Chapter 5

Kernels

Machine learning often considers problems where we profile objects with attribute-value vectors. This representation has, in principle, several limitations. Objects may be complex, and their vector-based representation is not trivial. Consider text documents, molecular structures, trees, graphs, and networks. For these, an alternative to feature-based representation is the utility of a function that can measure object-to-object similarity. Moreover, even if feature-based representation is available, it may be too weak to allow simpler models, like linear and logistic regression, to model more complex relations, like feature interactions. One way to surpass such limitations is kernels. In general, kernels are functions that map a pair of input objects to a number. One use of kernels is to consider a prototype object and then map input space into a latent representation, where selected modeling techniques may be more successful. When applied to a pair of data instances, we can regard kernels as functions that measure similarity. Smoothing kernels as used in kernel density estimation, which has a substantially different meaning. In this chapter, we look at a range of typical kernels and approaches that can use kernels in model inference.

In the previous chapters, we have introduced machine learning models that consider a set of training data to infer a predictive model. The training data is then discarded, and predictions for new inputs are formed entirely based on the model and its inferred parameters.

In this chapter, we introduce a different class of machine learning techniques that keeps the training data and uses it within the prediction phase. We have already exposed one such algorithm, namely k-nearest neighbors. We refer to algorithms of this kind as *lazy*, or *memory-based*. They typically require a metric of similarity of any two data points from the input space. We can recast many linear parametric models into an equivalent dual representation in which the predictions are based on a linear combination of a *kernel function* evaluated in the original, input space. For models, which are based on a fixed non-linear *feature space* *mapping* ϕ, the kernel function is given by the relation

$$\kappa(\boldsymbol{x}, \boldsymbol{x}') = \phi(\boldsymbol{x})^{\mathsf{T}} \phi(\boldsymbol{x}').$$

The kernel function is, from the definition, symmetric. Again, and importantly, the definition above says that instead of computing the dot product between the vectors in the latent, mapped space, we can compute the kernel function in the space of original features. This concept, thus formulated as an inner product in a feature space, allows us to develop extensions of many well-known regression and classification methods. All we need to do is to reformulate the methods to operate with dot products of input vectors. When introducing transformation to latent space, this product is then replaced with the kernel. We show the utility of the kernel trick in detail for one regression and for one classification approach: linear regression with ridge regularization and support vector machines.

In particular, support vector machines have received much attention in the past, but their importance has been in decay with the introduction of recent approaches, including deep networks. A particularly important driver of support vector machine's success was the utility of kernels on structured objects, like text, voice, images, and graphs.

5.1 Examples of Kernel Functions

A *kernel function*, or just a *kernel* is defined as:

$$\kappa : \mathcal{X} \times \mathcal{X} \to \mathbb{R},$$

where $\mathcal{X}$ is our variable space or typically, an input space. A kernel is thus a function $\kappa(x, x')$ that takes a pair of elements from the input space $x, x' \in \mathcal{X}$ and maps them to a real number. In practice we typically deal with kernel functions where $\kappa(x, x') \geq 0$ and $\kappa(x, x') = \kappa(x', x)$. That is, non-negative and symmetric kernel functions, which allows us to interpret them as *similarity measures*.

Notice that *kernel* has different meanings in different contexts. We will be here covering three:

- First, we will look at kernels in a very general sense - as functions that map a pair of elements from our (input, feature) space to a number.
- Then we will move on to positive-definite (or Mercer) kernels, which are a special case of the former (that is, with additional requirements) and allow for more efficient computation that is the basis for models such as SVM and kernel ridge regression.
- And third, we will introduce smoothing kernels that are used in kernel density estimation and has a substantially different meaning.

Polynomial Kernel

A standard kernel that is related to a transformation to a latent space that can, for instance, yield a linearly-unserable data instances manageble under linear models (e.g., Fig. 5.4) is a polynomial kernel:

$$\kappa(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^\intercal \mathbf{x}' + 1)^n.$$

For $n = 2$, and assuming that $x = [u_1 u_2]^\intercal$ and $x' = [v_1 v_2]^\intercal$ we get

$$\begin{aligned}(\mathbf{x}^\intercal \mathbf{x}' + 1)^2 &= (u_1 v_1 + u_2 v_2 + 1)^2 \\ &= u_1^2 v_1^2 + u_2^2 v_2^2 + 1 + 2u_1 v_1 + 2u_2 v_2 + 2u_1 v_1 u_2 v_2 \\ &= \langle 1, \sqrt{2}\, u_1, \sqrt{2}\, u_2, u_1^2, \sqrt{2}\, u_1 u_2, u_2^2 \rangle^\intercal \langle 1, \sqrt{2}\, v_1, \sqrt{2}\, v_2, v_1^2, \sqrt{2}\, v_1 v_2, v_2^2 \rangle\end{aligned}$$

Polynomial kernel of second degree returns a dot product of vectors in six-dimensional space.

Radial Basis Function Kernels

The squared exponential kernel, or *Gaussian* kernel is defined by:

$$\kappa(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{1}{2}(\mathbf{x} - \mathbf{x}')^\intercal \Sigma^{-1} (\mathbf{x} - \mathbf{x}')\right)$$

When Σ is diagonal, this kernel can be expressed as:

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

We can interpret σ_j as characteristic lenght scale of the dimension. If we assume that all characteristic length scales are equal, then we can write this kernel as:

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

where σ^2 is known as a bandwidth. Since this kernel depends only on a function $||\mathbf{x} - \mathbf{x}'||$, that is, only on a distance between a point $\mathbf{x}$ and, say, a reference $\mathbf{x}'$, this kernel is a radial basis function and is often referred to as an *RBF kernel*.

Notice that an RBF kernel has a parameter, σ, that needs to be either set by the user given some domain knowledge or inferred from the data through, for example, internal cross-validation.

Linear Kernel

When $\phi(\boldsymbol{x}) = \boldsymbol{x}$, we get a linear kernel defined as:

$$\kappa(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{x}^{\mathsf{T}} x'$$

This kernel is useful if the original data is already high dimensional, and if the original set of features are informative. Examples of such data sets are frequent in text mining and related with a bag of words representation of text documents, or data set from molecular biology that involve thousands of genes or millions of single-nucleotide polymorphisms. In these cases, a linear combination of features may represent a sufficiently accurate decision boundary, and it may not be required to use some other latent representation.

