Kernels for Structured Data

Thomas Gärtner[1,3], John W. Lloyd[2], and Peter A. Flach[3]

[1] Knowledge Discovery, Fraunhofer Institut Autonome Intelligente Systeme, Germany
Thomas.Gaertner@ais.fraunhofer.de
[2] Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University
jwl@csl.anu.edu.au
[3] Machine Learning, Department of Computer Science, University of Bristol, UK
Peter.Flach@bristol.ac.uk

Abstract. Learning from structured data is becoming increasingly important. However, most prior work on kernel methods has focused on learning from attribute-value data. Only recently have researchers started investigating kernels for structured data. This paper describes how kernel definitions can be simplified by identifying the structure of the data and how kernels can be defined on this structure. We propose a kernel for structured data, prove that it is positive definite, and show how it can be adapted in practical applications.

1 Introduction

Support vector machines and other kernel methods [3, 20] have successfully been applied to various tasks in attribute-value learning. Much 'real-world' data, however, is structured – it has no natural representation as a tuple of constants. Defining kernels on individuals that cannot easily be described by a feature vector means crossing the boundary between attribute-value and relational learning. It enables support vector machines and other kernel methods to be applied more easily to complex representation spaces.

From an engineering point of view, the most interesting property of kernel methods and other (dis-)similarity-based learning algorithms is their modularity. By basing the learning algorithm only on the (dis-)similarity between individuals, the learning task and search strategy on one hand, and the hypothesis language on the other hand, can be separated.

The *kernel trick* is to replace the inner product in the representation space by an inner product in some feature space. The definition of the inner product thus determines the hypothesis language. The same trick can also be used with representation spaces that do not have a natural inner product defined on them. By defining a 'valid' kernel on these representations, they can be embedded into some linear space. Using a different kernel corresponds to a different embedding and thus to a different hypothesis language.

Crucial to the success of kernel-based learning algorithms is the extent to which the semantics of the domain are reflected in the definition of the kernel.

A 'good' kernel calculates a high similarity for examples in the same class and low similarity for examples in different classes. To express the semantics of the data in a machine-readable form, often strongly typed syntaxes are used. Syntax-driven kernels are an attempt to define 'good' kernels based on the semantics of the domain as described by the syntax of the representation. The definition of a kernel on structured data and the proof that this kernel is 'valid' are the main contributions of this paper.

This kernel is to be seen as the default kernel for structured data in the same sense in which the canonical dot product can be seen as the default kernel for vectors of numbers. Such default kernels may not always be the best choice. For that reason, gaussian, polynomial, or normalised versions of default kernels can be used.

The outline of the paper is as follows. Section 2 introduces kernel methods and defines what is meant by 'valid' and 'good' kernels. Section 3 gives an account of our knowledge representation formalism, which is a typed higher-order logic. Section 4 defines a kernel on the terms of this logic. Section 5 describes how these kernels can be adapted to particular domains. Section 6 illustrates the application of this kernel by some examples. Finally, some concluding remarks are given.

2 Kernel Methods

We distinguish two components of kernel methods, the kernel machine and the kernel function. Different kernel machines tackle different learning tasks, e.g., support vector machines for supervised learning, support vector clustering [1] for unsupervised learning, and kernel principal component analysis [20] for feature extraction. Thus the kernel machine also implements the search strategy[1]; however, the hypothesis language can later be adapted by 'plugging-in' a different kernel function. Thus the kernel function encapsulates the hypothesis language and all knowledge about the problem domain. Whenever two learning tasks are considered on the same problem domain, one can use the same hypothesis language by simply using the same kernel.

2.1 Classes of Kernels

A useful distinction between different classes of kernels is based on 'driving-force'. We distinguish between semantics, syntax, model, and data as the driving-force of the kernel definition. A similar terminology has been used previously in the context of constructive induction algorithms [2]

Semantics is the ideal driving-force for the definition of proximities. It is related to so-called 'knowledge-driven' approaches of incorporating expert knowledge into the domain representation. *Syntax* is often used in typed systems to

[1] The search strategy determines how the hypothesis space is searched and which hypotheses are preferred over others. For example, the search strategy of SVMs prefers large margin hypotheses over small margin hypotheses.

formally describe the semantics of the data. It is the most common driving force. In the simplest case, i.e., untyped attribute-value representation, it treats every attribute in the same way. *Models* extract useful knowledge from previous learning attempts. While this is often done to learn the semantics of the data from the data itself, it violates the encapsulation of the search strategy for kernel methods. Model-driven kernels are the Fisher kernel [12], the dynamic alignment kernel [23], the inner product in the 'weight of evidence' feature space [9], and other recently defined kernels based on probabilistic models of the data [21, 22]. *Data-driven* approaches use results obtained by analysing the training data. This also violates the encapsulation of the search strategy, as some search is needed for analysing the data. A data-driven approach is described in [6] where kernels are adapted by optimising an empirical measure, the so-called kernel-target-alignment.

As we want to maintain the modularity aspect, we will focus on syntax-driven approaches throughout the remainder of this paper, where syntax is understood as being carefully engineered to reflect the underlying semantics of the data.

2.2 Valid Kernels

Technically, a kernel k calculates an 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 ϕ explicitly, a kernel $k(x, y) = \langle \phi(x), \phi(y) \rangle$ calculates the inner product directly and performs the feature transformation only implicitly.

Whether, for a given function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, a feature transformation ϕ into some Hilbert space[2] $\phi : \mathcal{X} \to \mathcal{H}$ exists, such that k is the inner product in $\mathcal{H}$ ($\mathcal{H} \supseteq \mathit{span}(\{\phi(x) \,|\, x \in \mathcal{X}\})$), can be checked by verifying that the function is positive definite. This means that any set, whether a linear space or not, that admits a positive definite kernel can be embedded into a linear space. Thus, 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 1. *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 some 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, 11].

