14 *Kernels*

14.1 Introduction

So far in this book, we have been assuming that each object that we wish to classify or cluster or process in anyway can be represented as a fixed-size feature vector, typically of the form $\mathbf{x}_i \in \mathbb{R}^D$. However, for certain kinds of objects, it is not clear how to best represent them as fixed-sized feature vectors. For example, how do we represent a text document or protein sequence, which can be of variable length? or a molecular structure, which has complex 3d geometry? or an evolutionary tree, which has variable size and shape?

One approach to such problems is to define a generative model for the data, and use the inferred latent representation and/or the parameters of the model as features, and then to plug these features in to standard methods. For example, in Chapter 28, we discuss deep learning, which is essentially an unsupervised way to learn good feature representations.

Another approach is to assume that we have some way of measuring the similarity between objects, that doesn't require preprocessing them into feature vector format. For example, when comparing strings, we can compute the edit distance between them. Let $\kappa(\mathbf{x}, \mathbf{x}') \geq 0$ be some measure of similarity between objects $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$, where $\mathcal{X}$ is some abstract space; we will call κ a **kernel function**. Note that the word "kernel" has several meanings; we will discuss a different interpretation in Section 14.7.1.

In this chapter, we will discuss several kinds of kernel functions. We then describe some algorithms that can be written purely in terms of kernel function computations. Such methods can be used when we don't have access to (or choose not to look at) the "inside" of the objects $\mathbf{x}$ that we are processing.

14.2 Kernel functions

We define a **kernel function** to be a real-valued function of two arguments, $\kappa(\mathbf{x}, \mathbf{x}') \in \mathbb{R}$, for $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$. Typically the function is symmetric (i.e., $\kappa(\mathbf{x}, \mathbf{x}') = \kappa(\mathbf{x}', \mathbf{x})$), and non-negative (i.e., $\kappa(\mathbf{x}, \mathbf{x}') \geq 0$), so it can be interpreted as a measure of similarity, but this is not required. We give several examples below.

14.2.1 RBF kernels

The **squared exponential kernel** (SE kernel) or **Gaussian kernel** is defined by

$$\kappa(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{1}{2}(\mathbf{x} - \mathbf{x}')^T \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \mathbf{x}')\right) \tag{14.1}$$

If $\boldsymbol{\Sigma}$ is diagonal, this can be written as

$$\kappa(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{1}{2}\sum_{j=1}^{D}\frac{1}{\sigma_j^2}(x_j - x_j')^2\right) \tag{14.2}$$

We can interpret the σ_j as defining the **characteristic length scale** of dimension j. If $\sigma_j = \infty$, the corresponding dimension is ignored; hence this is known as the **ARD kernel**. If $\boldsymbol{\Sigma}$ is spherical, we get the isotropic kernel

$$\kappa(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{||\mathbf{x} - \mathbf{x}'||^2}{2\sigma^2}\right) \tag{14.3}$$

Here σ^2 is known as the **bandwidth**. Equation 14.3 is an example of a a **radial basis function** or **RBF** kernel, since it is only a function of $||\mathbf{x} - \mathbf{x}'||$.

14.2.2 Kernels for comparing documents

When performing document classification or retrieval, it is useful to have a way of comparing two documents, $\mathbf{x}_i$ and $\mathbf{x}_{i'}$. If we use a bag of words representation, where x_{ij} is the number of times words j occurs in document i, we can use the **cosine similarity**, which is defined by

$$\kappa(\mathbf{x}_i, \mathbf{x}_{i'}) = \frac{\mathbf{x}_i^T \mathbf{x}_{i'}}{||\mathbf{x}_i||_2 ||\mathbf{x}_{i'}||_2} \tag{14.4}$$

This quantity measures the cosine of the angle between $\mathbf{x}_i$ and $\mathbf{x}_{i'}$ when interpreted as vectors. Since $\mathbf{x}_i$ is a count vector (and hence non-negative), the cosine similarity is between 0 and 1, where 0 means the vectors are orthogonal and therefore have no words in common.

Unfortunately, this simple method does not work very well, for two main reasons. First, if $\mathbf{x}_i$ has any word in common with $\mathbf{x}_{i'}$, it is deemed similar, even though some popular words, such as "the" or "and" occur in many documents, and are therefore not discriminative. (These are known as **stop words**.) Second, if a discriminative word occurs many times in a document, the similarity is artificially boosted, even though word usage tends to be bursty, meaning that once a word is used in a document it is very likely to be used again (see Section 3.5.5).

Fortunately, we can significantly improve performance using some simple preprocessing. The idea is to replace the word count vector with a new feature vector called the **TF-IDF** representation, which stands for "term frequency inverse document frequency". We define this as follows. First, the term frequency is defined as a log-transform of the count:

$$\text{tf}(x_{ij}) \triangleq \log(1 + x_{ij}) \tag{14.5}$$

This reduces the impact of words that occur many times within one document. Second, the inverse document frequency is defined as

$$\text{idf}(j) \triangleq \log \frac{N}{1 + \sum_{i=1}^{N} \mathbb{I}(x_{ij} > 0)} \tag{14.6}$$

where N is the total number of documents, and the denominator counts how many documents contain term j. Finally, we define

$$\text{tf-idf}(\mathbf{x}_i) \triangleq [\text{tf}(x_{ij}) \times \text{idf}(j)]_{j=1}^{V} \tag{14.7}$$

(There are several other ways to define the tf and idf terms, see (Manning et al. 2008) for details.) We then use this inside the cosine similarity measure. That is, our new kernel has the form

$$\kappa(\mathbf{x}_i, \mathbf{x}_{i'}) = \frac{\boldsymbol{\phi}(\mathbf{x}_i)^T \boldsymbol{\phi}(\mathbf{x}_{i'})}{||\boldsymbol{\phi}(\mathbf{x}_i)||_2 ||\boldsymbol{\phi}(\mathbf{x}_{i'})||_2} \tag{14.8}$$

where $\boldsymbol{\phi}(\mathbf{x}) = \text{tf-idf}(\mathbf{x})$. This gives good results for information retrieval (Manning et al. 2008).

A probabilistic interpretation of the tf-idf kernel is given in (Elkan 2005).

14.2.3 Mercer (positive definite) kernels

Some methods that we will study require that the kernel function satisfy the requirement that the **Gram matrix**, defined by

$$\mathbf{K} = \begin{pmatrix} \kappa(\mathbf{x}_1, \mathbf{x}_1) & \cdots & \kappa(\mathbf{x}_1, \mathbf{x}_N) \\ & \vdots & \\ \kappa(\mathbf{x}_N, \mathbf{x}_1) & \cdots & \kappa(\mathbf{x}_N, \mathbf{x}_N) \end{pmatrix} \tag{14.9}$$

be positive definite for any set of inputs $\{\mathbf{x}_i\}_{i=1}^N$. We call such a kernel a **Mercer kernel**, or **positive definite kernel**. It can be shown (Schoelkopf and Smola 2002) that the Gaussian kernel is a Mercer kernel as is the cosine similarity kernel (Sahami and Heilman 2006).

The importance of Mercer kernels is the following result, known as **Mercer's theorem**. If the Gram matrix is positive definite, we can compute an eigenvector decomposition of it as follows

$$\mathbf{K} = \mathbf{U}^T \boldsymbol{\Lambda} \mathbf{U} \tag{14.10}$$

where $\boldsymbol{\Lambda}$ is a diagonal matrix of eigenvalues $\lambda_i > 0$. Now consider an element of $\mathbf{K}$:

$$k_{ij} = (\boldsymbol{\Lambda}^{\frac{1}{2}} \mathbf{U}_{:,i})^T (\boldsymbol{\Lambda}^{\frac{1}{2}} \mathbf{U}_{:j}) \tag{14.11}$$

Let us define $\boldsymbol{\phi}(\mathbf{x}_i) = \boldsymbol{\Lambda}^{\frac{1}{2}} \mathbf{U}_{:i}$. Then we can write

$$k_{ij} = \boldsymbol{\phi}(\mathbf{x}_i)^T \boldsymbol{\phi}(\mathbf{x}_j) \tag{14.12}$$

Thus we see that the entries in the kernel matrix can be computed by performing an inner product of some feature vectors that are implicitly defined by the eigenvectors $\mathbf{U}$. In general, if the kernel is Mercer, then there exists a function $\boldsymbol{\phi}$ mapping $\mathbf{x} \in \mathcal{X}$ to $\mathbb{R}^D$ such that

$$\kappa(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^T \boldsymbol{\phi}(\mathbf{x}') \tag{14.13}$$

where $\boldsymbol{\phi}$ depends on the eigen *functions* of κ (so D is a potentially infinite dimensional space).

For example, consider the (non-stationary) **polynomial kernel** $\kappa(\mathbf{x}, \mathbf{x}') = (\gamma \mathbf{x}^T \mathbf{x}' + r)^M$, where $r > 0$. One can show that the corresponding feature vector $\boldsymbol{\phi}(\mathbf{x})$ will contain all terms up to degree M. For example, if $M = 2$, $\gamma = r = 1$ and $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^2$, we have

$$(1 + \mathbf{x}^T \mathbf{x}')^2 = (1 + x_1 x_1' + x_2 x_2')^2 \tag{14.14}$$

$$= 1 + 2x_1 x_1' + 2x_2 x_2' + (x_1 x_1)^2 + (x_2 x_2')^2 + 2x_1 x_1' x_2 x_2' \tag{14.15}$$

This can be written as $\boldsymbol{\phi}(\mathbf{x})^T\boldsymbol{\phi}(\mathbf{x}')$, where

$$\boldsymbol{\phi}(\mathbf{x}) = [1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1x_2]^T \tag{14.16}$$

So using this kernel is equivalent to working in a 6 dimensional feature space. In the case of a Gaussian kernel, the feature map lives in an infinite dimensional space. In such a case, it is clearly infeasible to explicitly represent the feature vectors.

An example of a kernel that is not a Mercer kernel is the so-called **sigmoid kernel**, defined by

$$\kappa(\mathbf{x}, \mathbf{x}') = \tanh(\gamma\mathbf{x}^T\mathbf{x}' + r) \tag{14.17}$$

(Note that this uses the $\tanh$ function even though it is called a sigmoid kernel.) This kernel was inspired by the multi-layer perceptron (see Section 16.5), but there is no real reason to use it. (For a true "neural net kernel", which is positive definite, see Section 15.4.5.)

In general, establishing that a kernel is a Mercer kernel is difficult, and requires techniques from functional analysis. However, one can show that it is possible to build up new Mercer kernels from simpler ones using a set of standard rules. For example, if κ_1 and κ_2 are both Mercer, so is $\kappa(\mathbf{x}, \mathbf{x}') = \kappa_1(\mathbf{x}, \mathbf{x}') + \kappa_2(\mathbf{x}, \mathbf{x}')$. See e.g., (Schoelkopf and Smola 2002) for details.

14.2.4 Linear kernels

Deriving the feature vector implied by a kernel is in general quite difficult, and only possible if the kernel is Mercer. However, deriving a kernel from a feature vector is easy: we just use

$$\kappa(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^T\boldsymbol{\phi}(\mathbf{x}') = \langle\boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\phi}(\mathbf{x}')\rangle \tag{14.18}$$

If $\boldsymbol{\phi}(\mathbf{x}) = \mathbf{x}$, we get the **linear kernel**, defined by

$$\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T\mathbf{x}' \tag{14.19}$$

This is useful if the original data is already high dimensional, and if the original features are individually informative, e.g., a bag of words representation where the vocabulary size is large, or the expression level of many genes. In such a case, the decision boundary is likely to be representable as a linear combination of the original features, so it is not necessary to work in some other feature space.

Of course, not all high dimensional problems are linearly separable. For example, images are high dimensional, but individual pixels are not very informative, so image classification typically requires non-linear kernels (see e.g., Section 14.2.7).

14.2.5 Matern kernels

The **Matern kernel**, which is commonly used in Gaussian process regression (see Section 15.2), has the following form

$$\kappa(r) = \frac{2^{1-\nu}}{\Gamma(\nu)}\left(\frac{\sqrt{2\nu}r}{\ell}\right)^{\nu} K_\nu\left(\frac{\sqrt{2\nu}r}{\ell}\right) \tag{14.20}$$

where $r = ||\mathbf{x} - \mathbf{x}'||$, $\nu > 0$, $\ell > 0$, and K_ν is a modified Bessel function. As $\nu \to \infty$, this approaches the SE kernel. If $\nu = \frac{1}{2}$, the kernel simplifies to

$$\kappa(r) = \exp(-r/\ell) \tag{14.21}$$

If $D = 1$, and we use this kernel to define a Gaussian process (see Chapter 15), we get the **Ornstein-Uhlenbeck process**, which describes the velocity of a particle undergoing Brownian motion (the corresponding function is continuous but not differentiable, and hence is very "jagged").

14.2.6 String kernels

The real power of kernels arises when the inputs are structured objects. As an example, we now describe one way of comparing two variable length strings using a **string kernel**. We follow the presentation of (Rasmussen and Williams 2006, p100) and (Hastie et al. 2009, p668).

Consider two strings $\mathbf{x}$, and $\mathbf{x}'$ of lengths D, D', each defined over the alphabet $\mathcal{A}$. For example, consider two amino acid sequences, defined over the 20 letter alphabet $\mathcal{A} = \{A, R, N, D, C, E, Q, G, H, I, L, K, M, F, P, S, T, W, Y, V\}$. Let $\mathbf{x}$ be the following sequence of length 110

```
IPTSALVKETLALLSTHRTLLIANETLRIPVPVHKNHQLCTEEIFQGIGTLESQTVQGGTV
ERLFKNLSLIKKYIDGQKKKCGEERRRVNQFLDYLQEFLGVMNTEWI
```

and let $\mathbf{x}'$ be the following sequence of length 153

```
PHRRDLCSRSIWLARKIRSDLTALTESYVKHQGLWSELTEAERLQENLQAYRTFHVLLA
RLLEDQQVHFTPTEGDFHQAIHTLLLQVAAFAYQIEELMILLEYKIPRNEADGMLFEKK
LWGLKVLQELSQWTVRSIHDLRFISSHQTGIP
```

These strings have the substring `LQE` in common. We can define the similarity of two strings to be the number of substrings they have in common.