Kernels for Comparing Text Documents

Notice that kernel provides a proxy for the similarity of data instances. Given two objects, we will be able to construct regressors or classifiers by only computing the kernels, that is, estimating the similarity between two objects. If the objects are text documents, we can represent the document with vectors that contain word frequencies. We often refer to this presentation as *bag of words*. Because we can consider documents of different lengths, the Euclidian distance would fail (why?). We can instead normalize the bag-of-words representation according to the document length or use the *cosine similarity* [1]:

$$\kappa(\boldsymbol{x}, \boldsymbol{x}') = \frac{\boldsymbol{x}^{\mathsf{T}} \boldsymbol{x}'}{||\boldsymbol{x}||\,||\boldsymbol{x}'||}$$

Cosine similarity measures the cosine of the angle between the two vectors $\boldsymbol{x}$ and $\boldsymbol{x}'$ that represent the corresponding documents. Since both vectors are count vectors, the cosine similarity will be between 0 and 1.

Bag of words representation may include punctuations and frequently occurring words, so-called stop words, that may obscure the differences between documents and yield document representations too similar to each other. Various techniques for text pre-processing to avoid this effect were proposed in the literature. Among the most frequently used are stop-words removal and transform called *term frequency-inverse document frequency*, which replaces word counts with weights so that to expose less frequent words.

String Kernels

Kernels that operate on strings report on string sequence similarity. In these times, it may not be hard to consider RNA sequences of viruses that have infected people at different continents.

[1]Euclidian distance between normalized vectors and cosine similarity are in practice almost identical. Find what is their relation mathematically!

Due to mutations, their sequence may be different, and so is the effect on a phenotype of a patient. We may predict these phenotypes using kernels that measure sequence similarity. Consider the following three sequences:

```
TCGGTTTAACGGATTATGGTAC
TCGGTCCAACGGATAATGGAAC
TCGGCGATTTAACGGATCGATTTATGGTAC
```

To compare them, we may, for instance, use edit distance, that is, the measure that reports how many atomic changes like deletions, insertion, or single nucleotide mutation we have to introduce in one sequence to derive another one. Sequence similarity may also be computed through the count of the substrings the two strings have in common, or through the length of the longest common substring, or similar. There is vast research on string similarity measures and means to compute them through sequence alignment in the literature of molecular biology, and interested readers should consult algorithms such as BLAST or CLUSTAL.

5.2 Kernelized Linear Models

One simple way to use kernels for classification or regression is to use *kernel machines*. A kernel machine is a generalized linear model where the input vector has the form

$$\phi(\boldsymbol{x}) = [\kappa(\boldsymbol{x}, \boldsymbol{\mu_1}), \ldots, \kappa(\boldsymbol{x}, \boldsymbol{\mu_K})]$$

where $\boldsymbol{\mu}_k$ is a set of *centroids* or *prototypes*, that is, a subset of examples from the training set. Considering kernels as proximity functions. Then, the above-defined *kernelized feature vector* can be, for a give data instance, regarded as a vector of similarity to the prototype data instances. The general idea is that wherever we use a linear term $\boldsymbol{\beta}^T\boldsymbol{x}$, that is, in linear regression, generalized linear models, ordinal regression or similar, we could instead transform the input vector x via a kernel function with respect to a set of *prototype* observations $\boldsymbol{\mu}_1$, ..., $\boldsymbol{\mu}_k$. After constructing these new features, we proceed with the inference as we would with the original modeling method, we have a different, transformed input space (the term used now is $\beta^T\phi(x)$.

Prototypes, if appropriately selected, may help linear models to model feature spaces with feature interactions. Consider an XOR problem and logistic regression. Using RBF kernel and four different prototypes, logistic regression would be able to infer a perfect model for this otherwise hard classification case. Notice, though, that choice of the prototypes here is essential. In general, though:

- The number of prototypes K can be less, above, or equal to the dimension of the original training set.

- We can choose the prototypes using some systematic approach, like clustering. Alternatively, we could use every training example, x, for a prototype.
- If the number of prototypes is large, we could use any of the sparsity-promoting priors on $\boldsymbol{\beta}$, as discussed in the chapter on regularization. We refer to such an approach is called *sparse vector machine*. The most natural choice is ℓ_1 regularization, an approach we refer to as *ℓ_1-regularized vector machine*, or L1VM. Another popular approach of creating a sparse kernel machine is a support vector machine, discussed in detail below.
- And, most importantly, worth emphasizing again – we can use this approach within any linear model we have learned so far. With this approach, we can produce non-parametric versions of those parametric approaches, where their expressiveness can grow with the number of data points.

5.3 Mercer Kernels

Mercer kernels are related to approaches that use kernels through the so-called kernel trick and where we define models where input data appears within the inner products of the input data instances. We here define the Mercer kernels, establish the equivalence between Mercer kernels and inner products in transformed space, and outline some of the rules to follow when constructing a new kernel.

Definition Mercer's Theorem

A kernel function κ of the form $\mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$ is said to be *symmetric positive semidefinite* if it is (a) symmetric: $\kappa(x, x') = \kappa(x', x)$ and (b) for any integer $m > 0$ and any set of m vectors $x_i \in \mathbb{R}^d$ the matrix

$$K = \begin{bmatrix} \kappa(x_1, x_1) & \dots & \kappa(x_1, x_m) \\ \vdots & \ddots & \vdots \\ \kappa(x_m, x_1) & \dots & \kappa(x_m, x_m) \end{bmatrix}$$

is positive semidefinite. This matrix is also called the *Gram* matrix.

Definition 1 *Mercer Kernel. A symmetric positive semidefinite kernel κ is also called a* Mercer kernel.