[2] A Hilbert space is a linear space endowed with a dot product and complete in the norm corresponding to that dot product.

2.3 Good Kernels

For a kernel method to perform well on some domain, validity of the kernel is not the only issue. While there is always a valid kernel that performs poorly ($k_0(x, y) = 0$), there is also always a valid kernel ($k_\nu(x, y) = \nu(x)\nu(y)$, where $\nu(z)$ is $+1$ if z is is a member of the concept and -1 otherwise) that performs ideally. We distinguish the following three issues crucial to 'good' kernels: completeness, correctness, and appropriateness. A similar terminology has been used previously in the context of constructive induction algorithms [2].

Completeness refers to the extent to which the knowledge incorporated in the proximity is sufficient for solving the problem at hand. A proximity is said to be complete if it takes into account all the information necessary to represent the concept that underlies the problem domain. *Correctness* refers to the extent to which the underlying semantics of the problem are obeyed in the proximity. *Appropriateness* refers to the extent to which examples that are close to each other in class membership are also 'close' to each other in the proximity space. Another frequently used term is 'smoothness of the kernel with respect to the class membership'.

Empirically, a correct and appropriate kernel exhibits two properties. A 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 generalises well, i.e., a learning algorithm achieves high accuracy when learning and validating on different parts of the data.

2.4 Kernels on Discrete Structures

Below we summarize prior work on syntax-driven kernels for discrete spaces that is most relevant in our context.

The best known kernel for representation spaces that are not mere attribute-value tuples is the convolution kernel proposed by Haussler [11]. It is defined there as

$$k_{conv}(x, y) = \sum_{\boldsymbol{x} \in R^{-1}(x), \boldsymbol{y} \in R^{-1}(y)} \prod_{d=1}^{D} k_d(x_d, y_d),$$

where R is a relation between instances x and their parts, i.e., R^{-1} decomposes an instance into a set of D-tuples. 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 a non-trivial task.

More specific kernels on discrete spaces have been described in [8]. There, kernels for elementary symbols, sets and multi-sets of elementary symbols, and Boolean domains are discussed along with concept classes that can be separated by linear classifiers using these kernels. An overview along with various extensions can be found in [10].

Other kernels on Boolean domains have recently been suggested in [19, 14, 15]. A string subsequence kernel is described in [17].

3 Knowledge Representation

For a syntax-driven kernel definition, one needs a knowledge representation formalism that is able to accurately and naturally model the underlying semantics of the data. The knowledge representation formalism we use is based on the principles of using a typed syntax and representing individuals as (closed) terms. The theory behind this knowledge representation formalism can be found in [16] and a brief outline is given in this section. The typed syntax is important for pruning search spaces and for modelling as closely as possible the semantics of the data in a human- and machine-readable form. The individuals-as-terms representation is a natural generalisation of the attribute-value representation and collects all information about an individual in a single term.

The setting is a typed, higher-order logic that provides a variety of important data types, including sets, multisets, and graphs for representing individuals. The logic is based on Church's simple theory of types [4] with several extensions. First, we assume there is given a set of type constructors $\mathfrak{T}$ of various arities. Included in $\mathfrak{T}$ is the constructor Ω of arity 0. The domain corresponding to Ω is the set containing just *True* and *False*, that is, the boolean values. The *types* of the logic are built up from the set of type constructors and a set of parameters (that is, type variables), using the symbol $\rightarrow$ (for function types) and $\times$ (for product types). For example, there is a type constructor *List* used to provide the list types. Thus, if α is a type, then *List* α is the type of lists whose elements have type α. A closed type is a type not containing any parameters, the set of all closed types is denoted by $\mathfrak{S}^c$. Standard types include *Nat* (the type of natural numbers).

There is also a set $\mathfrak{C}$ of constants of various types. Included in $\mathfrak{C}$ are $\top$ (true) and $\bot$ (false). Two different kinds of constants, *data constructors* and *functions*, are distinguished. In a knowledge representation context, data constructors are used to represent individuals. In a programming language context, data constructors are used to construct data values. (Data constructors are called functors in Prolog.) In contrast, functions are used to compute on data values; functions have definitions while data constructors do not. In the semantics for the logic, the data constructors are used to construct models (cf. Herbrand models for Prolog). A *signature* is the declared type of a constant. For example, the empty list constructor [] has signature *List* a, where a is a parameter. The list constructor : (usually written infix) has signature $a \rightarrow$ *List* $a \rightarrow$ *List* a.[3] Thus : expects two arguments, an element of type a and a list of type *List* a, and produces a new list of type *List* a. If a constant C has signature α, we denote this by $C : \alpha$.

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 from constants in $\mathfrak{C}$

[3] This could be read as $a \times$ *List* $a \rightarrow$ *List* a.

and a set of variables. $\mathfrak{L}$ denotes the set of all terms (obtained from a particular alphabet). A term of type Ω is called a *formula*. A function whose codomain type is Ω is called a *predicate*. In the logic, one can introduce the usual connectives and quantifiers as functions of appropriate types. Thus the connectives conjunction, $\wedge$, and disjunction, $\vee$, are functions of type $\Omega \to \Omega \to \Omega$. In addition, if t is of type Ω, the abstraction $\lambda x.t$ is written $\{x \mid t\}$ to emphasise its intended meaning as a set. There is also a tuple-forming notation $(\ldots)$. Thus, if $t_1, \ldots, t_n$ are terms of type $\tau_1, \ldots, \tau_n$, respectively, then $(t_1, \ldots, t_n)$ is a term of type $\tau_1 \times \cdots \times \tau_n$.

