Lecture 13:

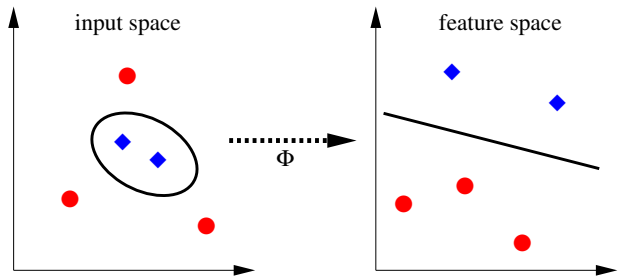
Kernel Machines

Sam Roweis

December 7, 2004

Feature Spaces

- The extended representation is called a *feature space*.
- An algorithm that is linear in the feature space may be highly nonlinear in the original space if the features contain nonlinear mappings of the raw data.
- Thus, we can think of having "promoted" our data x into a higher-dimensional feature space z using a nonlinear mapping $\phi(x)$ and then running our original algorithm in that new space.



Algorithms Based on Vector Data

- Recall our normal approach to many classification, regression, and unsupervised learning problems:
  Embed the input to the problem into a vector space (e.g. R^n) and then do some geometric, or linear algebraic operations.
- We can often make our algorithms more powerful by embedding the data into a richer (larger) space which includes some fixed, possibly nonlinear functions of the original measurements.
- For example, if we measure x_1, x_2, x_3 for each datapoint, we might use the representation $z = [1, x_1, x_2, x_3, x_1^2, x_2^2, x_3^2, x_1x_2, x_2x_3]$ in a regression or classification machine.
- We've seen this trick before: adding a bias term, quadratic regression, basis functions, generalized linear models.
- This trick has potential advantages (more power) and potential disadvantages (more computation, potential for overfitting).

How Should We Use the Feature Space

- The feature point $z = \phi(x)$ corresponding to an input point x is called the *image* of x; the input point x, if any, corresponding to a given feature vector z is called the *pre-image* of z.
- The naive way to use a feature space is to explicitly compute the image of every training point and testing point, and run our algorithm completely in feature space.
- Two potential problems:
  1. Problem: the feature space may be ultra-high dimensional or even infinite dimensional, so direct (explicit) calculations in feature space may not be practical or even possible.
  2. We may sometimes want to "bring back" an answer from feature space to input space and that involves finding pre-images which is hard and not always possible.
- We could restrict ourselves to "manageable" feature spaces, but...

The Kernel Trick

- It turns out that for some special feature spaces $z = \phi(x)$, it is possible to compute the *inner product* $z_1^\top z_2$ between images of two input points x_1, x_2 very efficiently. This is true when the components of ϕ are the eigenfunctions of a special class of positive definite functions called *Mercer Kernels*, in which case:

$$K(x_1, x_2) = z_1^\top z_2 = \phi(x_1)^\top \phi(x_2)$$

- The key idea of kernel machines is to reduce an algorithm to one which *depends only on dot products between data vectors* and then to *replace* the dot product evaluations in feature space with kernel function evaluations in the input space.
- This "kernel trick" allows us to run algorithms in (very high dimensional) feature spaces without ever going there, provided that all we do are dot products and that we only represent feature space points that are linear combinations of known input space images.

Properties of Kernels

- Kernels are *symmetric* in their arguments: $K(x_1, x_2) = K(x_2, x_1)$.
- They are positive valued for any inputs: $K(x_1, x_2) \geq 0$.
- The Cauchy-Schwartz inequality still holds: $K^2(x_1, x_2) \leq K(x_1, x_1)K(x_2, x_2)$.
- Technically, to use a function as a kernel, it must satisfy "Mercer's conditions" for a positive-definite operator.
- The intuition is easy to get for finite spaces.
  1. Discretize x space as densely as you want into buckets x_i.
  2. Between each two cells x_i, x_j, compute the kernel function, and write these values as a (symmetric) matrix $M_{ij} = K(x_i, x_j)$.
  3. If the matrix is positive definite, the kernel is OK.

