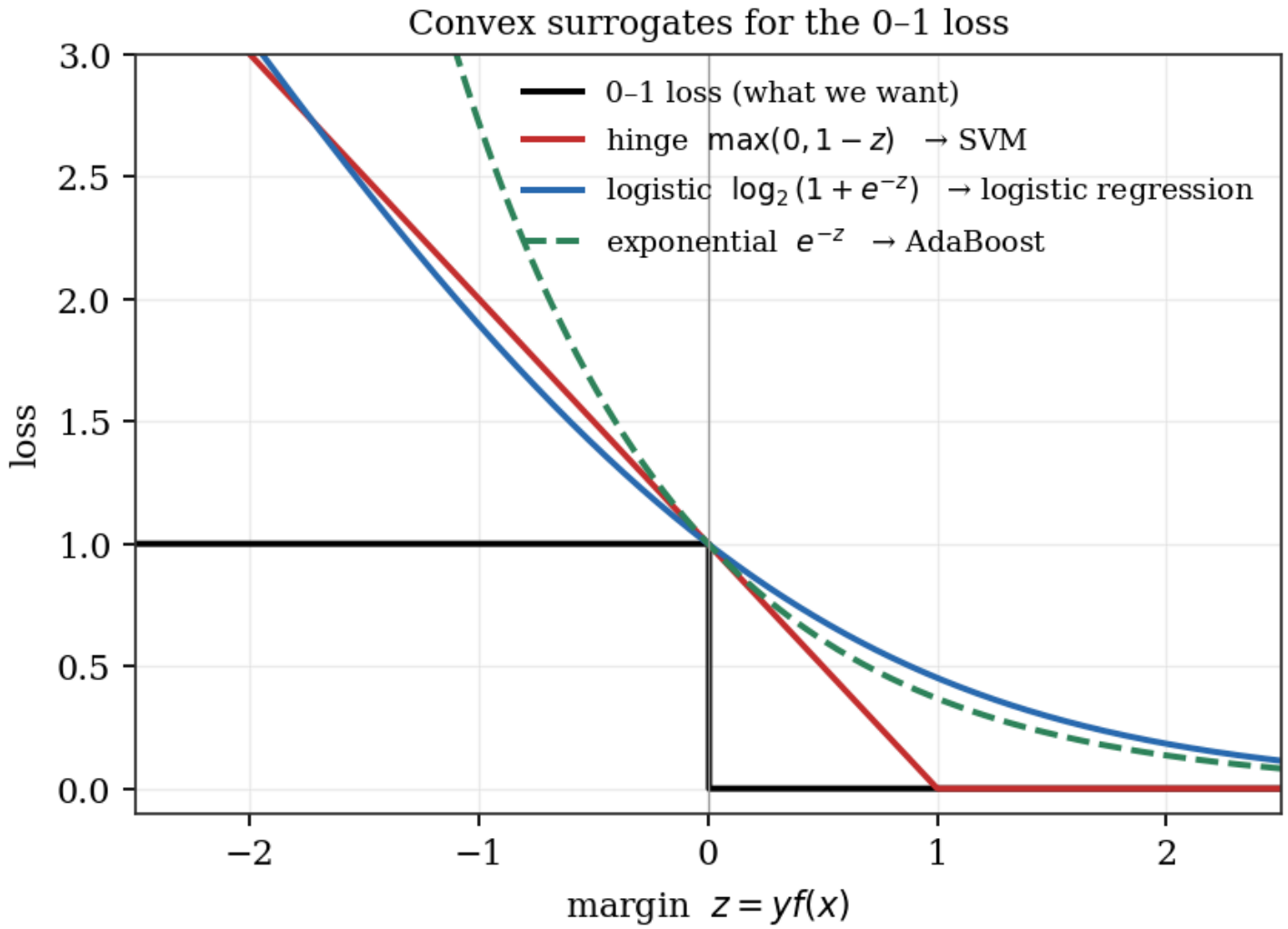
which is precisely the hinge loss. The slack variables were never really extra degrees of freedom — they were the loss function, written as constraints.

This makes explicit the connection between SVMs and empirical risk minimization(ERM), while preserving a geometric interpretation via the margin.

Why a surrogate loss at all?

What we would *like* to minimize is the number of mistakes — the **0–1 loss** $1[yf(x) \leq 0]$. We do not, for a blunt reason: minimizing 0–1 loss over linear classifiers is **NP-hard**. It is non-convex and discontinuous, its gradient is zero everywhere it exists, so there is nothing for an optimizer to descend.

The standard response is to replace it with a convex **surrogate** that upper-bounds it. Then minimizing the surrogate drives the true error down, and convexity means we can actually do the minimizing.



Writing $z = yf(x)$ for the margin, the common choices are:

loss	formula	gives you
0–1	$1[z \leq 0]$	what we want, NP-hard
hinge	$\max(0, 1 - z)$	SVM
logistic	$\log(1 + e^{-z})$	logistic regression
exponential	e^{-z}	AdaBoost

All three surrogates upper-bound the 0–1 loss at $z = 0$ and are convex. What distinguishes the hinge is the **flat region**: it is exactly zero once $z \geq 1$. Points comfortably on the right side of the margin contribute *nothing* — not a small amount, but nothing. That flat piece is the source of sparsity. The logistic and exponential losses are strictly positive everywhere, so under them every single training point keeps a nonzero influence forever, and no support-vector structure emerges.

All of these are also **classification-calibrated**: minimizing them over all measurable functions yields a classifier agreeing in sign with the Bayes-optimal rule. So the surrogate is not merely convenient — it is aiming at the right target.

SVM versus logistic regression

This is the comparison worth internalizing, because the two methods are far more similar than their usual presentations suggest. Both minimize

$$\underbrace{\frac{1}{2}\|w\|_2^2}_{\text{same } L_2 \text{ regularizer}} + C\sum_i \underbrace{\ell\big(y_i f(x_i)\big)}_{\text{the only difference}} .$$

Same linear model, same regularizer, same convex optimization. **The only difference is ℓ.** And essentially every practical difference between the two follows from that one choice:

	SVM (hinge)	**Logistic regression (log)**
solution depends on	support vectors only	every training point
sparse?	yes — the flat region kills most α_i	no
probabilities?	no (needs Platt scaling)	yes, natively
far-away outliers	ignored once correctly classified	still exert force
kernelizes?	naturally, via the dual	possible but loses sparsity
loss at $z = 0$	1	$\log 2 \approx 0.69$

Neither dominates. If you want calibrated probabilities, use logistic regression. If you want a sparse kernel machine and a decision boundary driven by the hard cases near the margin, use the SVM.

5.4. Dual Formulation of the Soft-Margin SVM

Following the same procedure as for the hard-margin but we introduce another lagrange multiplier $\mu_i \geq 0$ for $\xi_i \geq 0$, let's derive it :

I've already introduced the primal problem , now let's construct the primal lagrangian :

$$\mathcal{L}_{primal}(w, b, \xi, \alpha, \mu) = \frac{1}{2}\|w\|_2^2 + C\sum_i \xi_i - \sum_i \alpha_i[y_i(\langle w, x_i\rangle + b) - 1 + \xi_i] - \sum_i \mu_i \xi_i$$

stationarity w.r.t. primal variables

$$\frac{\partial \mathcal{L}}{\partial \xi_i} = C - \alpha_i - \mu_i$$

thus, $C - \alpha_i - \mu_i = 0$ => $\mu_i = C - \alpha_i$

And from knowing that $\mu_i \geq 0$:
$C - \alpha_i \geq 0$ => $\alpha_i \leq C$ => $0 \leq \alpha_i \leq C$

$$\frac{\partial \mathcal{L}}{\partial w} = w - \sum_i \alpha_i x_i y_i$$

$w = \sum_i \alpha_i x_i y_i$

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_i \alpha_i y_i$$

$\sum_i \alpha_i y_i = 0$

Substituting into primal yields the dual

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

Notice that μ_i has vanished entirely. The stationarity condition $\mu_i = C - \alpha_i$ was used to eliminate it, and its only surviving trace is the upper bound $\alpha_i \leq C$ — the dual objective itself is *identical* to the hard-margin one. The whole effect of the slack variables is to put a ceiling on the multipliers.

And finally the dual optimization problem :

$$\max_\alpha \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathcal{K}(x_i, x_j)$$

subject to

$$0 \leq \alpha_i \leq C, \sum_i \alpha_i y_i = 0.$$

Compared to the hard-margin case, the dual variables are now box-constrained.

The three-case KKT characterization

In Section 3.3 complementary slackness split the training points into two groups. With slack variables in play it splits them into three, and this classification is worth memorizing — it is the soft-margin

analogue of the sparsity argument, it is what the SVR picture in Section 6.6 mirrors, and it is literally what SMO checks to decide when to stop.

Recall the two slackness conditions and the relation $\mu_i = C - \alpha_i$:

$$\alpha_i \big[y_i f(x_i) - 1 + \xi_i \big] = 0, \qquad \mu_i \xi_i = 0.$$

Case $\alpha_i = 0$. Then $\mu_i = C > 0$, so $\xi_i = 0$. The first condition is satisfied trivially, leaving only the primal constraint $y_i f(x_i) \geq 1$. The point sits on or outside the margin, correctly classified, and contributes nothing to w.

Case $0 < \alpha_i < C$. Then $\mu_i = C - \alpha_i > 0$, forcing $\xi_i = 0$; and $\alpha_i > 0$ forces the bracket to vanish. Together:

$$y_i f(x_i) = 1.$$

The point lies **exactly on** the margin boundary. These are the *free* or *unbounded* support vectors, and they are the ones we used in Section 5.5 to recover b.

Case $\alpha_i = C$. Then $\mu_i = 0$, so ξ_i is unconstrained and may be positive. The bracket still vanishes, giving $y_i f(x_i) = 1 - \xi_i \leq 1$. The point is inside the margin ($0 < \xi_i \leq 1$) or outright misclassified ($\xi_i > 1$). These are the *bounded* support vectors.

Summarizing:

multiplier	margin	position
$\alpha_i = 0$	$y_i f(x_i) \geq 1$	outside the margin, not a support vector
$0 < \alpha_i < C$	$y_i f(x_i) = 1$	exactly on the margin boundary
$\alpha_i = C$	$y_i f(x_i) \leq 1$	inside the margin, or misclassified

Note that the implications run both ways, which is what makes this a usable stopping test: any α violating these conditions is not optimal.

