Learning with Kernels and Logical Representations

Paolo Frasconi and Andrea Passerini

Machine Learning and Neural Networks Group
Dipartimento di Sistemi e Informatica
Università degli Studi di Firenze, Italy
`http://www.dsi.unifi.it/neural/`

Abstract. In this chapter, we describe a view of statistical learning in the inductive logic programming setting based on kernel methods. The relational representation of data and background knowledge are used to form a kernel function, enabling us to subsequently apply a number of kernel-based statistical learning algorithms. Different representational frameworks and associated algorithms are explored in this chapter. In *kernels on Prolog proof trees*, the representation of an example is obtained by recording the execution trace of a program expressing background knowledge. In *declarative kernels*, features are directly associated with mereotopological relations. Finally, in *kFOIL*, features correspond to the truth values of clauses dynamically generated by a greedy search algorithm guided by the empirical risk.

1 Introduction

Kernel methods are one of the highly popular state-of-the-art techniques in machine learning [1,2]. They make it possible to design generic learning algorithms, abstracting away details about data types, a trait that makes them especially appealing in relational domains for dealing with structured objects such as sequences [3,4,5,6], trees [7,8], or graphs [9,10,11,12,13,14]. When using kernel machines, instances are mapped to a Hilbert space commonly called the *feature space*, where the kernel function is the inner product. In principle, there is no need to explicitly represent feature vectors as an intermediate step, as it happens for example with many propositionalization schemes [15,16]. This trick has often been exploited from the algorithmic point of view when the kernel function can be computed efficiently in spite of very high-dimensional feature spaces.

In the simplest supervised learning settings (such as classification and regression with independent examples), all representational issues are dealt with by the kernel, whereas the learning algorithm has mainly a statistical role. This also means that background knowledge about the problem at hand should be injected into the learning process mainly by encoding it into the kernel function. This activity is sometimes carried out in an ad-hoc manner by guessing interesting features. However, if domain knowledge has been encoded formally (e.g. in a declarative fashion using first-order logic, or by means of ontologies), then it makes sense to use these representations as a starting point for building the kernel. An example along these lines is the work by Cumby & Roth [16] that uses description logic to specify features and that has been subsequently extended to specify kernels [17].

Within the field of inductive logic programming (ILP), a related area of research is the definition of distances in relational domains [18,19,20]. For every kernel function (intuitively, a kernel corresponds to a notion of similarity) elementary geometry allows us to derive an associated distance function in the feature space. Turning distances into valid (positive semi-definite) kernels, however, is not possible in general as the axiomatic definition of distance imposes less constraints. Thus, work on distance-based relational learning cannot be immediately translated into equivalent kernel methods

In this chapter, we describe a number of methods based on the combination of logical representations, kernel machines, and statistical learning. There are several reasons why seeking links between kernel methods and probabilistic logical learning can be interesting. First, background knowledge about a domain may be already available and described in a logical representation language. As we noted above, kernels are usually the main entry point for plugging background knowledge in the learning process. Therefore, from an engineering point of view, developing a flexible and systematic approach to kernel design starting from logical representations seems to be a natural choice. Second, learning algorithms based on kernel machines are very efficient from a computational point of view. After the Gram matrix has been computed, learning often consists of finding the (unique) solution of a convex numerical optimization problem. Additional efficiency can be gained by exploiting the sparsity of the structure of the solution, as it happens for example with support vector machines [21]. This scenario contrasts with the computational requirements of many ILP schemes that need to search hypotheses in a complex discrete space of logical programs [22]. Third, several types of learning problems, besides classification, can be solved under a uniform framework, including regression [23], ranking (ordinal regression) [24], novelty detection (one-class classification) [25], clustering [26], and principal component analysis [27]. Logic-based learning, on the other hand, has mainly focused on classification while other tasks such as regression often need ad-hoc solutions, except perhaps in the case of decision trees [28,29]. Fourth, kernel based learning algorithms can be naturally linked to regularization theory, where the complexity of the function calculated by a learning algorithm can be controlled via its norm in the so-called reproducing kernel Hilbert space [30]. Regularization restores well-posedness in learning algorithms based on empirical risk minimization, i.e. it ensures that the solution to the learning problem is unique and stable (small perturbations in the data lead to small variations of the learned function). Of course, uncertainty can also be handled using other probabilistic logic learning schemes, like those extensively presented elsewhere in this book, but from a different and complementary angle. Kernel-based approaches can be seen as taking the discriminant direction of learning, i.e. they attempt to identify the optimal prediction function (i.e. the well known Bayes function in the case of binary classification). Theory shows that machines based on regularized empirical risk minimization, such as the support vector machine (SVM), do converge to the optimal function as the number of examples goes to infinity [31,32]. This is a major difference with respect to other probabilistic ILP approaches that take the generative direction of modeling. Generative models require more knowledge about the structural form of the probability densities than their discriminant counterparts. If the underlying assumptions are wrong, they may converge to a sub-optimal asymptotic error, although faster than discriminant models constructed on the same model space of probability distributions (a classic propositional example is the model pair formed by Naive Bayes and logistic regression [33]).

There are, on the other hand, disadvantages when embracing the above framework, compared to learning with other probabilistic logic representations. Since the learning process only focuses on the discriminant function, it does not discover any new portion of the theory explaining the phenomena that underly the data. Additionally the learned function does not provide any easily understandable explanations as to why certain predictions are associated with the input.

This chapter is a detailed review of several approaches that have been developed within APrIL II for statistical learning with kernels in the ILP setting. We start in Section 2 explaining some basic concepts about the statistical and the logical learning settings, in order to clarify our assumptions and for the benefit of readers who are not familiar with both areas. In Section 3 we present kernels on Prolog ground terms [34], a specialization to first-order logic of kernels on logical individuals introduced by Gaertner et al. [35]. In Section 4, we describe declarative kernels, a general approach for describing knowledge-informed kernels based on relations related to decomposition into parts and connection between parts. In Section 5, we present kernels based on Prolog proof trees [36], an approach where first a program is ran over instances, to inspect interesting features, and then program traces are compared by means of a kernel on ground terms. In Section 6, we describe kFOIL [37], an algorithm that is especially interesting from the point of view of the understandability of the learned solution. It constructs a kernel from data using a simple inductive logic programming engine (FOIL [38]) to generate the clauses that define the kernel. Finally, in Section 7, we report about two real-world applications that have been tackled with these techniques: information extraction from scientific literature, and prediction of protein folds. For the latter application, we also report novel results and comparisons for the task of multiclass protein fold classification.

2 Notation and Background Concepts

2.1 Supervised Learning in the Statistical Setting

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

2.2 Supervised Learning with Kernel Machines

A kernel is a positive semi-definite (psd) symmetric function $K : \mathcal{X} \times \mathcal{X} \mapsto R$ that generalizes the notion of inner product to arbitrary domains [2]. Positive semi-definite here means that for all m and all finite data sets of size m, the Gram matrix with entries $K(x_i, x_j), i, j = 1, \ldots m$ has nonnegative eigenvalues. Each instance x is mapped to a corresponding element $\phi(x)$ in a Hilbert space commonly called the *feature space*. For example, a feature of a graph may be associated with the existence of a path with certain node labels; in this way, a graph is mapped to a sequence of booleans, each associated with a string over the node labels alphabet. Given this mapping, the kernel function is, by definition, the inner product $K(x, x') = \langle \phi(x), \phi(x') \rangle$. Mercer's theorem ensures that for any symmetric and psd function $K : \mathcal{X} \times \mathcal{X} \mapsto R$ there exists a mapping in a Hilbert space where K is the inner product.

When using kernel methods in supervised learning, the hypothesis space, denoted $\mathcal{F}_K$, is the so-called reproducing kernel Hilbert space (RKHS) associated with K [30]. Learning consists of solving the following Tikhonov regularized problem:

$$f = \arg\min_{h \in \mathcal{F}_K} C \sum_{i=1}^{m} V(y_i, h(x_i)) + \|h\|_K \tag{1}$$

where $V(y, h(x))$ is a positive function measuring the loss incurred in predicting $h(x)$ when the target is y, C is a positive regularization constant, and $\|\cdot\|_K$ is the norm in the RKHS. Popular algorithms in this framework include support vector machines [21], obtained using the "hinge" loss $V(y, a) = \max\{1 - ya, 0\}$, kernel ridge regression [39,40], obtained using the quadratic loss $V(y, a) = (v - a)^2$, and support vector regression [23], obtained using the ϵ-insensitive loss $V(y, a) = \max\{|y - a| - \epsilon, 0\}$. The representer theorem [41] shows that the solution to the above problem can be expressed as a linear combination of the kernel basis functions evaluated at the training examples:

$$f(x) = \sum_{i=1}^{m} c_i K(x, x_i) \tag{2}$$

where c_i are real coefficients expressing the solution of Eq. (1). The above form also encompasses the solution found by other algorithms not based on Eq. (1), such as the kernel perceptron [42].

2.3 Convolution Kernels for Discrete Structures

Suppose the instance space $\mathcal{X}$ is a set of composite structures and for $x \in \mathcal{X}$ let $\vec{x} = x_1, \ldots, x_D$ denote a tuple of "parts" of x, with $x_d \in \mathcal{X}_d$ (the d-th part type) for all $i \in [1, D]$. This decomposition can be formally represented by a relation R on $\mathcal{X}_1 \times \cdots \times \mathcal{X}_D \times \mathcal{X}$. For each $x \in \mathcal{X}$, $R^{-1}(x) = \{\vec{x} \in \vec{\mathcal{X}} : R(\vec{x}, x)\}$ denotes the multiset of all possible decompositions of x.