More formally and more generally, let us say that s is a substring of x if we can write $x = usv$ for some (possibly empty) strings u, s and v. Now let $\phi_s(x)$ denote the number of times that substring s appears in string x. We define the kernel between two strings x and x' as

$$\kappa(x, x') = \sum_{s \in \mathcal{A}^*} w_s \phi_s(x) \phi_s(x') \tag{14.22}$$

where $w_s \geq 0$ and $\mathcal{A}^*$ is the set of all strings (of any length) from the alphabet $\mathcal{A}$ (this is known as the Kleene star operator). This is a Mercer kernel, and be computed in $O(|x| + |x'|)$ time (for certain settings of the weights $\{w_s\}$) using suffix trees (Leslie et al. 2003; Vishwanathan and Smola 2003; Shawe-Taylor and Cristianini 2004).

There are various cases of interest. If we set $w_s = 0$ for $|s| > 1$ we get a bag-of-characters kernel. This defines $\boldsymbol{\phi}(x)$ to be the number of times each character in $\mathcal{A}$ occurs in x. If we require s to be bordered by white-space, we get a bag-of-words kernel, where $\boldsymbol{\phi}(x)$ counts how many times each possible word occurs. Note that this is a very sparse vector, since most words

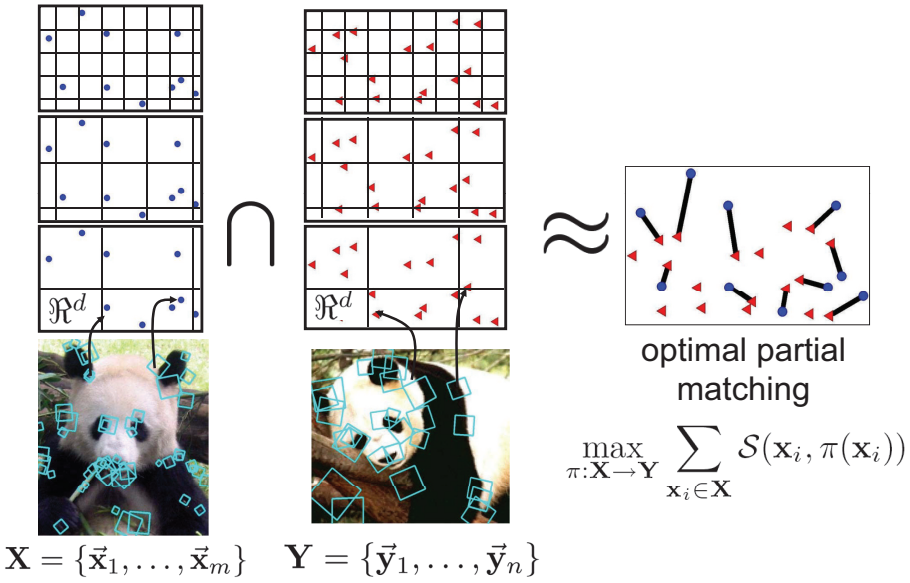


Figure 14.1 Illustration of a pyramid match kernel computed from two images. Used with kind permission of Kristen Grauman.

will not be present. If we only consider strings of a fixed length k, we get the **k-spectrum kernel**. This has been used to classify proteins into SCOP superfamilies (Leslie et al. 2003). For example if $k = 3$, we have $\phi_{LQE}(\mathbf{x}) = 1$ and $\phi_{LQE}(\mathbf{x}') = 2$ for the two strings above.

Various extensions are possible. For example, we can allow character mismatches (Leslie et al. 2003). And we can generalize string kernels to compare trees, as described in (Collins and Duffy 2002). This is useful for classifying (or ranking) parse trees, evolutionary trees, etc.

14.2.7 Pyramid match kernels

In computer vision, it is common to create a bag-of-words representation of an image by computing a feature vector (often using SIFT (Lowe 1999)) from a variety of points in the image, commonly chosen by an interest point detector. The feature vectors at the chosen places are then vector-quantized to create a bag of discrete symbols.

One way to compare two variable-sized bags of this kind is to use a **pyramid match kernel** (Grauman and Darrell 2007). The basic idea is illustrated in Figure 14.1. Each feature set is mapped to a multi-resolution histogram. These are then compared using weighted histogram intersection. It turns out that this provides a good approximation to the similarity measure one would obtain by performing an optimal bipartite match at the finest spatial resolution, and then summing up pairwise similarities between matched points. However, the histogram method is faster and is more robust to missing and unequal numbers of points. This is a Mercer kernel.

14.2.8 Kernels derived from probabilistic generative models

Suppose we have a probabilistic generative model of feature vectors, $p(\mathbf{x}|\boldsymbol{\theta})$. Then there are several ways we can use this model to define kernel functions, and thereby make the model suitable for discriminative tasks. We sketch two approaches below.

14.2.8.1 Probability product kernels

One approach is to define a kernel as follows:

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \int p(\mathbf{x}|\mathbf{x}_i)^\rho p(\mathbf{x}|\mathbf{x}_j)^\rho d\mathbf{x} \tag{14.23}$$

where $\rho > 0$, and $p(\mathbf{x}|\mathbf{x}_i)$ is often approximated by $p(\mathbf{x}|\hat{\boldsymbol{\theta}}(\mathbf{x}_i))$, where $\hat{\boldsymbol{\theta}}(\mathbf{x}_i)$ is a parameter estimate computed using a single data vector. This is called a **probability product kernel** (Jebara et al. 2004).

Although it seems strange to fit a model to a single data point, it is important to bear in mind that the fitted model is only being used to see how similar two objects are. In particular, if we fit the model to $\mathbf{x}_i$ and then the model thinks $\mathbf{x}_j$ is likely, this means that $\mathbf{x}_i$ and $\mathbf{x}_j$ are similar. For example, suppose $p(\mathbf{x}|\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\mu}, \sigma^2\mathbf{I})$, where σ^2 is fixed. If $\rho = 1$, and we use $\hat{\boldsymbol{\mu}}(\mathbf{x}_i) = \mathbf{x}_i$ and $\hat{\boldsymbol{\mu}}(\mathbf{x}_j) = \mathbf{x}_j$, we find (Jebara et al. 2004, p825) that

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \frac{1}{(4\pi\sigma^2)^{D/2}} \exp\left(-\frac{1}{4\sigma^2}||\mathbf{x}_i - \mathbf{x}_j||^2\right) \tag{14.24}$$

which is (up to a constant factor) the RBF kernel.

It turns out that one can compute Equation 14.23 for a variety of generative models, including ones with latent variables, such as HMMs. This provides one way to define kernels on variable length sequences. Furthermore, this technique works even if the sequences are of real-valued vectors, unlike the string kernel in Section 14.2.6. See (Jebara et al. 2004) for further details.

14.2.8.2 Fisher kernels

A more efficient way to use generative models to define kernels is to use a **Fisher kernel** (Jaakkola and Haussler 1998) which is defined as follows:

$$\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{g}(\mathbf{x})^T \mathbf{F}^{-1} \mathbf{g}(\mathbf{x}') \tag{14.25}$$

where $\mathbf{g}$ is the gradient of the log likelihood, or **score vector**, evaluated at the MLE $\hat{\boldsymbol{\theta}}$

$$\mathbf{g}(\mathbf{x}) \triangleq \nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}|\boldsymbol{\theta})\big|_{\hat{\boldsymbol{\theta}}} \tag{14.26}$$

and $\mathbf{F}$ is the Fisher information matrix, which is essentially the Hessian:

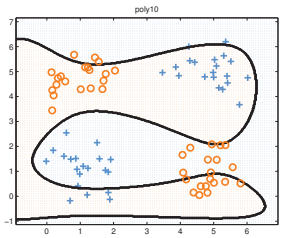
$$\mathbf{F} = \nabla\nabla \log p(\mathbf{x}|\boldsymbol{\theta})\big|_{\hat{\boldsymbol{\theta}}} \tag{14.27}$$

Note that $\hat{\boldsymbol{\theta}}$ is a function of all the data, so the similarity of $\mathbf{x}$ and $\mathbf{x}'$ is computed in the context of all the data as well. Also, note that we only have to fit one model.

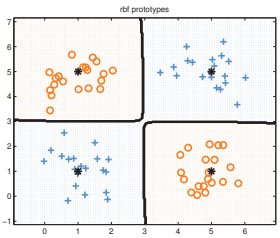
The intuition behind the Fisher kernel is the following: let $\mathbf{g}(\mathbf{x})$ be the direction (in parameter space) in which $\mathbf{x}$ would like the parameters to move (from $\hat{\boldsymbol{\theta}}$) so as to maximize its own

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

(a)



(b)



(c)

Figure 14.2 (a) xor truth table. (b) Fitting a linear logistic regression classifier using degree 10 polynomial expansion. (c) Same model, but using an RBF kernel with centroids specified by the 4 black crosses. Figure generated by `logregXorDemo`.

likelihood; call this the directional gradient. Then we say that two vectors $\mathbf{x}$ and $\mathbf{x}'$ are similar if their directional gradients are similar wrt the the geometry encoded by the curvature of the likelihood function (see Section 7.5.3).

Interestingly, it was shown in (Saunders et al. 2003) that the string kernel of Section 14.2.6 is equivalent to the Fisher kernel derived from an L'th order Markov chain (see Section 17.2). Also, it was shown in (Elkan 2005) that a kernel defined by the inner product of TF-IDF vectors (Section 14.2.2) is approximately equal to the Fisher kernel for a certain generative model of text based on the compound Dirichlet multinomial model (Section 3.5.5).

14.3 Using kernels inside GLMs

In this section, we discuss one simple way to use kernels for classification and regression. We will see other approaches later.

14.3.1 Kernel machines

We define a **kernel machine** to be a GLM where the input feature vector has the form

$$\boldsymbol{\phi}(\mathbf{x}) = [\kappa(\mathbf{x}, \boldsymbol{\mu}_1), \ldots, \kappa(\mathbf{x}, \boldsymbol{\mu}_K)] \tag{14.28}$$

where $\boldsymbol{\mu}_k \in \mathcal{X}$ are a set of K **centroids**. If κ is an RBF kernel, this is called an **RBF network**. We discuss ways to choose the $\boldsymbol{\mu}_k$ parameters below. We will call Equation 14.28 a **kernelised feature vector**. Note that in this approach, the kernel need not be a Mercer kernel.

We can use the kernelized feature vector for logistic regression by defining $p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathrm{Ber}(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}))$. This provides a simple way to define a non-linear decision boundary. As an example, consider the data coming from the **exclusive or** or **xor** function. This is a binary-valued function of two binary inputs. Its truth table is shown in Figure 14.2(a). In Figure 14.2(b), we have show some data labeled by the xor function, but we have **jittered** the points to make the picture clearer.[1] We see we cannot separate the data even using a degree 10 polynomial.

1. Jittering is a common visualization trick in statistics, wherein points in a plot/display that would otherwise land on top of each other are dispersed with uniform additive noise.

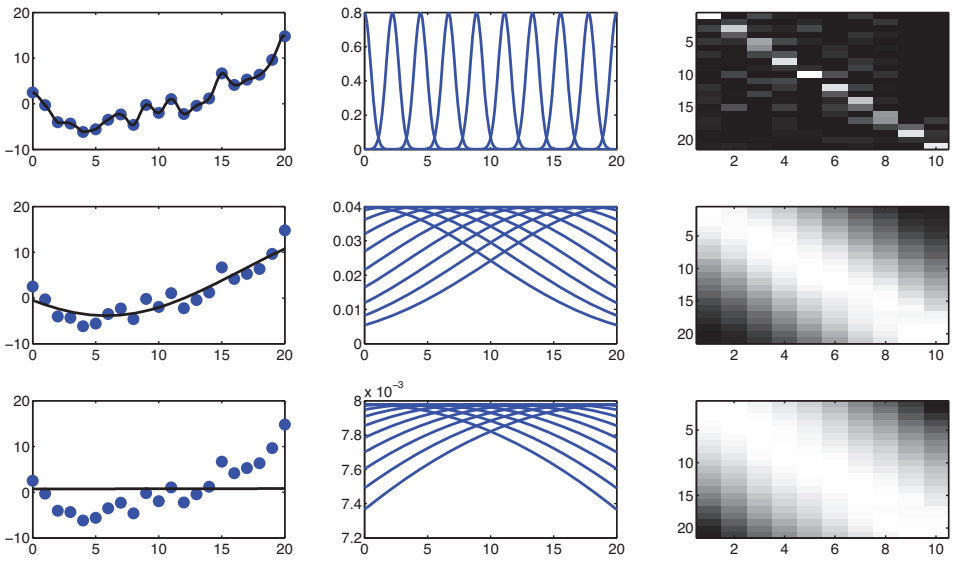


Figure 14.3 RBF basis in 1d. Left column: fitted function. Middle column: basis functions evaluated on a grid. Right column: design matrix. Top to bottom we show different bandwidths: $\tau = 0.1$, $\tau = 0.5$, $\tau = 50$. Figure generated by `linregRbfDemo`.

However, using an RBF kernel and just 4 prototypes easily solves the problem as shown in Figure 14.2(c).

We can also use the kernelized feature vector inside a linear regression model by defining $p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}), \sigma^2)$. For example, Figure 14.3 shows a 1d data set fit with $K = 10$ uniformly spaced RBF prototypes, but with the bandwidth ranging from small to large. Small values lead to very wiggly functions, since the predicted function value will only be non-zero for points $\mathbf{x}$ that are close to one of the prototypes $\boldsymbol{\mu}_k$. If the bandwidth is very large, the design matrix reduces to a constant matrix of 1's, since each point is equally close to every prototype; hence the corresponding function is just a straight line.

14.3.2 L1VMs, RVMs, and other sparse vector machines

The main issue with kernel machines is: how do we choose the centroids $\boldsymbol{\mu}_k$? If the input is low-dimensional Euclidean space, we can uniformly tile the space occupied by the data with prototypes, as we did in Figure 14.2(c). However, this approach breaks down in higher numbers of dimensions because of the curse of dimensionality. If $\boldsymbol{\mu}_k \in \mathbb{R}^D$, we can try to perform numerical optimization of these parameters (see e.g., (Haykin 1998)), or we can use MCMC inference, (see e.g., (Andrieu et al. 2001; Kohn et al. 2001)), but the resulting objective function / posterior is highly multimodal. Furthermore, these techniques is hard to extend to structured input spaces, where kernels are most useful.

Another approach is to find clusters in the data and then to assign one prototype per cluster

center (many clustering algorithms just need a similarity metric as input). However, the regions of space that have high density are not necessarily the ones where the prototypes are most useful for representing the output, that is, clustering is an unsupervised task that may not yield a representation that is useful for prediction. Furthermore, there is the need to pick the number of clusters.

