A Survey of Kernels for Structured Data

Thomas Gärtner
Fraunhofer Institut Autonome Intelligente Systeme, Germany;
Department of Computer Science, University of Bristol, United Kingdom; and
Department of Computer Science III, University of Bonn, Germany
thomas.gaertner@ais.fraunhofer.de

ABSTRACT

Kernel methods in general and support vector machines in particular have been successful in various learning tasks on data represented in a single table. Much 'real-world' data, however, is structured – it has no natural representation in a single table. Usually, to apply kernel methods to 'real-world' data, extensive pre-processing is performed to embed the data into a real vector space and thus in a single table. This survey describes several approaches of defining positive definite kernels on structured instances directly.

Keywords

Kernel methods, structured data, multi-relational data mining, inductive logic programming

1. INTRODUCTION

Most 'real-world' data has no natural representation as a single table, yet most data mining research has so far concentrated on discovering knowledge from single tables. In order to apply 'traditional' data mining methods to structured data, extensive pre-processing has to be performed. Research in inductive logic programming and multi-relational data mining [8] aims to reduce these pre-processing efforts by considering learning from multi-relational data descriptions directly. Some algorithms developed in that field are upgrades of algorithms, originally developed only with single tables data in mind. Among the so far upgraded algorithms are popular data mining methods such as decision trees, rule learners, and distance-based algorithms.

Support vector machines [3; 41] are among the most successful recent developments within the machine learning and data mining communities. Along with some other learning algorithms like Gaussian processes and kernel principal component analysis, they form the class of kernel methods [32; 37]. The computational attractiveness of kernel methods comes from the fact that they can be applied in high dimensional feature spaces without suffering the high cost of explicitly computing the mapped data. The *kernel trick* is to define a positive definite kernel on any set. For such functions it is know that there exists an embedding of the set in a linear space such that the kernel on the elements of the set corresponds to the inner product in this space.

While the inductive logic programming community has traditionally used logic programs to represent structured data, the scope has now extended and also includes other knowledge representation languages. Development of kernels for structured data has mostly been motivated and guided by 'real-world' problems. Although the structure of these problems is often such that they do not permit a natural representation in a single table, the full power of logic programs is hardly ever needed. This is one reason, kernel design has concentrated on different and sometimes less powerful knowledge representations.

This paper is not intended as an introduction to kernel-based learning algorithms, for such an introduction the reader is referred to one of the excellent books [5; 37] or tutorials [2; 32] on kernel methods ([2] is available online). Instead, this paper intends to give an introduction to kernel functions defined on structured data. For that, we assume some basic familiarity with linear algebra.

In this paper we distinguish kernels according to whether they are defined on the structure of the instances (syntax-driven kernels), or on the structure of the instance space (model-driven kernels). Furthermore, we distinguish according to the power of the knowledge representation used. The outline of the paper is as follows: Section 2 introduces kernel functions and characterizes valid and good kernels. It also presents a classification scheme for kernel functions. Model-driven kernels are then described in section 3. After that syntax-driven kernels are described in section 4. Finally, section 5 concludes .

2. KERNEL FUNCTIONS

Two components of kernel methods have to be distinguished: the kernel machine and the kernel function. While the kernel machine encapsulates the learning task and the way in which a solution is looked for, the kernel function encapsulates the hypothesis language, i.e., how the set of possible solutions is made up. Different kernel functions implement different hypothesis spaces or even different knowledge representations.

2.1 Kernels on Vectors

Before we give the definition of positive definite kernels, the traditionally used kernels (on vector spaces) are briefly reviewed in this section. Let $x, x' \in \mathbb{R}^n$ and let $\langle\cdot,\cdot\rangle$ denote the scalar product in $\mathbb{R}^n$. Apart from the linear kernel

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

and the normalized linear kernel

$$k(x, x') = \frac{\langle x, x' \rangle}{\sqrt{\langle x, x \rangle \langle x', x' \rangle}}$$

which corresponds to the cosine of the angle enclosed by the vectors, the two most frequently used kernels on vector spaces are the polynomial kernel and the Gaussian kernel. Given two parameters $l \in \mathbb{R}, p \in \mathbb{N}^+$ the polynomial kernel is defined as:

$$k(x, x') = (\langle x, x' \rangle + l)^p$$

The intuition behind this kernel definition is that it is often useful to construct new features as products of original features. This way for example the XOR problem can be turned into a linearly separable problem. The p in above definition is the maximal order of monomials making up the new feature space, the l is a bias towards lower order monomial. If $l = 0$ the feature space consists only of monomials of order p of the original features.

EXAMPLE: Consider the positive examples $(+1, +1), (-1, -1)$ and the negative examples $(+1, -1), (-1, +1)$. Clearly, there is no straight line separating positive from negative examples. However, if we use the transformation $\phi : (x_1, x_2) \to (x_1^2, \sqrt{2}x_1x_2, x_2^2)$ separation is possible with the plane orthonormal to $(0, 1, 0)$ as the the sign of x_1x_2 already corresponds to the class. To see that this transformation can be performed implicitly by a polynomial kernel, let $x = (x_1, x_2), z = (z_1, z_2)$ and $k(x, z) = \langle x, z \rangle^2$. Then

$$\begin{aligned} k(x, z) &= \langle (x_1, x_2), (z_1, z_2) \rangle^2 = (x_1z_1 + x_2z_2)^2 \\ &= (x_1z_1)^2 + 2x_1x_2z_1z_2 + (x_2z_2)^2 \\ &= \left\langle \left(x_1^2, \sqrt{2}x_1x_2, x_2^2\right), \left(z_1^2, \sqrt{2}z_1z_2, z_2^2\right) \right\rangle \\ &= \langle \phi(x), \phi(z) \rangle \end{aligned}$$

Now, let $x^+(x^-)$ be either of the positive (negative) examples given above. Any example x can be classified without explicitly transforming instances, as $k(x, x^+) - k(x, x^-)$ corresponds to implicitly projecting x onto the vector $\phi(x^+) - \phi(x^-) = (0, \sqrt{2}, 0)$.

Given the bandwidth parameter σ the Gaussian kernel is defined as:

$$k(x, x') = e^{-||x - x'||^2/\sigma^2}$$

Using this kernel function in a support vector machine can be seen as using a radial basis function network with Gaussian kernels centered at the support vectors. The images of the points from the vector space $\mathbb{R}^n$ under the map $\phi : \mathbb{R}^n \to \mathcal{H}$ with $k(x, x') = \langle \phi(x), \phi(x') \rangle$ lie all on the surface of a hyperball in the Hilbert space $\mathcal{H}$. No two images are orthogonal and any set of images is linearly independent. The parameter σ can be used to tune how much generalization is done. For very high σ, all vectors $\phi(x)$ are almost parallel and thus almost identical. For very small σ, the vectors $\phi(x)$ are almost orthogonal to each other and the Gaussian kernel behaves almost like the matching kernel $k_\delta(x, x') = 1 \Leftrightarrow x = x'$ and $k_\delta(x, x') = 0 \Leftrightarrow x \neq x'$. In applications this often causes a problem known as the *ridge problem*, which means that the learning algorithm functions more or less just as a lookup table.