Now we come to the key definition of basic terms. Intuitively, basic terms represent the individuals that are the subject of learning (in Prolog, these would be the ground terms). Basic terms fall into one of three kinds: those that represent individuals that are lists, trees, and so on; those that represent sets, multisets, and so on; and those that represent tuples. The second kind are abstractions. For example, the basic term representing the set $\{1, 2\}$ is

$$\lambda x.\mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \bot,$$

and

$$\lambda x.\mathit{if}\ x = A\ \mathit{then}\ 42\ \mathit{else}\ \mathit{if}\ x = B\ \mathit{then}\ 21\ \mathit{else}\ 0$$

is the representation of the multiset with 42 occurrences of A and 21 occurrences of B (and nothing else). Thus we adopt abstractions of the form

$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$$

to represent (extensional) sets, multisets, and so on. The term s_0 here is called a default term and for the case of sets is $\bot$ and for multisets is 0. Generally, one can define default terms for each (closed) type. The set of default terms is denoted by $\mathfrak{D}$. (Full details on default terms are given in [16].)

Definition 2. *The set of* basic terms, $\mathfrak{B}$, *is defined inductively as follows.*

1. *If C is a data constructor having signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$, $t_1, \ldots, t_n \in \mathfrak{B}$ $(n \geq 0)$, and t is $C\ t_1 \ldots t_n \in \mathfrak{L}$, then $t \in \mathfrak{B}$.*
2. *If $t_1, \ldots, t_n \in \mathfrak{B}$, $s_1, \ldots, s_n \in \mathfrak{B}$ $(n \geq 0)$, $s_0 \in \mathfrak{D}$ and t is*

$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{L},$$

   then $t \in \mathfrak{B}$.
3. *If $t_1, \ldots, t_n \in \mathfrak{B}$ $(n \geq 0)$ and t is $(t_1, \ldots, t_n) \in \mathfrak{L}$, then $t \in \mathfrak{B}$.*

Part 1 of the definition of the set of basic terms states, in particular, that individual natural numbers, integers, and so on, are basic terms. Also a term formed by applying a data constructor to (all of) its arguments, each of which is a basic term, is a basic term. For example, lists are formed using the data constructors [] having signature *List a*, and : having signature $a \to \mathit{List}\ a \to \mathit{List}\ a$. Thus $A : B : C : []$ is the basic term of type *List* α representing the

list $[A, B, C]$, where A, B, and C are constants having signature α. Basic terms coming from Part 1 of the definition are called *basic structures* and always have a type of the form $T\alpha_1 \ldots \alpha_n$.

The abstractions formed in Part 2 of the definition are "almost constant" abstractions since they take the default term s_0 as value for all except a finite number of points in the domain. They are called *basic abstractions* and always have a type of the form $\beta \to \gamma$. This class of abstractions includes useful data types such as (finite) sets and multisets (assuming $\perp$ and 0 are default terms). More generally, basic abstractions can be regarded as lookup tables, with s_0 as the value for items not in the table. In fact, the precise definition of basic terms in [16] is a little more complicated in that, in the definition of basic abstractions, $t_1, \ldots, t_n$ are ordered and $s_1, \ldots, s_n$ cannot be default terms. These conditions avoid redundant representations of abstractions.

Part 3 of the definition of basic terms just states that one can form a tuple from basic terms and obtain a basic term. These terms are called *basic tuples* and always have a type of the form $\alpha_1 \times \cdots \times \alpha_n$.

Compared with Prolog, our knowledge representation offers a type system which can be used to express the structure of the hypothesis space and thus acts as a declarative bias. The other important extension are the abstractions, which allow us to use genuine sets and multisets. In fact, Prolog only has data constructors (functors), which are also used to emulate tuples.

It will be convenient to gather together all basic terms that have a type more general than some specific closed type. In this definition, if α and β are types, then α is *more general than* β if there exists a type substitution ξ such that $\beta = \alpha\xi$.

Definition 3. *For each* $\alpha \in \mathfrak{S}^c$, *define* $\mathfrak{B}_\alpha = \{t \in \mathfrak{B} \mid t$ *has type more general than* $\alpha\}$.

The intuitive meaning of $\mathfrak{B}_\alpha$ is that it is the set of terms representing individuals of type α.

For use in the definition of a kernel, we introduce some notation. If $s \in \mathfrak{B}_{\beta\to\gamma}$ and $t \in \mathfrak{B}_\beta$, then $V(s\ t)$ denotes the "value" returned when s is applied to t. (The precise definition is in [16].) For example, if s is $\lambda x.$*if* $x = A$ *then* 42 *else if* $x = B$ *then* 21 *else* 0 and t is A, then $V(s\ t) = 42$. Also, if $u \in \mathfrak{B}_{\beta\to\gamma}$, the *support* of u, denoted $supp(u)$, is the set $\{v \in \mathfrak{B}_\beta \mid V(u\ v) \notin \mathfrak{D}\}$. Thus, for the s above, $supp(s) = \{A, B\}$.

As an example of the use of the formalism, for (directed) graphs, there is a type constructor *Graph* such that the type of a graph is *Graph* $\nu\ \varepsilon$, where ν is the type of information in the vertices and ε is the type of information in the edges. *Graph* is defined by

$$Graph\ \nu\ \varepsilon = \{Label \times \nu\} \times \{(Label \times Label) \times \varepsilon\},$$

where *Label* is the type of labels. Note that this definition corresponds closely to the mathematical definition of a graph: each vertex is uniquely labelled and each edge is uniquely labelled by the ordered pair of labels of the vertices it connects.

4 Embedding Basic Terms in Linear Spaces

Having introduced kernels (in Section 2) and our knowledge representation formalism (in Section 3), we are now ready to define default kernels for basic terms. This definition of a kernel on basic terms assumes the existence of kernels on the various sets of data constructors. More precisely, for each type constructor $T \in \mathfrak{T}$, κ_T is assumed to be a positive definite kernel on the set of data constructors associated with T. For example, for the type constructor Nat, κ_{Nat} could be the *product kernel* defined by $\kappa_{Nat}(m,n) = mn$. For a type constructor M, the *matching kernel* κ_M is defined by $\kappa_M(x,y) = k_\delta(x,y) = 1$ if $x = y$, and 0, otherwise.

