**STA561: Probabilistic machine learning**

# Kernels and Kernel Methods (10/09/13)

*Lecturer: Barbara Engelhardt* *Scribes: Yue Dai, Li Lu, Will Wu*

## 1 Kernel Functions

### 1.1 What are Kernels?

*Kernels* are a way to represent your data samples flexibly so that you can compare the samples in a complex space. Kernels have shown great utility in comparing

- images of different sizes
- protein sequences of different lengths
- object 3D structures
- networks with different numbers of edges and/or nodes
- text documents of different lengths and formats.

All of these objects have different numbers and types of features. We want to be able to cluster data samples to find which pairs are neighbors in this complex, high dimensional space. A kernel is an arbitrary function that lets us map objects in this complex space to a high dimensional space that enables comparisons of these complex features in a simple way. We have an $\mathcal{X}$ space of our samples, and a feature space that we define by first defining a kernel function. This helps with:

1. Comparing: it may be hard to compare two different text documents with different number of words. A properly-defined kernel gives us a metric by which we can quantify the similarities between two objects;
2. Classification: even if we can quantify similarity in our feature space, simple classifiers may not perform well in this space. We may want to project our data to a different space and classify our samples in this space.

Previously in class, we used a fixed set of features and a probabilistic modeling approach. We turn to large margin classifiers and a kernel-based approach in this lecture.

### 1.2 Kernel function.

Given some abstract space $\mathcal{X}$ (e.g., documents, images, proteins, etc.), function $\kappa : \mathcal{X} \times \mathcal{X} \mapsto \mathbb{R}$ is called a **kernel function**. Kernel functions are used to quantify similarity between a pair of objects $\mathbf{x}$ and $\mathbf{x}'$ in $\mathcal{X}$.

A kernel function typically satisfies the following two properties (but this is not required for all kernel methods). A kernel with these properties will loosely have the interpretation as a similarity quantification between the two objects.

**(symmetric)** $\forall \mathbf{x}, \mathbf{x}' \in \mathcal{X},\ \kappa(\mathbf{x}, \mathbf{x}') = \kappa(\mathbf{x}', \mathbf{x})$,

**(non-negative)** $\forall \mathbf{x}, \mathbf{x}' \in \mathcal{X},\ \kappa(\mathbf{x}, \mathbf{x}') \geq 0$.

## 1.3 Mercer Kernels

Let $X = \{x_1, \ldots, x_n\}$ be a finite set of $n$ samples from $\mathcal{X}$. The *Gram matrix* of $X$ is defined as $\mathbf{K}(X; \kappa) \in \mathbb{R}^{n \times n}$, or $\mathbf{K}$ for short, such that $(\mathbf{K})_{ij} = \kappa(x_i, x_j)$. If $\forall X \subseteq \mathcal{X}$, the matrix $\mathbf{K}$ is positive definite, $\kappa$ is called a *Mercer Kernel*, or a *positive definite kernel*. A Mercer kernel will be symmetric by definition (i.e., $\mathbf{K} = \mathbf{K}^T$).

**Mercer's theorem.** If the Gram matrix is positive definite, we can compute an eigenvector decomposition of the Gram matrix as:

$$\mathbf{K} = \mathbf{U}^T \mathbf{\Lambda} \mathbf{U} \tag{1}$$

where $\mathbf{\Lambda} = diag(\lambda_1, \ldots, \lambda_n)$ ($\lambda_i$ is the $i$-th eigenvalue of $\mathbf{K}$ and will be greater than 0 because the matrix is positive definite). Consider an element of $\mathbf{K}$, we have a dot product between two vectors:

$$\mathbf{K}_{ij} = (\mathbf{\Lambda}^{1/2} \mathbf{U}_{:,i})^T (\mathbf{\Lambda}^{1/2} \mathbf{U}_{:,j}) \tag{2}$$

Define $\phi(\mathbf{x}_i) = \mathbf{\Lambda}^{1/2} \mathbf{U}_{:,i}$. The equation above can be re-written as

$$\mathbf{K}_{ij} = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) \tag{3}$$

Let's think about this function: each element of the kernel can be described as the inner product of a function $\phi(\cdot)$ applied to objects $\mathbf{x}, \mathbf{x}'$. Each element of the Mercer kernel lives in the Hilbert space, where a Hilbert space is an abstract vector space defined by the inner product of two arbitrary vectors.

