Kernel Methods for Structured Data

Andrea Passerini

Kernel methods are a class of non-parametric learning techniques relying on kernels. A kernel generalizes dot products to arbitrary domains and can thus be seen as a similarity measure between data points with complex structures. The use of kernels allows to decouple the representation of the data from the specific learning algorithm, provided it can be defined in terms of distance or similarity between instances. Under this unifying formalism a wide range of methods have been developed, dealing with binary and multiclass classification, regression, ranking, clustering and novelty detection to name a few. Recent developments include statistical tests of dependency and alignments between related domains, such as documents written in different languages. Key to the success of any kernel method is the definition of an appropriate kernel for the data at hand. A well-designed kernel should capture the aspects characterizing similar instances while being computationally efficient. Building on the seminal work by D. Haussler on convolution kernels, a vast literature on kernels for structured data has arisen. Kernels have been designed for sequences, trees and graphs, as well as arbitrary relational data represented in first or higher order logic. From the representational viewpoint, this allowed to address one of the main limitations of statistical learning approaches, namely the difficulty to deal with complex domain knowledge. Interesting connections between the complementary fields of statistical and symbolic learning have arisen as one of the consequences. Another interesting connection made possible by kernels is between generative and discriminative learning. Here data are represented with generative models and appropriate kernels are built on top of them to be used in a discriminative setting.

In this chapter we revise the basic principles underlying kernel machines and describe some of the most popular approaches which have been developed. We give an extensive treatment of the literature on kernels for structured data and suggest some basic principles for developing novel ones. We finally discuss kernel methods for predicting structures. These algorithms deal with structured-output prediction, a learning setting in which the output is itself a structure which has to be predicted from the input one.

1 A Gentle Introduction to Kernel Methods

In the typical statistical learning framework a supervised learning algorithm is given a training set of input-output pairs $\mathscr{D} = \{(x_1, y_1), \ldots, (x_m, y_m)\}$, with $x_i \in \mathscr{X}$ and $y_i \in \mathscr{Y}$, sampled identically and independently from a fixed but unknown probability distribution ρ. The set $\mathscr{X}$ is called the input (or instance) space and can be any set. The set $\mathscr{Y}$ is called the output (or target) space. For instance, in the case of binary classification $\mathscr{Y} = \{-1, 1\}$ while the case of regression $\mathscr{Y}$ is the set of real numbers. The learning algorithm outputs a function $f : \mathscr{X} \mapsto \mathscr{Y}$ that approximates the probabilistic relation ρ between inputs and outputs. The class of functions that is searched is called the *hypothesis space*.

Andrea Passerini
Dipartimento di Ingegneria e Scienza dell'Informazione, Università degli Studi di Trento, Italy,
e-mail: passerini@disi.unitn.it

Intuitively, f should assign to a novel input x the same (or a similar) y of similar inputs already observed in the training set. A *kernel* is a function : $\mathscr{X} \times \mathscr{X} \mapsto \mathrm{I\!R}$ measuring the similarity between pairs of inputs. For example, the similarity between a pair of sequences could be the number of common subsequences of length up to a certain m. The kernel should satisfy the following equation:

$$k(x,x') = \langle \Phi(x), \Phi(x') \rangle.$$

It thus corresponds to mapping examples to a (typically high dimensional) *feature space* $\mathscr{H}$ and computing the dot product in that space. In the sequence example, $\mathscr{H}$ is made of vectors of booleans, with an entry for each possible sequence of length up to m given the alphabet. $\Phi(x)$ maps x to a vector with one for entries occurring in x and zero otherwise. However, the kernel function does not need to explicitly do the mapping (which can be even infinite dimensional) in computing the similarity.

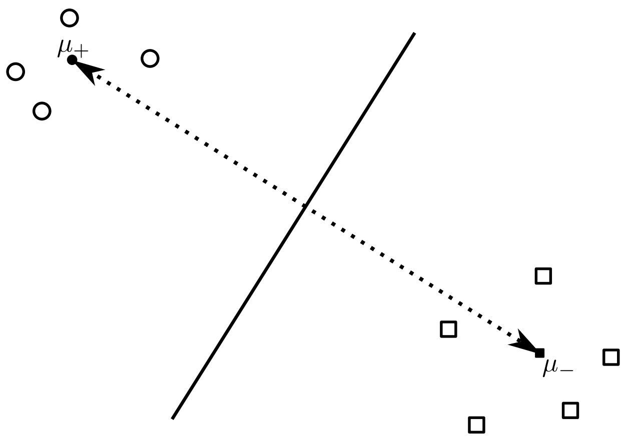


Fig. 1 A simple binary classifier in feature space: μ_+ and μ_- are the mean vectors of positive (circles) and negative (squares) examples respectively. The algorithm assigns a novel example to the class with the nearer mean. The decision boundary is a hyperplane (solid line) half-way down between the line linking the means (the dotted line).

Kernels allow to construct algorithms in dot product spaces and apply them to data with arbitrarily complex structures. Consider a binary classification task ($\mathscr{Y} = \{-1, 1\}$). A simple similarity-based classification function [68] could assign to x the y of the examples which on average are more similar to it (see Figure 1). The algorithm starts by computing the means of the training examples for the two classes in feature space:

$$\mu_+ = \frac{1}{n_+} \sum_{i:y_i=+1} \Phi(x_i) \qquad \mu_- = \frac{1}{n_-} \sum_{i:y_i=-1} \Phi(x_i)$$

where n_+ and n_- are the number of positive and negative examples respectively. It then assigns a novel example to the class of the closer mean:

$$f(x) = \mathrm{sgn}\left(\langle \mu_+, \Phi(x) \rangle - \langle \mu_-, \Phi(x) \rangle\right)$$

where $\mathrm{sgn}(z)$ returns the sign of the argument and we assumed for simplicity that the two means have the same distance from the origin (otherwise a bias term should be included). The decision boundary is represented by a hyperplane which is half way down on the line linking the means and orthogonal to it. By replacing the formulas for the means, we obtain:

$$f(x) = \mathrm{sgn}\left(\frac{1}{n_+} \sum_{i:y_i=+1} \langle \Phi(x_i), \Phi(x) \rangle - \frac{1}{n_-} \sum_{i:y_i=-1} \langle \Phi(x_i), \Phi(x) \rangle \right).$$

Feature space mappings $\Phi(\cdot)$ only appear in dot products and can thus be replaced by kernel functions. This is commonly known as the *kernel trick*. The resulting f can be compactly written as:

$$f(x) = \operatorname{sgn}\left(\sum_i c_i k(x_i, x)\right),$$

where $c_i = y_i / n_{y_i}$. The unthresholded version of f (i.e. before applying sgn) is a linear combination of kernel functions "centered" on training examples x_i. This is a common aspect characterizing (with minor variations) kernel machines, as we will see in the rest of the chapter.

2 Mathematical Foundations

The kernel trick allows to implicitly compute a dot product between instances in a possibly infinite feature space. In this section we will treat in more detail the theory underlying kernel functions, showing how to verify if a given function is actually a valid kernel, and given a valid kernel how to generate a feature space such that the kernel computes a dot product in that space. We will then highlight the connections between kernel machines and regularization theory, showing how most supervised kernel machines can be seen as instances of regularized empirical risk minimization. We will focus on real valued functions but results can be extended to complex ones as well. Details and proofs of the reported results can be found in [4, 6, 64, 68].

2.1 Kernels

Let's start by providing some geometric structure to our feature spaces.

Definition 1 (Inner Product).
Given a vector space $\mathscr{X}$, an inner product is a map $\langle\cdot,\cdot\rangle : \mathscr{X} \times \mathscr{X} \to \mathbb{R}$ such that for every $x, x', x'' \in \mathscr{X}, \alpha \in \mathbb{R}$:

1. $\langle x, x' \rangle = \langle x', x \rangle$ $\qquad (symmetry)$
2. $\langle x + x'', x' \rangle = \langle x, x' \rangle + \langle x'', x' \rangle,\ \langle \alpha x, x' \rangle = \alpha \langle x, x' \rangle$
   $\langle x, x' + x'' \rangle = \langle x, x' \rangle + \langle x, x'' \rangle,\ \langle x, \alpha x' \rangle = \alpha \langle x, x' \rangle$ $\qquad (bilinearity)$
3. $\langle x, x \rangle \geq 0$ $\qquad (positive\ definiteness)$

If in condition 3. equality only holds for $x = 0_{\mathscr{X}}$ the inner product is *strict*.

Inner products are also known as dot or scalar products. A vector space endowed with an inner product is called an *inner product space*. As a simple example, the standard dot product in the space of n-dimensional real vectors $\mathbb{R}^n$ is

$$\langle x, x' \rangle = \sum_{i=1}^{n} x_i x'_i.$$

A norm can be defined as $||x||_2 = \sqrt{\langle x, x \rangle}$ and a distance as $d(x, x') = ||x - x'||_2 = \sqrt{\langle x, x \rangle - 2\langle x, x' \rangle + \langle x', x' \rangle}$. A *Hilbert* space is an inner product space with two additional properties (completeness and separability) guaranteeing that it is isomorphic to some standard spaces ($\mathbb{R}^n$ or its infinite dimensional analogue L_2, the set of square convergent real sequences). Hilbert spaces are often infinite dimensional.

Feature maps $\Phi : \mathscr{X} \to \mathscr{H}$ map instances into a Hilbert space. In the following we will provide conditions guaranteeing that a kernel function k acts as a dot product in the Hilbert space of a certain map.

Definition 2 (Gram Matrix). Given a function $k : \mathscr{X} \times \mathscr{X} \to \mathbb{R}$ and patterns $x_1, \ldots, x_m$, the $m \times m$ matrix K such that

$$K_{ij} = k(x_i, x_j)$$

is called the *Gram matrix* of k with respect to $x_1, \ldots, x_m$.

Definition 3 (Positive Definite Matrix). A symmetric $m \times m$ matrix K is *positive definite* if

$$\sum_{i,j=1}^{m} c_i c_j K_{ij} \geq 0, \quad \forall \mathbf{c} \in \mathrm{I\!R}^m.$$

If equality only holds for $c = \mathbf{0}$, the matrix is *strictly positive definite*.

Alternative conditions for positive definiteness are that all its eigenvalues are non-negative, or that there exists a matrix B such that $K = B^T B$.

Definition 4 (Positive Definite Kernel). A function $k : \mathscr{X} \times \mathscr{X} \to \mathrm{I\!R}$ such that $\forall m \in \mathrm{I\!N}$ and $\forall x_1, \ldots, x_m \in \mathscr{X}$ it gives rise to a positive definite Gram matrix is called a *positive definite kernel*.[1]

Theorem 1 (Valid Kernels).
A kernel function $k : \mathscr{X} \times \mathscr{X} \to \mathrm{I\!R}$ *corresponds to a dot product in a Hilbert space* $\mathscr{H}$ *obtained by a feature map* Φ*:*

$$k(x, x') = \langle \Phi(x), \Phi(x') \rangle \tag{1}$$

if and only if it is positive definite.

Proof. The 'if' implication can be proved by building a map from $\mathscr{X}$ into a space where k acts as a dot product. We will actually build a map into a feature space of functions:

$$\Phi : \mathscr{X} \to \mathrm{I\!R}^{\mathscr{X}} \mid \Phi(x) = k(\cdot, x).$$

Φ maps an instance into a kernel function "centered" on the instance itself. In order to turn this space of functions into a Hilbert space, we need to make it a vector space and provide a dot product. A vector space is obtained taking the span of kernel k, that is all functions

$$f(\cdot) = \sum_{i=1}^{m} \alpha_i k(\cdot, x_i)$$

for all $m \in \mathrm{I\!N}$, $\alpha_i \in \mathrm{I\!R}$, $x_i \in \mathscr{X}$. A dot product in such space between f and another function

$$g(\cdot) = \sum_{j=1}^{m'} \beta_j k(\cdot, x'_j)$$

can be defined as

$$\langle f, g \rangle \;=\; \sum_{i=1}^{m} \sum_{j=1}^{m'} \alpha_i \beta_j k(x_i, x'_j). \tag{2}$$

Note that in order for eq. (2) to satisfy the positive definiteness property of an inner product (see Definition 1) the kernel $k(x, x')$ needs to be positive definite. For each given function f, it holds that

$$\langle k(\cdot, x), f(\cdot) \rangle \;=\; f(x). \tag{3}$$

In particular, for $f = k(\cdot, x')$ we have:

$$\langle k(\cdot, x), k(\cdot, x') \rangle \;=\; k(x, x').$$

[1] Note that part of the literature calls such kernels and matrices positive semi-definite, indicating with positive definite the strictly positive definite case.

By satisfying equation (1) we showed that each positive definite kernel can be seen as a dot product in another space. In order to show that the converse is also true, it suffices to prove that given a map Φ from $\mathscr{X}$ to a product space, the corresponding function $k(x,x') = \langle \Phi(x), \Phi(x') \rangle$ is a positive definite kernel. This can be proved by noting that for all $m \in \mathbb{N}$, $\mathbf{c} \in \mathbb{R}^m$ and $x_1, \ldots, x_m \in \mathscr{X}$ we have

$$\sum_{i,j=1}^{m} c_i c_j k(x_i, x_j) = \left\langle \sum_{i=1}^{m} c_i \Phi(x_i), \sum_{j=1}^{m} c_j \Phi(x_j) \right\rangle = \left|\left| \sum_{i=1}^{m} c_i \Phi(x_i) \right|\right|^2 \geq 0.$$

The existence of a map to a dot product space satisfying (1) is therefore an alternative definition for a positive definite kernel. □

A kernel satisfying equation (3) is said to have the *reproducing property*. The resulting space is named *Reproducing Kernel Hilbert Space* (RKHS). This is the hypothesis space we will deal with when developing supervised learning algorithms based on kernels. Note that while for a positive definite kernel k there is always a corresponding feature space of functions constructed as described in the proof[2] (and vice versa), there can be other (possibly finite dimensional) spaces working as well. A constructive example was presented in Section 1 for the common subsequence kernel. Other examples will be shown when describing kernels on structured data (Section 4).

2.2 Supervised Learning with Kernels

A key problem in supervised learning is defining an appropriate measure for the quality of the predictions. This is achieved by a *loss* function $V : \mathscr{Y} \times \mathscr{Y} \to [0, \infty)$, a non-negative function measuring the error between the actual and predicted output for a certain input $V(f(x), y)$. A simple example is the misclassification loss, which outputs one for an incorrect classification and zero otherwise. Learning aims at producing a function with the smallest possible expected risk, i.e. the probability of committing an error according to the data distribution ρ. Unfortunately, this distribution is usually unknown and one has to resort to the empirical risk, i.e. the average error on the training set $\mathscr{D}_m$:

$$R_{emp}[f] = \frac{1}{m} \sum_{i=1}^{m} V(f(x_i), y_i).$$

In order to prevent overfitting of training data, one has to impose some constraints on the possible hypotheses. The typical solution in machine learning is that of tending to prefer *simpler* hypotheses, by restricting the hypothesis space, biasing the learning algorithm for favouring them, or both. Most kernel machines rely on Tikhonov regularization [76], in which a regularization term $\Omega[f]$ is added to the empirical risk in order to bias learning towards more stable solutions:

$$R_{reg}[f] = R_{emp}[f] + \lambda \Omega[f].$$

The regularization parameter $\lambda > 0$ trades the effect of training errors with the complexity of the function. By choosing Ω to be convex, and provided $R_{emp}[f]$ is also convex, the problem has a unique global minimum. A common regularizer is the squared norm of the function, i.e. $\Omega[f] = ||f||^2$.

When the hypothesis space is a reproducing kernel Hilbert space $\mathscr{H}$ associated to a kernel k, the *representer theorem* [41] gives an explicit form of the minimizers of $R_{reg}[f]$.

Theorem 2 (Representer Theorem). *Let $D_m = \{(x_i, y_i) \in \mathscr{X} \times \mathbb{R}\}_{i=1}^{m}$ be a training set, V a convex loss function, $\mathscr{H}$ a RKHS with norm $||\cdot||_{\mathscr{H}}$. Then the general form of the solution of the regularized risk*

$$\frac{1}{m} \sum_{i=1}^{m} V(f(x_i), y_i) + \lambda ||f||^2_{\mathscr{H}}$$

[2] An alternative way of constructing a feature space corresponding to a positive definite kernel is provided by Mercer's Theorem [53].

is

$$f(x) = \sum_{i=1}^{m} c_i k(x_i, x). \tag{4}$$

The proof is omitted for brevity and can be found in [41]. The theorem states that regardless of the dimension of the RKHS $\mathscr{H}$, the solution lies on the span of the m kernels centered on the training points. Generalization of the representer theorem have been proved [68] for arbitrary cumulative loss functions $V((x_1, y_1, f(x_1)), \ldots, (x_m, y_m, f(x_m)))$ and strictly monotonic regularization functionals. A semi-parametric version of the theorem accounts for slightly more general solutions, including for instance a constant bias term.

3 Kernel Machines for Structured Input

Most supervised kernel machines can be seen as instantiations of the Tikhonov regularization framework for a particular choice of the loss function V. Kernel ridge regression [65, 59] employs the quadratic loss $V(f(x), y) = (f(x) - y)^2$ and is used for both regression and classification tasks. Kernel logistic regression [39] uses the negative log-likelihood of the probabilistic model, i.e. $\log(1 + \exp(-yf(x)))$ for binary classification. Support Vector Machines [13] (SVM) are the most popular class of kernel methods. Initially introduced for binary classification [9], they have been extended to deal with different tasks such as regression and multiclass classification. The common rationale of SVM algorithms is the use of a loss function forcing *sparsity* in the solution. That is, only a small subset of the c_i coefficients in eq. (4) will be non-zero. The corresponding training examples are termed *Support Vectors* (SV). In the following we detail SVM for binary classification and regression. SVM for multiclass classification will arise as a special case of structured-output prediction (see Section 6). We will then discuss a SV approach for novelty detection based on the estimation of the smallest enclosing hypersphere. Finally we will introduce kernel Principal Component Analysis [67] for non-linear dimensionality reduction. Additional algorithms can be found e.g. in [71].

