The Kernel Trick

Ryan P. Adams
COS 324 – Elements of Machine Learning
Princeton University

At first glance, the support vector machine looks like a lot of work for gains that are not totally obvious. The idea of maximizing the margin makes intuitive sense, but why do all the work setting up the dual optimization problem? Just to remind you, previously we started with the geometry of the margin for linearly separable data and eventually munged it into the following quadratic program:

$$\boldsymbol{w}^\star, b^\star = \arg\min_{\boldsymbol{w}, b} \left\{ \frac{1}{2} ||\boldsymbol{w}||_2^2 \right\} \quad \text{s.t.} \quad y_n(\boldsymbol{w}^\mathsf{T}\Phi(\boldsymbol{x}_n) + b) \geq 1 \; \forall n \in 1, \ldots, N \,. \tag{1}$$

Then we manipulated things via the Lagrangian and the KKT conditions into the dual:

$$\boldsymbol{\alpha}^\star = \arg\max_{\boldsymbol{\alpha}} \left\{ \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \sum_{n'=1}^N y_n y_{n'} \alpha_n \alpha_{n'} \Phi(\boldsymbol{x}_n)^\mathsf{T} \Phi(\boldsymbol{x}_{n'}) \right\} \tag{2}$$

$$\text{s.t.} \quad \sum_{n=1}^N y_n \alpha_n = 0 \quad \text{and} \quad \alpha_n \geq 0 \; \forall n \in 1, \ldots, N \,.$$

You can also arrive at this problem by reasoning about the distance between the convex hulls of the two classes. In any case, the primal is a problem with $J + 1$ variables to optimize (assuming a J-dimensional feature space), and the dual is a problem with N variables to optimize. The dual reveals why this construction is called a *support vector* machine: α_n will be zero for any training datum that is not right on the margin and "supporting" it. After solving the dual quadratic program, we can go back to the optimal weight vector via:

$$\boldsymbol{w}^\star = \sum_{n=1}^N y_n \alpha_n^\star \Phi(\boldsymbol{x}_n) \,. \tag{3}$$

This allows us to substitute the weights back into the classification function and make new predictions for previously-unseen inputs $\boldsymbol{x}$:

$$y = \text{sgn}\{(\boldsymbol{w}^\star)^\mathsf{T}\Phi(\boldsymbol{x}) + b^\star)\} \tag{4}$$

$$= \text{sgn}\left\{ b^\star + \sum_{n=1}^N y_n \alpha_n^\star \Phi(\boldsymbol{x}_n)^\mathsf{T} \Phi(\boldsymbol{x}) \right\} \,. \tag{5}$$

A very interesting thing to observe about Equations 2 and 5 is that they only depend on the data via inner products in feature space. If we had a way to perform this inner product without the computational cost depending directly on J (the dimensionality of the feature space) then we could make the feature space as large as we wanted—include many, many basis functions—but the cost of both training and prediction would only be a function of the size N of our finite data set.

This is precisely what *kernel functions* do. A kernel function $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a function such that there exists *some* vector function $\Phi : \mathcal{X} \to \mathbb{R}^J$ such that $K(\boldsymbol{x}, \boldsymbol{x}') = \Phi(\boldsymbol{x})^\mathsf{T}\Phi(\boldsymbol{x}')$. Interestingly, there are kernels for which J is infinity.

One of the things we can do with a kernel function is form a *Gram matrix*. This is the matrix arising from applying the kernel to all pairs in a set of N points in $\mathcal{X}$. If we denote that set as $\{\boldsymbol{x}_n\}_{n=1}^N$, then the Gram matrix is:

$$\boldsymbol{K} = \boldsymbol{\Phi}\boldsymbol{\Phi}^\mathsf{T} = \begin{bmatrix} K(\boldsymbol{x}_1, \boldsymbol{x}_1) & K(\boldsymbol{x}_1, \boldsymbol{x}_2) & \cdots & K(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ K(\boldsymbol{x}_2, \boldsymbol{x}_1) & K(\boldsymbol{x}_2, \boldsymbol{x}_2) & \cdots & K(\boldsymbol{x}_2, \boldsymbol{x}_N) \\ \vdots & \vdots & \ddots & \vdots \\ K(\boldsymbol{x}_N, \boldsymbol{x}_1) & K(\boldsymbol{x}_N, \boldsymbol{x}_2) & \cdots & K(\boldsymbol{x}_N, \boldsymbol{x}_N) \end{bmatrix}. \tag{6}$$