Definition 2 *Mercer's Theorem. If κ is a Mercer kernel then it is an inner product (dot product) $\kappa(x, x') = \langle \phi(x), \phi(x') \rangle$ for some (possibly infinite dimensional) mapping $\phi(x) : \mathbb{R}^m \rightarrow \mathcal{D}$.*

Note that the mapping ϕ is often called the *basis function* and the space $\mathcal{D}$ is called the *feature space*.

This theorem is fundamental. It justifies the *kernel trick* that we will see a couple of times later. The kernel trick is just a direct application of Mercer's theorem - if we have a method that depends only on the inner products, we can replace those inner products with any Mercer kernel to obtain the application of that method on the feature space $\mathcal{D}$ determined by that kernel. Of course, computing the kernel is, in most cases, much easier.

Polynomial Kernel Revisited

We can start by considering one-dimensional polynomial regression, which is equivalent to using the basis function $\phi : x \rightarrow (1, x, x^2, ..., x^r)$. For example, for quadratic polynomial regression ($r = 2$), $\phi : x \rightarrow (1, x, x^2)$ and for cubic polynomial regression ($r = 3$), $\phi : x \rightarrow (1, x, x^2, x^3)$. Similarly, for higher dimensional x, we have $\phi : (x_1, x_2) \rightarrow (1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$.

In practice, it turns out that the basis functions for polynomial kernels use slightly different weights. For the cubic example, instead of $\phi : x \rightarrow (1, x, x^2, x^3)$, we use $\phi : x \rightarrow (1, \sqrt{3}x, \sqrt{3}x^2, x^3)$. The squared-roots do not make any difference when considering linear combinations, as the inner two coefficients will just be scaled by $\sqrt{3}$. However, it does make the computation much more convenient:

$$\langle \phi(x), \phi(x') \rangle = 1 + 3xx' + 3x^2(x')^2 + x^3(x')^3 = (1 + xx')^3.$$

This can be generalized to arbitrary power r, obtaining the *polynomial kernel* $\kappa(x, x') = (1 + xx')^r$. The beauty of this kernel is that the computation of the inner product in feature space, which has $\binom{r+m}{m}$ dimensions, actually requires just one inner product in the original space (dimension m) adding one and taking power.

RBF Kernel Revisited

With no proof, we should state here that:

1. Radial basis function (RBF) kernel is a Mercer kernel.

2. RBF kernel corresponds to a basis function that transforms the data instance to an infinite-dimensional feature space, that is, to a space of an infinite sum of polynomial kernels. So not only is the feature space inner product more convenient to compute in original space, it would be impossible to compute in feature space.

Other Mercer Kernels

In general, it is difficult to verify if a kernel is a Mercer kernel. There are, however, operations that preserve the property. If κ_1 and κ_2 are Mercer kernels defined on the same feature space, then the following are also Mercer kernels (**ShaweTaylorCristianini2004**):

- $\kappa(x, x') = c\kappa_1(\boldsymbol{x}, \boldsymbol{x}')$, where $c > 0$ is a constant
- $\kappa(x, x') = f(\boldsymbol{x})\kappa_1(\boldsymbol{x}, \boldsymbol{x}')f(\boldsymbol{x}')$, where $f(x)$ is a real function with nonnegative coefficients
- $\kappa(x, x') = f(\kappa_1(\boldsymbol{x}, \boldsymbol{x}'))$
- $\kappa(x, x') = \kappa_1(\boldsymbol{x}, \boldsymbol{x}') + \kappa_2(\boldsymbol{x}, \boldsymbol{x}')$
- $\kappa(x, x') = \kappa_1(\boldsymbol{x}, \boldsymbol{x}')\kappa_2(\boldsymbol{x}, \boldsymbol{x}')$
- $\kappa(x, x') = \kappa_1(\phi(\boldsymbol{x}), \phi(\boldsymbol{x}'))$
- $\kappa(x, x') = \boldsymbol{x}^\intercal \boldsymbol{A}\boldsymbol{x}'$, where $\boldsymbol{A}$ is symmetric positive semidefinite matrix

Let us start with a linear function $\kappa(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\intercal \boldsymbol{x}')$, which is a kernel since $\boldsymbol{A} = \boldsymbol{I}$ is a positive definite matrix, (and as such also positive semi-definite). Then a simple polynomial kernel $\kappa(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\intercal \boldsymbol{x}')^2 =$ contains only terms of degree two, and is equal to the linear kernel squared, and is also a kernel. A slightly generalized function $\kappa(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\intercal \boldsymbol{x}' + c)^2$ with $c > 0$ is also a kernel since its expansion contains linear combinations of linear or polynomial kernels of degree two. Through similar reasoning, we can find taht $\kappa(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\intercal \boldsymbol{x}' + c)^M$ is a kernel for any $c \geq 0$.

Another commonly used kernel, a Gaussian kernel, takes the form

$$\kappa(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\left\|\boldsymbol{x} - \boldsymbol{x}'\right\|^2 / 2\sigma^2\right)$$

This is a valid kernel, sinceby expanding the square

$$\left\|\boldsymbol{x} - \boldsymbol{x}'\right\|^2 = \boldsymbol{x}^\intercal \boldsymbol{x} + (\boldsymbol{x}')^\intercal \boldsymbol{x}' - 2\boldsymbol{x}^\intercal \boldsymbol{x}'$$

we can write

$$\kappa(\boldsymbol{x}, \boldsymbol{x}') = \exp(-\boldsymbol{x}^\intercal \boldsymbol{x}/2\sigma^2)\exp(\boldsymbol{x}^\intercal \boldsymbol{x}'/\sigma^2)\exp(-(\boldsymbol{x}')^\intercal \boldsymbol{x}'/x\sigma^2)$$

and see that this is a valid kernel.

5.4 Application of the Theory of Mercer Kernels to Modelling

We want to express an existing linear method in terms of inner products and then replace them with a kernel to obtain the linear method in the transformed feature space, consequently constructing a non-linear model. That is, we apply the *kernel trick*. Of course, for this to work, the kernel needs to be a Mercer kernel, not just any kernel function. Below, we show that regularized linear regression has a dual form that uses dot products of input vectors, and hence it can be kernelized. We also derive support vector machines, a linear classification algorithm that uses a weighted sum of the dot product of pairs of data instances from the training data set. Furthermore, we briefly discuss a kernelized version of the k-nearest neighbors. For other methods, like support vector machine regression, kernelized principal component analysis, or kernelized k-means clustering, see (**2012-Murphy**).