3.1 SVM for Binary Classification

SVM for binary classification employ the so-called *hinge loss*:

$$V(f(x), y) = |1 - yf(x)|_+ = \begin{cases} 0 & \text{if } yf(x) \geq 1 \\ 1 - yf(x) & \text{otherwise} \end{cases}$$

As shown in Fig.2(a), a linear cost is paid in case the confidence in the correct class is below a certain threshold. By plugging the hinge loss in the Tikhonov regularization functional we obtain the optimization problem addressed by SVM:

$$\min_{f \in \mathscr{H}} \frac{1}{m} \sum_{i=1}^{m} |1 - y_i f(x_i)|_+ + \lambda ||f||^2_{\mathscr{H}}.$$

Slack variables $\xi_i = |1 - y_i f(x_i)|_+$ can be used to represent the cost paid for each example, giving the following quadratic optimization problem:

$$\begin{aligned} \min_{f \in \mathscr{H}, \boldsymbol{\xi} \in \mathrm{I\!R}^m} \quad & \frac{1}{m} \sum_{i=1}^{m} \xi_i + \lambda ||f||^2_{\mathscr{H}} \\ \text{subject to:} \quad & y_i f(x_i) \geq 1 - \xi_i \quad i = 1, \ldots, m \\ & \xi_i \geq 0 \quad i = 1, \ldots, m. \end{aligned}$$

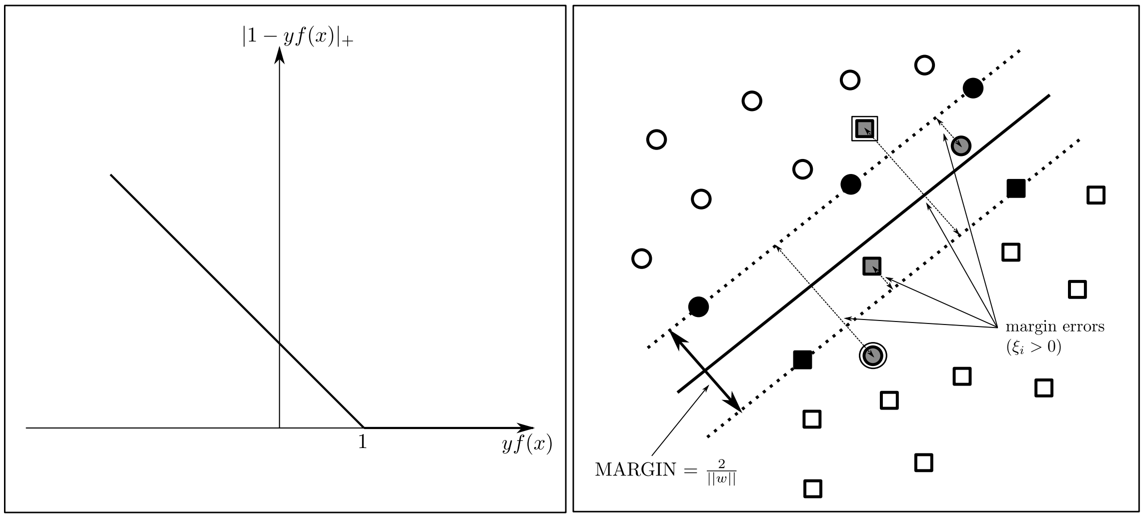


Fig. 2 SVM for binary classification: (a) Hinge loss. (b) Classification function. The solid line represents the separating hyperplane, while dotted lines are hyperplanes with confidence margin equal to one. Black points are unbound SVs, grey points are bound SVs and extra borders indicate bound SVs which are also training errors. All other points do not contribute to the function to be minimized. Dotted lines indicate the margin error ξ_i for bound SVs.

To see that this corresponds to the hinge loss, note that as we minimize over slack variables $\boldsymbol{\xi}$, ξ_i will be zero (it must be non-negative) if $y_i f(x_i) \geq 1$ (hard constraint satisfied) and $1 - y_i f(x_i)$ otherwise. By the representer Theorem we know that the solution of the above problem is given by eq. (4). As for the simple classification algorithm seen in Section 1, the decision function takes the sign of f to predict labels and the decision boundary is a separating hyperplane in the feature space. To see this, let $\Phi(\cdot)$ be a feature mapping associated with kernel k. Function f can be rewritten as:

$$f(x) = \sum_{i=1}^{m} c_i \langle \Phi(x_i), \Phi(x) \rangle = \langle \sum_{i=1}^{m} c_i \Phi(x_i), \Phi(x) \rangle = \langle \boldsymbol{w}, \Phi(x) \rangle.$$

The decision boundary $\langle \boldsymbol{w}, \Phi(x) \rangle = 0$ is a hyperplane of points orthogonal to $\boldsymbol{w}$. Note that $\boldsymbol{w}$ can be explicitly computed only if the feature mapping $\Phi(\cdot)$ is finite-dimensional. From the definition of the dot product in the RKHS $\mathscr{H}$ (see eq. (2)) we can compute the (squared) norm of f:

$$\begin{aligned} ||f||^2_{\mathscr{H}} = \langle f, f \rangle = \sum_{i=1}^{m} \sum_{j=1}^{m} c_i c_j k(x_i, x_j) &= \sum_{i=1}^{m} \sum_{j=1}^{m} c_i c_j \langle \Phi(x_i), \Phi(x_j) \rangle \\ = \langle \sum_{i=1}^{m} c_i \Phi(x_i), \sum_{j=1}^{m} c_j \Phi(x_j) \rangle &= \langle \boldsymbol{w}, \boldsymbol{w} \rangle = ||\boldsymbol{w}||^2. \end{aligned}$$

The minimization problem can be rewritten as:

$$\begin{aligned} \min_{\boldsymbol{w} \in \mathscr{H}, \boldsymbol{\xi} \in \mathrm{I\!R}^m} \quad & C \sum_{i=1}^{m} \xi_i + \frac{1}{2} ||\boldsymbol{w}||^2 \\ \text{subject to:} \quad & y_i \langle \boldsymbol{w}, \Phi(x_i) \rangle \geq 1 - \xi_i \quad i = 1, \ldots, m \\ & \xi_i \geq 0 \quad i = 1, \ldots, m, \end{aligned}$$

where we replaced $C = 2/\lambda m$ for consistency with most literature on SVM. Hyperplanes $\langle \boldsymbol{w}, \Phi(x) \rangle - 1 = 0$ and $\langle \boldsymbol{w}, \Phi(x) \rangle + 1 = 0$ are "confidence" boundaries for not paying a cost in predicting class $+1$ and -1 respectively. The distance between them $2/||w||$ is called geometric margin. By minimizing $||\boldsymbol{w}||^2$, this margin is maximized, while slack variables ξ_i account for margin errors. The minimizer thus trades off margin maximization and fitting of training data. Constraints in the optimization problem can be included in the minimization functional using Lagrange multipliers:

$$L(\boldsymbol{w},\boldsymbol{\alpha},\boldsymbol{\beta}) = C\sum_{i=1}^{m}\xi_i + \frac{1}{2}||\boldsymbol{w}||^2 - \sum_{i=1}^{m}\alpha_i(y_i\langle \boldsymbol{w},\Phi(x_i)\rangle - 1 + \xi_i) - \sum_{i=1}^{m}\beta_i\xi_i$$

where $\alpha_i, \beta_i \geq 0$ for all i. The Wolfe dual formulation for the Lagrangian amounts at maximizing it over α_i, β_i subject to the vanishing of the gradient of $\boldsymbol{w}$ and $\boldsymbol{\xi}$, i.e.:

$$\frac{\partial L}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_{i=1}^{m}\alpha_i y_i \Phi(x_i) = 0 \rightarrow \boldsymbol{w} = \sum_{i=1}^{m}\alpha_i y_i \Phi(x_i) \tag{5}$$

$$\frac{\partial L}{\partial \xi_i} = C - \alpha_i - \beta_i = 0 \rightarrow \alpha_i \in [0,C]. \tag{6}$$

The second implication comes from the non-negativity of both α_i and β_i. Substituting into the Lagrangian we obtain:

$$\max_{\boldsymbol{\alpha}\in\mathrm{I\!R}^m} \; -\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i y_i \alpha_j y_j \langle \Phi(x_i),\Phi(x_j)\rangle + \sum_{i=1}^{m}\alpha_i$$
$$\text{subject to:} \quad \alpha_i \in [0,C] \quad i = 1,\dots,m.$$

The problem can be solved using off-the-shelf quadratic programming tools. However, a number of ad-hoc algorithms have been proposed which exploit the specific characteristics of this problem to achieve substantial efficiency improvements [37, 58]. Note that replacing $\langle \Phi(x_i),\Phi(x_j)\rangle = k(x_i,x_j)$ we recover the kernel-based formulation where the feature mapping is only implicitly done. The general form for f (eq. (4)) can be recovered setting $c_i = \alpha_i y_i$ (see eq. (5)).

The Karush−Kuhn−Tucker (KKT) complementary conditions require that the optimal solution satisfies:

$$\alpha_i(y_i\langle \boldsymbol{w},\Phi(x_i)\rangle - 1 + \xi_i) = 0 \tag{7}$$

$$\beta_i\xi_i = 0 \tag{8}$$

for all i. Eq. (7) implies that $\alpha_i > 0$ only for examples where $y_i\langle \boldsymbol{w},\Phi(x_i)\rangle \leq 1$. These are the support vectors, all other examples do not contribute to the decision function f. If $\alpha_i < C$, equations (8) and (6) imply that $\xi_i = 0$. These are called *unbound* support vectors and lay on the confidence one hyperplanes. *Bound* support vectors ($\alpha_i = C$) are margin errors ($\xi_i > 0$). Figure 2(b) shows an example highlighting hyperplanes and support vectors.

Variants of SVM for binary classification have been developed in the literature. Most approaches include a bias term b to the classification function f. This can be obtained simply setting $k'(x,x') = k(x,x') + 1$. Linear penalties can be replaced with quadratic ones (ξ_i^2) in the minimization functional. The ν-SVM [70] allows to explicitly upper bound the number of margin errors. Further details can be found in several textbooks (see e.g. [13]).

3.2 SVM for Regression

SVM for regression enforce sparsity in the solution by tolerating small deviations from the desired target. This is achieved by the $\varepsilon - insensitive$ loss (see fig.3(a)):

$$V(f(x),y) = |y - f(x)|_\varepsilon = \begin{cases} 0 & \text{if } |y - f(x)| \leq \varepsilon \\ |y - f(x)| - \varepsilon & \text{otherwise} \end{cases}$$

which doesn't penalize deviations up to ε from the target value (the so-called *ε-tube*), and gives a linear penalty to further deviations. By introducing slack variables for penalties, we obtain the following minimization problem:

$$\begin{aligned} \min_{f \in \mathscr{H}, \boldsymbol{\xi}, \boldsymbol{\xi}^* \in \mathbb{R}^m} \quad & \frac{1}{m} \sum_{i=1}^{m} (\xi_i + \xi_i^*) + \lambda ||f||^2_{\mathscr{H}} \\ \text{subject to:} \quad & f(x_i) - y_i \leq \varepsilon + \xi_i \quad i = 1, \ldots, m \\ & y_i - f(x_i) \leq \varepsilon + \xi_i^* \quad i = 1, \ldots, m \\ & \xi_i, \xi_i^* \geq 0 \quad i = 1, \ldots, m. \end{aligned}$$

As for the binary classification case, we can rewrite the problem in terms of weight vector and feature mapping. The Lagrangian is obtained as ($C = 2/\lambda m$):

$$\begin{aligned} L(\mathbf{w}, \boldsymbol{\xi}, \boldsymbol{\xi}^*, \boldsymbol{\alpha}, \boldsymbol{\alpha}^*, \boldsymbol{\beta}, \boldsymbol{\beta}^*) = \frac{1}{2} ||\mathbf{w}||^2 + C \sum_{i=1}^{m} (\xi_i + \xi_i^*) - \sum_{i=1}^{m} (\beta_i \xi_i + \beta_i^* \xi_i^*) - \sum_{i=1}^{m} \alpha_i (\varepsilon + \xi_i + y_i - \langle \mathbf{w}, \Phi(x_i) \rangle) \\ - \sum_{i=1}^{m} \alpha_i^* (\varepsilon + \xi_i^* - y_i + \langle \mathbf{w}, \Phi(x_i) \rangle) \quad (9) \end{aligned}$$

with $\alpha_i, \alpha_i*, \beta_i, \beta_i^* \geq 0$, $\forall i \in [1, m]$. By vanishing the derivatives of L with respect to the primal variables we obtain:

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{w}} &= \mathbf{w} - \sum_{i=1}^{m} (\alpha_i^* - \alpha_i) \Phi(x_i) = 0 \rightarrow \mathbf{w} = \sum_{i=1}^{m} (\alpha_i^* - \alpha_i) \Phi(x_i) && (10) \\ \frac{\partial L}{\partial \xi_i} &= C - \alpha_i - \beta_i = 0 \rightarrow \alpha_i \in [0, C] && (11) \\ \frac{\partial L}{\partial \xi_i^*} &= C - \alpha_i^* - \beta_i^* = 0 \rightarrow \alpha_i^* \in [0, C]. \end{aligned}$$

Finally, substituting into the Lagrangian we derive the dual problem:

$$\begin{aligned} \max_{\boldsymbol{\alpha} \in \mathbb{R}^m} \quad & -\frac{1}{2} \sum_{i,j=1}^{m} (\alpha_i^* - \alpha_i)(\alpha_j^* - \alpha_j) \langle \Phi(x_i), \Phi(x_j) \rangle - \varepsilon \sum_{i=1}^{m} (\alpha_i^* + \alpha_i) + \sum_{i=1}^{m} y_i (\alpha_i^* - \alpha_i), \\ \text{subject to:} \quad & \alpha_i, \alpha_i^* \in [0, C], \quad \forall i \in [1, m]. \end{aligned}$$

The kernel-based formulation is again recovered setting $\langle \Phi(x_i), \Phi(x_j) \rangle = k(x_i, x_j)$, while the general form for f (eq. (4)) is obtained for $c_i = \alpha_i - \alpha_i^*$. (see eq. (10)). The KKT complementary conditions require that the optimal solution satisfies:

$$\begin{aligned} \alpha_i (\varepsilon + \xi_i + y_i - \langle \mathbf{w}, \Phi(x_i) \rangle) &= 0 \\ \alpha_i^* (\varepsilon + \xi_i^* - y_i + \langle \mathbf{w}, \Phi(x_i) \rangle) &= 0 \\ (C - \alpha_i) \xi_i &= 0 \\ (C - \alpha_i^*) \xi_i^* &= 0 \end{aligned}$$

These conditions enlighten some interesting analogies to the classification case:

- All patterns within the ε-tube, for which $|f(\mathbf{x}_i) - y_i| < \varepsilon$, have $\alpha_i, \alpha_i^* = 0$ and thus don't contribute to the estimated function f.
- Patterns for which either $0 < \alpha_i < C$ or $0 < \alpha_i^* < C$ are on the border of the ε-tube, that is $|f(\mathbf{x}_i) - y_i| = \varepsilon$. They are the unbound support vectors.
- The remaining training patterns are margin errors (either $\xi_i > 0$ or $\xi_i^* > 0$), and reside out of the ε-insensitive region. They are bound support vectors, with corresponding $\alpha_i = C$ or $\alpha_i^* = C$.

Figures 3(b),3(c),3(d) show examples of SVM regression for decreasing values of ε. In order to highlight the effect of the parameter on the approximation function, we focus on 1D regression and report the form of the function in the input (rather than feature) space. Note the increase in the number of support vectors when requiring tighter approximations. A Gaussian kernel (see Section 4.1) was employed in all cases.

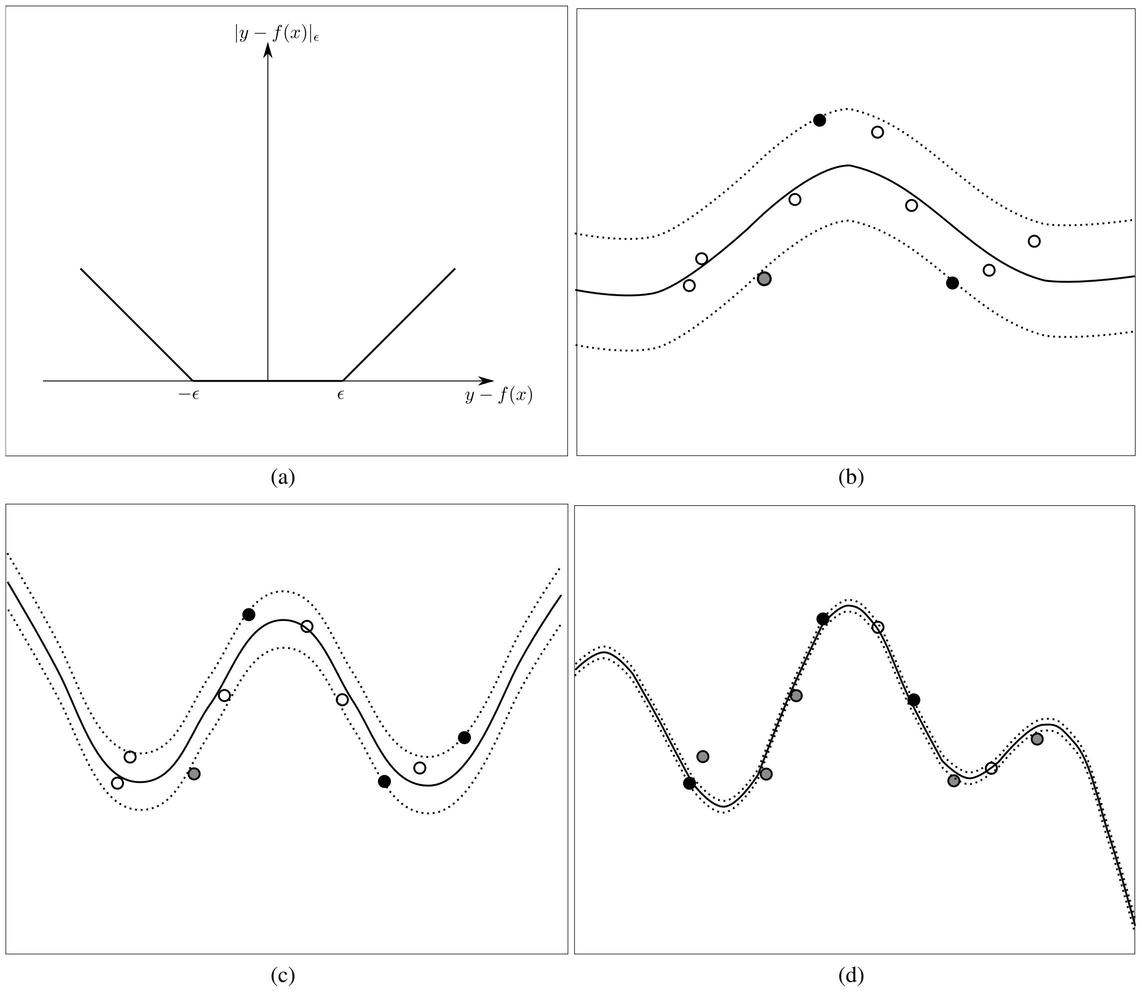


Fig. 3 SVM for regression: (a) Epsilon insensitive loss. (b)(c)(d) Regression functions for decreasing values of ε. Solid lines represent the regression function, dotted lines represent the ε-insensitive tube around the regression function. All points within this tube are considered correctly approximated. Black points are unbound SVs laying on the borders of the tube, gray points are bound SVs laying outside of it. All other points do not contribute to the regression function. Note the increase in the complexity of the function and the number of support vectors for decreasing values of ε.

3.3 Smallest Enclosing Hypersphere

A support vector algorithm has been proposed in [75, 66] in order to characterize a set of data in terms of support vectors, thus allowing to compute a set of contours which enclose the data points. The idea is finding the smallest hypersphere which encloses the points in the feature space. Outliers can be dealt with by relaxing the enclosing constraint and allowing some points to stay out of the sphere in feature space. The algorithm can be readily employed for novelty detection, by predicting whether a test instance lays outside of the enclosing hypersphere.

