

Kernels: Similarity, the Kernel Trick, and Kernelized Algorithms

Lecture Notes Consolidated with Claude AI — CS6140 Machine Learning

Part A (§1–§6) introduces kernels as *data similarity*: what they are, which functions qualify, and a gallery of kernels with the data each one suits. **Part B** (§7–§12) uses them to make algorithms we already know nonlinear: ridge regression (**HW3 Problem 2**), the perceptron, k -NN, and PCA (**Kernel PCA**, **HW3 Problem 4**). The SVM note then applies the same idea to support vector machines.

Notation (same as previous notes): a datapoint $\mathbf{x}$ is a $1 \times D$ row vector, X is the $N \times D$ data matrix, a weight vector $\mathbf{w}$ is $D \times 1$, so a linear score is $\mathbf{xw}$ and the dot product of two datapoints is $\mathbf{xz}^T$.

Part A — Kernels as Data Similarity

1 The Fundamental Idea: Similarity

Almost every method in this course relies, explicitly or not, on one question: *how similar are two datapoints?* k -NN (HW1) predicts from the most similar training points; a decision-tree leaf groups points that passed the same tests; clustering groups similar points together. Getting the similarity right is most of the work:

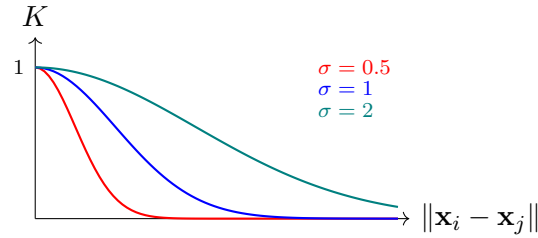
Key point: A good similarity function for the domain is (nearly) a good classifier: if same-class points are similar and different-class points are not, almost any method built on top of it will do well.

Distance measures dissimilarity; similarity is its opposite (large when the distance is small). Common choices for $\mathbf{x}_i, \mathbf{x}_j \in \mathbb{R}^D$:

- **Euclidean** $\|\mathbf{x}_i - \mathbf{x}_j\| = \sqrt{\sum_d (x_i^d - x_j^d)^2}$; **Manhattan** $\sum_d |x_i^d - x_j^d|$; more generally **p -norm** $(\sum_d |x_i^d - x_j^d|^p)^{1/p}$.
- **Mahalanobis** $\sqrt{(\mathbf{x}_i - \mathbf{x}_j)\Sigma^{-1}(\mathbf{x}_i - \mathbf{x}_j)^T}$: Euclidean after rescaling by the covariance Σ (with $\Sigma = I$ it is Euclidean; with a diagonal Σ each feature is divided by its own variance).
- **Hamming** (binary or categorical vectors): number of positions where they differ.
- **Dot product** $\mathbf{x}_i \mathbf{x}_j^T = \sum_d x_i^d x_j^d$: already a similarity (large when the vectors point the same way and are long). **Cosine** $\frac{\mathbf{x}_i \mathbf{x}_j^T}{\|\mathbf{x}_i\| \|\mathbf{x}_j\|}$ removes the length, keeping only the angle (common for text).
- **Gaussian (RBF)**, a similarity built from Euclidean distance:

$$K(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{\sigma^2}\right) \in (0, 1]. \quad (1)$$

It equals 1 when $\mathbf{x}_i = \mathbf{x}_j$ and decays to 0 as they move apart; the bandwidth σ sets *how far* similarity reaches.



Gaussian similarity as a function of distance: a small σ makes a point similar only to its immediate neighbors; a large σ makes everything look alike.

Remark. Conventions vary: some texts write $2\sigma^2$ in the denominator, and `sklearn` writes $\exp(-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|^2)$, so $\gamma = 1/\sigma^2$ in our notation.

Similarity is domain knowledge. Which points “should” be similar depends on the task:

- *Patients*, task diabetes: similar if age, weight, glucose, blood pressure are close (Euclidean on standardized features).
- *Emails*, task spam: similar if they share many (rare) words (cosine on word counts).
- *Images*, task object recognition: raw pixels are a poor representation (a one-pixel shift changes every pixel); better similarities compare shapes or local patterns.
- *Movies*, task recommendation: similar if the same users rated them highly — a similarity defined without any “features” at all.

2 Dot Products, Feature Maps, and the Kernel Matrix

2.1 The Linear Kernel

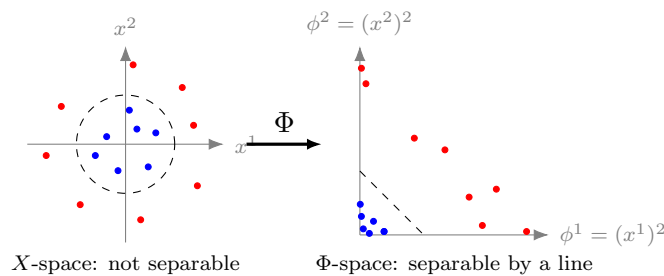
Collect all pairwise dot products of the data in an $N \times N$ matrix:

$$K = XX^T, \quad K_{ij} = \mathbf{x}_i \mathbf{x}_j^T = \langle \mathbf{x}_i, \mathbf{x}_j \rangle. \quad (2)$$

This is the **linear kernel** (also the **Gram matrix** of the data). It is symmetric, and positive semidefinite: $\mathbf{v}^T K \mathbf{v} = \mathbf{v}^T X X^T \mathbf{v} = \|X^T \mathbf{v}\|^2 \geq 0$.

2.2 Mapping to a New Space

Many datasets are not linearly separable in the original features but become separable after a nonlinear transformation Φ that maps each datapoint (row) to a new, usually higher-dimensional row vector $\Phi(\mathbf{x})$. Classic example: one class inside a circle, the other outside. With $\Phi(x^1, x^2) = ((x^1)^2, (x^2)^2, \sqrt{2}x^1x^2)$ the circle $(x^1)^2 + (x^2)^2 = r^2$ becomes the plane $\phi^1 + \phi^2 = r^2$ — a linear boundary in the new space:



(The third coordinate $\sqrt{2}x^1x^2$ is not drawn; the first two already separate the classes.)

The dot product in the new space is a similarity between the original points:

$$K(\mathbf{x}_i, \mathbf{x}_j) = \Phi(\mathbf{x}_i)\Phi(\mathbf{x}_j)^T, \quad K = \Phi(X)\Phi(X)^T \quad (N \times N), \quad (3)$$

where $\Phi(X)$ stacks the mapped rows. A function K of this form is called a **kernel**: *a similarity that is a dot product in some feature space*.

3 Which Similarities Are Valid Kernels?

Before collecting examples, we need the rule that decides which similarity functions qualify. Not every reasonable-looking similarity is a dot product in some space. The test:

Key point: K is a valid kernel (there exists Φ with $K(\mathbf{x}_i, \mathbf{x}_j) = \Phi(\mathbf{x}_i)\Phi(\mathbf{x}_j)^T$) if and only if for every finite set of points the kernel matrix K is **symmetric and positive semidefinite** ($\mathbf{v}^T K \mathbf{v} \geq 0$ for all $\mathbf{v}$, equivalently all eigenvalues ≥ 0).