Definition 4. *The function $k : \mathfrak{B} \times \mathfrak{B} \to \mathbb{R}$ is defined inductively on the structure of terms in $\mathfrak{B}$ as follows. Let $s, t \in \mathfrak{B}$.*

1. *If $s, t \in \mathfrak{B}_\alpha$, where $\alpha = T\ \alpha_1 \ldots \alpha_k$, for some $T, \alpha_1, \ldots, \alpha_k$, then*

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

   where s is $C\ s_1 \ldots s_n$ and t is $D\ t_1 \ldots t_m$.
2. *If $s, t \in \mathfrak{B}_\alpha$, where $\alpha = \beta \to \gamma$, for some β, γ, then*

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

3. *If $s, t \in \mathfrak{B}_\alpha$, where $\alpha = \alpha_1 \times \cdots \times \alpha_n$, for some $\alpha_1, \ldots, \alpha_n$, then*

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

   where s is $(s_1, \ldots, s_n)$ and t is $(t_1, \ldots, t_n)$.
4. *If there does not exist $\alpha \in \mathfrak{S}^c$ such that $s, t \in \mathfrak{B}_\alpha$, then $k(s,t) = 0$.*

Example 1. Suppose that κ_{List} is the matching kernel. Let M be a nullary type constructor and $A, B, C, D : M$. Suppose that κ_M is the matching kernel. Let s be the list $[A, B, C]$ and t the list $[A, D]$. Then

$$\begin{aligned} k(s,t) &= \kappa_{List}((:),(:)) + k(A,A) + k([B,C],[D]) \\ &= 1 + \kappa_M(A,A) + \kappa_{List}((:),(:)) + k(B,D) + k([C],[]) \\ &= 1 + 1 + 1 + \kappa_M(B,D) + \kappa_{List}((:),[]) \\ &= 3 + 0 + 0 \\ &= 3. \end{aligned}$$

Thus, the kernel counts 1 for the first list constructor in both terms, 1 for the matching heads of the list, and 1 for the second list constructor, after which the two terms differ. Notice that the recursive matching of the two lists as performed by the kernel is similar to the kind of matching that is performed in anti-unification.

Example 2. Suppose that κ_Ω is the matching kernel. Let M be a nullary type constructor and $A, B, C, D : M$. Suppose that κ_M is the matching kernel. If s is the set $\{A, B, C\} \in \mathfrak{B}_{M\to\Omega}$ and t is the set $\{A, D\} \in \mathfrak{B}_{M\to\Omega}$, then

$$\begin{aligned} k(s,t) &= k(A,A) + k(A,D) + k(B,A) + k(B,D) + k(C,A) + k(C,D) \\ &= \kappa_M(A,A) + \kappa_M(A,D) + \kappa_M(B,A) + \kappa_M(B,D) + \kappa_M(C,A) \\ &\quad + \kappa_M(C,D) \\ &= 1 + 0 + 0 + 0 + 0 + 0 \\ &= 1. \end{aligned}$$

Thus, the kernel performs a pairwise match of the elements of each set. This could equivalently be seen as the inner product of the bitvectors representing the two sets.

Example 3. Suppose that κ_{Nat} is the product kernel. Let M be a nullary type constructor and $A, B, C, D : M$. Suppose that κ_M is the matching kernel. If s is $\langle A, A, B, C, C, C\rangle \in \mathfrak{B}_{M\to Nat}$ (where $\langle A, A, B, C, C, C\rangle$ is the multiset containing two occurrences of A, one of B, and three of C) and t is $\langle B, C, C, D\rangle \in \mathfrak{B}_{M\to Nat}$, then

$$\begin{aligned} k(s,t) &= k(2,1)k(A,B) + k(2,2)k(A,C) + k(2,1)k(A,D) \\ &\quad + k(1,1)k(B,B) + k(1,2)k(B,C) + k(1,1)k(B,D) \\ &\quad + k(3,1)k(C,B) + k(3,2)k(C,C) + k(3,1)k(C,D) \\ &= 1 \times 1 + 6 \times 1 \\ &= 7. \end{aligned}$$

If we represent multisets by multiplicity vectors, we again have that the kernel computes their inner product.

We can now formulate the main theoretical result of the paper.

Proposition 1. *Let $k : \mathfrak{B} \times \mathfrak{B} \to \mathbb{R}$ be the function defined in Definition 4. For each $\alpha \in \mathfrak{S}^c$, k is a positive definite kernel on $\mathfrak{B}_\alpha$.*

The inductive proof of Proposition 1 is given in the appendix. Here is an intuitive outline. First, assume that those kernels occurring on the right-hand side of each kernel definition are positive definite. Then the positive definiteness of the (left-hand side) kernel follows from the closure properties of the class of positive definite kernels. The kernel on basic structures is positive definite because of closure under sum, zero extension, and direct sum, and because the kernels defined on the data constructors are assumed to be positive definite. The

kernel on basic abstractions is positive definite as the function *supp* returns a finite set, and kernels are closed under zero extension, sum, and tensor product. The kernel on basic tuples is positive definite because of closure under direct sum.

5 Adapting Kernels

The kernel defined in the previous section closely follows the type structure of the individuals that are used for learning. As indicated, the kernel assumes the existence of atomic kernels for all data constructors used. These kernels can be the product kernel for numbers, the matching kernel which just checks whether the two constructors are the same, or a user-defined kernel. In addition, kernel modifiers can be used to customise the kernel definition to the domain at hand. In this section we first describe some commonly used kernel modifiers. After that, we suggest how atomic kernels and kernel modifiers can be specified by an extension of the Haskell language [13].