2.2 Valid Kernels

Technically, a kernel k corresponds to the inner product in some feature space which is, in general, different from the representation space of the instances. The computational attractiveness of kernel methods comes from the fact that quite often a closed form of these 'feature space inner products' exists. Instead of performing the expensive transformation step explicitly, the kernel can be calculated directly, thus performing the feature transformation only implicitly. Whether, for a given function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, a feature transformation $\phi : \mathcal{X} \to \mathcal{H}$ into the Hilbert space $\mathcal{H}$ exists, such that $k(x, x') = \langle \phi(x), \phi(x') \rangle$ for all $x, x' \in \mathcal{X}$ can be checked by verifying that the function is positive definite [1]. This means that any set, whether a linear space or not, that admits a positive definite kernel can be embedded into a linear space. Throughout the paper, we take 'valid' to mean 'positive definite'. Here then is the definition of a positive definite kernel. ($\mathbb{Z}^+$ is the set of positive integers.)

DEFINITION: Let $\mathcal{X}$ be a set. A symmetric function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a *positive definite kernel* on $\mathcal{X}$ if, for all $n \in \mathbb{Z}^+$, $x_1, \ldots, x_n \in \mathcal{X}$, and $c_1, \ldots, c_n \in \mathbb{R}$, it follows that $\sum_{i,j \in \{1,\ldots,n\}} c_i\, c_j\, k(x_i, x_j) \geq 0$.

While it is not always easy to prove positive definiteness for a given kernel, positive definite kernels do have nice closure properties. In particular, they are closed under sum, direct sum, multiplication by a scalar, product, tensor product, zero extension, pointwise limits, and exponentiation [5].

2.3 Good Kernels

For a kernel method to perform well on some domain, validity of the kernel is not the only issue. To discuss the characteristics of good kernels, we need the notion of a concept class. Concepts c are functions $c : \mathcal{X} \to \Omega$, where $\mathcal{X}$ is often referred to as the instance space or problem domain (examples are elements of this domain) and Ω are Boolean labels. A concept class $\mathcal{C}$ is a set of concepts.

While there is always a valid kernel that performs poorly ($k_0(x, x') = 0$), there is also always a valid kernel ($k_c(x, x') = +1 \Leftrightarrow c(x) = c(x')$ and $k_c(x, x') = -1 \Leftrightarrow c(x) \neq c(x')$) that performs ideally. We distinguish the following three issues crucial to 'good' kernels: completeness, correctness, and appropriateness.

Completeness refers to the extent to which the knowledge incorporated in the kernel is sufficient for solving the problem at hand. A kernel is said to be complete if it takes into account all the information necessary to represent the concept that underlies the problem domain. Formally, we call a kernel complete if $k(x, \cdot) = k(x', \cdot)$ implies $x = x'$ [1]. With respect to some concept class, however, it is not important to distinguish between instances that are equally classified by all concepts in that particular concept class. We call a kernel complete with respect to a concept class $\mathcal{C}$ if $k(x, \cdot) = k(x', \cdot)$ implies $c(x) = c(x')$ for all $c \in \mathcal{C}$.

Correctness refers to the extent to which the underlying semantics of the problem are obeyed in the kernel. Correctness can formally only be expressed with respect to a certain concept class and a certain hypothesis language. In particular, we call a kernel correct with respect to a concept class $\mathcal{C}$ and support vector machines if for all concepts $c \in \mathcal{C}$ we can find $\alpha_i \in \mathbb{R}, x_i \in \mathcal{X}, \theta \in \mathbb{R}$ such that $\forall x \in \mathcal{X} : \sum_i \alpha_i k(x_i, x) \geq \theta \Leftrightarrow c(x)$. For the remainder of the paper we will use 'correct' only with respect to support vector machines and other kernel methods using a similar hypothesis language.

[1]This corresponds to the map ϕ, with $\langle \phi(x), \phi(x') \rangle = k(x, x')$ for all x, x', being injective.

Appropriateness refers to the extent to which examples that are close to each other in class membership are also 'close' to each other in feature space. Appropriateness can formally only be defined with respect to a concept class and a learning algorithm. A kernel is appropriate for learning concepts in a concept class if polynomial mistake bounds can be derived for some algorithm using this kernel. This problem is most apparent in the the matching kernel $k_\delta(x, x') = 1 \Leftrightarrow x = x'$ and $k_\delta(x, x') = 0 \Leftrightarrow x \neq x'$ which is always complete and correct but (in general) not appropriate.

Empirically, a complete, correct and appropriate kernel exhibits two properties. A complete and correct kernel separates the concept well, i.e., a learning algorithm achieves high accuracy when learning and validating on the same part of the data. An appropriate kernel generalizes well, i.e., a learning algorithm achieves high accuracy when learning and validating on different parts of the data.

2.4 Classes of Kernels

A useful conceptual distinction between different kernels is based on the 'driving-force' of their definition. We distinguish between syntax and models as the driving-force of the kernel definition.

Syntax is often used in typed systems to formally describe the semantics of the data. It is the most common driving force. In its simplest case, i.e., untyped attribute-value representations, it treats every attribute in the same way. More complex syntactic representations are graphs, restricted subsets of graphs such as lists and trees, or terms in some (possibly typed) logic. Whenever kernels are syntax-driven, they are either special case kernels, assuming some underlying semantics of the application, or they are parameterized and offer the possibility to adapt the kernel function to certain semantics.

Models contain some kind of knowledge about the instance space, i.e., about the relationships among instances. These models can either be generative models of instances or they can be given by some sort of transformation relations. Hidden Markov models are a frequently used generative model. Edit operations on a lists are one example for operations that transform one list into another. The graph defined on the set of lists by these operations can be seen as a model of the instance space. While each edge of a graph only contains local information about neighboring vertices, the set of all edges, i.e., the graph itself, also contains information about the global structure of the instance space.

3. MODEL-DRIVEN KERNELS

This section describes different kernel functions defined on models describing the instance space. These models are either constructed form background knowledge about the semantics of the domain at hand or learned from data. The first part of this section deals with kernel functions defined on probabilistic, generative models of the instance space. The second part of the this section describes kernel functions defined using some kind of similarity relation or transformation operation between instances.

3.1 Kernels from Generative Models

Generative models in general and hidden Markov models [35] in particular are widely used in computer science. One of their application areas is protein fold recognition where one tries to understand how proteins fold up in nature. Another application area is speech recognition. One motivation behind the development of kernels on generative models is to be able to apply kernel methods to sequence data. Sequences occur frequently in nature, for example, proteins are sequences of amino acids, genes are sequences of nucleic acids, and spoken words are sequences of phonemes. Another motivation is to improve the classification accuracy of generative models.