A simpler approach is to make each example $\mathbf{x}_i$ be a prototype, so we get

$$\boldsymbol{\phi}(\mathbf{x}) = [\kappa(\mathbf{x}, \mathbf{x}_1), \ldots, \kappa(\mathbf{x}, \mathbf{x}_N)] \tag{14.29}$$

Now we see $D = N$, so we have as many parameters as data points. However, we can use any of the sparsity-promoting priors for $\mathbf{w}$ discussed in Chapter 13 to efficiently select a subset of the training exemplars. We call this a **sparse vector machine**.

The most natural choice is to use ℓ_1 regularization (Krishnapuram et al. 2005). (Note that in the multi-class case, it is necessary to use group lasso, since each exemplar is associated with C weights, one per class.) We call this **L1VM**, which stands for "ℓ_1-regularized vector machine". By analogy, we define the use of an ℓ_2 regularizer to be a **L2VM** or "ℓ_2-regularized vector machine"; this of course will not be sparse.

We can get even greater sparsity by using ARD/SBL, resulting in a method called the **relevance vector machine** or **RVM** (Tipping 2001). One can fit this model using generic ARD/SBL algorithms, although in practice the most common method is the greedy algorithm in (Tipping and Faul 2003) (this is the algorithm implemented in Mike Tipping's code, which is bundled with PMTK).

Another very popular approach to creating a sparse kernel machine is to use a **support vector machine** or **SVM**. This will be discussed in detail in Section 14.5. Rather than using a sparsity-promoting prior, it essentially modifies the likelihood term, which is rather unnatural from a Bayesian point of view. Nevertheless, the effect is similar, as we will see.

In Figure 14.4, we compare L2VM, L1VM, RVM and an SVM using the same RBF kernel on a binary classification problem in 2d. For simplicity, λ was chosen by hand for L2VM and L1VM; for RVMs, the parameters are estimated using empirical Bayes; and for the SVM, we use CV to pick $C = 1/\lambda$, since SVM performance is very sensitive to this parameter (see Section 14.5.3). We see that all the methods give similar performance. However, RVM is the sparsest (and hence fastest at test time), then L1VM, and then SVM. RVM is also the fastest to train, since CV for an SVM is slow. (This is despite the fact that the RVM code is in Matlab and the SVM code is in C.) This result is fairly typical.

In Figure 14.5, we compare L2VM, L1VM, RVM and an SVM using an RBF kernel on a 1d regression problem. Again, we see that predictions are quite similar, but RVM is the sparsest, then L2VM, then SVM. This is further illustrated in Figure 14.6.

14.4 The kernel trick

Rather than defining our feature vector in terms of kernels, $\boldsymbol{\phi}(\mathbf{x}) = [\kappa(\mathbf{x}, \mathbf{x}_1), \ldots, \kappa(\mathbf{x}, \mathbf{x}_N)]$, we can instead work with the original feature vectors $\mathbf{x}$, but modify the algorithm so that it replaces all inner products of the form $\langle \mathbf{x}, \mathbf{x}' \rangle$ with a call to the kernel function, $\kappa(\mathbf{x}, \mathbf{x}')$. This is called the **kernel trick**. It turns out that many algorithms can be kernelized in this way. We give some examples below. Note that we require that the kernel be a Mercer kernel for this trick to work.

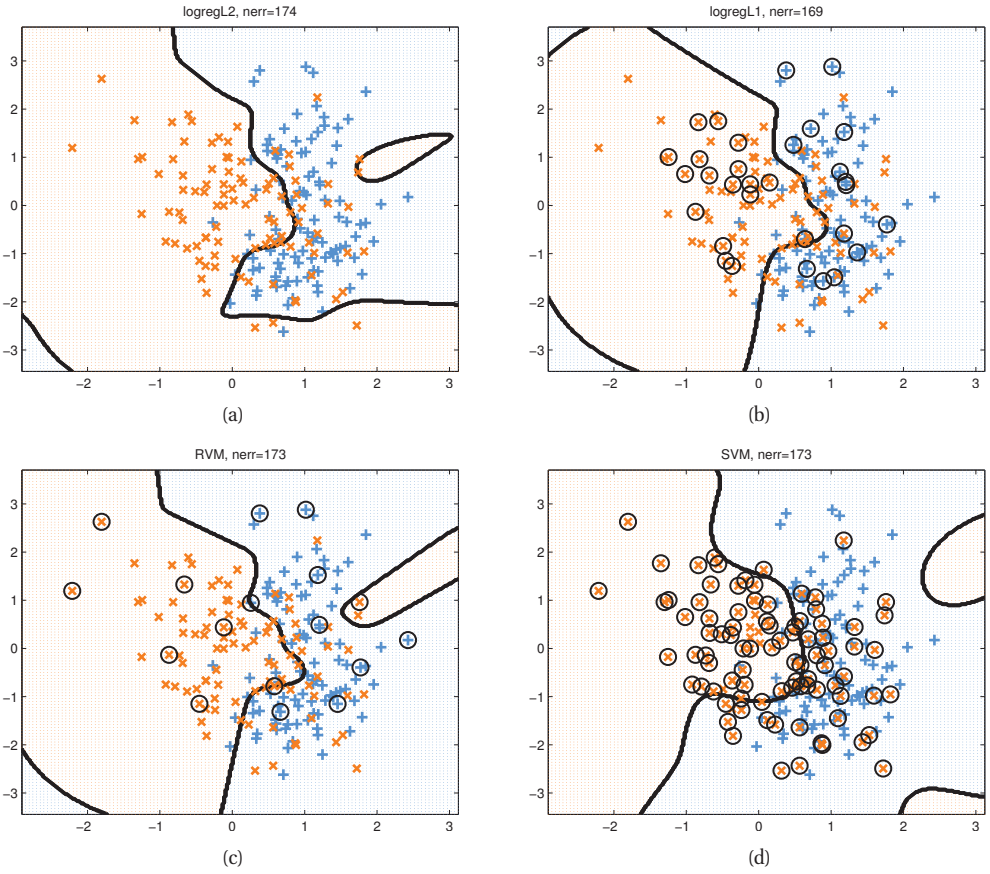


Figure 14.4 Example of non-linear binary classification using an RBF kernel with bandwidth $\sigma = 0.3$. (a) L2VM with $\lambda = 5$. (b) L1VM with $\lambda = 1$. (c) RVM. (d) SVM with $C = 1/\lambda$ chosen by cross validation. Black circles denote the support vectors. Figure generated by `kernelBinaryClassifDemo`.

14.4.1 Kernelized nearest neighbor classification

Recall that in a 1NN classifier (Section 1.4.2), we just need to compute the Euclidean distance of a test vector to all the training points, find the closest one, and look up its label. This can be kernelized by observing that

$$||\mathbf{x}_i - \mathbf{x}_{i'}||_2^2 = \langle \mathbf{x}_i, \mathbf{x}_i \rangle + \langle \mathbf{x}_{i'}, \mathbf{x}_{i'} \rangle - 2\langle \mathbf{x}_i, \mathbf{x}_{i'} \rangle \tag{14.30}$$

This allows us to apply the nearest neighbor classifier to structured data objects.

14.4.2 Kernelized K-medoids clustering

K-means clustering (Section 11.4.2.5) uses Euclidean distance to measure dissimilarity, which is not always appropriate for structured objects. We now describe how to develop a kernelized

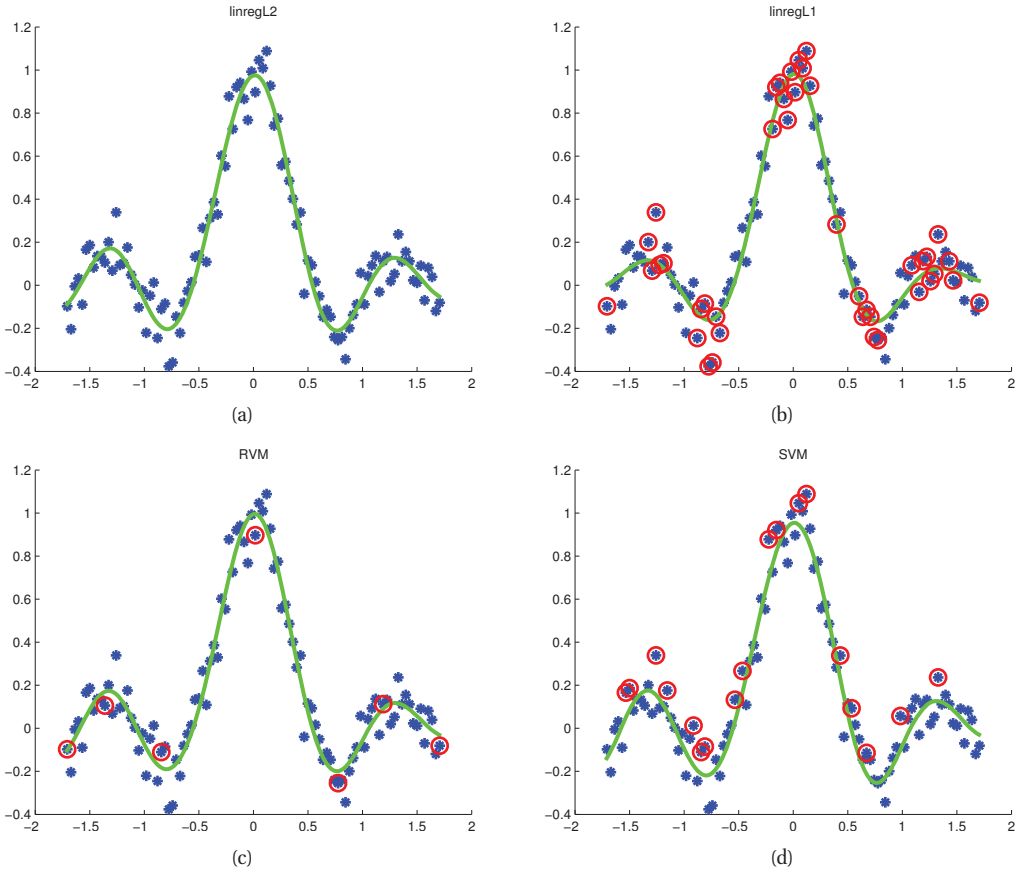


Figure 14.5 Example of kernel based regression on the noisy sinc function using an RBF kernel with bandwidth $\sigma = 0.3$. (a) L2VM with $\lambda = 0.5$. (b) L1VM with $\lambda = 0.5$. (c) RVM. (d) SVM regression with $C = 1/\lambda$ chosen by cross validation, and $\epsilon = 0.1$ (the default for SVMlight). Red circles denote the retained training exemplars. Figure generated by `kernelRegrDemo`.

version of the algorithm.

The first step is to replace the K-means algorithm with the **K-medoids algorothm**. This is similar to K-means, but instead of representing each cluster's centroid by the mean of all data vectors assigned to this cluster, we make each centroid be one of the data vectors themselves. Thus we always deal with integer indexes, rather than data objects. We assign objects to their closest centroids as before. When we update the centroids, we look at each object that belongs to the cluster, and measure the sum of its distances to all the others in the same cluster; we then pick the one which has the smallest such sum:

$$m_k = \underset{i:z_i=k}{\operatorname{argmin}} \sum_{i':z_{i'}=k} d(i,i') \tag{14.31}$$

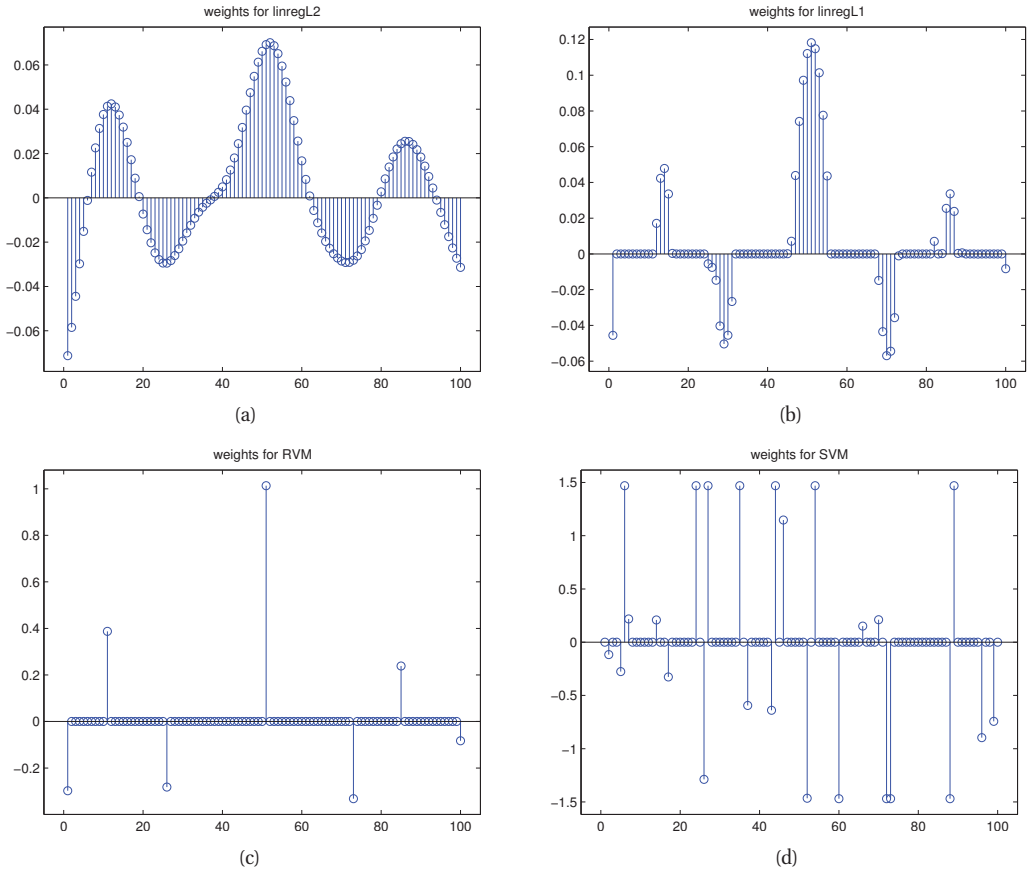


Figure 14.6 Coefficient vectors of length $N = 100$ for the models in Figure 14.6. Figure generated by `kernelRegrDemo`.

where

$$d(i, i') \triangleq ||\mathbf{x}_i - \mathbf{x}_{i'}||_2^2 \tag{14.32}$$

This takes $O(n_k^2)$ work per cluster, whereas K-means takes $O(n_k D)$ to update each cluster. The pseudo-code is given in Algorithm 5. This method can be modified to derive a classifier, by computing the nearest medoid for each class. This is known as **nearest medoid classification** (Hastie et al. 2009, p671).

