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. \( \mathbb{R}^n \)) 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 $z_1^T z_2$ between images of two input points $x_1, x_2$ very efficiently. This is true when the components of $\phi$ are the eigenfunctions of a special class of positive definite functions called Mercer Kernels, in which case:

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

- 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.

Examples of Kernels

- Linear: $K(x_1, x_2) = x_1^T x_2$  \hspace{1cm} ($\phi(x) = x$)
- Affine: $K(x_1, x_2) = x_1^T Q x_2$  \hspace{1cm} ($Q$ symmetric pos.def)
- Gaussian: $K(x_1, x_2) = \exp[-.5 \|x_1 - x_2\|^2]$  \hspace{1cm} (watch scaling!)
- Polynomial: $K(x_1, x_2) = (1 + x_1^T x_2)^k$
- Sigmoid: $K(x_1, x_2) = \tanh(a x_1^T x_2 + b)$
- 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:

\[
\|z_i\|^2 = K(x_i, x_i) + \sum_k K(x_k, x_i) - 2 \sum_k K(x_k, x_i x_i)
\]

Distance between a point and the mean of all others:

\[
\|z_i - \mu\|^2 = K(x_i, x_i) + \sum_k K(x_k, x_i) - \sum_k K(x_k, x_i x_i) - 2 \sum_k K(x_k, x_i x_i) + \sum_k \sum_i K(x_k, x_i) - 2 \sum_k \sum_i K(x_k, x_i x_i)
\]

Zero mean calculations in feature space:

\[
\langle z_i, z_i \rangle = K(x_i, x_i) + \sum_k K(x_k, x_i) - 2 \sum_k K(x_k, x_i x_i)
\]

The "Gram Matrix" is the N by N symmetric matrix of all pairwise kernel evaluations: \( G_{ij} = K(x_i, x_j) \).

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 \( \{x_i\} \).

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.

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 \( \{x_i\} \).

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 \( x_i \) appear only in dot products \( \langle x_i, x_j \rangle \).

Sometimes you need to make small modifications.

"Kernelizing" an algorithm can actually be pretty easy.

Thus a kernel function contains two modules: the kernel and the kernel function. Choosing the kernel function is a hard problem which we won’t discuss today.

Often you can do this and obtain an exactly equivalent algorithm to the one you started with.

"Kernelizing" an algorithm can actually be pretty easy.

Thus a kernel function contains two modules: the kernel and the kernel function. Choosing the kernel function is a hard problem which we won’t discuss today.

Often you can do this and obtain an exactly equivalent algorithm to the one you started with.

"Kernelizing" an algorithm can actually be pretty easy.

Thus a kernel function contains two modules: the kernel and the kernel function. Choosing the kernel function is a hard problem which we won’t discuss today.

Often you can do this and obtain an exactly equivalent algorithm to the one you started with.

"Kernelizing" an algorithm can actually be pretty easy.

Thus a kernel function contains two modules: the kernel and the kernel function. Choosing the kernel function is a hard problem which we won’t discuss today.

Often you can do this and obtain an exactly equivalent algorithm to the one you started with.
The regular perceptron (hyperplane classifier) was:
\[ f(x) = \text{sign}[w^T x + b] \]
To kernelize, we must represent the weights as linear combinations of the input vector images (representer theorem says this is OK):
\[ w = \sum_i (\alpha_i y_i) z_i \]
The original can be rewritten in terms of dot products:
\[ f(z) = \text{sign}\left[ \sum_i (\alpha_i y_i) z_i^T z_i \right] \]

Example: Ridge Regression

Think of the ridge regression cost function:
\[ \sum_i (y_i - w^T x_i)^2 + \lambda \| w \|^2 \]
minimizing this is equivalent to minimizing:
\[ \sum_i \eta_i^2 + \lambda \| w \|^2 \quad \text{subject to } \eta_i = (y_i - w^T x_i). \]
Let’s introduce Lagrange multipliers to enforce the constraints:
\[ \min \sum_i \eta_i^2 + \lambda \| w \|^2 + \sum_i \alpha_i (\eta_i - (y_i - w^T x_i)) \]
Setting partial derivatives to zero gives:
\[ w^* = (1/2\lambda) \sum_i \alpha_i x_i \quad \text{and} \quad \eta_i = \alpha_i/2 \]
Plugging back into the original cost gives:
\[ \min \sum_i y_i \alpha_i - \frac{1}{4\lambda} \sum_{ij} \alpha_i \alpha_j x_i^T x_j - \frac{1}{4} - \sum_i \alpha_i^2 \]
In matrix form:
\[ \min y^T \alpha - \frac{1}{4\lambda} \alpha^T G \alpha \quad \text{subject to } \alpha = \alpha^T \alpha \]
Completely Kernelized!