The first and most prominent kernel function based on a generative model is the Fisher kernel [17; 18]. The key idea is to use the gradient of the log-likelihood with respect to the parameters of a generative model as the features in a discriminative classifier. The motivation to use this feature space is that the gradient of the log-likelihood with respect to the parameters of a generative model captures the generative process of a sequence rather that just the posterior probabilities.

Let U_x be the gradient of the log-likelihood with respect to the parameters of the generative model $P(x|\theta)$ at x:

$$U_x = \nabla_\theta \log P(x|\theta)$$

Furthermore, let I be the Fisher information matrix, i.e., the expected value of the outer product $U_x U_x^\top$ over $P(x|\theta)$. The Fisher kernel is then defined as $k(x, x') = U_x^\top I^{-1} U_{x'}$. The Fisher kernel can be calculated whenever the probability model $P(x|\theta)$ of the instances given the parameters of the model has a twice differentiable likelihood and the Fisher information matrix is positive definite at the chosen θ. Learning algorithms using the Fisher kernel can be shown to perform well if the class variable is contained as a latent variable in the probability model. In [17] it has been shown that under this condition kernel machines using the Fisher kernel are asymptotically at least as good as choosing the maximum a posteriori class for each instance based on the model. In practice often the role of the Fisher information matrix is ignored, yielding the kernel $k(x, x') = U_x^\top U_{x'}$.

Usually, as a generative model a hidden Markov model is used and as a discriminative classifier a support vector machine is used. The Fisher kernel has successfully been applied in many learning problems where the instances are sequences of symbols, such as protein classification [16; 20] and promoter region detection [34].

The key ingredient of the Fisher kernel is the Fisher score mapping U_x that extracts a feature vector from a generative model. In [39] performance measures for comparing such feature extractors are discussed. Based on this discussion, a kernel is defined on models where the class is an explicit variable in the generative model, rather than only a latent variable as in the Fisher kernel. Empirically, this kernel performs favorably to the Fisher kernel on a protein fold recognition task. A similar approach has been applied to speech recognition [38].

Recently, a general framework for defining kernel functions on generative models has been described [40]. The so-called 'marginalized kernels' contain the above described Fisher kernel as a special case. The paper compares other marginalized kernels with the Fisher kernel and argues that these have some advantages over the Fisher kernel. While, for example, Fisher kernels only allow for the incorporation of second-order information by using a second-order hidden Markov model [7], other marginalized kernels allow for the use of second-order information with a first-order hidden

Markov model. In [40] it is shown that incorporating this second-order information in a kernel function is useful in the prediction of bacterial genera from their DNA.

In general, marginalized kernels are defined on any generative model with some visible and some hidden information. Let the visible information be an element of the finite set $\mathcal{X}$ and the hidden information be an element of the finite set $\mathcal{S}$. If the hidden information was known, a joint kernel $k_z : (\mathcal{X} \times \mathcal{S}) \times (\mathcal{X} \times \mathcal{S}) \to \mathbb{R}$ could be used. Usually, the hidden information is unknown but the expectation of a joint kernel with respect to the hidden information can be used. Let $x, x' \in \mathcal{X}$ and $s, s' \in \mathcal{S}$. Given a joint kernel k_z and the posterior distribution $p(s|x)$ (usually estimated from a generative model), the marginalized kernel in $\mathcal{X}$ is defined as:

$$k(x,x') \sum_{s,s' \in \mathcal{S}} p(s|x)p(s'|x')k_z((x,s),(x',s'))$$

To complete this section on kernels from generative models, the idea of defining kernel functions between sequences based on a pair hidden Markov model [7] has to be mentioned. Such kernels have been developed in [15] and [45]. Strictly speaking pair hidden Markov models are not models of the instance space, they are generative models of an aligned pair of sequences [7].

Recently, [36] has shown that syntactic string kernels (presented in section 4.2) can be seen as a special case of Fisher kernels of a k-stage Markov process with uniform distributions over the transitions.

3.2 Kernels from Transformations

This section describes kernels that are based on knowledge about common properties of instances or transformations between instances. The best known kernel in this class is the diffusion kernel.

The motivation behind diffusion kernels [24] is that it is often more easy to describe the local neighborhood of an instance than to describe the structure of the whole instance space or to compute a similarity between two arbitrary instances. The neighborhood of an instance might be all instances that differ with this one only by the presence or absence of one particular property. When working with molecules, for example, such properties might be functional groups or bonds. The neighborhood relation obviously induces global information about the make up of the instance space. The approach taken in the diffusion kernel is to try to capture this global information in a kernel function merely based on the neighborhood description.

The main mathematical tool used in diffusion kernels is matrix exponentiation. The exponential of a square matrix H is defined as

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

It is known that the limit always exists and that $e^{\beta H}$ is a positive definite matrix if H is symmetric. In this case it is also possible to compute $e^{\beta H}$ efficiently by first diagonalizing the matrix H such that $H = T^{-1}DT$ and then computing

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

where $e^{\beta D}$ can be computed component-wise (as D is diagonal).

The matrix H is called the 'generator'. The kernel matrix is defined as the exponential of the generator. In the case of instance spaces that have undirected graph structure, [24] suggests to use the negative Laplacian of the graph as the generator. Let $G = (\mathcal{V}, \mathcal{E})$ be an undirected graph with vertices $\mathcal{V} = \{\nu_i\}$ and edges $\mathcal{E} \subset 2^{\mathcal{V}}$. In particular $\{\nu_i, \nu_j\} \in \mathcal{E}$ if there is an edge between vertices ν_i and ν_j. Furthermore, let $\delta(\nu_i) = \{\nu_j \in \mathcal{V} : \{\nu_i, \nu_j\} \in \mathcal{E}\}$. Then the negative Laplacian of the graph is given by

$$[H]_{i,j} = \begin{cases} 1 & \{\nu_i, \nu_j\} \in \mathcal{E} \\ -|\delta(\nu_i)| & \nu_i = \nu_j \\ 0 & \text{otherwise} \end{cases}$$

where H_{ij} denotes the i,j-th component of the generator matrix. To generalize this to the case of graphs with weighted and/or parallel edges, the components of H_{ij} for $\nu_i \neq \nu_j$ are replaced by the sum over the weights of all edges between ν_i and ν_j.

If the instance space is big, the computation of $e^{\beta H}$ suggested above might still be too expensive. For some special instance space structures, such as regular trees, complete graphs, and closed chains [24] gives closed forms for directly computing the exponential of the generator and thus the kernel matrix. An application of the diffusion kernel to gene function prediction has been described in [43].

A similar idea of defining a kernel function on the structure of the instance space is described in [42]. In that paper it is described how global patterns of inheritance can be incorporated in a kernel function and how missing information can be dealt with using a probabilistic model of inheritance. The main difference to the diffusion kernel is that the structures considered are directed trees and that instances correspond to sets of vertices in these trees rather than single instances. Trees are connected acyclic graphs where one vertex has no incoming edge and all other nodes have exactly one incoming edge. The application considered in that paper is that of classifying phylogenetic profiles. A phylogenetic profile contains information about the organisms in which a particular gene occurs. The phylogenetic information considered in [42] is represented as a tree such that each leaf (a vertex with no outgoing edge) corresponds to one living organism and every other vertex corresponds to some ancestor of the living organisms. To represent genes, every vertex of the tree is assigned a random variable. The value of this random variable indicates whether the corresponding organism has a homologue of the particular gene or not.

