14 The Kernel Trick

KERNELS

Recall: with d input features, degree-p polynomials blow up to $O(d^p)$ features.
[When d is large, this gets computationally intractable really fast.
As I said in Lecture 4, if you have 100 features per feature vector and you want to use degree-4 decision functions, then each lifted feature vector has a length of roughly 4 million.]
Today, magically, we use those features without computing them!

Observation: In many learning algs,
- the weights can be written as a linear combo of sample points, &
- we can use inner products of $\Phi(x)$'s only $\Rightarrow$ don't need to compute $\Phi(x)$!

$$\text{Suppose } w = X^\top a = \sum_{i=1}^{n} a_i X_i \quad \text{for some } a \in \mathbb{R}^n.$$

Substitute this identity into alg. and optimize n dual weights a (aka dual parameters) instead of $d+1$ (or d^p) primal weights w.

Kernel Ridge Regression

Center X and y so their means are zero: $X_i \leftarrow X_i - \mu_X$, $y_i \leftarrow y_i - \mu_y$
This lets us replace I' with I in normal equations:

$$(X^\top X + \lambda I)w = X^\top y$$

[To dualize ridge regression, we need the weights to be a linear combination of the sample points. Unfortunately, that only happens if we penalize the bias term $w_{d+1} = \alpha$, as these normal equations do. Fortunately, when we center X and y, the "expected" value of the intercept is zero. The actual value won't usually be exactly zero, but it will often be close enough that we won't do much harm by penalizing the intercept.]

$$w = \frac{1}{\lambda}(X^\top y - X^\top X w) = X^\top a \quad \text{where } a = \frac{1}{\lambda}(y - Xw)$$

This shows that w is a linear combo of sample points! To compute a:

$$\lambda a = y - Xw = y - XX^\top a \quad \Rightarrow \quad a = (XX^\top + \lambda I)^{-1} y$$

a is a dual solution; solves the dual form of ridge regression:

$$\boxed{\text{Find } a \text{ that minimizes } |XX^\top a - y|^2 + \lambda |X^\top a|^2}$$

[We obtain this dual form by substituting $w = X^\top a$ into the original ridge regression cost function.]

Regression fn is

$$h(z) = w^\top z = a^\top X z = \sum_{i=1}^{n} a_i \, (X_i^\top z) \qquad \Leftarrow \text{ weighted sum of inner products}$$

Let $k(x, z) = x^\top z$ be kernel fn.
[Later, we'll replace x and z with $\Phi(x)$ and $\Phi(z)$, and that's where the magic will happen.]

Let $K = XX^\top$ be $n \times n$ kernel matrix. Note $K_{ij} = k(X_i, X_j)$.

K is singular if $n > d + 1$. [And sometimes even if it's not.]
In that case, no solution if $\lambda = 0$. [Then we must choose a positive λ. But that's okay.]

Dual ridge reg. alg:

```
$\forall i, j$,   $K_{ij} \leftarrow k(X_i, X_j)$             $\Leftarrow O(n^2 d)$ time
Solve $(K + \lambda I)\, a = y$ for $a$          $\Leftarrow O(n^3)$ time
for each test pt $z$
    $h(z) \leftarrow \sum_{i=1}^{n} a_i\, k(X_i, z)$          $\Leftarrow O(nd)$ time
```

Does not use X_i directly! Only k. [This will become important soon.]

Dual: solve $n \times n$ linear system, $O(n^3 + n^2 d)$ time
Primal: " $d \times d$ " " , $O(d^3 + d^2 n)$ time
We prefer dual when $d > n$. [Moreover, if we add polynomial terms as new features, the d in the primal running time increases, but we will see that the d in the kernelized dual running time does not increase.]

The Kernel Trick (aka Kernelization)

[Here's the magic part. We will see that we can compute a polynomial kernel that involves many monomial terms without actually computing those terms.]

The polynomial kernel of degree p is $k(x, z) = (x^\top z + 1)^p$

Theorem: $(x^\top z + 1)^p = \Phi(x)^\top \Phi(z)$ where $\Phi(x)$ contains every monomial in x of degree $0 \ldots p$.