This algorithm can be kernelized by using Equation 14.30 to replace the distance computation, $d(i, i')$.

Algorithm 14.1: K-medoids algorithm

initialize $m_{1:K}$ as a random subset of size K from $\{1, \ldots, N\}$;
repeat
  $z_i = \text{argmin}_k \, d(i, m_k)$ for $i = 1 : N$;
  $m_k \leftarrow \text{argmin}_{i:z_i=k} \sum_{i':z_{i'}=k} d(i, i')$ for $k = 1 : K$;
until *converged*;

14.4.3 Kernelized ridge regression

Applying the kernel trick to distance-based methods was straightforward. It is not so obvious how to apply it to parametric models such as ridge regression. However, it can be done, as we now explain. This will serve as a good "warm up" for studying SVMs.

14.4.3.1 The primal problem

Let $\mathbf{x} \in \mathbb{R}^D$ be some feature vector, and $\mathbf{X}$ be the corresponding $N \times D$ design matrix. We want to minimize

$$J(\mathbf{w}) = (\mathbf{y} - \mathbf{X}\mathbf{w})^T(\mathbf{y} - \mathbf{X}\mathbf{w}) + \lambda||\mathbf{w}||^2 \tag{14.33}$$

The optimal solution is given by

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}_D)^{-1}\mathbf{X}^T\mathbf{y} = (\sum_i \mathbf{x}_i\mathbf{x}_i^T + \lambda\mathbf{I}_D)^{-1}\mathbf{X}^T\mathbf{y} \tag{14.34}$$

14.4.3.2 The dual problem

Equation 14.34 is not yet in the form of inner products. However, using the matrix inversion lemma (Equation 4.107) we rewrite the ridge estimate as follows

$$\mathbf{w} = \mathbf{X}^T(\mathbf{X}\mathbf{X}^T + \lambda\mathbf{I}_N)^{-1}\mathbf{y} \tag{14.35}$$

which takes $O(N^3 + N^2D)$ time to compute. This can be advantageous if D is large. Furthermore, we see that we can partially kernelize this, by replacing $\mathbf{X}\mathbf{X}^T$ with the Gram matrix $\mathbf{K}$. But what about the leading $\mathbf{X}^T$ term?

Let us define the following **dual variables**:

$$\boldsymbol{\alpha} \triangleq (\mathbf{K} + \lambda\mathbf{I}_N)^{-1}\mathbf{y} \tag{14.36}$$

Then we can rewrite the **primal variables** as follows

$$\mathbf{w} = \mathbf{X}^T\boldsymbol{\alpha} = \sum_{i=1}^N \alpha_i\mathbf{x}_i \tag{14.37}$$

This tells us that the solution vector is just a linear sum of the N training vectors. When we plug this in at test time to compute the predictive mean, we get

$$\hat{f}(\mathbf{x}) = \mathbf{w}^T\mathbf{x} = \sum_{i=1}^N \alpha_i\mathbf{x}_i^T\mathbf{x} = \sum_{i=1}^N \alpha_i\kappa(\mathbf{x}, \mathbf{x}_i) \tag{14.38}$$

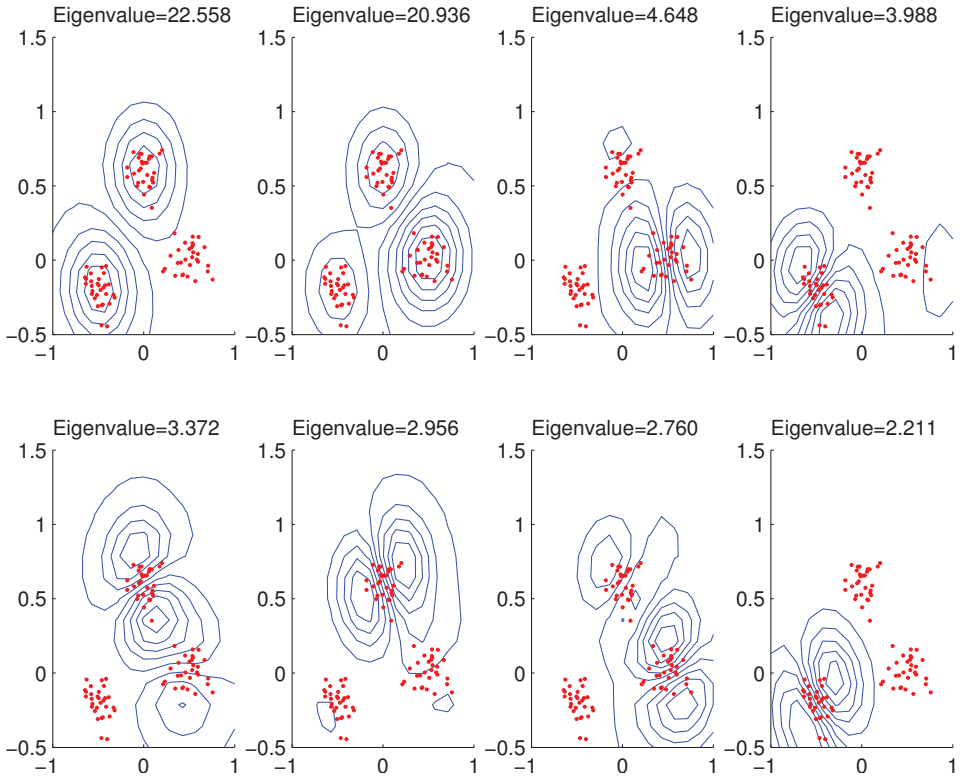


Figure 14.7 Visualization of the first 8 kernel principal component basis functions derived from some 2d data. We use an RBF kernel with $\sigma^2 = 0.1$. Figure generated by `kpcaScholkopf`, written by Bernhard Scholkopf.

So we have succesfully kernelized ridge regression by changing from primal to dual variables. This technique can be applied to many other linear models, such as logistic regression.

14.4.3.3 Computational cost

The cost of computing the dual variables $\boldsymbol{\alpha}$ is $O(N^3)$, whereas the cost of computing the primal variables $\mathbf{w}$ is $O(D^3)$. Hence the kernel method can be useful in high dimensional settings, even if we only use a linear kernel (c.f., the SVD trick in Equation 7.44). However, prediction using the dual variables takes $O(ND)$ time, while prediction using the primal variables only takes $O(D)$ time. We can speedup prediction by making $\boldsymbol{\alpha}$ sparse, as we discuss in Section 14.5.

14.4.4 Kernel PCA

In Section 12.2, we saw how we could compute a low-dimensional linear embedding of some data using PCA. This required finding the eigenvectors of the sample covariance matrix $\mathbf{S} =$

$\frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_i\mathbf{x}_i^T = (1/N)\mathbf{X}^T\mathbf{X}$. However, we can also compute PCA by finding the eigenvectors of the inner product matrix $\mathbf{X}\mathbf{X}^T$, as we show below. This will allow us to produce a nonlinear embedding, using the kernel trick, a method known as **kernel PCA** (Schoelkopf et al. 1998).

First, let $\mathbf{U}$ be an orthogonal matrix containing the eigenvectors of $\mathbf{X}\mathbf{X}^T$ with corresponding eigenvalues in $\mathbf{\Lambda}$. By definition we have $(\mathbf{X}\mathbf{X}^T)\mathbf{U} = \mathbf{U}\mathbf{\Lambda}$. Pre-multiplying by $\mathbf{X}^T$ gives

$$(\mathbf{X}^T\mathbf{X})(\mathbf{X}^T\mathbf{U}) = (\mathbf{X}^T\mathbf{U})\mathbf{\Lambda} \tag{14.39}$$

from which we see that the eigenvectors of $\mathbf{X}^T\mathbf{X}$ (and hence of $\mathbf{S}$) are $\mathbf{V} = \mathbf{X}^T\mathbf{U}$, with eigenvalues given by $\mathbf{\Lambda}$ as before. However, these eigenvectors are not normalized, since $||\mathbf{v}_j||^2 = \mathbf{u}_j^T\mathbf{X}\mathbf{X}^T\mathbf{u}_j = \lambda_j\mathbf{u}_j^T\mathbf{u}_j = \lambda_j$. So the normalized eigenvectors are given by $\mathbf{V}_{pca} = \mathbf{X}^T\mathbf{U}\mathbf{\Lambda}^{-\frac{1}{2}}$. This is a useful trick for regular PCA if $D > N$, since $\mathbf{X}^T\mathbf{X}$ has size $D \times D$, whereas $\mathbf{X}\mathbf{X}^T$ has size $N \times N$. It will also allow us to use the kernel trick, as we now show.

Now let $\mathbf{K} = \mathbf{X}\mathbf{X}^T$ be the Gram matrix. Recall from Mercer's theorem that the use of a kernel implies some underlying feature space, so we are implicitly replacing $\mathbf{x}_i$ with $\boldsymbol{\phi}(\mathbf{x}_i) = \boldsymbol{\phi}_i$. Let $\mathbf{\Phi}$ be the corresponding (notional) design matrix, and $\mathbf{S}_\phi = \frac{1}{N}\sum_i \boldsymbol{\phi}_i\boldsymbol{\phi}_i^T$ be the corresponding (notional) covariance matrix in feature space. The eigenvectors are given by $\mathbf{V}_{kpca} = \mathbf{\Phi}^T\mathbf{U}\mathbf{\Lambda}^{-\frac{1}{2}}$, where $\mathbf{U}$ and $\mathbf{\Lambda}$ contain the eigenvectors and eigenvalues of $\mathbf{K}$. Of course, we can't actually compute $\mathbf{V}_{kpca}$, since $\boldsymbol{\phi}_i$ is potentially infinite dimensional. However, we can compute the projection of a test vector $\mathbf{x}_*$ onto the feature space as follows:

$$\boldsymbol{\phi}_*^T\mathbf{V}_{kpca} = \boldsymbol{\phi}_*^T\mathbf{\Phi}\mathbf{U}\mathbf{\Lambda}^{-\frac{1}{2}} = \mathbf{k}_*^T\mathbf{U}\mathbf{\Lambda}^{-\frac{1}{2}} \tag{14.40}$$

where $\mathbf{k}_* = [\kappa(\mathbf{x}_*, \mathbf{x}_1), \ldots, \kappa(\mathbf{x}_*, \mathbf{x}_N)]$.

There is one final detail to worry about. So far, we have assumed the projected data has zero mean, which is not the case in general. We cannot simply subtract off the mean in feature space. However, there is a trick we can use. Define the centered feature vector as $\tilde{\boldsymbol{\phi}}_i = \boldsymbol{\phi}(\mathbf{x}_i) - \frac{1}{N}\sum_{j=1}^{N}\boldsymbol{\phi}(\mathbf{x}_j)$. The Gram matrix of the centered feature vectors is given by

$$\begin{aligned}
\tilde{K}_{ij} &= \tilde{\boldsymbol{\phi}}_i^T\tilde{\boldsymbol{\phi}}_j && (14.41)\\
&= \boldsymbol{\phi}_i^T\boldsymbol{\phi}_j - \frac{1}{N}\sum_{k=1}^{N}\boldsymbol{\phi}_i^T\boldsymbol{\phi}_k - \frac{1}{N}\sum_{k=1}^{N}\boldsymbol{\phi}_j^T\boldsymbol{\phi}_k + \frac{1}{N^2}\sum_{k=1}^{N}\sum_{l=1}^{M}\boldsymbol{\phi}_k^T\boldsymbol{\phi}_l && (14.42)\\
&= \kappa(\mathbf{x}_i, \mathbf{x}_j) - \frac{1}{N}\sum_{k=1}^{N}\kappa(\mathbf{x}_i, \mathbf{x}_k) - \frac{1}{N}\sum_{k=1}^{N}\kappa(\mathbf{x}_j, \mathbf{x}_k) + \frac{1}{N^2}\sum_{k=1}^{N}\sum_{l=1}^{M}\kappa(\mathbf{x}_k, \mathbf{x}_l) && (14.43)
\end{aligned}$$

This can be expressed in matrix notation as follows:

$$\tilde{\mathbf{K}} = \mathbf{H}\mathbf{K}\mathbf{H} \tag{14.44}$$

where $\mathbf{H} \triangleq \mathbf{I} - \frac{1}{N}\mathbf{1}_N\mathbf{1}_N^T$. is the **centering matrix**. We can convert all this algebra into the pseudocode shown in Algorithm 9.