Example: PCA

Not convinced yet? Let’s do PCA using only dot products!
Standard PCA (assume data is zero mean):
\[ C = \frac{1}{N} \sum_i x_i x_i^T \]
\[ \lambda v = Cv \]
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...):
\[ v = \sum_i \alpha_i x_i \]
Now we can rewrite the eigenvector condition:
\[ \lambda \sum_i \alpha_i x_i = \frac{1}{N} \sum_{ij} \alpha_i x_j x_i^T x_j \]
**Example: PCA**

- Eigenvector condition:
  \[ \lambda \sum_i \alpha_i x_i = \frac{1}{N} \sum_{ij} \alpha_i x_j x_i^T \]

- Take the dot product with \( x_k \) on left and right:
  \[ \lambda \sum_i \alpha_i x_k^T x_i = \frac{1}{N} \sum_{ij} \alpha_i x_k^T x_j x_i^T x_j \]

- The above is true for all \( k \), so write it as a vector equation in \( \alpha \):
  \[
  \begin{align*}
  N \lambda G \alpha &= G^2 \alpha \\
  N \lambda \alpha &= G \alpha
  \end{align*}
  \]

- Result: Form \( G \), find its eigenvectors \( \alpha \), 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.)

**Example: Fisher Discriminant**

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

- Hint: The optimal Fisher discriminant weight can be written as the eigenvector corresponding to largest eigenvalue of a particular matrix, which is the inverse of the average within-class covariance times the average between-class covariance.

- Express the optimal weight as a linear combination of the examples and the within-class and between-class covariances in terms of those linear combination coefficients and the Gram matrix.

- You get a new eigenvector problem of size equal to the number of datapoints as opposed to the dimension of the inputs.

**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:
  \[ z = \sum_i \alpha_i \phi(x_i) \]

- Then we do all our calculations with the dual variables \( \alpha_i \) and we never have to “touch” feature space directly.

- For very large datasets, it is desirable to have many of the coefficients \( \alpha_i \) 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 \( \|w\|^2 \) 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 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:
  \[
  \min_k \|w\|^2 \quad \text{subject to} \quad y_i[w^T x_i + b] \geq 1
  \]

![Diagram of linear separation with hyperplane](image)

Primal -> Dual

- Use Lagrange multipliers to enforce the constraint:
  \[
  \min \|w\|^2 - \sum_i \alpha_i (y_i[w^T x_i + b] - 1) \quad \alpha_i \geq 0
  \]
- set \( \partial/\partial w = 0 \) and \( \partial/\partial b = 0 \): \( w^* = \sum_i y_i \alpha_i x_i \quad \sum_i y_i \alpha_i = 0 \)
- The dual problem is now:
  \[
  \min \sum_i \alpha_i - \frac{1}{2} \sum_{ij} \alpha_i \alpha_j y_i y_j x_i^T x_j \quad \alpha_i \geq 0 \quad \sum_i y_i \alpha_i = 0
  \]
- This is a quadratic programming problem.
- It is convex. Unique solution!
- If we are allowing slack, we must also penalize the total amount of slack by adding \( \sum_i \xi_i \) to the (primal) objective function.

Maximum Margin = Minimum Norm

- Maximizing the margin is equivalent to picking the separating hyperplane that minimizes the norm of the weight vector:
  \[
  \min \|w\|^2 \quad \text{subject to} \quad y_i[w^T x_i + b] \geq 1
  \]
- Use Lagrange multipliers to enforce the constraint:
  \[
  \min \|w\|^2 - \sum_i \alpha_i (y_i[w^T x_i + b] - 1) \quad \alpha_i \geq 0
  \]
- We can convert it to dual form by setting partial derivatives to zero and substituting.
- This is just like ridge-regression or weight decay.
- We can also allow some points to violate the margin by being inside it or even by being on the wrong side of it. This is achieved by adding non-negative “slack variables” \( \xi_i \) and modifying the constraints to the form:
  \[
  y_i[w^T x_i + b] \geq 1 - \xi_i
  \]

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:
  \[
  w^* = \sum_i y_i \alpha_i x_i
  \]
- 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.
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.
Thanks! It’s been fun!

• Last class.
  Projects due Dec19, 9am, by email to csc2515@cs (attachment/url)
  Postscript or PDF ONLY, in NIPS format, max 8 pages.
  Readings must be also completed by Dec19, 9am,
  (csc2515readings@cs)
• Thanks for sticking with it.
  Hope you learned something, and had fun also.
  Sorry about all the math and about A2.
• Please send me comments/corrections for the notes
  so I can improve next year’s course.