6.867 Section 4: Kernels

Contents

1 Intro

So far, we have spent a lot of time looking at linear classifiers and regression methods, and finding that they arise in various places, expected (when we explicitly search for one as in SVMs) and unexpected (when Naive Bayes or LDA turns out to generate a linear separator).

We have also seen that, using an explicit transformation of the original input space to a new, typically higher-dimensional, *feature space*, we can find more complex hypotheses. These hypotheses are linear in the feature space, but non-linear when they are re-interpreted in the original input space.

Let x be an element of the original input space, generally $\mathbb{R}^d$; then $\Phi(x)$ is the corresponding element of the feature space, generally $\mathbb{R}^D$ where $D > d$. We already understand that we can use $\Phi(x)$ in place of x everywhere in our learning algorithms, and get solutions that are non-linear in the input space.

However, there are two big worries about this approach:

1. *Representational capacity and computational efficiency*: As D increases, the computational complexity of the learning algorithms increases.

2. *Model selection and regularization*: As D increases, the hypothesis space gets bigger and the risk of overfitting increases.

Maximizing the margin, and other regularization techniques can help address the problem of overfitting in a large hypothesis space. We'll concentrate in the next two lectures on a method for addressing the problem of learning in very large spaces.

2 Kernels

Following the development in *Kernel Methods for Pattern Analysis* by Shawe-Taylor and Cristianini.

The term "kernel" has two different, but ultimately related, uses in Machine Learning. Later in the course we'll talk about kernel density estimation, which uses kernels in a different way.

If we look at the dual form of the SVM or the perceptron, we see that it depends on the data only through dot products: $x^{(i)} \cdot x^{(j)}$. If we apply a feature mapping ϕ, then it will depend on $\phi(x^{(i)}) \cdot \phi(x^{(j)})$. We will call the matrix of these dot-products of feature vectors the *gram matrix* or *kernel matrix*:

$$G_{ij} = \phi(x^{(i)}) \cdot \phi(x^{(j)}) \quad .$$

This matrix is size $n \times n$, but when D is very high, then it might be easier to deal with than something that is $D \times D$.

A *kernel* is a function κ that for all $x, z \in \mathbb{R}^d$ satisfies

We will generalize this to other spaces later.

$$\kappa(x, z) = \phi(x) \cdot \phi(z) \quad ,$$

for some ϕ mapping from $\mathbb{R}^d$ to $\mathbb{R}^D$.

The magic idea here is that, for many kernels, there is a way to compute $\kappa(x, z)$ *without computing* $\phi(x)$ *and* $\phi(z)$*, and so to be (relatively) independent of* D.

Example Let $d = 2$, $D = 3$, and

$$\phi((x_1, x_2)) = (x_1^2, x_2^2, \sqrt{2}x_1x_2) \quad .$$

In this case, the space of linear functions is

$$g(x) = w_1x_1^2 + w_2x_2^2 + w_3\sqrt{2}x_1x_2 \quad .$$

Let's look at the kernel function:

$$\begin{aligned}\kappa(x,z) &= \phi(x)\cdot\phi(z)\\ &= (x_1^2, x_2^2, \sqrt{2}x_1x_2)\cdot(z_1^2, z_2^2, \sqrt{2}z_1z_2)\\ &= x_1^2z_1^2 + x_2^2z_2^2 + 2x_1x_2z_1z_2\\ &= (x_1z_1 + x_2z_2)^2\\ &= (x\cdot z)^2\end{aligned}$$

So, cool! We can compute dot products of data points in feature space without ever explicitly constructing the feature vectors!

This same kernel function computes the dot product in feature space with feature mapping

$$\phi((x_1, x_2)) = (x_1^2, x_2^2, x_1x_2, x_2x_1) \ ,$$

so there need not be a unique ϕ for a particular κ.