Therefore, if our kernel is a Mercer kernel, then there exists a function $\phi : \mathcal{X} \mapsto D$ such that

$$\kappa(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T \phi(\mathbf{x}'). \tag{4}$$

To foreshadow upcoming concepts, we will call $\phi(\cdot)$ a *basis function*, and we will describe the space $D$ as *feature space*. We can then say that we map our objects to a feature space using a basis function.

Basis function $\phi(\cdot)$ (for a Mercer kernel) can be written as a linear combination of eigen functions of $\kappa$. There are absolutely no restrictions on the dimensionality of feature space $D$; in fact, $D$ is potentially infinite dimensional. Note that:

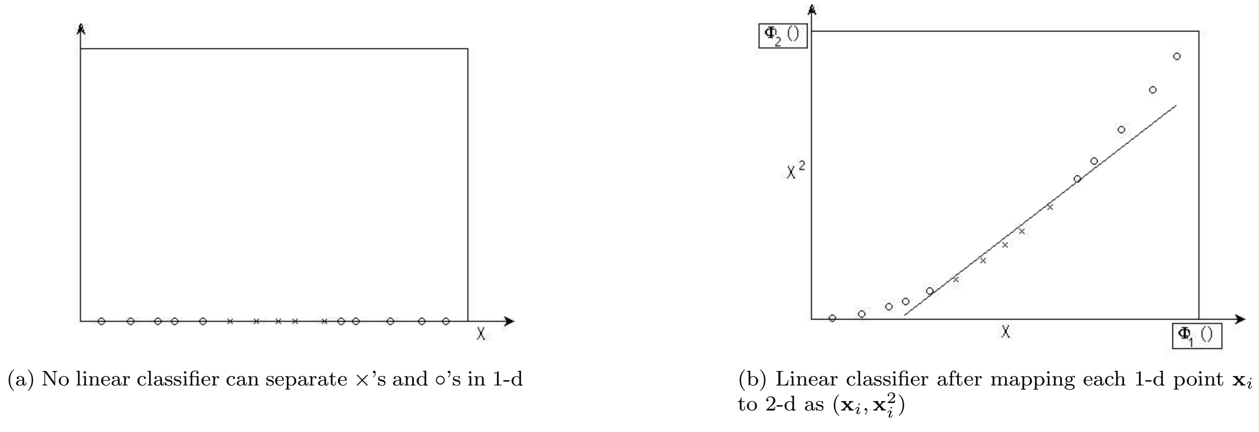
- Many kernel methods do not require us to explicitly compute $\phi(x)$, but instead we will compute the $n \times n$ Gram matrix using the kernel function $\kappa(\cdot, \cdot)$. In other words, we are able to build classifiers in arbitrarily complex $D$ feature space, but we do not have to compute any element of that space explicitly.

- Whereas computing $\phi(x)$ from $\kappa$ may be difficult (and is often unnecessary), it is straightforward to use intuitive basis functions $\phi(x)$ to construct the kernel $\kappa(\mathbf{x}, \mathbf{x}')$. We will see examples of this today.

### 1.4 Power of Kernels

Let's say we have $n$ scalar objects (i.e., a one dimensional input space), and a "kernelized" linear classifier. How can we separate $\times$'s and $\circ$'s with one line? The idea is to project them to a higher dimension, and use a linear classifier in that feature space.

Let $\phi(\mathbf{x}_i) = [\mathbf{x}_i, \mathbf{x}_i^2]$, and, by definition, $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \mathbf{x}_j + \mathbf{x}_i^2 \mathbf{x}_j^2$. This example projects an object in input space into a two-dimensional feature space using the basis function $\phi(\cdot)$, which is easy to compute. Now a linear classifier can separate the two classes perfectly (Figure 1).



(a) No linear classifier can separate $\times$'s and $\circ$'s in 1-d

(b) Linear classifier after mapping each 1-d point $\mathbf{x}_i$ to 2-d as $(\mathbf{x}_i, \mathbf{x}_i^2)$

Figure 1: Example showing the power of kernels for classification.