In order to complete the definition of convolution kernels, we assume that a kernel function $K_d : \mathcal{X}_d \times \mathcal{X}_d \to \mathbb{R}$ is given for each part type $\mathcal{X}_d, d = 1, \ldots, D$. The *R-convolution* kernel [43] is then defined as follows:

$$K_{R,\otimes}(x, z) = \sum_{(x_1,\ldots,x_D)\in R^{-1}(x)} \sum_{(z_1,\ldots,z_D)\in R^{-1}(z)} \prod_{d=1}^{D} K_d(x_d, z_d). \tag{3}$$

In the above formulation, a tensor product has been used to combine kernels between different part types. Haussler [43] showed that the tensor product is closed under positive definiteness and, therefore, R-convolution kernels that use tensor product as a combination operator are positive definite, provided that all K_d are. The result also holds for combinations based on other closed operators, such as direct sum, yielding

$$K_{R,\oplus}(x, z) = \sum_{(x_1,\ldots,x_D)\in R^{-1}(x)} \sum_{(z_1,\ldots,z_D)\in R^{-1}(z)} \sum_{d=1}^{D} K_d(x_d, z_d). \tag{4}$$

Convolution or decomposition kernels form a vast class of functions and need to be specialized to capture the correct notion of similarity required by the task at hand. For example, several kernels on discrete structures have been designed using $D = 1$ and defining a simple concept of part. These "all-substructures kernels" basically count the number of co-occurrences of substructures in two decomposable objects. Plain counting can be easily achieved by using the exact match kernel

$$\delta(x, z) = \begin{cases} 1 \text{ if } x = z \\ 0 \text{ otherwise.} \end{cases} \tag{5}$$

Interesting discrete data types that have been thoroughly studied in the literature include sequences [44,5,6], trees [45,8], and graphs [11,9]. The *set kernel* [2] is a special case of convolution kernel that will prove useful in defining logical kernels presented in this chapter and that has been also used in the context of multi-instance learning [46]. Suppose instances are sets and let us define the part-of relation as the usual set-membership. The kernel over sets K_{set} is then obtained from kernels between set members K_{member} as follows:

$$K_{set}(x, z) = \sum_{\xi\in x} \sum_{\zeta\in z} K_{member}(\xi, \zeta). \tag{6}$$

2.4 Normalization and Composition

In order to reduce the dependence on the dimension of the objects, kernels over discrete structures are often normalized. A common choice is that of using normalization in feature space, i.e., given a convolution kernel K_R:

$$K_{norm}(x,z) = \frac{K_R(x,z)}{\sqrt{K_R(x,x)}\sqrt{K_R(z,z)}}. \tag{7}$$

In the case of set kernels, an alternative is that of dividing by the cardinalities of the two sets, thus computing the mean value between pairwise comparisons[1]:

$$K_{mean}(x,z) = \frac{K_{set}(x,z)}{|x||z|}. \tag{8}$$

Richer families of kernels on data structures can be formed by applying composition to the feature mapping induced by a convolution kernel. For example, a convolution kernel K_R can be combined with a Gaussian kernel as follows:

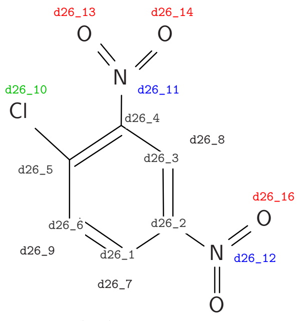
$$K(x,z) = \exp\left(-\gamma\Big(K_R(x,x) - 2K_R(x,z) + K_R(z,z)\Big)\right). \tag{9}$$

2.5 A Framework for Statistical Logical Learning

One of the standard ILP frameworks is that of learning from entailment. In this setting, the learner is given a set of positive and negative examples, $\mathcal{D}^+$ and $\mathcal{D}^-$, respectively (in the form of ground facts), and a background theory $\mathcal{B}$ (as a set of definite clauses) and has to induce a hypothesis $\mathcal{H}$ (also a set of definite clauses) such that $\mathcal{B} \cup \mathcal{H}$ covers all positive examples and none of the negative ones. More formally, $\forall p(x) \in \mathcal{D}^+ : \mathcal{B} \cup \mathcal{H} \models p(x)$ and $\forall p(x) \in \mathcal{D}^- : \mathcal{B} \cup \mathcal{H} \not\models p(x)$. Note that the meaning of term *hypothesis* in this context is related but not coincident with its meaning in statistical learning, where the hypothesis space is a class of functions mapping instances to targets.

We now develop a framework aiming to combine some of the advantages of the statistical and the ILP settings, in particular: efficiency, stability, generality, and the possibility of describing background knowledge in a flexible declarative language. As in the ILP setting, we assume that a background theory $\mathcal{B}$ is available as a set of definite clauses. This background theory is divided into *intensional* predicates, $\mathcal{B}_I$, and *extensional* predicates, $\mathcal{B}_E$, the former relevant to all examples, and the latter that specify facts about specific examples. As in [48], examples will simply be individuals, i.e., first-order logic objects, syntactically denoted by a unique identifier. This means that we shall effectively refer to the examples by their identifier x rather than use the associated set of extensional clauses, $p(x) \subset \mathcal{B}_E$. The instance space $\mathcal{X}$ is therefore a set of individuals contained in the overall universe of discourse $\mathcal{U}$. As in the statistical setting, we assume that a fixed and unknown distribution ρ is defined on $\mathcal{X} \times \mathcal{Y}$ and that training data $\mathcal{D}$ consist of input-output pairs (x_i, y_i) sampled identically and independently from ρ. Note that the latter assumption is reasonable in the case of relational domains with independent examples (such as mutagenesis) but not,

[1] Note that normalizations such as those of Equations (7) and (8) can give indefinite results iff one of the two arguments (say x) is the null vector of the feature space associated to the original kernel (i.e., K_R or K_{set}). In such a case, we will define $K_{norm}(x,z) = K_{mean}(x,z) = 0 \; \forall z \in \mathcal{X}, z \neq x$.



```
mutagenic(d26).
```

Extensional predicates

```
atm(d26,d26_1,c,22,-0.093).          bond(d26,d26_1,d26_2,7).
atm(d26,d26_2,c,22,-0.093).          bond(d26,d26_2,d26_3,7).
atm(d26,d26_3,c,22,-0.093).          bond(d26,d26_3,d26_4,7).
atm(d26,d26_4,c,22,-0.093).          bond(d26,d26_4,d26_5,7).
atm(d26,d26_5,c,22,-0.093).          bond(d26,d26_5,d26_6,7).
atm(d26,d26_6,c,22,-0.093).          bond(d26,d26_6,d26_1,7).
atm(d26,d26_7,h,3,0.167).            bond(d26,d26_1,d26_7,1).
atm(d26,d26_8,h,3,0.167).            bond(d26,d26_3,d26_8,1).
atm(d26,d26_9,h,3,0.167).            bond(d26,d26_6,d26_9,1).
atm(d26,d26_10,cl,93,-0.163).        bond(d26,d26_10,d26_5,1).
atm(d26,d26_11,n,38,0.836).          bond(d26,d26_4,d26_11,1).
atm(d26,d26_12,n,38,0.836).          bond(d26,d26_2,d26_12,1).
atm(d26,d26_13,o,40,-0.363).         bond(d26,d26_13,d26_11,2).
atm(d26,d26_14,o,40,-0.363).         bond(d26,d26_11,d26_14,2).
atm(d26,d26_15,o,40,-0.363).         bond(d26,d26_15,d26_12,2).
atm(d26,d26_16,o,40,-0.363).         bond(d26,d26_12,d26_16,2).
```

Intensional predicates

```
nitro(Drug,[Atom0,Atom1,Atom2,Atom3]) :-
   atm(Drug,Atom1,n,38,_),
   bondd(Drug,Atom0,Atom1,1),
   bondd(Drug,Atom1,Atom2,2),
   atm(Drug,Atom2,o,40,_),
   bondd(Drug,Atom1,Atom3,2),
   Atom3 @> Atom2,
   atm(Drug,Atom3,o,40,_).


bondd(Drug,Atom1,Atom2,Type) :-
   bond(Drug,Atom1,Atom2,Type).
bondd(Drug,Atom1,Atom2,Type) :-
   bond(Drug,Atom2,Atom1,Type).
```

```
benzene(Drug,Ring_list) :-
   atoms(Drug,6,Atom_list,[c,c,c,c,c,c]),
   ring6(Drug,Atom_list,Ring_list,[7,7,7,7,7,7]).

ring6(Drug,[Atom1|List],[Atom1,Atom2,Atom4,Atom6,Atom5,Atom3],
   [Type1,Type2,Type3,Type4,Type5,Type6]) :-
   bondd(Drug,Atom1,Atom2,Type1),   memberchk(Atom2,[Atom1|List]),
   bondd(Drug,Atom1,Atom3,Type2),   memberchk(Atom3,[Atom1|List]),
   Atom3 @> Atom2,
   bondd(Drug,Atom2,Atom4,Type3),   Atom4 \== Atom1,
   memberchk(Atom4,[Atom1|List]),
   bondd(Drug,Atom3,Atom5,Type4),   Atom5 \== Atom1,
   memberchk(Atom5,[Atom1|List]),
   bondd(Drug,Atom4,Atom6,Type5),   Atom6 \== Atom2,
   memberchk(Atom6,[Atom1|List]),
   bondd(Drug,Atom5,Atom6,Type6),   Atom6 \== Atom3.
```

Fig. 1. Example from the mutagenesis domain [47] illustrating the framework for statistical logic learning we use in this chapter

in general, when examples are linked by extensional predicates and collective prediction schemes are required (e.g. [49,50]).