Given a set of m examples $x_i \in \mathscr{X}, i = 1,\ldots,m$ and a feature mapping Φ , we can define the problem of finding the smallest enclosing sphere of radius R in the feature space as follows:

$$\begin{aligned} \min_{R\in\mathbb{R},\mathbf{o}\in\mathscr{H},\boldsymbol{\xi}\in\mathbb{R}^m} \quad & R^2 + C\sum_{i=1}^m \xi_i \\ \text{subject to} \quad & ||\Phi(x_i) - \mathbf{o}||^2 \le R^2 + \xi_i, \quad i = 1,\ldots,m \\ & \xi_i \ge 0, \quad i = 1,\ldots,m \end{aligned}$$

where $\mathbf{o}$ is the center of the sphere, ξ_i are slack variables allowing for soft constraints and C is a cost parameter balancing the radius of the sphere versus the number of outliers. We consider the Lagrangian

$$L(R,\mathbf{o},\boldsymbol{\xi},\boldsymbol{\alpha},\boldsymbol{\beta}) = R^2 + C\sum_{i=1}^m \xi_i - \sum_{i=1}^m \alpha_i(R^2 + \xi_i - ||\Phi(x_i) - \mathbf{o}||^2) - \sum_{i=1}^m \beta_i\xi_i,$$

with $\alpha_i \ge 0$ and $\beta_i \ge 0$ for all $i \in [1,m]$, and by vanishing the derivatives with respect to the primal variables R, $\mathbf{o}$ and ξ_i we obtain

$$\begin{aligned} \frac{\partial L}{\partial R} &= 1 - \sum_{i=1}^m \alpha_i = 0 \to \sum_{i=1}^m \alpha_i = 1 \\ \frac{\partial L}{\partial \mathbf{o}} &= \mathbf{o} - \sum_{i=1}^m \alpha_i\Phi(x_i) = 0 \to \mathbf{o} = \sum_{i=1}^m \alpha_i\Phi(x_i) \qquad (12) \\ \frac{\partial L}{\partial \xi_i} &= C - \alpha_i - \beta_i = 0 \to \alpha_i \in [0,C]. \end{aligned}$$

Substituting into the Lagrangian we derive the Wolf dual problem

$$\begin{aligned} \max_{\boldsymbol{\alpha}\in\mathbb{R}^m} \quad & \sum_{i=1}^m \alpha_i\Phi(x_i)^2 - \sum_{i=1}^m\sum_{j=1}^m \alpha_i\alpha_j\langle\Phi(x_i),\Phi(x_j)\rangle, \qquad (13) \\ \text{subject to} \quad & \sum_{i=1}^m \alpha_i = 1, \quad 0 \le \alpha_i \le C, \quad i = 1,\ldots,m. \end{aligned}$$

The distance of a given point x from the center of the sphere

$$R^2(x) = ||\Phi(x) - \mathbf{o}||^2$$

can be written using (12) as

$$R^2(x) = \langle\Phi(x),\Phi(x)\rangle - 2\sum_{i=1}^m \alpha_i\langle\Phi(x),\Phi(x_i)\rangle + \sum_{i=1}^m\sum_{j=1}^m \alpha_i\alpha_j\langle\Phi(x_i),\Phi(x_j)\rangle. \qquad (14)$$

As for the other SV algorithms, both (13) and (14) contain only dot products in the feature space, which can be substituted by a kernel function k. The KKT conditions [17] imply that at the saddle point of the Lagrangian

$$\beta_i \xi_i = 0$$
$$\alpha_i(R^2 + \xi_i - ||\Phi(x_i) - \mathbf{o}||^2) = 0$$

showing the presence of (see fig. 4):

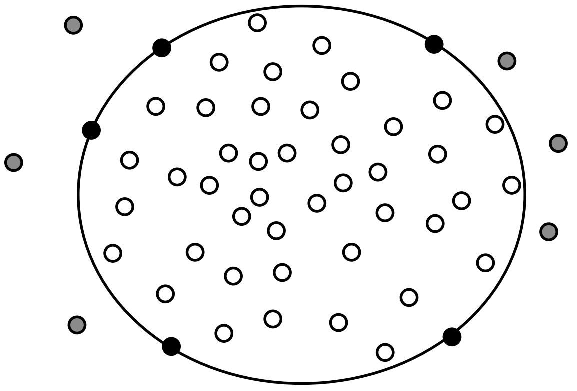


Fig. 4 Novelty detection by smallest enclosing hypersphere. The solid line is the border of the hypersphere enclosing most of the data and represents the decision boundary. Black points are unbound SV laying on the hypersphere border. Grey points are outliers which are left outside of the hypersphere. All other points do not contribute to the decision function.

- Unbound support vectors ($0 < \alpha_i < C$), whose images lie on the surface of the enclosing sphere.
- Bound support vectors ($\alpha_i = C$), whose images lie outside of the enclosing sphere, which correspond to outliers.
- All other points ($\alpha = 0$) with images inside the enclosing sphere.

The radius R^* of the enclosing sphere can be computed by (14) provided x is an unbound support vector. A decision function for novelty detection would predict a point as positive if it lays outside of the sphere and negative otherwise, i.e.:

$$f(x) = \text{sgn}\left(R^2(x) - (R^*)^2\right)$$

3.4 Kernel Principal Component Analysis

Principal Component Analysis (PCA) is a standard technique for linear dimensionality reduction which consists of projecting examples onto directions of maximal variance, thus retaining most of their information content. We will introduce it considering explicit feature mappings $\Phi(x)$, and then show how these can be computed only implicitly via kernels (see [71] for further details).

Given a set of orthonormal vectors $\boldsymbol{u}_1, \dots, \boldsymbol{u}_k$, the orthogonal projection of a point $\Phi(x)$ into the subspace V spanned by them is computed as:

$$P_V(\Phi(x)) = \Big(\langle \boldsymbol{u}_i, \Phi(x) \rangle\Big)_{i=1}^{k} \tag{15}$$

where each dot product computes the length of the projection in the corresponding direction. Let X be a matrix representation of a set of points $\{\Phi(x_1), \dots, \Phi(x_m)\}$, with row i representing point $\Phi(x_i)^T$. Let's assume that the points are centered around the origin, i.e. their mean is zero. This can always be obtained subtracting the mean from each point. The covariance matrix of the points C is computed as:

$$C = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{m} \Phi(x_i)\Phi(x_j)^T = \frac{1}{m}X^T X.$$

The directions of maximal variance are the eigenvectors of C with maximal eigenvalue. PCA basically projects points onto the first k eigenvectors of the covariance matrix, where k is the dimension of the reduced space. The dimension k can be set a priori or chosen according to the amount of variance captured, measured as the sum of the eigenvalues.

Standard PCA requires to explicitly use mappings $\Phi(x)$, both in computing covariance matrix C and in doing the projection (Eq. (15)). However, it is possible to avoid this thanks to the relationship between the eigen-decomposition of covariance and kernel matrices. Let $K = XX^T$ be the kernel matrix of the data. Let $(\lambda, \boldsymbol{v})$ be an eigenvalue-eigenvector pair of K. It holds that:

$$C(X^T\boldsymbol{v}) = \frac{1}{m}X^TXX^T\boldsymbol{v} = \frac{1}{m}X^TK\boldsymbol{v} = \frac{\lambda}{m}X^T\boldsymbol{v}$$

showing that $(\lambda/m, X^T\boldsymbol{v})$ is an eigenvalue-eigenvector pair of C. The norm of the eigenvector is:

$$||X^T\boldsymbol{v}||^2 = \boldsymbol{v}^TXX^T\boldsymbol{v} = \boldsymbol{v}^TK\boldsymbol{v} = \lambda\boldsymbol{v}^T\boldsymbol{v} = \lambda$$

where the last equality follows from the orthonormality of $\boldsymbol{v}$. The normalized eigenvector is thus $\boldsymbol{u} = 1/\sqrt{\lambda}X^T\boldsymbol{v}$. Projecting $\Phi(x)$ onto this direction can be computed as:

$$P_{\boldsymbol{u}}(\Phi(x)) = \langle 1/\sqrt{\lambda}X^T\boldsymbol{v}, \Phi(x)\rangle = 1/\sqrt{\lambda}\boldsymbol{v}^TX\Phi(x) = 1/\sqrt{\lambda}\sum_{i=1}^{m}\boldsymbol{v}_i\langle\Phi(x_i),\Phi(x)\rangle = 1/\sqrt{\lambda}\sum_{i=1}^{m}\boldsymbol{v}_ik(x_i,x)$$

where k is a kernel corresponding to the feature map $\Phi(\cdot)$. Note that $\boldsymbol{v}$ is computed as eigenvector of the kernel matrix K, thus we never need to explicitly compute $\Phi(x)$. Centering of the data in feature space can also be addressed simply using a modified kernel:

$$\begin{aligned}\hat{k}(x,x') &= \langle\Phi(x) - \frac{1}{m}\sum_{i=1}^{m}\Phi(x_i), \Phi(x') - \frac{1}{m}\sum_{i=1}^{m}\Phi(x_i)\rangle \\ &= k(x,x') - \frac{1}{m}\sum_{i=1}^{m}k(x_i,x') - \frac{1}{m}\sum_{i=1}^{m}k(x,x_i) + \frac{1}{m^2}\sum_{i=1}^{m}\sum_{j=1}^{m}k(x_i,x_j).\end{aligned}$$

A number of well-known dimensionality reduction techniques, like Locally Linear Embedding and Laplacian Eigenmaps, can be seen [28] as special cases of kernel PCA. Kernel PCA has been used for addressing a variety of problems, from novelty detection [30] to image denoising [40].

4 Kernels on Structured Data

Kernel design deals with the problem of choosing an appropriate kernel for the task at hand, that is a similarity measure of the data capable of best capturing the available information. We start by introducing a few basic kernels commonly used in practice, and show how to realize complex kernels by combination of simpler ones, allowing to treat different parts or characteristics of the input data in different ways. We will introduce the notion of kernels on discrete structures, providing examples of kernels for strings, trees and graphs. We will then discusses two classes of hybrid kernels, based on probabilistic generative models and logical formalisms respectively. For extensive treatments of kernels for structured data see also [22, 51].

4.1 Basic Kernels

Let's start with some basic kernels on inner product spaces. While they cannot be directly applied to structured objects, they will turn useful when defining complex kernels as combinations of simpler ones. The standard dot product is called linear kernel:

$$k(x,x') = \langle x,x' \rangle.$$

Its normalized version computes the cosine of the angle between the two vectors:

$$k_{norm}(x,x') = \frac{\langle x,x' \rangle}{\sqrt{\langle x,x \rangle \langle x',x' \rangle}}. \tag{16}$$

The *polynomial* kernel:

$$k_d(x,x') \;=\; (\langle x,x' \rangle + c\,)^d \tag{17}$$

with $d \in \mathbb{N}$ and $c \in \mathbb{R}_0^+$, allows to combine individual features taking their products. The corresponding feature space contains all possible monomials of degree up to d. Gaussian kernels are defined as:

$$k_\sigma(x,x') = \exp\left(-\frac{||x-x'||^2}{2\sigma^2}\right) = \exp\left(-\frac{\langle x,x \rangle - 2\langle x,x' \rangle + \langle x',x' \rangle}{2\sigma^2}\right) \tag{18}$$

with $\sigma > 0$. They are an example of *Universal* kernels [54], a class of kernels which can uniformly approximate any arbitrary continuous target function. Note that the smallest the variance σ^2, the most the prediction for a certain point will depend only on its nearest (training) neighbours, eventually leading to orthogonality between any pair of points and poor generalization. Tuning this *hyperparameter* according to the complexity of the problem at hand is another mean to control overfitting (see also Section 5).

The simplest possible kernel for arbitrary domains is the *matching* or *delta* kernel:

$$k_\delta(x,x') = \delta(x,x') = \begin{cases} 1 \text{ if } x = x' \\ 0 \text{ otherwise.} \end{cases} \tag{19}$$

While it clearly does not allow any generalization if used alone, it is another useful component for building more complex kernels.

4.2 Kernel Combination

The class of kernels has a few interesting closure properties useful for combinations. It is closed under addition, product, multiplication by a positive constant and pointwise limits [6], that is they form a closed convex cone. Note that the addition of two kernels corresponds to the concatenation of their respective features:

$$\begin{aligned}(k_1 + k_2)(x,x') &= k_1(x,x') + k_2(x,x') \\ &= \langle \Phi_1(x), \Phi_1(x') \rangle + \langle \Phi_2(x), \Phi_2(x') \rangle \\ &= \langle \Phi_1(x) \odot \Phi_2(x), \Phi_1(x') \odot \Phi_2(x') \rangle\end{aligned}$$

where $\odot$ denotes vector concatenation. Taking the product of two kernels amounts at taking the Cartesian product between their respective features:

$$
\begin{aligned}
(k_1 \times k_2)(x,x') &= k_1(x,x')k_2(x,x') \\
&= \sum_{i=1}^{n} \Phi_{1i}(x)\Phi_{1i}(x') \sum_{j=1}^{m} \Phi_{2j}(x)\Phi_{2j}(x') \\
&= \sum_{i=1}^{n}\sum_{j=1}^{m} (\Phi_{1i}(x)\Phi_{2j}(x))(\Phi_{1i}(x')\Phi_{2j}(x')) \\
&= \sum_{k=1}^{nm} \Phi_{12k}(x)\Phi_{12j}(x') = \langle \Phi_{12}(x), \Phi_{12}(x') \rangle
\end{aligned}
$$

where $\Phi_{12}(x) = \Phi_1(x) \times \Phi_2(x)$. Such properties are still valid in the case that the two kernels are defined on different domains [29]. If k_1 and k_2 are kernels defined respectively on $\mathscr{X}_1 \times \mathscr{X}_1$ and $\mathscr{X}_2 \times \mathscr{X}_2$, then their *direct sum* and *tensor product*:

$$
\begin{aligned}
(k_1 \oplus k_2)((x_1,x_2),(x_1',x_2')) = k_1(x_1,x_1') + k_2(x_2,x_2') \\
(k_1 \otimes k_2)((x_1,x_2),(x_1',x_2')) = k_1(x_1,x_1')k_2(x_2,x_2')
\end{aligned}
$$

are kernels on $(\mathscr{X}_1 \times \mathscr{X}_2) \times (\mathscr{X}_1 \times \mathscr{X}_2)$, with $x_1,x_1' \in \mathscr{X}_1$ and $x_2,x_2' \in \mathscr{X}_2$. These combinations allow to treat in a diverse way parts of an individual which have different meanings. Finally, if k is defined on $\mathscr{S} \times \mathscr{S}$ with $\mathscr{S} \subset \mathscr{X}$, a *zero extension* kernel on $\mathscr{X} \times \mathscr{X}$ can be obtained setting $k(x,x') = 0$ whenever x or x' do not belong to $\mathscr{S}$.

These concepts are at the basis of the so called *convolution* kernels [29, 85] for discrete structures. Suppose $x \in \mathscr{X}$ is a composite structure made of "parts" $x_1,\ldots,x_D$ such that $x_d \in \mathscr{X}_d$ for all $i \in [1,D]$. This can be formally represented by a relation R on $\mathscr{X}_1 \times \cdots \times \mathscr{X}_D \times \mathscr{X}$ such that $R(x_1,\ldots,x_D,x)$ is true iff $x_1,\ldots,x_D$ are the parts of x. For example if $\mathscr{X}_1 = \cdots = \mathscr{X}_D = \mathscr{X}$ are sets containing all finite strings over a finite alphabet $\mathscr{A}$, we can define a relation $R(x_1,\ldots,x_D,x)$ which is true iff $x = x_1 \circ \cdots \circ x_D$, with $\circ$ denoting concatenation of strings. Note that in this example x can be decomposed in multiple ways. If their number is finite, the relation is said to be finite. Let $R^{-1}(x) = \{x_1,\ldots,x_D : R(x_1,\ldots,x_D,x)\}$ return the set of decompositions for x. Given a set of kernels $k_d : \mathscr{X}_d \times \mathscr{X}_d \to \mathbb{R}$, one for each of the parts of x, the *R-convolution* kernel is defined as

$$
(k_1 \star \cdots \star k_D)(x,x') = \sum_{(x_1,\ldots,x_D)\in R^{-1}(x)} \; \sum_{(x_1',\ldots,x_D')\in R^{-1}(x')} \prod_{d=1}^{D} k_d(x_d,x_d') \tag{20}
$$

where the sums run over all the possible decompositions of x and x'. For finite relations R, this can be shown to be a valid kernel [29]. Let X,X' be sets and let $R(\xi,X)$ be the membership relation, i.e. $\xi \in R^{-1}(X) \iff \xi \in X$. The *set kernel* is defined as:

$$
k_{set}(X,X') = \sum_{\xi \in X} \sum_{\xi' \in X'} k_{member}(\xi,\xi') \tag{21}
$$

where k_{member} is a kernel on set elements. Simple examples of set kernels include the *intersection kernel*:

$$
k_{\cap}(X,X') = |X \cap X'|
$$

and the *Tanimoto kernel*:

$$
k_{Tanimoto}(X,X') = \frac{|X \cap X'|}{|X \cup X'|}.
$$

A more complex type of R-convolution kernel is the so called *analysis of variance* (ANOVA) kernel [83]. Let $\mathscr{X} = \mathscr{S}^n$ be the set of n-sized tuples built over elements of a certain set $\mathscr{S}$. Let $k_i : \mathscr{S} \times \mathscr{S} \to \mathbb{R}$, $i \in [1,n]$ be a set of kernels, which will typically be the same function. For $D \in [1,n]$, the ANOVA kernel of order D, $k_{Anova} : \mathscr{S}^n \times \mathscr{S}^n \to \mathbb{R}$, is defined by

$$k_{Anova}(x,x') = \sum_{1 \le i_1 < \cdots < i_D \le n} \prod_{d=1}^{D} k_{i_d}(x_{i_d}, x'_{i_d}).$$

Note that the sum ranges over all possible subsets of cardinality D. For $D = n$, the sum consists only of the term for which $(i_1 = 1, \ldots, i_D = n)$, and k becomes the tensor product $k_1 \otimes \cdots \otimes k_n$. Conversely, for $D = 1$, each product collapses to a single factor, while i_1 ranges from 1 to n, giving the direct sum $k_1 \oplus \cdots \oplus k_n$. By varying D we can run between these two extremes. In order to reduce the computational cost of kernel evaluations, recursive procedures are usually employed [80].

Mapping kernels [72] were recently introduced as a generalization of R-convolution kernels, in which the kernel sums over a subset of all possible decompositions of x and x'. If the *mapping system* $M_{x,x'}$ selecting the subset is *transitive*, i.e. $(x_d, x'_d) \in M_{x,x'} \wedge (x'_d, x''_d) \in M_{x',x''} \Rightarrow (x_d, x''_d) \in M_{x,x''}$, then the resulting mapping kernel is positive definite.