This a special case of a more general *polynomial kernel*.

$$\kappa(x,z) = (x\cdot z + c)^p \ ,$$

for any positive integer p and input dimension d. For $p = 2$ it corresponds to

$$\phi((x_1,\ldots,x_d)) = (x_ix_j \mid i,j \in \{1\ldots d\}) + (\sqrt{2c}x_i \mid i \in \{1\ldots d\}) + (c) \ .$$

where $+$ here means concatenation of feature vectors. The parameter c controls the weighting between the quadratic and constant terms.

In general, ϕ maps features in a d-dimensional space to a $\binom{d+p}{p}$-dimensional space. κ computes dot products of elements of this space in $O(n)$ time.

The *linear kernel* is just the polynomial kernel with $p = 1$.

An important property is *modularity*: If you have the dual form of an algorithm, you can pass in the Gram (kernel) matrix, of all dot products of all points in the feature space, and the algorithm operates on it unchanged. So it's easy to experiment with different kernels.

2.1 When is κ a kernel?

A matrix K is *positive semi-definite* if all its eigenvalues are non-negative. A more useful (equivalent) definition is that K is PSD iff for any z, $z^TKz \geqslant 0$.

Result: Kernel matrices are positive semi-definite and symmetric.

Theorem: κ is a kernel if and only if it is a symmetric function such that all kernel matrices generated on a finite subset of its domain are positive semi-definite. (Mercer)

Given a kernel function, there is a systematic way to construct the corresponding feature space.

But it's kind of hairy and we won't go into it.

You can think of κ as a measure of similarity of x and z: it captures the correlation among its features.

This positive-semidefinite property applies to kernels on all spaces, not just $\mathbb{R}^d$.

Designing a kernel function is a way to put our knowledge about a domain into the representation of a problem.

2.2 Creating new kernels

If κ_1 and κ_2 are kernels over $\mathbb{R}^d \times \mathbb{R}^d$, f is a real-valued function on $\mathbb{R}^d$, $\phi : \mathbb{R}^d \to \mathbb{R}^D$, κ_3 is a kernel over $\mathbb{R}^D \times \mathbb{R}^D$, $\mathbf{B}$ is a symmetric positive-definite $d \times d$ matrix, q is a polynomial with non-negative coefficients, $a > 0$ is a constant, then the following functions are kernels:

- $\kappa(x, z) = \kappa_1(x, z) + \kappa_2(x, z)$
- $\kappa(x, z) = a\kappa_1(x, z)$
- $\kappa(x, z) = \kappa_1(x, z)\kappa_2(x, z)$
- $\kappa(x, z) = f(x)f(z)$
- $\kappa(x, z) = \kappa_3(\phi(x), \phi(z))$
- $\kappa(x, z) = q(\kappa_1(x, z))$
- $\kappa(x, z) = \exp(\kappa_1(x, z))$
- $\kappa(x, z) = x^T\mathbf{B}z$

2.3 Gaussian kernel

Also called the *radial basis function* (RBF) kernel.

Here is a simple and important kernel:

$$\kappa(x, z) = \exp(-\beta\|x - z\|^2) \quad .$$

for $\beta > 0$ (called the *bandwidth*).

Be careful: some software packages ask you to supply $1/\beta$.

In an SVM, any data-set is separable using this kernel, for any value of β (but it's easier if it's small).

Why is this a kernel? Let $f(x) = \exp(-\beta x \cdot x)$:

$$\begin{aligned}
\kappa(x, z) &= \exp(-\beta\|x - z\|^2) \\
&= \exp(-\beta(x - z) \cdot (x - z)) \\
&= \exp(-\beta(x \cdot x + z \cdot z - 2x \cdot z)) \\
&= \exp(-\beta x \cdot x)\exp(-\beta z \cdot z)\exp(2\beta x \cdot z) \\
&= f(x)\exp(2\beta x \cdot z)f(z) \\
&= f(x)[1 + 2\beta(x \cdot z) + \beta^2(x \cdot z)^2 + \frac{1}{3}\beta^3(x \cdot z)^3 + \ldots]f(z)
\end{aligned}$$

What $\phi(x)$ corresponds to the RBF kernel? It is an *infinite* feature space, index by $w \in \mathbb{R}^d$.