- *Only if:* $\mathbf{v}^T \Phi(X)\Phi(X)^T \mathbf{v} = \|\Phi(X)^T \mathbf{v}\|^2 \geq 0$, exactly as for the linear kernel.
- *If* (for a finite dataset): decompose $K = U\Lambda U^T$ with $\Lambda \geq 0$ (spectral decomposition, as for Σ in the PCA note) and let $\Phi(X) = U\Lambda^{1/2}$, i.e. row i of $U\Lambda^{1/2}$ is $\Phi(\mathbf{x}_i)$. Then $\Phi(X)\Phi(X)^T = U\Lambda U^T = K$. **Mercer's theorem** (optional section) extends this to functions on continuous domains.
- *A counterexample:* $K(\mathbf{x}, \mathbf{z}) = -\|\mathbf{x} - \mathbf{z}\|^2$ (“closer is more similar”). For $x = 0, z = 1$: $K = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$, eigenvalues ± 1 . Not PSD, so no feature space exists.
- *A practical one:* “similar if within a tolerance on each measurement” sounds harmless, but A can be within tolerance of B and B of C while A is far from C — no feature space can do that. HW3 optional Problem 10 (Dr. Lawler’s patients) shows what then breaks in Kernel PCA and in an SVM, and how to repair it.

Building new kernels from old ones. Checking positive semidefiniteness directly is often hard; instead, build kernels from known ones. If K_1, K_2 are kernels, so are: $K_1 + K_2$; aK_1 for $a > 0$; $K_1 K_2$; $f(\mathbf{x})f(\mathbf{z})$ for any real function f ; $K_1(\Phi(\mathbf{x}), \Phi(\mathbf{z}))$; a polynomial with nonnegative coefficients of K_1 ; $\exp(K_1)$. Each kernel in the next section is valid by one of these rules.

4 A Gallery of Kernels: What Each One Considers Similar

The key question for each kernel is: *which points does it call similar?* Figure 1 fixes one point $\mathbf{z}$ (the star) and colors every location $\mathbf{x}$ of the plane by $K(\mathbf{x}, \mathbf{z})$. Figure 2 shows what a kernel method (here kernel ridge regression on ± 1 labels, §8, thresholded at 0) can do with each kernel on three toy datasets.

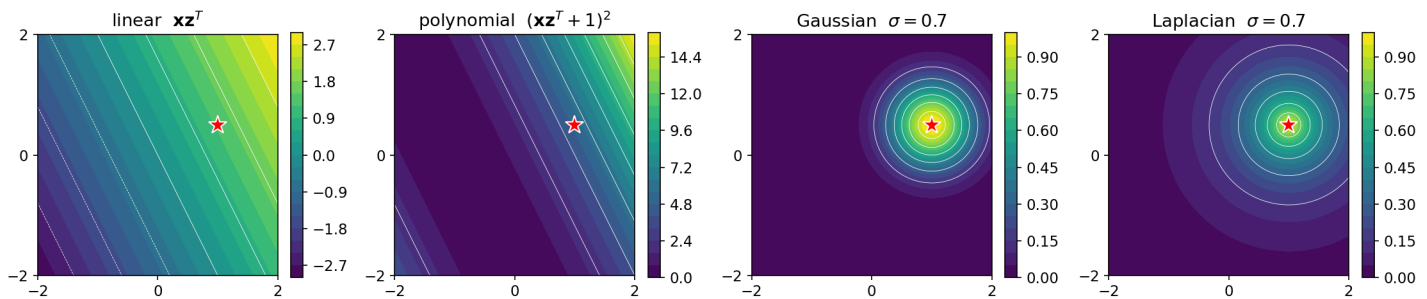


Figure 1: Similarity landscapes $K(\mathbf{x}, \mathbf{z})$ for a fixed $\mathbf{z} = (1, 0.5)$ (star). Linear and polynomial similarity are *global* (they keep growing in the direction of $\mathbf{z}$, even far past it); Gaussian and Laplacian similarity are *local* (a bump centered at $\mathbf{z}$, close to 0 elsewhere).

4.1 Linear Kernel $K(\mathbf{x}, \mathbf{z}) = \mathbf{x}\mathbf{z}^T$

- **Similarity:** large when $\mathbf{x}$ points in the same direction as $\mathbf{z}$ and is long. Level sets are straight lines perpendicular to $\mathbf{z}$ (Figure 1): not local at all.
- **Feature space:** $\Phi(\mathbf{x}) = \mathbf{x}$, i.e. no mapping; kernel methods reduce to their ordinary linear versions (top-left of Figure 2; it fails on circles and XOR).
- **Works well when** the data already has many informative features, especially high-dimensional and sparse data where classes are close to linearly separable. *Example:* text classification with bag-of-words features (Joachims 1998, Reuters-21578); for text, linear kernels are typically about as accurate as nonlinear ones at a fraction of the cost, which is why large-scale linear solvers such as LIBLINEAR (Fan et al. 2008) exist. In this course: Spambase, 20 Newsgroups.
- **Cosine** similarity is the linear kernel on length-normalized vectors $\mathbf{x}/\|\mathbf{x}\|$ (valid by the $f(\mathbf{x})f(\mathbf{z})K_1$ rules); the standard choice for TF-IDF document vectors.

4.2 Polynomial Kernel $K(\mathbf{x}, \mathbf{z}) = (\mathbf{x}\mathbf{z}^T + c)^p$

- **Similarity:** still global (Figure 1), but now a curved function of the dot product.
- **Feature space:** all *products of up to p features*. For $D = 2, p = 2$: expanding $(1 + x^1 z^1 + x^2 z^2)^2$,

$$\Phi(\mathbf{x}) = (1, \sqrt{2}x^1, \sqrt{2}x^2, (x^1)^2, (x^2)^2, \sqrt{2}x^1x^2) \quad (6\text{-D}). \quad (4)$$

Without the $+c$, $(\mathbf{x}\mathbf{z}^T)^2$ keeps only the degree-2 terms: $\Phi(\mathbf{x}) = ((x^1)^2, (x^2)^2, \sqrt{2}x^1x^2)$. That loses the linear coordinates, and they matter: the parabola $x^2 = (x^1)^2$ is the line $\phi^3 = \sqrt{2}\phi^4$ in the 6-D space (it uses $\phi^3 = \sqrt{2}x^2$), but with degree-2 terms only, $\Phi(\mathbf{x}) = \Phi(-\mathbf{x})$, so $(0, 1)$ and $(0, -1)$ would always get the same class — although they lie on opposite sides of the parabola:

