SVM

kartik virmani

September 2025

1 objective function

What's going on.

- We want a line/plane (a *hyperplane*) that separates two classes and stays as far as possible from nearby points (big *margin*).
- **Hard margin:** assumes perfect separation. Constraint for every sample (x_i, y_i) (with $y_i \in \{-1, +1\}$):

$$y_i(\theta^\top x_i + \theta_0) \;\geq\; 1.$$

  Objective: make the margin wide (equivalently make $\|\theta\|$ small):

$$\min_{\theta} \; \frac{1}{2}\|\theta\|^2.$$

- **Soft margin:** real data is messy—some points lie inside the margin or on the wrong side. Introduce *slack* $\xi_i \geq 0$ to allow controlled violations:

$$y_i(\theta^\top x_i + \theta_0) \;\geq\; 1 - \xi_i.$$

  Penalize total violation while still keeping a wide margin:

$$\min_{\theta, \{\xi_i\}} \; \frac{1}{2}\|\theta\|^2 \;+\; C\sum_i \xi_i.$$

- C **(the knob):** Bigger $C \Rightarrow$ fewer violations (stricter fit, risk of overfitting). Smaller $C \Rightarrow$ wider margin with more tolerated violations (more robust).
- **Support vectors:** The training points that touch the margin (or violate it). They alone "support" the decision boundary.

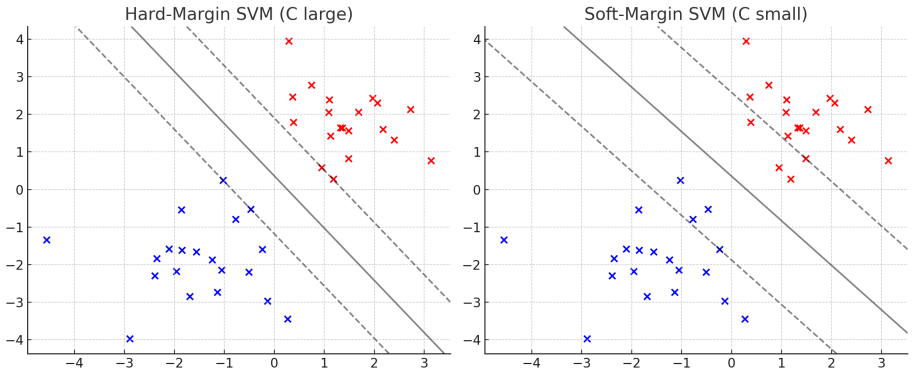

Figure 1: **Left: Hard-margin SVM.** Perfect separation, no violations allowed. **Right: Soft-margin SVM.** Allows some violations (slack) to handle noise/overlap.

2 support samples

- Support vectors are the **training points closest to the hyperplane**.
- In the hard-margin case, they lie exactly on the margin boundaries.
- In the soft-margin case, they may lie on the margin, inside it, or even be misclassified — but they still determine the boundary.
- Other points far away do not affect the hyperplane. Only support vectors "support" or fix its position.
- Intuition: they are like **guardrails** that hold the separating margin in place.

3 downsampling

Each MNIST image is originally $28 \times 28 = 784$ pixels. To reduce computation for SVM training, we downsample the images to $14 \times 14 = 196$ pixels.

One simple way to do this is to average over non-overlapping 2×2 blocks:

$$B_{i,j} \;=\; \frac{1}{4}\Big(A_{2i,\,2j} + A_{2i+1,\,2j} + A_{2i,\,2j+1} + A_{2i+1,\,2j+1}\Big),$$

where A is the original 28×28 image and B is the downsampled 14×14 image.

This reduces the dimensionality of each sample from 784 features to 196 features, while preserving the overall structure of the digit. We then use these downsampled vectors for SVM training.

4 svm classifier

We create the SVM classifier in scikit-learn using:

```
from sklearn import svm
classifier = svm.SVC(C=1.0, kernel='rbf', gamma='auto')
```

Meaning of parameters.

- C: regularization parameter controlling the trade-off between margin size and classification errors. Large $C \Rightarrow$ stricter fit (less slack, risk of overfitting). Small $C \Rightarrow$ wider margin, more tolerance for errors. **Default:** $C = 1.0$.
- γ: kernel width parameter for the RBF kernel $K(x_i, x_j) = \exp(-\gamma\|x_i - x_j\|^2)$. Small $\gamma \Rightarrow$ smoother boundary. Large $\gamma \Rightarrow$ more complex boundary. **Default:** "scale" (i.e. $1/(n_{features} \cdot \mathrm{Var}(X))$).