The corresponding feature "vector" is actually a function:

$$\phi(x)(w) = c(\beta, d)\mathcal{N}(w; x, 1/\beta) \quad .$$

The $c(\beta, d)$ factor is a constant. The second part can be thought of as specifying how likely it is that w would have been drawn from a Gaussian with mean x and variance $1/\beta$.

A dot-product of two functions is an integral over the index space:

$$\kappa(x, z) = \int_w \phi(x)(w)\phi(z)(w)dw \quad .$$

When β is large, the boundary is more jagged, points have more local influence. When β is small, boundary is smoother, more points are support vectors in SVM. Tune β using cross validation.

We can use a different notion other than euclidean distance between the points. So, if κ_1 is a valid kernel, then we can make a Gaussian that uses κ_1 as the distance:

$$\kappa(x, z) = \exp\left(-\beta(\kappa_1(x, x) + \kappa_1(z, z) - 2\kappa_1(xz))\right) \quad .$$

2.4 Symbolic kernels

What is amazingly powerful is that our original space doesn't have to be $\mathbb{R}^d$: it can be, for example, a space of sets or strings or trees.

2.4.1 All subsets

If our original space is a space of *sets* of objects, and we are interested in knowing whether the same subsets of elements occur together in each set, then we could imagine making feature vectors where each feature corresponds to the presence of a particular subset within the set. So, define $x_a = I(a \in x)$ for all $a \in \mathcal{A}$ where $\mathcal{A}$ is the domain of the sets; that is, there is a feature in the original space for each element in $\mathcal{A}$ indicating whether it is in set x. Now define subset features:

$$\phi_A(x) = \prod_{a \in A} I(a \in x) = \prod_{a \in A} x_a$$

for every $A \subset \mathcal{A}$. The corresponding kernel is:

$$\begin{aligned} \kappa(x, z) &= \sum_{A \subset \mathcal{A}} \phi_A(x)\phi_A(z) \\ &= \sum_{A \subset \mathcal{A}} \prod_{a \in A} x_a z_a \\ &= \prod_{a \in \mathcal{A}} (1 + x_a z_a) \end{aligned}$$

2.4.2 String kernels

How to compare two strings, especially if they are of different lengths? This might be important in a language or computational biology application.

Following "The spectrum kernel: A string kernel for SVM protein classification," Christina Leslie, Eleazar Eskin, William Stafford Noble, *Pacific Symposium on Biocomputing*, 2002.

Let $\phi_k(x)$ be a vector with a feature ϕ_a for each possible substring a of length k, where

$$\phi_a(x) = \text{number of times } a \text{ occurs in } x \quad .$$

So $\phi(x)$ is a really big feature vector in general. The associated kernel is called the *k-spectrum* kernel:

$$K_k(x, z) = \phi_k(x) \cdot \phi_k(z) \quad .$$

It can be computed very efficiently by building suffix trees. But here's a reasonably cheap method, that depends on the sparseness:

- Make a sorted list of all length k sub-strings of x (there will be $|x| - k + 1$ of them)
- Make a sorted list of all length k sub-strings of z (there will be $|z| - k + 1$ of them)
- Walk down both sorted lists accumulating the $\phi_a(x)\phi_a(z)$ terms for the sub-strings a that occur in both x and z.

This method takes $O(n \log n)$ in the length of the longer sequence.

Can be extended to handle, efficiently, all sub-strings of any length, or all non-contiguous sub-sequences using dynamic programming methods, or building more complicated data structures.