To incorporate domain knowledge into the kernel definition, it will frequently be necessary to modify the default kernels for a type. Below we formally describe these modifications in terms of a function $\kappa_{\text{modifier}} : \mathcal{P} \to (\mathcal{X} \times \mathcal{X} \to \mathbb{R}) \to (\mathcal{X} \times \mathcal{X} \to \mathbb{R})$ that – given a modifier and its parameters (an element of the parameter space $\mathcal{P}$) – maps any kernel to the modified kernel. For these modifiers, several choices are offered.

By default, no modifier is used, i.e.,

$$\kappa_{\text{default}}(k)(x, y) = k(x, y).$$

Instead, a polynomial version of the default kernel can be used:

$$\kappa_{\text{polynomial}}(p, l)(k)(x, y) = (k(x, y) + l)^p \,. \qquad (l \geq 0,\ p \in \mathbb{Z}^+)$$

Or a gaussian version:

$$\kappa_{\text{gaussian}}(\gamma)(k)(x, y) = e^{-\gamma[k(x,x)-2k(x,y)+k(y,y)]}. \qquad (\gamma > 0)$$

Another frequently used modification is the normalisation kernel:

$$\kappa_{\text{normalised}}(k)(x, y) = \frac{k(x, y)}{\sqrt{k(x, x)k(y, y)}}.$$

Other modifiers can be defined by the user.

We suggest that kernels be defined directly on the type structure (specifying the structure of the domain and the declarative bias). We introduce our suggested kernel definition syntax by means of an example: the East/West challenge [18].

```
eastbound :: Train -> Bool
type Train = Car -> Bool with modifier (gaussian 0.1)
type Car = (Shape,Length,Roof,Wheels,Load)
```

```
data Shape = Rectangle | Oval
data Length = Long | Short
data Roof = Flat | Peaked | None with kernel roofKernel
type Wheels = Int with kernel discreteKernel
type Load = (LShape,LNumber)
data LShape = Rectangle | Circle | Triangle
type LNumber = Int
```

The first line declares the learning target `eastbound` as a mapping from trains to Booleans. A train is a set of cars, and a car is a 5-tuple describing its shape, its length, its roof, its number of wheels, and its load. All of these are specified by data constructors except the load, which itself is a pair of data constructors describing the shape and number of loads.

The `with` keyword describes a property of a type, in this case kernels and kernel modifiers. The above declarations state that on trains we use a Gaussian kernel modifier with bandwidth 0.1. By default, for `Shape`, `Length` and `LShape` the matching kernel is used, while for `LNumber` the product kernel is used. The default kernel is overridden for `Wheels`, which is defined as an integer but uses the matching kernel instead. Finally, `Roof` has been endowed with a user-defined atomic kernel which could be defined as follows:

```
roofKernel :: Roof -> Roof -> Real
roofKernel x x = 1
roofKernel Flat Peaked = 0.5
roofKernel Peaked Flat = 0.5
roofKernel x y = 0
```

This kernel counts 1 for identical roofs, 0.5 for matching flat against peaked roofs, and 0 in all other cases (i.e., whenever one car is open and the other is closed).

Finally, the normalisation modifier could be implemented as follows:

```
normalised :: (t->t->Real) -> t -> t -> Real
normalised k x y = (k x y) / sqrt ((k x x) * (k y y))
```

6 Example Applications

Having presented our kernel definition, it is important to note that aside from being used with kernel methods, our kernel function can also be used with other (dis-)similarity-based algorithms. A normalised kernel is a canonical similarity function; a metric can be defined on the kernel in the standard manner ($d(x, y) = \sqrt{k(x, x) - 2k(x, y) + k(y, y)}$). Thus learning algorithms like nearest neighbour and k-means can easily be extended to structured data.

In this section, we empirically investigate the appropriateness of our kernel definitions on some domains. The implementation of most algorithms mentioned below has been simplified by using the Weka data mining toolkit [24].

6.1 East/West Challenge

We performed some experiments with the East/West challenge dataset. We used the default kernels for all types, i.e., the product kernel for all numbers, the matching kernel for all other atomic types, and no kernel modifiers. As this toy data set only contains 20 labelled instances, the aim of our experiments was not to achieve a high predictive accuracy but to check whether this problem can actually be separated using our default kernel. For that, we applied a support vector machine and a 3-nearest-neighbour classifier to the full data set. In both experiments, we achieved 100% accuracy, verifying that the data is indeed separable with the default kernels.

6.2 Spatial Clustering

Consider the problem of clustering spatially close and thematically similar data points. This problem occurs, for example, when given demographic data about households in a city and trying to optimise facility locations given this demographic data. The location planning algorithms can usually only deal with a fairly small number of customers (less than 1000) and even for small cities the number of households easily exceeds 10000. Therefore, several households have to be aggregated so that as little information as possible is lost. Thus the households that are aggregated have to be spatially close (so that little geographic information is lost) and similar in their demographic description (so that little demographic information is lost). The problem is to automatically find such an aggregation using an unsupervised learning algorithm.

Due to the difficulty in obtaining suitable data, we investigated this problem on a slightly smaller scale. The demographic data was already aggregated for data protection and anonymity reasons such that information is given not on a household level but on a (part of) street level. The data set describes roughly 500 points in a small German city by its geographic co-ordinates and 76 statistics, e.g., the number of people above or below certain age levels, the number of people above or below certain income levels, and the number of males or females living in a small area around the data point.

The simplest way to represent this data is a feature vector with 78 entries (2 for the x, y co-ordinates and 76 for the statistics). Drawing the results of a simple k-means algorithm on this representation clearly shows that although the spatial co-ordinates are taken into account, spatially compact clusters cannot be achieved. This is due to the fact that the semantics of the co-ordinates and the demographic statistics are different.