Finally, once a kernel k on an arbitrary type of data has been defined, it can readily be composed with basic kernels on inner-product spaces, like the ones seen in the previous section (just set $\langle x, x' \rangle = k(x,x')$). This will produce an overall feature mapping which is the composition of the mappings $\Phi(\cdot)$ and $\Phi'(\cdot)$ associated to the two kernels:

$$\Phi^* : \mathscr{X} \to \mathscr{H}' \mid \Phi^* = \Phi' \circ \Phi.$$

Polynomial (eq. 17) and Gaussian (eq. 18) kernels are commonly employed to allow for nonlinear combinations of features in the mapped space of the first kernel. Cosine normalization (eq. 16) is often used to reduce the dependence on the size of the objects. In the case of set kernels, an alternative is that of dividing by the product of the cardinalities of the two sets, thus computing the mean value between pairwise comparisons:

$$k_{mean}(X,X') = \frac{k_{set}(X,X')}{|X||X'|}.$$

4.3 Kernels on Discrete Structures

R-convolution and mapping kernels are very general classes of kernels, which can be used to model similarity between objects with discrete structures, such as strings, trees and graphs. A large number of kernels on structures have been defined in the literature, mostly as instantiations of these kernel classes. A very common approach consists of defining similarity in terms of counts of common substructures. However, efficiency is a key issue in order to develop kernels of practical utility. A kernel can also be thought of as a procedure efficiently implementing a given dot product in feature space. In the following, we will report a series of kernels developed for efficiently treating objects with discrete structure. We do not aim at providing an exhaustive enumeration of all kernels on structured data developed in the literature. We will focus on some representative approaches, whose description should help in figuring out how kernel design works, while providing pointers to additional literature. In most of the cases, we will explicitly show a feature space corresponding to the kernel. This is one of the most common ways of proving that a kernel is positive definite. An alternative is showing that it belongs to a class of known valid kernels, like the up-mentioned *R*-convolution and mapping ones.

4.3.1 Strings

Strings allow to represent data consisting of sequences of discrete symbols. They account for variable length objects in which the ordering of the elements matters. Biological sequences, for instance, can be represented as strings of symbols, amino-acids for proteins or nucleotides for DNA and RNA. Text documents can be represented as strings of characters. We introduce some notation before describing a number of common kernels for strings.

Consider a finite alphabet $\mathscr{A}$. A string s is a finite sequence of (possibly zero) characters from $\mathscr{A}$. We define by $|s|$ the length of string s, $\mathscr{A}^n$ the set of all strings of length n, and

$$\mathscr{A}^* = \bigcup_{n=0}^{\infty} \mathscr{A}^n$$

the set of all strings. Concatenation between strings s and t is simply represented as st. A (possibly non-contiguous) subsequence u of s is defined as $u := s(\mathbf{i}) := s(i_1)\dots s(i_{|u|})$, with $1 \le i_1 < \dots < i_{|u|} \le |n|$ and $s(i)$ the i^{th} element of s. The length $l(\mathbf{i})$ of the subsequence u in s is $i_{|u|} - i_1 + 1$. Note that if $\mathbf{i}$ is not contiguous, $l(\mathbf{i}) > |u|$.

The *spectrum kernel* [46] is a simple string kernel originally introduced for protein classification. The *k-spectrum* of a string is the set of all its *k-mers*, i.e. contiguous substrings of length k. The feature space $\mathscr{H}_k = \mathbb{R}^{|\mathscr{A}|^k}$ of the spectrum kernel has a coordinate for each possible k-length sequence given the alphabet $\mathscr{A}$. Its corresponding feature map is:

$$\Phi(s) = (\phi_u(s))_{u \in \mathscr{A}^k}$$

where

$$\phi_u(s) = \text{number of times in which } u \text{ occurs in } s$$

giving a weighted representation of the k-spectrum. The k-spectrum kernel k_k is the inner product in this feature space:

$$k_k(s,t) = \langle \Phi(s), \Phi(t) \rangle = \sum_{u \in \mathscr{A}^k} \phi_u(s)\phi_u(t).$$

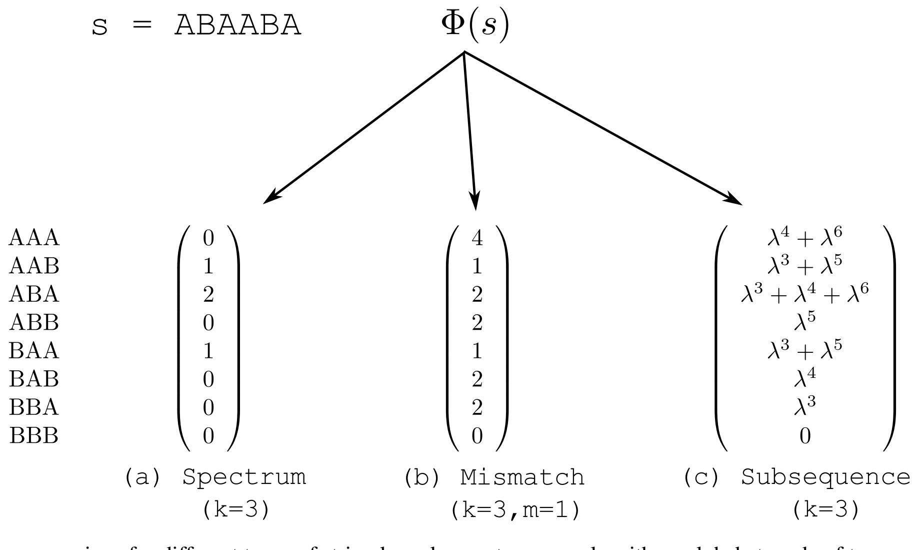


Fig. 5 Feature mappings for different types of string kernels on a toy example with an alphabet made of two symbols ($\mathscr{A} = \{\mathrm{A},\mathrm{B}\}$). The feature space is indexed by all substrings of length $k = 3$ in all cases. (a) Mapping for spectrum kernel. Each entry is the number of occurrences of the corresponding 3-mer in s. (b) Mapping for mismatch string kernel ($m = 1$). Note that each entry is equal to the sum of the entries of the spectrum kernel mapping for all 3-mers with one mismatch (e.g. AAA,AAB,ABA,BAA for entry AAA). (c) Mapping for the string subsequence kernel. Each entry is the number of times the corresponding substring is found in s, possibly with gaps, weighted according to the length of the match. Substring ABA for instance has matches ABA,ABAA,ABAABA with zero, one and three gaps respectively.

Figure 5(a) shows the feature mapping corresponding to a 3-spectrum kernel for a toy example with a simple alphabet of two symbols. For real-world cases, the feature space will have a very large dimension and feature vectors for strings will be typically extremely sparse. A very efficient procedure to compute the kernel can be devised using *suffix trees* [27]. A suffix tree for a given string s of length n is a tree with exactly n leaves, where each path from the root to a leaf is a suffix of the string s. The suffix tree for the string s can be constructed in $O(n)$ time using Ukkonen's algorithm [79]. In our case, a suffix tree can be used to identify all k-mers contained in the given sequence, simply following all the possible paths of size k starting from the root of the tree. Moreover, the problem of calculating the number of occurrences of each k-mer can be solved just counting the number of leaves in the subtree that starts at the end of the corresponding path. Given that the number of leaves of the tree is simply the size of the represented string, we have a linear-time method to calculate the k-spectrum of a string. Further modifications are needed to avoid the need of directly calculating the scalar product of the two feature vectors for the computation of the kernel. A generalized suffix tree is a suffix tree constructed using more than one string [27]. Given a set of strings there exists a variant of Ukkonen's algorithm that can build the corresponding generalized suffix tree in a time linear in the sum of the sizes of all the strings. A generalized suffix tree can be used to calculate the k-spectrum kernel of two strings at once, just traveling the tree in a depth first manner and summing up the products of the number of occurrences of every k-mer in the two strings. The procedure can also be used to compute a whole Gram matrix at once.

Spectrum kernels can be generalized to consider weighted combinations of substrings of arbitrary length:

$$k(s,t) = \sum_{u \in \mathscr{A}^*} w_u \phi_u(s) \phi_u(t) \tag{22}$$

where the non-negative coefficients w_u can be used for instance to give different weights according to the substring length. The k-spectrum kernel, for instance, is recovered setting $w_u = 1$ if $|u| = k$ and zero otherwise. Viswanathan and Smola [82] devised an efficient procedure for computing these types of kernels which exploits feature sparsity. The basic idea is to use the suffix tree representation for sorting all non-zero entries in $(\phi_u(s))_{u \in \mathscr{A}^*}$ and $(\phi_u(t))_{u \in \mathscr{A}^*}$ for s and t and evaluate only the matching ones.

The *mismatch string kernel* [47] is a variant of the spectrum kernel allowing for approximate matches between k-mers. Let the $(k,m) - neighbourhood$ of a k-mer u the set of all k-mers v which differ from u by at most m mismatches. Let $N_{(k,m)}(u)$ indicate this neighbourhood. The feature map of a k-mer u is:

$$\phi_{k,m}(u) = (\phi_v(u))_{v \in \mathscr{A}^k}$$

where $\phi_v(u) = 1$ if $v \in N_{(k,m)}(u)$ and zero otherwise. The feature map of a string s is computed summing the feature vectors of all its k-mers:

$$\Phi(s) = \sum_{k\text{-mers } u \text{ } in \text{ } s} \phi_{k,m}(u).$$

Figure 5(b) shows the feature mapping corresponding to a mismatch string kernel with $k = 3$ and $m = 1$, on the same toy example used for the spectrum kernel. Note the increase in density of the feature vector. On real-world cases with large alphabets this will nonetheless be still very sparse. A (k,m)-mismatch tree is a data structure similar to a suffix tree. At each depth-k node, it allows to compute the number of k-mers in the string which have at most m mismatches with the one along the path from the root to the node. Generalized (k,m)-mismatch trees can be created as for the suffix tree in order to directly compute the kernel for pairs of strings or full Gram matrices.

The *string subsequence kernel* (SSK) [49] is an alternative string kernel also considering gapped substrings in computing similarities. The feature map Φ for a string s is defined as:

$$\Phi(s) = (\phi_u(s))_{u \in \mathscr{A}^k} = \left(\sum_{\mathbf{i}:s(\mathbf{i})=u} \lambda^{l(\mathbf{i})} \right)_{u \in \mathscr{A}^k}$$

where $0 < \lambda \leq 1$ is a weight decay penalizing gaps. Such feature measures the number of occurrences of u in s, weighted according to their lengths. Note that the longer the occurrence, the more gaps in the alignment between u and s. Figure 5(c) shows the feature mapping obtained for $k = 3$ and arbitrary λ on the toy example already discussed, highlighting the contribution of the different occurrences of each substring. The inner product between strings s and t is computed as:

$$k_k(s,t) = \sum_{u \in \mathscr{A}^k} \phi_u(s)\phi_u(t) = \sum_{u \in \mathscr{A}^k} \sum_{\mathbf{i}:s(\mathbf{i})=u} \sum_{\mathbf{j}:t(\mathbf{j})=u} \lambda^{l(\mathbf{i})+l(\mathbf{j})}.$$

In order to make this product computationally efficient, we first introduce the auxiliary function

$$k'_i(s,t) = \sum_{u \in \mathscr{A}^i} \sum_{\mathbf{i}:s(\mathbf{i})=u} \sum_{\mathbf{j}:t(\mathbf{j})=u} \lambda^{|s|+|t|-i_1-j_1+2}$$

for $i = 1,\ldots,k-1$, counting the length from the beginning of the substring match to the end of s and t instead of $l(\mathbf{i})$ and $l(\mathbf{j})$. The SSK can now be computed by the following recursive procedure, where $a \in \mathscr{A}$:

$$\begin{aligned}
k_0(s,t) &= 1 \quad \forall s,t \in \mathscr{A}^* \\
k'_i(s,t) &= 0 \quad \text{if } \min(|s|,|t|) < i \\
k_i(s,t) &= 0 \quad \text{if } \min(|s|,|t|) < i \\
k'_i(sa,t) &= \lambda k'_i(s,t) + \sum_{j:t(j)=a} k'_{i-1}(s,t[1,\ldots,j-1])\lambda^{|t|-j+2}, \forall i \in [1,k-1] \\
k_k(sa,t) &= k_k(s,t) + \sum_{j:t(j)=a} k'_{k-1}(s,t[1,\ldots,j-1])\lambda^2.
\end{aligned} \tag{23}$$

To prove the correctness of the procedure note that $k_k(sa,t)$ is computed by adding to $k_k(s,t)$ all the terms resulting by the occurrences of substrings terminated by a, matching t anywhere and sa on its right terminal part. In fact, in the second term of the recursion step for k_k, k'_{k-1} will count any matching substring found in s as if it finished at $|s|$, and the missing λ for the last element a is added for both s and t.

This kernel can be readily expanded to consider substrings of different lengths, i.e. by using a linear combination like

$$k(s,t) = \sum_k c_k k_k(s,t)$$

with $c_k \geq 0$. In such case, we simply compute k'_i for all i up to one less than the largest k required, and then apply the last recursion in (23) for each k such that $c_k > 0$, using the stored values of k'_i.

A number of fast kernels for inexact string matching have been proposed in [48]. Alternative types of string kernels will be discussed in Section 4.4.

4.3.2 Trees

Trees allow to represent structured objects which include hierarchical relationships (without cycles). Phylogenetic trees, for instance, are commonly used in biology to represent the evolutionary relationships among different species or biological sequences. Parse trees are a standard way to represent the syntactic structure of a string in a certain language. Let's start with some definitions useful for describing examples of kernels on trees.

A tree is a connected graph without cycles. A *rooted* tree is a tree where one node is chosen as the root. A natural orientation arises in rooted trees, moving along paths starting from the root. The nodes on the path from the root to v are called *ancestors* of v, with the last one being its *parent*. The nodes on the path leaving v are its *descendants*. The direct descendants of v are its *children*. A leaf is a node with no children. An *ordered* tree is a tree with a total order relationship among the children of each node. A *labelled* tree is a tree where each node is labelled with a symbol from an alphabet $\mathscr{A}$. Let $l(v)$ return the label of node v. A *subtree* t' of t is a tree made of a subset of nodes and edges in t. A *proper subtree* t' of t is a tree made of a node a and all its descendants in t. A *subset tree* is a subtree with more than one node, having either all children of a node or none of them. Figure 6 shows examples of different types of subtrees.

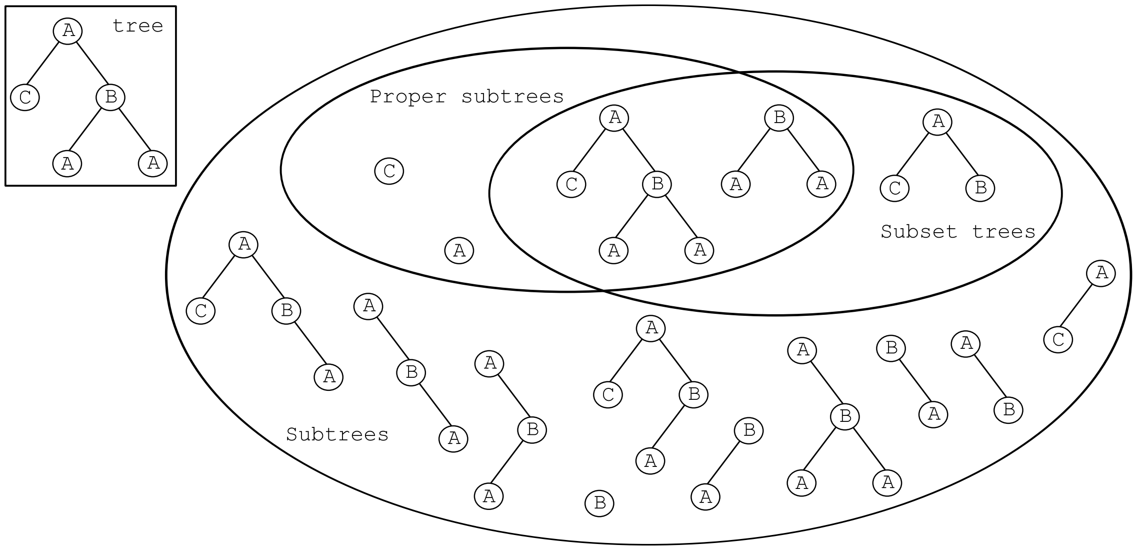


Fig. 6 Examples of different types of subtrees for a rooted ordered labelled tree. Tree kernels typically construct feature spaces based on these kind of fragments.

First of all, note that a generic string kernel can be applied to trees provided they are turned into a suitable string representation. Viswanathan and Smola [82] show how to employ the weighted substring kernel of eq. (22) to develop a kernel matching arbitrary subtrees. First, a procedure $tag(v)$ encodes a tree rooted at v in a string as follows:

- if v is an unlabelled leaf then $tag(v) = []$;
- if v is a labelled leaf then $tag(v) = [l(v)]$;
- if v is an unlabelled node with children $v_1, \ldots, v_m$ then $tag(v) = [tag(v_1) \cdots tag(v_m)]$;
- if v is a labelled node with children $v_1, \ldots, v_m$ then $tag(v) = [l(v)tag(v_1) \cdots tag(v_m)]$.

If the tree is unordered, the tags of the children are sorted in lexicographic order. The top leftmost tree in Fig. 6, for instance, would be encoded as: `[A[C][B[A][A]]]`.

The resulting string is fed to the weighted substring kernel, again relying on its suffix tree representation to exploit feature sparsity. Different choices for the w_s lead to different subtree features. Only proper subtrees, for instance, can be used by setting $w_s = 0$ for substrings not starting and ending with balanced brackets.

Collins and Duffy [10, 11] introduced a *subset tree kernel* based on subset tree matches in the field of Natural Language Processing (NLP). Here parse trees are rooted ordered labelled trees representing the syntactic structure of a sentence according to an underlying (stochastic) context free grammar (CFG). Each subtree consisting of a node and the set of its children is a production rule of the grammar. Only subset trees are considered as valid features for the kernel. The rationale behind this choice is not splitting production rules in defining subtrees.

Let $\mathscr{T}$ be a set of rooted ordered labelled trees. The feature space has a coordinate for each possible subset tree in $\mathscr{T}$. Given by M the number of these fragments, a tree t is mapped to:

$$\Phi(t) = (\phi_i(t))_{i \in [1,M]}$$

where:

$$\phi_i(t) = \text{number of times in which the } i^{th} \text{ tree fragment occurs in } t$$

We define the set of nodes in t_1 and t_2 as N_1 and N_2 respectively. We further define an indicator function $I_i(n)$ to be 1 if subset tree i is seen rooted at node n and 0 otherwise. The kernel between t_1 and t_2 can now be written as

$$k(t_1,t_2) = \sum_{i=1}^{M} \phi_i(t_1)\phi_i(t_2) = \sum_{i=1}^{M} \sum_{n_1 \in N_1} I_i(n_1) \sum_{n_2 \in N_2} I_i(n_2) = \sum_{n_1 \in N_1} \sum_{n_2 \in N_2} C(n_1,n_2)$$

where we define $C(n_1,n_2) = \sum_{i=1}^{M} I_i(n_1)I_i(n_2)$, that is the number of common subset trees rooted at both n_1 and n_2. Given two nodes n_1 and n_2, we say that they *match* if their have the same label, same number of children and each child of n_1 has the same label of the corresponding child of n_2. The following recursive definition permits to compute $C(n_1,n_2)$ in polynomial time:

- If n_1 and n_2 don't match $C(n_1,n_2) = 0$.
- if n_1 and n_2 match, and they are both pre-terminals[3] $C(n_1,n_2) = 1$.
- Else

$$C(n_1,n_2) = \prod_{j=1}^{nc(n_1)} (1 + C(ch(n_1,j),ch(n_2,j))) \quad (24)$$

  where $nc(n_1)$ is the number of children of n_1 (equal to that of n_2 for the definition of match) and $ch(n_1,j)$ is the j^{th} child of n_1.