The idea behind kernel methods is to take a set of observations and project each of them to a space within which comparisons between points are straightforward. Because the dimension of the feature space is arbitrarily high, we can often use simple classifiers within this complex feature space, but we will need to be careful about testing for over fitting (although this comes later).

## 2 Examples of Kernels

### 2.1 Linear Kernels

Let $\phi(\mathbf{x}) = \mathbf{x}$, we get the **linear kernel**, defined by just the dot product between the two object vectors:

$$\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T \mathbf{x}' \tag{5}$$

The dimension of the feature space $D$ of a linear kernel is the same as the dimension of the input space $\mathcal{X}$, equivalent to the number of features of each object, and $\phi(x) = x$.

You might use a linear kernel when it is not necessary to work in an alternative feature space, if, for example, the original data are already high dimensional, comparable, and may be linearly separable into respective classes within the input space.

Linear kernels are good for objects represented by a large number of features of a fixed length, e.g., bag-of-words. Consider a vocabulary $V$ with $|V| = m$ distinct words. A feature vector $\mathbf{x} \in \mathbb{R}^m$ represents a document $\mathcal{D}$ using the count of each word of $V$ in the document (Figure 2). Note that, despite the documents possibly having very different word lengths, the feature vector for every document in a collection is exactly length $m$.

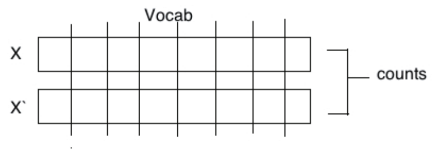


Figure 2: Feature vectors for bag-of-words

## 2.2 Gaussian Kernels

The *Gaussian kernel*, (also known as the *squared exponential kernel* – SE kernel – or *radial basis function* – RBF) is defined by

$$\kappa(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{1}{2}(\mathbf{x}-\mathbf{x}')^T \Sigma^{-1} (\mathbf{x}-\mathbf{x}')\right) \tag{6}$$

$\Sigma$, the covariance of each feature across observations, is a $p$-dimensional matrix. When $\Sigma$ is a diagonal matrix, this kernel can be written as

$$\kappa(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{1}{2}\sum_{j=1}^{p}\frac{1}{\sigma_j^2}.(x_j - x_j')^2\right) \tag{7}$$

$\sigma_j$ can be interpreted as defining the **characteristic length scale** of feature $j$. Furthermore, if $\Sigma$ is spherical, i.e., $\sigma_j = \sigma$, $\forall j$,

$$\kappa(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{||\mathbf{x}-\mathbf{x}'||^2}{2\sigma^2}\right) \tag{8}$$

For this kernel, the dimension of the feature space defined by $\phi(\cdot)$ is $D = \infty$. Our methods will allow us to avoid explicit computation of $\phi(\cdot)$. We can easily compute the $n \times n$ Gram matrix using this relative of the Mahalanobis Distance, even though we have implicitly projected our objects to an infinite dimensional feature space.

## 2.3 String Kernels

If we're interested in matching all substrings (for example) instead of representing an object as a bag of words, we can use a *string kernel*:

The quick br$\underline{\text{own}}$ fox ...
Yesterday I went to t$\underline{\text{own}}$ ...

Let $A$ denote an alphabet, e.g., $\{a, ..., z\}$, and $A^* = [A, A^2, \ldots, A^m]$, where $m$ is the length of the longest string we would like to match. Then, just as in the bag-of-words scenario, a basis function $\phi(\mathbf{x})$ will map a string $\mathbf{x}$ to a vector of length $|A^*|$, where each element $j$ is the number of times we observe substring $A_j^*$ in string $\mathbf{x}$, where $j = 1 : |A^*|$. [1]

[1]Superscripts here are regular expression operators. $A^i$ means the set of all possible strings of length $i$, with each position occupied by any character from the alphabet $A$. $^*$ is known as the Kleene star operator.