An alternative representation along with the kernel specification is as follows:

```
type SpacialStat = GeoInfo -> Statistics
type GeoInfo = (Real,Real) with modifier (gaussian 0.1)
type Statistics = (Real,Real,...,Real) with modifier normalised
```

Using this representation and applying a version of the k-means algorithm[4] with the given kernel shows that the clusters are spatially compact (compactness depending on the choice of the kernel bandwidth).

Illustrations of results can be found online[5]. Instances belonging to the same cluster are represented by the same colour. The street map and the buildings of the city are shown in grey.

6.3 Drug Activity Prediction

A frequently used concept class on data with limited structure are multi-instance concepts. Multi-instance learning problems occur whenever individuals cannot be described by a single characteristic feature vector, but require a bag of vectors. A popular real-world example of a multi-instance problem is the prediction of drug activity, introduced in [7].

It is common in drug activity prediction to represent a molecule by a bag of descriptions of its different conformations. A drug is active if one of its conformations binds well to enzymes or cell-surface receptors. Each conformation is described by a feature vector where each component corresponds to one ray emanating from the origin and measuring the distance to the molecule surface. In the musk domain, 162 uniformly distributed rays have been chosen to represent each conformation. Additionally, four further features are used that describe the position of an oxygen atom in the conformation.

The formal specification of the structure of the musk data set along with the kernel applied in [10] is as follows:

```
type Molecule = Con -> Int
type Con = (Rays,Distance,Offset) with modifier (gaussian 1e-5.5)
type Rays = (Real,Real,...,Real)
type Offset = (Real,Real,Real)
type Distance = Real
```

The best result achieved in the literature on musk1 is 96.8%, and the next five best results range between 92.4% and 88.9%. The support vector machine using the above kernel achieved 87% accuracy, better results can be achieved with smaller γ. The best result achieved in the literature on musk2 is 96.0%, and the next five best results range between 89.2% and 82.5%. The support vector machine using the above kernel achieved 92.2% accuracy. For a more detailed evaluation and discussion of these results, the reader is referred to [10].

7 Conclusions

Bringing together kernel methods and structured data is an important direction for practical machine learning research. This can be done by defining a positive

[4] Note that performing k-means in feature space requires some modifications of the algorithm. A description is beyond the scope of this paper.

[5] `http://www.ais.fraunhofer.de/~thomasg/SpatialClustering/`

definite kernel on structured data and thus embedding structured data into a linear space. In this paper we defined a kernel on structured data, proved that it positive definite, and showed some example applications.

Our kernel definition follows a 'syntax-driven' approach making use of a knowledge representation formalism that is able to accurately and naturally model the underlying semantics of structured data. It is based on the principles of using a typed syntax and representing individuals as (closed) terms. The typed syntax is important for pruning search spaces and for modelling as closely as possible the semantics of the data in a human- and machine-readable form. The individuals-as-terms representation is a simple and natural generalisation of the attribute-value representation and collects all information about an individual in a single term. In spite of this simplicity, the knowledge representation formalism is still powerful enough to accurately model highly structured data such as graphs.

The definition of our kernel, along with the example applications presented above, show that structured data can reasonably be embedded in linear spaces. The embedding is complete in the sense that it incorporates all information that is present in an individual. Its correctness has been illustrated on a toy example of classifying trains; its appropriateness has been verified on a drug activity prediction domain.

Acknowledgements

Research supported in part by the Esprit V project (IST-1999-11495) *Data Mining and Decision Support for Business Competitiveness: Solomon Virtual Enterprise* and by the BMBF funded project *KogiPlan*.

References

1. A. Ben-Hur, D. Horn, H. T. Siegelmann, and V. Vapnik. Support vector clustering. *Journal of Machine Learning Research*, 2:125–137, Dec. 2001.
2. E. Bloedorn, R. Michalski, and J. Wnek. Matching methods with problems: A comparative analysis of constructive induction approaches. Technical report, Machine Learning and Inference Laboratory, MLI 94-2, George Mason University, Fairfax, VA., 1994.
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, Pittsburgh, PA, July 1992. ACM Press.
4. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
5. N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines (and Other Kernel-Based Learning Methods)*. Cambridge University Press, 2000.
6. N. Cristianini, J. Shawe-Taylor, A. Elisseeff, and J. Kandola. On kernel-target alignment. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14. The MIT Press, 2002.

7. 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.
8. T. Gärtner. Kernel-based feature space transformation in inductive logic programming. Master's thesis, University of Bristol, 2000.
9. T. Gärtner and P. A. Flach. WBCsvm: Weighted bayesian classification based on support vector machines. In C. E. Brodley and A. P. Danyluk, editors, *Proceedings of the 18th International Conference on Machine Learning*, pages 207–209. Morgan Kaufmann, June 2001.
10. T. Gärtner, P. A. Flach, A. Kowalczyk, and A. J. Smola. Multi-instance kernels. In *Proceedings of the 19th International Conference on Machine Learning*, to appear.
11. D. Haussler. Convolution kernels on discrete structures. Technical report, Department of Computer Science, University of California at Santa Cruz, 1999.
12. T. S. Jaakkola and D. Haussler. Exploiting generative models in discriminative classifiers. In *Advances in Neural Information Processing Systems*, volume 10, 1999.
13. S. P. Jones and J. H. (editors). Haskell98: A non-strict purely functional language. Available at `http://haskell.org/`.
14. R. Khardon, D. Roth, and R. Servedio. Efficiency versus convergence of boolean kernels for on-line learning algorithms. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14. The MIT Press, 2002.
15. A. Kowalczyk, A. J. Smola, and R. C. Williamson. Kernel machines and boolean functions. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14. The MIT Press, 2002.
16. J. Lloyd. Knowledge representation, computation, and learning in higher-order logic. Available at `http://csl.anu.edu.au/~jwl`, 2001.
17. 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. The MIT Press, 2001.
18. D. Michie, S. Muggleton, D. Page, and A. Srinivasan. To the international computing community: A new eastwest challenge. Technical report, Oxford University Computing laboratory, Oxford,UK, 1994.
19. K. Sadohara. Learning of boolean functions using support vector machines. In N. Abe, R. Khardon, and T. Zeugmann, editors, *Proceedings of the 12th Conference on Algorithmic Learning Theory*, pages 106–118. Springer-Verlag, 2001.
20. B. Schölkopf and A. J. Smola. *Learning with Kernels*. The MIT Press, 2002.
21. 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. The MIT Press, 2002.
22. 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. The MIT Press, 2002.
23. C. Watkins. Dynamic alignment kernels. In A. Smola, P. Bartlett, and B. Schölkopf, editors, *Advances in large margin classifiers*, pages 39–50. The MIT Press, 2000.
24. I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java implementations*. Morgan Kaufmann, 2000.