3 Kernelizing learning algorithms

3.1 Perceptron

We can easily make a dual form of the perceptron:

- $\alpha_i = 0$ for $i \in 1, \dots, n$
- Repeat until no sample is misclassified:
  - For $i = 1, \dots, n$:
    if
    $$y^{(i)} \sum_{j=1}^{n} \alpha_j y^{(i)} K(x^{(i)}, x^{(j)}) \leqslant 0 \quad :$$
    then
    $$\alpha_i := \alpha_i + 1$$

3.2 Ridge Regression

Objective is to find weights that minimize penalized squared error (maximize penalized likelihood):

Assume that the data are centered so we don't have to deal with w_0 specially.

$$J(w) = \frac{1}{2} \sum_{i=1}^{n} \left(w^T x^{(i)} - y^{(i)} \right)^2 + \frac{\lambda}{2} w^T w \quad .$$

Start with normal equations (derivative of objective set to 0):

$$X^T X w + \lambda w = X^T y$$

Ridge regression:

$$w = (X^T X + \lambda \mathbf{I})^{-1} X^T y \quad .$$

Prediction:

$$\begin{aligned} h(x) &= w \cdot x \\ &= ((X^T X + \lambda \mathbf{I})^{-1} X^T y)^T x \\ &= y^T X (X^T X + \lambda \mathbf{I})^{-1} x \end{aligned}$$

Dual form: rewrite normal equation

$$\begin{aligned} X^T X w + \lambda w &= X^T y \\ \lambda w &= X^T (y - Xw) \\ w &= \lambda^{-1} X^T (y - Xw) \\ w &= X^T \alpha \end{aligned}$$

where

$$\alpha_i = \lambda^{-1} (y^{(i)} - x^{(i)} \cdot w) \quad .$$

So, w can be written as a linear combination of training examples:

Interesing! α_i looks like an error...reminds us of perceptrons.

$$w = \sum_{i=1}^{n} \alpha_i x^{(i)} \quad .$$

Let's play with the form a bit, substituting $w = X^T \alpha$ into the objective function:

$$\begin{aligned} J(w) &= \frac{1}{2}(Y - Xw)^T(Y - Xw) + \frac{\lambda}{2} w^T w \\ J(\alpha) &= \frac{1}{2}(Y - X(X^T\alpha))^T(Y - X(X^T\alpha)) + \frac{\lambda}{2}(X^T\alpha)^T(X^T\alpha) \\ &= \frac{1}{2}Y^TY + \frac{1}{2}\alpha^T XX^T XX^T \alpha - \alpha^T XX^T Y + \frac{\lambda}{2}\alpha^T XX^T \alpha \end{aligned}$$

Let $K = XX^T$ be the Gram (Kernel) matrix, $K_{ij} = K(x^{(i)}, x^{(j)})$. Then

$$J(\alpha) = \frac{1}{2}Y^TY + \frac{1}{2}\alpha^T KK\alpha - \alpha^T KY + \frac{\lambda}{2}\alpha^T K\alpha \quad .$$

Take the gradient with respect to α, set to 0, and solve.

$$\begin{aligned} \nabla J(\alpha) &= KK\alpha - KY + \lambda K\alpha \\ 0 &= K\alpha - Y + \lambda\alpha \\ \alpha(K + \lambda I) &= Y \\ \alpha &= (K + \lambda I)^{-1} Y \end{aligned}$$

Note that once we have the αs, we can make a prediction for a new x as:

$$\begin{aligned} y(x) &= w^T X \\ &= \alpha^T X x \\ &= k(x)^T \alpha \end{aligned}$$

where $k(x)$ is a vector with elements $k(x)_i = k(x^{(i)}, x)$.

Yay! We can find the α values only using the kernel matrix. Of course, solving this requires time $O(n^3)$ where solving the primal version requires time $O(D^3)$, so whether doing this in the dual is a good idea depends on the relative sizes of the spaces.

4 More generally

If we look at the SVM, it is seeking w that minimize

$$J(w) = \sum_{i=1}^{n} [1 - y^{(i)} h(x^{(i)})]_+ + \lambda \|w\|^2 \quad ,$$

where $h(x) = x \cdot w + w_0$ and $[z]_+ = z$ if $z > 0$ and 0 otherwise.