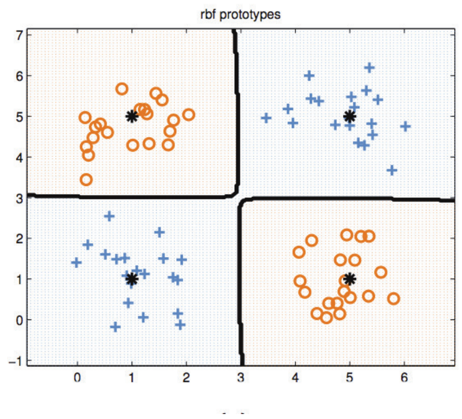


Figure 3: Fitting a linear logistic regression classifier using a Gaussian kernel with centroids specified by the 4 black crosses.

The *string kernel* measures the similarity of two strings $x$ and $x'$:

$$\kappa(x, x') = \sum_{s \in A^*} w_s \phi_s(x) \phi_s(x') \tag{9}$$

where $\phi_s(x)$ denotes the number of occurrences of substring $s$ in string $x$.

The size of the underlying space, $|A^*|$ is very large. Regardless of the size of the substring space $A^*$, $\kappa(x, x')$ can be computed in $O(|x| + |x'|)$, or linear time, for fixed weight function $w$, using suffix trees. A suffix tree contains all possible suffixes in a particular string. To compute $\kappa(\mathbf{x}, \mathbf{x}')$, build a $m$ level suffix tree using one string $\mathbf{x}$, and search in that tree for matches with the second string $\mathbf{x}'$. This is a linear time process.

We can design our kernel for our application by setting the weights $w$ to specific values. Here are a couple of special cases for the choice of weight function $w$.

- $w_s = 0$ for $|s| > 1$: comparing the alphabet between strings (substrings of length one)

- $w = 0$ for all words outside of a vocabulary: equivalent to (weighted) bag-of-words kernel

## 2.4 Fisher Kernels

We can construct a kernel based on a chosen generative model using the concept of a Fisher kernel. The idea is that this kernel represents the distance in likelihood space between different objects for a fitted generative model. A *Fisher kernel* is defined as

$$\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{g}(\mathbf{x})^T \mathbf{F}^{-1} \mathbf{g}(\mathbf{x}') \tag{10}$$

where $\mathbf{g}$ is the gradient of the log likelihood, evaluated at $\widehat{\theta}_{MLE}$, and $\mathbf{F}$ is the Hessian, i.e. $\mathbf{g}(\mathbf{x}) = \nabla_\theta \log p(\mathbf{x}|\theta)|_{\widehat{\theta}_{MLE}}$ and $\mathbf{F} = \nabla\nabla \log p(\mathbf{x}|\theta)|_{\widehat{\theta}_{MLE}}$

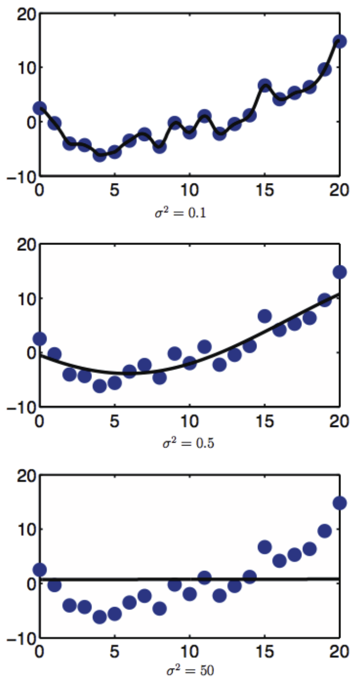


Figure 4: Fitting a 1D data set using $D = 10$ uniformly spaced Gaussian prototypes, with varying length scale $\sigma^2$ (0.1, 0.5, 50 from top to bottom)

## 3 Applications of Kernels

### 3.1 Kernelized Machine

$$\phi(\mathbf{x}) = [\kappa(\mathbf{x}, \mu_1), \kappa(\mathbf{x}, \mu_2), \ldots, \kappa(\mathbf{x}, \mu_D)] \tag{11}$$

where $\mu_d \in \mathcal{X}$ are a set of $D$ centroids. $\phi(\mathbf{x})$ is called a **kernelized feature vector**.