Whereas linear PCA is limited to using $L \leq D$ components, in kPCA, we can use up to N components, since the rank of $\mathbf{\Phi}$ is $N \times D^*$, where D^* is the (potentially infinite) dimensionality of embedded feature vectors. Figure 14.7 gives an example of the method applied to some $D = 2$ dimensional data using an RBF kernel. We project points in the unit grid onto the first

Algorithm 14.2: Kernel PCA

Input: $\mathbf{K}$ of size $N \times N$, $\mathbf{K}_*$ of size $N_* \times N$, num. latent dimensions L;
$\mathbf{O} = \mathbf{1}_N \mathbf{1}_N^T / N$;
$\tilde{\mathbf{K}} = \mathbf{K} - \mathbf{OK} - \mathbf{KO} + \mathbf{OKO}$;
$[\mathbf{U}, \mathbf{\Lambda}] = \text{eig}(\tilde{\mathbf{K}})$;
for $i = 1 : N$ **do**
  $\mathbf{v}_i = \mathbf{u}_i / \sqrt{\lambda_i}$
$\mathbf{O}_* = \mathbf{1}_{N_*} \mathbf{1}_N^T / N$;
$\tilde{\mathbf{K}}_* = \mathbf{K}_* - \mathbf{O}_* \mathbf{K}_* - \mathbf{K}_* \mathbf{O}_* + \mathbf{O}_* \mathbf{K}_* \mathbf{O}_*$;
$\mathbf{Z} = \tilde{\mathbf{K}}_* \mathbf{V}(:, 1 : L)$

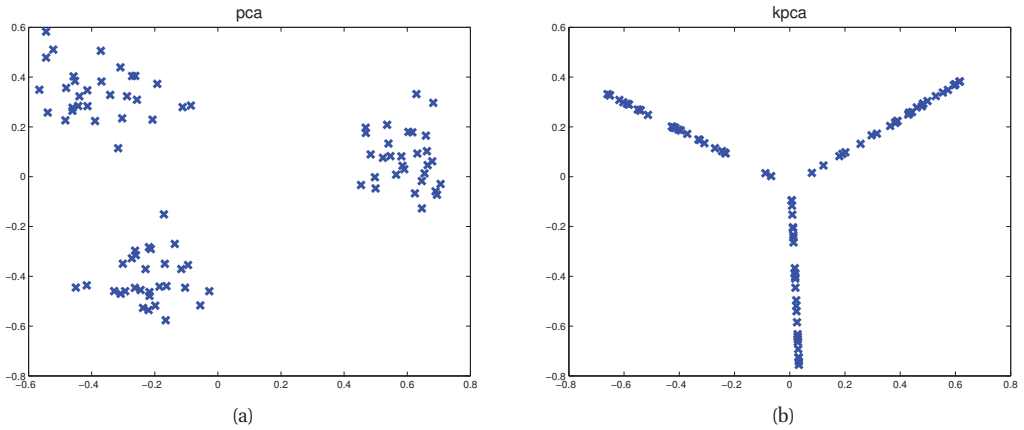
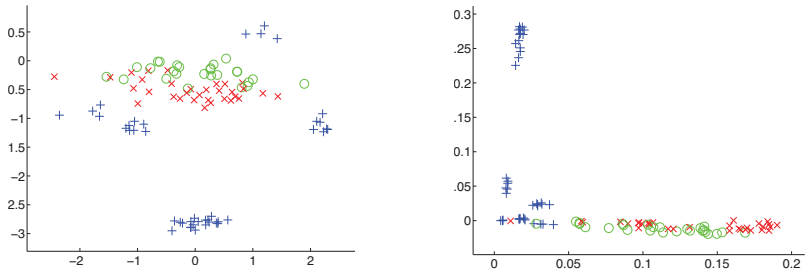


Figure 14.8 2d visualization of some 2d data. (a) PCA projection. (b) Kernel PCA projection. Figure generated by `kpcaDemo2`, based on code by L.J.P. van der Maaten.

8 components and visualize the corresponding surfaces using a contour plot. We see that the first two component separate the three clusters, and following components split the clusters.

Although the features learned by kPCA can be useful for classification (Schoelkopf et al. 1998), they are not necessarily so useful for data visualization. For example, Figure 14.8 shows the projection of the data from Figure 14.7 onto the first 2 principal bases computed using PCA and kPCA. Obviously PCA perfectly represents the data. kPCA represents each cluster by a different line.

Of course, there is no need to project 2d data back into 2d. So let us consider a different data set. We will use a 12 dimensional data set representing the three known phases of flow in an oil pipeline. (This data, which is widely used to compare data visualization methods, is synthetic, and comes from (Bishop and James 1993).) We project this into 2d using PCA and kPCA (with an RBF kernel). The results are shown in Figure 14.9. If we perform nearest neighbor classification in the low-dimensional space, kPCA makes 13 errors and PCA makes 20 (Lawrence



(a) (b)

Figure 14.9 2d representation of 12 dimensional oil flow data. The different colors/symbols represent the 3 phases of oil flow. (a) PCA. (b) Kernel PCA with Gaussian kernel. Compare to Figure 15.10(b). From Figure 1 of (Lawrence 2005). Used with kind permission of Neil Lawrence.

2005). Nevertheless, the kPCA projection is rather unnatural. In Section 15.5, we will discuss how to make kernelized versions of *probabilistic* PCA.

Note that there is a close connection between kernel PCA and a technique known as multidimensional scaling or MDS. This methods finds a low-dimensional embedding such that Euclidean distance in the embedding space approximates the original dissimilarity matrix. See e.g., (Williams 2002) for details.

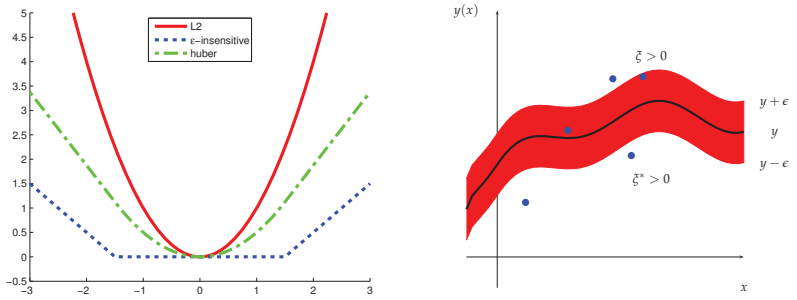
14.5 Support vector machines (SVMs)

In Section 14.3.2, we saw one way to derive a sparse kernel machine, namely by using a GLM with kernel basis functions, plus a sparsity-promoting prior such as ℓ_1 or ARD. An alternative approach is to change the objective function from negative log likelihood to some other loss function, as we discussed in Section 6.5.5. In particular, consider the ℓ_2 regularized empirical risk function

$$J(\mathbf{w}, \lambda) = \sum_{i=1}^{N} L(y_i, \hat{y}_i) + \lambda ||\mathbf{w}||^2 \tag{14.45}$$

where $\hat{y}_i = \mathbf{w}^T \mathbf{x}_i + w_0$. (So far this is in the original feature space; we introduce kernels in a moment.) If L is quadratic loss, this is equivalent to ridge regression, and if L is the log-loss defined in Equation 6.73, this is equivalent to logistic regression.

In the ridge regression case, we know that the solution to this has the form $\hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$, and plug-in predictions take the form $\hat{w}_0 + \hat{\mathbf{w}}^T\mathbf{x}$. As we saw in Section 14.4.3, we can rewrite these equations in a way that only involves inner products of the form $\mathbf{x}^T\mathbf{x}'$, which we can replace by calls to a kernel function, $\kappa(\mathbf{x}, \mathbf{x}')$. This is kernelized, but not sparse. However, if we replace the quadratic/ log-loss with some other loss function, to be explained below, we can ensure that the solution is sparse, so that predictions only depend on a subset of the training data, known as **support vectors**. This combination of the kernel trick plus a modified loss function is known as a **support vector machine** or **SVM**. This technique was



(a) (b)

Figure 14.10 (a) Illustration of ℓ_2, Huber and ϵ-insensitive loss functions, where $\epsilon = 1.5$. Figure generated by `huberLossDemo`. (b) Illustration of the ϵ-tube used in SVM regression. Points above the tube have $\xi_i > 0$ and $\xi_i^* = 0$. Points below the tube have $\xi_i = 0$ and $\xi_i^* > 0$. Points inside the tube have $\xi_i = \xi_i^* = 0$. Based on Figure 7.7 of (Bishop 2006a).

originally designed for binary classification, but can be extended to regression and multi-class classification as we explain below.

Note that SVMs are very unnatural from a probabilistic point of view. First, they encode sparsity in the loss function rather than the prior. Second, they encode kernels by using an algorithmic trick, rather than being an explicit part of the model. Finally, SVMs do not result in probabilistic outputs, which causes various difficulties, especially in the multi-class classification setting (see Section 14.5.2.4 for details).

It is possible to obtain sparse, probabilistic, multi-class kernel-based classifiers, which work as well or better than SVMs, using techniques such as the LIVM or RVM, discussed in Section 14.3.2. However, we include a discussion of SVMs, despite their non-probabilistic nature, for two main reasons. First, they are very popular and widely used, so all students of machine learning should know about them. Second, they have some computational advantages over probabilistic methods in the structured output case; see Section 19.7.

14.5.1 SVMs for regression

The problem with kernelized ridge regression is that the solution vector $\mathbf{w}$ depends on all the training inputs. We now seek a method to produce a sparse estimate.

Vapnik (Vapnik et al. 1997) proposed a variant of the Huber loss function (Section 7.4) called the **epsilon insensitive loss function**, defined by

$$L_\epsilon(y, \hat{y}) \triangleq \begin{cases} 0 & \text{if } |y - \hat{y}| < \epsilon \\ |y - \hat{y}| - \epsilon & \text{otherwise} \end{cases} \tag{14.46}$$

This means that any point lying inside an ϵ-**tube** around the prediction is not penalized, as in Figure 14.10.

The corresponding objective function is usually written in the following form

$$J = C \sum_{i=1}^{N} L_\epsilon(y_i, \hat{y}_i) + \frac{1}{2}||\mathbf{w}||^2 \tag{14.47}$$

where $\hat{y}_i = f(\mathbf{x}_i) = \mathbf{w}^T\mathbf{x}_i + w_0$ and $C = 1/\lambda$ is a regularization constant. This objective is convex and unconstrained, but not differentiable, because of the absolute value function in the loss term. As in Section 13.4, where we discussed the lasso problem, there are several possible algorithms we could use. One popular approach is to formulate the problem as a constrained optimization problem. In particular, we introduce **slack variables** to represent the degree to which each point lies outside the tube:

$$y_i \leq f(\mathbf{x}_i) + \epsilon + \xi_i^+ \tag{14.48}$$

$$y_i \geq f(\mathbf{x}_i) - \epsilon - \xi_i^- \tag{14.49}$$

Given this, we can rewrite the objective as follows:

$$J = C\sum_{i=1}^{N}(\xi_i^+ + \xi_i^-) + \frac{1}{2}||\mathbf{w}||^2 \tag{14.50}$$

This is a quadratic function of $\mathbf{w}$, and must be minimized subject to the linear constraints in Equations 14.48-14.49, as well as the positivity constraints $\xi_i^+ \geq 0$ and $\xi_i^- \geq 0$. This is a standard quadratic program in $2N + D + 1$ variables.

One can show (see e.g., (Schoelkopf and Smola 2002)) that the optimal solution has the form

$$\hat{\mathbf{w}} = \sum_i \alpha_i \mathbf{x}_i \tag{14.51}$$

where $\alpha_i \geq 0$. Furthermore, it turns out that the $\boldsymbol{\alpha}$ vector is sparse, because we don't care about errors which are smaller than ϵ. The $\mathbf{x}_i$ for which $\alpha_i > 0$ are called the **support vectors**; thse are points for which the errors lie on or outside the ϵ tube.

Once the model is trained, we can then make predictions using

$$\hat{y}(\mathbf{x}) = \hat{w}_0 + \hat{\mathbf{w}}^T\mathbf{x} \tag{14.52}$$

Plugging in the definition of $\hat{\mathbf{w}}$ we get

$$\hat{y}(\mathbf{x}) = \hat{w}_0 + \sum_i \alpha_i \mathbf{x}_i^T\mathbf{x} \tag{14.53}$$

Finally, we can replace $\mathbf{x}_i^T\mathbf{x}$ with $\kappa(\mathbf{x}_i, \mathbf{x})$ to get a kernelized solution:

$$\hat{y}(\mathbf{x}) = \hat{w}_0 + \sum_i \alpha_i \kappa(\mathbf{x}_i, \mathbf{x}) \tag{14.54}$$

14.5.2 SVMs for classification

We now discuss how to apply SVMs to classification. We first focus on the binary case, and then discuss the multi-class case in Section 14.5.2.4.

14.5.2.1 Hinge loss

In Section 6.5.5, we showed that the negative log likelihood of a logistic regression model,

$$L_{\text{nll}}(y, \eta) = -\log p(y|\mathbf{x}, \mathbf{w}) = \log(1 + e^{-y\eta}) \tag{14.55}$$

was a convex upper bound on the 0-1 risk of a binary classifier, where $\eta = f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0$ is the log odds ratio, and we have assumed the labels are $y \in \{1, -1\}$ rather than $\{0, 1\}$. In this section, we replace the NLL loss with the **hinge loss**, defined as

$$L_{\text{hinge}}(y, \eta) = \max(0, 1 - y\eta) = (1 - y\eta)_+ \tag{14.56}$$

Here $\eta = f(\mathbf{x})$ is our "confidence" in choosing label $y = 1$; however, it need not have any probabilistic semantics. See Figure 6.7 for a plot. We see that the function looks like a door hinge, hence its name. The overall objective has the form

$$\min_{\mathbf{w}, w_0} \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{N}(1 - y_i f(\mathbf{x}_i))_+ \tag{14.57}$$

Once again, this is non-differentiable, because of the max term. However, by introducing slack variables ξ_i, one can show that this is equivalent to solving

$$\min_{\mathbf{w}, w_0, \boldsymbol{\xi}} \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{N}\xi_i \quad \text{s.t.} \quad \xi_i \geq 0, \;\; y_i(\mathbf{x}_i^T\mathbf{w} + w_0) \geq 1 - \xi_i, i = 1 : N \tag{14.58}$$

This is a quadratic program in $N + D + 1$ variables, subjet to $O(N)$ constraints. We can eliminate the primal variables $\mathbf{w}$, w_0 and ξ_i, and just solve the N dual variables, which correspond to the Lagrange multipliers for the constraints. Standard solvers take $O(N^3)$ time. However, specialized algorithms, which avoid the use of generic QP solvers, have been developed for this problem, such as the **sequential minimal optimization** or **SMO** algorithm (Platt 1998). In practice this can take $O(N^2)$. However, even this can be too slow if N is large. In such settings, it is common to use linear SVMs, which take $O(N)$ time to train (Joachims 2006; Bottou et al. 2007).

One can show that the solution has the form

$$\hat{\mathbf{w}} = \sum_i \alpha_i \mathbf{x}_i \tag{14.59}$$

where $\alpha_i = \lambda_i y_i$ and where $\boldsymbol{\alpha}$ is sparse (because of the hinge loss). The $\mathbf{x}_i$ for which $\alpha_i > 0$ are called support vectors; these are points which are either incorrectly classified, or are classified correctly but are on or inside the margin (we disuss margins below). See Figure 14.12(b) for an illustration.

At test time, prediction is done using

$$\hat{y}(\mathbf{x}) = \text{sgn}(f(\mathbf{x})) = \text{sgn}\left(\hat{w}_0 + \hat{\mathbf{w}}^T\mathbf{x}\right) \tag{14.60}$$

Using Equation 14.59 and the kernel trick we have

$$\hat{y}(\mathbf{x}) = \text{sgn}\left(\hat{w}_0 + \sum_{i=1}^{N}\alpha_i\kappa(\mathbf{x}_i, \mathbf{x})\right) \tag{14.61}$$

This takes $O(sD)$ time to compute, where $s \leq N$ is the number of support vectors. This depends on the sparsity level, and hence on the regularizer C.

