CSE 252C: Computer Vision III

Lecturer: Serge Belongie
Scribe: Catherine Wah

LECTURE 15
Kernel Machines

15.1. Kernels

We will study two methods based on a special kind of function $k(\boldsymbol{x}, \boldsymbol{y})$ called a *kernel*:

- Kernel PCA (to help build our intuition) and
- Support Vector Machines, which are a powerful discriminative learning method.

What is a kernel? In this context, it is a function that satisfies the following condition:

$$\sum_{i,j} k(\boldsymbol{x}^i, \boldsymbol{x}^j) c_i c_j \geq 0, \tag{15.1}$$

for some set of feature vectors $\boldsymbol{x}$ and any vector $\boldsymbol{c}$. We say the kernel is positive semi-definite, or that it satisfies Mercer's condition.

[1]Department of Computer Science and Engineering, University of California, San Diego.

December 31, 2009

15.2. Kernel Trick

What's interesting about this, and what forms the basis of the so-called "kernel trick" (Aizerman *et al.* , 1964), is that kernels that satisfy this condition are equivalent to a simple dot product in some higher dimensional space:

$$k(\boldsymbol{x}, \boldsymbol{y}) = \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{y}), \tag{15.2}$$

where $\phi(\cdot)$ maps a vector from "input space" to "feature space" (usually higher, or infinite, dimensional). In some cases we specify $\phi(\cdot)$, in other cases, it's implicit for a choice of $k(\cdot, \cdot)$. The "kernel trick" refers to the idea of swapping in $k(\boldsymbol{x}, \boldsymbol{y})$ for $\boldsymbol{x} \cdot \boldsymbol{y}$ in algorithms that call for a dot product between two vectors, for example, the perceptron, SVMs (supervised), PCA (unsupervised).

In some cases, it may not be obvious that the algorithms, at their heart, come down to dot products, but these and several others have been formulated in such a way to exhibit this characteristic. So what does this buy us? In the supervised case, classes that aren't linearly separable in input space might be linearly separable in feature space (Figure 1). Consider the

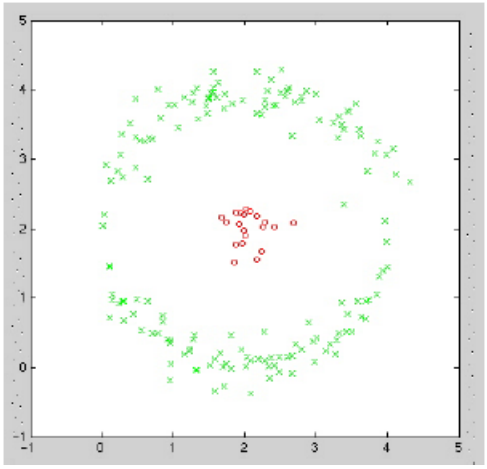

Figure 1. The clump and annulus problem. These classes are linearly inseparable in the input space. Note: a quadratic surface would do the trick here, but linear algorithms are simpler and better understood.

mapping $\Phi : (x_1, x_2)^\mathsf{T} \mapsto (x_1, x_2, x_1^2 + x_2^2)^\mathsf{T}$, where $\boldsymbol{x} \in \mathbb{R}^2$ and $\phi(\boldsymbol{x}) \in \mathbb{R}^3$. In this way, by adding a new dimension, it becomes possible to separate the clump from the annulus.

This hints to us that high dimensionality can be good (greater classification power)! Your instinct might be telling you that high dimensionality can also get you into trouble, conceptually or computationally; this instinct is correct.

This idea is the subject of Vapnik-Chervonenkis (VC) theory, a critical component of statistical learning theory, which addresses questions such as how best to perform classification given nothing but labelled example data (with no prior knowledge). We won't study VC theory in this class, but it provides us with answers to questions such as "under what conditions are high dimensional representations good?"

Supposing we restrict ourselves to cases in which the high dimensional mapping is good, there is still the problem of computation time. For example, a simple polynomial mapping on 28×28 MNIST digits can easily explode the dimensionality into the billions. This is where the kernel trick comes in. Except for these toy examples, in practice, one never makes explicit use of $\Phi(\cdot)$, or even necessarily needs to know what it is. We only need to know that it exists, which Mercer's condition answers for us.