If the genomes of all ancestor organisms were available, this information could be used to define a kernel function reflecting the similarity between evolutions. A subtree of the above described tree along with an assignment of indicator values to this tree is called an evolution pattern. Ideally the kernel function on two genes would be defined as the number of evolution patterns that agree with the phylogenetic histories of both genes. An evolution pattern agrees with a phylogenetic history if the assignment of indicator variables is the same for all vertices in the evolution pattern. As the genomes are only known for some ancestor organisms, a probabilistic model that can be used to estimate missing indicator variables is suggested in [42].

For two given phylogenetic profiles x_L, y_L the kernel is defined as follows. Let T be a tree, L be the leaf nodes and $C(T)$ the set of all possible subtrees of T. One particular evolution pattern can be expressed as a subtree $S \in C(T)$ and the corresponding assignment z_S of indicator values. $p(z_s)$ denotes the probability of such an evolution pattern having occurred in nature and $p(x_L|z_S)$ is the probability of observing a particular phylogenetic profile x_L given the evolution pattern z_S (obtained from the probabilistic model mentioned above). Then the tree kernel for phylogenetic profiles is defined as:

$$k(x_L, y_L) = \sum_{S \in C(T)} \sum_{z_S} p(z_S)p(x_L|z_S)p(y_L|z_S)$$

In [42] it is shown that this kernel function can be computed in time linear in the size of the tree.

4. SYNTAX-DRIVEN KERNELS

This section describes different kernel definitions based on the syntax of the representation. The simplest way to apply kernel methods to multi-relational data it to first transform the data into a single table. This process is called propositionalization [25]. The first application of support vector machines to propositionalized data is described in [26]. We will not consider such approaches in more detail in this paper, we will rather concentrate on kernels defined directly on structured data.

The most prominent kernel for representation spaces that are not mere attribute-value tuples, the convolution kernel, is introduced first. Its key idea is to define a kernel on a composite object by means of kernels on the parts of the objects. This idea is also reflected in many of the kernel functions developed later and described thereafter in this section. Several kernel functions have been defined on sequences of discrete symbols. The idea is always to extract all possible subsequences (of some kind) and to define the kernel function based on the occurrence and similarity of these subsequences. This idea can be generalized to extracting subtrees of trees and defining a kernel function based occurrence and similarity of subtrees in two trees. A rather different approach is that of representing the instances of the learning task as terms in a typed higher-order logic. These terms are powerful enough to allow for a close modeling of the semantics of different objects by means of the syntax of the representation. The last kernel described in this section is a kernel defined on instances represented by graphs.

4.1 Convolutions

The best known kernel for representation spaces that are not mere attribute-value tuples is the convolution kernel proposed by Haussler [15]. The basic idea of convolution kernels is that the semantics of composite objects can often be captured by a relation R between the object and its parts. The kernel on the object is then made up from kernels defined on different parts.

Let $x, x' \in \mathcal{X}$ be the objects and $\vec{x}, \vec{x}' \in \mathcal{X}_1 \times \cdots \times \mathcal{X}_D$ be tuples of parts of these objects. Given the relation $R : (\mathcal{X}_1 \times \cdots \times \mathcal{X}_D) \times \mathcal{X}$ we can define the decomposition R^{-1} as $R^{-1}(x) = \{\vec{x} : R(\vec{x}, x)\}$. Then the convolution kernel is defined as

$$k_{conv}(x, x') = \sum_{\vec{x} \in R^{-1}(x), \vec{x}' \in R^{-1}(x')} \prod_{d=1}^{D} k_d(x_d, x'_d)$$

The term 'convolution kernel' refers to a class of kernels that can be formulated in the above way. The advantage of convolution kernels is that they are very general and can be applied in many different problems. However, because of that generality, they require a significant amount of work to adapt them to a specific problem, which makes choosing R in 'real-world' applications a non-trivial task.

4.2 Strings

While the traditional kernel function used for text classification is simply the scalar product of two texts in their 'bag-of-words' representation [19], this kernel function does not take the the structure of the text or words into account but simply the number of times each word occurs. More sophisticated approaches try to define a kernel function on the sequence of characters. Similar approaches define kernel functions on other sequences of symbols, e.g., on the sequence of symbols each corresponding to one amino acid and together describing a protein.

The first kernel function defined on strings can be found in [46; 31] and is also described in [5; 37]. The idea behind this kernel is to base the similarity of two strings on the number of common subsequences. These subsequences need not be contiguous in the strings but the more gaps in the occurrence of the subsequence, the less weight is given to it in the kernel function. This can be best illustrated by an example.

Consider the two strings 'cat' and 'cart'. The common subsequences are 'c', 'a', 't', 'ca', 'at', 'ct', 'cat'. As mentioned above, it is useful to penalize gaps in the occurrence of the subsequence. This can be done using the total length of a subsequence in the two strings. We now list the common subsequences again and give the total length of their occurrence in 'cat' and 'cart' as well: 'c':1/1, 'a':1/1, 't':1/1, 'ca':2/2, 'at':2/3, 'ct':3/4, 'cat':3/4. Usually, an exponential decay is used. With a decay factor λ the penalty associated with each substring is 'c':($\lambda^1\lambda^1$), 'a':($\lambda^1\lambda^1$), 't':($\lambda^1\lambda^1$), 'ca':($\lambda^2\lambda^2$), 'at':($\lambda^2\lambda^3$), 'ct':($\lambda^3\lambda^4$), 'cat':($\lambda^3\lambda^4$). The kernel function is then simply the sum over these penalties, i.e., $k(\text{'cat'}, \text{'cart'}) = 2\lambda^7 + \lambda^5 + \lambda^4 + 3\lambda^2$.

Let Σ be a finite alphabet, Σ^n the set of strings of length n from that alphabet and Σ^* the set of all strings from that alphabet. Let $|s|$ denote the length of the string $s \in \Sigma^*$ and let the symbols in s be indexed such that $s = s_1 s_2 \ldots s_{|s|}$. For a set of indices i let $s[i]$ be the subsequence of s induced by this set of indices and let $l(i)$ denote the total length of $s[i]$ in s, i.e., the biggest index in i minus the smallest index in i plus one. We are now able to define the feature transformation ϕ underlying the string kernel. For some string $u \in \Sigma^n$ the value of the feature $\phi_u(s)$ is defined as:

$$\phi_u(s) = \sum_{i : u = s[i]} \lambda^{l(i)}$$

The kernel between two strings $s, t \in \Sigma^*$ is then simply the scalar product of $\phi(s)$ and $\phi(t)$ and can be written as:

$$k_n(s, t) = \sum_{u \in \Sigma^n} \phi_u(s)\phi_u(t) = \sum_{u \in \Sigma^n} \sum_{i : u = s[i]} \sum_{j : u = t[j]} \lambda^{l(i)+l(j)}$$

While this computation appears very expensive, recursive computation can be reduced to $\mathcal{O}(n|s||t|)$ [31].