A Proof of the Proposition

Before giving the proof, which is an induction argument, some preparation is needed. The key idea is to base the induction on a 'bottom-up' definition of $\mathfrak{B}$. Here is the relevant definition.

Definition 5. *Define* $\{\mathfrak{B}_m\}_{m\in\mathbb{N}}$ *inductively as follows.*

$$\begin{aligned}
\mathfrak{B}_0 &= \{C \mid C \textit{ is a data constructor of arity } 0\}\\
\mathfrak{B}_{m+1} &= \{C\ t_1 \ldots t_n \in \mathfrak{L} \mid C \textit{ is a data constructor of arity } n \textit{ and}\\
&\qquad\qquad t_1, \ldots, t_n \in \mathfrak{B}_m (n \geq 0)\}\\
&\quad \cup \{\lambda x.\textit{if } x = t_1 \textit{ then } s_1 \textit{ else } \ldots \textit{ if } x = t_n \textit{ then } s_n \textit{ else } s_0 \in \mathfrak{L} \mid\\
&\qquad\qquad t_1, \ldots, t_n \in \mathfrak{B}_m, s_1, \ldots, s_n \in \mathfrak{B}_m, \textit{ and } s_0 \in \mathfrak{D}\}\\
&\quad \cup \{(t_1, \ldots, t_n) \in \mathfrak{L} \mid t_1, \ldots, t_n \in \mathfrak{B}_m\}.
\end{aligned}$$

One can prove that $\mathfrak{B}_m \subseteq \mathfrak{B}_{m+1}$, for $m \in \mathbb{N}$, and that $\mathfrak{B} = \bigcup_{m\in\mathbb{N}} \mathfrak{B}_m$. Now the proof of Proposition 1 can be given.

Proof. First the symmetry of k on each $\mathfrak{B}_\alpha$ is established. For each $m \in \mathbb{N}$, let $SYM(m)$ be the property:

> For all $\alpha \in \mathfrak{S}^c$ and $s, t \in \mathfrak{B}_\alpha \cap \mathfrak{B}_m$, it follows that $k(s,t) = k(t,s)$.

It is shown by induction that $SYM(m)$ holds, for all $m \in \mathbb{N}$. The symmetry of k on each $\mathfrak{B}_\alpha$ follows immediately from this since, given $s, t \in \mathfrak{B}_\alpha$, there exists an m such that $s, t \in \mathfrak{B}_m$ (because $\mathfrak{B} = \bigcup_{m\in\mathbb{N}} \mathfrak{B}_m$ and $\mathfrak{B}_m \subseteq \mathfrak{B}_{m+1}$, for all $m \in \mathbb{N}$).

First it is shown that $SYM(0)$ holds. In this case, s and t are data constructors of arity 0 associated with the same type constructor T, say. By definition, $k(s,t) = \kappa_T(s,t)$ and the result follows because κ_T is symmetric.

Now assume that $SYM(m)$ holds. It is proved that $SYM(m+1)$ also holds. Thus suppose that $\alpha \in \mathfrak{S}^c$ and $s, t \in \mathfrak{B}_\alpha \cap \mathfrak{B}_{m+1}$. It has to be shown that $k(s,t) = k(t,s)$. There are three cases to consider corresponding to α having the form $T\ \alpha_1 \ldots \alpha_k$, $\beta \to \gamma$, or $\alpha_1 \times \cdots \times \alpha_m$. In each case, it is easy to see from the definition of k and the induction hypothesis that $k(s,t) = k(t,s)$. This completes the proof that k is symmetric on each $\mathfrak{B}_\alpha$.

For the remaining part of the proof, for each $m \in \mathbb{N}$, let $PD(m)$ be the property:

> For all $n \in \mathbb{Z}^+, \alpha \in \mathfrak{S}^c, t_1, \ldots, t_n \in \mathfrak{B}_\alpha \cap \mathfrak{B}_m$, and $c_1, \ldots, c_n \in \mathbb{R}$, it follows that $\sum_{i,j\in\{1,\ldots,n\}} c_i\ c_j\ k(t_i, t_j) \geq 0$.

It is shown by induction that $PD(m)$ holds, for all $m \in \mathbb{N}$. The remaining condition for positive definiteness follows immediately from this since, given $t_1, \ldots, t_n \in \mathfrak{B}_\alpha$, there exists an m such that $t_1, \ldots, t_n \in \mathfrak{B}_m$.

First it is shown that $PD(0)$ holds. In this case, each t_i is a data constructor of arity 0 associated with the same type constructor T, say. By definition, $k(t_i, t_j) =$

$\kappa_T(t_i, t_j)$, for each i and j, and the result follows since κ_T is assumed to be positive definite.