This matrix is symmetric and will always be positive semi-definite, i.e.,

$$\boldsymbol{z}^\mathsf{T}\boldsymbol{K}\boldsymbol{z} \geq 0 \, \forall \boldsymbol{z} \in \mathbb{R}^N. \tag{7}$$

This property that the Gram matrix is always positive semi-definite for all finite sets of data in $\mathcal{X}$ is equivalent to saying that the *function* $K(\cdot, \cdot)$ is positive semi-definite.

The point of this is that if we have a kernel function for the features $\Phi(\cdot)$ that we're using, we can go in and replace our inner products with this kernel function. The dual problem becomes

$$\boldsymbol{\alpha}^\star = \arg\max_{\boldsymbol{\alpha}} \left\{ \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n=1}^N \sum_{n'=1}^N y_n y_{n'} \alpha_n \alpha_{n'} K(\boldsymbol{x}_n, \boldsymbol{x}_{n'}) \right\} \tag{8}$$

$$\text{s.t.} \quad \sum_{n=1}^N y_n \alpha_n = 0 \quad \text{and} \quad \alpha_n \geq 0 \, \forall n \in 1, \ldots, N.$$

and with optimal $\boldsymbol{\alpha}^\star$, the prediction problem becomes

$$y = \text{sgn}\left\{ b^\star + \sum_{n=1}^N y_n \alpha_n^\star K(\boldsymbol{x}_n, \boldsymbol{x}) \right\}. \tag{9}$$

Suddenly we can train and make predictions with the SVM and never have to pay any costs that directly depend on J.

A Simple Example

What are some interesting kernel functions? Let's assume here that $\boldsymbol{x} \in \mathbb{R}^D$ and try to understand how a kernel function defined on two points in $\mathbb{R}^D$ might lead to some interesting implied $\Phi(\cdot)$

functions. The trivial one is of course just the inner product itself $K(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^{\mathsf{T}}\boldsymbol{x}'$, but we don't get any interesting savings or insight from that. However, if we square that kernel, things take on a new flavor:

$$
\begin{aligned}
K(\boldsymbol{x}, \boldsymbol{x}') &= (\boldsymbol{x}^{\mathsf{T}}\boldsymbol{x}')^2 && (10)\\
&= \left(\sum_{d=1}^{D} x_d x'_d\right)^2 = \left(\sum_{d=1}^{D} x_d x'_d\right)\left(\sum_{d=1}^{D} x_d x'_d\right) = \sum_{d=1}^{D}\sum_{d'=1}^{D} x_d x'_d x_{d'} x'_{d'} && (11)\\
&= \sum_{d=1}^{D}\sum_{d'=1}^{D} (x_d x_{d'})(x'_d x'_{d'})\,. && (12)
\end{aligned}
$$

If we define $\Phi(\boldsymbol{x})$ to be the basis of products of all pairs of dimensions of $\boldsymbol{x}$:

$$
\Phi(\boldsymbol{x}) = \begin{bmatrix} x_1 x_1 \\ x_1 x_2 \\ \vdots \\ x_1 x_d \\ x_2 x_1 \\ \vdots \\ x_D x_D \end{bmatrix} \tag{13}
$$

then we can see that this kernel is an inner product in this feature representation. Even in this simple example we're already getting some computational mileage. The dimension J of this feature representation is D^2 but by using the kernel we can compute it in $\mathcal{O}(D)$ time.