An alternative to the above kernel has been used in [33] and [27] where only contiguous substrings of a given string are considered. A string is then represented by the number of times each unique substring of (up to) n symbols occurs in the sequence. This representation of strings by their contiguous substrings is known as the spectrum of a string or as its n-gram representation. The kernel function is simply the scalar product in this representation. It is shown in [27] that this kernel can be computed in time linear in the length of the strings and the length of the considered substrings. In [33] not all possible n-grams are used but a simple statistical test is employed as a feature subset selection heuristic. This kernel has been applied to protein [27] and spoken text [33] classification. Spoken text is represented by a sequence of phonemes, syllables, or words.

Many empirical results comparing the above kernel functions can be found in [30]. As shown in [36] these kernels can be seen as a special case of the Fisher kernel (presented in section 3.1). This perspective leads to a generalization of the above kernel that is able to deal with variable length substrings. In [44] and [28] string kernels similar to the n-gram kernel are considered and it is shown how these can be computed efficiently by using suffix and mismatch trees, respectively. The main conceptual difference to the n-gram kernel is that a given number of mismatches is allowed when comparing the n-grams to the substrings.

Another kernel on strings can be found in [47]. The focus of that paper is on the recognition of translation inition sites in DNA or mRNA sequences. This problem is quite different from the above considered applications. The main difference is that rather than classifying whole sequences, in this task one codon (three consecutive symbols) from a sequence has to be identified as the translation inition site. Each sequence can have arbitrarily many candidate solutions of which one has to be chosen. However, in [47] and earlier work this problem is converted into a traditional classification problem on sequences of equal length.

Fixed length windows of symbols centered at each candidate solution are extracted from each sequence. The class of one such window corresponds to the candidate solution being a true translation inition site or not. One valid kernel on these sequences is simply the number of symbols that coincide in the two sequences. Other kernels can, for example, be defined as a polynomial of this kernel.

Better classification accuracy is, however, reported in [47] for a kernel that puts more emphasis on local correlations. Let n be the length of each window and Σ be the set of possible symbols, then each window is an element of Σ^n. For $x, x' \in \Sigma^n$ let x_i, x'_i denote the i-th element of each sequence. Using the matching kernel k_δ on Σ ($k_\delta : \Sigma \times \Sigma \to \mathbb{R}$ is defined as $k_\delta(x_i, x'_i) = 1$ if $x_i = x'_i$ and $k_\delta(x_i, x'_i) = 0$ if $x_i \neq x'_i$) first a polynomial kernel on a small sub-window of length $2l + 1$ with weights $w_j \in \mathbb{R}$ and power d_1 is defined:

$$k_i(x, x') = \left(\sum_{j=-l}^{l} w_j k_\delta(x_{i+j}, x'_{i+j}) \right)^{d_1}$$

Then the kernel on the full window is simply the polynomial of power d_2 of the kernels on the sub-windows:

$$k(x, x') = \left(\sum_{i=l}^{n-l} k_i(x, x') \right)^{d_2}$$

This kernel is called the 'locality-improved' kernel. An empirical comparison to other general purpose machine learning algorithms shows competitive results for $l = 0$ and better results for larger l. These results were obtained on the above mentioned translation inition site recognition task.

Even better results can be achieved by replacing the symbol at some position in the above definition with the conditional probability of that symbol given the previous symbol. Let $p_{i,\mathrm{TIS}}(x_i|x_{i-1})$ be the probability of symbol x_i at position i given symbol x_{i-1} at position $i - 1$, estimated over all true translation inition sites. Furthermore, let $p_{i,\mathrm{ALL}}(x_i|x_{i-1})$ be the probability of symbol x_i at position i given symbol x_{i-1} at position $i - 1$, estimated over all candidate sites. Then we define

$$s_i(x) = \log p_{i,\mathrm{TIS}}(x_i|x_{i-1}) - \log p_{i,\mathrm{ALL}}(x_i|x_{i-1})$$

and replace x_{i+j} in the locality-improved kernel by $s_{i+j}(x)$ and the matching kernel by the product. A support vector machine using this kernel function outperforms all other approaches applied in [47].

4.3 Trees

A kernel function that can be applied in many natural language processing tasks is described in [4]. The instances of the learning task are considered to be labeled ordered directed trees. The key idea to capture structural information about the trees in the kernel function is to consider all subtrees occurring in a parse tree. Here, a subtree is defined as a connected subgraph of a tree such that either all children or no child of a vertex is in the subgraph. The children of a vertex are the vertices that can be reached from the vertex by traversing one directed edge. The kernel function is the inner product in the space which describes the number of occurrences of all possible subtrees.

Consider some enumeration of all possible subtrees and let $h_i(T)$ be the number of times the i-th subtree occurs in tree T. For two trees T_1, T_2 the kernel is then defined as

$$k(T_1, T_2) = \sum_i h_i(T_1) h_i(T_2)$$

Furthermore, for the sets of vertices $\mathcal{V}_1$ and $\mathcal{V}_2$ of the trees T_1 and T_2, let $S(v_1, v_2)$ with $v_1 \in \mathcal{V}_1, v_2 \in \mathcal{V}_2$ be the number of subtrees rooted at vertex v_1 and v_2 that are isomorphic. Then the tree kernel can be computed as

$$k(T_1, T_2) = \sum_i h_i(T_1) h_i(T_2) = \sum_{v_1 \in \mathcal{V}_1, v_2 \in \mathcal{V}_2} S(v_1, v_2)$$

Let $label(v)$ be a function that returns the label of vertex v, let $|\delta^+(v)|$ denote the number of children of vertex v, and let $\delta^+(v, j)$ be the j-th child of vertex v (only ordered trees are considered). $S(v_1, v_2)$ can efficiently be calculated as follows: $S(v_1, v_2) = 0$ if $label(v_1) \neq label(v_2)$. $S(v_1, v_2) = 1$ if $label(v_1) = label(v_2)$ and $|\delta^+(v_1)| = |\delta^+(v_2)| = 0$. Otherwise[2],

$$S(v_1, v_2) = \prod_{k=1}^{|\delta^+(v_1)|} \left(1 + S(\delta^+(v_1, j), \delta^+(v_2, j))\right)$$

This recursive computation has time complexity $\mathcal{O}(|\mathcal{V}_1||\mathcal{V}_2|)$. Experiments investigated how much the application of a kernelized perceptron algorithm to trees generated by a probabilistic context free grammar outperforms the use of the

[2]Note that in this case $|\delta^+(v_1)| = |\delta^+(v_2)|$, as actually the number of children of a vertex is determined by its label. This is due to the nature of the natural language processing applications that are considered.

probabilistic context free grammar alone. The empirical results achieved in [4] are promising. The kernel function used in these experiments is actually a weighted variant of the kernel function presented above.