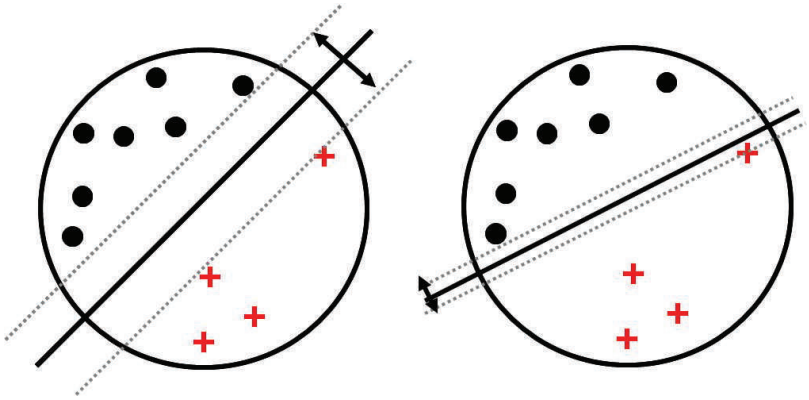


Figure 14.11 Illustration of the large margin principle. Left: a separating hyper-plane with large margin. Right: a separating hyper-plane with small margin.

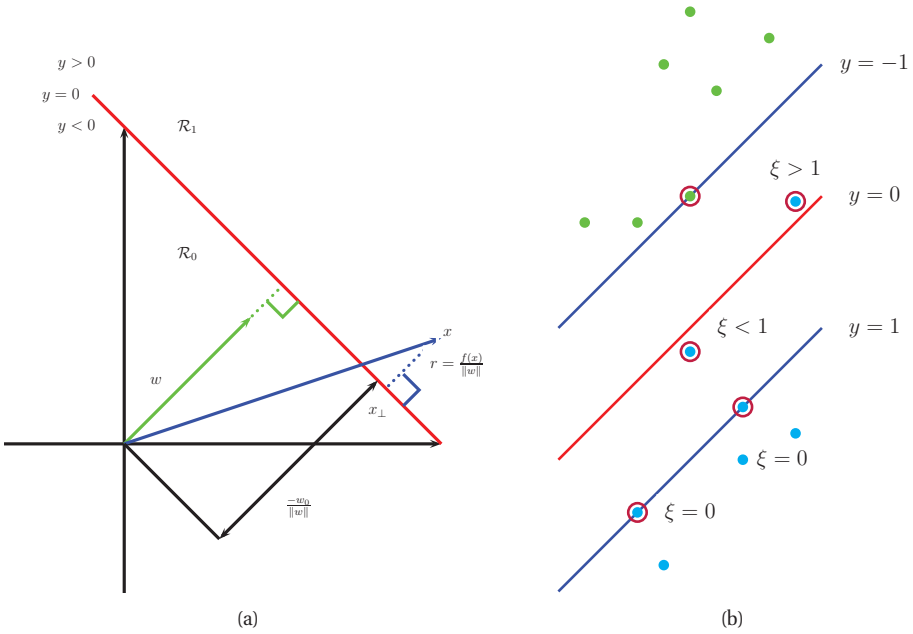


Figure 14.12 (a) Illustration of the geometry of a linear decision boundary in 2d. A point $\mathbf{x}$ is classified as belonging in decision region $\mathcal{R}_1$ if $f(\mathbf{x}) > 0$, otherwise it belongs in decision region $\mathcal{R}_2$; here $f(\mathbf{x})$ is known as a **discriminant function**. The decision boundary is the set of points such that $f(\mathbf{x}) = 0$. $\mathbf{w}$ is a vector which is perpendicular to the decision boundary. The term w_0 controls the distance of the decision boundary from the origin. The signed distance of $\mathbf{x}$ from its orthogonal projection onto the decision boundary, $\mathbf{x}_\perp$, is given by $f(\mathbf{x})/||\mathbf{w}||$. Based on Figure 4.1 of (Bishop 2006a). (b) Illustration of the soft margin principle. Points with circles around them are support vectors. We also indicate the value of the corresponding slack variables. Based on Figure 7.3 of (Bishop 2006a).

14.5.2.2 The large margin principle

In this section, we derive Equation 14.58 form a completely different perspective. Recall that our goal is to derive a discriminant function $f(\mathbf{x})$ which will be linear in the feature space implied by the choice of kernel. Consider a point $\mathbf{x}$ in this induced space. Referring to Figure 14.12(a), we see that

$$\mathbf{x} = \mathbf{x}_\perp + r\frac{\mathbf{w}}{||\mathbf{w}||} \tag{14.62}$$

where r is the distance of $\mathbf{x}$ from the decision boundary whose normal vector is $\mathbf{w}$, and $\mathbf{x}_\perp$ is the orthogonal projection of $\mathbf{x}$ onto this boundary. Hence

$$f(\mathbf{x}) \quad = \quad \mathbf{w}^T\mathbf{x} + w_0 = (\mathbf{w}^T\mathbf{x}_\perp + w_0) + r\frac{\mathbf{w}^T\mathbf{w}}{||\mathbf{w}||} \tag{14.63}$$

Now $f(\mathbf{x}_\perp) = 0$ so $0 = \mathbf{w}^T\mathbf{x}_\perp + w_0$. Hence $f(\mathbf{x}) = r\frac{\mathbf{w}^T\mathbf{w}}{\sqrt{\mathbf{w}^T\mathbf{w}}}$, and $r = \frac{f(\mathbf{x})}{||\mathbf{w}||}$.

We would like to make this distance $r = f(\mathbf{x})/||\mathbf{w}||$ as large as possible, for reasons illustrated in Figure 14.11. In particular, there might be many lines that perfectly separate the training data (especially if we work in a high dimensional feature space), but intuitively, the best one to pick is the one that maximizes the margin, i.e., the perpendicular distance to the closest point. In addition, we want to ensure each point is on the correct side of the boundary, hence we want $f(\mathbf{x}_i)y_i > 0$. So our objective becomes

$$\max_{\mathbf{w},w_0} \min_{i=1}^{N} \frac{y_i(\mathbf{w}^T\mathbf{x}_i + w_0)}{||\mathbf{w}||} \tag{14.64}$$

Note that by rescaling the parameters using $\mathbf{w} \to k\mathbf{w}$ and $w_0 \to kw_0$, we do not change the distance of any point to the boundary, since the k factor cancels out when we divide by $||\mathbf{w}||$. Therefore let us define the scale factor such that $y_i f_i = 1$ for the point that is closest to the decision boundary. We therefore want to optimize

$$\min_{\mathbf{w},w_0} \frac{1}{2}||\mathbf{w}||^2 \quad \text{s.t.} \quad y_i(\mathbf{w}^T\mathbf{x}_i + w_0) \geq 1, i = 1:N \tag{14.65}$$

(The fact of $\frac{1}{2}$ is added for convenience and doesn't affect the optimal parameters.) The constraint says that we want all points to be on the correct side of the decision boundary with a margin of at least 1. For this reason, we say that an SVM is an example of a **large margin classifier**.

If the data is not linearly separable (even after using the kernel trick), there will be no feasible solution in which $y_i f_i \geq 1$ for all i. We therefore introduce slack variables $\xi_i \geq 0$ such that $\xi_i = 0$ if the point is on or inside the correct margin boundary, and $\xi_i = |y_i - f_i|$ otherwise. If $0 < \xi_i \leq 1$ the point lies inside the margin, but on the correct side of the decision boundary. If $\xi_i > 1$, the point lies on the wrong side of the decision boundary. See Figure 14.12(b).

We replace the hard constraints that $y_i f_i \geq 0$ with the **soft margin constraints** that $y_i f_i \geq 1 - \xi_i$. The new objective becomes

$$\min_{\mathbf{w},w_0,\boldsymbol{\xi}} \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{N}\xi_i \quad \text{s.t.} \quad \xi_i \geq 0, \;\; y_i(\mathbf{x}_i^T\mathbf{w} + w_0) \geq 1 - \xi_i \tag{14.66}$$

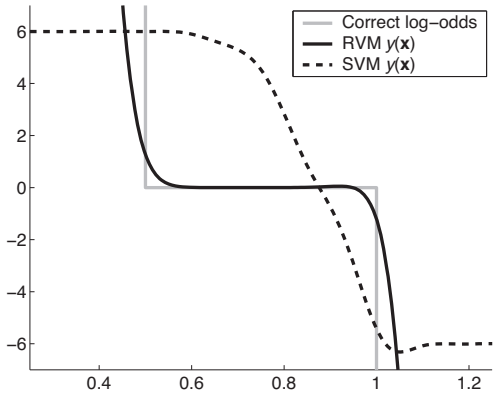


Figure 14.13 Log-odds vs x for 3 different methods. Based on Figure 10 of (Tipping 2001). Used with kind permission of Mike Tipping.

which is the same as Equation 14.58. Since $\xi_i > 1$ means point i is misclassified, we can interpret $\sum_i \xi_i$ as an upper bound on the number of misclassified points.

The parameter C is a regularization parameter that controls the number of errors we are willing to tolerate on the training set. It is common to define this using $C = 1/(\nu N)$, where $0 < \nu \leq 1$ controls the fraction of misclassified points that we allow during the training phase. This is called a **ν-SVM classifier**. This is usually set using cross-validation (see Section 14.5.3).

14.5.2.3 Probabilistic output

An SVM classifier produces a hard-labeling, $\hat{y}(\mathbf{x}) = \text{sign}(f(\mathbf{x}))$. However, we often want a measure of confidence in our prediction. One heuristic approach is to interpret $f(\mathbf{x})$ as the log-odds ratio, $\log \frac{p(y=1|\mathbf{x})}{p(y=0|\mathbf{x})}$. We can then convert the output of an SVM to a probability using

$$p(y = 1|\mathbf{x}, \boldsymbol{\theta}) = \sigma(af(\mathbf{x}) + b) \tag{14.67}$$

where a, b can be estimated by maximum likelihood on a separate validation set. (Using the training set to estimate a and b leads to severe overfitting.) This technique was first proposed in (Platt 2000).

However, the resulting probabilities are not particularly well calibrated, since there is nothing in the SVM training procedure that justifies interpreting $f(\mathbf{x})$ as a log-odds ratio. To illustrate this, consider an example from (Tipping 2001). Suppose we have 1d data where $p(x|y = 0) = \text{Unif}(0, 1)$ and $p(x|y = 1) = \text{Unif}(0.5, 1.5)$. Since the class-conditional distributions overlap in the middle, the log-odds of class 1 over class 0 should be zero in $[0.5, 1.0]$, and infinite outside this region. We sampled 1000 points from the model, and then fit an RVM and an SVM with a Gaussian kenel of width 0.1. Both models can perfectly capture the decision boundary, and achieve a generalizaton error of 25%, which is Bayes optimal in this problem. The probabilistic output from the RVM is a good approximation to the true log-odds, but this is not the case for the SVM, as shown in Figure 14.13.

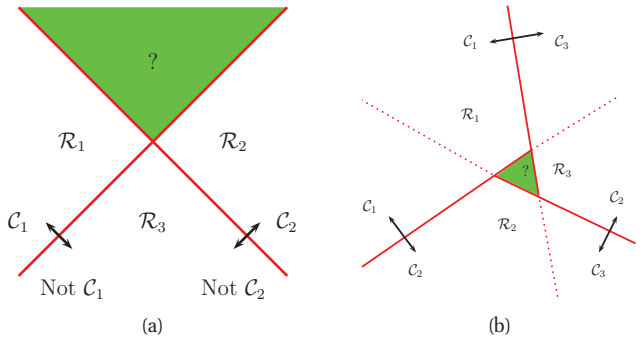


Figure 14.14 (a) The one-versus-rest approach. The green region is predicted to be both class 1 and class 2. (b) The one-versus-one approach. The label of the green region is ambiguous. Based on Figure 4.2 of (Bishop 2006a).

14.5.2.4 SVMs for multi-class classification

In Section 8.3.7, we saw how we could "upgrade" a binary logistic regression model to the multi-class case, by replacing the sigmoid function with the softmax, and the Bernoulli distribution with the multinomial. Upgrading an SVM to the multi-class case is not so easy, since the outputs are not on a calibrated scale and hence are hard to compare to each other.

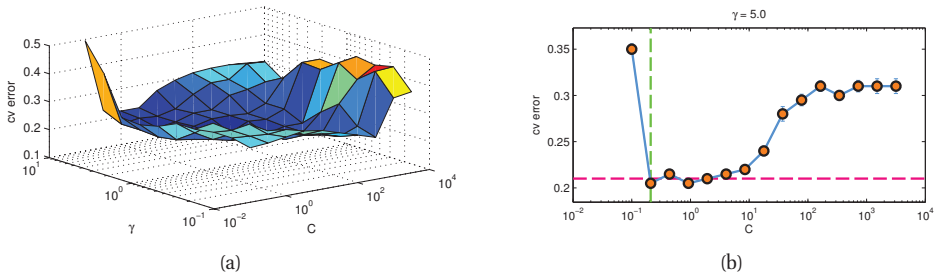
The obvious approach is to use a **one-versus-the-rest** approach (also called **one-vs-all**), in which we train C binary classifiers, $f_c(\mathbf{x})$, where the data from class c is treated as positive, and the data from all the other classes is treated as negative. However, this can result in regions of input space which are ambiguously labeled, as shown in Figure 14.14(a).

A common alternative is to pick $\hat{y}(\mathbf{x}) = \arg\max_c f_c(\mathbf{x})$. However, this technique may not work either, since there is no guarantee that the different f_c functions have comparable magnitudes. In addition, each binary subproblem is likely to suffer from the **class imbalance** problem. To see this, suppose we have 10 equally represented classes. When training f_1, we will have 10% positive examples and 90% negative examples, which can hurt performance. It is possible to devise ways to train all C classifiers simultaneously (Weston and Watkins 1999), but the resulting method takes $O(C^2N^2)$ time, instead of the usual $O(CN^2)$ time.

Another approach is to use the **one-versus-one** or OVO approach, also called **all pairs**, in which we train $C(C-1)/2$ classifiers to discriminate all pairs $f_{c,c'}$. We then classify a point into the class which has the highest number of votes. However, this can also result in ambiguities, as shown in Figure 14.14(b). Also, it takes $O(C^2N^2)$ time to train and $O(C^2N_{sv})$ to test each data point, where N_{sv} is the number of support vectors.[2] See also (Allwein et al. 2000) for an approach based on error-correcting output codes.

It is worth remembering that all of these difficulties, and the plethora of heuristics that have been proposed to fix them, fundamentally arise because SVMs do not model uncertainty using probabilities, so their output scores are not comparable across classes.