In Figure 1 we exemplify our framework in the well known mutagenesis domain [47]. The extensional predicates are in this case `atm/5` and `bond/4`, describing the input portion of the data. The predicate `mutagenic/1` is also extensional. It describes the target class y and is not included in $\mathcal{B}$. The instance identifier in this case is `d26`, while $p($`d26`$)$, the extensional clauses associated with example `d26` $\in \mathcal{X}$, are listed in the upper right box of Figure 1. Intensional predicates include, among others, `nitro/2` and `benzene/2`, listed in the bottom box of Figure 1.

The output produced by statistical and ILP-based learning algorithms is also typically different. Rather than having to find a set of clauses that, added to the background theory, covers the examples, the main goal of a statistical learning algorithm is to find a function f that maps instances into their targets and whose general form is given by the representer theorem as in Eq. (2). Concerning the methods reviewed in this chapter, when the kernel function is fixed before learning, (as it happens in the methods presented in Sections 3, 4, and 5), predictions on new instances will be essentially opaque. However, when the kernel is learned together with the target function (see Section 6), the learning process also produces a collection of clauses, like an hypothesis in the ILP setting.

2.6 Types

A finer level of granularity in the definition of some of the logic-based kernels presented in this chapter can be gained from the use of typed terms. This extra flexibility may be necessary to specify different kernel functions associated with constants (e.g. to distinguish between numerical and categorical constants) or to different arguments of compound terms.

Following [51], we use a ranked set of type constructors $\mathcal{T}$, that contains at least the nullary constructor $\perp$. We allow polymorphism through type parameters. For example $list\alpha$ is a unary type constructor for the type of lists whose elements have type α. The arity of a type constructor is the number of type parameters it accepts. The set $\mathcal{T}$ is closed with respect to type variable substitution. Thus if $\tau\alpha_1, \ldots, \alpha_m \in \mathcal{T}$ is an m-ary type constructor (with type variables $\alpha_1, \ldots, \alpha_m$) and $\tau_1, \ldots, \tau_m \in \mathcal{T}$ then $\tau\tau_1, \ldots, \tau_m \in \mathcal{T}$.

The type signature of a function of arity n has the form $\tau_1\times, \ldots, \times\tau_n \mapsto \tau'$ where $n \geq 0$ is the number of arguments, $\tau_1, \ldots, \tau_k \in \mathcal{T}$ their types, and $\tau' \in \mathcal{T}$ the type of the result. Functions of arity 0 have signature $\perp \mapsto \tau'$ and can be therefore interpreted as constants of type τ'. The type signature of a predicate of arity n has the form $\tau_1\times, \ldots, \times\tau_n \mapsto \Omega$ where $\Omega \in \mathcal{T}$ is the type of booleans. We write $t : \tau$ to assert that t is a term of type τ.

A special case is when $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ is a partition of $\mathcal{U}$. In this case $\mathcal{T}$ can be viewed as an equivalence relation $=_T$ as follows: $\forall x, y \in \mathcal{U}\ x =_T y$ iff $\exists\tau_i \in \mathcal{T} s.t.(x : \tau_i \Leftrightarrow y : \tau_i)$. Another interesting situation is when type names are hierarchically organized in a partial order $\prec_T \subset \mathcal{T} \times \mathcal{T}$, with $\sigma \prec_T \tau$ meaning that σ *is a* τ (e.g. dog $\prec_T$ animal).

3 Kernels on Prolog Ground Terms

3.1 Motivations

We begin linking statistical and logic learning by introducing a family of kernels for Prolog terms. Convolution kernels over complex individuals have been recently defined using higher order logic abstractions [52]. The functions defined in this section can be seen as a specialization of such kernels to the case of Prolog and are motivated by the following considerations. First, Prolog and first-order logic representations provide a *simpler* representational framework than higher order logics. Second, Prolog expressiveness is *sufficient* for most application domains (for example, higher order structures such as sets can be simulated and types can also be introduced). Third, Prolog is a widespread and well supported language and many inductive logic programming systems and knowledge bases are actually based on (fragments of) first order logic. Finally, no probabilistic logic representations (like those thoroughly discussed elsewhere in this book) are yet available for higher-order logics.

The kernels introduced here have of course interesting connections to relational distances such as those described in [53,54]. It should be noted, however, that a distance function can trivially obtained from a kernel just by taking the

Euclidean distance in feature space, while a metric does not necessarily map into a Mercer kernel.

3.2 Untyped Terms

We begin with kernels on untyped terms. Let $\mathcal{C}$ be a set of constants and $\mathcal{F}$ a set of functors, and denote by $\mathcal{U}$ the corresponding Herbrand universe (the set of all ground terms that can be formed from constants in $\mathcal{C}$ and functors in $\mathcal{F}$). Let $f^{/n} \in \mathcal{F}$ denote a functor having name f and arity n. The kernel between two terms t and s is a function $K : \mathcal{U} \times \mathcal{U} \mapsto \mathbb{R}$ defined inductively as follows:

– if $s \in \mathcal{C}$ and $t \in \mathcal{C}$ then
$$K(s,t) = \kappa(s,t) \tag{10}$$
where $\kappa : \mathcal{C} \times \mathcal{C} \mapsto \mathbb{R}$ is a valid kernel on constants;

– else if s and t are compound terms and have different functors, i.e., $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$, then
$$K(s,t) = \iota(f^{/n}, g^{/m}) \tag{11}$$
where $\iota : \mathcal{F} \times \mathcal{F} \mapsto \mathbb{R}$ is a valid kernel on functors;

– else if s and t are compound terms and have the same functor, i.e., $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$, then
$$K(s,t) = \iota(f^{/n}, f^{/n}) + \sum_{i=1}^{n} K(s_i, t_i) \tag{12}$$

– in all other cases $K(s,t) = 0$.

Functions κ and ι are *atomic* kernels that operate on non-structured symbols. A special but useful case is the atomic exact match kernel δ defined in Eq. (5).

3.3 Typed Terms

The kernel between two typed terms t and s (see Section 2.6) is defined inductively as follows:

– if $s \in \mathcal{C}$, $t \in \mathcal{C}$, $s : \tau$, $t : \tau$ then
$$K(s,t) = \kappa_\tau(s,t) \tag{13}$$
where $\kappa_\tau : \mathcal{C} \times \mathcal{C} \mapsto \mathbb{R}$ is a valid kernel on constants of type τ;

– else if s and t are compound terms that have the same type but different functors or signatures, i.e., $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$, $s : \sigma_1 \times, \ldots, \times \sigma_n \mapsto \tau'$, $t : \tau_1 \times, \ldots, \times \tau_m \mapsto \tau'$, then
$$K(s,t) = \iota_{\tau'}(f^{/n}, g^{/m}) \tag{14}$$
where $\iota_{\tau'} : \mathcal{F} \times \mathcal{F} \mapsto \mathbb{R}$ is a valid kernel on functors that construct terms of type τ'

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

$$K(s,t) = \begin{cases} \kappa_{\tau_1 \times, \ldots, \times \tau_n \mapsto \tau'}(s,t) \\ \qquad\qquad \text{if } (\tau_1 \times, \ldots, \times \tau_n \mapsto \tau') \in \overline{\mathcal{T}} \\ \iota_{\tau'}(f^{/n}, f^{/n}) + \sum_{i=1}^{n} K(s_i, t_i) \quad \text{otherwise} \end{cases} \tag{15}$$

where $\overline{\mathcal{T}} \subset \mathcal{T}$ denotes a (possibly empty) set of *distinguished* type signatures that can be useful to specify ad-hoc kernel functions on certain compound terms, and $\kappa_{\tau_1 \times, \ldots, \times \tau_n \mapsto \tau'} : \mathcal{U} \times \mathcal{U} \mapsto I\!R$ is a valid kernel on terms having distinguished type signature $\tau_1 \times, \ldots, \times \tau_n \mapsto \tau' \in \overline{\mathcal{T}}$.

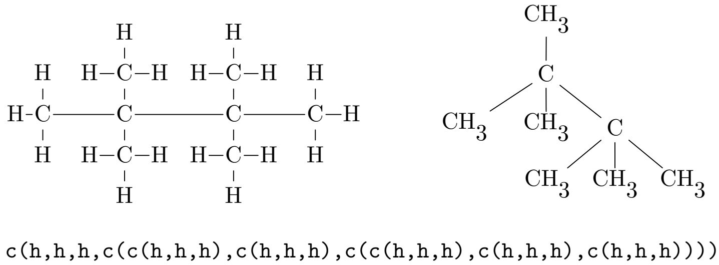
– in all other cases $K(s,t) = 0$.

Positive semi-definiteness of these kernels follows from their being special cases of decomposition kernels (see [55] for details). Variants where direct summations over sub-terms are replaced by tensor products are also possible.

3.4 A Guided Example: Alkanes

We demonstrate here the use of kernels over logical terms in a simple application of quantitative structure-property relationship (QSPR) consisting in the prediction of boiling point of alkanes [56]. Alkanes (except cycloalkanes, which are not considered here) are naturally represented as trees and a root can be chosen using a very simple procedure. The resulting rooted trees are encoded as Prolog ground terms. Figure 2 shows an example of molecule encoding, where we actually employed a reversed ordering of the children of each node with respect to the procedure described in [56], in order to have the backbone of the molecule on the right hand side of the tree.