We can modify this kernel slightly with an additional constant $c > 0$ via:

$$
\begin{aligned}
K(\boldsymbol{x}, \boldsymbol{x}') &= (\boldsymbol{x}^{\mathsf{T}}\boldsymbol{x}' + c)^2 && (14)\\
&= \left(\sum_{d=1}^{D} x_d x'_d + c\right)^2 = \left(\sum_{d=1}^{D} x_d x'_d + c\right)\left(\sum_{d=1}^{D} x_d x'_d + c\right) && (15)\\
&= \sum_{d=1}^{D}\sum_{d'=1}^{D} (x_d x'_d + c)(x_{d'} x'_{d'} + c) && (16)\\
&= \sum_{d=1}^{D}\sum_{d'=1}^{D} x_d x'_d x_{d'} x'_{d'} + c x_{d'} x'_{d'} + c x_d x'_d + c^2 && (17)\\
&= c^2 + \sum_{d=1}^{D}\sum_{d'=1}^{D} (x_d x_{d'})(x'_d x'_{d'}) + 2c \sum_{d=1}^{D} (\sqrt{2c}\, x_d)(\sqrt{2c}\, x'_d)\,. && (18)
\end{aligned}
$$

This corresponds to an inner product where the feature space now includes first order terms and a

constant term:

$$\Phi(\boldsymbol{x}) = \begin{bmatrix} x_1 x_1 \\ x_1 x_2 \\ \vdots \\ x_1 x_d \\ x_2 x_1 \\ \vdots \\ x_D x_D \\ \sqrt{2c} x_1 \\ \vdots \\ \sqrt{2c} x_D \\ c \end{bmatrix} . \tag{19}$$

More generally, we can get a feature space with all terms up to degree R via the kernel:

$$K(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^{\mathsf{T}} \boldsymbol{x}' + c)^R . \tag{20}$$

This is exciting because the number of terms grows as $O(D^R)$ and so the cost of computing things with inner products can be exponentially bad, but this always has $O(D)$ cost via the kernel.

Rules for Composing Kernels

There are several ways to combine together kernel functions to make new valid (positive semi-definite) kernel functions. Assume we have two valid kernels $K_1(\boldsymbol{x}, \boldsymbol{x}')$ and $K_2(\boldsymbol{x}, \boldsymbol{x}')$. We can generate new a new kernel using these via:

$$K(\boldsymbol{x}, \boldsymbol{x}') = K_1(\boldsymbol{x}, \boldsymbol{x}') + K_2(\boldsymbol{x}, \boldsymbol{x}') \qquad \text{(addition)} \tag{21}$$

$$K(\boldsymbol{x}, \boldsymbol{x}') = K_1(\boldsymbol{x}, \boldsymbol{x}')\, K_2(\boldsymbol{x}, \boldsymbol{x}') \qquad \text{(multiplication)} \tag{22}$$

$$K(\boldsymbol{x}, \boldsymbol{x}') = c\, K_1(\boldsymbol{x}, \boldsymbol{x}') \quad \text{for } c > 0 \qquad \text{(scaling)} \tag{23}$$

$$K(\boldsymbol{x}, \boldsymbol{x}') = f(\boldsymbol{x}) K_1(\boldsymbol{x}, \boldsymbol{x}') f(\boldsymbol{x}') \quad \text{for any } f(\boldsymbol{x}) \qquad \text{(outer product)} \tag{24}$$

These properties let us define a variety of new kind of kernels. In particular, we can come up with rich transformations via tools like power series. Let $g(z)$ be an analytic function with power series

$$g(z) = \sum_{k=0}^{\infty} a_k z^k \quad \text{where } a_k \geq 0 . \tag{25}$$

Assuming this series converges on the set of interest, $g(K(\boldsymbol{x}, \boldsymbol{x}'))$ is a valid kernel by successive application of the rules above. One particularly important example is where $g(z) = e^z$ and the $a_k = \frac{1}{k!}$. Note that this is a sum of kernels of the form shown earlier that generate polynomials, but now it has polynomials of all positive degrees. That is, this effectively takes J to infinity.