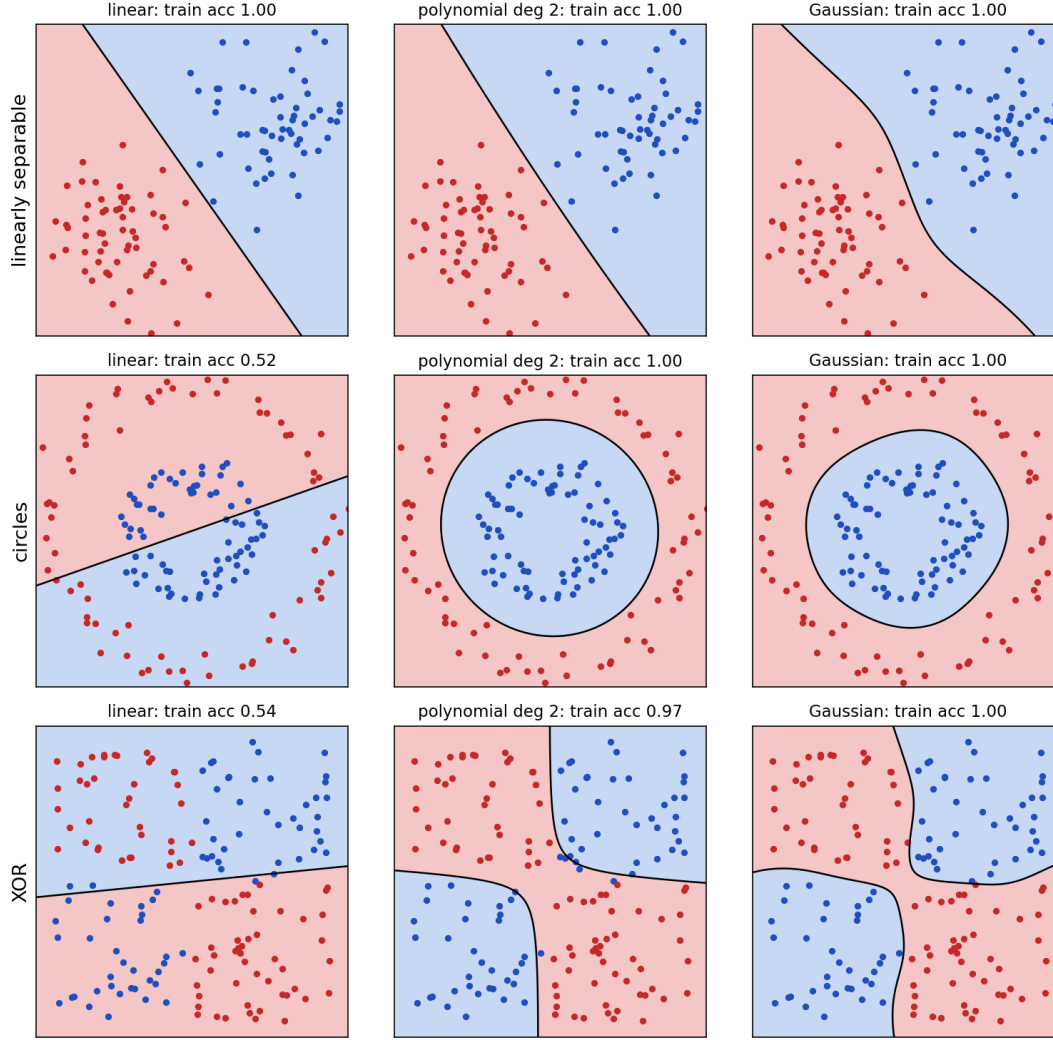
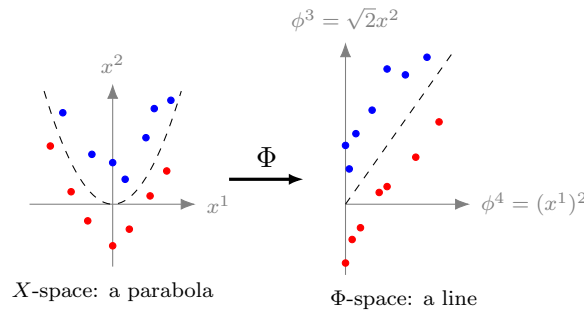


Figure 2: Decision boundaries of kernel ridge regression (labels ± 1 , predict the sign) with three kernels, on three datasets. Rows: linearly separable blobs; concentric circles; XOR. Columns: linear, polynomial of degree 2 ($\mathbf{xz}^T + 1$)², Gaussian.



With $c > 0$ the polynomial kernel has every degree up to p , so it can use both. The feature map is *not unique*: $\Phi'(\mathbf{x}) = ((x^1)^2, x^1x^2, x^2x^1, (x^2)^2)$ gives the same dot product — the kernel is the well-defined object. In general there are $\binom{D+p}{p}$ features (6 for $D = p = 2$; about 177,000 for $D = 100, p = 3$), yet K costs one D -dimensional dot product.

- **Works well when** what matters are *interactions* between features: “feature a AND feature b ” (a product $x^a x^b$). A degree-2 boundary is a conic section: circles, ellipses, hyperbolas — exactly what

circles and XOR need (Figure 2, middle column). *Examples*: handwritten digits (Schölkopf, Burges & Vapnik 1995, USPS digits; a degree-4 polynomial kernel on MNIST pixels among the methods compared in LeCun et al. 1998), where pixel-pair interactions describe strokes; natural language processing with word-conjunction features (e.g. dependency parsing, Yamada & Matsumoto 2003).

- **Caution**: high degrees blow up numerically for $|\mathbf{xz}^T| > 1$ and overfit; standardize features and keep p small (2–4).

4.3 Gaussian (RBF) Kernel $K(\mathbf{x}, \mathbf{z}) = \exp(-\|\mathbf{x} - \mathbf{z}\|^2/\sigma^2)$

- **Similarity**: *local*. A round bump centered at $\mathbf{z}$ with width σ (Figure 1): points within about σ are similar, points farther away are essentially unrelated, regardless of direction.
- **Feature space**: infinite-dimensional. In one dimension,

$$e^{-(x-z)^2/\sigma^2} = e^{-x^2/\sigma^2} e^{-z^2/\sigma^2} e^{2xz/\sigma^2} = \sum_{n=0}^{\infty} \underbrace{\left(e^{-x^2/\sigma^2} \sqrt{\frac{2^n}{\sigma^{2n} n!}} x^n \right)}_{\phi_n(x)} \underbrace{\left(e^{-z^2/\sigma^2} \sqrt{\frac{2^n}{\sigma^{2n} n!}} z^n \right)}_{\phi_n(z)} \quad (5)$$

using $e^u = \sum_n u^n/n!$ with $u = 2xz/\sigma^2$: one feature for every power x^n (a polynomial kernel of *every* degree at once). It is valid by the rules: $f(\mathbf{x})f(\mathbf{z}) \cdot \exp(2\mathbf{xz}^T/\sigma^2)$.

- **Works well when** the data is low-to-moderate dimensional and dense, and the class regions have arbitrary smooth shapes; prediction becomes a smooth, similarity-weighted vote of nearby training points (much like k -NN). It handles all three toy datasets (Figure 2, right column). It is the usual default nonlinear kernel: the LIBSVM practical guide (Hsu, Chang & Lin) recommends trying it first. In this course: TwoSpirals, ThreeCircles, WAVE, CRESCENT (HW3).
- **Geometry of Φ** : $K(\mathbf{x}, \mathbf{x}) = 1$, so $\|\Phi(\mathbf{x})\| = 1$: every point is mapped onto a unit sphere. And $K > 0$, so any two mapped points are less than 90° apart.
- **It can fit anything**: for distinct points the Gaussian kernel matrix has full rank (in exact arithmetic; for large σ it is numerically close to singular). So $K\alpha = \mathbf{y}$ has a solution for *any* labels or targets $\mathbf{y}$: kernel ridge regression with $\lambda \rightarrow 0$ fits the training targets exactly, and an SVM with a large C classifies every training point correctly. A perfect training fit therefore says nothing; this is why σ (and λ or C) must be chosen on validation data (HW3 THEORY Problem 5).
- **Caution**: σ is a crucial hyperparameter (too small: every point is similar only to itself and the model memorizes; too large: everything looks alike); choose it on a validation set. Standardize features first, since distances mix units.