Kernelized Ridge Regression and the Kernel Trick

First, recall what we've learned about ridge regression, that is, L_2-regularized linear regression. Again, let $X \in \mathbb{R}^{N\times D}$ be our independent variables and $y \in \mathbb{R}^N$ our dependent variable. Given a regularization parameter λ the objective is to find coefficients β that minimize the squared error and the squared sum of coefficients β,

$$\hat{\beta} = \arg\min_{\beta}\left(\|X\beta - y\|_2^2 + \lambda\|\beta\|_2^2\right).$$

We know that this has a closed-form solution $\hat{\beta} = (X^TX + \lambda I_d)^{-1}X^Ty$.

Now we will rewrite this solution in an alternative (dual) form, which will facilitate the use of a kernel. Observe that

$$(X^TX + \lambda I_d)X^T = X^TXX^T + \lambda X^T = X^T(XX^T + \lambda I_n).$$

Multiplying the leftmost and rightmost terms by $(X^TX+\lambda I_d)^{-1}$ on the left and $(XX^T+\lambda I_d)^{-1}$ on the right-hand side, we get

$$X^T(XX^T + \lambda I_n)^{-1} = (X^TX + \lambda I_d)^{-1}X^T.$$

So, we have found an alternative formulation of the closed-form solution:

$$\hat{\beta} = X^T(XX^T + \lambda I_n)^{-1}y.$$

Using this new formulation as depicted above, we can define the prediction for a new observation as

$$\hat{y}(x') = \hat{\beta}^Tx' = (x')^T\hat{\beta} = (x')^TX^T(XX^T + \lambda I_n)^{-1}y.$$

The critical observation here is that *the prediction depends on X and x′ only through standard inner products.*

More precisely, $(x')^TX^T = \begin{bmatrix}\langle x', x_1\rangle \\ \vdots \\ \langle x', x_n\rangle\end{bmatrix}^T$ and $XX^T = \begin{bmatrix}\langle x_1, x_1\rangle & \dots & \langle x_1, x_n\rangle \\ \vdots & \ddots & \vdots \\ \langle x_n, x_1\rangle & \dots & \langle x_n, x_n\rangle\end{bmatrix}$ is the Gram matrix.

So, we can apply the *kernel* trick and replace these inner products with the more general kernelized formulation.

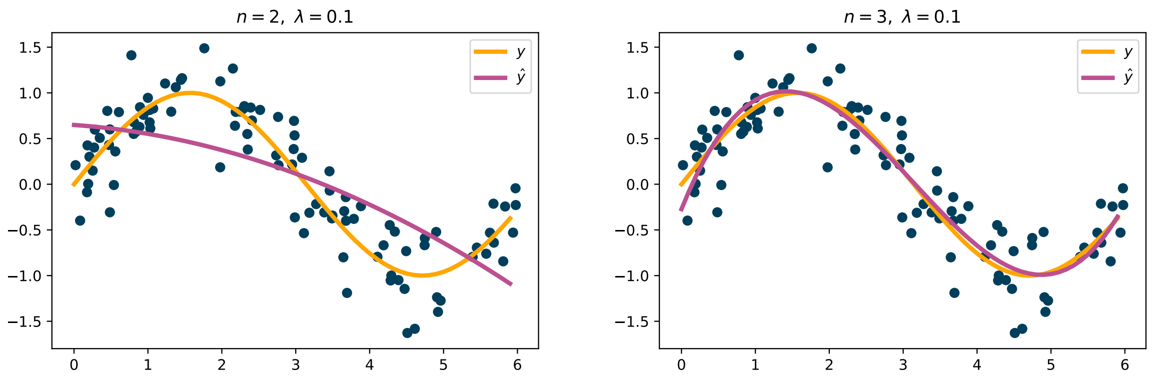

Figure 5.1: Kernelized ridge regression with polynomial kernel on one-dimensional data. Choice of hyper-parameter of the kernel can greatly influence the degree of fit.

$$\hat{y}(x') = k(x')(K + \lambda I_n)^{-1} y,$$

where $k(x') = \begin{bmatrix} \kappa(x', x_1) \\ \vdots \\ \kappa(x', x_n) \end{bmatrix}^T$ and $K = \begin{bmatrix} \kappa(x_1, x_1) & \dots & \kappa(x_1, x_n) \\ \vdots & \ddots & \vdots \\ \kappa(x_n, x_1) & \dots & \kappa(x_n, x_n) \end{bmatrix}$.

That is, using a kernel, we can perform ridge regression in the space whose inner product is represented by the kernel.

An alternative view is to explicitly introduce the (dual) variable $\alpha = (K + \lambda I_n)^{-1} y$ to allow us to express the closed-form solution as

$$\hat{\beta} = X^T \alpha = \sum \alpha_i x_i.$$

The solution to the problem is just a linear combination of the observations!

Plugging this into the prediction for the new observation, we get

$$\hat{y}(x') = (x')^T \sum \alpha_i x_i = \sum \alpha_i (x')^T x_i = \sum \alpha_i \kappa(x', x_i),$$

which illustrates that the prediction for a new observation is just a weighted sum of training observations' values of y (weighted by the similarity of those observations with the new observation x')! For example, for linear regression, where we just have the standard inner product, observations that have a smaller angle (closer to 0 or 180) have a higher weight.

The degree of fit of kernelized ridge regression depends on the choice of kernel and its parameters, and on the choice of degree of regularization (see Figs. 5.5 and **??**).