This vector projects a new data point $x$ into the $D$-dimensional feature space by computing the similarity between $\mathbf{x}$ and each centroid $\mu_{1:D}$ via $\kappa(\cdot, \cdot)$. Then we can use this *kernelized feature vector* for logistic regression by defining $p(y|\mathbf{x}, \theta) = \text{Ber}(\mathbf{w}^T \phi(\mathbf{x}))$.

$\phi(x)$ cannot be computed using the kernel trick (see below) and must be computed explicitly, so $D$ must be a reasonable size for computation. The kerneled feature vector quantifies the distance of a particular $\mathbf{x}$ to each of the predefined centroids. This provides a simple way to define a non-linear decision boundary using a linear classifier like logistic regression, as shown in Figure 3, using (for example) a Gaussian kernel $\kappa$ with four centroids.

We can also use the kernelized feature vector inside linear regression model by defining $p(y|\mathbf{x}, \theta) = \mathcal{N}(\beta^T \phi(\mathbf{x}), \sigma^2)$. Figure 4 shows the result of fitting a 1D data set with $D = 10$ uniformly spaced Gaussian prototypes, with varying length scale.

- $\sigma$ too small (e.g. $\sigma = 0.1$) $\Rightarrow$ overfitting.
- $\sigma$ generalizeable (e.g. $\sigma = 0.5$) $\Rightarrow$ right line nice generalization properties
- $\sigma$ too large (e.g. $\sigma = 50$) $\Rightarrow$ underfitting.

Consider the transition from high-dimension linear classifier to a one-dimension linear classifier. We first map the input one- dimensional space $\mathbf{x}$ up to a very high-dimensional feature space and fit a hyperplane in feature space that is the linear classifier. The hyperplane projected back onto the one-dimensional input space is not necessarily linear, as we see in this example.

**Kernel trick** Recall that our basis function $\phi(\mathbf{x})$ projects our $\mathbf{x}$ into feature space. In the above two methods, we define our feature vector in terms of the basis function, $\phi(x)$. In a number of other methods and statistical models, the feature vector $\mathbf{x}$ only enters the model through comparisons with other feature vectors $\mathbf{x}^T\mathbf{x}'$. If we *kernelize* these inner products, we get $\phi(x)^T\phi(\mathbf{x}) = \kappa(\mathbf{x}, \mathbf{x}')$. When we are allowed to compute only $\kappa(\mathbf{x}, \mathbf{x}')$ in computation, and avoid working in feature space, we enable large, possibly infinite-dimensional, feature spaces ($m \gg n$), whereas because we are using the kernelized inner products, we work in an $n \times n$ space (the Gram matrix). This is known as the **kernel trick**.

## 3.2 Kernelized K-Nearest Neighbor Classification

Given a data set $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$. Compute the Gram matrix of $x_1 : n$ (an $n \times n$ matrix) using kernel $\kappa(\mathbf{x}, \mathbf{x}')$. For new point $\mathbf{x}^*$, compute the vector $\kappa(\mathbf{x}_i, \mathbf{x}*)$ for all $\mathbf{x}_i, i = 1 : n$. Find the $k$ nearest neighbors by similarity in the kernel vector: points closest to $\mathbf{x}^*$ in the feature space. Return the classifications of those $k$ nearest neighbor points. This method exploits the kernel trick.

## 3.3 Support Vector Machines (SVMs)

As shown in Figure 6, an SVM classifier tries to find a linear separator (hyperplane) between training data points projected into feature space that maximizes the margin between the two classes of data points. It is a type of "large margin classifier".