We designed a kernel on untyped terms by using exact match for comparing functors (carbon atoms), and the null function for comparing constants (hydrogen atoms). The resulting kernel counts the number of carbon atoms in corresponding positions of two alkanes. As an additional source of information, we



```
c(h,h,h,c(c(h,h,h),c(h,h,h),c(c(h,h,h),c(h,h,h),c(h,h,h))))
```

Fig. 2. An alkane, its canonical representation as a rooted tree, and the corresponding Prolog ground term

extracted the depths of the trees representing the molecules, and summed their product to the term kernel, obtaining a more informed kernel K'. The resulting function was composed with a Gaussian kernel.

The above kernel was used in conjunction with ridge regression to solve the boiling point prediction problem. Performance was evaluated by a ten fold cross validation procedure, removing the methane compound from the test results as suggested in [56], being it an outlier with basically no structure. Hyperparameters (namely, the Gaussian width and the regularization parameter), were chosen by a hold-out procedure on the training set of the first fold, and kept fixed for the successive 10 fold cross validation procedure. When using kernel K we obtained an average mean square error of 4.6 Celsius degrees while using K' the error can be reduced to 3.8 degrees. These results are comparable to those produced by the highly tuned neural networks developed in [56].

4 Declarative Kernels

We present in this section a logical framework for kernel specification that provides a simple interface for the incorporation of background knowledge. The relational feature generation process is controlled by an additional set of facts and axioms, developed on the basis of the available background theory $\mathcal{B}$. Although, in general, any set of relational features could be used, we start from a specific setting in which these additional facts and axioms refer to special and germane relations for reasoning about *parts* and *places*.

4.1 Mereotopology

The parthood relation has been formally investigated by logicians and philosophers for almost a century since the early work of Leśniewski [57] followed by Leonard & Goodman's calculus of individuals [58]. The axiomatic theory of parts is referred to as *mereology* (from the Greek $\mu\epsilon\rho o\zeta$, "part"). It has obvious connections to decomposition of data structures in convolution kernels (see Section 2.3). The theory can be enriched with additional topological predicates and axioms aiming to describe wholeness. As pointed out by Varzi [59], topology is much needed because "mereological reasoning by itself cannot do justice to the notion of a whole (a one-piece, self-connected whole, such as a stone or a whistle, as opposed to a scattered entity made up of several disconnected parts, such as a broken glass, an archipelago, or the sum of two distinct cats)." These ideas can be also leveraged in machine learning to increase the kernel expressiveness with respect to pure decompositional approaches like the all-substructures kernels discussed in Section 2.3 that are only based on the notion of parts.

We formally introduce two special predicates: $\preceq_P$ and Connected, with the following intended meaning. For any two objects x and y, $x \preceq_P y$ declares x to be a part of y and Connected(x, y) declares x to be connected to y. Well-behaved definitions of parthood and connection should satisfy some given axiomatic structure [59]. In the context of knowledge representation, it is widely accepted that $\preceq_P$ should be a partial order, i.e. $\forall x, y, z \in \mathcal{U}$

$$x \preceq_P x \tag{P1}$$
$$x \preceq_P y \wedge y \preceq_P x \Rightarrow y =_P x \tag{P2}$$
$$x \preceq_P y \wedge y \preceq_P z \Rightarrow x \preceq_P z \tag{P3}$$

The theory defined by the above axioms is referred to as *ground mereology*. Interestingly, the above theory immediately provides us with a natural identity predicate $=_P$ that may be used as a basic elementary operator for comparing parts. Additional useful relations are supported by the theory, in particular

$$x \prec_P y \text{ iff } x \preceq_P y \wedge \neg y \preceq_P x \qquad \text{proper part} \tag{16}$$
$$\text{Overlap}(x, y) \text{ iff } \exists z.(z \preceq_P x \wedge z \preceq_P y) \qquad \text{overlap} \tag{17}$$
$$\text{Underlap}(x, y) \text{ iff } \exists z.(x \preceq_P z \wedge y \preceq_P z) \qquad \text{underlap} \tag{18}$$

The *supplementation* axiom, if added to the theory, supports the notion of extensionality:

$$\forall z.(z \preceq_P x \Rightarrow \text{Overlap}(z, y)) \Rightarrow x \preceq_P y. \tag{P4}$$

Following [59], the following axioms characterize topology and its link to mereology:

$$\text{Connected}(x, x) \tag{C1}$$
$$\text{Connected}(x, y) \Rightarrow \text{Connected}(y, x) \tag{C2}$$
$$x \preceq_P y \Rightarrow \forall z.(\text{Connected}(z, x) \Rightarrow \text{Connected}(z, y)) \tag{C3}$$

Additional useful relations are supported by the theory, in particular

$$\text{Externally\_Connected}(x, y) \text{ iff } \text{Connected}(x, y) \wedge \neg\text{Overlap}(x, y) \tag{19}$$

Mereotopology can be used to enrich the given background theory in the hope that it will generate further instances of the parthood and connection relations that will be *useful* for learning. It may also serve the purpose of checking the correctness of the declared parts and connections. When used for generating new instances of mereotopological relations, axioms should be used wisely to avoid an explosion of uninteresting parts and connections. Thus, depending on the application domain, axioms can be selectively omitted — for example (P4) will be typically avoided.

4.2 Mereotopological Relations

Several mereotopological relations (MR) can be introduced to characterize an instance x, for example:

i) The proper parts of x: $\mathcal{R}_P(x) = \{y : y \prec_P x\}$;
ii) The connected proper parts of x: $\mathcal{R}_C(x) = \{(y, z) : y \prec_P x \wedge z \prec_P x \wedge \text{Connected}(y, z)\}$;
iii) The overlapping parts in x, along with their common proper parts: $\mathcal{R}_I(x) = \{(y, z, w) : y \neq z \wedge y \prec_P x \wedge z \prec_P x \wedge w \prec_P y \wedge w \prec_P z\}$;

iv) The externally connected parts in x along with the associated linking terminals:

$$\mathcal{R}_L(x) = \{(y,z,u,v) : z \prec_P x \wedge y \prec_P x \wedge \neg\text{Overlap}(z,y) \wedge u \prec_P z \wedge v \prec_P y \\ \wedge\text{Connected}(u,v)\}.$$

Additional MRs can be defined if necessary. We denote by $\mathcal{M}$ the set of declared MRs. As detailed below, a declarative kernel compares two instances by comparing the corresponding MRs, so adding relations to $\mathcal{M}$ plays a crucial role in shaping the feature space.

4.3 The Contribution of Parts

The kernel on parts, denoted K_P, is naturally defined as the *set kernel* between the sets of proper parts:

$$K_P(x,x') = \sum_{y\in\mathcal{P}(x)} \sum_{y'\in\mathcal{P}(x')} k_P(y,y') \tag{20}$$

where k_P denotes a kernel function on parts, defined recursively using K_P. Types (see Section 2.6) can be used to fine-tune the definition of k_P. It types can be viewed as an equivalence relation ($\mathcal{T}$ is a partition of $\mathcal{U}$), then

$$k_P(y,y') = \begin{cases} \iota(y,y') & \text{if } y =_T y' \text{ and } y,y' \text{ are atomic objects;} \\ K_P(y,y') + \iota_(y,y') & \text{if } y =_T y' \text{ and } y,y' \text{ are non atomic objects;} \\ 0 & \text{otherwise (i.e. } y \neq_T y'). \end{cases} \tag{21}$$

In the above definition, $\iota(y,y')$ is a kernel function that depends on properties or attributes of y and y' (not on their parts).

If types are hierarchically organized, then the test for type equality in Eq. (21) can be replaced by a more relaxed test on type compatibility. In particular, if $y:\tau$, $y':\tau'$ and there exists a least general supertype $\sigma : \tau \prec_T \sigma, \tau' \prec_T \sigma$, then we may type cast y and y' to σ and evaluate κ on the generalized objects $\sigma(y)$ and $\sigma(y')$. In this case the function $\iota(y,y')$ depends only on properties or attributes that are common to y and y' (i.e. those that characterize the type σ).

4.4 The Contribution of Other MRs

The kernel on connected parts compares the sets of objects $\mathcal{R}_C(x)$ and $\mathcal{R}_C(x')$ as follows:

$$K_C(x,x') = \sum_{(y,z)\in\mathcal{R}_C(x)} \sum_{(y',z')\in\mathcal{R}_C(x')} K_P(y,y') \cdot K_P(z,z'). \tag{22}$$

The kernel on overlapping parts compares the sets of objects $\mathcal{R}_I(x)$ and $\mathcal{R}_I(x')$ as follows:

$$K_I(x,x') = \sum_{(y,z,w)\in\mathcal{R}_I(x)} \sum_{(y',z',w')\in\mathcal{R}_L(x')} K_P(w,w')\delta(y,y')\delta(z,z') \tag{23}$$

where $\delta(x,y) = 1$ if x and y have the same type and 0 otherwise. The kernel $K_L(x,x')$ on externally connected parts is defined in a similar way:

$$K_L(x,x') = \sum_{(y,z,u,v)\in\mathcal{R}_L(x)} \sum_{(y',z',u',v')\in\mathcal{R}_L(x')} K_P(u,u')K_P(v,v')\delta(y,y')\delta(z,z'). \tag{24}$$

4.5 The General Case

Given a set $\mathcal{M}$ of MRs (such as those defined above), the final form of the kernel is

$$K(x,x') = \sum_{M\in\mathcal{M}} K_M(x,x'). \tag{25}$$

Alternatively, a convolution-type form of the kernel can be defined as

$$K(x,x') = \prod_{M\in\mathcal{M}} K_M(x,x'). \tag{26}$$

To equalize the contributions due to different MRs, the kernels K_M can be normalized before combining them with sum or product. Positive semi-definiteness follows, as in the case of convolution kernels, from the closeness with respect to direct sum and tensor product operators [43].