This was foreshadowed in the second homework, where we considered the Mahalanobis distance between two vectors:

$$(\boldsymbol{x}^i - \boldsymbol{x}^j)^\mathsf{T}\Sigma^{-1}(\boldsymbol{x}^i - \boldsymbol{x}^j) = (\boldsymbol{y}^i - \boldsymbol{y}^j)^\mathsf{T}(\boldsymbol{y}^i - \boldsymbol{y}^j) = \|\boldsymbol{y}^i - \boldsymbol{y}^j\|^2 \tag{15.3}$$

where, if $\sum^{-1}$ is positive semi-definite, we can write $\sum^{-1} = QQ^\mathsf{T}$ and set $\boldsymbol{y} = Q^\mathsf{T}\boldsymbol{x}$. In this case, Q represents a simple case of a feature mapping; the Mahalanobis distance is equivalent to a dot product in a different space.

15.3. Kernel PCA

Now let's look at an application of the kernel trick: Kernel PCA (Schölkopf *et al.* , 1998). This is an unsupervised example, used for dimensionality reduction when the data doesn't live on a linear manifold. We use the kernel trick to perform "classic" PCA in high dimensional space, then map it back to input space, where the "axes" will appear curved (Figure 2).

Before we "kernelize" PCA, let's review how classic PCA works (assuming centered data: $\sum_{i=1}^N \boldsymbol{x}^i = \boldsymbol{0}$). PCA finds the eigenvectors of the covariance matrix C:

$$C = \frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{x}^i)(\boldsymbol{x}^i)^\mathsf{T}. \tag{15.4}$$

By construction, C is positive semidefinite. Eigenvectors $\lambda\boldsymbol{v} = C\boldsymbol{v}$ are used to capture principal axes of variation.

We can use C in the following way:

(a) Find the eigenvectors and eigenvalues, sorted in decreasing order by eigenvalue, possibly truncating
(b) Project test data onto eigenvectors
(c) Use the projections for classification, denoising, compression, etc.

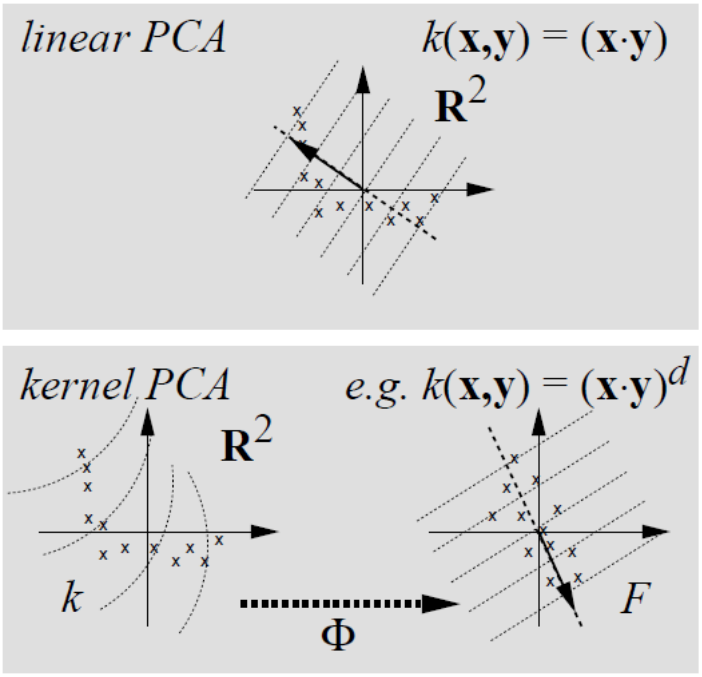

Figure 2. The basic idea of kernel PCA with 2D inputs. In some high-dimensional feature space F (bottom right), we are performing linear PCA, just like a PCA in input space (top). Since F is nonlinearly related to the input space (via Φ), the contour lines of constant projections onto the principal eigenvector become nonlinear in input space (Schölkopf *et al.* , 1998).