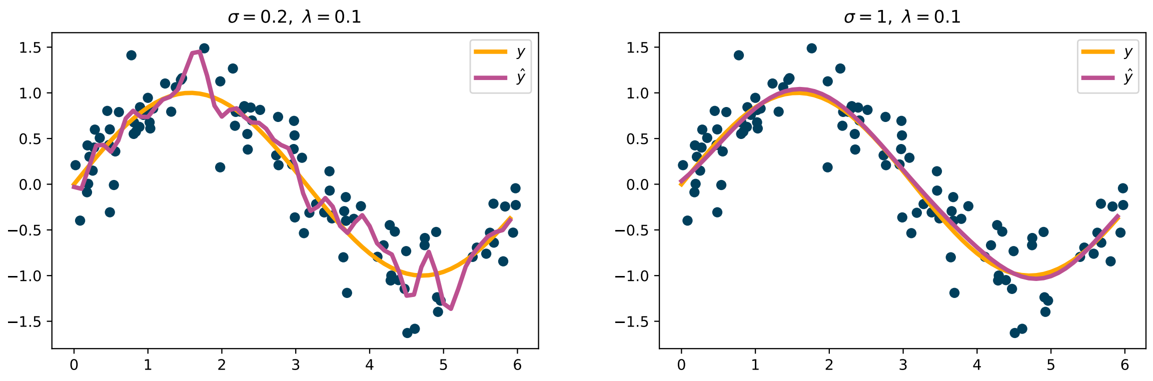

Figure 5.2: Wrong choice of hyperparameters can lead to overfitting, as shown in the plots of regression on one-dimensional data set with Gaussian kernel.

Support Vector Machines

Support vector machines are, in essence, linear classifiers that infer separating hyper-planes with maximal margins to the neighboring data points. We here derive the formulation of this classifier, show how to its power with a kernel trick, and describe extension for non-linearly separable cases. We also derive the formulation for support vector machines using hinge loss.

The Large Margin Principle

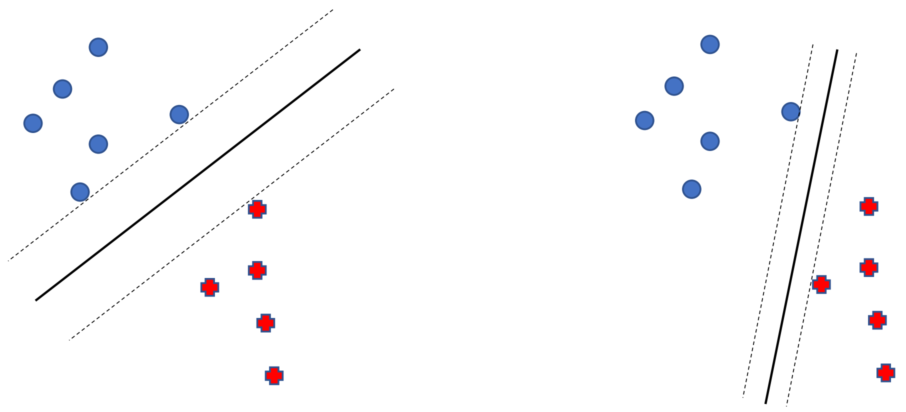

Figure 5.3: A separating hyper-plane with large (left) and small margin (right).

Consider a binary classification problem, with $y = \pm 1$, where data instances of the two classes are linearly separable. Our aim is to define a separating hyper-plane that splits the feature space so that the margin between the closest positive and the closest negative data instance is the largest 5.3. Intuitively, a large margin would steer us from overfitting and may yield best accuracy on yet unseen data. Let us denote the margin with γ. We would therefore like to find $\boldsymbol{w}$ that defines the direction of a separating hyper-plane $\boldsymbol{w}^\intercal \boldsymbol{x} + w_0 = 0$, where w_0 is the intercept, with largest margin γ. With no loss of generality, let us choose w so that the data points on the margin are one unit away from separating hyper-plane. The equations for the margins are thus:

$$\boldsymbol{w}^\intercal \boldsymbol{x} + w_0 = 1$$
$$\boldsymbol{w}^\intercal \boldsymbol{x} + w_0 = -1$$

Consider now a point $\boldsymbol{x}$ on one margin and its projection $\boldsymbol{x}_\perp$ accross separating hyperplane to the other margin:

$$\boldsymbol{x} = \boldsymbol{x}_\perp + 2\gamma \frac{\boldsymbol{w}}{||\boldsymbol{w}||}$$

Using the equations of the margin, so that $\boldsymbol{w}^\intercal \boldsymbol{x} + w_0 = 1$ and $\boldsymbol{w}^\intercal \boldsymbol{x}_\perp + w_0 = -1$, we obtain:

$$\boldsymbol{w}^\intercal \left(\boldsymbol{x}_\perp + 2\gamma \frac{\boldsymbol{w}}{||\boldsymbol{w}||} \right) + w_0 = 1,$$

$$\boldsymbol{w}^\intercal \boldsymbol{x} + w_0 + 2\gamma \frac{\boldsymbol{w}^\intercal \boldsymbol{w}}{||\boldsymbol{w}||} = 1,$$

$$-1 + 2\gamma \frac{\boldsymbol{w}^\intercal \boldsymbol{w}}{||\boldsymbol{w}||} = 1,$$

and finally, considering $\boldsymbol{w}^\intercal \boldsymbol{w} = ||w||^2$,

$$\gamma = \frac{1}{||\boldsymbol{w}||}$$

To maximize the margin, we need to minimize the length of $\boldsymbol{w}$. Formally, and considering that all data points in our linearly separable classification problem have to lie on or outside the margin, we therefore want to opimize

$$\min_{w,w_0} \frac{1}{2} ||w||^2 \;\; s.t. \;\; y_i(\boldsymbol{w}^\intercal \boldsymbol{x}_i + w_0) \geq 1, \; i = 1 \ldots N$$

For convenience, we have added $\frac{1}{2}$ and are optimizing the squared norm instead of the norm, which, of course, should be mathematically more convenient and leads to the same solution. We can now use the method of Lagrange multipliers to find the minima of our criteria function that succombs to the set of constrains. The primal Lagrangian is:

$$L(\boldsymbol{w}, w_0, \boldsymbol{\alpha}) = \frac{1}{2} ||w||^2 - \sum_{i=1}^{N} \alpha_i [y_i(\boldsymbol{w}^\intercal \boldsymbol{x}_i + w_0) - 1]$$

where $\alpha_i \geq 0$ are Lagrange multipliers. Optimal value of parameters that define the separating hyper-plane is where the gradient of the Lagrangian is zero:

$$\begin{aligned}\nabla L &= \frac{\partial L}{\partial \boldsymbol{w}} \\ &= \boldsymbol{w} - \sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x}_i = 0\end{aligned}$$

Therefore,

$$\boldsymbol{w} = \sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x}_i \tag{5.1}$$

Separating hyperlane is defined through a normal vector that is a weighted sum of vectors that define our training data instances!