4.6 Remarks

The kernel of Eq. (25) could have been obtained also without the support of logic programming. However, deductive reasoning greatly simplifies the task of recognizing parts and connected parts and at the same time, the declarative style of programming makes it easy and natural to define the features that are implicitly defined by the kernel.

Declarative kernels and Haussler's convolution kernels [43] are intimately related. However the concept of *parts* in [43] is very broad and does not necessarily satisfy mereological assumptions.

4.7 A Guided Example: Mutagenesis

Defining and applying declarative kernels involves a three-step process: (1) collect data and background knowledge; (2) interface mereotopology to the available data and knowledge; (3) calculate the kernel on pairs of examples. We illustrate the process in the mutagenesis domain. The first step in this case simply consists of acquiring the atom-bond data and the ring theory developed by Srinivasan *et al.* [47], that comes in the usual form described in Figure 1. The second step consists of interfacing the available data and knowledge to the kernel. For this purpose, we first need to provide a set of declarations for types, objects, and basic instances of mereotopological relations. Objects are declared using the predicate `obj(X,T)` meaning that `X` is an object of type `T`. For example types include atoms and functional groups (see Figure 3a).

a
```
type(instance).
type(atm).
type(benzene).

obj(X,atm) :-
        atm(Drug,X,_,_,_).
obj(X,benzene) :-
        benzene(Drug,X).
```

b
```
partof(X,X) :-          % P1 axiom
        obj(X,_SomeType).
equalp(X,Y) :-          % P2 axiom
        partof(X,Y), partof(Y,X).
partof(X,Y) :-          % P3 axiom (base)
        has_part(X,Y).
partof(X,Y) :-          % P3 axiom (induction)
        has_part(X,Z), partof(Z,Y).
ppartof(X,Y) :-         % (proper part)
     partof(X,Y), \+ partof(Y,X).
ppartsof(Parts,Y) :-     % MR i)
     setof(X,ppartof(X,Y),Parts).
```

c
```
has_part(B,Drug) :-
        obj(Drug,instance),
        benzene(Drug,B).
```

Fig. 3. Code fragments for the guided example (see text)

Then we declare basic proper parts via the predicate `has_part(X,Y)` that is true when `Y` is known to be a proper part of `X`. For example if an instance D (a molecule in this case) contains a benzene ring B, then $B \prec_P D$ (Figure 3b).

Note that the use of a predicate called `has_part` (rather than `partof`) is necessary to avoid calling a recursive predicate in the Prolog implementation. The third step is independent of the domain. To calculate the kernel, we first make use of mereotopology to construct the MRs associated with each instance (for example, the code for computing proper parts is shown in Figure 3c). The resulting sets of ground facts are then passed to a modified version of SVM^{light} [60] for fast kernel calculation.

We can construct here an example where connected parts may produce interesting features. Let us denote by x the molecule, by y the benzene ring, by v the nitro group consisting of atoms `d26_11`, `d26_13`, and `d26_14`, and by w the nitro group consisting of atoms `d26_12`, `d26_15`, and `d26_16`. Then $(y, v, \texttt{d26\_4}, \texttt{d26\_11}) \in \mathcal{R}_L(x)$ and $(y, w, \texttt{d26\_2}, \texttt{d26\_12}) \in \mathcal{R}_L(x)$.

To show how learning takes place in this domain, we run a series of 10-fold cross-validation experiments on the regression friendly data set of 188 compounds. First, we applied a mature ILP technique constructing an ensemble of

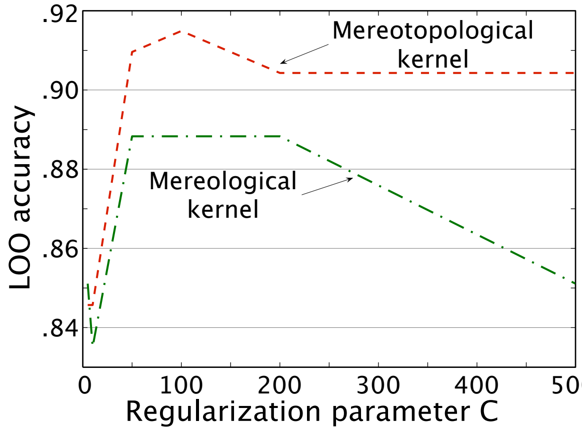


Fig. 4. LOO accuracy on the regression friendly mutagenesis data set

25 Aleph theories [61]. Aleph parameters *search*, *evalfn*, *clauselength* and *nodes* were set to be *bf*, *coverage*, 4 and 20000 respectively. The two tunable parameters *minacc* and *voting threshold* were selected by applying 3-fold cross validation in the training set of the first fold. Voting threshold ranges from 1 to the size of the ensemble and the set of values for minacc are given by $\{0.75, 0.9\}$. We obtained accuracy .88 $\pm$ .07 using atom-bond data and .89 $\pm$ .05 by adding the background ring theory. Next we applied declarative kernels with support vector machines (SVM), obtaining accuracy .90 $\pm$ .07. CPU time was of the order of minutes for the declarative kernel and days for the Aleph ensemble. Finally, we compared the expressive power of ground mereological relations with that of the full mereotopological theory. Figure 4 reports LOO accuracy for different values of the regularization parameter C, for both mereological and mereotopological kernels, showing the latter achieves both better optimal accuracy and more stable performances.

5 Kernels on Prolog Proof Trees

The main idea behind this family of kernels is the exploitation of program traces to define the kernel function. Traces have been extensively used in ILP and program synthesis (e.g. [62,63,64,65,66]). Kernels on Prolog proof trees are based on a new framework for learning from example-traces. The main assumption is that we are given a target program (called the *visitor*), that reflects background knowledge and that takes single examples as its input. The task consists of learning from the training set of traces obtained by executing the visitor program on each example. Hence, the statistical learning algorithm will employ a kernel on program traces rather than using directly a kernel on examples. The visitor acts therefore as a knowledge-based mediator between the data and the statistical learning algorithm. The bottom line is that similar instances should produce similar traces when probed with programs that express background knowledge and examine characteristics they have in common. These characteristics can be more general than parts. Hence, trace kernels can be introduced with the aim of achieving a greater generality and flexibility with respect to various decomposition kernels (including declarative kernels). These ideas will be developed in detail for logic programs, although nothing prevents, in principle, to use them in the context of different programming paradigms and in conjunction with alternative models of computation such as finite state automata or Turing machines.

Formally, a *visitor* program for a background theory $\mathcal{B}$ and domain $\mathcal{X}$ is a set $\mathcal{V}$ of definite clauses that contains at least one special clause (called a *visitor*) of the form $V \leftarrow B_1, \ldots, B_N$ and such that

– V is a predicate of arity 1
– for each $j = 1, \ldots, N$, B_j is declared in $\mathcal{B} \cup \mathcal{V}$;

Intuitively, if `visit/1` is a visitor in $\mathcal{V}$, by answering the query `visit(ex)?` we explore the features of the instance whose constant identifier `ex` is passed to the visitor. Having multiple visitors in the program $\mathcal{V}$ allows us to explore different aspects of the examples and include multiple sources of information.

The visitor clauses should be designed to "inspect" examples using other predicates declared in $\mathcal{B}$, keeping in mind that the similarity between two examples is the similarity between the execution traces of visitors. Thus, we are not only simply interested in determining whether certain clauses succeed or fail on a particular example, but rather to ensure that visitors will construct useful features during their execution. This is a major difference with respect to other approaches in which features are explicitly constructed by computing the truth value for predicates [67].

The learning setting can be briefly sketched as follows. The learner is given a data set $\mathcal{D} = \{(x_1, y_1), \ldots, (x_m, y_m)\}$, background knowledge $\mathcal{B}$, and a visitor program $\mathcal{V}$. For each instance x_i, a trace T_{x_i} (see Eq. 28) is obtained by running the visitor program. A kernel machine (e.g., an SVM) is then trained to form the function $f : \mathcal{X} \mapsto \mathcal{Y}$ defined as

$$f(x) = \sum_{i=1}^{m} c_i K(T_{x_i}, T_x).$$

In the following, we give some details about the definition of traces and kernels between traces.

5.1 Traces and Proof Trees

In order to record a trace, we should store all steps in the proofs of a given visitor goal called on a given example. We may think that SLD-trees are a rather obvious representation of proofs when using Prolog. A path in an SLD-tree is indeed an execution sequence of the Prolog interpreter. Unfortunately, SLD-trees are too complex for our purposes, containing too many details and prone to generate irrelevant features such as those associated with failed paths. In order to obtain simple and still useful traces we prefer proof trees (see e.g. [68]). Given a program $\mathcal{P}$ and a goal G, the proof tree for G is empty if $\mathcal{P} \not\models G$ or, otherwise, it is a tree t recursively defined as follows:

- if there is a fact f in $\mathcal{P}$ and a substitution θ such that $G\theta = f\theta$, then $G\theta$ is a leaf of t.
- otherwise there must be a clause $H \leftarrow B_1, ..., B_n \in \mathcal{P}$ and a substitution θ' such that $H\theta' = G\theta'$ and $\mathcal{P} \models B_j\theta'$ $\forall j$, $G\theta'$ is the root of t and there is a subtree of t for each $B_j\theta'$ that is a proof tree for $B_j\theta'$.