5.5. Decision Function (Kernelized)

The final classifier has the form

$$f(x) = \text{sign}(\sum_i \alpha_i y_i \mathcal{K}(x_i, x) + b).$$

The bias term b is recovered exactly as in Section 3.3, but now the choice of support vector matters. It must be a point with

$$0 < \alpha_k < C,$$

that is, a support vector lying **exactly on** the margin boundary. Such a point has $\mu_k = C - \alpha_k > 0$, which by complementary slackness ($\mu_k \xi_k = 0$) forces $\xi_k = 0$, so its constraint really is tight and

$$b = y_k - \sum_i \alpha_i y_i \mathcal{K}(x_i, x_k).$$

Both endpoints must be excluded. If $\alpha_k = 0$ the point is not a support vector at all and its constraint may be slack; if $\alpha_k = C$ then $\mu_k = 0$, so ξ_k is unconstrained and possibly nonzero, and the point may sit inside the margin or be outright misclassified. In either case $y_k f(x_k) = 1$ need not hold.

If no α_k falls strictly inside $(0, C)$ — which can genuinely happen for small C — then b is **not unique**. This is the soft-margin failure of uniqueness promised in Sections 1.3 and 2.1: any b in the interval permitted by the KKT conditions is optimal.

5.6. An Aside: the ν-SVM Parameterization

C has an awkward property as a hyperparameter: it is a raw penalty weight with no interpretable scale. Is $C = 10$ large? It depends on n, on the feature scaling, on the kernel. You can only find out by searching.

The **ν-SVM** reparameterizes the same problem so the knob means something. Replace C with $\nu \in (0, 1]$ and solve

$$\min_{w,b,\xi,\rho} \frac{1}{2}\|w\|_2^2 - \nu\rho + \frac{1}{n}\sum_i \xi_i$$

subject to $y_i(\langle w, x_i \rangle + b) \geq \rho - \xi_i$ and $\xi_i \geq 0, \rho \geq 0$. The margin width ρ is now itself a variable being maximized, rather than pinned to 1.

The payoff is a genuine interpretation:

> ν is an **upper bound on the fraction of margin errors** and a **lower bound on the fraction of support vectors**.

So setting $\nu = 0.05$ says "let at most about 5% of my training points violate the margin." That is a statement you can reason about *before* running anything, which C never permits. The two

formulations produce the same set of solutions as their parameters vary — the choice is purely one of which knob you would rather turn. In scikit-learn these are `SVC` and `NuSVC`, and `NuSVR` likewise.

6. Support Vector Regression (SVR)

Prerequisites: §5 — slack variables and the box-constrained dual.
Establishes: the ϵ-insensitive tube, the SVR primal and dual, and how to tune ϵ in practice.

Support Vector Regression extends the maximum-margin principle to real-valued outputs by introducing an ε-insensitive loss. Instead of separating classes, SVR seeks a function that approximates the data while remaining as flat as possible.

6.1. Regression as Function Approximation

Given training data

$$\{x_i, y_i\}_{i=1}^n, x_i \in \mathbb{R}^d, y_i \in \mathbb{R},$$

We consider linear predictors of the form

$$f(x) = \langle w, x \rangle + b.$$

As in classification, flatness is measured by the norm $||w||_2^2$. SVR therefore minimizes $||w||_2^2$ subject to an *error tolerance*.

6.2. ϵ-Insensitive Tube

SVR introduces a tube of radius $\epsilon > 0$ around the regression function

$$|y_i - f(x_i)| \leq \epsilon.$$

Errors inside this tube are ignored(only deviations $> \epsilon$ are penalized) which gives a better robustness to outliers.

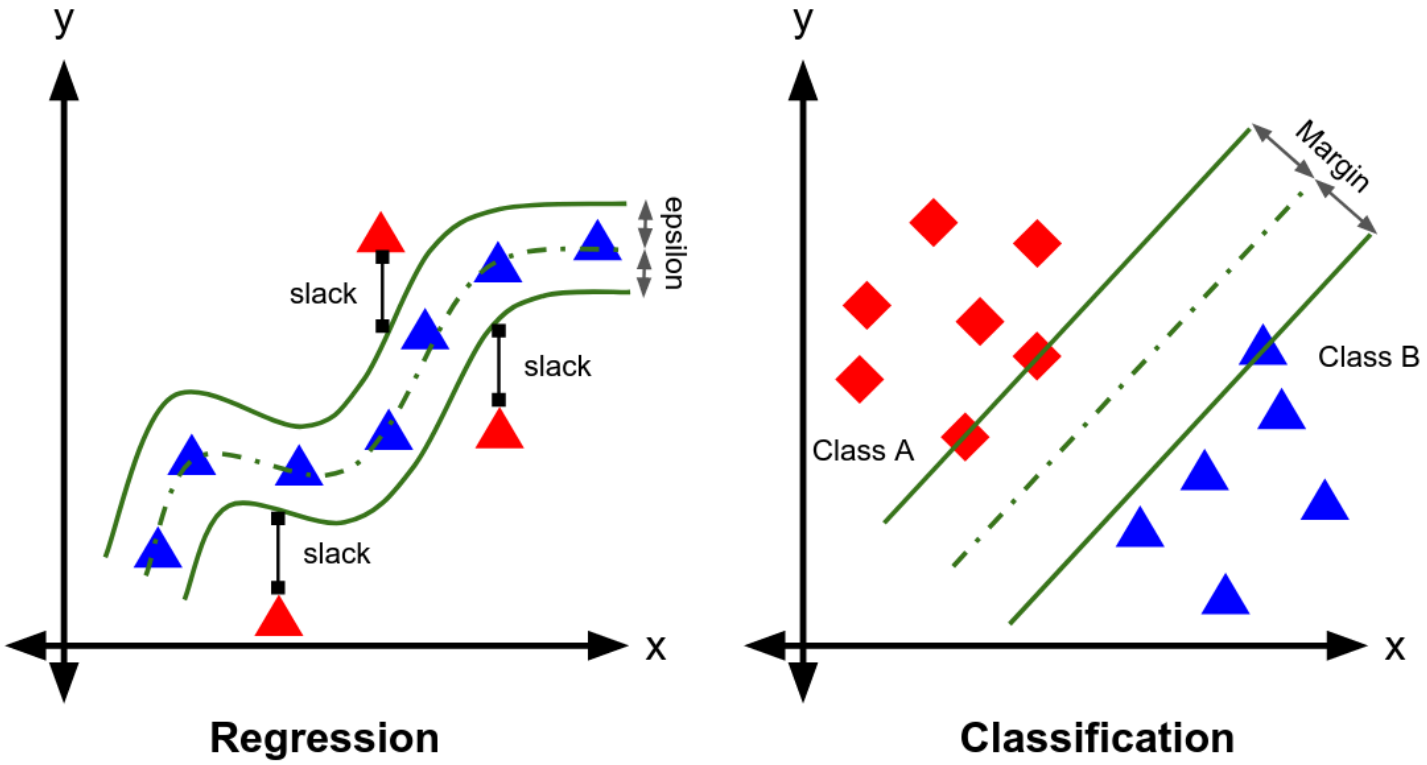
We also include 2 slack variables to allow violations :

- $\xi_i \geq 0$ for points above the tube
- $\xi_i^* \geq 0$ for points below the tube

The constraints become :

$$y_i - (\langle w, x_i \rangle + b) \leq \epsilon + \xi_i,$$

$$(\langle w, x_i \rangle + b) - y_i \leq \epsilon + \xi_i^*.$$



source

6.3. Primal Optimization Problem for SVR

The ϵ-SVR primal problem is:

$$\min_{w,b,\xi,\xi^*} \frac{1}{2} \|w\|_2^2 + C \sum_i (\xi_i + \xi_i^*)$$

subject to :

$$y_i - (\langle w, x_i \rangle + b) \leq \epsilon + \xi_i,$$

$$(\langle w, x_i \rangle + b) - y_i \leq \epsilon + \xi_i^*,$$

$$\xi_i \geq 0, \quad \xi_i^* \geq 0.$$

(These are two separate non-negativity constraints, one per slack variable — *not* a constraint on their product.)

Same as in classification :

- C controls the penalty for deviations outside the tube.
  *The problem is still convex

The ϵ-Insensitive Loss

The primal problem is equivalent to minimizing a regularized empirical risk with ϵ-insensitive loss:

$$\ell_\epsilon(y, f(x)) = \max\{0, |y - f(x)| - \epsilon\}.$$

Thus:

$$\min_{w,b} \frac{1}{2} \|w\|_2^2 + C \sum_i \max\{0, |y_i - (\langle w, x_i \rangle + b)| - \epsilon\}.$$

6.4. Dual Formulation of SVR

Introduce Lagrange multipliers :

- $\alpha_i, \alpha_i^* \geq 0$ for the ϵ-tube constraints,
- $\mu_i, \mu_i^* \geq 0$ for non-negativity of slack variables.

After minimizing the primal Lagrangian w.r.t. w and b , the dual problem becomes:

$$\max_{\alpha,\alpha^*} -\frac{1}{2} \sum_i \sum_j (\alpha_i - \alpha_i^*)(\alpha_j - \alpha_j^*) \mathcal{K}(x_i, x_j) - \epsilon \sum_i (\alpha_i + \alpha_i^*) + \sum_i y_i (\alpha_i - \alpha_i^*)$$

subject to:

$$0 \leq \alpha_i \leq C, \quad 0 \leq \alpha_i^* \leq C, \quad i = 1, ..., n$$

$$\sum_i (\alpha_i - \alpha_i^*) = 0$$

One further fact falls out of complementary slackness and is worth recording, because it explains why the pair (α_i, α_i^*) never fights itself:

$$\alpha_i \alpha_i^* = 0 \quad \text{for every } i.$$

A point cannot lie above the tube *and* below it simultaneously, so at most one of the two constraints can be active, and therefore at most one of the two multipliers can be nonzero.