2. We can reduce the test time by structuring the classes into a DAG (directed acyclic graph), and performing $O(C)$ pairwise comparisons (Platt et al. 2000). However, the $O(C^2)$ factor in the training time is unavoidable.



(a) (b)

Figure 14.15 (a) A cross validation estimate of the 0-1 error for an SVM classifier with RBF kernel with different precisions $\gamma = 1/(2\sigma^2)$ and different regularizer $\lambda = 1/C$, applied to a synthetic data set drawn from a mixture of 2 Gaussians. (b) A slice through this surface for $\gamma = 5$ The red dotted line is the Bayes optimal error, computed using Bayes rule applied to the model used to generate the data. Based on Figure 12.6 of (Hastie et al. 2009). Figure generated by `svmCgammaDemo`.

14.5.3 Choosing C

SVMs for both classification and regression require that you specify the kernel function and the parameter C. Typically C is chosen by cross-validation. Note, however, that C interacts quite strongly with the kernel parameters. For example, suppose we are using an RBF kernel with precision $\gamma = \frac{1}{2\sigma^2}$. If $\gamma = 5$, corresponding to narrow kernels, we need heavy regularization, and hence small C (so $\lambda = 1/C$ is big). If $\gamma = 1$, a larger value of C should be used. So we see that γ and C are tightly coupled. This is illustrated in Figure 14.15, which shows the CV estimate of the 0-1 risk as a function of C and γ.

The authors of libsvm recommend (Hsu et al. 2009) using CV over a 2d grid with values $C \in \{2^{-5}, 2^{-3}, \ldots, 2^{15}\}$ and $\gamma \in \{2^{-15}, 2^{-13}, \ldots, 2^{3}\}$. In addition, it is important to standardize the data first, for a spherical Gaussian kernel to make sense.

To choose C efficiently, one can develop a path following algorithm in the spirit of lars (Section 13.3.4). The basic idea is to start with λ large, so that the margin $1/||\mathbf{w}(\lambda)||$ is wide, and hence all points are inside of it and have $\alpha_i = 1$. By slowly decreasing λ, a small set of points will move from inside the margin to outside, and their α_i values will change from 1 to 0, as they cease to be support vectors. When λ is maximal, the function is completely smoothed, and no support vectors remain. See (Hastie et al. 2004) for the details.

14.5.4 Summary of key points

Summarizing the above discussion, we recognize that SVM classifiers involve three key ingredients: the kernel trick, sparsity, and the large margin principle. The kernel trick is necessary to prevent underfitting, i.e., to ensure that the feature vector is sufficiently rich that a linear classifier can separate the data. (Recall from Section 14.2.3 that any Mercer kernel can be viewed as implicitly defining a potentially high dimensional feature vector.) If the original features are already high dimensional (as in many gene expression and text classification problems), it suffices to use a linear kernel, $\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T\mathbf{x}'$, which is equivalent to working with the original features.

Method	Opt. $\mathbf{w}$	Opt. kernel	Sparse	Prob.	Multiclass	Non-Mercer	Section
L2VM	Convex	EB	No	Yes	Yes	Yes	14.3.2
L1VM	Convex	CV	Yes	Yes	Yes	Yes	14.3.2
RVM	Not convex	EB	Yes	Yes	Yes	Yes	14.3.2
SVM	Convex	CV	Yes	No	Indirectly	No	14.5
GP	N/A	EB	No	Yes	Yes	No	15

Table 14.1 Comparison of various kernel based classifiers. EB = empirical Bayes, CV = cross validation. See text for details.

The sparsity and large margin principles are necessary to prevent overfitting, i.e., to ensure that we do not use all the basis functions. These two ideas are closely related to each other, and both arise (in this case) from the use of the hinge loss function. However, there are other methods of achieving sparsity (such as ℓ_1), and also other methods of maximizing the margin (such as boosting). A deeper discussion of this point takes us outside of the scope of this book. See e.g., (Hastie et al. 2009) for more information.

14.5.5 A probabilistic interpretation of SVMs

In Section 14.3, we saw how to use kernels inside GLMs to derive probabilistic classifiers, such as the L1VM and RVM. And in Section 15.3, we will discuss Gaussian process classifiers, which also use kernels. However, all of these approaches use a logistic or probit likelihood, as opposed to the hinge loss used by SVMs. It is natural to wonder if one can interpret the SVM more directly as a probabilistic model. To do so, we must interpret $Cg(m)$ as a negative log likelihood, where $g(m) = (1-m)_+$, where $m = yf(\mathbf{x})$ is the margin. Hence $p(y=1|f) = \exp(-Cg(f))$ and $p(y=-1|f) = \exp(-Cg(-f))$. By summing over both values of y, we require that $\exp(-Cg(f)) + \exp(-Cg(-f))$ be a constant independent of f. But it turns out this is not possible for any $C > 0$ (Sollich 2002).

However, if we are willing to relax the sum-to-one condition, and work with a pseudo-likelihood, we *can* derive a probabilistic interpretation of the hinge loss (Polson and Scott 2011). In particular, one can show that

$$\exp(-2(1-y_i\mathbf{x}_i^T\mathbf{w})_+) = \int_0^\infty \frac{1}{\sqrt{2\pi\lambda_i}} \exp\left(-\frac{1}{2}\frac{(1+\lambda_i - y_i\mathbf{x}_i^T\mathbf{w})^2}{\lambda_i}\right) d\lambda_i \tag{14.68}$$

Thus the exponential of the negative hinge loss can be represented as a Gaussian scale mixture. This allows one to fit an SVM using EM or Gibbs sampling, where λ_i are the latent variables. This in turn opens the door to Bayesian methods for setting the hyper-parameters for the prior on $\mathbf{w}$. See (Polson and Scott 2011) for details. (See also (Franc et al. 2011) for a different probabilistic interpretation of SVMs.)

14.6 Comparison of discriminative kernel methods

We have mentioned several different methods for classification and regression based on kernels, which we summarize in Table 14.1. (GP stands for "Gaussian process", which we discuss in Chapter 15.) The columns have the following meaning:

- Optimize $\mathbf{w}$: a key question is whether the objective $J(\mathbf{w}) = -\log p(\mathcal{D}|\mathbf{w}) - \log p(\mathbf{w})$ is convex or not. L2VM, L1VM and SVMs have convex objectives. RVMs do not. GPs are Bayesian methods that do not perform parameter estimation.
- Optimize kernel: all the methods require that one "tune" the kernel parameters, such as the bandwidth of the RBF kernel, as well as the level of regularization. For methods based on Gaussians, including L2VM, RVMs and GPs, we can use efficient gradient based optimizers to maximize the marginal likelihood. For SVMs, and L1VM, we must use cross validation, which is slower (see Section 14.5.3).
- Sparse: L1VM, RVMs and SVMs are sparse kernel methods, in that they only use a subset of the training examples. GPs and L2VM are not sparse: they use all the training examples. The principle advantage of sparsity is that prediction at test time is usually faster. In addition, one can sometimes get improved accuracy.
- Probabilistic: All the methods except for SVMs produce probabilistic output of the form $p(y|\mathbf{x})$. SVMs produce a "confidence" value that can be converted to a probability, but such probabilities are usually very poorly calibrated (see Section 14.5.2.3).
- Multiclass: All the methods except for SVMs naturally work in the multiclass setting, by using a multinoulli output instead of Bernoulli. The SVM can be made into a multiclass classifier, but there are various difficulties with this approach, as discussed in Section 14.5.2.4.
- Mercer kernel: SVMs and GPs require that the kernel is positive definite; the other techniques do not.

Apart from these differences, there is the natural question: which method works best? In a small experiment[3], we found that all of these methods had similar accuracy when averaged over a range of problems, provided they have the same kernel, and provided the regularization constants are chosen appropriately.

Given that the statistical performance is roughly the same, what about the computational performance? GPs and L2VM are generally the slowest, taking $O(N^3)$ time, since they don't exploit sparsity (although various speedups are possible, see Section 15.6). SVMs also take $O(N^3)$ time to train (unless we use a linear kernel, in which case we only need $O(N)$ time (Joachims 2006)). However, the need to use cross validation can make SVMs slower than RVMs. L1VM should be faster than an RVM, since an RVM requires multiple rounds of ℓ_1 minimization (see Section 13.7.4.3). However, in practice it is common to use a greedy method to train RVMs, which is faster than ℓ_1 minimization. This is reflected in our empirical results.

The conclusion of all this is as follows: if speed matters, use an RVM, but if well-calibrated probabilistic output matters (e.g., for active learning or control problems), use a GP. The only circumstances under which using an SVM seems sensible is the structured output case, where likelihood-based methods can be slow. (We attribute the enormous popularity of SVMs not to their superiority, but to ignorance of the alternatives, and also to the lack of high quality software implementing the alternatives.)

Section 16.7.1 gives a more extensive experimental comparison of supervised learning methods, including SVMs and various non kernel methods.

3. See `http://pmtk3.googlecode.com/svn/trunk/docs/tutorial/html/tutKernelClassif.html`.

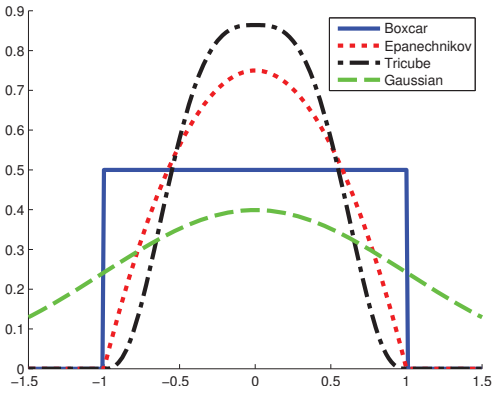


Figure 14.16 A comparison of some popular smoothing kernels. The boxcar kernel has compact support but is not smooth. The Epanechnikov kernel has compact support but is not differentiable at its boundary. The tri-cube has compact support and two continuous derivatives at the boundary of its support. The Gaussian is differentiable, but does not have compact support. Based on Figure 6.2 of (Hastie et al. 2009). Figure generated by `smoothingKernelPlot`.

14.7 Kernels for building generative models

There is a different kind of kernel known as a smoothing kernel which can be used to create non-parametric density estimates. This can be used for unsupervised density estimation, $p(\mathbf{x})$, as well as for creating generative models for classification and regression by making models of the form $p(y, \mathbf{x})$.

14.7.1 Smoothing kernels

A **smoothing kernel** is a function of one argument which satisfies the following properties:

$$\int \kappa(x)dx = 1, \int x\kappa(x)dx = 0, \int x^2\kappa(x)dx > 0 \tag{14.69}$$

A simple example is the **Gaussian kernel**,

$$\kappa(x) \triangleq \frac{1}{(2\pi)^{\frac{1}{2}}} e^{-x^2/2} \tag{14.70}$$

We can control the width of the kernel by introducing a **bandwidth** parameter h:

$$\kappa_h(x) \triangleq \frac{1}{h}\kappa(\frac{x}{h}) \tag{14.71}$$

We can generalize to vector valued inputs by defining an RBF kernel:

$$\kappa_h(\mathbf{x}) = \kappa_h(||\mathbf{x}||) \tag{14.72}$$

In the case of the Gaussian kernel, this becomes

$$\kappa_h(\mathbf{x}) = \frac{1}{h^D(2\pi)^{D/2}} \prod_{j=1}^{D} \exp(-\frac{1}{2h^2}x_j^2) \tag{14.73}$$

Although Gaussian kernels are popular, they have unbounded support. An alternative kernel, with compact support, is the **Epanechnikov kernel**, defined by

$$\kappa(x) \triangleq \frac{3}{4}(1-x^2)\mathbb{I}(|x| \leq 1) \tag{14.74}$$

This is plotted in Figure 14.16. Compact support can be useful for efficiency reasons, since one can use fast nearest neighbor methods to evaluate the density.

Unfortunately, the Epanechnikov kernel is not differentiable at the boundary of its support. An alterative is the **tri-cube kernel**, defined as follows:

$$\kappa(x) \triangleq \frac{70}{81}(1-|x|^3)^3\mathbb{I}(|x| \leq 1) \tag{14.75}$$

This has compact support and has two continuous derivatives at the boundary of its support. See Figure 14.16.

The **boxcar kernel** is simply the uniform distribution:

$$\kappa(x) \triangleq \mathbb{I}(|x| \leq 1) \tag{14.76}$$

We will use this kernel below.

14.7.2 Kernel density estimation (KDE)

Recall the Gaussian mixture model from Section 11.2.1. This is a parametric density estimator for data in $\mathbb{R}^D$. However, it requires specifying the number K and locations $\boldsymbol{\mu}_k$ of the clusters. An alternative to estimating the $\boldsymbol{\mu}_k$ is to allocate one cluster center per data point, so $\boldsymbol{\mu}_i = \mathbf{x}_i$. In this case, the model becomes

$$p(\mathbf{x}|\mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{N}(\mathbf{x}|\mathbf{x}_i, \sigma^2\mathbf{I}) \tag{14.77}$$

We can generalize the approach by writing

$$\hat{p}(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N}\kappa_h\left(\mathbf{x}-\mathbf{x}_i\right) \tag{14.78}$$

This is called a **Parzen window density estimator**, or **kernel density estimator** (**KDE**), and is a simple non-parametric density model. The advantage over a parametric model is that no model fitting is required (except for tuning the bandwidth, usually done by cross-validation). and there is no need to pick K. The disadvantage is that the model takes a lot of memory to store, and a lot of time to evaluate. It is also of no use for clustering tasks.

Figure 14.17 illustrates KDE in 1d for two kinds of kernel. On the top, we use a boxcar kernel, $\kappa(x) = \mathbb{I}(-1 \leq z \leq 1)$. The result is equivalent to a **histogram** estimate of the density, since we just count how many data points land within an interval of size h around x_i. On the bottom, we use a Gaussian kernel, which results in a smoother fit.