A second aspect is that we would like to deal with ground traces in order to simplify the definition of the kernel. On the other hand, proof trees may contain free variables. There are at least three ways of ensuring that proof trees are ground: first, we can use skolemization (naming existentially quantified variables with a specific constant symbol). A second option is to require that all clauses be range-restricted. Finally, we can make specific assumptions about the mode of head variables not occurring in the body, ensuring that these variables will be instantiated when proving the goal.

Goals can be satisfied in multiple ways, thus each query generates a (possibly empty) forest of proof trees. Since multiple visitors may be available, the trace of an instance is actually a *tuple of proof forests*. Formally, let N be the number of visitors in $\mathcal{V}$ and for each $l = 1, \ldots, N$ let $T_{lj,x}$ denote the proof tree that represents the j-th proof of the goal $V_l(x)$, i.e., a proof that $\mathcal{B} \cup \mathcal{V} \models V_l(x)$. Let

$$T_{l,x} = \{T_{l1,x}, \ldots, T_{ls_{l,x},x}\} \tag{27}$$

where $s_{l,x} \geq 0$ is the number of alternative proofs of goal $V_l(x)$. The trace of an instance x is then defined as the tuple

$$T_x = [T_{1,x}, \ldots, T_{N,x}]. \tag{28}$$

A proof tree can be pruned to remove unnecessary details and reduce the complexity of the feature space. Let us explain this concept with an example based on mutagenesis (see Figure 1). In this domain, it may be useful to define visitors that explore groups such as benzene rings:

```
atoms(X,[]).                  visit_benzene(X):-
atoms(X,[H|T]):-                  benzene(X,Atoms),
    atm(X,H,_,_,_),               atoms(X,Atoms).
    atoms(X,T).
```

If we believe that the presence of the ring and the nature of the involved atoms represent a sufficient set of features, we may want to ignore details about the proof of the predicate `benzene` by pruning the corresponding proof subtree. This can be accomplished by including the following fact in the visitor program:

```
leaf(benzene(_,_)).
```

5.2 Kernels on Traces

A kernel over program traces can be defined in a top-down fashion. First, let us decompose traces into parts associated with different visitors (i.e., the elements of the tuple in Eq. (28)). The direct sum decomposition kernel of Eq. (4) applied to these parts yields:

$$K(T_x, T_z) = \sum_{l=1}^{N} K_l(T_{l,x}, T_{l,z}). \tag{29}$$

We always compare proofs of the same visitor since there is a unique decomposition of T_x and T_z. By definition of trace (see Eq. (28)), $T_{l,x}$ and $T_{l,z}$, $l = 1, \ldots, N$, are proof forests. Hence, the set kernel of Eq. (6) yields:

$$K_l(T_{l,x}, T_{l,z}) = \sum_{p=1}^{s_{l,x}} \sum_{q=1}^{s_{l,z}} K_{tree}(T_{lp,x}, T_{lq,z}). \tag{30}$$

We now need to introduce a kernel K_{tree} over individual proof trees. In principle, existing tree kernels (e.g. [7,8]) could be used for this purpose. However, we suggest here representing proof trees as typed Prolog ground terms. This option allows us to provide a fine-grained definition of kernel by exploiting type information on constants and functors (so that each object type can be compared by its own sub-kernel). Moreover, the kernel on ground terms introduced in Section 3 is able to compares sub-proofs only if they are reached as a result of similar inference steps. This distinction would be difficult to implement with traditional tree kernels. A ground term can be readily constructed from a proof tree as follows:

– Base step: if a node contains a fact, this is already a ground term.
– Induction: if a node contains a clause, then let n be the number of arguments in the head and m the number of atoms in the body (corresponding to the m children of the node). A ground compound term t having $n+1$ arguments is then formed as follows:
  - the functor name of t is the functor name of the head of the clause;
  - the first n arguments of t are the arguments of the clause head;
  - the last argument of t is a compound term whose functor name is a Prolog constant obtained from the clause number[2], and whose m arguments are the ground term representations of the m children of the node.

At the highest level of kernel between visitor programs (Eq. (29)), it is advisable to employ a feature space normalization using Eq. (7). In some cases it may also be useful to normalize lower-level kernels, in order to rebalance contributions of individual parts. In particular, the mean normalization of Eq. (8) can be applied to the kernel over individual visitors (Eq. (30)) and it is also possible to normalize kernels between individual proof trees, in order to reduce the influence of the proof size. Of course it is easy to gain additional expressiveness by defining specific kernels on proof trees that originate from different visitors.

In order to employ kernels on typed terms (see Section 3), we need a typed syntax for representing proof trees as ground terms. Constants can be of two main types: `num` (numerical) and `cat` (categorical). Types for compounds terms include `fact` (leaves) `clause` (internal nodes), and `body` (containing the body of a clause).

A number of special cases of kernels can be implemented in this framework. The simplest kernel is based on proof equivalence (two proofs being equivalent if the same sequence of clauses is proven in the two cases, and the head arguments in corresponding clauses satisfy a given equivalence relation): $K_{equiv}(s,t) = 1$ iff $s \equiv t$.

The *functor equality* kernel can be used when we want to ignore the arguments in the head of a clause. Given two ground terms $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$, it is defined as:

$$K_f(s,t) = \begin{cases} 0 & \text{if } type(s) \neq type(t) \\ \delta(f^{/n}, g^{/m}) & \text{if } s,t : \texttt{fact} \\ \delta(f^{/n}, g^{/m}) \star K(s_n, t_m) & \text{if } s,t : \texttt{clause} \\ K(s,t) & \text{if } s,t : \texttt{body} \end{cases} \tag{31}$$

[2] Since numbers cannot be used as functor names, this constant can be simply obtained by prefixing the clause number by '`cbody`'.

where K is a kernel on ground terms and the operator $\star$ can be either sum or product. Note that if s and t represent clauses (i.e., internal nodes of the proof tree), the comparison skips clause head arguments, represented by the first $n-1$ (resp. $m-1$) arguments of the terms, and compares the bodies (the last argument) thus proceeding on the children of the nodes. This kernel allows to define a non trivial equivalence between proofs (or parts of them) checking which clauses are proved in sequence and ignoring the specific values of their head arguments.

5.3 A Guided Example: Bongard Problems

One nice example showing the potential of learning from program traces is a very simple Bongard problem [69] in which the goal is to classify two-dimensional scenes consisting of sets of nested polygons (triangles, rectangles, and circles). In particular, we focus on the target concept defined by the pattern *triangle-*X^n*-triangle* for a given n, meaning that a positive example is a scene containing two triangles nested into one another with exactly n objects (possibly triangles) in between. Figure 5 shows a pair of examples of such scenes with their representation as Prolog facts and their classification according to the pattern for $n = 1$.

A possible example of background knowledge introduces the concepts of *nesting* in containment and *polygon* as a generic object, and can be represented by the following intensional predicates:

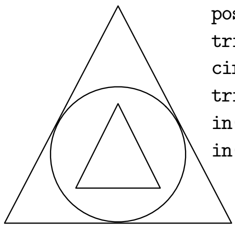
```
inside(X,A,B):- in(X,A,B).                          % clause nr 1
inside(X,A,B):-                                     % clause nr 2
    in(X,A,C),
    inside(X,C,B).
polygon(X,A) :-  triangle(X,A).                     % clause nr 3
polygon(X,A) :-  rectangle(X,A).                    % clause nr 4
polygon(X,A) :-  circle(X,A).                       % clause nr 5
```

A visitor exploiting such background knowledge, and having hints on the target concept, could be looking for two polygons contained one into the other. This can be represented as:

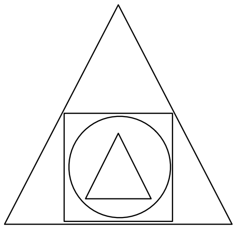
```
visit(X):-                                          % clause nr 6
    inside(X,A,B),polygon(X,A),polygon(X,B).
```

Figure 6 shows the proofs trees obtained running such a visitor on the first Bongard problem in Figure 5. A very simple kernel can be employed to solve such a task, namely an equivalence kernel with functor equality for nodewise comparison. For any value of n, such a kernel maps the examples into a feature space where there is a single feature discriminating between positive and negative examples, while the simple use of ground facts without intensional background knowledge would not provide sufficient information for the task.

The data set was generated by creating m scenes each containing a series of ℓ randomly chosen objects nested one into the other, and repeating the procedure



```
positive(bong1).
triangle(bong1,o1).
circle(bong1,o2).
triangle(bong1,o3).
in(bong1,o1,o2).
in(bong1,o2,o3).
```



```
negative(bong4).
triangle(bong4,o1).
rectangle(bong4,o2).
circle(bong4,o3).
triangle(bong4,o4).
in(bong4,o1,o2).
in(bong4,o2,o3).
in(bong4,o3,o4).
```

Fig. 5. Graphical and Prolog facts representation of two Bongard scenes. The left and right examples are positive and negative, respectively, according to the pattern *triangle-X-triangle*.