6.5. Decision Function (SVR)

The regression function is:

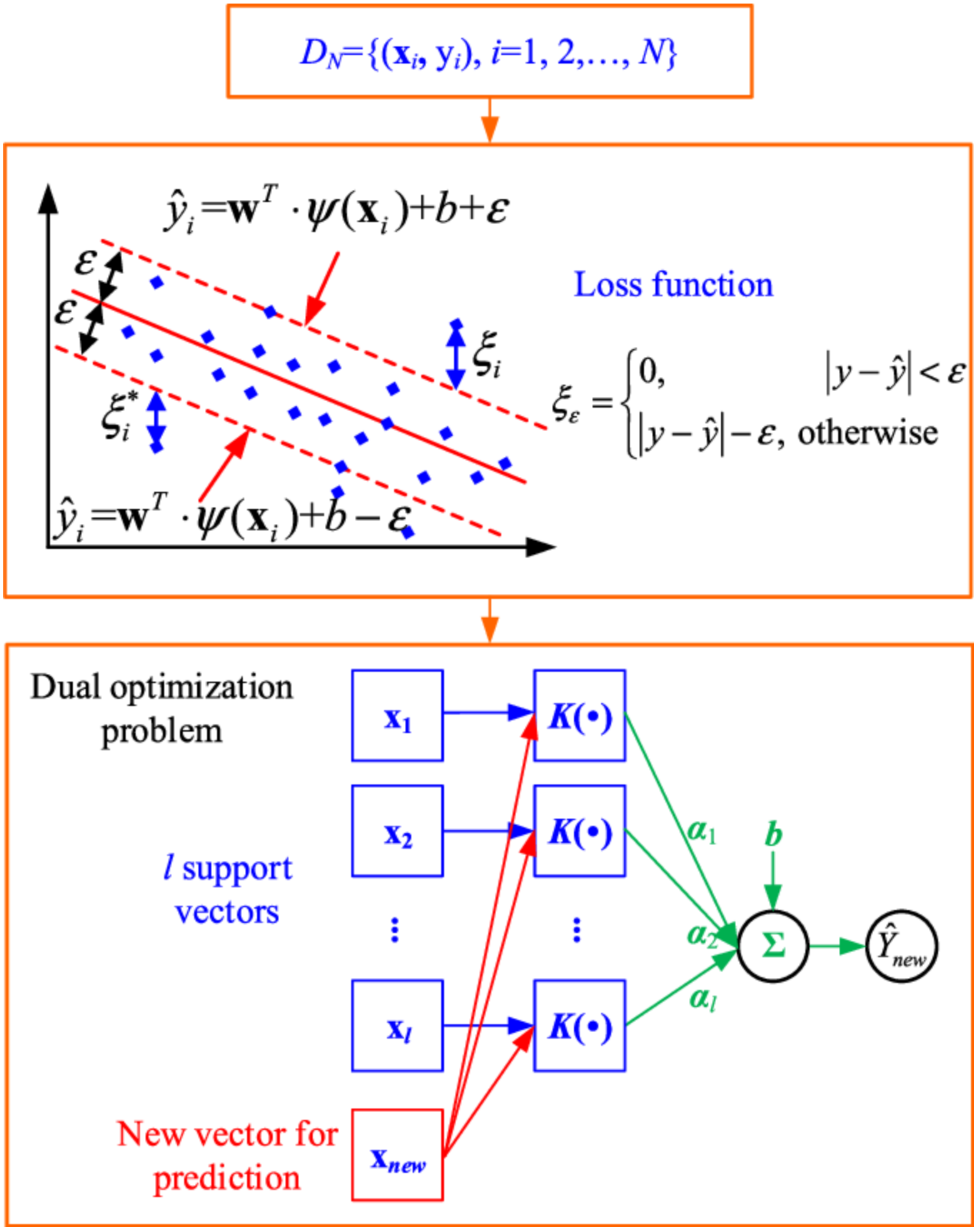
$$f(x) = \sum_i (\alpha_i - \alpha_i^*) \mathcal{K}(x_i, x) + b.$$

Only points with $|\alpha_i - \alpha_i^*| > 0$ contribute (these are the support vectors)

6.6. Geometrical Interpretation

- Points inside the
  ϵ-tube : $\alpha_i = \alpha_i^* = 0$
- Points on the boundary of the tube(supp. vec-s):
  $0 < \alpha_i < C$
  or $0 < \alpha_i^* < C$
- Points outside the tube : $\alpha_i = C$
  or $\alpha_i^* = C$

Thus, as in SVC, SVR yields a sparse output.

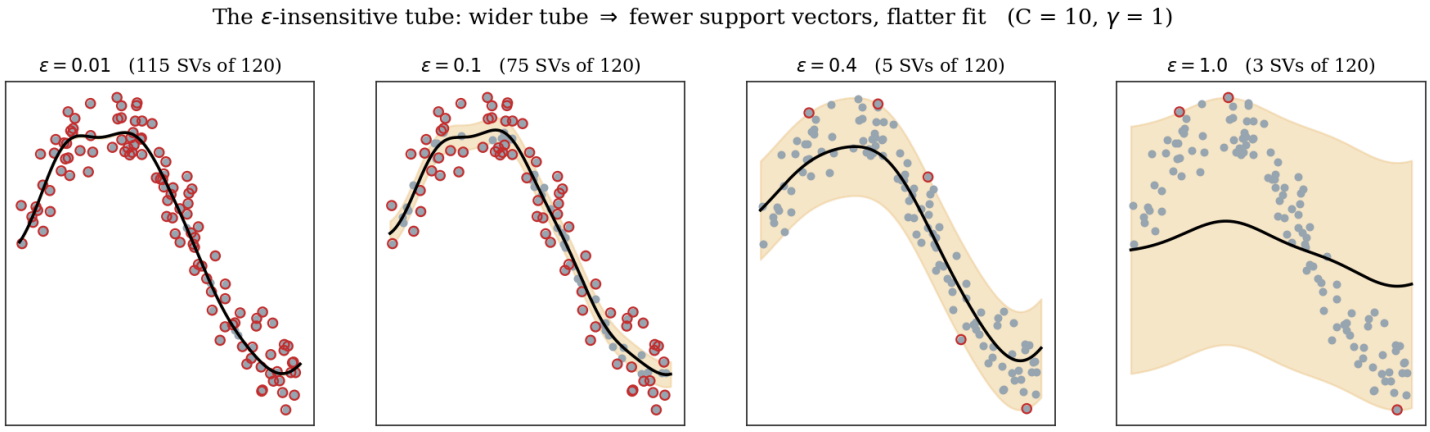

$D_N=\{(\mathbf{x}_i, y_i), i=1, 2,\ldots, N\}$
$\hat{y}_i=\mathbf{w}^T\cdot\psi(\mathbf{x}_i)+b+\varepsilon$
$\hat{y}_i=\mathbf{w}^T\cdot\psi(\mathbf{x}_i)+b-\varepsilon$
Loss function
$\xi_\varepsilon=\begin{cases}0, & |y-\hat{y}|<\varepsilon\\ |y-\hat{y}|-\varepsilon, & \text{otherwise}\end{cases}$
Dual optimization problem
l support vectors
New vector for prediction
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_l$
$\mathbf{x}_{new}$
$K(\bullet)$
α_1
α_2
α_l
b
Σ
$\hat{Y}_{new}$

source

6.7. SVR in Practice

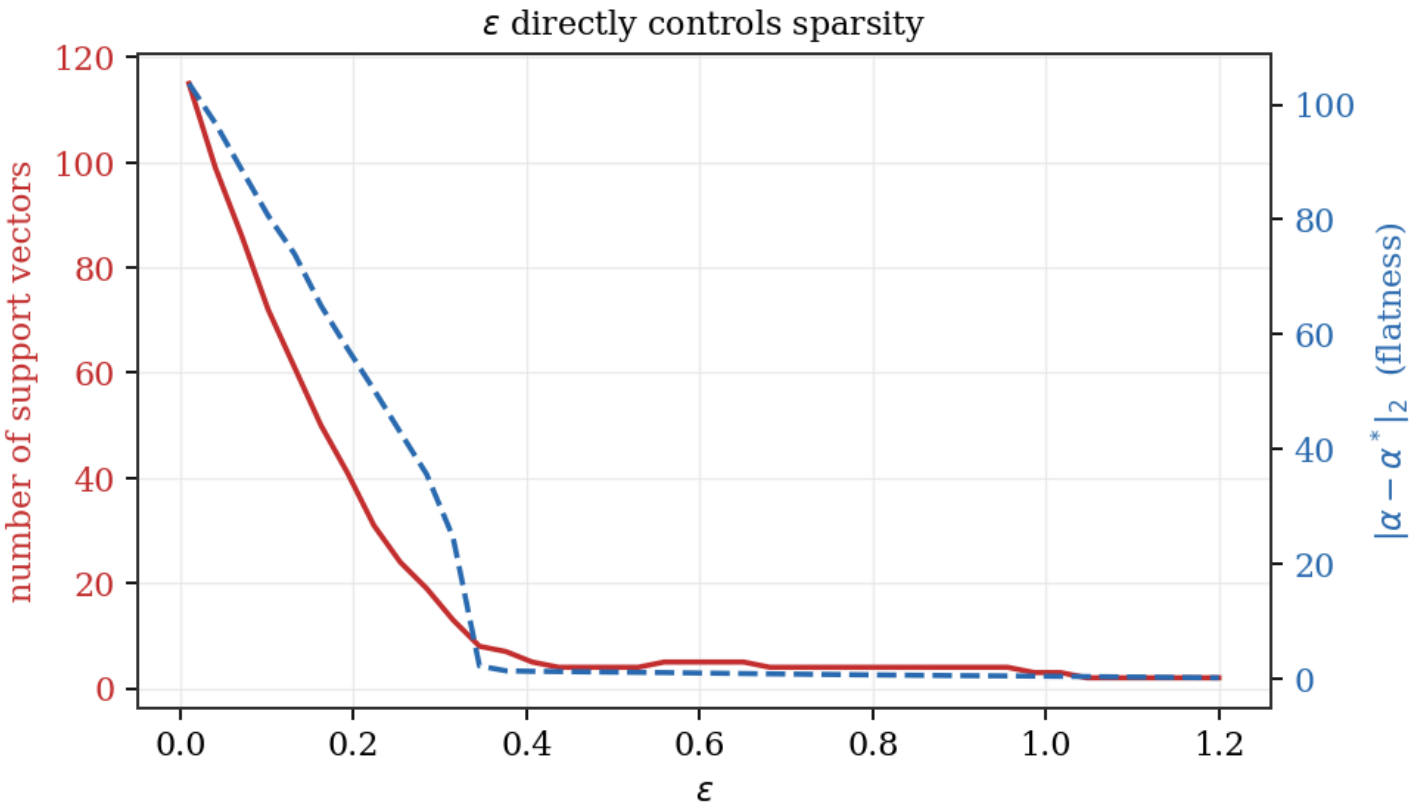
What ϵ actually does

ϵ has no analogue in classification, and it is the parameter people get wrong. Watch what happens as the tube widens on $y = \sin(x) + \text{noise}$:



At $\epsilon = 0.01$ the tube is thinner than the noise, so almost every point sits outside it: **115 of 120 points become support vectors** and the curve chases the noise. At $\epsilon = 1.0$ the tube swallows the entire dataset, only 3 points remain support vectors, and the fit has collapsed to nearly a straight line. Between them, $\epsilon = 0.1$ — comparable to the noise standard deviation of 0.18 — tracks the true signal with 75 support vectors.