4.4 Laplacian Kernel $K(\mathbf{x}, \mathbf{z}) = \exp(-\|\mathbf{x} - \mathbf{z}\|/\sigma)$

- **Similarity**: local like the Gaussian, but with a sharp peak at $\mathbf{z}$ and heavier tails (Figure 1): very close points are much more similar than merely close ones, while far points keep a little similarity. (A variant uses the Manhattan distance $\sum_d |x^d - z^d|$.)
- **Works well when** the target function is less smooth (has kinks or abrupt changes) than the Gaussian assumes, or the data has outliers.

4.5 Histogram Kernels (Images, Bag-of-Features)

When each datapoint is a histogram (counts of visual words, color bins, word counts), similarity should compare bins:

- **Histogram intersection** $K(\mathbf{x}, \mathbf{z}) = \sum_d \min(x^d, z^d)$: the overlap of the two histograms (Swain & Ballard 1991, color indexing; Maji, Berg & Malik 2008, pedestrian detection).
- χ^2 **kernel** $K(\mathbf{x}, \mathbf{z}) = \exp\left(-\gamma \sum_d \frac{(x^d - z^d)^2}{x^d + z^d}\right)$: a Gaussian-like kernel built from a distance designed for histograms. Standard for object recognition with bag-of-visual-words features before deep learning (Zhang, Marszałek, Lazebnik & Schmid 2007; Caltech-101, PASCAL VOC).

4.6 A Kernel That Is Not Valid: Sigmoid

$K(\mathbf{x}, \mathbf{z}) = \tanh(a \mathbf{x} \mathbf{z}^T + c)$ was popular because a model built on it resembles a one-hidden-layer neural network. It is *not* positive semidefinite for many choices of a, c , so it is not a true kernel in general; methods may still run, but without the guarantees. A reminder to check validity (§3) rather than assume it.

5 Kernels for Non-Vector Data

A kernel only needs a similarity between two *objects*, so kernel methods work on data that was never a feature vector. Each of these is a valid kernel (a dot product of an implicit, often huge, count vector), computed efficiently without building that vector:

- **String kernels** (sequences: text, DNA, proteins). Similarity = number of shared substrings. The *spectrum kernel* counts shared substrings of length k : “cat” $\{ca, at\}$ and “cart” $\{ca, ar, rt\}$ share one bigram, so $K = 1$. Used for protein classification (Leslie, Eskin & Noble 2002, SCOP protein families) and text (Lodhi et al. 2002, string subsequence kernel, Reuters-21578), where gaps and mismatches are also allowed.
- **Tree kernels** (parse trees, XML, program syntax). Similarity = number of shared subtrees. Collins & Duffy 2002 used them to rerank parse trees on the Penn Treebank.
- **Graph kernels** (whole graphs, e.g. molecules). Similarity = number of shared walks, paths, or neighborhood patterns. Random-walk kernels (Gärtner et al. 2003; Vishwanathan et al. 2010) and the Weisfeiler–Lehman kernel (Shervashidze et al. 2011) are standard on chemical-compound datasets such as MUTAG and NCI1 (predict mutagenicity / anticancer activity of molecules).
- **Diffusion kernels** (nodes *within* one graph: social networks, web pages, gene networks). Similarity = how much “heat” spreads from one node to another along the edges (Kondor & Lafferty 2002).
- **Fisher kernels**. Fit a probabilistic model (e.g. a hidden Markov model) to the data; two objects are similar if the model’s parameters would be adjusted in similar ways to fit them (Jaakkola & Haussler 1999, protein homology detection).

6 The Kernel Trick

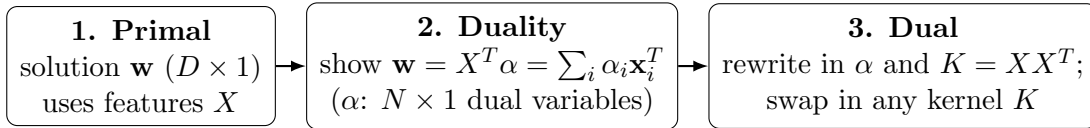
Key point: Kernel trick. If an algorithm uses the data *only* through dot products $\mathbf{x}_i \mathbf{x}_j^T$ (equivalently, only through $K = XX^T$), replace them with a kernel $K(\mathbf{x}_i, \mathbf{x}_j)$. The algorithm then runs, unchanged, in the feature space of Φ — without ever computing Φ . A linear method (a hyperplane, a linear fit, a linear projection) becomes nonlinear in the original features.

Most algorithms are written with $\mathbf{w}$ and the features X (the **primal** form), not with dot products. Part B shows how to rewrite them in the **dual** form, where only K appears.

Part B — Kernelizing Algorithms We Already Know

7 The Pattern

Every algorithm below follows the same three steps:



Once $\mathbf{w} = \sum_i \alpha_i \mathbf{x}_i^T$, a prediction on a new point $\mathbf{z}$ needs only similarities to the training points:

$$\mathbf{z} \mathbf{w} = \sum_{i=1}^N \alpha_i \mathbf{z} \mathbf{x}_i^T = \sum_{i=1}^N \alpha_i K(\mathbf{x}_i, \mathbf{z}). \quad (6)$$

The model is now “a weighted vote of similarities to the training data,” with weights α_i — one per datapoint instead of one per feature (Figure 3). It looks like a network with one hidden layer whose hidden units are the training points; the Neural Networks weeks will learn such units instead of storing them.

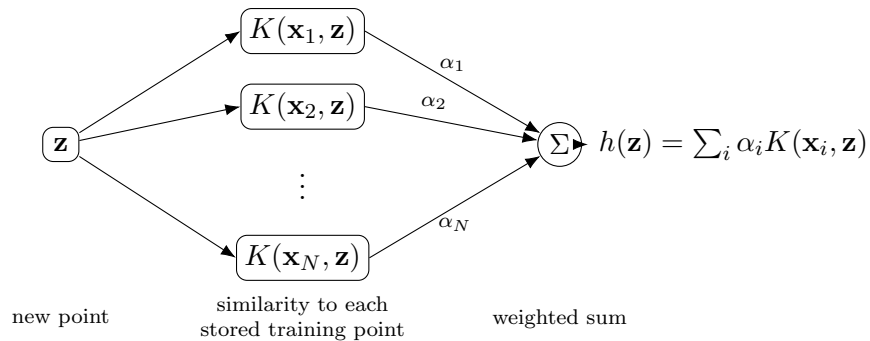


Figure 3: What a trained kernel model computes (Eq. 6). Training only chooses the weights α_i ; the “features” of $\mathbf{z}$ are its similarities to the training points.

8 Kernel Ridge Regression

Primal (Regression note): minimize $\|X\mathbf{w} - \mathbf{y}\|^2 + \lambda\|\mathbf{w}\|^2$, giving the normal equations

$$(X^T X + \lambda I) \mathbf{w} = X^T \mathbf{y} \quad (D \times D \text{ system}). \quad (7)$$

Duality. Rearrange: $\lambda \mathbf{w} = X^T \mathbf{y} - X^T X \mathbf{w} = X^T (\mathbf{y} - X \mathbf{w})$, so

$$\mathbf{w} = X^T \alpha, \quad \alpha = \frac{1}{\lambda} (\mathbf{y} - X \mathbf{w}). \quad (8)$$

The weight vector is a combination of the datapoints, with α_i proportional to the residual on datapoint i .