Example for $d = 2$, $p = 2$:

$$\begin{aligned} (x^\top z + 1)^2 &= x_1^2 z_1^2 + x_2^2 z_2^2 + 2 x_1 z_1 x_2 z_2 + 2 x_1 z_1 + 2 x_2 z_2 + 1 \\ &= [x_1^2 \quad x_2^2 \quad \sqrt{2} x_1 x_2 \quad \sqrt{2} x_1 \quad \sqrt{2} x_2 \quad 1]\, [z_1^2 \quad z_2^2 \quad \sqrt{2} z_1 z_2 \quad \sqrt{2} z_1 \quad \sqrt{2} z_2 \quad 1]^\top \\ &= \Phi(x)^\top \Phi(z) \qquad \text{[This is how we're defining } \Phi.\text{]} \end{aligned}$$

[Notice the factors of $\sqrt{2}$. If you try a higher polynomial degree p, you'll see a wider variety of these constants. We have no control of the constants that appear in $\Phi(x)$, but they don't matter very much, because the primal weights w will scale themselves to compensate. Even though we won't be directly computing the primal weights ... they still implicitly exist.]

Key win: compute $\Phi(x)^\top \Phi(z)$ in $O(d)$ time instead of $O(d^p)$, even though $\Phi(x)$ has length $O(d^p)$.

Kernel ridge regression replaces X_i with $\Phi(X_i)$:

Let $k(x, z) = \Phi(x)^\top \Phi(z)$ But don't compute $\Phi(x)$ or $\Phi(z)$; compute $k(x, z) = (x^\top z + 1)^p$!

[I think what we've done here is pretty mind-blowing: we can now do polynomial regression with an exponentially long, high-order polynomial in less time than it would take even to compute the final polynomial. The running time is sublinear, actually much smaller than linear, in the length of the Φ vectors.]

Kernel Perceptrons

Featurized perceptron alg:

```
$w \leftarrow y_1\,\Phi(X_1)$                    [starting point is arbitrary, but can't be zero]
while some $y_i\,\Phi(X_i)\cdot w < 0$
    $w \leftarrow w + \epsilon\, y_i\,\Phi(X_i)$
for each test pt $z$
    $h(z) \leftarrow w \cdot \Phi(z)$
```

Let $\Phi(X)$ be $n \times D$ matrix with rows $\Phi(X_i)^\top$, D = length of $\Phi(\cdot)$, $K = \Phi(X)\,\Phi(X)^\top$
Dualize with $w = \Phi(X)^\top a$. Then the code "$a_i \leftarrow a_i + \epsilon\, y_i$" has same effect as "$w \leftarrow w + \epsilon\, y_i\,\Phi(X_i)$"
[So the dual weight a_i records what multiple of sample point i the perceptron algorithm has added to w.]
$\Phi(X_i)\cdot w = (\Phi(X)w)_i = (\Phi(X)\,\Phi(X)^\top a)_i = (Ka)_i$

Dual perceptron alg:

```
$a \leftarrow [y_1 \quad 0 \quad \dots \quad 0]^\top$          [starting point is arbitrary, but can't be zero]
$\forall i, j, \quad K_{ij} \leftarrow k(X_i, X_j)$       $\Leftarrow O(n^2 d)$ time (kernel trick)
while some $y_i\,(Ka)_i < 0$
    $a_i \leftarrow a_i + \epsilon\, y_i$               $\Leftarrow O(1)$ time; update $Ka$ in $O(n)$ time
for each test pt $z$
    $h(z) \leftarrow \sum_{j=1}^{n} a_j\, k(X_j, z)$     $\Leftarrow O(nd)$ time [kernel trick]
```

[A big deal is that the running times depend on the original d, not on the length D of $\Phi(\cdot)$!]
OR we can compute $w = \Phi(X)^\top a$ once in $O(nD)$ time & evaluate test pts in $O(D)$ time/pt
[...which is a win if the numbers of training points and test points both exceed D/d.]

Kernel Logistic Regression

[The stochastic gradient descent step for logistic regression is just a small modification of the step for perceptrons. But recall that we're no longer looking for misclassified sample points. Instead, we apply the gradient descent rule to sample points in a stochastic, random order—or, alternatively, to all the points at once.]