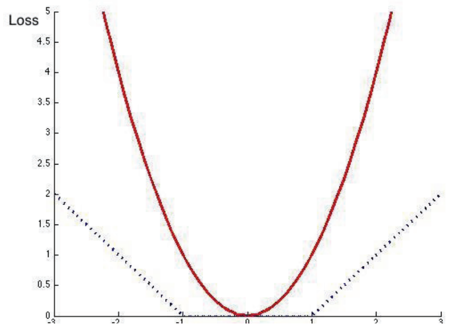


Figure 5: hinge loss function

A **support vector machine (SVM)** may be used for binary classification. First define $\eta = f(\mathbf{x}^*) = \mathbf{w}^T\mathbf{x}^* + w_0$, $y^* \in \{-1, 1\}$. This is our prediction for $y^*$ given a new $x^*$ and a fitted set of weights. It is the distance from our prediction to the hyperplane. We define the **hinge loss** as follows (Figure 5).

$$L_{hinge}(y, \eta) = \max(0, 1 - y\eta) \tag{12}$$

$1 - y\eta$ term can be thought of as the residual: the larger this value is, the worse our predictions for $y^*$ are compared with the truth; poor predictions result in a larger value for the hinge loss function. When prediction $|\eta| \geq 1$, even if the prediction sign matches the truth $y$, the hinge loss is zero. When the prediction

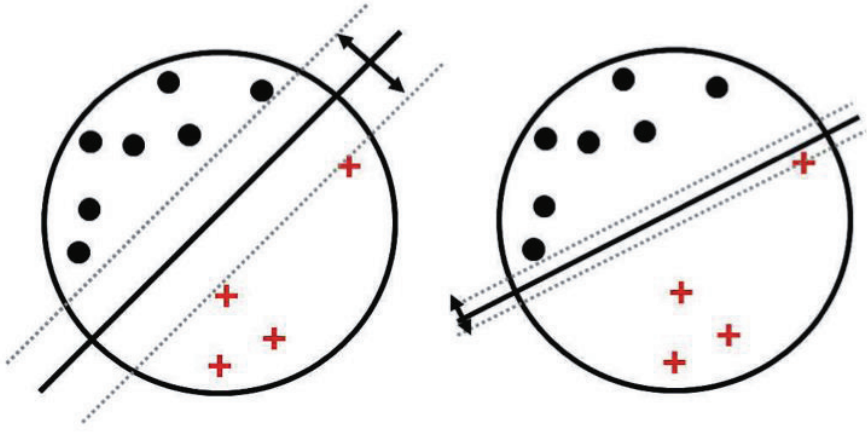


Figure 6: Illustration of the large margin principle. Left: a separating hyper-plane with large margin. Right: a separating hyper-plane with small (non-maximum) margin.

is between the margins (i.e., $0 \geq |\eta| \geq 1$) or incorrect (i.e., the sign of the prediction does not match the sign of the truth), the prediction incurs a loss.

The problem contains sparsity because any point that does not lie on the margin does not play a role in the optimization problem. The *support vectors* are the points that define the margin.

This is formally defined as the following optimization problem.

$$\min_{\mathbf{w},w_0} \frac{1}{2}||\mathbf{w}||^2 + c \cdot \sum_{i=1}^{n}(1 - y_i f(\mathbf{x}_i))_+ \tag{13}$$

This expression is not differentiable. But, by including a slack term $\xi_i$, and by replacing the hard constraint that $y_i f_i \geq 0$ with *soft margin constraints* that $y_i f_i \geq 1 - \xi_i$ (i.e., allowing some mistakes in classification), we get the following dual formulation:

$$\begin{gathered} \min_{\mathbf{w},w_0,\xi} \frac{1}{2}||\mathbf{w}||^2 + c \cdot \sum_{i=1}^{n} \xi_i, \;\; s.t. \\ \xi_i \geq 0 \\ y_i(\mathbf{x}_i^T \mathbf{w} + w_0) \geq 1 - \xi_i \end{gathered} \tag{14}$$

This is a quadratic program, and it takes $O(n^2)$ time to solve. Its solution takes the form $\hat{w} = \sum_{i=1}^{n} \alpha_i \mathbf{x}_i$, where $\alpha_i$ is sparse and selects support vectors that define either side of the margin. The prediction is done using:

$$\hat{y}(x) = \text{sign}(\hat{w_0} + \hat{\mathbf{w}}^T \mathbf{x}) \tag{15}$$

SVM can exploit the kernel trick to define a margin in feature space:

$$\hat{y}(\mathbf{x}^*) = \text{sign}(\hat{w_0} + \sum_{i=1}^{n} \alpha_i \kappa(\mathbf{x}_i, \mathbf{x}^*)). \tag{16}$$