Dual. Substitute $\mathbf{w} = X^T \alpha$ into the definition of α : $\lambda \alpha = \mathbf{y} - X X^T \alpha = \mathbf{y} - K \alpha$, so

$$\boxed{\alpha = (K + \lambda I)^{-1} \mathbf{y}} \quad (N \times N \text{ system}), \quad \boxed{h(\mathbf{z}) = \sum_{i=1}^N \alpha_i K(\mathbf{x}_i, \mathbf{z})} \quad (9)$$

Training needs only the $N \times N$ kernel matrix of the training data; prediction needs the kernel between the test point and every training point. With the linear kernel this is exactly ridge regression (same predictions); with an RBF or polynomial kernel it fits a nonlinear curve.

Worked example (1-D, no intercept, linear kernel, $\lambda = 1$): $X = (1, 2, 3)^T$, $\mathbf{y} = (1, 2, 2)^T$.

- Primal: $w = \frac{\sum_i x_i y_i}{\sum_i x_i^2 + \lambda} = \frac{1 + 4 + 6}{14 + 1} = \frac{11}{15}$.
- Dual: $K = X X^T = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}$, and solving $(K + I)\alpha = \mathbf{y}$ gives $\alpha = \frac{1}{15}(4, 8, -3)^T$. Check:
 $X^T \alpha = \frac{4+16-9}{15} = \frac{11}{15} = w$. ✓
- Prediction at $z = 4$: primal $4 \cdot \frac{11}{15} = \frac{44}{15}$; dual $\sum_i \alpha_i (x_i z) = \frac{4 \cdot 4 + 8 \cdot 8 - 3 \cdot 12}{15} = \frac{44}{15}$. ✓

The intercept. As written, kernel ridge regression has no bias term b ($h(\mathbf{z}) = \sum_i \alpha_i K(\mathbf{x}_i, \mathbf{z})$, and `sklearn's KernelRidge` is the same). Without one, the model is pulled toward predicting 0 far from the data, and the linear kernel can only fit lines through the origin. The simplest fix is to **center the labels**: fit on $\mathbf{y} - \bar{y}$ (with $\bar{y}$ the training mean) and add $\bar{y}$ back to every prediction, $h(\mathbf{z}) = \bar{y} + \sum_i \alpha_i K(\mathbf{x}_i, \mathbf{z})$. (Alternatives: add a constant feature, which for kernels means using $K + 1$; its weight is then regularized too.)

Primal or dual? The primal solves a $D \times D$ system, the dual an $N \times N$ one. The dual is preferable when $D \gg N$, and it is the only option when Φ is huge or infinite (Gaussian kernel). Its costs: storing K (N^2 numbers), solving the system ($\sim N^3$), and a prediction that touches every training point.

Remark. Bandwidth. With the RBF kernel, σ (together with λ) controls how wiggly the fitted curve is. Both are hyperparameters, chosen on a validation set.

9 The Dual Perceptron

Primal (Perceptron note, labels $y \in \{-1, +1\}$, dummy feature $x^0 = 1$, $\eta = 1$): start at $\mathbf{w} = \mathbf{0}$; whenever $y_j(\mathbf{x}_j \mathbf{w}) \leq 0$ (a mistake), update $\mathbf{w} \leftarrow \mathbf{w} + y_j \mathbf{x}_j^T$.

Duality. $\mathbf{w}$ starts at zero and only ever changes by adding $\pm$ a datapoint. So at any time

$$\mathbf{w} = \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i^T, \quad \alpha_i = \text{number of mistakes made so far on datapoint } i. \quad (10)$$

Key point: The dual variables have a direct meaning: α_i counts how many times $\mathbf{x}_i$ was a mistake. Points that are always classified correctly have $\alpha_i = 0$ and do not appear in the classifier at all; the hard points, near the boundary, carry all the weight.

Dual. The score of datapoint j becomes $\mathbf{x}_j \mathbf{w} = \sum_i \alpha_i y_i K_{ij}$, and an update on j is just $\alpha_j \leftarrow \alpha_j + 1$:

Algorithm: Dual (Kernel) Perceptron

1. Compute the kernel matrix $K_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$. Initialize $\alpha = \mathbf{0}$.
2. Repeat until a full pass makes no mistakes: for each j , if $y_j \sum_i \alpha_i y_i K_{ij} \leq 0$ then $\alpha_j \leftarrow \alpha_j + 1$.
3. Predict: $h(\mathbf{z}) = \text{sign} \left(\sum_i \alpha_i y_i K(\mathbf{x}_i, \mathbf{z}) \right)$.

With the linear kernel it makes *exactly* the same mistakes, in the same order, as the primal perceptron. (The dummy feature $x^0 = 1$ is part of every $\mathbf{x}$; for a general kernel the same effect comes from using $K+1$.)

Worked example: Boolean AND (the Perceptron note's dataset, $\mathbf{x} = (1, x^1, x^2)$):

i	$\mathbf{x}_i$	y_i
1	(1, 0, 0)	-1
2	(1, 0, 1)	-1
3	(1, 1, 0)	-1
4	(1, 1, 1)	+1

$$K = XX^T = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 2 \\ 1 & 1 & 2 & 2 \\ 1 & 2 & 2 & 3 \end{bmatrix}$$

First epoch, starting from $\alpha = (0, 0, 0, 0)$:

- $j = 1$: score 0 , $y_1 \cdot 0 \leq 0$: mistake, $\alpha = (1, 0, 0, 0)$.
- $j = 2$: score $\alpha_1 y_1 K_{12} = -1$, $y_2 \cdot (-1) = 1 > 0$: correct. $j = 3$: score $-K_{13} = -1$: correct.
- $j = 4$: score $-K_{14} = -1$, $y_4 \cdot (-1) < 0$: mistake, $\alpha = (1, 0, 0, 1)$.

Here $\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i^T = -\mathbf{x}_1^T + \mathbf{x}_4^T = (0, 1, 1)^T$, the same weights the primal perceptron has after its first epoch. Continuing, the algorithm makes 2, 3, 3, 2, 2, 3, 2, 1 mistakes in epochs 1–8 and none in epoch 9, ending with

$$\alpha = (2, 5, 4, 7), \quad \mathbf{w} = -2\mathbf{x}_1^T - 5\mathbf{x}_2^T - 4\mathbf{x}_3^T + 7\mathbf{x}_4^T = (-4, 3, 2)^T, \quad (11)$$

exactly the separating $\mathbf{w}^*$ quoted in the Perceptron note (scores $-4, -2, -1, +1$, all correct). Point 4, the only positive one, sits closest to the boundary and was a mistake most often.

Worked example: XOR with a polynomial kernel. Points $(1, 1), (1, -1), (-1, 1), (-1, -1)$ with labels $y = x^1 x^2 = +1, -1, -1, +1$. No line separates them; even with the bias ($K = XX^T + 1$) the perceptron makes 4 mistakes every epoch, forever. With $K(\mathbf{x}, \mathbf{z}) = (\mathbf{xz}^T + 1)^2$:

$$K = \begin{bmatrix} 9 & 1 & 1 & 1 \\ 1 & 9 & 1 & 1 \\ 1 & 1 & 9 & 1 \\ 1 & 1 & 1 & 9 \end{bmatrix}, \quad (12)$$

the first epoch makes one mistake per point ($\alpha = (1, 1, 1, 1)$), giving scores $\sum_i \alpha_i y_i K_{ij} = (8, -8, -8, 8)$, all correct: the second epoch is clean. The polynomial feature space contains $\sqrt{2}x^1 x^2$, in which XOR is linearly separable.