When the Kernel Trick Works

- The kernel trick allows us to efficiently compute dot products in very high dimensional spaces using the kernel function, but it doesn't help us do addition, subtraction, outer products, etc.
- Not a big loss, since many interesting feature spaces are very high or infinite dimensional, so we couldn't even write down the results of such operations anyway.
- Because of this, in general, everything we represent in the high-dimensional feature space must be expressable as a *linear combination of the images of training data points*.
- Actually this turns out to be fine for many problems, e.g. the optimal weights in a perceptron classifier run in kernel space are guaranteed to be representable. This is true of many other algorithms, e.g. support vector machines. (There is more theory about this under (surprise surprise) the "Representer's Theorem". [originally by Kimeldorf and Wahaba, reproved by Schoelkopf et al]).

Examples of Kernels

- Linear: $K(\mathbf{x}_1, \mathbf{x}_2) = \mathbf{x}_1^\top \mathbf{x}_2$ $\quad (\phi(x) = x)$
- Affine: $K(\mathbf{x}_1, \mathbf{x}_2) = \mathbf{x}_1^\top \mathbf{Q} \mathbf{x}_2$ $\quad$ (Q symmetric pos.def)
- Gaussian: $K(\mathbf{x}_1, \mathbf{x}_2) = \exp[-.5 \, \|\mathbf{x}_1 - \mathbf{x}_2\|^2]$
- Polynomial: $K(\mathbf{x}_1, \mathbf{x}_2) = (1 + \mathbf{x}_1^\top \mathbf{x}_2)^k$ $\quad$ (watch scaling!)
- Sigmoid: $K(\mathbf{x}_1, \mathbf{x}_2) = \tanh(a \, \mathbf{x}_1^\top \mathbf{x}2 + b)$
- Closure rules:
  - The sum of any two kernels is a kernel.
  - The product of any two kernels is a kernel.
  - A kernel plus a constant is a kernel.
  - A scalar times a kernel is a kernel.

Geometry of Feature Space

- Dot product between two points $= K(x_i, x_j) > 0$, and so all points lie in a single orthant in feature space.
- Length of a point in feature space:
$$\|z_i\|^2 = K(x_i, x_i)$$
(so for Gaussian kernels, everybody lies on surface of unit sphere)
- Distance between two points in feature space:
$$\|z_1 - z_2\|^2 = K(x_1, x_1) + K(x_2, x_2) - 2K(x_1, x_2)$$
- Distance between a point and the mean of all others:
$$\|z_k - \bar{z}\|^2 = K(x_k, x_k) + \frac{1}{N^2}\sum_{ij} K(x_i, x_j) - 2\sum_i K(x_k, x_i)$$
- Zero mean calculations in feature space:
$$< z_i - \bar{z}, z_j - \bar{z} >= K(x_i, x_j) + \frac{1}{N^2}\sum_{k\ell} K(x_k, x_\ell) - \sum_k K(x_k, x_i) \sum_k K(x_j$$

Gram Matrix

- But first, some notation...
- The "Gram Matrix" is the N by N symmetric matrix of all pairwise kernel evaluations: $G_{ij} = K(x_i, x_j)$.
- If you successfully "kernelize" an algorithm, then your algorithm will only need to consult/compute entries of the Gram matrix as it runs, because it depends only on dot products between the data points.
- An equivalent characterization (due to Saitoh) of Mercer's conditions is that a valid kernel generates symmetric positive definite Gram matrices for any finite sample of raw data $\{x_i\}$.

"Kernelizing" an Algorithm

- The art of designing a kernel machine is to take a standard algorithm and massage it so that all references to the original data vectors x appear only in dot products $< x_i, x_j >$.
- Often you can do this and obtain an exactly equivalent algorithm to the one you started with.
Sometimes you need to make small modifications.
- Thus, a kernel machine contains two modules: the algorithm and the kernel function. Choosing the kernel function is a hard problem which we won't discuss today.
- "Kernelizing" an algorithm can actually be pretty easy.
How about a few examples...