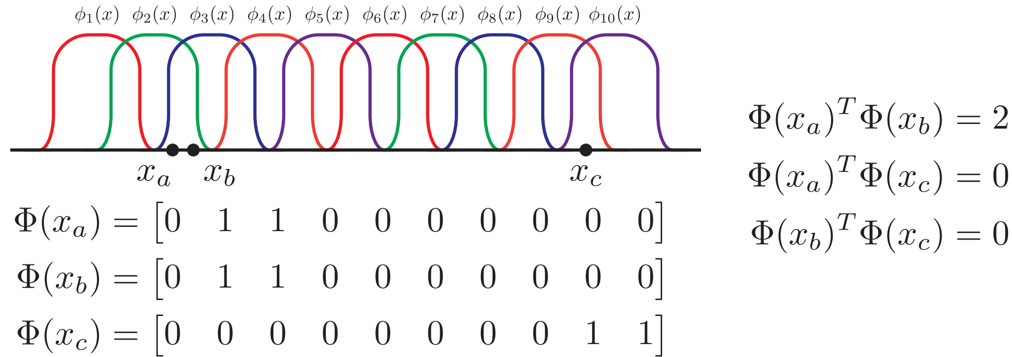

Figure 1: One way to think about how you could turn distance into similarity via an inner product. Here there are ten basis functions that are local to a specific region. The inner product essentially reflects how many of these they have in common.

Kernels and Similarity

The rules above can be used to see how it might be possible to arrive at a popular kernel like

$$K(\boldsymbol{x}, \boldsymbol{x}') = \exp\left\{-\frac{1}{2\ell^2}||\boldsymbol{x} - \boldsymbol{x}'||_2^2\right\}, \tag{26}$$

which has no obvious relationship to an inner product. This kernel is essentially the familiar Gaussian shape, falling off toward zero rapidly as a function of the L^2 distance between the two points. This is one of many example of kernels that depend primarily on the distance between the two input points — often called *stationary* kernels. Most of these have the property that they are big when the two points are similar and decrease monotonically as the distance becomes larger. This is a common theme in thinking about why kernels can be interesting: beyond just inner products, they can give us a way to build implicit feature spaces based on ideas about similarity between data. It is often easier to engineer a machine learning system based on intuition about similarity between data than it is to build it around clever choices of basis function.

You might reasonably wonder how inner products in feature space can result in a direct similarity measure. We can build intuition for this in two ways. First, imagine that the data we wish to model live on a (hyper-) sphere, but are represented in their Cartesian coordinates in space. For example, our data could live in $\mathbb{R}^2$ but be constrained to a circle. Then if we wish to compute the squared L^2 distance between two of these points $\boldsymbol{x}$ and $\boldsymbol{x}'$ we write:

$$||\boldsymbol{x} - \boldsymbol{x}'||_2^2 = (\boldsymbol{x} - \boldsymbol{x}')^{\mathsf{T}}(\boldsymbol{x} - \boldsymbol{x}') = \boldsymbol{x}^{\mathsf{T}}\boldsymbol{x}' - 2\boldsymbol{x}^{\mathsf{T}}\boldsymbol{x}' + (\boldsymbol{x}')^{\mathsf{T}}\boldsymbol{x}'\,. \tag{27}$$

Because the points are constrained to a sphere, all their norms are the same and the terms $\boldsymbol{x}^{\mathsf{T}}\boldsymbol{x} = (\boldsymbol{x}')^{\mathsf{T}}\boldsymbol{x}' = r^2$ where r is the radius of the sphere. The term $\boldsymbol{x}^{\mathsf{T}}\boldsymbol{x}'$ has its maximum of the same value r^2 when the two vectors project directly onto each other. So the overall squared distance can be seen to be

$$||\boldsymbol{x} - \boldsymbol{x}'||_2^2 = 2(r^2 - \boldsymbol{x}^{\mathsf{T}}\boldsymbol{x}')\,. \tag{28}$$

Thus the distance is completely determined by the inner product between these two vectors, i.e., larger distance corresponds to smaller "similarity" as measured by the inner product.

A second way to gain intuition for this is to imagine a basis of "top hats" as shown in Figure 1. Each of the top hats corresponds to one of the entries of the vector function $\Phi(x)$. If we imagine putting a value $x_a \in \mathbb{R}$ into this basis, it would have values close to zero in all of the entries of $\Phi(x_a)$ except for the few that were sitting right on top of x_a. Another value x_b in the same representation would have an almost identical feature representation, with zeros everywhere except for the top hats right on top of it. The more top hats x_a and x_b share, the larger their inner product will be. On the other hand, if we introduce a point x_c some distance away, then it will have non-zero entries in $\Phi(x_c)$ in a disjoint set of dimensions. Thus its inner product with both $\Phi(x_a)$ and $\Phi(x_b)$ will be close to zero. The inner product in feature space gives a direct notion of distance-based similarity.

Changelog

- 17 October 2018 – Initial version.