10 k -NN and Kernel Smoothing

k -NN (HW1) is already similarity-based. Replacing “vote among the k nearest” by “every training point votes, weighted by its similarity” gives

$$h(\mathbf{z}) = \arg \max_c \sum_{i=1}^N K(\mathbf{x}_i, \mathbf{z}) \mathbf{1}[y_i = c], \quad (13)$$

and, for regression, $h(\mathbf{z}) = \sum_i K(\mathbf{x}_i, \mathbf{z}) y_i / \sum_i K(\mathbf{x}_i, \mathbf{z})$. With the Gaussian kernel this is a smooth version of HW1’s fixed-window classifier: σ plays the role of the window radius. Dividing a class’s sum by the class size N_c gives the kernel density estimate of $P(\mathbf{z} \mid c)$ (HW1 Problem 4C, up to the kernel’s normalizing constant); multiplying back by the prior N_c/N recovers the plain sum, so the two classifiers agree. (See the Local Classifiers note.)

11 Kernel PCA

11.1 Recap: PCA

From the PCA note, with centered data $\tilde{X}$ ($N \times D$) and $\Sigma = \frac{1}{N} \tilde{X}^T \tilde{X}$:

- The principal directions are eigenvectors, $\Sigma \mathbf{e} = \lambda \mathbf{e}$, $\|\mathbf{e}\| = 1$, sorted by decreasing λ .
- The new coordinate of a (centered) datapoint on direction $\mathbf{e}$ is $\tilde{\mathbf{x}}\mathbf{e}$; for all datapoints at once, $\tilde{X}\mathbf{e}$ ($N \times 1$).
- Each direction is a combination of the datapoints: $\mathbf{e} = \tilde{X}^T \beta$ (PCA note, §8.1).

PCA finds the best *linear* aggregates of the features. On data like concentric circles or spirals (HW3 Problem 4) no linear aggregate separates the structure. Goal: do PCA in a feature space Φ , using only a kernel.

11.2 PCA Written with Dot Products Only

Let $K = \tilde{X}\tilde{X}^T$ ($N \times N$) and substitute $\mathbf{e} = \tilde{X}^T\beta$ into $\Sigma\mathbf{e} = \lambda\mathbf{e}$:

$$\frac{1}{N}\tilde{X}^T\tilde{X}\tilde{X}^T\beta = \lambda\tilde{X}^T\beta. \quad (14)$$

Multiply on the left by $\tilde{X}$ and use $\tilde{X}\tilde{X}^T = K$:

$$K K\beta = N\lambda K\beta, \quad (15)$$

which is satisfied by every solution of

$$\boxed{K\beta = (N\lambda)\beta} \quad (16)$$

So the dual vector β is an **eigenvector of the kernel matrix**, with eigenvalue $\nu = N\lambda$.

- **Coordinates** of all training points on direction $\mathbf{e}$: $\tilde{X}\mathbf{e} = \tilde{X}\tilde{X}^T\beta = K\beta$. For a new (centered) point $\tilde{\mathbf{z}}$: $\tilde{\mathbf{z}}\mathbf{e} = \sum_i \beta_i \tilde{\mathbf{z}}\tilde{\mathbf{x}}_i^T$, again only dot products.
- **Normalization** (easy to forget, and it changes the results). We need $\|\mathbf{e}\| = 1$: $\|\mathbf{e}\|^2 = \beta^T\tilde{X}\tilde{X}^T\beta = \beta^TK\beta = \nu\beta^T\beta$. So $\|\beta\| = 1/\sqrt{\nu}$. An eigen-solver returns a unit-length eigenvector $\mathbf{u}$; rescale it: $\beta = \mathbf{u}/\sqrt{\nu}$. Then the training coordinates are $K\beta = \nu\beta = \sqrt{\nu}\mathbf{u}$.
- With r directions, stack the β 's as columns of B ($N \times r$): all coordinates at once are KB ($N \times r$).

Check on the PCA note's example (linear kernel). Centered rows $(3, 1), (1, 3), (-1, -3), (-3, -1)$:

$$K = \tilde{X}\tilde{X}^T = \begin{bmatrix} 10 & 6 & -6 & -10 \\ 6 & 10 & -10 & -6 \\ -6 & -10 & 10 & 6 \\ -10 & -6 & 6 & 10 \end{bmatrix}, \quad (17)$$

with eigenvalues $32, 8, 0, 0$, i.e. $N\lambda$ for PCA's $\lambda = 8, 2$ (with $N = 4$). The top unit eigenvector is $\mathbf{u} = \frac{1}{2}(1, 1, -1, -1)$ (up to sign), so the coordinates are $\sqrt{32}\mathbf{u} = (2.83, 2.83, -2.83, -2.83) = \pm\frac{4}{\sqrt{2}}$, exactly PCA's first coordinates. ✓

11.3 The Kernel Trick: Kernel PCA

Nothing above used the features except through $K_{ij} = \tilde{\mathbf{x}}_i\tilde{\mathbf{x}}_j^T$. Replace it with a kernel: $K_{ij} = K(\mathbf{x}_i, \mathbf{x}_j) = \Phi(\mathbf{x}_i)\Phi(\mathbf{x}_j)^T$. This is PCA on the mapped data $\Phi(X)$ — principal directions in a space we never construct. With the Gaussian kernel, the first few Kernel PCA coordinates can untangle concentric circles or spirals so that a *linear* classifier on them succeeds (HW3 Problem 4).

PCA is “rotate the data so its axes line up with the eigenvectors, then keep the top t ” (Figure 4, top). Kernel PCA does the same rotation, but after the map Φ , which it never computes: only K is used (bottom). A level line of the new coordinate z_1 is straight in the feature space, as in PCA, but curved back in the input space.