To prove the correctness of (24), note that each child of n_1 contributes exactly $1+C(ch(n_1,j),ch(n_2,j))$ common subset trees for n_1,n_2, the first with the child alone, and the other $C(ch(n_1,j),ch(n_2,j))$ with the common subset trees rooted at the child itself. The product in (24) considers all possible combinations of subset trees contributed by different children.

Given the large difference in size of trees to be compared, it is usually convenient to employ a normalized version of the kernel (see eq. (16)). Moreover, the kernel tends to produce extremely large values for very similar trees, thus making the algorithm behave like a one-nearest neighbour rule. This effect can be reduced by restricting the depth of the allowed subset trees to a fixed value d, or by scaling their relative importance with their size. To this extent we can introduce a parameter $0 < \lambda \leq 1$, turning the last two points of the definition of C into:

- if n_1 and n_2 match, and they are both pre-terminals $C(n_1,n_2) = \lambda$.
- Else

$$C(n_1,n_2) = \lambda \prod_{j=1}^{nc(n_1)} (1 + C(ch(n_1,j),ch(n_2,j))).$$

This corresponds to a modified inner product

$$k(t_1,t_2) = \sum_{i=1}^{M} \lambda^{size_i} \phi_i(t_1)\phi_i(t_2)$$

where $size_i$ is the number of nodes of the corresponding subset tree.

Most tree kernels developed in the literature are extensions of the subset tree kernel. Moschitti [55] proposed the *partial tree kernel* in which arbitrary subtrees are used as fragments in place of subset trees. The kernel is obtained replacing equation (24) with:

$$C(n_1,n_2) = 1 + \sum_{J_1,J_2:|J_1|=|J_2|} \prod_{i=1}^{|J_1|} C(ch(n_1,J_{1i}),ch(n_2,J_{2i})))$$

where $J_1 = (J_{11},J_{12},\ldots,J_{1|J_1|})$ and $J_2 = (J_{21},J_{22},\ldots,J_{2|J_2|})$ and index sequences associated with ordered child sequences of n_1 and n_2 respectively. The *elastic tree kernel* extends the subset tree kernel by allowing: 1) matches between nodes with different number of children, provided the comparison still follows their left-to-right ordering; 2) approximate label matches, by introducing a similarity measure between them; 3) elastic matching between subtrees, where subtrees can be "stretched" in a tree provided the relative

[3] A pre-terminal is the parent of a leaf.

positions of the nodes in the subtree are preserved. Aiolli et al. [1] show that kernels defined on routes between tree nodes provide competitive discriminative power at reduced computational complexity with respect to most tree kernels. For a detailed treatment of the literature on tree kernels see [50].

4.3.3 Graphs

Graphs are a natural and powerful way to represent structured objects in a variety of domains, ranging from chemo- and bio-informatics to the World Wide Web. Here we will focus on graphs representing individual objects, like a chemical compound with atoms as vertices and bonds as edges. A different problem is that of treating objects that are related to each other by a graph structure. For a description of kernels on this type of data see [42, 21, 20, 69]. Let's start with some useful definitions concerning graphs.

A graph $G = (\mathscr{V}, \mathscr{E})$ is a finite set of vertices (also called nodes) $\mathscr{V}$ and edges $\mathscr{E} \in \mathscr{V} \times \mathscr{V}$. In *directed* graphs edges (v_i, v_j) are oriented from initial node v_i to terminal node v_j. In *undirected* graphs edges have no orientation. This can be represented for instance by letting $(v_i, v_j) \in \mathscr{E} \iff (v_j, v_i) \in \mathscr{E}$. A *labelled* graph is a graph with a set of labels $\mathscr{L}$ and a function long$(\cdot)$ assigning labels to nodes (*node-labelled* graph), edges (*edge-labelled* graph) or both (*fully-labelled* graph). In the following we will call node-labelled graphs simply labelled graphs unless otherwise specified. A labelled graph can also be represented by its *adjacency* and label matrices A and L. The adjacency matrix is such that $A_{ij} = 1$ if $(v_i, v_j) \in \mathscr{E}$ and zero otherwise. The node-label matrix L is such that $L_{ij} = 1$ if $l(v_j) = \ell_i$ and zero otherwise. A *walk* in a graph is a sequence of vertices $(v_1, \ldots, v_{n+1})$ with $v_i \in \mathscr{V}$ and $(v_i, v_{i+1}) \in \mathscr{E}$ for all i. The length of a walk is the number of its edges (n in the example before). Let $W_n(G)$ return all n-length walks in graph G. A *path* is a walk such that $v_i \neq v_j \iff i \neq j$. A *cycle* is a path such that $(v_{n+1}, v_1) \in \mathscr{E}$. A walk can contain an arbitrary number of cycles. A *connected* graph is a graph having at least one undirected path for each pair of its nodes. The *distance* between two nodes is the length of the minimal undirected path between them (if any). Given a graph G a *subgraph* G' is a graph made of a subset of the nodes and edges of G. Given a set of nodes $\mathscr{V}' \subset \mathscr{V}$, the subgraph *induced* by $\mathscr{V}'$ is made of nodes $\mathscr{V}'$ and all edges $\mathscr{E}' \subset \mathscr{E}$ connecting them. Two graphs G and G' are *isomorphic* if there is an isomorphism, a bijection ϕ such that for any two nodes $v_1, v_2 \in \mathscr{V}$ there is an edge $(v_1, v_2) \in \mathscr{E} \iff (\phi(v_1), \phi(v_2)) \in \mathscr{E}'$. For labelled graphs the isomorphism must also preserve label information, i.e. $l(v) = l(\phi(v))$. An isomorphism basically defines equivalence classes for graphs.

First, note that as discussed in the case trees, string kernels can be applied to the string encoding of a graph. The Simplified Molecular Input Line Entry Specification (SMILES), for instance, is a popular string notation for chemical molecules. Kernels for SMILES strings are described in [74], among other kernels for 2D and 3D molecular representations.

In defining kernels on graphs, we of course do not want to distinguish among isomorphic graphs. Given the set $\mathscr{G}$ of all graphs, the subgraph feature space is the space of all possible subgraphs of $\mathscr{G}$ modulo isomorphism, i.e. any two isomorphic subgraph are mapped to the same coordinate. Gärtner et al. [23] proved that no polynomial time algorithm can be devised for computing an inner product in such space (unless $P = NP$). The same holds if we restrict the feature space to consider paths only. Most existing kernels on generic graphs either use features based on graph walks, or employ some strategy to limit the set of valid subgraphs.

Examples of the first approach can be found in [19]. Consider the case of undirected node-labelled graphs. The simplest examples compute similarities in terms of walks in the two graphs which start and end with the same labels, i.e. $(v_1, \ldots, v_{n+1}) \in W_n(G)$ and $(v'_1, \ldots, v'_{m+1}) \in W_m(G')$ for which $l(v_1) = l(v'_1)$ and $l(v_{n+1}) = l(v'_{m+1})$. The feature space of the kernel is defined in terms of features for label pairs:

$$\Phi(G) = \left(\phi_{\ell_i,\ell_j}(G)\right)_{\ell_i,\ell_j \in \mathscr{L}}$$

where:

$$\phi_{\ell_i,\ell_j}(G) = \sum_{n=1}^{\infty} \lambda_n |\{(v_1, \ldots, v_{n+1}) \in W_n(G) : l(v_1) = \ell_i \wedge l(v_{n+1}) = \ell_j\}|.$$

and $\boldsymbol{\lambda}$ is a sequence of non-negative weights ($\lambda_n \in \mathbb{R}_0^+$ for all $n \in \mathbb{N}$). The feature map for a label pair ℓ_i, ℓ_j returns a weighted sum of the number of walks of length n starting with ℓ_i and ending with ℓ_j for all possible lengths n.

The feature mapping can be computed by operations on the graph matrices. The adjacency matrix has the useful property that its n power A^n extends the adjacency concept to walks of length n, that is $(A^n)_{ij}$ is the number of walks of length n from v_i to v_j. By introducing the label matrix we obtain that $(LA^nL^T)_{ij}$ is the number of walks of length n which start and end with labels ℓ_i and ℓ_j respectively. Thus:

$$\phi_{\ell_i,\ell_j}(G) = \left(\sum_{n=1}^{\infty} \lambda_n L A^n L^T\right)_{\ell_i,\ell_j}$$

and the corresponding kernel is:

$$k(G,G') = \langle L\left(\sum_{i=1}^{\infty} \lambda_i A^i\right) L^T, L'\left(\sum_{j=1}^{\infty} \lambda_j A'^j\right) L'^T\rangle$$

where the dot product between two matrices M, M' is defined as:

$$\langle M, M'\rangle = \sum_{i,j} M_{ij} M'_{ij}. \tag{25}$$

Note that the kernel has to be absolute convergent in order to be a valid positive definite kernel. A sufficient condition for *absolute convergent graph kernels* can be found in [19]. An example of valid kernel is the *exponential graph kernel* defined as:

$$k_{exp}(G,G') = \langle L e^{\beta A} L^T, L' e^{\beta A'} L'^T\rangle$$

where $\beta \in \mathbb{R}$ is a parameter and the exponential of a matrix M is defined as

$$e^{\beta M} = \lim_{n\to\infty} \sum_{i=0}^{n} \frac{(\beta M)^i}{i!}.$$

Feasible matrix exponentiation usually requires diagonalizing the matrix. If we can diagonalize A such that $A = T^{-1}DT$, we can easily compute any power of A as $A^n = (T^{-1}DT)^n = T^{-1}D^nT$, where the power of the diagonal matrix D is calculated component-wise $[D^n]_{ij} = [D_{ij}]^n$. Therefore we have

$$e^{\beta A} = T^{-1} e^{\beta D} T$$

where $e^{\beta D}$ is calculated component-wise.

It's straightforward to extend this kernel to graphs with weighted edges by setting $A_{ij} = weight(v_i, v_j)$. The feature space of these kernels has a dimension equal to the square of the number of labels. When there are few possible labels, this can prevent the realization of an informative similarity measure. Extensions to this type of kernels consider the labels along the entire walk instead of only those of the terminal nodes. The feature space in this case is indexed by strings u, i.e.:

$$\Phi(G) = (\phi_u(G))_{u\in\mathscr{A}*}$$

where:

$$\phi_u(G) = \lambda_n |(v_1,\ldots,v_{n+1}) \in W_n(G) : n = |u|-1 \wedge l(v_1) = u_1 \wedge \cdots \wedge l(v_{n+1}) = u_{n+1}\}|.$$

Note that it is straightforward to consider edge-labelled or fully-labelled graphs by replacing or adding edge labels in the comparisons. The kernel can be computed based on the powers of the adjacency matrix of the direct product [4] of the two graphs. A further extension accounts for up to $m \geq 0$ mismatches in the walk

[4] The direct product of two graphs G, G' has nodes $(v,v') \in \mathscr{V} \times \mathscr{V}' : l(v) = l(v')$ and edges $((v,v'),(u,u')) \in (\mathscr{V} \times \mathscr{V}')^2 : (v,u) \in \mathscr{E} \wedge (v',u') \in \mathscr{E}' \wedge\ l(v,u) = l(v',u')$. Nodes and edges in the direct product inherit the labels of the corresponding nodes and edges in two graphs.

labels. The kernel can be computed by a combination of the direct product of the labelled and unlabelled graphs respectively. See [22] for the details of these kernels.

A large number of kernels have been developed relying on possibly domain-inspired strategies for choosing which subgraphs to include in the feature space. The *cyclic pattern kernel* [32] extracts features consisting of cycle and tree patterns. Cycles are directly extracted from the graph. A set of trees (called a forest) is obtained by removing from the graph all edges of all cycles. A canonical string representation for cycles and trees is employed in order to map each of them to a distinct feature coordinate modulo isomorphism (the pattern). Note that cyclic pattern kernels still cannot be computed in polynomial time in general [32], but it's sufficient to limit the computation to a subset of well-behaved graphs with a small enough number of cycles. The *shortest-path kernel* [8] considers the shortest-path between pairs of nodes. The *graph fragment kernel* [84] considers all connected subgraphs up to a given number of edges.

Note that kernels on graphs are not limited to exact or approximate matches between substructures. The *weighted decomposition kernel* (WDK) [52], for instance, compares two graphs by a combination of kernels between node pairs together to their contexts. Recalling the general form for R-convolution kernels (see eq. (20)), let $R^{-1}(G)$ be a decomposition of the graph into a node v and its *context* V in the graph. A possible context is the subgraph induced by all nodes at distance at most d from v (called its n-neighbourhood subgraph). The WDK is defined as:

$$k(G,G') = \sum_{(v,V)\in R^{-1}(G)} \sum_{(v',V')\in R^{-1}(G')} \delta(l(v),l(v'))k_{neigh}(V,V')$$

where δ is the matching kernel (see eq.(19)) and k_{neigh} is a kernel on the neighbourhood subgraph. Further details on graph kernels can be found e.g. in [22, 7].

4.4 Kernels from Generative Models

Generative models such as Hidden Markov Models [61] are a principled way to represent the probability distribution underlying the generation of data, and allow to treat aspects like uncertainty and missing information under a unifying formalism. On the other hand, discriminative methods such as kernel machines are an effective way to build decision boundaries, and often outperform generative models in prediction tasks. It would thus be desirable to have a learning method able to combine these complementary approaches. In the following we will present some examples of kernels derived from generative models, by directly modeling joint probability distributions [85, 29], defining a similarity measure between the models underlying two examples [35, 34], or defining arbitrary kernels over observed and hidden variables and marginalizing over the hidden ones [78, 38]. For an additional general class of kernels from probability distributions see [36].

4.4.1 Dynamic Alignment Kernels

Joint probability distributions are a natural way of representing relationships between objects. The similarity of two objects can be modeled as a joint probability distribution that assigns high probabilities to pairs of related objects and low probabilities to pairs of unrelated objects. These considerations have been used in [85] to propose a kernel based on joint probability distributions. An analogous kernel was independently presented in [29] as a special case of convolution kernel (see section 4.2).

Definition 5. A joint probability distribution is *conditionally symmetrically independent* (CSI) if it is a mixture of a finite or countable number of symmetric conditionally independent distributions.

In order to show that a CSI joint p.d. is a positive definite kernel, let's write it as a dot product. Let X,Z,C be three discrete random variables such that

$$p(x,z) = P(X=x,Z=z) = p(z,x)$$

and

$$p(x,z|c) = P(X = x, Z = z|C = c) = p(x|c)p(z|c)$$

for all possible realizations of X, Z, C. We can thus write

$$p(x,z) = \sum_c p(x|c)p(z|c)p(c) = \sum_c \left(p(x|c)\sqrt{p(c)}\right)\left(p(z|c)\sqrt{p(c)}\right)$$

where the sum is over all possible realizations $c \in \mathscr{C}$ of C. This corresponds to a dot product with feature map

$$\Phi(x) = \{p(x|c)\sqrt{p(c)} \mid c \in \mathscr{C}\}.$$

For a more general proof see [85].

A joint p.d. for a finite symbol sequence can be defined with a pair Hidden Markov Model. Such models generate two symbol sequences simultaneously, and are used in bioinformatics to align pairs of protein or DNA sequences [16]. A PHMM can be defined as follows, where A, B represent the two sequences modeled.

- A finite set S of states, given by the disjoint union of:

  S^{AB} - states that emit one symbol for A and one for B,
  S^{A} - states that emit one symbol only for A,
  S^{B} - states that emit one symbol only for B,
  a starting state START and an ending state END, which don't emit symbols.

- An $|S| \times |S|$ state transition probability matrix T.
- An alphabet $\mathscr{A}$.
- For states emitting symbols:
  - for $s \in S^{AB}$ a probability distribution over $\mathscr{A} \times \mathscr{A}$,
  - for $s \in S^{A}$ or $s \in S^{B}$ a probability distribution over $\mathscr{A}$.

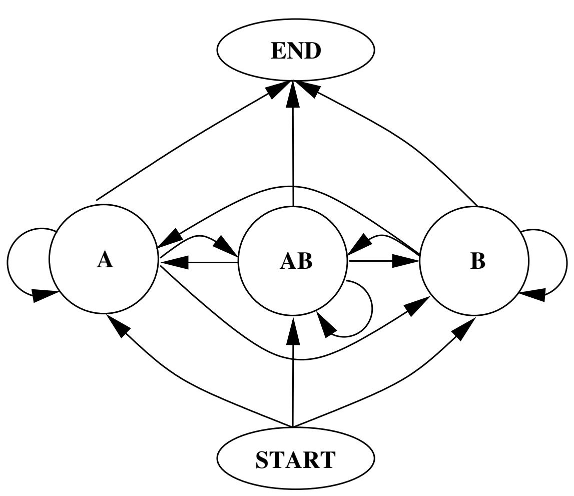


Fig. 7 State diagram for PHMM modeling pairs of sequences *AB*. The state *AB* emits common or similar symbols for both sequences, while the states *A* and *B* model insertions in sequence *A* and *B* respectively.

The state diagram for this PHMM is represented in figure 7. The state *AB* emits matching or nearly matching symbols for both sequences, while states *A* an *B* model insertions, that is symbols found in one sequence but not in the other. The joint p.d. for two sequences is given by the combination of all possible paths from START to END, weighted by their probabilities. This can be efficiently computed by well known dynamic programming algorithms [61]. Sufficient conditions for a PHMM to be CSI can be found in [85].

4.4.2 Fisher Kernel

The basic idea of the Fisher Kernel [35, 34] is that of representing the generative processes underlying two examples into a metric space, and compute a similarity measure in such space. Given a generative probability model $P(X|\theta)$ parametrized by $\theta = (\theta^1, \dots, \theta^r)$, the gradient of its loglikelihood with respect to θ, $V_\theta(X) := \nabla_\theta \log P(X|\theta)$, is called *Fisher score*. It indicates how much each parameter θ^i contributes to the generative process of a particular example. The gradient is directly related to the expected *sufficient statistics* for the parameters. In the case that the generative model is an HMM, such statistics come as a by product of the forward backward algorithm [61] used to compute $P(X|\theta)$, without any additional cost. The derivation of the gradient for HMM and its relation to sufficient statistics is described in [33].

A class of models $P(X|\theta)$, $\theta \in \Theta$ defines a Riemannian manifold M_Θ (see [2, 3]), with metric tensor given by the covariance of the Fisher score, called *Fisher information matrix* and computed as

$$F := E_p[V_\theta(X) V_\theta(X)^T]$$

where the expectation is over $P(X|\theta)$. The direction of steepest ascent of the loglikelihood along the manifold is given by the *natural gradient* $\tilde{V}_\theta(X) = F^{-1} V_\theta(X)$ (see [3] for a proof). The inner product between such natural gradients relative to the Riemannian metric,

$$k(X,X') = \tilde{V}_\theta(X)^T F \tilde{V}_\theta(X) = V_\theta(X)^T F^{-1} V_\theta(X)$$

is called *Fisher kernel*. When the Fisher information matrix is too difficult to compute, it can be approximated by $F \approx \sigma^2 I$, where I is the identity matrix and σ a scaling parameter. Moreover, as $V_\theta(X)$ maps X to a vectorial feature space, we can simply use the dot product in such space, giving rise to the *plain* kernel

$$k(X,X') = V_\theta(X)^T V_\theta(X).$$

The Fisher kernel has been successfully employed for instance for detecting remote protein homologies [33], where the generative model is chosen to be an HMM representing a given protein family.