Similar holds for w_0,

$$\frac{\partial L}{\partial b} = -\sum_{i=1}^{N} \alpha_i y_i = 0$$

and hence

$$\sum_{i=1}^{N} \alpha_i y_i = 0$$

We now insert the values of our optimal parameters back to the Lagrangian:

$$\begin{aligned} L &= \frac{1}{2}\left(\sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x}_i\right)\left(\sum_{j=1}^{N} \alpha_j y_j \boldsymbol{x}_j\right) - \left(\sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x}_i\right)\left(\sum_{j=1}^{N} \alpha_j y_j \boldsymbol{x}_j\right) - \sum_{i=1}^{N} \alpha_i y_i + \sum_{i=1}^{N} \alpha_i \\ &= \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i \boldsymbol{x}_j \end{aligned}$$

Support vector classifier is then defined through the following optimization problem:

$$\begin{aligned} \text{maximize} \quad & L(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i \boldsymbol{x}_j \\ \text{subject to} \quad & \sum_{i=1}^{N} y_i \alpha_i = 0, \\ & \alpha_i \geq 0,\ i = 1, \ldots, N \end{aligned}$$

The problem of finding the separating hyper-plane with maximal margin thus translates to quadratic programming, for which a standard solver can be used.

At this stage, notice that finding $\boldsymbol{\alpha}$ through quadratic programming provides us the weights $\boldsymbol{w}$ (see Eq. 5.1) and consequently provides for b. Notice that the solution of a quadratic programming problem, that is, the vector $\boldsymbol{\alpha}$ is typically sparse. That is, most α_i will be zero. That is, the weights $\boldsymbol{w}$ will be defined by only a small number of input data instances. These data instances are those that define the margin and are called *support vectors*. The support vectors are, therefore, training data instances for which $\alpha_i > 0$.

The decision rule for classification of data instance $\boldsymbol{x}$ to the positive class ($y = 1$) is

$$\boldsymbol{w}^\mathsf{T}\boldsymbol{x} + b \geq 0$$

and if we substitute for $\boldsymbol{w}$ we obtain:

$$\sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x_i}^\mathsf{T}\boldsymbol{x} + b \geq 0$$

Notice that in essence we do not need to compute the weights $\boldsymbol{w}$ for classification. All we need is to remember the instances for the training data set and compute the scalar product between training data instances and the vector the represents data instance to be classified. We can notice that in the quadratic programming formulation, where we solve for $\boldsymbol{\alpha}$, its dependency on the training data is again expressed through the scalar product $\boldsymbol{x}_i^\mathsf{T}\boldsymbol{x}_j$.

Support Vector Machines and Kernel Trick

Consider a data set with linearly non-separable classes (Fig. 5.4, left panel), and a transformation $\Phi(\boldsymbol{x}) = \langle x_1^2, x_2^2, \sqrt{2}\, x_1 x_2 \rangle$ that takes each two-dimensional data instance and transforms it into a latent space with three dimension. The transformed data set becomes linearly separable (Fig. 5.4, right panel). If $\boldsymbol{x}$ and $\boldsymbol{x}'$ are two vectors in original space, what is their dot product $\Phi(\boldsymbol{x})\Phi(\boldsymbol{x}')$ in transformed space?

$$\begin{aligned}
\Phi(\boldsymbol{x})\Phi(\boldsymbol{x}') &= \langle x_1^2, x_2^2, \sqrt{2}\, x_1 x_2 \rangle^\mathsf{T} \langle x_1'^2, x_2'^2, \sqrt{2}\, x_1' x_2' \rangle \\
&= x_1^2 x_1'^2 + 2 x_1 x_2 x_1' x_2' + x_2^2 x_2'^2 \\
&= (x_1 x_1' + x_2 x_2') \\
&= (\boldsymbol{x}^\mathsf{T}\boldsymbol{x}')^2
\end{aligned}$$

So, if, instead of using a dot product between two vectors $\boldsymbol{x}^\mathsf{T}\boldsymbol{x}'$ in formulation of SVM, we can replace it with a $\kappa(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^\mathsf{T}\boldsymbol{x}')^2$. This substitution is called a kernel trick. The kernel trick avoids the explicit mapping that is needed to, say, get a linear model to learn a nonlinear function or decision boundary. For all $\boldsymbol{x}$ and $\boldsymbol{x}'$ in the input space, certain functions $\kappa(\boldsymbol{x}, \boldsymbol{x}')$ can be expressed as an inner product in another space. The function κ is referred to as a kernel or a kernel function. Notice that the kernel trick allows us never to transform the data to the latent space, that is, never to use the transformation function $\Phi(\boldsymbol{x})$. In this way, the latent space could, in principle, have infinite dimensions, yet all we need is to compute the scalar product in this space. For any kernel, there is a corresponding function that transforms