The relationship is direct and monotone:



> **ϵ is the sparsity knob.** In classification, sparsity is a by-product you cannot control directly. In SVR you set it explicitly: a wider tube means fewer support vectors, a flatter function, and faster prediction.

The trap: ϵ is in the units of y

This is the mistake that quietly ruins SVR models, and it follows straight from the constraint $|y_i - f(x_i)| \leq \epsilon$ — the tube is measured in **whatever units your target has**.

scikit-learn's default is `epsilon=0.1`. If your target is a house price in the hundreds of thousands, a tube of ± 0.1 dollars is effectively zero width, every point becomes a support vector, and you have thrown away both the robustness and the sparsity that motivated SVR in the first place.

On the diabetes dataset, whose targets run from about 25 to 350:

```
eps= 0.01  R2= 0.350  nSV= 309 of 309
eps= 0.10  R2= 0.351  nSV= 307 of 309     <- the default
eps= 1.00  R2= 0.355  nSV= 305 of 309
eps= 5.00  R2= 0.364  nSV= 290 of 309
eps=20.00  R2= 0.370  nSV= 213 of 309
eps=50.00  R2= 0.387  nSV= 118 of 309
```

At the default, **307 of 309 training points are support vectors** — no sparsity whatsoever, and the worst R^2 in the table. Raising ϵ to 50 both improves accuracy and cuts the model to a third of the size.

Two ways to avoid this. Either scale your target as well as your features (`TransformedTargetRegressor(regressor=SVR(), transformer=StandardScaler())`), after which the default $\epsilon = 0.1$ is at least on a sane scale; or set ϵ to a fraction of your target's standard deviation and tune from there. **Never leave it at the default without checking the scale of y.**

The API

```
from sklearn.svm import SVR, LinearSVR, NuSVR
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import GridSearchCV
import numpy as np

pipe = make_pipeline(StandardScaler(), SVR(kernel="rbf"))

grid = GridSearchCV(
    pipe,
    {
        "svr__C": np.logspace(-1, 3, 9),
        "svr__gamma": np.logspace(-4, 1, 9),
        "svr__epsilon": [0.01, 0.1, 1.0, 10.0],  # scale these to YOUR y
    },
    cv=5,
    n_jobs=-1,
)
grid.fit(X_train, y_train)
```

estimator	use it when
`SVR`	nonlinear kernel, $n \lesssim 10^4$
`LinearSVR`	linear kernel — far faster, scales to large n (as `LinearSVC` does for classification)
`NuSVR`	you would rather set ν (Section 5.6) than ϵ

Attributes map to the theory as in Section 10.1, with one change: `dual_coef_` holds the differences $\alpha_i - \alpha_i^*$ rather than $\alpha_i y_i$, which is why it can be either sign and why $|\alpha_i - \alpha_i^*| > 0$ is the support-vector test from Section 6.5.

Is SVR actually worth using?

Here is an honest comparison on the diabetes dataset (442 samples, 10 features), everything tuned fairly:

```
SVR (rbf, default)     R2= 0.137  MAE= 53.38
SVR (rbf, tuned)       R2= 0.371  MAE= 44.92
Ridge                  R2= 0.392  MAE= 44.68
GradientBoosting       R2= 0.281  MAE= 48.22
```

Two lessons, neither flattering to SVR.

First, **the default is disastrous** — $R^2 = 0.137$ versus 0.371 tuned. An untuned SVR is not a baseline, it is close to noise. Compare this with `RandomForestRegressor`, which is usually respectable out of the box. SVR does not forgive.

Second, **plain Ridge regression beats the tuned SVR here**, and it fits instantly with one hyperparameter. This dataset is small, mostly linear, and low-dimensional — exactly the regime where the kernel machinery buys you nothing.

That is the honest position. SVR earns its place when the relationship is genuinely nonlinear, n is moderate, and you want robustness to outliers (the ϵ-insensitive loss ignores small errors entirely and grows only linearly for large ones, so it is far less outlier-sensitive than squared error). Outside that regime, fit a Ridge and a gradient booster first — and only reach for SVR if they leave something on the table.

7. Solving the Dual in Practice: Sequential Minimal Optimization

Prerequisites: §5.4 — the box-constrained dual and the KKT table.
Establishes: how libsvm and scikit-learn actually solve the QP.

We have derived a quadratic program and said nothing about how anyone solves it. That gap matters, because the answer is neither "call a generic QP solver" nor "do gradient descent" — it is a specific algorithm, **SMO**, and every production SVM you will ever use (libsvm, and therefore scikit-learn's `SVC`) is built on it.

7.1. Why the Obvious Approaches Fail

The dual is

$$\max_{\alpha} \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathcal{K}(x_i, x_j), \qquad 0 \le \alpha_i \le C, \quad \sum_i \alpha_i y_i = 0.$$

A general-purpose QP solver needs the full Gram matrix, which is $n \times n$. At $n = 50{,}000$ that is 2.5×10^9 doubles — 20 GB. Before any arithmetic happens, you are out of memory.

The natural fallback is coordinate ascent: optimize one α_i at a time. But the equality constraint forbids it. If every other multiplier is fixed, then

$$\alpha_k y_k = -\sum_{i \neq k} \alpha_i y_i$$

is **completely determined**. There is no freedom left to optimize. One variable cannot move alone.

7.2. The Idea: Move Exactly Two

Platt's insight (1998) was that two is the smallest number of multipliers you *can* move, so move exactly two and solve that subproblem in closed form.

Pick α_1, α_2 and hold the rest fixed. The equality constraint becomes

$$\alpha_1 y_1 + \alpha_2 y_2 = -\sum_{i \geq 3} \alpha_i y_i =: \zeta \quad (\text{a constant}).$$

Multiplying by y_1 and using $y_1^2 = 1$ gives $\alpha_1 = (\zeta - \alpha_2 y_2) y_1$, so α_1 is an affine function of α_2. Substituting into the dual leaves a **one-dimensional quadratic in α_2**, subject to the box $0 \le \alpha_2 \le C$.

A one-dimensional quadratic on an interval has an analytic solution. No solver, no iteration — just algebra.

7.3. The Update

Geometrically, the pair is confined to a line segment: the line $\alpha_1 y_1 + \alpha_2 y_2 = \zeta$ intersected with the box $[0, C]^2$. That segment gives bounds $L \le \alpha_2 \le H$, and the two cases depend on whether the labels agree:

$$y_1 \neq y_2: \quad L = \max(0,\ \alpha_2 - \alpha_1), \qquad H = \min(C,\ C + \alpha_2 - \alpha_1)$$

$$y_1 = y_2: \quad L = \max(0,\ \alpha_1 + \alpha_2 - C), \qquad H = \min(C,\ \alpha_1 + \alpha_2)$$

Let $E_i = f(x_i) - y_i$ be the prediction error on point i, and define

$$\eta = 2\mathcal{K}(x_1, x_2) - \mathcal{K}(x_1, x_1) - \mathcal{K}(x_2, x_2).$$

η is the second derivative of the objective along the constraint line, and it is ≤ 0 whenever $\mathcal{K}$ is PSD — the objective is concave along that line, so the stationary point is a maximum. The unconstrained optimum is

$$\alpha_2^{\text{new}} = \alpha_2 - \frac{y_2(E_1 - E_2)}{\eta},$$

which we then clip back into the feasible segment:

$$\alpha_2^{\text{clipped}} = \begin{cases} H, & \alpha_2^{\text{new}} > H \\ \alpha_2^{\text{new}}, & L \leq \alpha_2^{\text{new}} \leq H \\ L, & \alpha_2^{\text{new}} < L \end{cases}$$

and recover the partner from the equality constraint:

$$\alpha_1^{\text{new}} = \alpha_1 + y_1 y_2 \left(\alpha_2 - \alpha_2^{\text{clipped}}\right).$$

The bias b is then re-derived from whichever of the two updated points is a free support vector, exactly as in Section 5.5.

7.4. Choosing the Pair, and Stopping

The remaining question is which pair to pick, and here the KKT table from Section 5.4 does the work. Optimality is *equivalent* to every point satisfying its case, so SMO simply looks for a point that **violates** its KKT condition — those are the only points whose multipliers can profitably move. The standard heuristic:

- outer loop: iterate over points violating KKT (preferring free support vectors, $0 < \alpha_i < C$, since those change most often);
- inner loop: given α_1, choose α_2 maximizing $|E_1 - E_2|$, the crude proxy for largest step size.

Stop when all points satisfy KKT to within a tolerance (typically 10^{-3}).

Why this is fast

Two reasons, and both are about memory rather than arithmetic.

First, **no Gram matrix is ever formed**. Each step needs a handful of kernel evaluations, computed on demand and cached. Memory is $O(n)$ plus whatever cache you allow, not $O(n^2)$.

Second, **sparsity compounds**. Most α_i end at exactly 0 and never move again, so as the algorithm converges the set of points it has to think about shrinks toward the support vectors alone.

Empirically SMO scales somewhere between $O(n)$ and $O(n^{2.3})$ depending on C and how separable the data is — far better than the $O(n^3)$ of a dense QP solve, though as Section 9 discusses, still not enough to make SVMs a large-n method.

8. Multiclass SVMs

Prerequisites: any binary formulation.
Establishes: one-vs-rest, one-vs-one, and which one scikit-learn uses where.

Everything so far has been binary — the labels $y_i \in \{-1, +1\}$ were baked into the geometry from the first line. Real problems usually have more than two classes, and the SVM has no native multiclass formulation. The standard approach is to decompose the problem into binary ones.

8.1. One-vs-Rest (OvR)

Train K classifiers, one per class, each separating class k from all others combined. Predict with whichever has the largest decision value:

$$\hat{y}(x) = \arg\max_k \; f_k(x).$$

Cost: K classifiers, each on all n points.

The catch: each binary problem is artificially imbalanced (1 class against $K-1$), and the f_k are trained independently, so their decision values are **not calibrated against each other**. Taking an $\arg\max$ over scores from separately-trained models is theoretically shaky — it works in practice more often than it deserves to.

8.2. One-vs-One (OvO)

Train a classifier for every *pair* of classes — $\binom{K}{2}$ of them — and let them vote, the class with the most pairwise wins taking it.