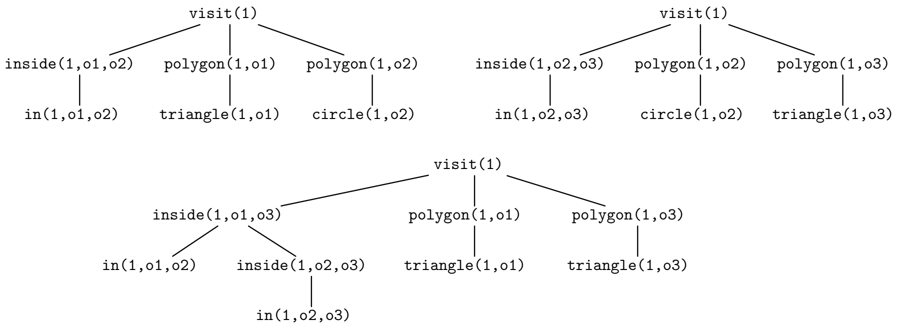


Fig. 6. Proof trees obtained by running the visitor on the first Bongard problem in Figure 5

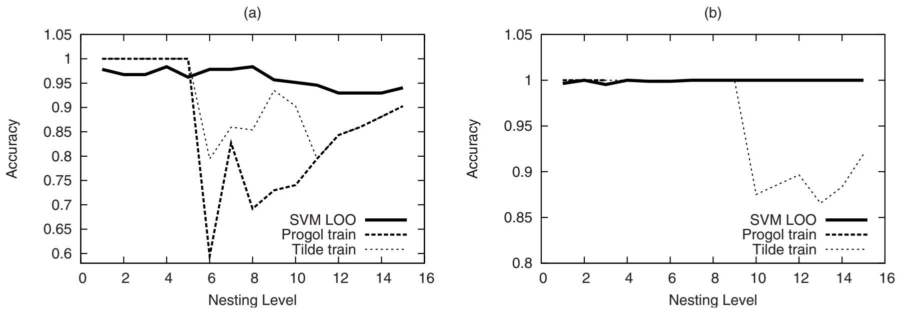


Fig. 7. Comparison between SVM leave-one-out error, Progol and Tilde empirical error in learning the *triangle-X^n-triangle* for different values of n, for data sets corresponding to $m = 10$ (a) and $m = 50$ (b)

for ℓ varying from 2 to 20. Moreover, we generated two different data sets by choosing $m = 10$ and $m = 50$ respectively. Finally, for each data set we obtained 15 experimental settings denoted by $n \in [0, 14]$. In each setting, positive examples were scenes containing the pattern *triangle-X^n-triangle*. We run an SVM with the above mentioned proof tree kernel and a fixed value $C = 10$ for the regularization parameter, on the basis that the data set is noise free. We evaluated its performance with a leave-one-out (LOO) procedure, and compared it to the empirical error of Tilde and Progol trained on the same data and background knowledge (including the visitor). Here we focus on showing that ILP algorithms have troubles finding a consistent hypothesis for this problem, hence we did not measure their generalization.

Figure 7(a) plots results for $m = 10$. Both Tilde and Progol stopped learning the concept for $n > 4$. Progol found the trivial empty hypothesis for all $n > 4$ apart from $n = 6$, and Tilde for all $n > 9$. While never learning the concept with 100% generalization accuracy, the SVM performance was much more stable when increasing the nesting level corresponding to positive examples. Figure 7(b) plots results for $m = 50$. Progol was extremely expensive to train with respect to the other methods. It successfully learned the concept for $n \leq 2$, but we stopped training for $n = 3$ after more than one week training time on a 3.20 GHz PENTIUM IV. Tilde stopped learning the concept for $n > 8$, and found the trivial empty hypothesis for $n > 12$. Conversely, the SVM was almost always able to learn the concept with 100% generalization accuracy, regardless of its complexity level.

Note that in order for the ILP algorithms to learn the target concept regardless of the nesting level, it would be necessary to provide a more informed `inside` predicate, which explicitly contains such nesting level as one of its arguments. The ability of the kernel to extract information from the predicate proof, on the other hand, allows our method to be employed when only partial background knowledge is available, which is typically the case in real world applications.

6 kFOIL

The above approaches for combining ILP and kernel can be expected to be highly effective from several points of view, in particular stability (i.e. robustness to noise), uniformity (i.e. classification and regression tasks can be handled in a uniform way) and expressivity (a rich hypothesis space is explored in domains consisting of independent relational objects). However, the function determined by these methods as a solution to the supervised learning problem is opaque, i.e. does not provide human-readable insights. In addition, although the feature space is rich, its definition must be specified *before* learning takes place. The idea behind kFOIL is radically different from this point of view. Unlike previous approaches, the feature space in kFOIL is *dynamically* constructed during learning (using a FOIL-like [38] covering algorithm) and can be effectively seen as an additional output of the learning problem (besides the prediction function). In this sense, kFOIL is similar to a recently introduced probabilistic ILP algorithm, nFOIL, that combines Naive Bayes and FOIL [70]. While nFOIL takes the generative direction of modeling, kFOIL is based on regularized empirical risk minimization (e.g. support vector machine learning). kFOIL preserves all the advantages of previously introduced kernels, in particular uniformity of representation across different supervised learning tasks, stability and robustness with respect to noise, expressivity of the representation language, and ability to reuse declarative background knowledge. The strength of kFOIL is its ability to provide additional explanations about the domain that can be read in the set of constructed clauses. However, since FOIL is used as an internal subroutine, the efficiency of other kernel based learning approaches cannot be preserved.

6.1 The Feature Space of kFOIL

In the kFOIL setting, the output of the learning process consists of both a prediction function f (as in Eq. (2)) and a kernel function K between examples. Each example $x \in \mathcal{X}$ is a first-order individual and $p(x)$ denotes the associated extensional clauses, as explained in Section 2.5. The function K is represented by means of a collection of clauses

$$\mathcal{H} = \{c_1, \dots, c_n\}$$

that play the same role of a hypothesis in the learning from entailment ILP setting. In particular, the feature space associated with K consists of Boolean vectors, indexed by clauses in the current hypothesis $\mathcal{H}$. Formally, the feature space representation can be written as $\phi_{\mathcal{H}}(x) = \phi_{\mathcal{H},1}(x), \dots, \phi_{\mathcal{H},n}(x)$ where

$$\phi_{\mathcal{H},i}(x) = \begin{cases} 1 \text{ if } \mathcal{B}_I \cup \{c_i\} \models p(x) \\ 0 \text{ otherwise} \end{cases}$$

The feature space representation is defined by the clauses in the current hypothesis and each feature simply check whether $p(x)$ is logically entailed by background knowledge and one given clause.

In this way, the kernel between two examples x and x' is simply the number of clauses firing on both examples, in the context of the given background knowledge:

$$K_{\mathcal{H}}(x, x') = \#ent_{\mathcal{H}}(p(x) \wedge p(x')) \tag{32}$$

where $\#ent_{\mathcal{H}}(a) = |\{c \in \mathcal{H} | \mathcal{B}_I \cup \{c\} \models a\}|$. The prediction function f has the standard form of Eq. (2), using $K_{\mathcal{H}}$ as kernel.

6.2 The kFOIL Learning Algorithm

The hypothesis $\mathcal{H}$ is induced by a modified version of the well-known FOIL algorithm [38], which essentially implements a *separate-and-conquer* rule learning algorithm in a relational setting.

```
kFOIL($\mathcal{D}, \mathcal{B}, \epsilon$)
1 $\mathcal{H} := \emptyset$
2 repeat
3         $c :=$ "$pos(x) \leftarrow$"
4         repeat
5                 $c := \arg\max_{c' \in \rho(c)}$ Score($\mathcal{D}, \mathcal{H} \cup \{c'\}, \mathcal{B}$)
6           until stopping criterion
7         $\mathcal{H} := \mathcal{H} \cup \{c\}$
8     until score improvement is smaller than $\epsilon$
9 return $\mathcal{H}$
```

The kFOIL algorithm, sketched in the above pseudo-code, is similar to the general FOIL algorithm. It repeatedly searches for clauses that score well with respect to the data set $\mathcal{D}$ and the current hypothesis $\mathcal{H}$ and adds them to the current hypothesis. The most general clause which succeeds on all examples is "$pos(x) \leftarrow$" where *pos* is the predicate being learned. The "best" clause c is found in the inner loop according to a general-to-specific hill-climbing search strategy, using a refinement operator $\rho(c)$ that generates the set of all possible refinements of clause c. In the case of kFOIL, each refinement is obtained by simply adding a literal to the right-hand side of c. Different choices for the scoring function Score have been used with FOIL. The scoring function of kFOIL is computed by wrapping around a kernel machine (such as an SVM). Specifically, Score($\mathcal{D}, \mathcal{H}, \mathcal{B}$) is computed by training a kernel machine on $\mathcal{D}$ and measuring the empirical risk

$$\text{Score}(\mathcal{D}, \mathcal{H}, \mathcal{B}) = \sum_{(x_i, y_i) \in \mathcal{D}} V(y_i, f(x_i))$$

being V a suitable loss function (that depends on the specific kernel machine, see Eq. (1) and following). kFOIL is stopped when the score improvement between two consecutive iterations falls below a given threshold ϵ. This a smoothed version of FOIL's criterion which is stopped when no clause can be found that cover additional positive examples. Finally, note that the data set size is reduced at each iteration of FOIL by removing examples that are already covered. However, this step is omitted from kFOIL as the kernel machine needs to be retrained (with a different kernel) on the entire data set.

In the case of kFOIL, a significant speedup can be obtained by working explicitly in a sparse feature space, rather than evaluating the kernel function according to its definition. This is because, especially at the early iterations, many examples are mapped to the same point in feature space and can be merged in a single point (multiplying the corresponding contribution to the loss function by the number of collisions).