4.4.3 Marginalized Kernels

Marginalized kernels [78] are a hybrid class of kernels combining probabilistic models and arbitrary kernels over structures. Assume that a reasonable probabilistic model for the examples should include both observed variables x and hidden ones h. For instance, a set of images of handwritten characters could come from a number of different writers. Let $p(h|x)$ be the posterior probability of hidden variables given observed ones. Let $z = (x,h)$ and let $k_z(z,z')$ be a *joint* kernel over both observed and hidden variables. A *marginalized* kernel is obtained taking the expectation of the joint kernel with respect to the hidden variables, i.e.:

$$k(x,x') = \sum_h \sum_{h'} p(h|x) p(h'|x') k_z(z,z').$$

A first example of this type of kernel is the marginalized count kernel [78] for strings. Let x be a string of symbols from an alphabet $\mathscr{A}_x$. A very simple kernel is the 1-mer spectrum kernel (see Section 4.3.1), based on the counts of symbol co-occurrences:

$$k(x,x') = \sum_{u \in \mathscr{A}_x} \phi_u(x) \phi_u(x')$$

where $\phi_u(x)$ counts the number of occurrences of symbol u in x. This kernel treats all elements of the string the same, i.e. as if they were coming from the same distribution. Knowledge of the domain can suggest us that considering a number of different distributions should be more appropriate. For instance, in treating protein sequences we could distinguish between residues which are exposed at the surface and those that are buried in the protein core. We would thus like to compute separate counts for residues in the two conditions. As this information is not directly available from the sequence, we will model it using hidden variables. Let h be a string of hidden variables from an alphabet $\mathscr{A}_h$ ($\mathscr{A}_h = \{E, B\}$ in the protein example), with $|h| = |x|$. The marginalized count kernel is defined as:

$$k(x,x') = \sum_h \sum_{h'} p(h|x)p(h'|x') \sum_{u_x \in \mathscr{A}_x} \sum_{u_h \in \mathscr{A}_h} \phi_{u_x,u_h}(z)\phi_{u_x,u_h}(z')$$

where

$$\phi_{u_x,u_h}(z) = \text{number of times in which } u_x \text{ and } u_h \text{ appear in the same position in } x \text{ and } h \text{ respectively}$$

The kernel can be written in terms of marginalized counts $\hat{\phi}_{u_x,u_h}$:

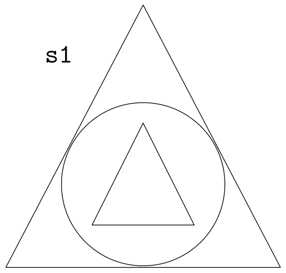
$$k(x,x') = \sum_{u_x \in \mathscr{A}_x} \sum_{u_h \in \mathscr{A}_h} \underbrace{\sum_h p(h|x)\phi_{u_x,u_h}(z)}_{\hat{\phi}_{u_x,u_h}(x)} \underbrace{\sum_{h'} p(h'|x')\phi_{u_x,u_h}(z')}_{\hat{\phi}_{u_x,u_h}(x')} .$$

Note that counts can be easily generalized to k-mers with $k > 1$. Marginalized kernels have been defined for graphs [38] by defining transition probabilities between nodes and computing kernels in terms of random walks. These are tightly related to the walk-based kernels described in Section 4.3.3. See [81] for a unifying framework.

4.5 Kernels on Logical Representations

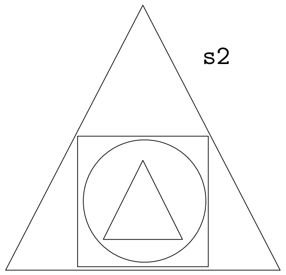
Logic representation formalisms allow to naturally express complex domain knowledge and perform reasoning on it. Developing kernels capable of handling this type of representation can greatly enhance their expressive power, allowing them to incorporate the semantics of the domain under consideration. In this section we will describe a generic class of kernels on logic terms. We will then show how to employ it to define kernels based on logic proofs. We will focus on the widespread Prolog programming language [73], which is based on first order logic enriched with partial support for arithmetic operations and some higher order structures like sets. Further details and additional examples of logic kernels can be found in [18]. For a general treatment of kernels on higher order logic representations see [24].

Let's first introduce some definitions and notation. A *definite clause* is an expression of the form $h \leftarrow b_1, ..., b_n$, where h and the b_i are atoms and commas indicate logical conjunctions. Atoms are expressions of the form $p(t_1, ..., t_n)$ where $p^{/n}$ is a predicate symbol of arity n and the t_i are *terms*. Terms are constants (denoted by lower case), variables (denoted by upper case), or structured terms. Structured terms are expressions of the form $f(t_1, ..., t_k)$, where $f^{/k}$ is a functor symbol of arity k and $t_1, ..., t_k$ are terms. A term is *ground* if it contains no variable. The atom h is also called the head of the clause, and $b_1, ..., b_n$ its body. Intuitively, a clause represents that the head h will hold whenever the body $b_1, ..., b_n$ holds. A clause with an empty body is called a *fact*. A set of clauses forms a *knowledge base* $\mathscr{B}$. When representing a domain of interest, we typically distinguish between *extensional* knowledge modeling single individuals of the domain (typically as a set of ground facts) and *intensional* knowledge providing general rules. A *substitution* $\theta = (V_1/t_1, \ldots, V_n/t_n)$ is an assignment which applied to a formula F (written as $F\theta$, where F is a clause, atom or term) replaces each occurrence of variable V_i in h with term t_i, for all i. Queries can be used in order to derive novel information from a knowledge base. A query is a special clause (called *goal*) with an empty head. A goal $\leftarrow g$ is *entailed* by a knowledge base $\mathscr{B}$ (written as $\mathscr{B} \models g$) if and only if can be proved using clauses in $\mathscr{B}$ (otherwise $\mathscr{B} \not\models g$). A correct answer for g is a (possibly empty) substitution θ used to prove the goal.



Extensional knowledge

```
triangle(s1,o1).     triangle(s2,o1).
circle(s1,o2).       rectangle(s2,o2).
triangle(s1,o3).     circle(s2,o3).
in(s1,o1,o2).        triangle(s2,o4).
in(s1,o2,o3).        in(s2,o1,o2).
                     in(s2,o2,o3).
                     in(s2,o3,o4).
```



Intensional knowledge

```
polygon(X,A) ← triangle(X,A).    inside(X,A,B) ← in(X,A,B).
polygon(X,A) ← circle(X,A).      inside(X,A,B) ← in(X,A,C),inside(X,C,B).
polygon(X,A) ← rectangle(X,A).
```

Fig. 8 Example of Prolog-based representation for an artificial domain. Top: extensional knowledge for two sample scenes. Bottom: intensional knowledge representing generic polygons and nesting in containment. The first argument of all predicates indicates the scene, the other arguments are objects within the scene.

Figure 8 shows an artificial domain with scenes containing nested polygons, highlighting extensional and intensional knowledge. The `polygon/2` predicate models generic polygons, while `inside/3` models the concept of nesting in containment. Novel facts can be derived from the knowledge base. Substitution $\theta =$ (`X/bong1,A/o3`), for instance, is a correct answer for goal $\leftarrow$ `polygon(X,A)`, while `inside(bong2,o2,o4)` can be proved from goal $\leftarrow$ `inside(X,A,B)` with answer $\theta =$ (`X/bong2,A/o2,B/o4`).

4.5.1 Kernels on Ground Terms

Complex objects can be represented as structured terms. In the *individual-as-term* representation an entire individual in the domain (e.g. a molecule) is encoded as a single structured term. Defining kernels between terms allows to apply kernel methods on this type of representation.

Let $\mathscr{C}$ be a set of constants and $\mathscr{F}$ a set of functors, and denote by $\mathscr{U}$ the corresponding Herbrand universe (the set of all ground terms that can be formed from constants in $\mathscr{C}$ and functors in $\mathscr{F}$). Let's assume that a type syntax exists. Types of constants are indicated with single characters (e.g. τ). Structured terms $f(t_1,...,t_k)$ have type signatures $\tau_1 \times, \dots, \times \tau_k \mapsto \tau'$, where τ' is the type of the term and $\tau_1 \times, \dots, \times \tau_k$ the types of its arguments. We write $s : \tau$ to indicate that s is of type τ. The kernel between two ground typed terms t and s is a function $k : \mathscr{U} \times \mathscr{U} \to \mathbb{R}$ defined inductively as follows:

- if $s \in \mathscr{C}$, $t \in \mathscr{C}$, $s : \tau$, $t : \tau$ then
$$k(s,t) = \kappa_\tau(s,t)$$
where $\kappa_\tau : \mathscr{C} \times \mathscr{C} \mapsto \mathbb{R}$ is a valid kernel on constants of type τ;
- else if s and t are structured terms that have the same type but different functors or signatures, i.e., $s = f(s_1,\dots,s_n)$ and $t = g(t_1,\dots,t_m)$, $s : \sigma_1 \times, \dots, \times \sigma_n \mapsto \tau'$, $t : \tau_1 \times, \dots, \times \tau_m \mapsto \tau'$, then
$$k(s,t) = \iota_{\tau'}(f^{/n}, g^{/m})$$
where $\iota_{\tau'} : \mathscr{F} \times \mathscr{F} \mapsto \mathbb{R}$ is a valid kernel on functors that construct terms of type τ'

- else if s and t are structured terms and have the same functor and type signature, i.e., $s = f(s_1,\dots,s_n)$, $t = f(t_1,\dots,t_n)$, and $s,t : \tau_1 \times,\dots,\times \tau_n \mapsto \tau'$, then

$$k(s,t) = \iota_{\tau'}(f^{/n}, f^{/n}) + \sum_{i=1}^{n} k(s_i,t_i) \tag{26}$$

- in all other cases $k(s,t) = 0$.

As a simple example consider data structures intended to describe scientific references:

$r = \texttt{article}(\texttt{"Kernels on Gnus and Gnats"},\texttt{journal}(\texttt{ggj},2004))$
$s = \texttt{article}(\texttt{"The Logic of Gnats"},\texttt{conference}(\texttt{icla},2004))$
$t = \texttt{article}(\texttt{"Armadillos in Hilbert space"},\texttt{journal}(\texttt{ijaa},2004))$

Using $\kappa_\tau(x,z) = \delta(x,z)$ for all τ and $x,z \in \mathscr{C}$ and $\iota_{\tau'}(x,z) = \delta(x,z)$ for all τ' and $x,z \in \mathscr{F}$, we obtain $k(r,s) = 1$, $k(r,t) = 3$, and $k(s,t) = 1$. The fact that all papers are published in the same year does not contribute to $k(r,s)$ or $k(s,t)$ since these pairs have different functors describing the venue of the publication; it does contribute to $k(r,t)$ as they are both journal papers. Note that strings have been treated as constants. A more informed similarity measure can be obtained employing a string kernel for comparing constants of type string.

Positive semi-definiteness of kernels on ground terms follows from their being special cases of decomposition kernels (see [57] for details). Variants where direct summations over sub-terms are replaced by tensor products are also possible. These kernels can be generalized to deal with higher order logic representations, see [24] for a detailed treatment.

4.5.2 Kernels on Proof Trees

In proving goals, an interpreter is required to recursively prove a number of subgoals according to a certain ordering (top-down for different clauses in the knowledge base and left-to-right in the clause body for Prolog). The proof has a structure which contains relevant information for the reasons for the final outcome. It would be desirable to be able to exploit this additional information in computing similarity between examples. This can be achieved by defining kernels directly on proofs [57], instead of simply on their outcomes in terms of goal satisfaction and goal variable substitutions. Given a knowledge-base $\mathscr{B}$ and a goal g, the proof tree for g is empty if $\mathscr{B} \not\models g$ or, otherwise, it is a tree t recursively defined as follows:

- if there is a fact f in $\mathscr{B}$ and a substitution θ such that $g\theta = f\theta$, then $g\theta$ is a leaf of t.
- otherwise there must be a clause $h \leftarrow b_1,...,b_n \in \mathscr{B}$ and a substitution θ' such that $h\theta' = g\theta'$ and $\mathscr{B} \models b_j\theta'\ \forall j$, $g\theta'$ is the root of t and there is a subtree of t for each $b_j\theta'$ that is a proof tree for $b_j\theta'$.

Each internal node contains a clause head and its ordered set of children is the ordered set of atoms in its body. A simple bottom-up recursive procedure to turn a proof tree into a structured term consists of appending after the clause arguments the term representation of each of its children. A kernel on Prolog ground terms like the ones defined in the previous section can be applied to this representation. Note that a goal g can often be proved in multiple ways (if it is satisfiable), leading to a set of proof trees. A set kernel (eq. 21) can be used to compare two sets of proof trees by combining each pairwise comparison between proofs.

Let's recall the artificial example in Figure 8. Assume that positive scenes contain two triangles nested into one another with exactly n objects (possibly triangles) in between. The scene on the left would be positive for $n = 1$, the one on the right for $n = 2$. Having hints on the target concept, one could define a predicate (let's call it *visit*) looking for two polygons contained one into the other:

```
visit(X) ← inside(X,A,B),polygon(X,A),polygon(X,B)
```

Figure 9 shows the proofs trees obtained running such a visitor on the first scene in Figure 8, plus their representation as terms. Note that all the information required to identify the target concept is buried in

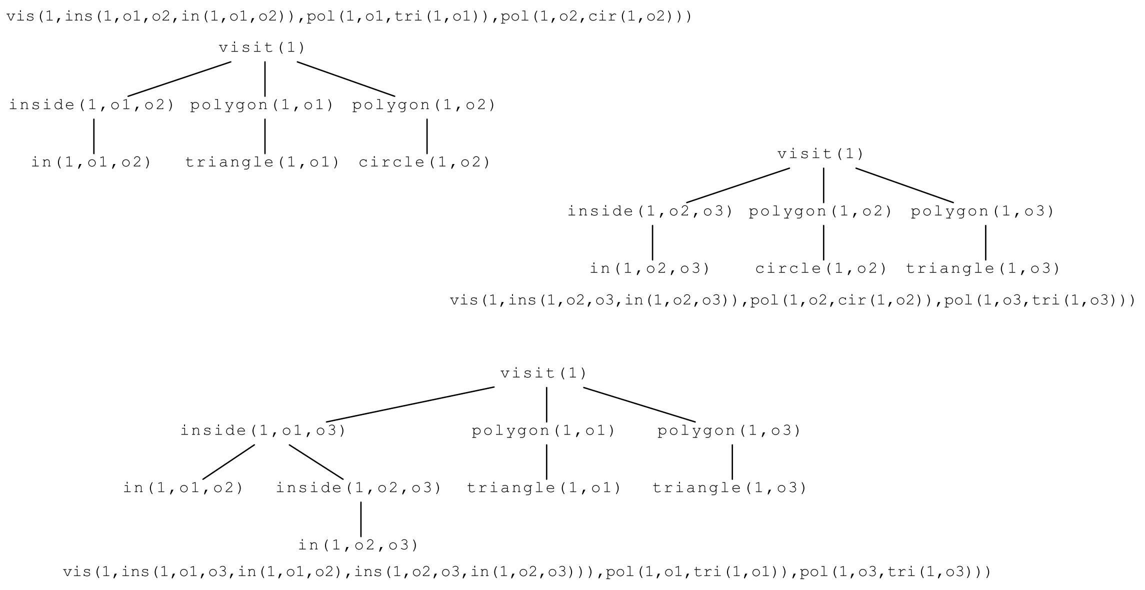


Fig. 9 Proof trees obtained by running the visitor on the first scene in Fig. 8. Representation of proof trees as ground structured terms (functor name abbreviated).

the tree/term structure. A very simple kernel capable of exploiting this information would employ product (instead of sum) between terms (see eq. (26)), delta kernel (see eq. (19)) between functors, and a kernel between constants always evaluating to one. Applied to a pair of proof trees, this kernel evaluates to one if they both contain the same pair of polygons (e.g. a triangle and a circle) nested one into the other with the same number of objects in between, and zero otherwise. For any value of n, such a kernel maps the examples into a feature space where there is a single feature discriminating between positive and negative examples. Simply using extensional knowledge would not allow to produce effective similarities for this task. See [57] for a more extensive treatment of proof tree kernels and real-world applications.

5 Learning Kernels

Choosing the most appropriate kernel for the problem at hand is usually a rather hard task. Standard approaches consist of trying a number of candidate alternatives (e.g. Gaussian kernels with varying width, spectrum kernels with varying k) and selecting the best one according to some cross validation procedure. Measures of kernel quality can also be used to this aim. *Kernel target alignment* [14], for instance, computes the alignment between a kernel and the "target" kernel, $k_t(x_i,x_j) = y_i y_j$ for the binary classification case. Given a Gram matrix K and a target matrix $Y = yy^T$, the kernel target alignment is computed as:

$$A(K,Y) = \frac{\langle K,Y \rangle}{\sqrt{\langle K,K \rangle \langle Y,Y \rangle}} \tag{27}$$

where the dot product is defined as in Eq. (25). The selection procedure can be problematic in the common situation in which multiple kernels provide useful complementary features, or when there is no clue on which kernels to try. An effective alternative consists in jointly learning the kernel and the kernel machine based on it (i.e. the coefficients c_i of the kernel combination). In the following we present two rather complementary approaches to this aim: learning a sparse convex combination of basic kernels and learning a logic kernel relying on Inductive Logic Programming techniques.

5.1 Learning Kernel Combinations

It is often the case that multiple different kernels can be defined on a certain domain. Using the closure properties of the kernel class, it is always possible to combine all of them on an equal basis, for instance by direct summation or product. However, this is not necessarily the best choice in case many of them actually provide noisy or redundant features. Learning functions based on a large set of kernels has drawbacks also from an interpretation viewpoint, as it becomes difficult to identify the most relevant features for the discrimination. Recent approaches to kernel learning try to overcome these issues by learning weighted combinations of kernels and forcing sparsity in the weights. The overall kernel becomes a convex combination of *basis* kernels:

$$k(x,x') = \sum_{k=1}^{K} d_k k_k(x,x') \tag{28}$$

where K is the total number of kernels, $d_k \geq 0$ for all m and $\sum_{k=1}^{K} d_k = 1$. Multiple kernel learning (MKL) amounts at jointly learning the coefficients c_i of the overall function f and the weights d_k of the kernel combination. A sparse combination, in which only few d_k are different from zero, is typically enforced using some sparsity-inducing norm like the one-norm. MKL was initially addressed using semidefinite programming techniques [43]. A number of alternative solutions have been later proposed in the literature, especially for boosting efficiency towards large-scale applicability. Here we report a simple and efficient formulation named *SimpleMKL* [62].