Cost: $\binom{K}{2}$ classifiers, which sounds worse, but each trains on only the points belonging to its two classes. Since SVM training is superlinear in n, the total work is usually **less** than OvR. For balanced classes each subproblem sees $2n/K$ points, and $\binom{K}{2}$ problems of size $(2n/K)^2$ totals $O(n^2)$ — versus OvR's K problems of size n^2.

The catch: ties in the voting, resolved by falling back on summed decision values.

8.3. What scikit-learn Actually Does

This trips people up constantly, so it is worth stating plainly:

estimator	strategy
`SVC`	**one-vs-one** — despite `decision_function_shape='ovr'`, which only *reshapes* the OvO output
`NuSVC`	one-vs-one
`LinearSVC`	**one-vs-rest**

So `SVC` and `LinearSVC` do not merely differ in their solver — they solve genuinely different multiclass problems and can disagree on the same data. If you switch between them and your accuracy shifts, this is often why.

There also exist true multiclass formulations (Crammer–Singer, available as `LinearSVC(multi_class='crammer_singer')`) that optimize a single joint objective. They are more principled and rarely worth it in practice — comparable accuracy, noticeably slower.

9. Computational Cost, and When SVMs Stop Being the Right Tool

Prerequisites: none.
Establishes: complexity, why prediction cost grows with n, and when to choose a different model.

An article that derives the theory without stating the cost leaves the reader to discover it at 100,000 rows. So:

9.1. Complexity

resource	cost
training time	between $O(n^2 d)$ and $O(n^3 d)$ for kernel SVMs
training memory	$O(n^2)$ if the Gram matrix is materialized; $O(n \cdot \text{cache})$ with SMO
prediction time	$O(n_{SV} \cdot d)$ per sample

resource	cost
model size	$O(n_{SV} \cdot d)$

The quadratic-to-cubic training cost is the headline. Doubling your data multiplies training time by roughly four to eight. This is not an implementation detail to be engineered away — it follows from the kernel matrix being $n \times n$, i.e. from the structure of the method itself.

`LinearSVC` is the exception. With a linear kernel there is no need to touch pairwise kernel values at all: the problem can be solved in the primal directly in $O(nd)$, which is why `LinearSVC` handles datasets that would make `SVC` hang indefinitely. If your kernel is linear, never use `SVC`.

9.2. Prediction Cost Scales With Support Vectors

A subtlety worth flagging: **the number of support vectors typically grows linearly with n.** Since prediction cost is proportional to n_{SV}, a kernel SVM trained on more data is not only slower to train but permanently slower to *predict*. A model with 40,000 support vectors requires 40,000 kernel evaluations per prediction.

This is why "sparsity" deserves an asterisk. The solution is sparse relative to a dense representation, but n_{SV} is often 20–50% of the training set — not the handful of points the textbook diagrams suggest. Low C and poorly-chosen γ both inflate it further.

9.3. When Not to Use an SVM

Being honest about this is more useful than advocacy:

Reach for something else when:

- $n \gtrsim 100{,}000$ — training becomes impractical. Use `LinearSVC`, SGD with hinge loss, or gradient boosting.
- **you need calibrated probabilities** — SVMs do not produce them, and the workaround (Section 10.5) is unsatisfying.
- **you need interpretability** — a kernel SVM's decision function is a weighted sum over thousands of training points. There is nothing to report as feature importance.
- **your data is heterogeneous tabular** — mixed categorical and numeric with missing values. Gradient boosting (XGBoost, LightGBM) wins on this category so reliably that it is the default choice.
- **you have many classes** — the cost of $\binom{K}{2}$ or K binary problems compounds.

SVMs remain genuinely strong when:

- n is small to moderate (hundreds to tens of thousands) and d is large — text classification with TF-IDF features being the classic case, where $d \gg n$ and a linear SVM is close to unbeatable.
- the decision boundary is genuinely nonlinear and you lack the data to train a neural network.
- you need a strong, low-variance baseline with few hyperparameters to tune.

The honest summary: SVMs dominated applied machine learning from roughly 1995 to 2010 and were displaced not because the theory was wrong — it remains among the most elegant in the field — but because gradient boosting proved better on tabular data and deep networks proved better on perceptual data, and both scale to n where SVMs simply cannot follow.

10. Implementing an SVM from Scratch (Soft-Margin)

Prerequisites: §5.3 — the unconstrained hinge-loss form.
Note: this is a teaching implementation using subgradient descent on the primal. Production solvers use SMO on the dual — see §7.

Requirements : Python 3.8+, NumPy, matplotlib, scikit-learn

10.1. Core Algorithm

```
# svm.py <-----
import numpy as np


class SVM:
    def __init__(self, C=1.0):
        # C = hyperparameter for penalty
        self.C = C
        self.w = None
        self.b = 0.0

    def hingeloss(self, w, b, x, y):
        # Regularization term (L_2).
        # NOTE: w @ w.T is the squared NORM. Writing (w * w) would multiply
        # elementwise and leave you with a vector, not a scalar.
        reg = 0.5 * float(w @ w.T)

        # The hinge term is a SUM over every sample, so it has to accumulate.
        total = 0.0
        for i in range(x.shape[0]):
            # Optimization term
            opt_term = y[i] * (np.dot(w, x[i]) + b)

            # calculating loss
            total += max(0, 1 - float(opt_term))

        return reg + self.C * total

    def fit(self, X, Y, batch_size=100, learning_rate=0.001, epochs=1000):

        number_of_features = X.shape[1]
        number_of_samples = X.shape[0]

        c = self.C

        # Creating ids from 0 to number_of_samples - 1
        ids = np.arange(number_of_samples)

        # Random shuffle of samples
        np.random.shuffle(ids)
```

```
        w = np.zeros((1, number_of_features))
        b = 0
        losses = []

        # Gradient Descent logic
        for i in range(epochs):
            # Calculating the Hinge Loss
            l = self.hingeloss(w, b, X, Y)
            losses.append(l)

            for batch_initial in range(0, number_of_samples, batch_size):
                grad_w = 0
                grad_b = 0

                for j in range(batch_initial, batch_initial + batch_size):
                    if j < number_of_samples:
                        x = ids[j]
                        ti = Y[x] * (np.dot(w, X[x].T) + b)

                        if ti > 1:
                            grad_w += 0
                            grad_b += 0
                        else:
                            # Calculating the gradients
                            grad_w += c * Y[x] * X[x]
                            grad_b += c * Y[x]

                # Updating weights and bias
                w = w - learning_rate * w + learning_rate * grad_w
                b = b + learning_rate * grad_b

        # Store the result only AFTER every epoch has run.
        # Keeping these inside the epoch loop would return after a single pass.
        self.w = w
        self.b = b

        return self.w, self.b, losses

    def predict(self, X):
        prediction = np.dot(X, self.w[0]) + self.b
        return np.sign(prediction)
```

You may be confused why we update w and b like that . But the reason is quite simple , let learning rate = η and our cost function (objective func.)

$$\mathcal{J}(w, b) = \frac{1}{2} \|w\|_2^2 + \text{hingeloss}(w, b)$$

Also let

$$\ell_i = \max(0, 1 - y_i f_i)$$

that's the loss per sample,

$$\text{hingeloss}$$

be the loss over the whole dataset. Taking the gradients with respect to w and b yields:

$$\nabla_w \mathcal{J} = w + \nabla_w \, \text{hingeloss}$$

and

$$\nabla_b \mathcal{J} = \nabla_b \, \text{hingeloss}$$

Also taking the gradients of hingeloss per sample w.r.t. w and b gives

$$\nabla_w \ell_i = \frac{\partial}{\partial w} \max\left[0,\ 1 - y_i(\langle w, x_i \rangle + b)\right] = \begin{cases} -y_i x_i, & \text{if } y_i f_i < 1 \\ 0, & \text{if } y_i f_i > 1 \end{cases}$$

and

$$\nabla_b \ell_i = \frac{\partial}{\partial b} \max\left[0,\ 1 - y_i(\langle w, x_i \rangle + b)\right] = \begin{cases} -y_i, & \text{if } y_i f_i < 1 \\ 0, & \text{if } y_i f_i > 1 \end{cases}$$

Two things deserve emphasis here.

First, the hinge loss is applied to the **raw score** $f_i = \langle w, x_i \rangle + b$, never to $\text{sign}(f_i)$. This is not a cosmetic distinction: sign is flat everywhere it is differentiable, so its gradient is zero almost everywhere and no learning could take place at all. The margin has to be measured on a continuous scale for gradient descent to have anything to descend.

Second, at $y_i f_i = 1$ the loss has a kink and is not differentiable. Any value in $[-y_i x_i,\ 0]$ is a valid *subgradient* there, and we simply pick one — this is subgradient descent, not gradient descent, though in practice the distinction rarely bites.

Summing over the violating samples only:

$$\nabla_w \text{hingeloss} = -C \sum_{i\,:\,y_i f_i < 1} y_i x_i$$

and

$$\nabla_b \text{hingeloss} = -C \sum_{i\,:\,y_i f_i < 1} y_i$$

That restriction to violating samples is exactly what the `if ti > 1` branch in the code above implements: points already outside the margin contribute nothing to the update.