Example: K-NN Classification

- Distance between two points in feature space:
$$\|\mathbf{z}_1 - \mathbf{z}_2\|^2 = K(\mathbf{x}_1, \mathbf{x}_1) + K(\mathbf{x}_2, \mathbf{x}_2) - 2K(\mathbf{x}_1, \mathbf{x}_2)$$

Example: K-Means Clustering

- Distance between two points in feature space:

$$\|\mathbf{z}_1 - \mathbf{z}_2\|^2 = K(\mathbf{x}_1, \mathbf{x}_1) + K(\mathbf{x}_2, \mathbf{x}_2) - 2K(\mathbf{x}_1, \mathbf{x}_2)$$

- Represent cluster centres as linear combinations of data points:

$$\mathbf{c}_k = \sum_i \alpha_{ik} \mathbf{z}_i$$

- Distance between a new point and a cluster centre in feature space:

$$\|\mathbf{z} - \mathbf{c}_k\|^2 = K(\mathbf{x}, \mathbf{x}) + \sum_{ij} \alpha_{ik}\alpha_{jk} K(\mathbf{x}_i, \mathbf{x}_j) - 2\sum_i \alpha_{ik} K(\mathbf{x}, \mathbf{x}_i)$$

- True or false?
  In regular K-Means (running directly in the input space), the cluster centres are always linear combinations of the data points?

Kernel Perceptron

- The update rule for the weight vector in the perceptron can also be rewritten in terms of dot products only.
- Old rule: if $y_i \mathbf{w}^\top \mathbf{x}_i + b \leq 0$ then $\mathbf{w} \leftarrow \mathbf{w} + y_i \mathbf{x}_i$.
- Recall our new representation: $\mathbf{w} = \sum_i (\alpha_i y_i) \mathbf{z}_i$
- Equivalent new update rule:
  if $y_i \sum_j \alpha_j y_j \mathbf{z}_i^\top \mathbf{z}_j + b \leq 0$ then $\alpha_i \leftarrow \alpha_i + 1$.

Example: Perceptron Classification

- The regular perceptron (hyperplane classifier) was:

$$f(\mathbf{x}) = \text{sign}[\mathbf{w}^\top \mathbf{x} + b]$$

- To kernelize, we must represent the weights as linear combinations of the input vector images (representer theorem says this is OK):

$$\mathbf{w} = \sum_i (\alpha_i y_i) \mathbf{z}_i$$

- The original can be rewritten in terms of dot products:

$$f(\mathbf{z}) = \text{sign}[\sum_i (\alpha_i y_i) \mathbf{z}^\top \mathbf{z}_i]$$

Example: Ridge Regression

- Think of the ridge regression cost function:
  $\sum_i (y_i - \mathbf{w}^\top \mathbf{x}_i)^2 + \lambda \|\mathbf{w}\|^2$
  minimizing this is equivalent to minimizing:
  $\sum_i \eta_i^2 + \lambda \|\mathbf{w}\|^2$ subject to $\eta_i = (y_i - \mathbf{w}^\top \mathbf{x}_i)$.
- Let's introduce Lagrange multipliers to enforce the constraints:
  $\min \sum_i \eta_i^2 + \lambda \|\mathbf{w}\|^2 + \sum_i \alpha_i (\eta_i - (y_i - \mathbf{w}^\top \mathbf{x}_i))$
- Setting partial derivatives to zero gives:
  $\mathbf{w}^* = (1/2\lambda) + \sum_i \alpha_i \mathbf{x}_i$ and $\eta_i = \alpha_i / 2$
- Plugging back into the original cost gives:
  $\min \sum_i y_i \alpha_i - \frac{1}{4\lambda} \sum_{ij} \alpha_i \alpha_j \mathbf{x}_i^\top \mathbf{x}_j - \frac{1}{4} - \sum_i \alpha_i^2$
- In matrix form:
  $\min \mathbf{y}^\top \alpha - \frac{1}{4\lambda} \alpha^\top G \alpha \frac{1}{4} \alpha^\top \alpha$
- Completely Kernelized!

Example: PCA

- Not convinced yet. Let's do PCA using only dot products!
- Standard PCA (assume data is zero mean):

$$C = \frac{1}{N} = \sum_i \mathbf{x}_i \mathbf{x}_i^\top$$
$$\lambda \mathbf{v} = C\mathbf{v}$$

- All eigenvectors with nonzero eigenvalues must lie in the span of the data, and thus can be written as linear combinations of the data (think about why...):

$$\mathbf{v} = \sum_i \alpha_i \mathbf{x}_i$$

- Now we can rewrite the eigenvector condition:

$$\lambda \sum_i \alpha_i \mathbf{x}_i = \frac{1}{N} \sum_{ij} \alpha_i \mathbf{x}_j \mathbf{x}_i^\top \mathbf{x}_j$$