Let $f(x) = \sum_{k=1}^{K} f_k(x)$, where each f_k belongs to a different RKHS $\mathcal{H}_k$ with associated kernel k_k. The SimpleMKL formulation addresses the following constrained optimization problem:

$$\begin{aligned} \min_{d} \quad & J(d) \\ \text{subject to:} \quad & \sum_{k=1}^{K} d_k = 1 \\ & d_k \geq 0 \quad k = 1,\ldots,K \end{aligned}$$

where:

$$J(d) = \begin{cases} \min_{f,\xi} & \frac{1}{2}\sum_{k=1}^{K} \frac{1}{d_k} ||f_k||^2_{\mathcal{H}_k} + C\sum_{i=1}^{m} \xi_i \\ \text{subject to:} & y_i \sum_{k=1}^{K} f_k(x_i) \geq 1 - \xi_i \quad i = 1,\ldots,m \\ & \xi_i \geq 0 \quad i = 1,\ldots,m. \end{cases}$$

Solving $J(d)$ for a particular value of d amounts at solving a standard SVM classification problem with a convex combination kernel as in equation (28). To see this, let's compute the Lagrangian and derive the dual formulation as in Section 3.1. The Lagrangian is given by:

$$L(f,\alpha,\beta) = C\sum_{i=1}^{m} \xi_i + \frac{1}{2}\sum_{k=1}^{K} \frac{1}{d_k} ||f_k||^2_{\mathcal{H}_k} - \sum_{i=1}^{m} \alpha_i (y_i \sum_{k=1}^{K} f_k(x_i) - 1 + \xi_i) - \sum_{i=1}^{m} \beta_i \xi_i$$

where $\alpha_i, \beta_i \geq 0$ for all i are the Lagrange multipliers. Zeroing the gradient with respect to the primal variables gives:

$$\begin{aligned} \frac{\partial L}{\partial f_k(\cdot)} &= \frac{1}{d_k} f_k(\cdot) - \sum_{i=1}^{m} \alpha_i y_i k(\cdot, x_i) \quad k = 1,\ldots,K \\ \frac{\partial L}{\partial \xi_i} &= C - \alpha_i - \beta_i = 0 \quad i = 1,\ldots,m \end{aligned}$$

where we used the reproducing property (see eq. (3)) to derive $\frac{\partial f_k(x_i)}{\partial f_k(\cdot)} = \frac{\partial \langle k(\cdot,x_i), f_k(\cdot)\rangle}{\partial f_k(\cdot)} = k(\cdot,x_i)$. Substituting into the Lagrangian we obtain:

$$\max_{\boldsymbol{\alpha}\in\mathbb{R}^m} \quad -\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i y_i \alpha_j y_j \sum_{k=1}^{K} d_k k_k(x_i,x_j) + \sum_{i=1}^{m}\alpha_i$$
$$\text{subject to:} \quad \alpha_i \in [0,C] \quad i = 1,\dots,m$$

which is the standard SVM dual formulation for $k(x_i,x_j)$ as in eq. (28) and can be solved with one of the available SVM solvers. Differentiating J with respect to the weights d gives:

$$\frac{\partial J}{\partial d_k} = -\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i^* y_i \alpha_j^* y_j k_k(x_i,x_j) \quad k = 1,\dots,K$$

where α^* are the solutions of the inner dual maximization problem. This gradient is used to update d along a gradient descent direction which retains its equality and non-negativity constraints. For a detailed description of the optimization algorithm see [62].

5.2 Learning Logical Kernels

Inductive Logic Programming [56] (ILP) amounts at learning a logic hypothesis capable of explaining a set of observations. In the most common setting of binary classification, the hypothesis should cover positive examples and not cover negative ones. More formally, let $\mathscr{B}$ be a logic knowledge base. Let $\mathscr{D} = \{(x_1,y_1),\dots,(x_m,y_m)\}$ be a dataset of input-output pairs, where inputs are identifiers for entities in the knowledge base. A hypothesis H is a set of definite clauses. Let $\mathscr{H}$ be the space of all hypotheses which can be constructed from a language $\mathscr{L}$. A generic ILP algorithm aims at addressing the following maximization problem:

$$\max_{H\in\mathscr{H}} S(H,\mathscr{D},\mathscr{B})$$

where $S(H,\mathscr{D},\mathscr{B})$ is an appropriate scoring function for evaluating the quality of the hypothesis, e.g. accuracy of classification. The hypothesis space is structured by a (partial) generality relation $\preceq$: a hypothesis H_1 is more general than a hypothesis H_2 ($H_1 \preceq H_2$) if and only if any example covered by H_2 is also covered by H_1. Search in the hypothesis space is conducted in a general-to-specific or specific-to-general fashion, using an appropriate *refinement operator* [56, 15]. The computational cost of searching in a discrete space typically forces one to resort to heuristic search algorithms, such as (variations of) greedy search (see [15] for more details).

Figure 10 shows a pair of chemical compounds from the NCI-HIV database which were measured active in inhibiting HIV. Both extensional and intensional knowledge are (partially) reported. Clause `active(M)` $\leftarrow$ `atom(M,A1,o),bond(M,A1,A2,2)`, for instance, covers both compounds, while its specialization `active(M)` $\leftarrow$ `atom(M,A1,o),bond(M,A1,A2,2),atom(M,A2,s)` covers `m1` but not `m2`.

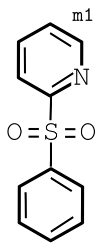
The integration of ILP and statistical learning has the appealing potential of combining the advantages of the respective approaches, namely the expressivity and interpretability of ILP with the effectiveness and efficiency of statistical learning as well as its ability to deal with other tasks than standard binary classification. ILP and kernel machines can be integrated by defining an appropriate kernel based on logic hypotheses. Given that a hypothesis is a set of clauses, the simplest kernel consists of counting the number of clauses which cover both examples. This corresponds to a feature space with one binary feature for each clause.

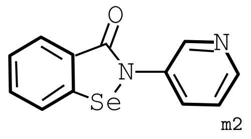
Consider again the example of Figure 10. Let $H = \{c_1, c_2, c_3\}$ be a hypothesis consisting of the following three clauses:

Extensional knowledge

```
atom(m1,a1_1,c).      bond(m1,a1_1,a1_2,ar).
atom(m1,a1_2,c).      bond(m1,a1_1,a1_6,ar).
atom(m1,a1_3,c).      bond(m1,a1_2,a1_3,ar).
....                  ....
atom(m1,a1_13,n).     bond(m1,a1_10,a1_11,ar).
atom(m1,a1_14,o).     bond(m1,a1_11,a1_12,ar).
atom(m1,a1_15,o).     bond(m1,a1_12,a1_13,ar).

atom(m2,a2_1,c).      bond(m2,a2_1,a2_2,ar).
atom(m2,a2_2,c).      bond(m2,a2_1,a2_6,ar).
atom(m2,a2_3,c).      bond(m2,a2_2,a2_3,ar).
....                  ....
atom(m2,a2_14,n).     bond(m2,a2_13,a2_14,ar).
atom(m2,a2_15,c).     bond(m2,a2_14,a2_15,ar).
atom(m2,a2_16,c).     bond(m2,a2_15,a2_16,ar).
```





Intensional knowledge

```
benzene(Drug,List) ←
   ring6(Drug,List,[ar,ar,ar,ar,ar,ar]),
   atoms(Drug,6,List,[c,c,c,c,c,c]).

hetero_aromatic_6_ring(Drug,List) ←
   ring6(Drug,List,[ar,ar,ar,ar,ar,ar]),
   hetero(Drug,List).

hetero(Drug,List) ←
   member(A,List),
   atom(Drug,A,T),
   T ≠ c.
```

Fig. 10 Example of logical representations for the NCI-HIV dataset of compounds. Extensional knowledge encodes the atom-bond representation of molecules: `atom(m,a,t)` indicates that molecule `m` has atom `a` which is a `t` chemical element; `bond(m,a1,a2,t)` indicates a chemical bond of type `t` (e.g. `ar` for aromatic bond) between atoms `a1` and `a2` of molecule `m`. Intensional knowledge encodes functional groups such as benzene and other aromatic rings. An aromatic ring is a ring of atoms connected by aromatic bonds. A benzene is an aromatic ring of six carbon atoms. A hetero aromatic ring is an aromatic ring containing at least one non-carbon atom.

```
active(M) ← atom(M,A1,o),bond(M,A1,A2,2),atom(M,A2,s).
active(M) ← hetero_aromatic_6_ring(M,List),member(A,List),atom(M,A,n).
active(M) ← benzene(M,[A1,A2,A3,A4,A5,A6]),bond(M,A5,A10,ar),atom(M,A10,se).
```

Clauses c_1 and c_2 succeed on molecule `m1`, while clauses c_2 and c_3 succeed on molecule `m2`. The resulting feature space representation according to the up-mentioned kernel is given by:

$$\Phi_H(\mathtt{m1}) = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \qquad \Phi_H(\mathtt{m2}) = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}.$$

More complex kernels can be obtained using kernel composition (e.g. polynomial or Gaussian kernel, see Section 4.2).

Given a kernel over logical hypothesis, we can construct a prediction function as:

$$f(x;H,\mathscr{B}) = \sum_{i=1}^{m} c_i k(x,x_i;H,\mathscr{B}).$$

The generic maximization problem becomes:

$$\max_{H \in \mathscr{H}} \max_{f \in \mathscr{F}_H} S(f,\mathscr{D},\mathscr{B})$$

where $\mathscr{F}_H$ is the set of all functions that can be generated with hypothesis H. Learning these logic kernel machines [45] amounts at jointly learning the kernel, in terms of the logic hypothesis H, and the function in the RKHS associated with it. Learning convex combinations of basic kernel functions, as seen in Section 5.1, can be cast into constrained optimization problems for which efficient algorithms exist. Conversely, here we face the discrete space of logic hypotheses and heuristic search algorithms need to be employed.

kFOIL [44] is a simple example of this paradigm, based on an adaptation of the well-known FOIL algorithm [60]. The algorithm is briefly sketched in Algorithm 1. It repeatedly searches for clauses that score well with respect to the data set and the current hypothesis and adds them to the current hypothesis. In the inner loop, kFOIL greedily searches for a clause that scores well. To this aim, it employs a general-to-specific hill-climbing search strategy. Let $p/^n$ denote the predicate that is being learned. Then the most general clause, which succeeds on all examples, is "$p(X_1,...,X_n) \leftarrow$". The set of all refinements of a clause

Algorithm 1 kFOIL learning algorithm.

```
Initialize $H := \emptyset$
repeat
    Initialize $c := p(X_1, \cdots, X_n) \leftarrow$
    repeat
        $c := \mathrm{argmax}_{c' \in \rho(c)} \max_{f \in \mathscr{F}_{H \cup \{c'\}}} S(f, \mathscr{D}, \mathscr{B})$
    until stopping criterion
    $H := H \cup \{c\}$
until stopping criterion
return $\mathrm{argmax}_{f \in \mathscr{F}_H} S(f, \mathscr{D}, \mathscr{B})$
```

c is produced by a refinement operator $\rho(c)$. For our purposes, a refinement operator specializes a clause $h \leftarrow b_1, \cdots, b_k$ by adding new literals b_{k+1} to the clause, though other refinements have also been used in the literature. Scoring a clause amounts at training a kernel machine using the current hypothesis, including the candidate clause, and returning a measure of its performance (e.g. its accuracy on the training set). Learning in kFOIL is stopped when there is no improvement in score between two successive iterations. Several extensions and improvements have been proposed. Substantial efficiency gains, for instance, can be obtained using a kernel quality measure such as kernel target alignment (see Eq. (27)) to score clauses, instead of training a kernel machine every time. Further details and extensions can be found in [45].

6 Supervised Kernel Machines for Structured Output

Many relevant learning problems require to predict outputs which are themselves structures. Speech recognition or protein secondary structure prediction, for instance, can eventually be formalized as sequential labelling tasks. A key problem in NLP consists in predicting the parse tree of a sentence. Many techniques have been developed in the literature to address this kind of problems, often based on directed or undirected graphical models. A quite general approach consists of learning a function $f(x,y)$ over both input and output which basically evaluates the quality of y as an output for x (e.g. the conditional probability of the output given the input i.e. $P(Y = y|X = x)$). Prediction then amounts at finding the output maximizing this score, i.e.:

$$y^* = \underset{y \in \mathscr{Y}}{\mathrm{argmax}}\, f(x,y). \tag{29}$$

Kernel machines can be generalized to this kind of setting by defining a *joint feature map* $\Phi(x,y)$ over both input and output, with corresponding kernel $k((x,y),(x',y'))$. A common approach consists of defining basic feature maps for input and output components. The joint feature map is then obtained as a combination of tensor products and direct sums (see Section 4.2) involving these basic maps. Multiclass classification can be obtained for instance setting:

$$\Phi(x,y) = \Phi(x) \otimes \hat{\Phi}(y) \tag{30}$$

where $\hat{\Phi}(y)$ is the one-hot encoding of the class label. A sequential labelling algorithm encoding Markov chain assumptions for dependencies would use a feature mapping like:

$$\Phi(x,y) = \left[\sum_{t=1}^{T-1} \hat{\phi}(y_t) \otimes \hat{\phi}(y_{t+1})\right] \odot \left[\sum_{t=1}^{T} \phi(x_t) \otimes \hat{\phi}(y_t)\right]. \tag{31}$$

Figure 11 shows an example in which the sequential labelling task is predicting coding (exons) and non-coding (introns) regions in the genome. The joint feature map is obtained from Eq. (31) using one-hot encoding for the input and output basic mappings. More complex kernels on structures can also be employed in principle, (see Section 4.3). Note however that limitations to the form of the usable kernels are often needed in order to retain efficiency in computing the argmax in Eq. (29).

$$\begin{array}{l}\mathtt{x}\ =\ \mathtt{ACCCCGGGTTTTAA}\\ \mathtt{y}\ =\ \mathtt{EEEIIIIEEEIIEE}\end{array}\qquad \Phi(x,y)=\begin{pmatrix}5\\2\\2\\4\\3\\2\\1\\2\\0\\2\\2\\2\end{pmatrix}\begin{array}{ll}\left.\begin{array}{l}\mathrm{EE}\\ \mathrm{EI}\\ \mathrm{IE}\\ \mathrm{II}\end{array}\right\} & \left[\sum_{t=1}^{T-1}\hat{\phi}(y_t)\otimes\hat{\phi}(y_{t+1})\right]\\ \left.\begin{array}{l}\mathrm{AE}\\ \mathrm{CE}\\ \mathrm{GE}\\ \mathrm{TE}\\ \mathrm{AI}\\ \mathrm{CI}\\ \mathrm{GI}\\ \mathrm{TI}\end{array}\right\} & \left[\sum_{t=1}^{T}\phi(x_t)\otimes\hat{\phi}(y_t)\right]\end{array}$$

Fig. 11 Example of joint feature map for sequential labelling. One-hot encoding is used for feature mappings of both input and output symbols.

A generalized version of the representer theorem gives the form of the solution of Tikhonov regularized problems.

Theorem 3 (Generalized Representer Theorem). *Let $D_m = \{(x_i,y_i) \in \mathscr{X} \times \mathscr{Y}\}_{i=1}^m$ be a training set, $V(x_i,y_i,f)$ a general loss function, $\mathscr{H}$ a RKHS with norm $||\cdot||_{\mathscr{H}}$. Then the general form of the solution of the regularized risk*

$$\frac{1}{m}\sum_{i=1}^{m} V(x_i,y_i,f) + \lambda ||f||^2_{\mathscr{H}}$$

is

$$f(x,y) = \sum_{i=1}^{m}\sum_{y'\in\mathscr{Y}} c_{iy'}k((x_i,y'),(x,y)).$$

The solution is as a linear combination of kernel functions centered on “augmented” training examples (x_i,y'), where x_i are training inputs and $y' \in \mathscr{Y}$ possible outputs.

Structured-Output Support Vector Machines [77] (SO-SVM) generalize large-margin classification to this setting, by enforcing a large separation between correct and incorrect output assignments. A hinge loss for structured-output prediction can be written as:

$$V(x,y,f) = |1-(f(x,y) - \operatorname*{argmax}_{y'\neq y} f(x,y'))|_+.$$

The resulting Tikhonov regularized functional is:

$$\min_{f\in\mathscr{H}} \frac{1}{m}\sum_{i=1}^{m} |1-(f(x,y) - \operatorname*{argmax}_{y'\neq y} f(x,y'))|_+ + \lambda ||f||^2_{\mathscr{H}}.$$

As for the scalar case, f can be written as a dot product between the feature space representation of the example and a parameter vector, i.e.:

$$f(x,y) = \sum_{i=1}^{m}\sum_{y'\in\mathscr{Y}} c_{iy'}\langle\Phi(x_i,y'),\Phi(x,y)\rangle = \langle\sum_{i=1}^{m}\sum_{y'\in\mathscr{Y}} c_{iy'}\Phi(x_i,y'),\Phi(x,y)\rangle = \langle\boldsymbol{w},\Phi(x,y)\rangle.$$

By introducing slack ξ_i for the cost paid for each incorrect prediction we obtain the following quadratic optimization problem:

$$\begin{aligned}
&\min_{\boldsymbol{w}\in\mathscr{H},\boldsymbol{\xi}\in\mathbb{R}^m} \quad C\sum_{i=1}^m \xi_i + \frac{1}{2}||\boldsymbol{w}||^2 \\
&\text{subject to:} \quad \langle \boldsymbol{w}, \Phi(x_i,y_i)\rangle - \operatorname*{argmax}_{y'\neq y_i}\langle \boldsymbol{w}, \Phi(x_i,y')\rangle \geq 1-\xi_i \quad i=1,\dots,m \\
&\qquad\qquad\quad \xi_i \geq 0 \quad i=1,\dots,m
\end{aligned}$$

where again we replaced $C = 2/\lambda m$ for consistency with most literature on SO-SVM. Multiclass SVM [12] can be seen as a simple instantiation of this optimization problem, with a joint feature map as in Eq. (30). Note that the margin constraint for each training example can be equivalently replaced with a set of constraints, one for each possible output y, all sharing the same slack variable ξ_i. The Langragian of the optimization problem becomes:

$$L(\boldsymbol{w},\boldsymbol{\alpha},\boldsymbol{\beta}) = C\sum_{i=1}^m \xi_i + \frac{1}{2}||\boldsymbol{w}||^2 - \sum_{i=1}^m\sum_{y'\neq y_i}\alpha_{iy'}(\langle \boldsymbol{w},\Phi(x_i,y_i)\rangle - \langle \boldsymbol{w},\Phi(x_i,y')\rangle - 1 + \xi_i) - \sum_{i=1}^m \beta_i\xi_i$$

where $\alpha_{iy'},\beta_i \geq 0$ for all i and y'. Zeroing the derivatives with respect to the primal variables gives:

$$\begin{aligned}
\frac{\partial L}{\partial \boldsymbol{w}} &= \boldsymbol{w} - \sum_{i=1}^m\sum_{y'\neq y_i}\alpha_{iy'}\underbrace{\Phi(x_i,y_i)-\Phi(x_i,y')}_{\delta\Phi_i(y')} = 0 \rightarrow \boldsymbol{w} = \sum_{i=1}^m\sum_{y'\neq y_i}\alpha_{iy'}\delta\Phi_i(y') \\
\frac{\partial L}{\partial \xi_i} &= C - \sum_{y'\neq y_i}\alpha_i - \beta_i = 0 \rightarrow \sum_{y'\neq y_i}\alpha_i \in [0,C]
\end{aligned}$$

where we replaced $\delta\Phi_i(y') = \Phi(x_i,y_i) - \Phi(x_i,y')$ for compactness. Substituting into the Lagrangian we obtain:

$$\begin{aligned}
&\max_{\boldsymbol{\alpha}\in\mathbb{R}^m} \quad -\frac{1}{2}\sum_{i=1}^m\sum_{j=1}^m\sum_{y'\neq y_i}\sum_{y''\neq y_j}\alpha_{iy'}\alpha_{jy''}\langle\delta\Phi_i(y'),\delta\Phi_j(y'')\rangle + \sum_{i=1}^m\sum_{y'\neq y_i}\alpha_{iy'} \qquad (32)\\
&\text{subject to:} \quad \sum_{y'\neq y_i}\alpha_{iy'} \in [0,C] \quad i=1,\dots,m.
\end{aligned}$$

Note that replacing $\langle\delta\Phi_i(y'),\delta\Phi_j(y'')\rangle = \langle\Phi(x_i,y_i)-\Phi(x_i,y'),\Phi(x_j,y_j)-\Phi(x_j,y'')\rangle = k((x_i,y_i),(x_j,y_j)) - k((x_i,y_i),(x_j,y'')) - k((x_i,y'),(x_j,y_j)) + k((x_i,y'),(x_j,y''))$ we recover the kernel-based formulation where the feature mapping is only implicitly done.