We defined negatives of those grads as grad_w and grad_b respectively.
Remembering the grad descent update rule :

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{J}$$

Substituting our variables of interest and their gradients gives us :

$$w \leftarrow w - \eta(w + \nabla_w \,\text{hingeloss})$$

$$w \leftarrow w - \eta w - \eta(-C \sum_i y_i x_i)$$

$$w \leftarrow w - \eta w + \eta \text{gradw}$$

and

$$b \leftarrow b - \eta \nabla_b \,\text{hingeloss}$$

$$b \leftarrow b + \eta \text{gradb}$$

10.2. Creating a Dataset

```
import numpy as np
from sklearn import datasets
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

np.random.seed(42)  # fit() shuffles, so seed it if you want reproducibility

X, y = datasets.make_blobs(
    n_samples=100, n_features=2, centers=2, cluster_std=1, random_state=67
)

# Cause binary classification
y = np.where(y == 0, -1, 1)
# Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.5, random_state=228
)
```

10.3. Training

```
from svm import SVM

svm = SVM()
w, b, losses = svm.fit(X_train, y_train)
```

10.4. Predictions and Evaluation

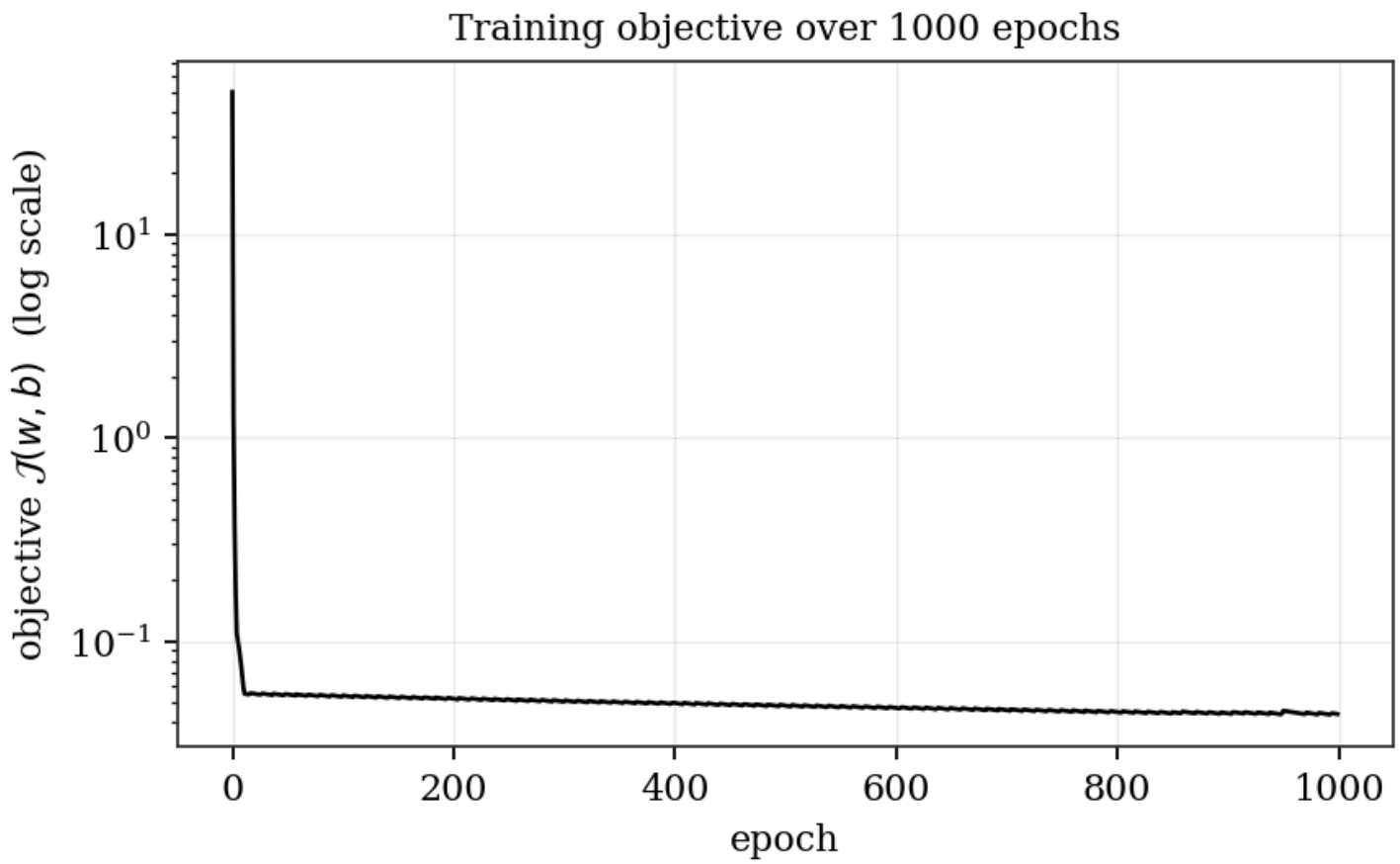
```
preds = svm.predict(X_test)

print("Epochs recorded:", len(losses))
print("Loss:", losses[-1])
print("Accuracy:", accuracy_score(preds, y_test))
print("w, b:", [w, b])
```

Output[1] :

```
Epochs recorded: 1000
Loss: 0.044000289488343516
Accuracy: 1.0
w, b: [array([[ 0.17396288, -0.23991995]]), 0.11900000000000008]
```

Note `losses[-1]` rather than `losses.pop()` — `pop` mutates the list, which is a nuisance if you then want to plot the curve. And that curve is worth a look, because it is the cheapest possible check that the optimizer is actually optimizing:

```
import matplotlib.pyplot as plt

plt.plot(losses)
plt.yscale("log")
plt.xlabel("epoch")
plt.ylabel("objective")
plt.show()
```



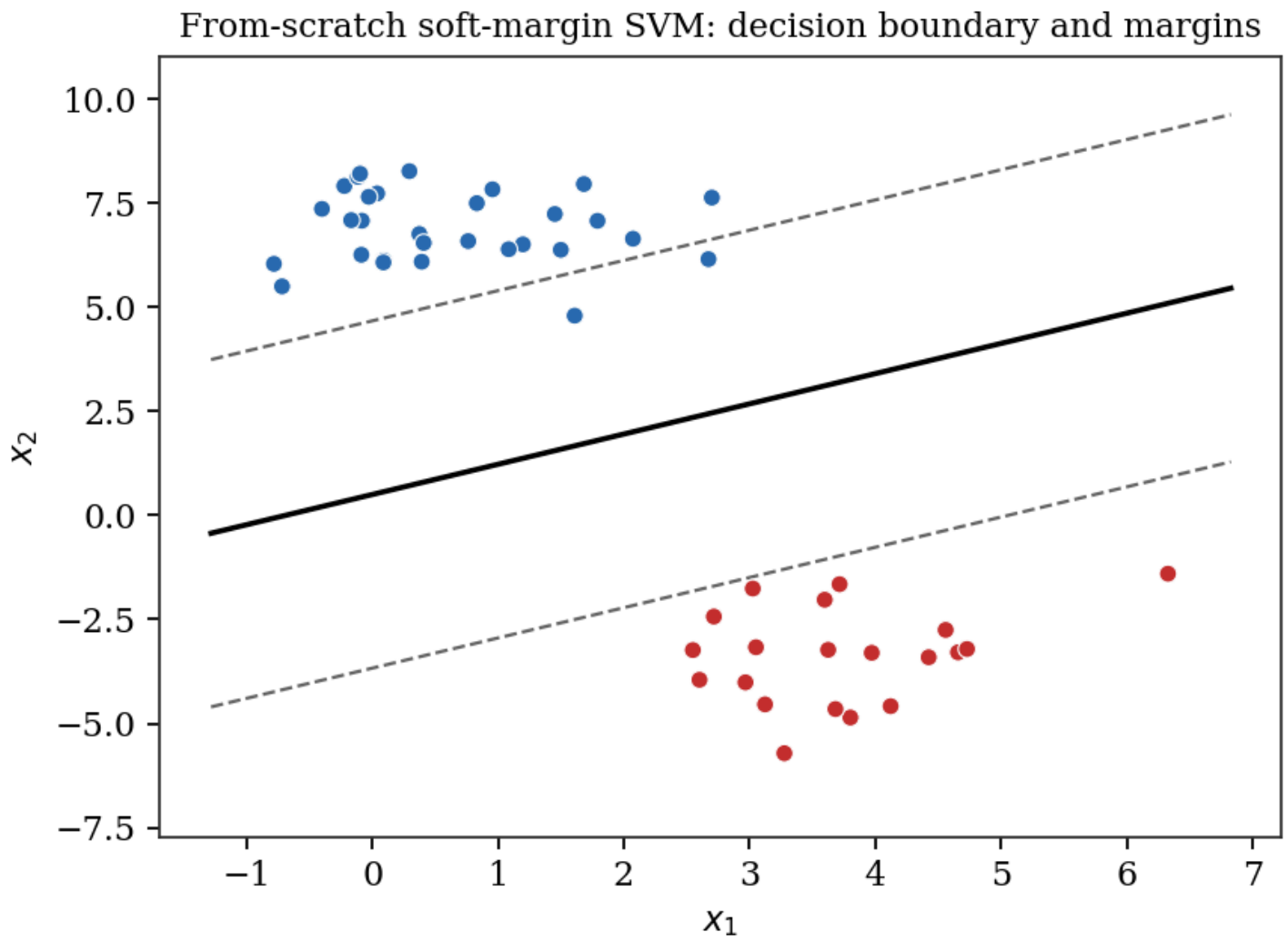
The objective falls from 50.0 to 0.044 over the 1000 epochs. It is not perfectly monotone — we are taking mini-batch subgradient steps, not exact

gradient steps, so small upticks are expected and harmless.

10.5. Visualizing the Result

```
import matplotlib.pyplot as plt


def visualize_svm():
    def get_hyperplane_value(x, w, b, offset):
        # From  w0*x0 + w1*x1 + b = offset  we get
        #       x1 = (offset - b - w0*x0) / w1
        # so b is SUBTRACTED. (Our predict() uses +b; snippets written for a
        # `np.dot(x, w) - b` convention will have this sign flipped.)
        return (-w[0][0] * x - b + offset) / w[0][1]

    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    plt.scatter(X_test[:, 0], X_test[:, 1], marker="o", c=y_test)

    x0_1 = np.amin(X_test[:, 0])
    x0_2 = np.amax(X_test[:, 0])

    x1_1 = get_hyperplane_value(x0_1, w, b, 0)
    x1_2 = get_hyperplane_value(x0_2, w, b, 0)

    x1_1_m = get_hyperplane_value(x0_1, w, b, -1)
    x1_2_m = get_hyperplane_value(x0_2, w, b, -1)

    x1_1_p = get_hyperplane_value(x0_1, w, b, 1)
    x1_2_p = get_hyperplane_value(x0_2, w, b, 1)

    ax.plot([x0_1, x0_2], [x1_1, x1_2], "y--")
    ax.plot([x0_1, x0_2], [x1_1_m, x1_2_m], "k")
    ax.plot([x0_1, x0_2], [x1_1_p, x1_2_p], "k")

    x1_min = np.amin(X[:, 1])
    x1_max = np.amax(X[:, 1])
    ax.set_ylim([x1_min - 3, x1_max + 3])

    plt.show()


visualize_svm()
```

Output[2] :



10.6. Checking Our Work Against scikit-learn

Plotting a plausible-looking line is not evidence that the implementation is correct. The real test is whether it finds the same hyperplane as a mature solver optimizing the same objective. `LinearSVC` with `loss='hinge'` minimizes exactly the objective we derived, so it makes a fair reference:

```python
from sklearn.svm import LinearSVC

ref = LinearSVC(C=1.0, loss="hinge", max_iter=100000).fit(X_train, y_train)

cos = (w[0] @ ref.coef_[0]) / (np.linalg.norm(w[0]) * np.linalg.norm(ref.coef_[0]))

print("ours    w, b:", np.round(w[0], 4), round(float(b), 4))
print("sklearn w, b:", np.round(ref.coef_[0], 4), round(float(ref.intercept_[0]), 4))
print("cosine similarity of normals:", round(float(cos), 6))
print(
    "accuracy  ours:",
    accuracy_score(svm.predict(X_test), y_test),
    " sklearn:",
    accuracy_score(ref.predict(X_test), y_test),
)
```

```
ours    w, b: [ 0.174  -0.2399] 0.119
sklearn w, b: [ 0.1942 -0.2335] 0.0874
cosine similarity of normals: 0.997796
accuracy  ours: 1.0  sklearn: 1.0
```

The two normal vectors have cosine similarity **0.9978** — they point in essentially the same direction, which is what actually defines the hyperplane's orientation. The magnitudes and biases differ slightly because the two optimizers take different paths and neither is run to machine precision, but the geometry agrees.