A generalization of this kernel to also take into account other substructures of the trees is described in [22]. A substructure of a tree is defined as a tree such that there is a descendants preserving mapping from vertices in the substructure to vertices in the tree[3]. Another generalization considered in that paper is that of allowing labels to partially match. Promising results have been achieved with this kernel function in HTML document classification tasks.

Recently, [44] proposed the application of string kernels to trees by representing each tree by the sequence of labels generated by a depth-first traversal of the trees, written in preorder notation. To ensure that trees only differing in the order of their children are represented in the same way, the children of each vertex are ordered according to the lexical order of their string representation.

4.4 Basic Terms

In [14] a framework has been been proposed that allows for the application of kernel methods to different kinds of structured data. This approach is based on the idea of having a powerful representation that allows for modeling the semantics of an object by means of the syntax of the representation. The underlying principle is that of representing individuals as (closed) terms in a typed higher-order logic [29]. The typed syntax is important for pruning search spaces and for modeling as closely as possible the semantics of the data in a human- and machine-readable form. The individuals-as-terms representation is a natural generalization of the attribute-value representation and collects all information about an individual in a single term.

The key idea is to have a fixed type structure. This type structure expresses the semantics and composition of individuals from their parts. The type structure is made up by function types, product types, and type constructors. Function types are used to represent types corresponding to sets, multisets, and so on. Product types are used to represent types corresponding to fixed size tuples. Type constructors are used to represent types corresponding to arbitrary size structured objects such as lists, trees, and so on. The set of type constructors also contains types corresponding to symbols and numbers.

To define, for example, a type corresponding to a subset of $\{A, B, C, D\}$ one first defines a type constructor of arity zero, corresponding to elements of this set. Then the type corresponding to a sets of these elements is a function type where the function is from the set of elements to the set of Boolean values. The type corresponding to a multi-set of these elements is a function type where the function is from the set of elements to the set of natural numbers. It is important to note that the elements of these sets, lists, etc. are not restricted to be mere symbols but can again be structured objects. Thus one can, for example, define a type corresponding to a set of lists, or a list of sets.

Each type defines a set of terms that represent instances of that type, these are called the basic terms. The terms of the logic are the terms of the typed λ-calculus, which are formed in the usual way by abstraction, tupling, and application. Abstraction corresponds to building instances of a function type, tupling to building instances of a product type, and application to building instances of a type constructor.

To this end the biggest difference to terms of a first-order logic is the presence of abstractions that allows modeling sets, multisets, and so on. For example, the basic terms s, t representing the set $\{1,2\}$ and the multiset with 42 occurrences of A and 21 occurrences of B (and nothing else) are, respectively:

$$s = \lambda x.\mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \bot$$
$$t = \lambda x.\mathit{if}\ x = A\ \mathit{then}\ 42\ \mathit{else}\ \mathit{if}\ x = B\ \mathit{then}\ 21\ \mathit{else}\ 0$$

Now, we need some additional notation. For a basic abstraction r, $V(r\ u)$ denotes the value of r when applied to u, i.e., $V(s\ 2) = \top$ and $V(t\ C) = 0$. The default term is the value of the abstraction that has no condition, i.e., the default term of s is $\bot$ and the default term of t is 0. The support of an abstraction is the set of terms u for which $V(r\ u)$ differs from the default term, i.e., $\mathit{supp}(s) = \{1, 2\}$ and $\mathit{supp}(t) = \{A, B\}$. More details of the knowledge representation can be found in [29]. The basic term kernel [14] is then defined inductively on the structure of terms.

If s, t are basic terms formed by application, i.e., instances of a type constructor, then s is $C\ s_1 \dots s_n$ and t is $D\ t_1 \dots t_m$, where C, D are data constructors and s_i, t_j are basic terms. The kernel is then defined as

$$k(s,t) = \begin{cases} \kappa_T(C, D) & \text{if } C \neq D \\ \kappa_T(C, C) + \sum_{i=1}^{n} k(s_i, t_i) & \text{otherwise} \end{cases}$$

If s, t are basic terms formed by abstraction, i.e., instances of a function type, then the kernel is defined as

$$k(s,t) = \sum_{\substack{u \in \mathit{supp}(s) \\ v \in \mathit{supp}(t)}} k(V(s\ u), V(t\ v)) \cdot k(u, v).$$

If s, t are basic terms formed by tupling, i.e., instances of a product type, then s is $(s_1, \dots, s_n)$ and t is $(t_1, \dots, t_n)$, where s_i, t_j are basic terms. The kernel is then defined as

$$k(s,t) = \sum_{i=1}^{n} k(s_i, t_i),$$

In [14] additional 'modifiers' are described that allow for the modification of the default kernels in order to reflect the semantics of the domain better. We will now describe some applications along with the type definitions.

In drug activity prediction it is common to represent the shape of a molecule by measuring the distance from the center of the molecule to the surface in several directions. Thus a shape can be described by a tuple of real numbers. As, however, the shape of the molecule changes as its energy state changes, each molecule can only be described by a set of shapes (conformations). A molecule is active if one of its conformations satisfies some conditions, this is known as a 'multi-instance' concept. The task of classifying a molecule as active or inactive given the set of its conformations has been introduced in [6]. In [12] each molecule is represented by a set of conformations (i.e., an abstraction mapping a conformation to 'true' if it is a member of that set and to 'false' otherwise), and each conformation is represented by a

[3]A descendant of a vertex v is any vertex that occurs in the subtree rooted at v.

tuple of real numbers. It can be shown that the corresponding basic term kernel is correct with respect to support vector machines and multi-instance concepts. In an empirical evaluation, support vector machines proofed competitive to the best results achieved in literature.

For spatial clustering it is important to gather data points in a cluster that are not only spatially close but also demographically similar. Applying a simple clustering algorithm to vectors containing both the spatial and the demographic information fails to find spatially compact clusters. Modeling the spatial and demographic information as two different types and defining the type of the individuals as a function type mapping the spatial information to the corresponding demographic information leads to a kernel function that can be used by a simple clustering algorithm to find spatially compact clusters [14].

To elucidate the structure of molecules, spectroscopic methods such as ^{13}C NMR are frequently used. A spectrum contains information about the resonance frequency of each (chemically different) carbon atom in the molecule. Additional information can be obtained describing the number of protons directly connected with each carbon atom (the multiplicity). The task of predicting the skeleton structure of molecules from their NMR spectrum has been introduced in [9]. A spectrum is modeled as an abstraction mapping each resonance frequency to the multiplicity of the corresponding carbon atom and mapping every other frequency to zero. A support vector machine using the corresponding kernel function and and a nearest neighbor algorithm using the corresponding metric have been shown to outperform all other algorithms applied in literature to this problem.

4.5 Graphs

Labeled graphs are widely used in computer science to model data. Some of the work described above can be seen as kernels on some restricted set of graphs, e.g., on strings or on trees. In this section we consider recent work [13] on labeled graphs with arbitrary structure. Such kernel functions can, for example, be useful in learning tasks on molecules.