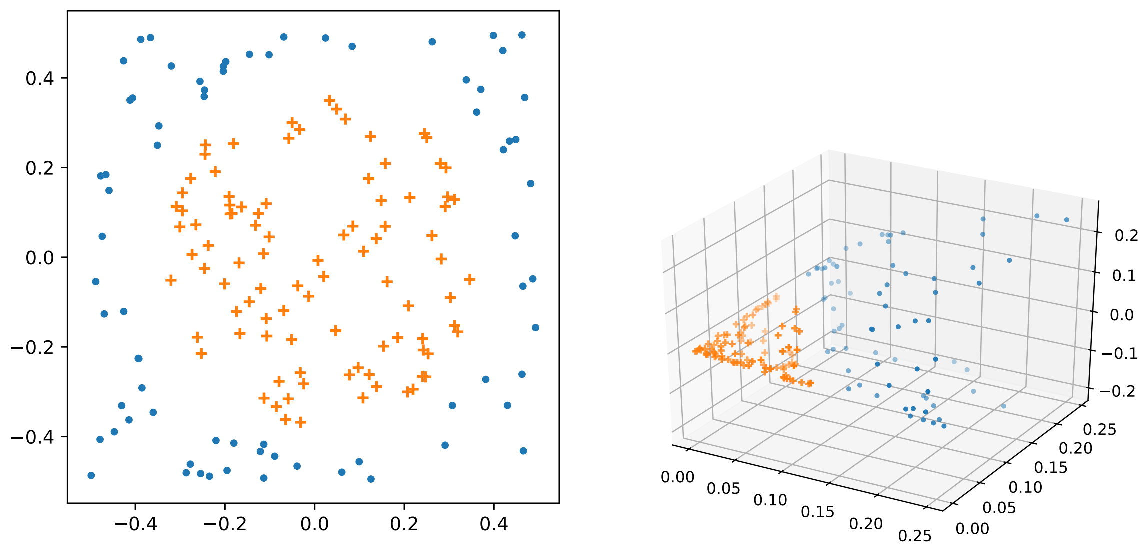

Figure 5.4: A two-feature binary classification data set (left), where the data points of different classes are not linearly separated. The same data set, where each point $\boldsymbol{x}$ was transformed through $\Phi(\boldsymbol{x})$, so that $\Phi(\boldsymbol{x}) = \langle x_1^2, x_2^2, \sqrt{2}\, x_1 x_2 \rangle$ (right).

the data from original space to latent space, and there are some kernels where the required dimensionality of latent space is infinite. We will discuss various kernel functions and their properties a bit later in this chapter.

Non-Linearly Separable Data

If the data is not linearly separated, which is expected from any real data that contains some noise, the support vector machine that we have defined so far does not have any solution. For such cases, we need to introduce slack variables $\xi_i \geq 0$ such that slack is zero if the data instance is on or inside the correct margin boundary, or the slack is positive and equal to

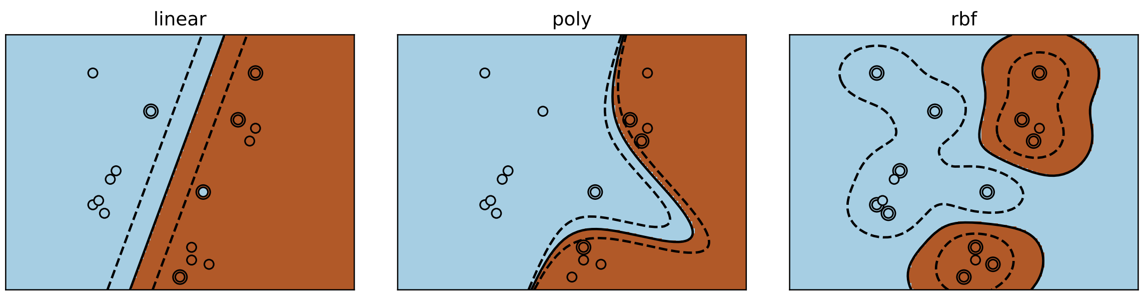

Figure 5.5: Classification in two-dimensional feature space shows decision boundaries of support vector machines with linear, polynomial ($n = 3$), and Gaussian kernel. Double-circled data points are support vectors. Code for figures by Gaël Varoquaux.

the distance to the corresponding margin. Notice that if $0 < \xi_i \geq 1$ the point lies inside the margin, but on the corre side of decision boundary. To solve for this problem, we replace the hard constrains with *soft margin constraints* with the new objective, where, as before, we would like to maximize the margin with minimal use of slack:

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

The corresponding Lagrangian is

$$L(\boldsymbol{w}, w_0, \boldsymbol{\alpha}) = \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}\xi_i - \sum_{i=1}^{N}\alpha_i[y_i(\boldsymbol{w}^\intercal \boldsymbol{x}_i + w_0) - 1 + \xi_i] - \sum_{i=1}^{N}\mu_i\xi_i$$

After computing the corresponding gradients and replacing the results in the primal Lagrangian, the dual form of Lagrangian is the same as before,

$$\sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j \boldsymbol{x}_i \boldsymbol{x}_j$$

with the only exception of a box constrain $0 \leq \alpha_i \leq C$. Notice that the parameter C is a regularization parameter that controls the number of errors we are willing to tolerate on the training set. This parameter is commonly defined as $C = 1/(\nu N)$, where $0 < \nu \leq 1$ is the fraction of misclassified points we allow during the training phase. This version of the algorithm is called ν-SVM classifier.

Multi-Class Classification

We have defined support vectors classifiers for supervised learning on binary classification data. Unlike some other approaches, like softmax for logistic regression, SVMs do not have a natural extension that would be appropriate to treat multi-class data. Standard approaches that solve multi-class problems through inference of a set of binary classifiers include a one-versus-all approach and one-versus-one approach. However, the problem to use them is that SVM, in its original formulation, is not a probabilistic classifier.

Choosing the Value of Hyper-Parameters

Regularization parameters like C and ν are parameters of the algorithm, that is, hyper-parameters that need to be set before the inference of the model. To search for the most appropriate value, we can use approaches such as the search over a fixed set of parameters and estimating their appropriateness through cross-validated accuracy on the training data. We then use the parameter value with the highest estimated accuracy to infer the model from the entire training set.

Hinge Loss

An alternative derivation of support vector machine comes from the utility of *hige loss*, a variant of a loss function for binary classification, defined as:

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

Here, $\eta = f(\boldsymbol{x}) = \boldsymbol{w}^\top x + w_0$ can be regarded as "confidence" in choosing label $y = 1$. The overall objective is again to maximize the margin while minimizing the loss, therefore:

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

This objective function is non-differentiable because of the max term. We can replace this term with slack variables, and request in the revised objective that the slack is minimized. The new objective function

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

is exactly the same as the one derived above, in the section on the treatment of non-linearably separable data.