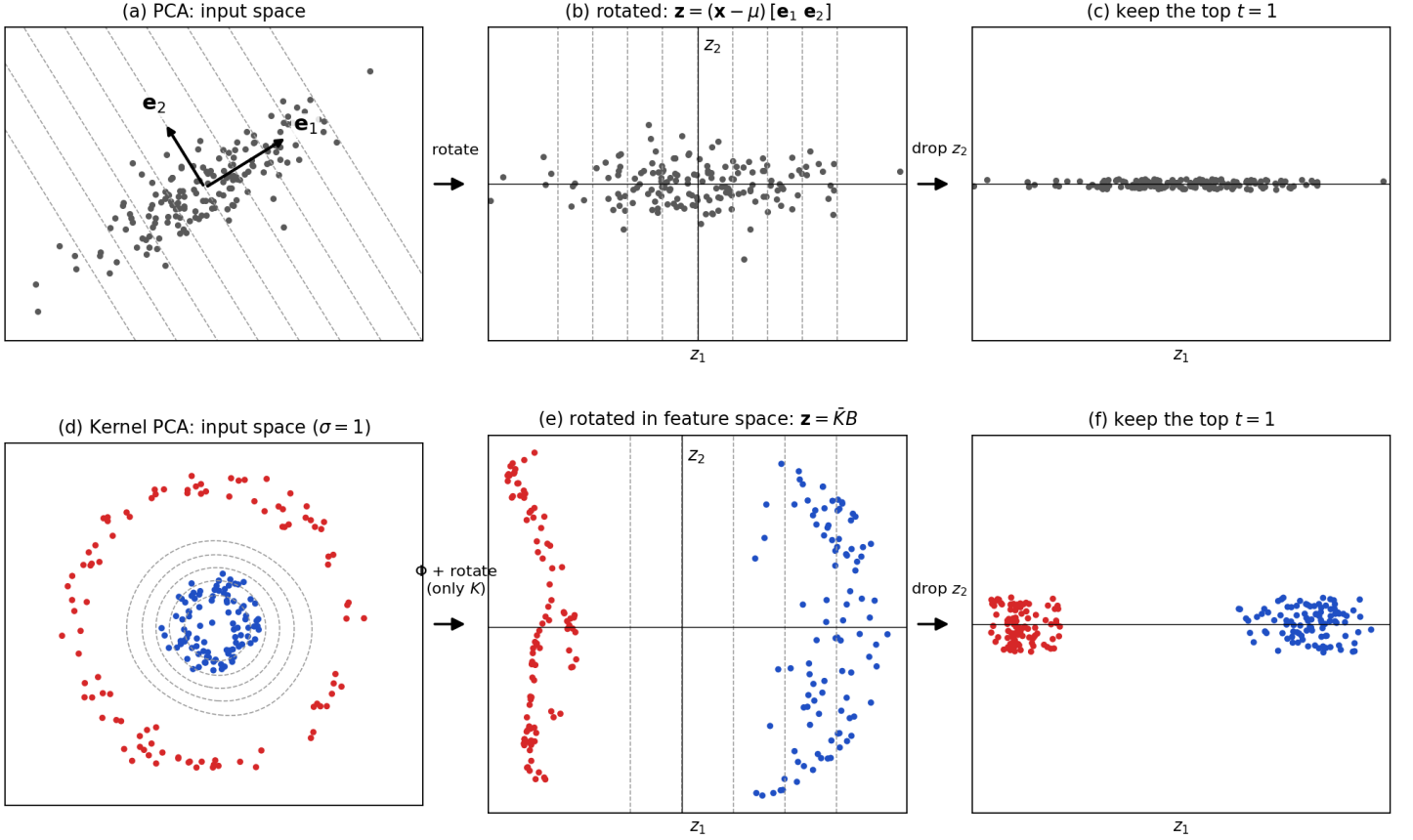


Figure 4: Top: PCA on a 2-D cloud. Dashed lines are points with equal $z_1 = (\mathbf{x} - \mu)\mathbf{e}_1$; rotating makes them vertical, and keeping $t = 1$ drops z_2 . Bottom: Gaussian Kernel PCA ($\sigma = 1$) on two rings. The same steps, done in the feature space: the level lines of z_1 are circles in the input space (d) and vertical lines in the new coordinates (e), and z_1 alone already separates the rings (f).

11.4 Centering Inside K

PCA requires centered data, but in the feature space we cannot subtract the mean $\mathbf{m} = \frac{1}{N} \sum_t \Phi(\mathbf{x}_t)$: we never compute Φ . Instead, compute the kernel of the centered mapped points directly. Writing $\Phi_i = \Phi(\mathbf{x}_i)$:

$$\bar{K}_{ij} = (\Phi_i - \mathbf{m})(\Phi_j - \mathbf{m})^T = \left(\Phi_i - \frac{1}{N} \sum_t \Phi_t \right) \left(\Phi_j - \frac{1}{N} \sum_l \Phi_l \right)^T \quad (18)$$

$$= K_{ij} - \frac{1}{N} \sum_t K_{tj} - \frac{1}{N} \sum_l K_{il} + \frac{1}{N^2} \sum_{t,l} K_{tl} \quad (19)$$

(original entry, minus the column mean, minus the row mean, plus the grand mean). With U the $N \times N$ matrix with every entry $1/N$:

$$\boxed{\bar{K} = K - UK - KU + UKU} \quad (20)$$

(The last sign is +: the grand mean is subtracted twice by the two middle terms and must be added back once. A quick check: every row of $\bar{K}$ sums to 0.)

New (test) points. For a test point $\mathbf{z}$, the same expansion with $\Phi(\mathbf{z}) - \mathbf{m}$ on one side, centered with the *training* mean:

$$\bar{K}(\mathbf{x}_i, \mathbf{z}) = K(\mathbf{x}_i, \mathbf{z}) - \frac{1}{N} \sum_l K(\mathbf{x}_l, \mathbf{z}) - \frac{1}{N} \sum_l K_{il} + \frac{1}{N^2} \sum_{t,l} K_{tl}. \quad (21)$$

For M test points at once, with K_{test} ($M \times N$, entry $K(\mathbf{z}_m, \mathbf{x}_i)$) and U' the $M \times N$ matrix of $1/N$'s:

$$\boxed{\bar{K}_{\text{test}} = K_{\text{test}} - U'K - K_{\text{test}}U + U'KU}, \quad \text{test coordinates} = \bar{K}_{\text{test}} B. \quad (22)$$

Algorithm: Kernel PCA

1. **Fit:** training kernel K ($N \times N$); center it: $\bar{K} = K - UK - KU + UKU$.
 2. Eigendecompose $\bar{K}$; sort by decreasing eigenvalue $\nu_1 \geq \nu_2 \geq \dots$; keep the top r with $\nu > 0$; $\beta_k = \mathbf{u}_k / \sqrt{\nu_k}$, stacked into B .
 3. **Transform training data:** $\bar{K}B$ (equivalently $\mathbf{u}_k \sqrt{\nu_k}$ in column k).
 4. **Transform test data:** K_{test} ($M \times N$, kernel against the *training* points); center with the training statistics: $\bar{K}_{\text{test}} = K_{\text{test}} - U'K - K_{\text{test}}U + U'KU$; coordinates $\bar{K}_{\text{test}}B$.
-

12 Why This Always Works: The Representer Theorem

In each case above the solution turned out to be a combination of datapoints, $\mathbf{w} = \sum_i \alpha_i \mathbf{x}_i^T$. This is not luck.

Key point: Representer Theorem. For any loss function L and $\lambda > 0$, the regularized problem

$$\min_{\mathbf{w}} \sum_{i=1}^N L(y_i, \mathbf{x}_i \mathbf{w}) + \lambda \|\mathbf{w}\|^2$$

has a minimizer in the span of the datapoints: $\mathbf{w}^* = \sum_{i=1}^N \alpha_i \mathbf{x}_i^T$ for some α . (In a feature space: $\mathbf{w}^* = \sum_i \alpha_i \Phi(\mathbf{x}_i)^T$.)

Why. Split any $\mathbf{w}$ into a part in the span of the datapoints and a part perpendicular to all of them, $\mathbf{w} = \mathbf{w}_{\parallel} + \mathbf{w}_{\perp}$. The perpendicular part does not change any prediction on the training data ($\mathbf{x}_i \mathbf{w}_{\perp} = 0$ for every i), so the loss term is unchanged, but it adds $\|\mathbf{w}_{\perp}\|^2$ to the penalty ($\|\mathbf{w}\|^2 = \|\mathbf{w}_{\parallel}\|^2 + \|\mathbf{w}_{\perp}\|^2$). Dropping $\mathbf{w}_{\perp}$ can only lower the objective, so there is a minimizer with $\mathbf{w}_{\perp} = 0$ (and if L is convex, the minimizer is unique and in the span).

Consequences.