This is the check worth running whenever you implement a method from scratch: not "does the output look reasonable" but "does it agree with a known-correct implementation of the same objective."

11. SVMs in Practice with scikit-learn

Prerequisites: none for the recipes; §5 to understand the knobs.
Establishes: the API-to-theory mapping, feature scaling, tuning C and γ, probabilities, and class imbalance.

The from-scratch model above is a teaching device. Nobody uses subgradient descent on the primal in production — as Section 7 explained, real implementations solve the dual with SMO. This section

covers what you actually need to use them well.

11.1. Mapping the API Back to the Mathematics

Every attribute of a fitted `SVC` corresponds to something we derived. This mapping is the whole point of having read the theory:

```
from sklearn.svm import SVC
from sklearn import datasets

X, y = datasets.make_moons(n_samples=200, noise=0.28, random_state=7)
clf = SVC(kernel="rbf", C=1.0, gamma=1.0).fit(X, y)

print(
    "n_support_           :",
    clf.n_support_,
    "total",
    clf.n_support_.sum(),
    "of",
    len(y),
)
print("support_vectors_     :", clf.support_vectors_.shape)
print("dual_coef_ shape     :", clf.dual_coef_.shape)
print(
    "dual_coef_ range     : [%.4f, %.4f]" % (clf.dual_coef_.min(), clf.dual_coef_.max())
)
print("intercept_ (b)       :", clf.intercept_)
print("decision_function[:3]:", clf.decision_function(X[:3]).round(4))
print("predict[:3]          :", clf.predict(X[:3]))
```

```
n_support_           : [30 32] total 62 of 200
support_vectors_     : (62, 2)
dual_coef_ shape     : (1, 62)
dual_coef_ range     : [-1.0000, 1.0000]
intercept_ (b)       : [-0.109891]
decision_function[:3]: [-0.6325  1.4807 -0.5467]
predict[:3]          : [0 1 0]
```

Reading that against the theory:

attribute	what it is
`support_vectors_`	the x_i with $\alpha_i > 0$ — Section 3.3
`dual_coef_`	the products $\alpha_i y_i$, **not** α_i alone (hence the sign)
`intercept_`	our b, recovered as in Section 5.5
`decision_function(x)`	$\sum_i \alpha_i y_i \mathcal{K}(x_i, x) + b$ — the value *before* sign
`predict(x)`	sign of the above, relabelled to $\{0, 1\}$
`n_support_`	support vector count per class

Two things this makes concrete. First, `dual_coef_` is bounded by C in absolute value — that is the box constraint $0 \le \alpha_i \le C$ appearing directly in the API. In this fit 53 of the 62 support vectors sit exactly at $|\alpha_i| = C = 1$ (bounded SVs, inside the margin) and only 9 are free ($0 < \alpha_i < C$, exactly on the boundary) — precisely the three-case split of Section 5.4.

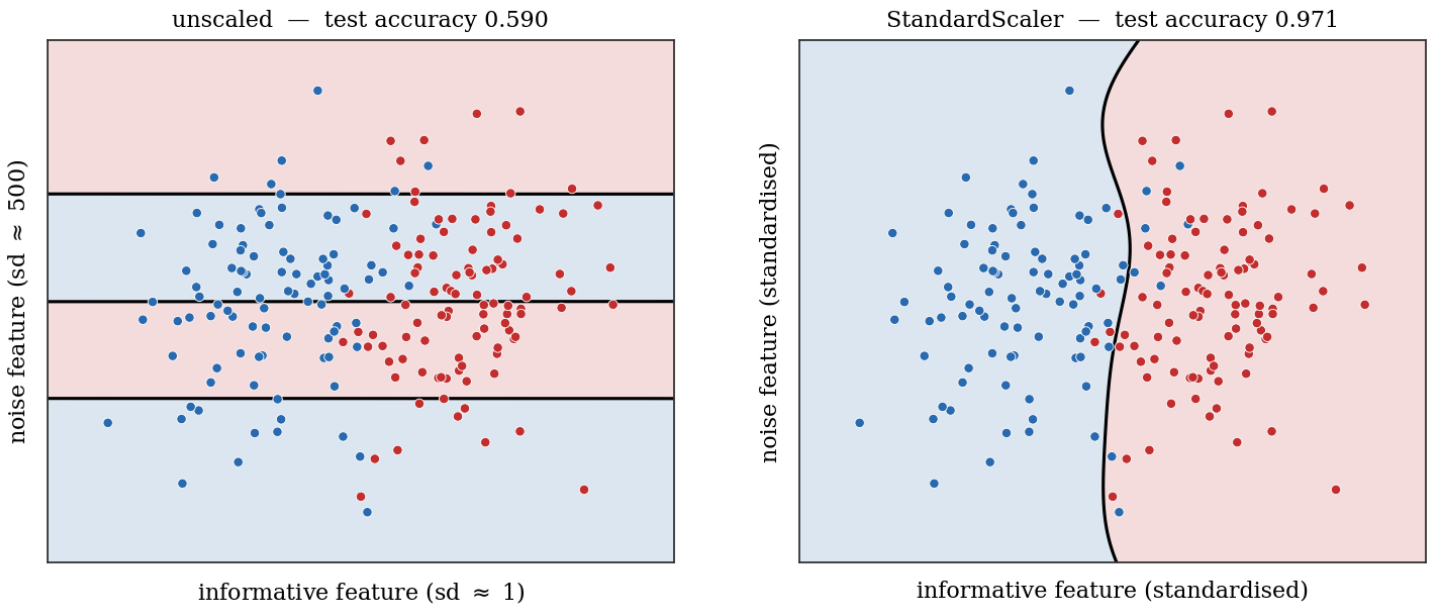
Second, note that **62 of 200 points are support vectors** — 31%. This is the asterisk on "sparsity" from Section 9.2: it is real, but far from the two or three points textbook diagrams suggest.

If you need the margin rather than the label — for ranking, thresholding, or ROC curves — always use `decision_function`, never `predict`.

11.2. Scale Your Features. Always.

This is the single most common way to silently ruin an SVM, and it follows directly from the theory: the margin is a *distance*, and the RBF kernel $\exp(-\gamma \|x - z\|^2)$ is a function of a *distance*. If one feature ranges over $[0, 1]$ and another over $[0, 10^5]$, the second one entirely determines every pairwise distance and the first becomes invisible.

Here is the effect, on data where the informative feature has standard deviation ≈ 1 and an uninformative noise feature has standard deviation ≈ 500:




```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

raw = SVC(kernel="rbf").fit(X_train, y_train)
scaled = make_pipeline(StandardScaler(), SVC(kernel="rbf")).fit(X_train, y_train)


unscaled : 0.590
scaled   : 0.971
```

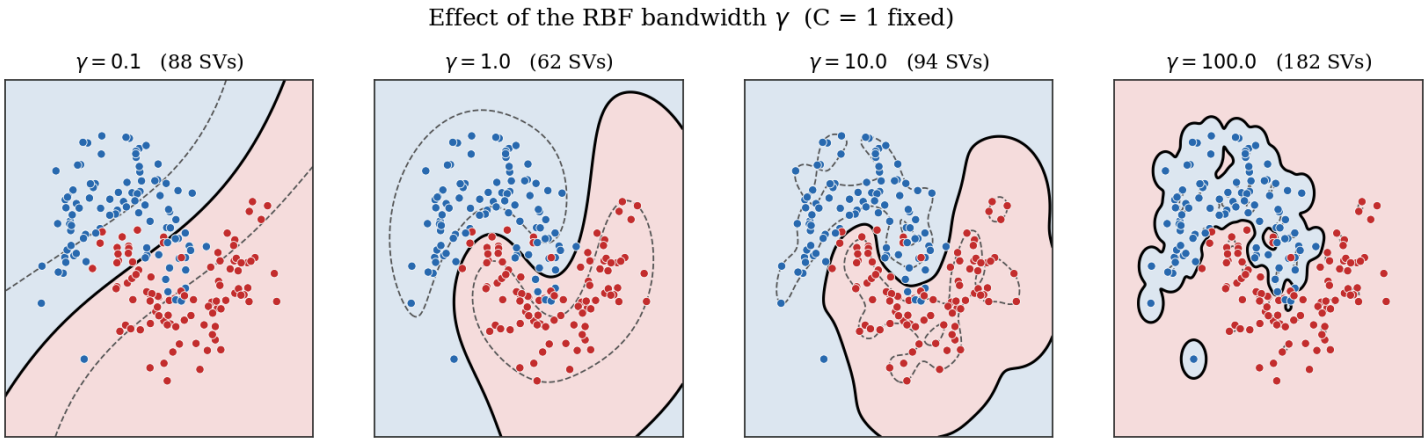
From 59% to 97% — the difference between useless and excellent, on identical data with identical hyperparameters. Note also that the unscaled model is barely above the 50% chance baseline: it is not slightly degraded, it has failed completely.

Use a `Pipeline`, never a bare `StandardScaler` call. Fitting the scaler on the full dataset before splitting leaks test-set statistics into training; a pipeline refits the scaler inside each cross-validation fold automatically.