Experiments. We trained the SVM on the downsampled 14×14 MNIST data (1000 samples per class, 10k total). We then reported:

- Validation error = [*0.7868*],
- Ratio of support vectors to training samples = [*1.0*],
- Confusion matrix: plotted below. It shows that digits such as 4 and 9, or 3 and 5, are often confused due to their similar handwritten shapes.

Data preparation (as required).

1. Load MNIST (70,000 images, 28×28) and convert labels to integers.
2. Sample 1,000 images per digit (10,000 total), then split into 80% train (8,000) and 20% validation (2,000) with stratification.
3. Normalize pixel values to $[0, 1]$ and **downsample only the images** from 28×28 to 14×14 (196 features) via non-overlapping 2×2 block averaging.

Classifier. We use scikit-learn's RBF-kernel SVM:

$K(x_i, x_j) = \exp\big(-\gamma\,\|x_i - x_j\|^2\big)$, `SVC`($C = 1.0$, `kernel='rbf'`, γ = `'auto'`).

- C **(regularization):** trades off a wide margin vs. training errors (slack). Larger $C \Rightarrow$ fewer violations but higher overfitting risk; smaller $C \Rightarrow$ wider margin, more tolerance. *Default in sklearn:* $C = 1.0$.
- γ **(RBF width):** controls how local each support vector's influence is. Larger $\gamma \Rightarrow$ more wiggly boundary; smaller $\gamma \Rightarrow$ smoother boundary. *Default in sklearn:* `'scale'` ($1/(d\,\mathrm{Var}(X))$), but we use `'auto'` ($1/d$) per the prompt.

What we report.

- **Validation accuracy/error:**

$$val\_acc = \frac{\#correctonval}{\#val}, \qquad val\_err = 1 - val\_acc.$$

  Measured: [`0.3257`].

- **Support-vector ratio:**

$$\frac{\#supportvectors}{\#trainingsamples} = \frac{|\mathcal{S}|}{N_{train}}.$$

  Measured: [`1.0`].

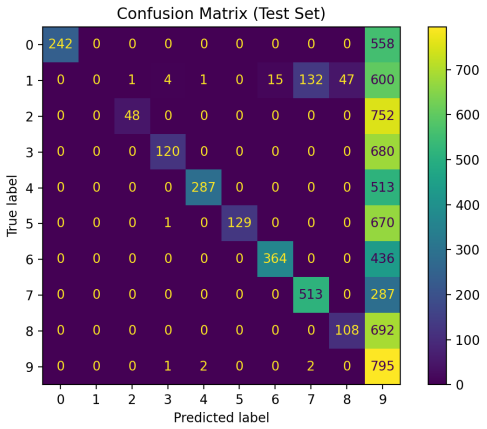

Figure 2: Confusion matrix on the held-out test set for SVC(RBF, C=1.0, γ=`auto`) using 14×14 inputs.

Notes (brief). Typical confusions occur between visually similar digits (e.g., 4 vs. 9, or 3 vs. 5). A larger C or γ can reduce some training errors but may overfit; grid search in 1(g) tunes C systematically.

Support-Vector Ratio: What it is and why it can be 1

Definition. The support-vector (SV) ratio measures how many training points become support vectors:

$$SVratio = \frac{|\mathcal{S}|}{N_{train}}.$$

A value of 1.0 means *every* training point is a support vector.

Why it can be ≈ 1 in our setup.

- **Multi-class one-vs-one (10 classes $\Rightarrow$ 45 binary SVMs).** A sample only needs to be an SV in *any* subproblem to count; the union often covers most points.
- **Flexible RBF kernel.** With C=1.0 and γ=`auto`, many points lie near/inside margins, so many become SVs.
- **Downsampling $28\times28 \rightarrow 14\times14$ increases overlap/noise,** pushing more samples close to the decision boundary.

Implications. Large model size and slower prediction (cost scales with $|\mathcal{S}|$); may indicate a complex boundary and potential overfitting.

How to reduce the ratio.

- Lower C (e.g., 0.3 or 0.1): wider margin, fewer SVs.
- Use smaller γ (e.g., `scale` or a small numeric value) for smoother boundaries.
- Apply PCA (fit on train; apply to val/test), e.g., 50–100 components, to denoise and reduce redundancy.
- Try a linear SVM as a baseline.

5 reading manual

Options you may not have seen in class (brief meanings and defaults).