- Substitute $\mathbf{w} = X^T \alpha$ into the objective: predictions become $K\alpha$ and the penalty $\|\mathbf{w}\|^2 = \alpha^T K \alpha$. The problem becomes an N -dimensional problem in α that uses only K — kernelizable, even if Φ is infinite-dimensional.
- Ridge regression is the case $L = \text{squared error}$ (§8). Regularized logistic regression is the case $L = \text{log-loss}$, giving kernel logistic regression (optional below).
- The perceptron and PCA are not of this exact form, but their solutions lie in the span of the data for the reasons we saw: the perceptron builds $\mathbf{w}$ by adding datapoints, and $\Sigma \mathbf{e} = \lambda \mathbf{e}$ forces $\mathbf{e} = \tilde{X}^T \beta$.
- The regularizer is what makes this work: it removes every component of $\mathbf{w}$ that the data cannot “see.”

13 Implementation Notes (HW3)

- **Problem 2** (kernel ridge regression): compute K on the training data with a library kernel function (`sklearn.metrics.pairwise.rbf_kernel`, `polynomial_kernel`, `linear_kernel`); center the labels ($\mathbf{y} - \bar{y}$), solve $(K + \lambda I)\alpha = \mathbf{y} - \bar{y}$ with `np.linalg.solve` (not an explicit inverse); predict $\bar{y}$ plus the $M \times N$ test-vs-train kernel times α . Center the labels the same way for the library baseline. Library check: `sklearn`'s `KernelRidge` with `alpha =  $\lambda$` and `gamma =  $1/\sigma^2$` solves the same system. Choose λ , σ (or degree) on a validation split.
- **Problem 4** (Kernel PCA): follow the algorithm box. The three classic bugs: forgetting to center (or centering test data with its own statistics), forgetting the $1/\sqrt{\nu}$ normalization, and `np.linalg.eigh`'s increasing eigenvalue order. Library check: `sklearn.decomposition.KernelPCA(n_components=r, kernel='rbf', gamma=1/ $\sigma^2$ )`; its `transform` matches $\bar{K}B$ and $\bar{K}_{\text{test}}B$ up to the sign of each column. Drop eigenvalues that are ≤ 0 (numerical noise).
- **Cost**: kernel methods store $N \times N$ matrices. Fine for HW3's datasets (about 1000 points); for large N , subsample.

14 Optional Material

Nothing in this section is needed for HW3's required problems.

14.1 Kernel Logistic Regression

Regularized logistic regression, $\min_{\mathbf{w}} - \sum_i [y_i \log p_i + (1 - y_i) \log(1 - p_i)] + \lambda \|\mathbf{w}\|^2$ with $p_i = \text{sigmoid}(\mathbf{x}_i \mathbf{w})$ and $y_i \in \{0, 1\}$, fits the Representer Theorem. With $\mathbf{w} = X^T \alpha$, the scores are $K\alpha$ and the penalty $\alpha^T K \alpha$:

$$J(\alpha) = - \sum_i [y_i \log p_i + (1 - y_i) \log(1 - p_i)] + \lambda \alpha^T K \alpha, \quad p = \text{sigmoid}(K\alpha), \quad (23)$$

$$\nabla_{\alpha} J = K(p - \mathbf{y}) + 2\lambda K\alpha, \quad (24)$$

and gradient descent on α is the GD/Logistic Regression note's algorithm, with the kernel in place of the features. Predict $\text{sigmoid}(\sum_i \alpha_i K(\mathbf{x}_i, \mathbf{z}))$.

14.2 Mercer's Theorem

Let K be continuous and symmetric on a compact domain, with $\iint K(\mathbf{x}, \mathbf{z}) f(\mathbf{x}) f(\mathbf{z}) d\mathbf{x} d\mathbf{z} \geq 0$ for every square-integrable f (the continuous version of "positive semidefinite"). Then $K(\mathbf{x}, \mathbf{z}) = \sum_{j=1}^{\infty} \lambda_j \phi_j(\mathbf{x}) \phi_j(\mathbf{z})$ with $\lambda_j \geq 0$ — a feature map $\Phi = (\sqrt{\lambda_1} \phi_1, \sqrt{\lambda_2} \phi_2, \dots)$ exists. This is the continuous counterpart of the finite-data construction $\Phi(X) = U\Lambda^{1/2}$ in §3.

14.3 Scaling Up: Random Fourier Features and Nyström

The problem. Every kernel method in this note stores the $N \times N$ matrix K and solves with it: memory grows like N^2 , kernel ridge regression's solve like N^3 , and each prediction needs $K(\mathbf{x}_i, \mathbf{z})$ for all N training points. Fine for HW3's 1000 points, hopeless for a million. The fix: go back to an *explicit*, finite feature map $\mathbf{z}(\mathbf{x})$ with R features, chosen so that $\mathbf{z}(\mathbf{x})\mathbf{z}(\mathbf{x}')^T \approx K(\mathbf{x}, \mathbf{x}')$, and run the ordinary *linear* algorithm on it. Cost: $N \times R$ instead of $N \times N$.

Random Fourier features (Rahimi & Recht, 2007). A kernel that depends only on $\mathbf{x} - \mathbf{x}'$ (like the Gaussian) can be written as an average of cosines over random frequencies (Bochner’s theorem). For our Gaussian kernel $\exp(-\|\mathbf{x} - \mathbf{x}'\|^2/\sigma^2)$:

Random Fourier features for the Gaussian kernel

Draw a $D \times R$ matrix W with independent entries from $\mathcal{N}(0, 2/\sigma^2)$, and R phases b_r uniform on $[0, 2\pi]$.

Map each datapoint: $\mathbf{z}(\mathbf{x}) = \sqrt{2/R} \cos(\mathbf{x}W + \mathbf{b})$ (a $1 \times R$ row; cosine taken entrywise).

Then $\mathbf{z}(\mathbf{x})\mathbf{z}(\mathbf{x}')^T \approx K(\mathbf{x}, \mathbf{x}')$, and the error shrinks like $1/\sqrt{R}$.

Example: on 200 random 3-D points ($\sigma = 1.5$), the largest entry of $|ZZ^T - K|$ was 0.39 with $R = 100$, 0.13 with $R = 1000$, and 0.04 with $R = 10,000$. Watch the variance: it is $2/\sigma^2$ for this note’s convention (σ^2 in the denominator, not $2\sigma^2$); with variance $1/\sigma^2$ the approximation converges to the wrong kernel.

Then just go linear. Kernel ridge regression becomes ordinary ridge regression on the $N \times R$ matrix Z : solve an $R \times R$ system instead of $N \times N$, and predict with $\mathbf{z}(\mathbf{x})\mathbf{w}$ (cost R , not N). The same works for a linear SVM (e.g. Pegasos, SVM Advanced note, §7) or logistic regression on Z .

Nystrom. Instead of random frequencies, use M randomly chosen training points as “landmarks” and approximate

$$K \approx K_{NM} K_{MM}^{-1} K_{MN},$$

where K_{NM} holds the kernel values between all N points and the M landmarks. This is exact for the landmarks themselves and good when K has only a few large eigenvalues (as in Kernel PCA, a few directions carry most of the variation). On the same 200 points, the relative error was 41% with $M = 10$ landmarks and 6% with $M = 50$.

Library: in `sklearn.kernel_approximation`, `RBFSampler` (with `gamma=1/\sigma^2`, `n_components=R`) and `Nystroem` (with `kernel='rbf'`, `gamma=1/\sigma^2`, `n_components=M`), followed by any linear model. (Optional HW3 Problem 9 compares random features with exact kernel ridge regression on WAVE.)