A graph G is described by a finite set of vertices $\mathcal{V}$ and a finite set of edges $\mathcal{E}$. For directed graphs, the set of edges is a subset of the Cartesian product of the set of vertices with itself ($\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$) such that that $(\nu_i, \nu_j) \in \mathcal{E}$ if and only if there is an edge from ν_i to ν_j in graph G. For labeled graphs there is additionally a set of labels along with a function assigning a label to each edge and/or vertex.

The definition of a complete graph kernel is slightly different from the definition of complete kernels given above. In general it is not desired that learning algorithms distinguish between isomorphic graphs, i.e., graphs that only differ in the enumeration of their vertices. Complete graph kernels are those positive definite functions on graphs for which $k(G, \cdot) = k(G', \cdot)$ if and only if G, G' are isomorphic.

Using a complete graph kernel and computing $k(G,G) - 2k(G,G') + k(G,G)$ one could decide whether G, G' are isomorphic [13]. As deciding graph isomorphism is suspected to be computationally hard, one can conclude that no efficiently computable complete graph kernel exists. It can also be shown that some complete graph kernels are *NP*-hard to compute.

Consider a graph kernel that has one feature Φ_H for each possible graph H, each feature $\Phi_H(G)$ measuring how many subgraphs of G have the same structure as graph H. Using the inner product in this feature space, graphs satisfying certain properties can be identified. In particular, one could decide whether a graph has a Hamiltonian path [13], i.e., a sequence of adjacent vertices that contains every vertex exactly once. Now this problem is known to be *NP*-hard, i.e., it is strongly believed that this problem can not be solved in polynomial time.

The two results given above motivate the search for alternative graph kernels which are less expressive and therefore less expensive to compute. Conceptually, the graph kernels presented in [10; 13; 21; 23] are based on a measure of the walks in two graphs that have some or all labels in common. In [10] walks with equal initial and terminal label are counted, in [21; 23] the probability of random walks with equal label sequences is computed, and in [13] walks with equal label sequences, possibly containing gaps, are counted.

We consider here the work presented in [13]. In this work, efficient computation of these – possibly infinite – walks is made possible by using the direct product graph and computing the limit of matrix power series involving its adjacency matrix.

The two graphs generating the product graph are called the factor graphs. The vertex set of the direct product of two graphs is a subset of the Cartesian product of the vertex sets of the factor graphs. The direct product graph has a vertex if and only if the labels of the corresponding vertices in the factor graphs are the same. There is an edge between two vertices in the product graph if there is an edge between the corresponding vertices in both factor graphs and both edges have the same label. To build its adjacency matrix, assume some arbitrary enumeration $\{\nu_i\}_i$ of the vertices. Each component of the adjacency matrix $E_\times$ is defined by $[E_\times]_{ij} = 1 \Leftrightarrow (\nu_i, \nu_j) \in \mathcal{E}_\times$ and $[E_\times]_{ij} = 0 \Leftrightarrow (\nu_i, \nu_j) \notin \mathcal{E}_\times$, where $\mathcal{E}_\times$ denotes the edge set of the direct product graph. With a sequence of weights $\lambda_0, \lambda_1, \ldots$ ($\lambda_i \in \mathbb{R}; \lambda_i \geq 0$ for all $i \in \mathbb{N}$) the direct product kernel is then defined as

$$k_\times(G_1, G_2) = \sum_{i,j=1}^{|\mathcal{V}_\times|} \left[\sum_{n=0}^{\infty} \lambda_n E_\times^n \right]_{ij}$$

if the limit exists. Using exponential series $(\lambda_i = \beta^i / i!)$ or geometric series $(\lambda_i = \gamma^{-i})$, this kernel can be computed in cubic time. Extensions suggested in [13] include a kernel for counting label sequences with gaps, and one for handling transition graphs, i.e., graphs with a probability distribution over all edges leaving the same vertex.

One interesting application of such graph kernels is an experiment in a relational reinforcement learning setting, described in [11]. In that paper Gaussian processes were applied with graph kernels as the covariance function. Experiments were performed in blocks worlds of up to ten blocks with three different goals. In this setting Gaussian processes with graph kernels proofed competitive or superior to all previous implementations of relational reinforcement learning algorithms, although it did not use any sophisticated instance selection strategy.

5. CONCLUSIONS

It has often been argued that relational data mining and inductive logic programming approaches should have successful propositional learning algorithms as a special case. Support vector machines are one of the most successful re-

cent developments within the machine learning area. Thus developing algorithms that can be a applied to structured data and have support vector machines as a special case is a promising research direction. This can be achieved by defining positive definite kernels on structured data.
Such positive definite kernels allow algorithms to be applied to structured data that solve learning problems so far not considered by the inductive logic programming community. One example is kernel principal component analysis which finds optimal embeddings of structured data into low dimensional spaces. Such an embedding can, for example, be used to visualize the relationship among instances, or to find a good attribute-value representation of the data.
Support vector machines have many computational and learning theoretical properties that make them a very interesting and popular learning algorithm. Many of these properties follow from the kernel function being positive definite. The problem of finding a hyperplane which is maximally distant from points on either side of the hyperplane can be formulated as a quadratic program. Such problems are convex if and only if the matrix they are operating on is positive definite. A convex problem has only one local - and thus global - optimum. This optimum can be found efficiently by well known optimization algorithms.
In this paper we described several approaches to define positive definite kernel functions on various kinds of structured data. On the one hand, kernel functions that are applicable to similar kinds of data have been compared and their respective application areas briefly described. On the other hand, conceptual differences between approaches have been clarified and summarized.
We believe that investigating kernels on structured data is an important and promising research area. While several kernels have been defined so far, this research area is still very young and there is space for more work.

6. ACKNOWLEDGMENTS

Research supported in part by the EU Framework V project (IST-1999-11495) *Data Mining and Decision Support for Business Competitiveness: Solomon Virtual Enterprise* and the DFG project (WR 40/2-1) *Hybride Methoden und Systemarchitekturen für heterogene Informationsräume*. The author thanks Peter Flach, Tamás Horváth, John Lloyd, and Stefan Wrobel for valuable discussions and comments.

7. REFERENCES

[1] N. Aronszajn. Theory of reproducing kernels. *Transactions of the American Mathematical Society*, 68, 1950.

[2] K. Bennett and C. Campbell. Support vector machines: Hype or hallelujah? *SIGKDD Explorations*, 2(2), 2000. http://www.acm.org/sigs/sigkdd/explorations/issue2-2/bennett.pdf.

[3] B. E. Boser, I. M. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In D. Haussler, editor, *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, pages 144–152. ACM Press, July 1992.

[4] 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*, volume 14. MIT Press, 2002.

[5] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines (and Other Kernel-Based Learning Methods)*. Cambridge University Press, 2000.

[6] T. G. Dietterich, R. H. Lathrop, and T. Lozano-Pérez. Solving the multiple instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89(1–2):31–71, 1997.

[7] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

[8] S. Džeroski and N. Lavrač, editors. *Relational Data Mining*. Springer-Verlag, 2001.