- `kernel` $\in \{linear, poly, rbf, sigmoid\}$ (**default: rbf**): chooses feature map.
- `gamma` (**default: "scale"** $= 1/(d\cdot\mathrm{Var}(X))$): kernel width for RBF/poly/sigmoid; smaller $\Rightarrow$ smoother boundary.
- `degree` (**default: 3**): only for `poly`.
- `coef0` (**default: 0.0**): only for `poly`/`sigmoid`, controls offset of kernel.
- `C` (**default: 1.0**): soft–margin trade-off (penalizes slack).
- `class_weight` (**default: None**; can use "balanced" or a dict): handles class imbalance.
- `probability` (**default: False**): enables calibrated class probabilities via Platt scaling (extra CV pass).
- `decision_function_shape` $\in \{ovr, ovo\}$ (**default: ovr**): shape of the output scores for multi-class; *training* still uses one-vs-one internally.

- `break_ties` (**default: False**): when `decision_function_shape=ovr`, break near ties using distances.
- `cache_size` (**default: 200 MB**): memory for kernel cache (speed).
- `tol` (**default:** 10^{-3}): stopping tolerance for the optimizer.
- `max_iter` (**default:** -1 = no limit): cap on training iterations.
- `shrinking` (**default: True**): enable the shrinking heuristic (explained below).

What does `shrinking` do? It enables LIBSVM's *shrinking heuristic*: during training, variables (Lagrange multipliers) that appear to satisfy KKT conditions are temporarily removed from the working set. This reduces the active problem size, often speeding convergence *without changing the final solution.* Setting `shrinking=False` disables this heuristic (usually slower).

What optimization algorithm does SVC use? `SVC` in scikit-learn wraps **LIBSVM**. Training uses a decomposition/SMO-style algorithm that repeatedly solves very small QP subproblems (pairs of variables) with a kernel cache and the shrinking heuristic. For multi-class, LIBSVM trains *one-vs-one* binary SVMs and combines their outputs.

6 Mathematical formulation

What `SVC` does (scikit-learn / LIBSVM). For K classes, `SVC` trains **one-vs-one** (OvO) binary SVMs for each pair of classes: there are $K(K-1)/2$ classifiers. At prediction time, each pairwise classifier votes for one class; the final label is the class with the most votes (ties may be broken using decision values).

Alternative multi-class strategies using binary SVMs.

- **One-vs-rest (OvR):** Train K binary SVMs, each class vs. the rest. Predict the class whose classifier has the largest (signed) decision value.
- **Error-Correcting Output Codes (ECOC):** Design a code matrix (rows=classes, columns=binary tasks). Train one binary SVM per column; predict the class whose codeword is closest to the vector of binary outputs.
- **Hierarchical / DAG-SVM:** Organize a tree of binary decisions; traverse from root to a leaf (a class).

Direct (non-reduction) multi-class SVMs (for context). There are also *native* multi-class SVM formulations (e.g., Crammer–Singer, Weston–Watkins) that optimize a single objective with K weight vectors simultaneously, but `SVC` does not use these by default.

Gabor Filters for MNIST Classification - LaTeX Section

"'latex

7 Gabor Filter-Based Feature Extraction for MNIST Classification

7.1 Introduction to Gabor Filters

Gabor filters are mathematical functions that act like specialized detectors for edges and textures in images. Named after Hungarian physicist Dennis Gabor (who invented holography), these filters are inspired by how cells in the mammalian visual cortex process visual information.

To understand Gabor filters intuitively, imagine you're looking at a digital image through a series of "smart magnifying glasses." Each magnifying glass is designed to highlight specific patterns:

- Some highlight horizontal lines
- Others detect vertical edges
- Some focus on diagonal patterns at 45°
- Different glasses are sensitive to fine details vs. broad patterns

The mathematical representation of a Gabor filter is:

$$g(x, y; \theta, F, \sigma_x, \sigma_y) = \exp(i2\pi Fp) \exp\left(-\pi\left(\frac{p^2}{\sigma_x^2} + \frac{q^2}{\sigma_y^2}\right)\right) \tag{1}$$

where $p = x\cos\theta + y\sin\theta$ and $q = -x\sin\theta + y\cos\theta$.

The key parameters are:

- F: **Frequency** - how fine or coarse the patterns are (like zoom level)
- θ: **Orientation** - which direction the filter looks for edges (0°, 45°, 90°, etc.)
- σ_x, σ_y: **Standard deviations** - the size and shape of the detection area
- **Bandwidth**: Controls how selective the filter is (inversely related to standard deviation)

7.2 Building a Gabor Filter Bank

Instead of using just one filter, we create a "filter bank" - a collection of many different Gabor filters that together can detect various patterns in our images. Think of this like having a toolkit with different specialized tools, each designed for a specific job.