A main problem in optimizing (32) is the number of variables α, which is often exponential in the size of the output. The problem is addressed using a cutting plane algorithm, which iteratively solves larger optimization problems obtained incrementally adding the most violated constraint. Algorithm 2 reports a sketch of the algorithm. Starting from an empty set of constraints, the algorithm repeatedly iterates over training examples. For each training example the most violated constraint according to the current version of f is computed ($H(y)$ is the cost paid for predicting y). Its cost is compared with that of previous constraints involving the same example (or zero if none exists). If the new cost is larger by more than a user-defined tolerance ε, the constraint is added and a new optimization problem is solved. The algorithm terminates when no further constraint is added for any of the training examples.

In both training and classification phases, the argument maximizing f needs to be returned. The efficiency of this computation is crucial to the applicability of the approach. Dynamic programming techniques can be employed in a number of common cases, like the Viterbi algorithm for sequential labelling or its extension to probabilistic context free grammars for predicting parse trees.

A number of variants of the approach described here exist. A common extension consists in adding a loss function $\Delta(y_i,y')$ to the constraints, measuring the loss incurred in predicting y' in place of y. Alternative approaches have also been proposed in the literature, for instance to address cases in which exact com-

Algorithm 2 Cutting plane algorithm for SO-SVM.

```
Initialize $S_i := \emptyset$ for all $i$
repeat
    for $i = 1, \dots, m$ do
        $H(y) = 1 - \langle \boldsymbol{w}, \delta\Phi_i(y) \rangle$
        where $w = \sum_j \sum_{y' \in S_j} \alpha_{jy'} \delta\Phi_j(y')$
        compute $\hat{y} = \mathrm{argmax}_{y \neq y_i} H(y)$
        compute $\xi_i = \max\{0, \max_{y \in S_i} H(y)\}$
        if $H(\hat{y}) > \xi_i + \varepsilon$ then
            $S_i := S_i \cup \{\hat{y}\}$
            solve problem (32) restricted to variables $S = \bigcup_i S_i$
        end if
    end for
until no $S_i$ has changed during iteration
```

putation of argmax is intractable. For a comprehensive treatment of kernel methods for structured-output prediction see [5].

7 Conclusions

In this chapter we provided a comprehensive introduction to kernel machines for structured data. We reviewed the mathematical foundations underlying kernel methods and described a number of popular kernel machines for both supervised and unsupervised learning. We gave an extensive treatment of kernels for structured data, including strings, trees and graphs, as well as kernels based on generative models and logical representations. Techniques for learning kernels from data were also discussed. Finally we introduced kernel machines for predicting complex output structures, a promising research direction combining kernel methods with probabilistic graphical models and search-based approaches.

Research on kernel methods is extremely active and a large number of diverse problems have been tackled under this formalism. This chapter is aimed at providing a clear and detailed explanation on kernel methods and their use for dealing with structured data, rather than a complete survey on all approaches which have been developed. Techniques which were not covered include Gaussian Processes [63] and distribution embeddings. The former take a Bayesian viewpoint and allow to provide predictions in terms of expected value and variance, where uncertainty is reduced in the proximity of observed points. The latter is a recent trend of research aimed at embedding probability distributions in RKHS. This allows to develop effective non-parametric techniques for problems like testing whether two samples come from the same distribution [25] or measuring the strength of dependency between two variables [26]. Additional references to advanced material on kernel methods can be found e.g. in [31].

References

1. Fabio Aiolli, Giovanni Da San Martino, and Alessandro Sperduti. Route kernels for trees. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 17–24, New York, NY, USA, 2009. ACM.
2. S.I. Amari. Mathematical foundations of neurocomputing. *Proceedings of the IEEE*, 78(9):1443–1463, 1990.
3. S.I. Amari. Natural gradient works efficiently in learning. *Neural Computation*, 10:251–276, 1998.
4. N. Aronszajn. Theory of reproducing kernels. *Trans. Amer. Math. Soc.*, 686:337–404, 1950.
5. Gükhan H. Bakir, Thomas Hofmann, Bernhard Schölkopf, Alexander J. Smola, Ben Taskar, and S. V. N. Vishwanathan. *Predicting Structured Data (Neural Information Processing)*. The MIT Press, 2007.
6. C. Berg, J.P.R. Christensen, and P. Ressel. *Harmonic Analysis on Semigroups*. Springer-Verlag, New York, 1984.
7. K. M. Borgwardt. *Graph Kernels*. PhD thesis, Ludwig-Maximilians-University Munich, 2007.
8. Karsten M. Borgwardt and Hans-Peter Kriegel. Shortest-path kernels on graphs. In *Proceedings of the Fifth IEEE International Conference on Data Mining*, ICDM '05, pages 74–81, Washington, DC, USA, 2005. IEEE Computer Society.

9. B.E. Boser, I.M. Guyon, and V.N. Vapnik. A training algorithm for optimal margin classifier. In *Proc. 5th ACM Workshop on Computational Learning Theory*, pages 144–152, Pittsburgh, PA, July 1992.
10. M. Collins and N. Duffy. Convolution kernels for natural language. In T.G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14*. MIT Press, 2002.
11. M. Collins and N. Duffy. New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL 2002)*, pages 263–270, Philadelphia, PA, USA, 2002.
12. K. Crammer and Y. Singer. On the algorithmic implementation of multiclass kernel-based vector machines. *J. Mach. Learn. Res.*, 2:265–292, 2002.
13. N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines*. Cambridge University Press, 2000.
14. Nello Cristianini, Jaz Kandola, Andre Elisseeff, and John Shawe-Taylor. On kernel-target alignment. In *Advances in Neural Information Processing Systems 14*, volume 14, pages 367–373, 2002.
15. L. De Raedt. *Logical and Relational Learning*. Springer-Verlag, 2008.
16. R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.
17. R. Fletcher. *Practical Methods of Optimization, Second Edition*. John Wiley & Sons, 1987.
18. P. Frasconi and A. Passerini. Learning with kernels and logical representations. In Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton, editors, *Probabilistic inductive logic programming*, pages 56–91. Springer-Verlag, Berlin, Heidelberg, 2008.
19. T. Gärtner. Exponential and geometric kernels for graphs. In *NIPS Workshop on Unreal Data: Principles of Modeling Nonvectorial Data*, 2002.
20. T. Gärtner, P. Flach, A. Kowalczyk, and A.J. Smola. Multi-instance kernels. In C.Sammut and A. Hoffmann, editors, *Proceedings of the* 19^{th} *International Conference on Machine Learning*, pages 179–186. Morgan Kaufmann, 2002.
21. T. Gärtner, J.W. Lloyd, and P.A. Flach. Kernels for structured data. In S. Matwin and C. Sammut, editors, *Proceedings of the 12th International Conference on Inductive Logic Programming*, volume 2583 of *Lecture Notes in Artificial Intelligence*, pages 66–83. Springer-Verlag, 2003.
22. Thomas Gärtner. *Kernels for Structured Data*. PhD thesis, Universität Bonn, 2005.
23. Thomas Gärtner, Peter Flach, and Stefan Wrobel. On Graph Kernels: Hardness Results and Efficient Alternatives. In Bernhard Schölkopf and Manfred Warmuth, editors, *Learning Theory and Kernel Machines*, volume 2777 of *Lecture Notes in Computer Science*, pages 129–143, Berlin, Heidelberg, 2003. Springer Berlin / Heidelberg.
24. Thomas Gärtner, John W. Lloyd, and Peter A. Flach. Kernels and distances for structured data. *Mach. Learn.*, 57:205–232, December 2004.
25. A. Gretton, K. M. Borgwardt, M. J. Rasch, B. Schölkopf, and A. J. Smola. A kernel method for the two-sample-problem. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 513–520. MIT Press, Cambridge, MA, 2007.
26. A. Gretton, K. Fukumizu, C.H. Teo, L. Song, B. Schölkopf, and A. J. Smola. A kernel statistical test of independence. In J.C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*. MIT Press, Cambridge, MA, 2008.
27. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
28. Jihun Ham, Daniel D. Lee, Sebastian Mika, and Bernhard Schölkopf. A kernel view of the dimensionality reduction of manifolds. In *Proceedings of the twenty-first international conference on Machine learning*, ICML '04, pages 47–, New York, NY, USA, 2004. ACM.
29. D. Haussler. Convolution kernels on discrete structures. Technical Report UCSC-CRL-99-10, University of California, Santa Cruz, 1999.
30. H. Hoffmann. Kernel pca for novelty detection. *Pattern Recogn.*, 40:863–874, March 2007.
31. T. Hofmann, B. Schölkopf, and A. J. Smola. Kernel methods in machine learning. *Annals of Statistics*, 36(3):1171–1220, 2008.
32. Tamás Horváth, Thomas Gärtner, and Stefan Wrobel. Cyclic pattern kernels for predictive graph mining. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '04, pages 158–167, New York, NY, USA, 2004. ACM.
33. T. Jaakkola, M. Diekhans, and D. Haussler. A discriminative framework for detecting remote protein homologies. *Journal of Computational Biology*, 7(1–2):95–114, 2000.
34. T. Jaakkola and D. Haussler. Probabilistic kernel regression models. In *Proc. of Neural Information Processing Conference*, 1998.
35. T. Jaakkola and D. Haussler. Exploiting generative models in discriminative classifiers. In *Proceedings of the 1998 conference on Advances in neural information processing systems II*, pages 487–493, Cambridge, MA, USA, 1999. MIT Press.
36. Tony Jebara, Risi Kondor, and Andrew Howard. Probability product kernels. *J. Mach. Learn. Res.*, 5:819–844, December 2004.
37. T. Joachims. Making large-scale SVM learning practical. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods – Support Vector Learning*, chapter 11, pages 169–185. MIT Press, 1998.
38. Hisashi Kashima, Koji Tsuda, and Akihiro Inokuchi. Marginalized kernels between labeled graphs. In *Proceedings of the Twentieth International Conference on Machine Learning*, pages 321–328. AAAI Press, 2003.

39. S. S. Keerthi, K. B. Duan, S. K. Shevade, and A. N. Poo. A fast dual algorithm for kernel logistic regression. *Mach. Learn.*, 61:151–165, November 2005.
40. K. Kim, M. O. Franz, and B. Schölkopf. Iterative kernel principal component analysis for image modeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(9):1351–1366, September 2005.
41. G. Kimeldorf and G. Wahba. Some results on tchebycheffian spline functions. *J. Math. Anal. Applic.*, 33:82–95, 1971.
42. R.I. Kondor and J. Lafferty. Diffusion kernels on graphs and other discrete input spaces. In C.Sammut and A. Hoffmann, editors, *Proc. of the* 19^{th} *International Conference on Machine Learning*, pages 315–322. Morgan Kaufmann, 2002.
43. Gert R. G. Lanckriet, Nello Cristianini, Peter Bartlett, Laurent El Ghaoui, and Michael I. Jordan. Learning the kernel matrix with semidefinite programming. *J. Mach. Learn. Res.*, 5:27–72, December 2004.
44. Niels Landwehr, Andrea Passerini, Luc De Raedt, and Paolo Frasconi. kfoil: learning simple relational kernels. In *Proceedings of the 21st national conference on Artificial intelligence - Volume 1*, pages 389–394. AAAI Press, 2006.
45. Niels Landwehr, Andrea Passerini, Luc Raedt, and Paolo Frasconi. Fast learning of relational kernels. *Mach. Learn.*, 78:305–342, March 2010.
46. C. Leslie, E. Eskin, and W.S. Noble. The spectrum kernel: a string kernel for svm protein classification. In *Proc. of the Pacific Symposium on Biocomputing*, pages 564–575, 2002.
47. C. Leslie, E. Eskin, J. Weston, and W.S. Noble. Mismatch string kernels for svm protein classification. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, pages 1417–1424. MIT Press, Cambridge, MA, 2003.
48. C. Leslie, R. Kuang, and E. Eskin. Inexact matching string kernels for protein classificatio. In B. Schölkopf, K. Tsuda, and J.-P. Vert, editors, *Kernel Methods in Computational Biology*. MIT Press, 2004. In press.
49. H. Lodhi, J. Shawe-Taylor, N. Cristianini, and C. Watkins. Text classification using string kernels. In *Advances in Neural Information Processing Systems*, pages 563–569, 2000.
50. G. Da San Martino. *Kernel Methods for Tree Structured Data*. PhD thesis, Department of Computer Science, University of Bologna, 2009.
51. Sauro Menchetti. *Learning Preference and Structured Data: Theory and Applications*. PhD thesis, Dipartimento di Sistemi e Informatica, DSI, Università di Firenze, Italy, December 2005.
52. Sauro Menchetti, Fabrizio Costa, and Paolo Frasconi. Weighted decomposition kernels. In *Proceedings of the 22nd international conference on Machine learning*, ICML '05, pages 585–592, New York, NY, USA, 2005. ACM.
53. J. Mercer. Functions of positive and negative type and their connection with the theory of integral equations. *Philos. Trans. Roy. Soc. London*, A 209:415–446, 1909.
54. C. A. Micchelli, Y. Xu, and H. Zhang. Universal kernels. *J. Mach. Learn. Res.*, 7:2651–2667, 2006.
55. A. Moschitti. Efficient convolution kernels for dependency and constituent syntactic trees. In *ECML*, pages 318–329. Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Proceedings, September 2006.
56. S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.
57. A. Passerini, P. Frasconi, and L. De Raedt. Kernels on prolog proof trees: Statistical learning in the ILP setting. *Journal of Machine Learning Research*, 7:307–342, 2006.
58. J. C. Platt. Fast training of support vector machines using sequential minimal optimization. In C. Burges B. Schölkopf, editor, *Advances in Kernel Methods–Support Vector Learning*. MIT press, 1998.
59. T. Poggio and S. Smale. The mathematics of learning: Dealing with data. *Notices of the American Mathematical Society*, 50(5):537–544, 2003.
60. J. R. Quinlan. Learning Logical Definitions from Relations. *Machine Learning*, 5:239–266, 1990.
61. L.R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.
62. Alain Rakotomamonjy, Francis R. Bach, St&#233;phane Canu, and Yves Grandvalet. SimpleMKL. *Journal of Machine Learning Research*, 9:2491–2521, November 2008.
63. Carl E. Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, December 2005.
64. S. Saitoh. *Theory of Reproducing Kernels and its Applications*. Longman Scientific Technical, Harlow, England, 1988.
65. G. Saunders, A. Gammerman, and V. Vovk. Ridge regression learning algorithm in dual variables. *Proc. 15th International Conf. on Machine Learning*, pages 515–521, 1998.
66. B. Schölkopf, J.C. Platt, J. Shawe-Taylor, A.J. Smola, and R.C. Williamson. Estimating the support of a high dimensional distribution. *Neural Computation*, 13:1443–1471, 2001.
67. B. Schölkopf, A. Smola, and K.R. Müller. Kernel principal component analysis. In *Advances in Kernel Methods–Support Vector Learning*, pages 327–352. MIT press, 1999.
68. B. Schölkopf and A.J. Smola. *Learning with Kernels*. The MIT Press, Cambridge, MA, 2002.
69. B. Schölkopf and M.K. Warmuth, editors. *Kernels and Regularization on Graphs*, volume 2777 of *Lecture Notes in Computer Science*. Springer, 2003.
70. Bernhard Schölkopf, Alex J. Smola, Robert C. Williamson, and Peter L. Bartlett. New support vector algorithms. *Neural Comput.*, 12:1207–1245, May 2000.
71. John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, New York, NY, USA, 2004.
72. Kilho Shin and Tetsuji Kuboyama. A generalization of haussler's convolution kernel: mapping kernel. In *Proceedings of the 25th international conference on Machine learning*, ICML '08, pages 944–951, New York, NY, USA, 2008. ACM.

73. Leon Sterling and Ehud Shapiro. *The art of Prolog (2nd ed.): advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1994.
74. S. Joshua Swamidass, Jonathan Chen, Jocelyne Bruand, Peter Phung, Liva Ralaivola, and Pierre Baldi. Kernels for small molecules and the prediction of mutagenicity, toxicity and anti-cancer activity. *Bioinformatics*, 21:359–368, January 2005.
75. D.M.J. Tax and R.P.W. Duin. Support vector domain description. *Pattern Recognition Letters*, 20:1991–1999, 1999.
76. A.N. Tikhonov. On solving ill-posed problem and method of regularization. *Dokl. Akad. Nauk USSR*, 153:501–504, 1963.
77. I. Tsochantaridis, T. Joachims, T. Hofmann, and Y. Altun. Large margin methods for structured and interdependent output variables. *jmlr*, 6:1453–1484, 2005.
78. Koji Tsuda, Taishin Kin, and Kiyoshi Asai. Marginalized kernels for biological sequences. *Bioinformatics*, 18(suppl 1):S268–S275, 2002.
79. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.
80. V.N. Vapnik. *Statistical Learning Theory*. Wiley, New York, 1998.
81. S. V. N. Vishwanathan, Nicol N. Schraudolph, Risi Kondor, and Karsten Borgwardt. Graph kernels. *Journal of Machine Learning Research*, 11:1201–1242, 2010.
82. S. V. N. Vishwanathan and Alex Smola. Fast Kernels for String and Tree Matching. *Advances in Neural Information Processing Systems*, 15, 2003.
83. G. Wahba. *Splines Models for Observational Data*. Series in Applied Mathematics, Vol. 59, SIAM, Philadelphia, 1990.
84. Nikil Wale, Ian A. Watson, and George Karypis. Comparison of descriptor spaces for chemical compound retrieval and classification. *Knowl. Inf. Syst.*, 14:347–375, March 2008.
85. C. Watkins. Dynamic alignment kernels. In A.J. Smola, P. Bartlett, B. Schölkopf, and D. Schuurmans, editors, *Advances in Large Margin Classiers*, pages 39–50. MIT Press, 2000.