We'd like to be able to do this for nonlinear data, therefore, the solution is to kernelize. In order to do so, we start by expressing PCA in terms of dot products:

$$C\boldsymbol{v} = \frac{1}{N}\sum_i \boldsymbol{x}^i \boldsymbol{x}^{i\mathsf{T}} \boldsymbol{v} = \lambda \boldsymbol{v}, \tag{15.5}$$

therefore

$$\boldsymbol{v} = \frac{1}{N\lambda}\sum_j \boldsymbol{x}^j \boldsymbol{x}^{j\mathsf{T}} \boldsymbol{v} = \frac{1}{N\lambda}\sum_j (\boldsymbol{x}^j \cdot \boldsymbol{v})\boldsymbol{x}^j. \tag{15.6}$$

(Note here that $(\boldsymbol{x}^j \cdot \boldsymbol{v})$ is just a scalar.) By using dot products, all solutions $\boldsymbol{v}$ with $\lambda \neq 0$ lie in the span of $\{\boldsymbol{x}^i\}$, or $v = \sum_i \alpha_i \boldsymbol{x}^i$, where the α_i's are the dot products.

Now suppose we pick some mapping $\phi(\cdot)$. Assume we center the data (a homework problem), and as before, we write

$$C = \frac{1}{N}\sum_i \phi(\boldsymbol{x}^i)\phi(\boldsymbol{x}^i)^\mathsf{T}, \tag{15.7}$$

which is possibly high dimensional, and we diagonalize it as $\lambda\boldsymbol{v} = C\boldsymbol{v}$. Like before, $\boldsymbol{v}$ lies in the span of $\phi(\boldsymbol{x}^i)$'s.

Now we dot both sides by $\phi(\boldsymbol{x}^j)$:

$$\lambda(\phi(\boldsymbol{x}^j)\cdot\boldsymbol{v}) = \phi(\boldsymbol{x}^j)\cdot C\boldsymbol{v},\ \forall j \tag{15.8}$$

and note that $\boldsymbol{v} = \sum_i \alpha_i\phi(\boldsymbol{x}^i)$. The motivation for this is that we want to avoid explicit use of $\phi(\cdot)$; we only want to use it via $k(\cdot,\cdot)$.

Therefore,

$$\lambda\sum_i \alpha_i\left(\phi(\boldsymbol{x}^j)\cdot\phi(\boldsymbol{x}^i)\right) = \frac{1}{N}\sum_i \alpha_i\left(\phi(\boldsymbol{x}^j)\cdot\sum_l\phi(\boldsymbol{x}^l)\right)\left(\phi(\boldsymbol{x}^l)\cdot\phi(\boldsymbol{x}^i)\right),\forall k \tag{15.9}$$

or

$$N\lambda K\boldsymbol{\alpha} = K^2\boldsymbol{\alpha} \tag{15.10}$$

where $K_{ij} = \phi(\boldsymbol{x}^i)\cdot\phi(\boldsymbol{x}^j)$ and $\boldsymbol{\alpha}$ is a vector of α_i's. Note that since K is full rank, it follows that

$$N\lambda\boldsymbol{\alpha} = K\boldsymbol{\alpha}. \tag{15.11}$$

This is our eigenvector problem, now on K instead of the covariance matrix.

So in the end, kernel PCA just requires us to diagonalize K instead of C, a different (and bigger) covariance matrix defined on vectors in a high dimensional space. In practice, the data needs to be centered first, which can be done wia a simple operation on K; see Homework 4.

How do we use these principal components? Consider some test point $\boldsymbol{x}$ (*e.g.* , any point in $\mathbb{R}^2$). To project it onto eigenvectors, we compute:

$$\boldsymbol{v}^k\cdot\phi(\boldsymbol{x}) = \sum_i \alpha_i k(\boldsymbol{x}^i,\boldsymbol{x}) \tag{15.12}$$

for the kth kernel principal component coefficient. We demonstrate this for a 2D toy example with 3 gaussian clumps, using a gaussian kernel (Figure 3). Recall that regular PCA would just give two orthogonal axes in $\mathbb{R}^2$. Surface brightness shows the value of the kth enigenvector projection; for linear PCA you would just have 2 ramps. Note the similarity to spectral clustering; NCut used $D^{-1/2}WD^{-1/2}$ instead of centering in feature space. It allows extrapolation to the full plane; this is the same thing as the Nyström extension, which is used in spectral clustering. Note in this example, each cluster is first "found," then a local x-y coordinate system is extracted for