In ridge regression, we are minimizing

$$J(w) = \sum_{i=1}^{n} (y^{(i)} - h(x^{(i)}))^2 + \lambda \|w\|^2 \quad .$$

In general, we can describe a loss function $L(a, h)$ and write the criterion as

This is not exactly the same as the prediction loss functions we have looked at before, because we are considering not just the actual prediction made by the SVM, for example, but the h value before it is thresholded.

$$J(w) = \sum_{i=1}^{n} L(y^{(i)}, h(x^{(i)})) + \lambda \|w\|^2 \quad .$$

4.1 Representer theorem

If we seek to minimize

$$J(h) = \sum_{i=1}^{n} L(y^{(i)}, h(x^{(i)})) + \lambda \|h\|^2 \ ,$$

the solutions have the form

We are taking the norm of a function here...you can think of it as a kind of penalty on the roughness of the function.

$$h^*(x) = w_0 + \sum_{i=1}^{n} \hat{\alpha}_i K(x, x^{(i)}) \ ,$$

for some vector $\hat{\alpha} \in \mathbb{R}^n$. The equivalent finite-dimensional criterion is:

$$\min_{\alpha} L(Y, \mathbf{K}\hat{\alpha}) + \lambda \hat{\alpha}^T \mathbf{K} \hat{\alpha} \ ,$$

4.2 Loss functions

A comparison of the loss functions of logistic regression, SVMs, and squared error for classification. See figure 1. Black is 0-1 loss, blue is hinge loss (SVM), green is squared error, and red is a scaled version of the logistc regression error function.

For logistic regression we have

$$L(y, h(x)) = \log(1 + \exp(-yh(x))) \ .$$

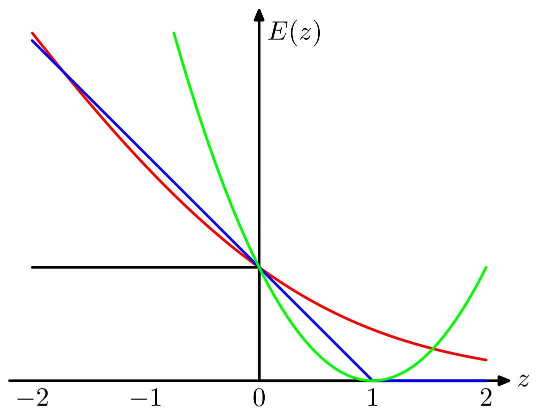

Figure 1: Comparison of loss functions (Bishop Figure 7.5).

5 Sparsity

SVMs aren't the only way to get a sparse solution.

5.1 Import vector machine

Kernel logistic regression with greedy algorithm to pick a subset of the $x^{(i)}$ to use in the representation of α. Goal is to find a small subset of vectors that approximates the nominal solution with error less than ϵ.

Zhu and Hastie, 2001, "Kernel logistic regression and the import vector machine."

5.2 Relevance vector machine

RVMs are Bayesian methods that use a prior that tends to force α_i to go to zero. They have the advantage that you don't need to use cross-validation to pick C, and they generate a probabilistically interpretable output value. However the optimization problem is non-convex.

ARD prior has a concentration parameter α_j for each weight.

$$\Pr(w \mid \alpha) = \frac{1}{2\pi}^{d/2} \left(\prod_{j=1}^{d} \alpha_j^{1/2} \right) \exp\left(-\frac{1}{2} \sum_{j=1}^{d} \alpha_j w_j^2 \right) \quad .$$

Put a Gamma distribution on α_j. Then $\Pr(w_j)$ is a Student's T distribution: fatter tails than Gaussian. Equivalent penalty function is $\sum_j \log|w_j|$. Encourages sparsity just as the Lasso does. Tends to be sparser than SVM.

"Sparse Bayesian learning and the relevance vector machine," Michael Tipping, JMLR, 2001.