Example: Fisher Discriminant

- See if you can figure this one out on your own...
  (or look it up)

Example: PCA

- Eigenvector condition:

$$\lambda \sum_i \alpha_i \mathbf{x}_i = \frac{1}{N} \sum_{ij} \alpha_i \mathbf{x}_j \mathbf{x}_i^\top \mathbf{x}_j$$

- Take the dot product with $\mathbf{x}_k$ on left and right:

$$\lambda \sum_i \alpha_i \mathbf{x}_k^\top \mathbf{x}_i = \frac{1}{N} \sum_{ij} \alpha_i \mathbf{x}_k^\top \mathbf{x}_j \mathbf{x}_i^\top \mathbf{x}_j$$

- The above is true for all k, so write it as a vector equation in α:

$$N\lambda G\alpha = G^2\alpha$$
$$N\lambda\alpha = G\alpha$$

- Result: Form G, find its eigenvectors α, and use these to construct linear combinations of original data points which are the eigenvectors of the original covariance matrix.
  (Careful! Zero mean and normalization of eigenvectors.)

The Desire for Sparsity

- In kernel machines, the principal trick is to convert the problem into a *dual* form, which usually involves representing everything in the feature space as a linear combination of images of the training points: $\mathbf{z} = \sum_i \alpha_i \phi(\mathbf{x}_i)$
- Then we do all our calculations with the dual variables α_i and we never have to "touch" feature space directly.
- For very large datasets, it is desirable to have many of the coefficients α_i be exactly zero (sparsity) to reduce computational load, especially at test time.
- As a *separate trick*, different from the kernel trick, we can look for ways to make things sparse.
- These tricks are often confused, because the most famous kernel machine (the SVM) used them both.

Controlling Overfitting

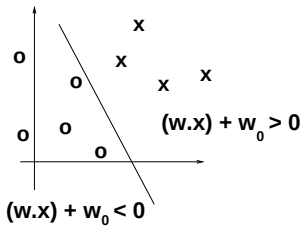
- A *third* aspect to kernel machines is how to control overfitting.
- For example, in the perceptron, if we use a very nonlinear kernel, we might always be able to separate our data exactly.
  Then we could be seriously overfitting.
- We can use *weight decay* to prevent this, by penalizing $\|\mathbf{w}\|^2$ in addition to trying to separate our training sample in the feature space. This is equivalent to *maximum margin*.
- A deep motivation for weight decay in this context comes from minimizing error bounds based on the "VC dimension" theory.
- This idea is often confused with the kernel & sparsity tricks because the most famous kernel machine (the SVM) also used weight decay to control overfitting and discussed the VC motivation.

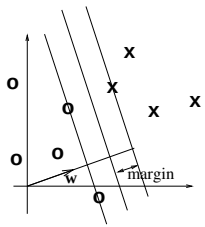
Maximum Margin = Minimum Norm

- Maximizing the margin is equivalent to picking the separating hyperplane that minimizes the norm of the weight vector:
  $\min \|\mathbf{w}\|^2$ subject to $y_i[\mathbf{w}^\top \mathbf{x}_i + b] \geq 1$
- Use Lagrange multipliers to enforce the constraint:
  $\min \|\mathbf{w}\|^2 - \sum_i \alpha_i (y_i[\mathbf{w}^\top \mathbf{x}_i + b] - 1) \qquad \alpha_i \geq 0$
- We can convert it to *dual form* by setting partial derivatives to zero and substituting.
- This is just like ridge-regression or weight decay.

Maximum Margin Hyperplane

- Margin = minimum distance to the plane of any point.
- Principle: of all the hyperplanes that separate the data perfectly, pick the one which maximizes the margin
- Since the scale is arbitrary, we will set the numerical value of the margin to be 1.
- Now maximizing the margin is equivalent to picking the separating hyperplane that minimizes the norm of the weight vector:
  $\min \|\mathbf{w}\|^2$ subject to $y_i[\mathbf{w}^\top \mathbf{x}_i + b] \geq 1$