Stochastic gradient descent step:

$$a_i \leftarrow a_i + \epsilon\,(y_i - s((Ka)_i)) \qquad \text{[where } s \text{ is the logistic function]}$$

[Just like with perceptrons, every time you update one weight a_i, you can update Ka in $O(n)$ time so you don't have to compute it from scratch on the next iteration. If you prefer batch gradient descent ...]

Batch gradient descent step:

$$a \leftarrow a + \epsilon\,(y - s(Ka)) \qquad \Leftarrow \text{ applying } s \text{ component-wise to vector } Ka$$

For each test pt z:

$$h(z) \leftarrow s\left(\sum_{j=1}^{n} a_j\, k(X_j, z)\right)$$

[If you're using logistic regression as a classifier and you don't care about the posterior probabilities, you can skip the logistic function and just compute the summation, like in the perceptron algorithm.]

The Gaussian Kernel

[Mind-blowing as the polynomial kernel is, I think our next trick is even more mind-blowing. Since we can now do fast computations in spaces with exponentially large dimensions, why don't we go all the way and generate feature vectors in infinite-dimensional space?]

Gaussian kernel, aka radial basis fn kernel: there exists a $\Phi(x)$ such that

$$k(x, z) = \exp\left(-\frac{|x - z|^2}{2\sigma^2}\right) \qquad \text{[This kernel takes } O(d) \text{ time to compute.]}$$

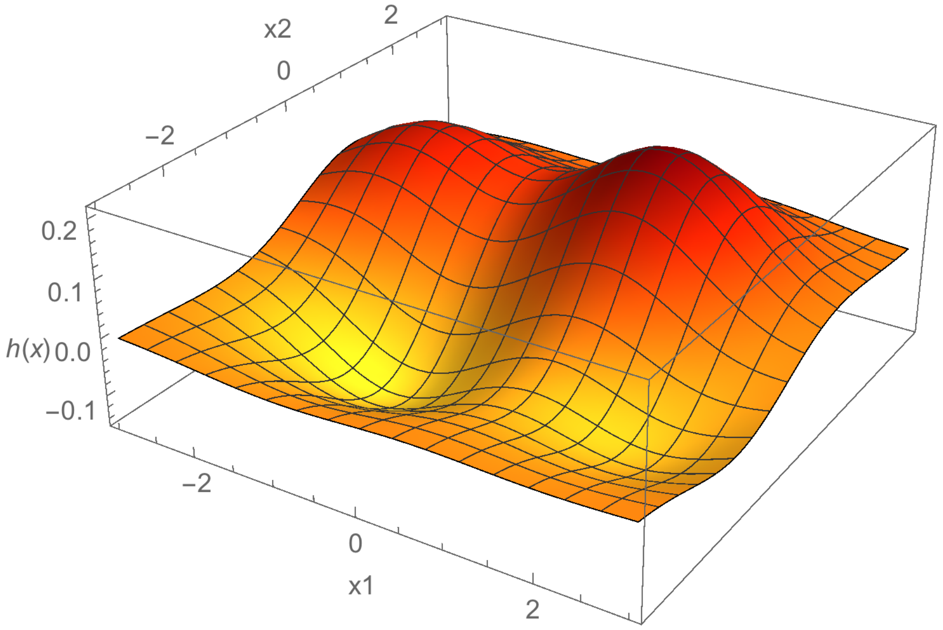
[In case you're curious, here's the feature vector that gives you this kernel, for the case where you have only one input feature per sample point.]

e.g. for $d = 1$,

$$\Phi(x) = \exp\left(-\frac{x^2}{2\sigma^2}\right)\left[1, \frac{x}{\sigma\sqrt{1!}}, \frac{x^2}{\sigma^2\sqrt{2!}}, \frac{x^3}{\sigma^3\sqrt{3!}}, \ldots\right]^\top$$

[This is an infinite vector, and $\Phi(x)\cdot\Phi(z)$ is a series that converges to $k(x, z)$. Nobody actually uses this value of $\Phi(x)$ directly, or even cares about it; they just use the kernel function $k(\cdot, \cdot)$.]
[At this point, it's best *not* to think of points in a high-dimensional space. It's no longer a useful intuition. Instead, think of the kernel k as a measure of how similar or close together two points are to each other.]