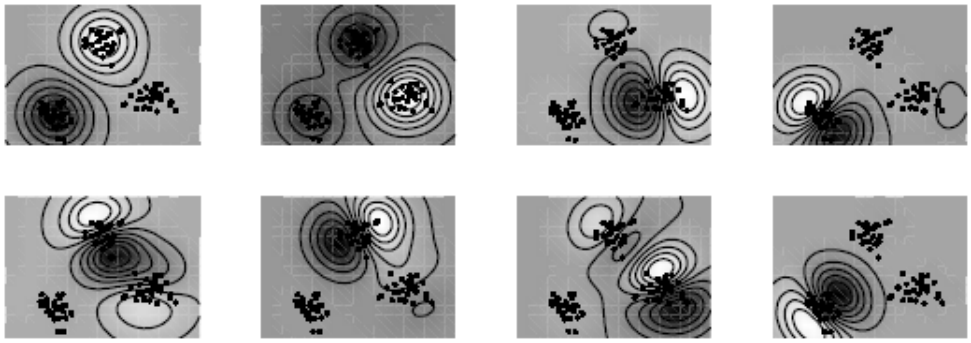

Figure 3. Toy example with three data clusters; first eight nonlinear principal components are extracted with a radial basis function (*i.e.* , Gaussian) for the kernel. Note that the first two principal components (top left) nicely separate the three clusters (Schölkopf *et al.* , 1998).

it. For denoising, we use a fixed point iteration to find the preimage; this is related to mean-shift.

15.4. Support Vector Machines

Now let's look at a discriminative example: the support vector machine (SVM). Now we have data labeled (positive and negative):

$$\{(\boldsymbol{x}^1, y^1), (\boldsymbol{x}^2, y^2), \ldots, (\boldsymbol{x}^N, y^N)\},\ y \in \{+1, -1\} \tag{15.13}$$

In the linear case, we want to find a separating hyperplane with maximum margin, *i.e.* , we want a hyperplane with maximal distance to the nearest data point (Figure 4). The linear classifier (decision boundary) has the form:

$$\boldsymbol{w} \cdot \boldsymbol{x} - b = 0, \tag{15.14}$$

where $\boldsymbol{w}$ is the normal vector for the hyperplane and b is the offset.

SVM learning requires the solution of a quadratic programming problem (beyond the scope of this class) that returns the data points that "prop up" the hyperplanes that define the margin, parallel to the separating hyperplane. This can be generalized to the soft-margin case, when perfect separation is not possible.

The extension to cases requiring a nonlinear separation boundary requires kernelizing; the linear separating boundary looks curved in input space. When testing what side of the decision boundary you are on, we use a kernel in place of the dot product, which we can think of as the distance (or similarity) to exemplars right on either side of the margin; *e.g.* , for face gender classification, these are highly hermaphroditic faces $\pm\epsilon$.

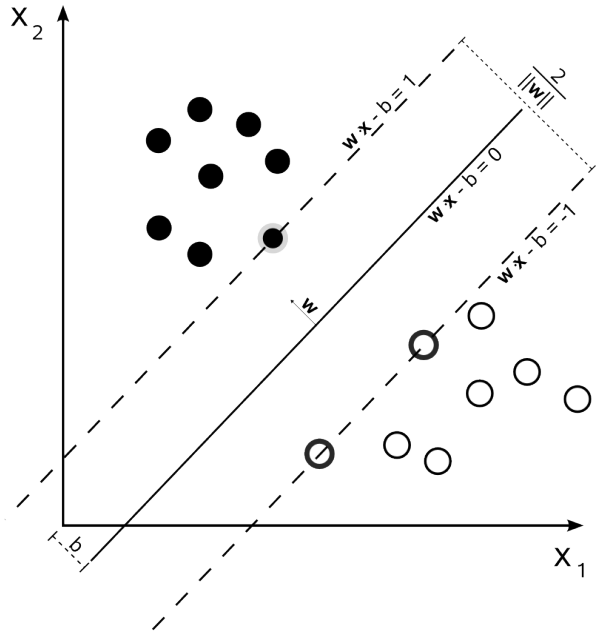

Figure 4. Maximum-margin hyperplane and margins for a SVM trained with samples from two classes. Samples on the margin are called the support vectors. (http://en.wikipedia.org/wiki/Support_vector_machine)

To summarize, SVMs are a highly general method with excellent results. The usual questions of what kernel to use, what value of σ, etc. can be addressed with experience and cross validation.