Primal –> Dual

- Use Lagrange multipliers to enforce the constraint:
  $\min \|\mathbf{w}\|^2 - \sum_i \alpha_i (y_i[\mathbf{w}^\top \mathbf{x}_i + b] - 1) \qquad \alpha_i \geq 0$
- set $\partial/\partial \mathbf{w} = 0$ and $\partial/\partial b = 0$: $\mathbf{w}^* = \sum_i y_i \alpha_i \mathbf{x}_i \qquad \sum_i y_i \alpha_i = 0$
- The dual problem is now: $\min \sum_i \alpha_i - \frac{1}{2} \sum_{ij} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^\top \mathbf{x}_j$
  $\alpha_i \geq 0 \qquad \sum_i y_i \alpha_i = 0$
- This is a *quadratic programming* problem.
- It is convex. Unique solution!

Sparsity of Solution

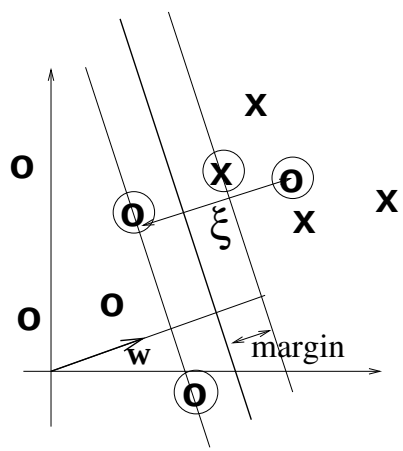
- Not only is the solution unique, but it is also sparse.
- Only the training points nearest to the separating hyperplane (ie with margin exactly 1) have $\alpha_i > 0$. These points are called the "active" points, or the *support vectors* since the final weight vector depends only on them:

$$\mathbf{w}^* = \sum_i y_i \alpha_i \mathbf{x}_i$$

- This is a lucky coincidence that has confused many people: in the case of SVM classification the two goals of controlling overfitting and inducing sparsity can both be achieved simultaneously with only a single trick: maximum margin (minimum weight norm).
- But it is not always like this.

Support Vector Machines

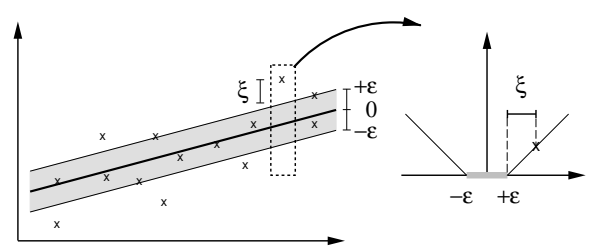
- A *support vector machine* (SVM) is nothing more than a kernelized maximum-margin hyperplane classifier.
- You train it by solving the dual quadratic programming problem.
- You run it by evaluating the kernel function between the test point and each of the "active" training points, called support vectors.
- This combination of (1) kernel trick, (2) maximum margin (minimum norm) and (3) the resulting sparsity has turned out to be very effective and popular.
- In practice, the hard part from a learning point of view is selecting the kernel function (there is a lot of research on this) and from a computational point of view it is solving the large QP efficiently.



Sparsity in Regression

- To introduce sparsity in regression, Vapnik introduced the *epsilon-insensitive loss* function:

$$l(\hat{y}) = 0 \qquad \text{if} \qquad |y - \hat{y}| \leq \epsilon$$
$$l(\hat{y}) = |y - \hat{y}| \qquad \text{if} \qquad |y - \hat{y}| > \epsilon$$



Lots More

- VC Dimension and Error Bounds, the theoretical origins of support vector machines.
- Lots of variations of SVMS, including the ν-SVM, 1-norm SVM, 1-class SVM, etc.
- Kernelized versions of lots of standard algorithms (PCA, Fisher's discriminant, Canonical Correlations Analysis, ...)
- Other kernel machines (e.g. Gaussian processes, links to boosting)
- see `http://www.kernel-machines.org` for lots of papers/tutorials, etc

Thanks! It's been fun!

- Last class.
  Projects due Dec20, 5pm, by email to csc2515@cs (attachment/url)
  Postscript or PDF ONLY, in NIPS format, max 8 pages.
  Readings must be completed by Dec20, 5pm, (csc2515readings@cs)
- Thanks for sticking with it.
  Hope you learned something, and had fun also.
  Sorry about all the math.
- Please send me comments/corrections for my book and for next year's course.