Kernelized k-Nearest Neighbors

k-nearest neighbor algorithms are based on distance, that is, they find the observations closest to the one we are predicting for. If using an Euclidean distance, this can be expressed in terms of inner products

$$||x, x'||^2 = ||x||^2 + ||x'||^2 - 2\langle x, x' \rangle = \langle x, x \rangle^2 + \langle x', x' \rangle^2 - 2\langle x, x' \rangle.$$

Thus, we can replace these dot products with a kernel, and in this way perform k-nearest neighbors in the feature space of a chosen kernel.

Support Vector Machines Regression

Here, we motivate SVM with the problem of kernelized ridge regression, and actually with other approaches discussed so far that use the kernel trick, where the solution is not sparse. More precisely, the solution of kernelized ridge regression $\hat{\beta} = \sum \alpha_i \boldsymbol{x}_i$ and subsequently the predictions $\hat{y}(x') = \sum \alpha_i \kappa(\boldsymbol{x}', \boldsymbol{x}_i)$ depend on all training observations.

A key idea of SVMs is to introduce sparsity through the loss function. Vapnik proposed the *epsilon insensitive loss function*:

$$L_\epsilon(y, \hat{y}) = |y - \hat{y}| - \epsilon \text{ if } |y - \hat{y}| \geq \epsilon \text{ and } 0 \text{ otherwise.}$$

Basically, anything that has an absolute error less than ϵ is not penalized. The objective function we want to minimize is then

$$J = C \sum L_\epsilon(y_i, \hat{y}_i) + \frac{1}{2}||\beta||^2,$$

where $\hat{y}_i = f(\boldsymbol{x}_i) = \beta^T \boldsymbol{x}_i + \beta_0$ and $C = \frac{1}{\lambda}$ is a regularization constant.

This optimization problem is non-differentiable because of the absolute value in the loss function. Typically, it is reformulated as a constrained optimization problem by introducing slack variables ξ:

$$y_i - f(\boldsymbol{x}_i) \leq +\epsilon + \xi_i^+,$$
$$y_i - f(\boldsymbol{x}_i) \geq -\epsilon - \xi_i^-.$$

Now we can reformulate the objective function.

$$J = C \sum (\xi_i^+ + \xi_i^-) + \frac{1}{2}||\boldsymbol{\beta}||^2.$$

This is a quadratic function of $\boldsymbol{\beta}$ and we need to minimize it, subject to the above constraints and $\xi_i^+ \geq 0$, $\xi_i^- \geq 0$. Standard quadratic programming black-box solvers can be used.

If it is not immediately clear what we did with the slack variables: First, we deliberately put the $f(\boldsymbol{x}_i)$ on the left-hand side of the inequality constraints above. This is to illustrate the fact that for observations that we can fit within ϵ, the constraint will immediately be true, and both ξ_i can be 0. For all other observations, one of the inequality constraints will be violated, but not both. So, one of the slack variables will be 0, and the other will be exactly the amount the fit exceeds ϵ. This is what forces the $f(\boldsymbol{x}_i)$ as close to y_i as possible, in the absolute sense; the only way to make ξ_i as close to 0 as possible, again, in the absolute sense, is to make the $f(x_i)$ as close to y_i as possible. That is, in the optimal solution, the sum of ξ will be exactly the total sum of all exceedances of ϵ), so by minimizing the one, we minimize the other. But we got rid of the absolute term.

The solution of the above optimization problem (**2012-Murphy**) has, not surprisingly, the form

$$\hat{\boldsymbol{\beta}} = \sum_i \alpha_i \boldsymbol{x}_i,$$

where $\alpha_i \geq 0$. Notice that this solution is typically sparse, as most $\alpha_i = 0$. The $\boldsymbol{x}_i$ where $\alpha_i > 0$ are again called *support vectors*. We should emphasize that the sparseness comes from the objective function (ϵ). In practice, and depending on training data, we sometimes do get unlucky as it turns out that most x_i from the training data are support vectors.

SVM regression is typically solved in its dual form. The form above is the primal form. The two forms are the same as the optimization problem is convex. The dual formulation is also where the result comes that the solution is a linear combination of observations, very similar to kernelized ridge regression.

As with kernelized ridge regression, the prediction for a new observation is

$$\begin{aligned}\hat{y}(x') &= \hat{\beta}_0 + x'^T\hat{\beta} \\ &= \hat{\beta}_0 + x'^T \sum \alpha_i x_i \\ &= \hat{\beta}_0 + \sum \alpha_i x'^T x_i \\ &= \hat{\beta}_0 + \sum \alpha_i \kappa(x', x_i)\end{aligned}$$

So, we can also use the kernel trick here and do SVM regression in a feature space determined by the choice of kernel.

Kernel Density Estimation

Here, we will discuss on a substantially different kind of kernels, the so-called *smoothing kernels*. We will use the smoothing kernels to create non-parametric density estimates $p(\mathbf{x})$, as well as for creating generative models for classification and regression of the form $p(y, \mathbf{x})$.

A smoothing kernel is a one-argument function that satisfies the following properties:

$$\int \kappa(x)dx = 1,$$

$$\int x\kappa(x)dx = 0,$$

$$\int x^2\kappa(x)dx > 0.$$

A simple example of a smoothing kernel is a *Gaussian kernel*,

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

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

$$\kappa_h(x) = \frac{1}{h}\kappa\frac{x}{h}$$

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

$$\kappa_h(\boldsymbol{x}) = \kappa_h(\|\boldsymbol{x}\|).$$

Kernel density estimators are different from, say, Gaussian mixture models, a parametric density estimator that requires speficying K prototypes. An alternative is to allocate one cluster center per data point, so that the kernel density estimator, also called the *Parzen window density estimator* becomes

$$\hat{p}(\boldsymbol{x}) = \frac{1}{N}\sum_{i=1}^{N} \kappa_h(\boldsymbol{x} - \boldsymbol{x}_i).$$

The advantage of kernel density estimator over the parametric models is no need for model fitting and no need to pick the prototypes. However, we need to tune for the bandwidth, which can be performed by internal cross-validation. Notice also that the choice of h influences the bias-variance tradeoff. More smoothing decreases bias but increases variance, less smoothing decreases variance but increases bias. The optimal point is somewhere in between.