Now assume that $PD(m)$ holds. It is proved that $PD(m+1)$ also holds. Thus suppose that $n \in \mathbb{Z}^+, \alpha \in \mathfrak{S}^c, t_1, \ldots, t_n \in \mathfrak{B}_\alpha \cap \mathfrak{B}_{m+1}$, and $c_1, \ldots, c_n \in \mathbb{R}$. It has to be shown that $\sum_{i,j \in \{1,\ldots,n\}} c_i \; c_j \; k(t_i, t_j) \geq 0$. There are three cases to consider.

1. Let $\alpha = T \; \alpha_1 \ldots \alpha_k$. Suppose that $t_i = C_i \; t_i^{(1)} \ldots t_i^{(m_i)}$, where $m_i \geq 0$, for $i = 1, \ldots, n$. Let $\mathcal{C} = \{C_i \mid i = 1, \ldots, n\}$. Then

$$\begin{aligned}
&\sum_{i,j \in \{1,\ldots,n\}} c_i \; c_j \; k(t_i, t_j) \\
&= \sum_{i,j \in \{1,\ldots,n\}} c_i \; c_j \; \kappa_T(C_i, C_j) \\
&\quad + \sum_{\substack{i,j \in \{1,\ldots,n\} \\ C_i = C_j}} c_i \; c_j \sum_{l \in \{1,\ldots,arity(C_i)\}} k(t_i^{(l)}, t_j^{(l)}).
\end{aligned}$$

   Now

$$\sum_{i,j \in \{1,\ldots,n\}} c_i \; c_j \; \kappa_T(C_i, C_j) \geq 0$$

   using the fact that κ_T is a positive definite kernel on the set of data constructors associated with T. Also

$$\begin{aligned}
&\sum_{\substack{i,j \in \{1,\ldots,n\} \\ C_i = C_j}} c_i \; c_j \sum_{l \in \{1,\ldots,arity(C_i)\}} k(t_i^{(l)}, t_j^{(l)}) \\
&= \sum_{C \in \mathcal{C}} \sum_{\substack{i,j \in \{1,\ldots,n\} \\ C_i = C_j = C}} \sum_{l \in \{1,\ldots,arity(C)\}} c_i \; c_j \; k(t_i^{(l)}, t_j^{(l)}) \\
&= \sum_{C \in \mathcal{C}} \sum_{l \in \{1,\ldots,arity(C)\}} \sum_{\substack{i,j \in \{1,\ldots,n\} \\ C_i = C_j = C}} c_i \; c_j \; k(t_i^{(l)}, t_j^{(l)}) \\
&\geq 0,
\end{aligned}$$

   by the induction hypothesis.

2. Let $\alpha = \beta \to \gamma$. Then

$$\begin{aligned}
&\sum_{i,j\in\{1,\ldots,n\}} c_i\ c_j\ k(t_i, t_j) \\
&= \sum_{i,j\in\{1,\ldots,n\}} c_i\ c_j \sum_{\substack{u\in supp(t_i)\\ v\in supp(t_j)}} k(V(t_i\ u), V(t_j\ v)) \cdot k(u, v) \\
&= \sum_{i,j\in\{1,\ldots,n\}} \sum_{\substack{u\in supp(t_i)\\ v\in supp(t_j)}} c_i\ c_j\ k(V(t_i\ u), V(t_j\ v)) \cdot k(u, v) \\
&= \sum_{\substack{(i,u),(j,v)\in\\ \{(k,w)\mid k=1,\ldots,n \text{ and } w\in supp(t_k)\}}} c_i\ c_j\ k(V(t_i\ u), V(t_j\ v)) \cdot k(u, v) \\
&\geq 0.
\end{aligned}$$

For the last step, we proceed as follows. By the induction hypothesis, k is positive definite on both $\mathfrak{B}_\beta \cap \mathfrak{B}_m$ and $\mathfrak{B}_\gamma \cap \mathfrak{B}_m$. Hence the function

$$h : ((\mathfrak{B}_\beta \cap \mathfrak{B}_m) \times (\mathfrak{B}_\gamma \cap \mathfrak{B}_m)) \times ((\mathfrak{B}_\beta \cap \mathfrak{B}_m) \times (\mathfrak{B}_\gamma \cap \mathfrak{B}_m)) \to \mathbb{R}$$

defined by

$$h((u, y), (v, z)) = k(u, v) \cdot k(y, z)$$

is positive definite, since h is a tensor product of positive definite kernels [20]. Now consider the set

$$\{(u, V(t_i\ u)) \mid i = 1, \ldots, n \text{ and } u \in supp(t_i)\}$$

of points in $(\mathfrak{B}_\beta \cap \mathfrak{B}_m) \times (\mathfrak{B}_\gamma \cap \mathfrak{B}_m)$ and the corresponding set of constants

$$\{c_{i,u} \mid i = 1, \ldots, n \text{ and } u \in supp(t_i)\},$$

where $c_{i,u} = c_i$, for all $i = 1, \ldots, n$ and $u \in supp(t_i)$.

3. Let $\alpha = \alpha_1 \times \cdots \times \alpha_m$. Suppose that $t_i = (t_i^{(1)}, \ldots, t_i^{(m)})$, for $i = 1, \ldots, n$. Then

$$\begin{aligned}
&\sum_{i,j\in\{1,\ldots,n\}} c_i\ c_j\ k(t_i, t_j) \\
&= \sum_{i,j\in\{1,\ldots,n\}} c_i\ c_j (\sum_{l=1}^{m} k(t_i^{(l)}, t_j^{(l)})) \\
&= \sum_{l=1}^{m} \sum_{i,j\in\{1,\ldots,n\}} c_i\ c_j\ k(t_i^{(l)}, t_j^{(l)}) \\
&\geq 0,
\end{aligned}$$

by the induction hypothesis. □