11.3. What C and γ Actually Do

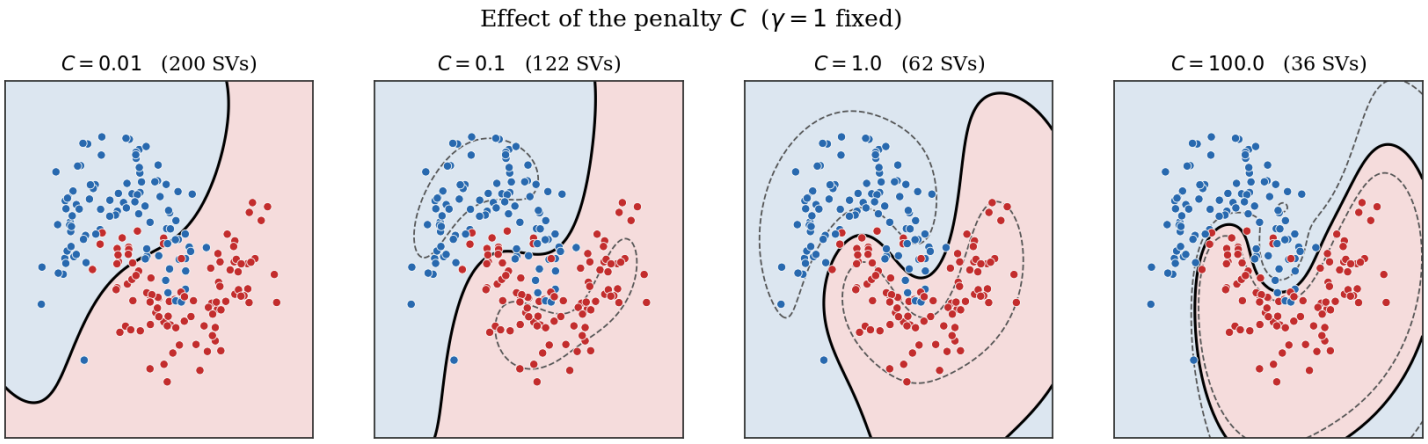
These are the only two hyperparameters that matter for an RBF SVM, and they have clean geometric meanings.

γ controls the reach of each support vector. Recall $\gamma = \frac{1}{2\sigma^2}$, so large γ means small σ — a narrow Gaussian, and thus a support vector that influences only its immediate neighbourhood.



At $\gamma = 0.1$ every point influences the whole space and the boundary is nearly linear — underfitting. At $\gamma = 100$ each point's influence is so local that the model grows an island around individual training points — textbook overfitting, and note the support vector count climbing from 62 to 182 as the model degenerates into memorization.

C controls how much margin violation is tolerated, exactly as in Section 5.2.



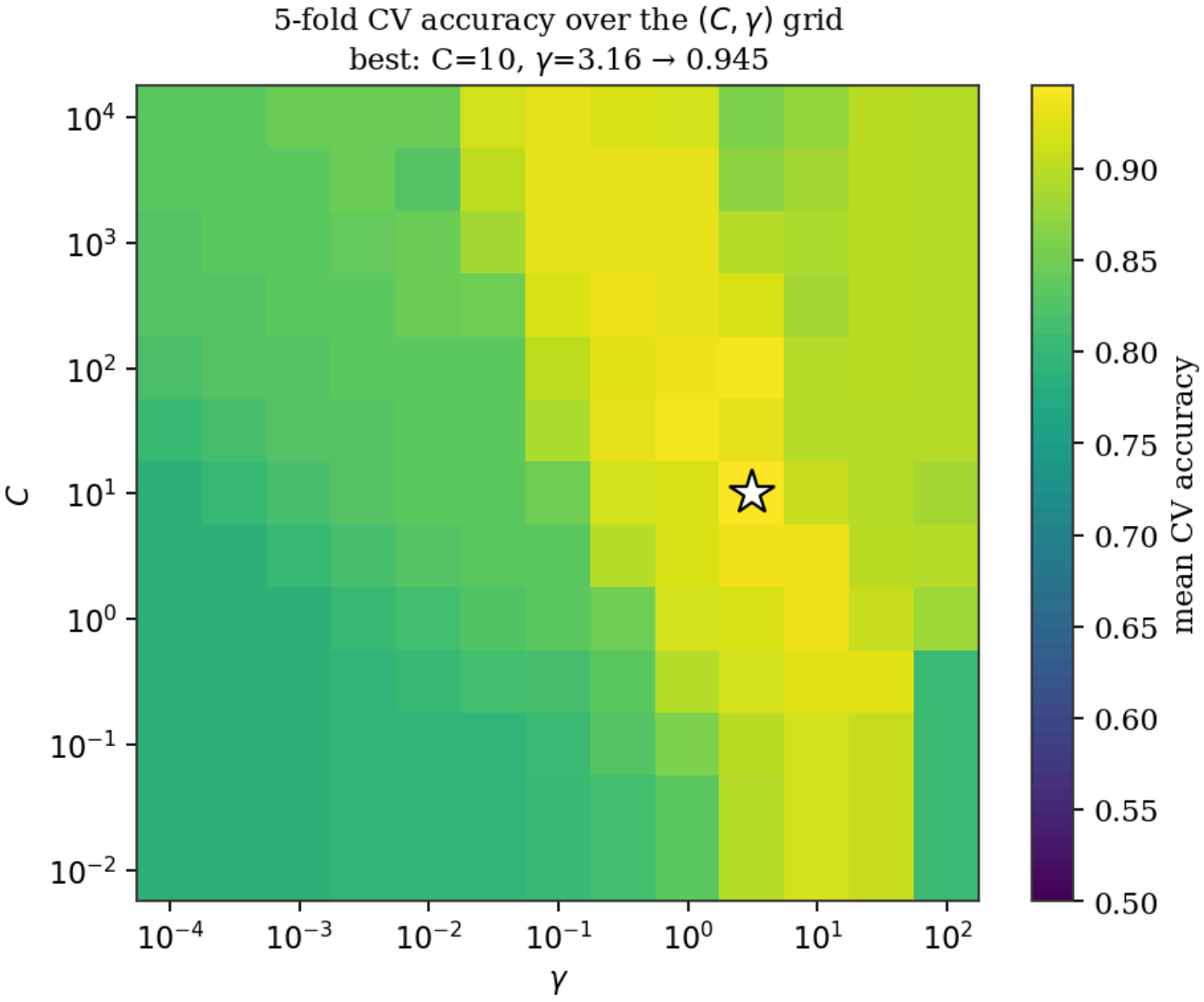
Small C buys a wide, smooth margin at the cost of accepting misclassifications (high bias); large C forces the boundary to contort itself to classify every training point (high variance).

The two interact strongly, which is why they must be tuned **jointly** — a good γ at one C may be terrible at another.

11.4. Choosing Them: Grid Search on a Log Scale

Because both parameters act multiplicatively, they must be searched on a logarithmic grid. Searching $C \in \{1, 2, 3, \ldots, 10\}$ is a common and wasted effort; the interesting range spans orders of magnitude.

```python
from sklearn.model_selection import GridSearchCV
import numpy as np

param_grid = {"svc__C": np.logspace(-2, 4, 13), "svc__gamma": np.logspace(-4, 2, 13)}

grid = GridSearchCV(
    make_pipeline(StandardScaler(), SVC(kernel="rbf")), param_grid, cv=5, n_jobs=-1
)
grid.fit(X, y)
print(grid.best_params_, round(grid.best_score_, 4))
```



The heatmap shows the structure you should expect: a **diagonal ridge** of good performance rather than an isolated peak. This is the C–γ interaction made visible — a larger γ can be compensated by a smaller C and vice versa. Practical consequences:

- there is rarely one uniquely correct setting, so do not over-tune to the third decimal place;
- if your best value sits on the **edge** of the grid, your grid is too small — extend it and re-run;
- start coarse (powers of 10), then refine near the ridge.

For larger search spaces `RandomizedSearchCV` typically finds the ridge with far fewer fits than an exhaustive grid.

11.5. Probabilities: the Awkward Part

SVMs have no probabilistic interpretation. The decision function is a signed distance, not a likelihood — nothing in the derivation produced a probability, because the hinge loss is not a log-likelihood for any distribution.

`SVC(probability=True)` does not change that. It fits the SVM as usual, then fits a **sigmoid** to the decision values using internal 5-fold cross-validation — a post-hoc calibration known as **Platt scaling**:

$$P(y = 1 \mid x) \approx \frac{1}{1 + \exp\left(A f(x) + B\right)},$$

with A, B estimated from held-out folds. Three consequences people run into:

1. **It is slow** — an internal 5-fold CV means roughly 5× the training time.
2. **It can disagree with `predict`.** The sigmoid is fit on cross-validated decision values while `predict` uses the final model's sign, so the two are separate objects. On overlapping data I measured **15 disagreements out of 240 test points** (6.3%) between `predict` and `argmax predict_proba`. On clean, well-separated data the disagreement is usually zero or one — but "usually" is doing real work in that sentence.
3. **Set `random_state`** or the internal CV makes results non-reproducible.

If you genuinely need calibrated probabilities, use logistic regression, or wrap the SVM in `CalibratedClassifierCV` where the calibration is at least explicit. If you only need to rank or threshold, use `decision_function` and skip the whole problem.

11.6. Class Imbalance

The hinge loss weights every point equally, so a rare class contributes proportionally little to the objective and the optimizer is happy to sacrifice it. The fix is a per-class penalty — C_+ for the positive class and C_- for the negative — which `class_weight='balanced'` sets to be inversely proportional to class frequency.

The effect on a dataset with 4.4% positives:

```
SVC(kernel="rbf", class_weight="balanced")
```

```
class_weight=None       acc=0.962   balanced_acc=0.574   minority_recall=0.148
class_weight=balanced   acc=0.923   balanced_acc=0.872   minority_recall=0.815
```

Look carefully at what happened. The unweighted model has the **higher accuracy** — 96.2% — and is nearly useless: it finds 15% of the minority class. The weighted model is 4 points *worse* on accuracy and finds 82%.

This is the accuracy paradox in miniature. On imbalanced data, accuracy rewards ignoring the rare class, which is usually the class you care about. Score with balanced accuracy, F1, or AUC instead — and note that `GridSearchCV` optimizes accuracy by default, so you must pass `scoring='balanced_accuracy'` or your search will happily tune toward the useless model.

11.7. A Sensible Default Recipe

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
import numpy as np

pipe = make_pipeline(StandardScaler(), SVC(kernel="rbf", class_weight="balanced"))

grid = GridSearchCV(
    pipe,
    {"svc__C": np.logspace(-2, 4, 7), "svc__gamma": np.logspace(-4, 2, 7)},
    cv=5,
    scoring="balanced_accuracy",
    n_jobs=-1,
)
grid.fit(X_train, y_train)
```

Scale inside a pipeline, RBF kernel, log-spaced joint grid, a scoring metric appropriate to your class balance. If $n > 100{,}000$, swap `SVC` for `LinearSVC` or `SGDClassifier(loss='hinge')` and revisit Section 9.3.

Closing

I hope this article works for you as an assist to understanding the SVM algorithm under the hood. Thank youuu for reading ❤️ !!