For our MNIST digit classification task, we systematically create filters with:

- **Orientations**: $\theta = [0, \frac{\pi}{4}, \frac{\pi}{2}, \frac{3\pi}{4}]$ (0°, 45°, 90°, 135°)
- **Frequencies**: $F = [0.05, 0.2, 0.35]$ (coarse, medium, fine patterns)
- **Bandwidths**: $B = [0.3, 0.6, 0.9]$ (different selectivity levels)

This gives us $4 \times 3 \times 3 = 36$ different filters in our bank. Each filter is like a different "question" we ask about each image: "Does this image contain vertical edges?" "Are there diagonal patterns?" "What about fine horizontal textures?"

7.3 Feature Extraction Process

When we apply our 36 Gabor filters to a 14×14 MNIST image, we perform the following steps:

1. **Convolution**: Each filter is mathematically "convolved" with the image, producing a response map showing where the filter's pattern is detected
2. **Feature Extraction**: From each response map, we can extract features in two ways:
    - **Statistical Method**: Calculate summary statistics (mean, standard deviation, minimum, maximum) from each response map, giving us $36 \times 4 = 144$ features per image
    - **All-Pixel Method**: Use every pixel value from each response map, giving us $36 \times 196 = 7,056$ features per image (since each downsampled image is 14×14 = 196 pixels)

7.4 The Curse of Dimensionality and PCA Solution

The all-pixel method creates a significant challenge: we transform each 196-pixel image into a 7,056-dimensional feature vector. This is called the "curse of dimensionality" - having too many features relative to our number of training examples can cause machine learning algorithms to perform poorly.

With only 1,000 training examples (100 per digit class) but 7,056 features, our data becomes very sparse in this high-dimensional space. It's like trying to find patterns in a room with 7,056 dimensions when you only have 1,000 data points to learn from!

To solve this problem, we use **Principal Component Analysis (PCA)**:

- PCA identifies the most important directions (principal components) in our high-dimensional feature space
- It reduces dimensionality while preserving the most significant information
- We typically retain components that capture 95% of the total variance in the data
- This might reduce our 7,056 features to perhaps 200-500 more meaningful features

7.5 Experimental Setup

Our experimental methodology compares different approaches:

1. **Baseline**: Standard SVM using raw pixel intensities (196 features)
2. **Small Filter Bank**: 12 Gabor filters with basic parameter variations
3. **Standard Filter Bank**: 36 Gabor filters as specified in the problem
4. **Large Filter Bank**: 80 Gabor filters with finer parameter sampling

For each filter bank configuration, we test both statistical features and all-pixel features (with PCA when necessary).

7.6 Why Gabor Filters Work Well for Digit Recognition

Gabor filters are particularly effective for MNIST digit classification because:

- **Edge Detection**: Digits are primarily composed of edges and curves, which Gabor filters excel at detecting
- **Orientation Sensitivity**: Different digits have characteristic orientations - '1' has vertical edges, '7' has diagonal lines, 'O' has curved boundaries
- **Scale Invariance**: Multiple frequency parameters help detect both thick and thin strokes
- **Biological Inspiration**: The human visual system uses similar mechanisms, suggesting this approach aligns with effective visual processing

7.7 Expected Results and Analysis

We expect the following performance hierarchy:

1. Raw pixels (baseline) < Statistical Gabor features < All-pixel Gabor features (with PCA)

2. Larger filter banks should generally perform better, up to a point where overfitting begins

3. The 36-filter bank should provide a good balance between feature richness and computational efficiency

The improvement comes from the fact that Gabor features capture more meaningful visual patterns than raw pixel intensities. While a raw pixel approach might struggle to distinguish between digits that are similar overall but differ in local edge patterns, Gabor filters can identify these subtle but important differences.

7.8 Computational Considerations

The computational complexity scales as:

- **Feature extraction time**: $O(N \times F \times P)$ where N is number of images, F is number of filters, and P is pixels per image
- **Memory usage**: $O(N \times F \times P)$ for all-pixel method, $O(N \times F \times S)$ for statistical method where S is number of statistics
- **SVM training time**: Depends on final feature dimensionality after PCA

This explains why PCA becomes essential for the high-dimensional case - it makes the problem computationally tractable while preserving the most important information for classification. “‘