The usual way to pick h is to minimize an estimate (such as cross validation) of the frequentist risk (see e.g., (Bowman and Azzalini 1997)). In Section 25.2, we discuss a Bayesian approach to non-parametric density estimation, based on Dirichlet process mixture models, which allows us

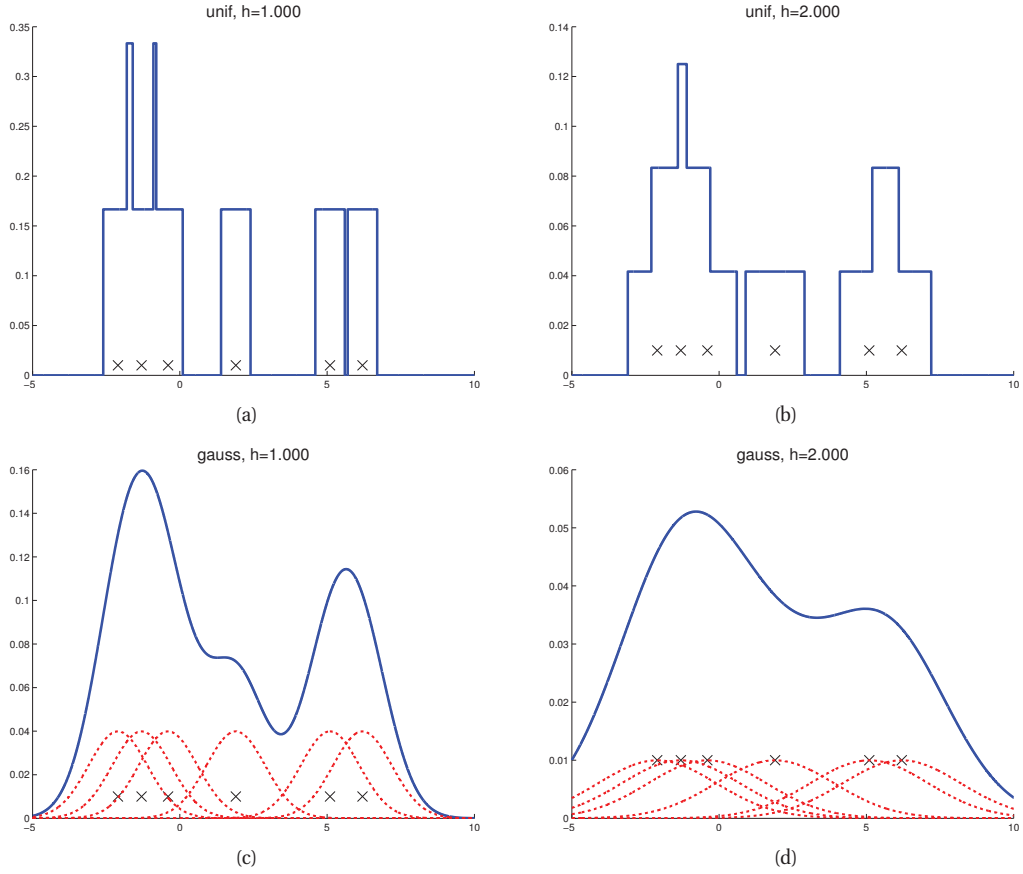


Figure 14.17 A nonparametric (Parzen) density estimator in 1D estimated from 6 data points, denoted by x. Top row: uniform kernel. Bottom row: Gaussian kernel. Rows represent increasingly large bandwidth parameters. Based on `http://en.wikipedia.org/wiki/Kernel_density_estimation`. Figure generated by `parzenWindowDemo2`.

to infer h. DP mixtures can also be more efficient than KDE, since they do not need to store all the data. See also Section 15.2.4 where we discuss an empirical Bayes approach to estimating kernel parameters in a Gaussian process model for classification/ regression.

14.7.3 From KDE to KNN

We can use KDE to define the class conditional densities in a generative classifier. This turns out to provide an alternative derivation of the nearest neighbors classifier, which we introduced in Section 1.4.2. To show this, we follow the presentation of (Bishop 2006a, p125). In kde with a boxcar kernel, we fixed the bandwidth and count how many data points fall within the hyper-cube centered on a datapoint. Suppose that, instead of fixing the bandwidth h, we instead

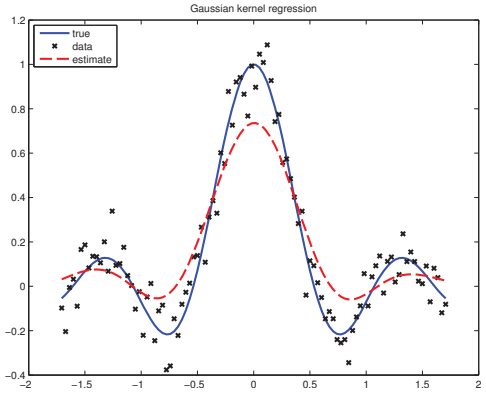


Figure 14.18 An example of kernel regression in 1d using a Gaussian kernel. Figure generated by `kernelRegressionDemo`, based on code by Yi Cao.

allow the bandwidth or volume to be different for each data point. Specifically, we will "grow" a volume around $\mathbf{x}$ until we encounter K data points, regardless of their class label. Let the resulting volume have size $V(\mathbf{x})$ (this was previously h^D), and let there be $N_c(\mathbf{x})$ examples from class c in this volume. Then we can estimate the class conditional density as follows:

$$p(\mathbf{x}|y=c,\mathcal{D}) = \frac{N_c(\mathbf{x})}{N_c V(\mathbf{x})} \tag{14.79}$$

where N_c is the total number of examples in class c in the whole data set. The class prior can be estimated by

$$p(y=c|\mathcal{D}) = \frac{N_c}{N} \tag{14.80}$$

Hence the class posterior is given by

$$p(y=c|\mathbf{x},\mathcal{D}) = \frac{\frac{N_c(\mathbf{x})}{N_c V(\mathbf{x})}\frac{N_c}{N}}{\sum_{c'}\frac{N_{c'}(\mathbf{x})}{N_{c'}V(\mathbf{x})}\frac{N_{c'}}{N}} = \frac{N_c(\mathbf{x})}{\sum_{c'}N_{c'}(\mathbf{x})} = \frac{N_c(\mathbf{x})}{K} \tag{14.81}$$

where we used the fact that $\sum_c N_c(\mathbf{x}) = K$, since we choose a total of K points (regardless of class) around every point. This is equivalent to Equation 1.2, since $N_c(\mathbf{x}) = \sum_{i\in N_K(\mathbf{x},\mathcal{D})}\mathbb{I}(y_i = c)$.

14.7.4 Kernel regression

In Section 14.7.2, we discussed the use of kernel density estimation or KDE for unsupervised learning. We can also use KDE for regression. The goal is to compute the conditional expectation

$$f(\mathbf{x}) = \mathbb{E}\left[y|\mathbf{x}\right] = \int y\, p(y|\mathbf{x})dy = \frac{\int y\, p(\mathbf{x},y)dy}{\int p(\mathbf{x},y)dy} \tag{14.82}$$

We can use KDE to approximate the joint density $p(\mathbf{x}, y)$ as follows:

$$p(\mathbf{x}, y) \approx \frac{1}{N} \sum_{i=1}^{N} \kappa_h(\mathbf{x} - \mathbf{x}_i)\kappa_h(y - y_i) \tag{14.83}$$

Hence

$$f(\mathbf{x}) = \frac{\frac{1}{N}\sum_{i=1}^{N} \kappa_h(\mathbf{x} - \mathbf{x}_i) \int y\kappa_h(y - y_i)dy}{\frac{1}{N}\sum_{i=1}^{N} \kappa_h(\mathbf{x} - \mathbf{x}_i) \int \kappa_h(y - y_i)dy} \tag{14.84}$$

$$= \frac{\sum_{i=1}^{N} \kappa_h(\mathbf{x} - \mathbf{x}_i)y_i}{\sum_{i=1}^{N} \kappa_h(\mathbf{x} - \mathbf{x}_i)} \tag{14.85}$$

To derive this result, we used two properties of smoothing kernels. First, that they integrate to one, i.e., $\int \kappa_h(y - y_i)dy = 1$. And second, the fact that $\int y\kappa_h(y - y_i)dy = y_i$. This follows by defining $x = y - y_i$ and using the zero mean property of smoothing kernels:

$$\int (x + y_i)\kappa_h(x)dx = \int x\kappa_h(x)dx + y_i \int \kappa_h(x)dx = 0 + y_i = y_i \tag{14.86}$$

We can rewrite the above result as follows:

$$f(\mathbf{x}) = \sum_{i=1}^{N} w_i(\mathbf{x})y_i \tag{14.87}$$

$$w_i(\mathbf{x}) \triangleq \frac{\kappa_h(\mathbf{x} - \mathbf{x}_i)}{\sum_{i'=1}^{N} \kappa_h(\mathbf{x} - \mathbf{x}_{i'})} \tag{14.88}$$

We see that the prediction is just a weighted sum of the outputs at the training points, where the weights depend on how similar $\mathbf{x}$ is to the stored training points. This method is called **kernel regression**, **kernel smoothing**, or the **Nadaraya-Watson** model. See Figure 14.18 for an example, where we use a Gaussian kernel.

Note that this method only has one free parameter, namely h. One can show (Bowman and Azzalini 1997) that for 1d data, if the true density is Gaussian and we are using Gaussian kernels, the optimal bandwidth h is given by

$$h = \left(\frac{4}{3N}\right)^{1/5} \hat{\sigma} \tag{14.89}$$

We can compute a robust approximation to the standard deviation by first computing the **mean absolute deviation**

$$\text{MAD} = \text{median}(|\mathbf{x} - \text{median}(\mathbf{x})|) \tag{14.90}$$

and then using

$$\hat{\sigma} = 1.4826 \text{ MAD} = \frac{1}{0.6745} \text{ MAD} \tag{14.91}$$

The code used to produce Figure 14.18 estimated h_x and h_y separately, and then set $h = \sqrt{h_x h_y}$.

Although these heuristics seem to work well, their derivation rests on some rather dubious assumptions (such as Gaussianity of the true density). Furthermore, these heuristics are limited to tuning just a single parameter. In Section 15.2.4 we discuss an empirical Bayes approach to estimating multiple kernel parameters in a Gaussian process model for classification/ regression, which can handle many tuning parameters, and which is based on much more transparent principles (maximizing the marginal likelihood).

14.7.5 Locally weighted regression

If we define $\kappa_h(\mathbf{x} - \mathbf{x}_i) = \kappa(\mathbf{x}, \mathbf{x}_i)$, we can rewrite the prediction made by kernel regression as follows

$$\hat{f}(\mathbf{x}_*) = \sum_{i=1}^{N} y_i \frac{\kappa(\mathbf{x}_*, \mathbf{x}_i)}{\sum_{i'=1}^{N} \kappa(\mathbf{x}_*, \mathbf{x}_{i'})} \tag{14.92}$$

Note that $\kappa(\mathbf{x}, \mathbf{x}_i)$ need not be a smoothing kernel. If it is not, we no longer need the normalization term, so we can just write

$$\hat{f}(\mathbf{x}_*) = \sum_{i=1}^{N} y_i \kappa(\mathbf{x}_*, \mathbf{x}_i) \tag{14.93}$$

This model is essentially fitting a constant function locally. We can improve on this by fitting a linear regression model for each point $\mathbf{x}_*$ by solving

$$\min_{\boldsymbol{\beta}(\mathbf{x}_*)} \sum_{i=1}^{N} \kappa(\mathbf{x}_*, \mathbf{x}_i)[y_i - \boldsymbol{\beta}(\mathbf{x}_*)^T \boldsymbol{\phi}(\mathbf{x}_i)]^2 \tag{14.94}$$

where $\boldsymbol{\phi}(\mathbf{x}) = [1, \mathbf{x}]$. This is called **locally weighted regression**. An example of such a method is **LOESS**, aka **LOWESS**, which stands for "locally-weighted scatterplot smoothing" (Cleveland and Devlin 1988). See also (Edakunni et al. 2010) for a Bayesian version of this model.

We can compute the paramters $\boldsymbol{\beta}(\mathbf{x}_*)$ for each test case by solving the following weighted least squares problem:

$$\boldsymbol{\beta}(\mathbf{x}_*) = (\boldsymbol{\Phi}^T \mathbf{D}(\mathbf{x}_*) \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^T \mathbf{D}(\mathbf{x}_*) \mathbf{y} \tag{14.95}$$

where $\boldsymbol{\Phi}$ is an $N \times (D+1)$ design matrix and $\mathbf{D} = \text{diag}(\kappa(\mathbf{x}_*, \mathbf{x}_i))$. The corresponding prediction has the form

$$\hat{f}(\mathbf{x}_*) = \boldsymbol{\phi}(x_*)^T \boldsymbol{\beta}(\mathbf{x}_*) = (\boldsymbol{\Phi}^T \mathbf{D}(\mathbf{x}_*) \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^T \mathbf{D}(\mathbf{x}_*) \mathbf{y} = \sum_{i=1}^{N} w_i(\mathbf{x}_*) y_i \tag{14.96}$$

The term $w_i(\mathbf{x}_*)$, which combines the local smoothing kernel with the effect of linear regression, is called the **equivalent kernel**. See also Section 15.4.2.

Exercises

Exercise 14.1 Fitting an SVM classifier by hand

(Source: Jaakkola.) Consider a dataset with 2 points in 1d: $(x_1 = 0, y_1 = -1)$ and $(x_2 = \sqrt{2}, y_2 = 1)$. Consider mapping each point to 3d using the feature vector $\boldsymbol{\phi}(x) = [1, \sqrt{2}x, x^2]^T$. (This is equivalent to

using a second order polynomial kernel.) The max margin classifier has the form

$$\min ||\mathbf{w}||^2 \quad \text{s.t.} \tag{14.97}$$

$$y_1(\mathbf{w}^T\phi(\mathbf{x}_1) + w_0) \geq 1 \tag{14.98}$$

$$y_2(\mathbf{w}^T\phi(\mathbf{x}_2) + w_0) \geq 1 \tag{14.99}$$

a. Write down a vector that is parallel to the optimal vector $\mathbf{w}$. Hint: recall from Figure 7.8 (12Apr10 version) that $\mathbf{w}$ is perpendicular to the decision boundary between the two points in the 3d feature space.

b. What is the value of the margin that is achieved by this $\mathbf{w}$? Hint: recall that the margin is the distance from each support vector to the decision boundary. Hint 2: think about the geometry of 2 points in space, with a line separating one from the other.

c. Solve for $\mathbf{w}$, using the fact the margin is equal to $1/||\mathbf{w}||$.

d. Solve for w_0 using your value for $\mathbf{w}$ and Equations 14.97 to 14.99. Hint: the points will be on the decision boundary, so the inequalities will be tight.

e. Write down the form of the discriminant function $f(x) = w_0 + \mathbf{w}^T\boldsymbol{\phi}(x)$ as an explicit function of x.

Exercise 14.2 Linear separability

(Source: Koller..) Consider fitting an SVM with $C > 0$ to a dataset that is linearly separable. Is the resulting decision boundary guaranteed to separate the classes?