Key observation: hypothesis $h(z) = \sum_{j=1}^{n} a_j\, k(X_j, z)$ is a linear combo of Gaussians centered at sample pts.
[The dual weights are the coefficients of the linear combination.]
[The Gaussians are a basis for the hypothesis.]



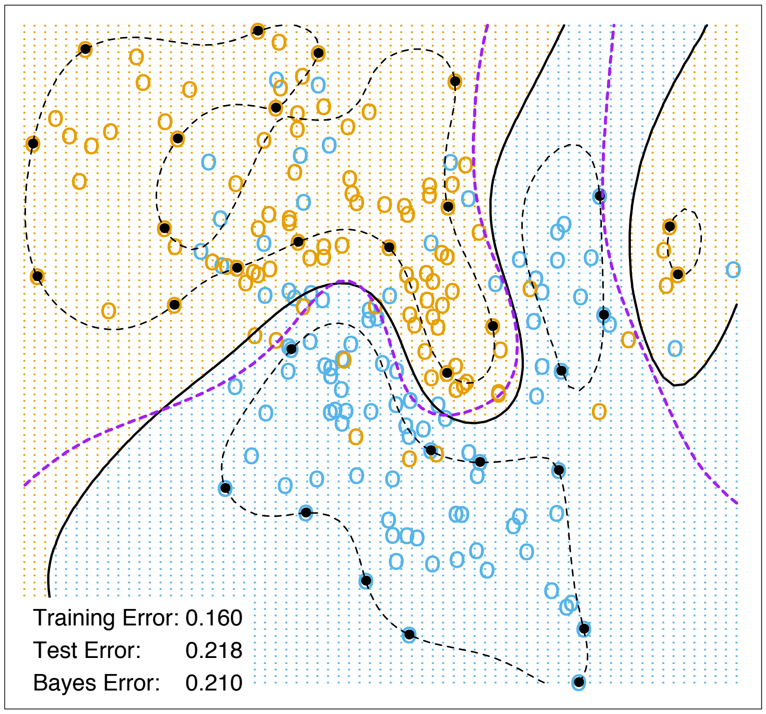
gausskernel.pdf [A hypothesis h that is a linear combination of Gaussians centered at four sample points, two with positive weights and two with negative weights. If you use ridge regression with a Gaussian kernel, your "linear" regression will look something like this.]

Very popular in practice! Why?

– Gives very smooth h [In fact, h is infinitely differentiable; it's C^∞-continuous.]
– Behaves somewhat like k-nearest neighbors, but smoother
– Oscillates less than polynomials (depending on σ)
– $k(x, z)$ interpreted as a similarity measure. Maximum when $z = x$; goes to 0 as distance increases.
– Sample points "vote" for value at z, but closer points get weightier vote.

[The "standard" kernel $x \cdot z$ assigns more weight to vectors that point in roughly the same direction as z. By contrast, the Gaussian kernel assigns more weight to sample points near z.]

σ trades off bias vs. variance:
larger σ $\rightarrow$ wider Gaussians, smoother h $\rightarrow$ more bias, less variance
Choose σ by (cross-)validation.



$\boxed{\text{gausskernelsvm.pdf (ESL, Figure 12.3)}}$ [The decision boundary (solid black) of a soft-margin SVM with a Gaussian kernel. Observe that in this example, it comes reasonably close to the Bayes optimal decision boundary (dashed purple). The dashed black curves are the boundaries of the margin. The small black disks are the support vectors that lie on the margin boundary.]

[By the way, there are many other kernels that, like the Gaussian kernel, are defined directly as kernel functions without worrying about Φ. But not every function can be a kernel function. A function is qualified only if it always generates a positive semidefinite kernel matrix, for every sample. There is an elaborate theory about how to construct valid kernel functions. However, you probably won't need it. The polynomial and Gaussian kernels are the two most popular by far.]