[9] S. Džeroski, S. Schulze-Kremer, K. Heidtke, K. Siems, D. Wettschereck, and H. Blockeel. Diterpene structure elucidation from ^{13}C NMR spectra with inductive logic programming. *Applied Artificial Intelligence*, 12(5):363–383, July-Aug. 1998. Special Issue on First-Order Knowledge Discovery in Databases.

[10] T. Gärtner. Exponential and geometric kernels for graphs. In *NIPS Workshop on Unreal Data: Principles of Modeling Nonvectorial Data*, 2002.

[11] T. Gärtner, K. Driessens, and J. Ramon. Graph kernels and gaussian processes for relational reinforcement learning. In *Proceedings of the 13th International Conference on Inductive Logic Programming*, 2003.

[12] T. Gärtner, P. A. Flach, A. Kowalczyk, and A. J. Smola. Multi-instance kernels. In C. Sammut and A. Hoffmann, editors, *Proceedings of the 19th International Conference on Machine Learning*, pages 179–186. Morgan Kaufmann, June 2002.

[13] T. Gärtner, P. A. Flach, and S. Wrobel. On graph kernels: Hardness results and efficient alternatives. In *Proceedings of the 16th Annual Conference on Computational Learning Theory and the 7th Kernel Workshop*, 2003.

[14] T. Gärtner, J. W. Lloyd, and P. A. Flach. Kernels for structured data. In *Proceedings of the 12th International Conference on Inductive Logic Programming*. Springer-Verlag, 2002.

[15] D. Haussler. Convolution kernels on discrete structures. Technical report, Department of Computer Science, University of California at Santa Cruz, 1999.

[16] T. Jaakkola, M. Diekhans, and D. Haussler. A discriminative framework for detecting remote protein homologies. *Journal of Computational Biology*, 7(1,2), 2000.

[17] T. Jaakkola and D. Haussler. Exploiting generative models in discriminative classifiers. In *Advances in Neural Information Processing Systems*, volume 10, 1999.

[18] T. Jaakkola and D. Haussler. Probabilistic kernel regression models. In *Proceedings of the 1999 Conference on AI and Statistics*, 1999.

[19] T. Joachims. *Learning to Classify Text using Support Vector Machines.* Kluwer Academic Publishers, 2002.

[20] R. Karchin, K. Karplus, and D. Haussler. Classifying g-protein coupled receptors with support vector machines. *Bioinformatics*, 18(1):147–159, 2002.

[21] H. Kashima and A. Inokuchi. Kernels for graph classification. In *ICDM Workshop on Active Mining*, 2002.

[22] H. Kashima and T. Koyanagi. Kernels for semi-structured data. In C. Sammut and A. Hoffmann, editors, *Proceedings of the 19th International Conference on Machine Learning.* Morgan Kaufmann, 2002.

[23] H. Kashima, K. Tsuda, and A. Inokuchi. Marginalized kernels between labeled graphs. In *Proceedings of the 20th International Conference on Machine Learning*, 2003.

[24] R. I. Kondor and J. Lafferty. Diffusion kernels on graphs and other discrete input spaces. In C. Sammut and A. Hoffmann, editors, *Proceedings of the 19th International Conference on Machine Learning*, pages 315–322. Morgan Kaufmann, 2002.

[25] S. Kramer, N. Lavrač, and P. A. Flach. Propositionalization approaches to relational data mining. In Džeroski and Lavrač [8], chapter 11.

[26] M.-A. Krogel and S. Wrobel. Transformation-based learning using multirelational aggregation. In C. Rouveirol and M. Sebag, editors, *Proceedings of the 11th International Conference on Inductive Logic Programming.* Springer-Verlag, 2001.

[27] C. Leslie, E. Eskin, and W. Noble. The spectrum kernel: A string kernel for svm protein classification. In *Proceedings of the Pacific Symposium on Biocomputing*, pages 564–575, 2002.

[28] C. Leslie, E. Eskin, J. Weston, and W. Noble. Mismatch string kernels for svm protein classification. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems*, volume 15. MIT Press, 2003.

[29] J. W. Lloyd. *Logic for Learning.* Springer-Verlag, 2002.

[30] H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins. Text classification using string kernels. *Journal of Machine Learning Research*, 2, 2002.

[31] H. Lodhi, J. Shawe-Taylor, N. Christianini, and C. Watkins. Text classification using string kernels. In T. Leen, T. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems*, volume 13. MIT Press, 2001.

[32] K.-R. Müller, S. Mika, G. Rätsch, K. Tsuda, and B. Schölkopf. An introduction to kernel-based learning algorithms. *IEEE Transactions on Neural Networks*, 2(2), 2001.

[33] G. Paass, E. Leopold, M. Larson, J. Kindermann, and S. Eickeler. Svm classification using sequences of phonemes and syllables. In T. Elomaa, H. Mannila, and H. Toivonen, editors, *Proceedings of the 6th European Conference on Principles of Data Mining and Knowledge Discovery*, pages 373–384. Springer-Verlag, 2002.

[34] P. Pavlidis, T. Furey, M. Liberto, D. Haussler, and W. Grundy. Promoter region-based classification of genes. In *Proceedings of the Pacific Symposium on Biocomputing*, pages 151–163, 2001.

[35] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–285, Feb. 1989.

[36] C. Saunders, J. Shawe-Taylor, and A. Vinokourov. String kernels, fisher kernels and finite state automata. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems*, volume 15. MIT Press, 2003.

[37] B. Schölkopf and A. J. Smola. *Learning with Kernels.* MIT Press, 2002.

[38] N. Smith and M. Gales. Speech recognition using SVMs. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14. MIT Press, 2002.

[39] K. Tsuda, M. Kawanabe, G. Rätsch, S. Sonnenburg, and K.-R. Müller. A new discriminative kernel from probabilistic models. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14. MIT Press, 2002.

[40] K. Tsuda, T. Kin, and K. Asai. Marginalized kernels for biological sequences. *Bioinformatics*, 2002.

[41] V. Vapnik. *The Nature of Statistical Learning Theory.* Springer-Verlag, 1995.

[42] J.-P. Vert. A tree kernel to analyze phylogenetic profiles. *Bioinformatics*, 2002.

[43] J.-P. Vert and M. Kanehisa. Graph driven features extraction from microarray data using diffusion kernels and kernel cca. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems*, volume 15. MIT Press, 2003.

[44] S. Vishwanathan and A. Smola. Fast kernels for string and tree matching. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems*, volume 15. MIT Press, 2003.

[45] C. Watkins. Dynamic alignment kernels. Technical report, Department of Computer Science, Royal Holloway, University of London, 1999.

[46] C. Watkins. Kernels from matching operations. Technical report, Department of Computer Science, Royal Holloway, University of London, 1999.

[47] A. Zien, G. Ratsch, S. Mika, B. Schölkopf, T. Lengauer, and K.-R. Muller. Engineering support vector machine kernels that recognize translation initiation sites. *Bioinformatics*, 16(9):799–807, 2000.