6.3 A Guided Example: Biodegradability

In order to apply kFOIL to a certain learning task, three steps have to be accomplished: (1) collect data and background knowledge; (2) write the inductive bias

a
```
nitro(mol30, [atom1,atom2,atom3], [atom4]).
methyl(mol32, [atom1,atom2,atom3.atom4], [atom5]).bond(mol1,atom1,atom2,1).
atm(mol1,atom1,h,0,0).
logP(mol1, 0.35).
mweight(mol1,0.258).
```

b
```
gt(X,Y):- X > Y.
lt(X,Y):- X < Y.

num(N):-
  member(N,[0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]).
sbond(Mol,Atom1,Atom2,Bondtype):-
  bond(Mol,Atom1,Atom2,Bondtype);bond(Mol,Atom2,Atom1,Bondtype).
```

c
```
rmode(lt(+,N)):-numrmode(lt(+,N)):-num(N).
type(nitro(compound,struct1,struct2)).
type(lt(number,number)).
rmode(atom(+,+-,c)).
rmode(nitro(+,-,-)).
```

Fig. 8. Code fragments for the kFOIL guided example on biodegradability

that will determine all possible refinements of clauses; (3) run kFOIL. We will show an example of such process on a real world task concerning biodegradability of molecules. Degradation is the process by which chemicals are transformed into components which are not considered pollutants. A number of different pathways are responsible for such process, depending on environmental conditions. Blockeel *et al.* [71] conducted a study focused on aqueous biodegradation under aerobic conditions. Low and high estimates of half life time degradation rate were collected for 328 commercial chemical compounds. In this application domain, one is interested in the half-life time of the biodegradation process. The regression task consists in predicting the natural logarithm of the arithmetic mean of the low and high estimate for a given molecule. Available data include the atom/bond representation of molecules as well as global physico-chemical properties such as weight and logP. Rings and functional groups within a molecule are also represented as facts[3], where each groups is described by its constituent atoms as well as the atoms connecting it to the rest of the molecule. Figure 8(a) shows extracts of such data[4]. Additional background knowledge (see Figure 8(b) for an extract) includes comparison operators between numbers (`lt`, `gt`) and the set of allowed numerical values (`num`) as well as a predicate (`sbond`) defining symmetric bonds. The second step consists of writing the configuration file for the FOIL part of the algorithm, as a combination of type and mode declarations. Figure 8(c) contains an extract of such configuration. The final step consists of running kFOIL in regression mode providing as inputs data, background knowledge and configuration files. Note that the first two steps are independent of the type of task to be learned (e.g. binary classification or regression), which will only influence the type of kernel machine to be employed in computing the score of a given clause and in producing the output for a test example (e.g. SVM or Support Vector Regression).

[3] Intensional predicates representing functional groups were saturated on the examples in this dataset, thus generating extensional predicates.

[4] Note that we are not considering facts representing counts of groups and small substructures, which were also included in [71], as they slightly degrade performances for all methods in almost all cases.

Table 1. Result on the Biodegradability dataset. The results for Tilde and S-CART have been taken from [71]. 5 runs of 10 fold cross-validation have been performed, on the same splits into training and test set as used in [71]. We report both Pearson correlation and RMSE as evaluation measures. • indicates that the result for kFOIL is significantly better than for other method (unpaired two-sided t-test, p = 0.05).

Evaluation measure	**kFOIL**	**Tilde**	**S-CART**
Correlation	0.609 ± 0.047	0.616 ± 0.021	0.605 ± 0.023
Root Mean Squared Error	1.196 ± 0.023	1.265 ± 0.033•	1.290 ± 0.038•

Table 1 shows the regression performance of kFOIL on the Biodegradability dataset, as compared to the results reported in [71] for Tilde and S-CART. As our aim here is showing that kFOIL is competitive to other state-of-the-art techniques, and not to boost performance, we did not try to specifically optimize any parameter. We thus used default settings for the FOIL parameters: maximum number of clauses in a hypothesis was set to 25, maximum number of literals in a clause to 10 and the threshold for the stopping criterion to 0.1%. However, we performed a beam search with beam size 5 instead of simple greedy search. The kernel machine employed was support vector regression, with regularization constant $C = 0.01$ and ϵ tube parameter set to 0.001. A polynomial kernel of degree 2 was used on top of the kernel induced by the learned clauses. The results obtained show that kFOIL is competitive with the first-order decision tree systems S-CART and Tilde at maximizing correlation, and slightly superior at minimizing RMSE.

7 Applications

7.1 Declarative Kernels for Information Extraction

In these experiments we apply declarative kernels to the extraction of relational information from free text. Specifically, we focus on multi-slot extraction of binary relations between candidate named entities. Our experiments were carried out on the yeast protein localization data set described in [72] and subsequently used as a testbed for state-of-the-art methods based on ILP [73]. The task consists of learning the relation `protein_location` between two named entities representing *candidate* protein names and cell locations. Instances are ordered pairs of noun phrases (NP) extracted from MEDLINE abstracts and with stemmed words. An instance is positive iff the first NP is a protein and the second NP is a location, for example:

```
protein_location("the mud2 gene product","earli spliceosom assembl",pos).
protein_location("sco1", "the inner mitochondri membran",pos).
protein_location("the ept1 gene product","membran topographi",pos).
protein_location("a reductas activ", "the cell", neg).
protein_location("the ace2 gene", "multipl copi", neg).
```

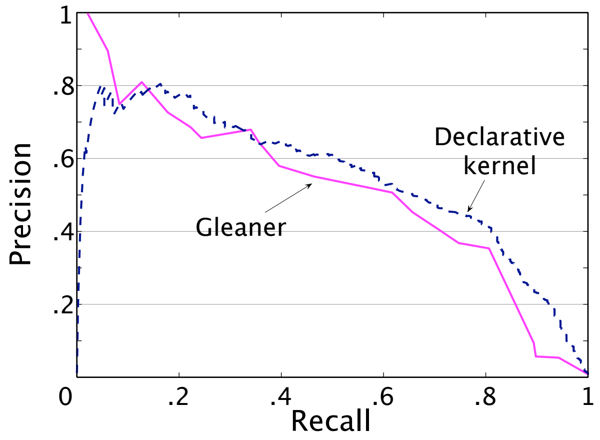
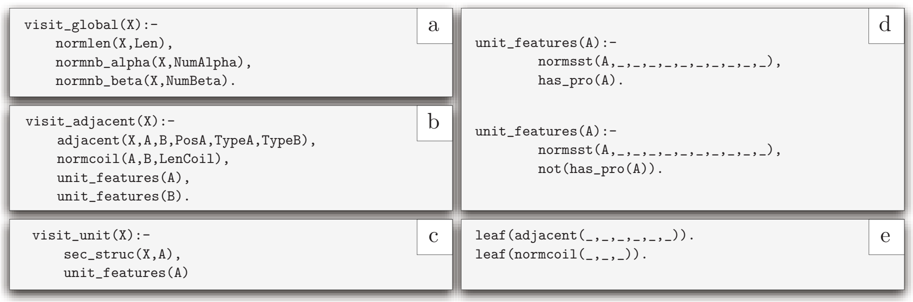


Fig. 9. Comparing Gleaner and the declarative kernel on the information extraction task (fold 5)

The data set is a collection of 7, 245 sentences from 871 abstracts, yielding 1, 773 positive and 279, 154 negative instances. The data is enriched by a large body of domain knowledge, including relations about the structure of sentences and abstracts, lexical knowledge, and biological knowledge derived from several specialized vocabularies and ontologies such as MeSH and Gene Ontology. For simplicity, only a fraction of the available knowledge has been used in our experiments. The main data types in this domain are: `instance` (pairs of candidate NP's); `cp_NP` (candidate protein NP); `cl_NP` (candidate location NP); `word_p` (word in a protein NP); `word_l` (word in a location NP). Basic parthood rules in the ontology declare that phrases (`cp_NP` and `cl_NP`) are parts of instances and words are parts of phrases. For this task we used a minimal mereological kernel with no connections and no axiomatic theory to avoid explosion of features due to words appearing both as part of NP's and instances. We compared declarative kernels to state-of-the-art ILP-based system for this domain: Aleph and Gleaner [73]. We used the same setting as in [73], performing a five-fold cross validation, with approximately 250 positive and 120, 000 negative examples in each fold (split at the level of abstracts), and measuring the quality of the predictor by means of the *area under the recall-precision curve* (AURPC). As reported in [73], Aleph attains its best performance (area .45) by learning on the order of 10^8 rules, while Gleaner attains similar performance ($.43 \pm .6$) but using several orders of magnitude less rules [74]. We trained five SVMs using the declarative kernel composed with a Gaussian kernel. Gaussian width and the regularization parameter were selected by reserving a validation set inside each fold. The obtained AURPC was $.47 \pm .7$. Figure 9 compares the recall precision curve reported in [73], which is produced by Gleaner using 1, 000, 000 candidate clauses on fold five, with that obtained by the declarative kernel. The result is very encouraging given that only a fraction of the available knowledge has been used. Training took less than three hours on a single 3.00GHz Pentium while Aleph and Gleaner run for several days on a large PC cluster on the same task [74].

7.2 Proof Tree Kernels for Protein Fold Classification

Binary Classification. In our first experiment, we tested our methodology on the protein fold classification problem studied by Turcotte et al. [75]. The task consists of classifying proteins into Scop folds, given their high-level logical descriptions about secondary structure and amino acid sequence. Scop is a manually curated database of proteins hierarchically organized according to their structural properties. At the top level Scop groups proteins into four main classes (all-α, all-β, α/β, and $\alpha + \beta$). Each class is then divided into folds that group together proteins with similar secondary structures and three-dimensional arrangements. We used the data set made available as a supplement to the paper by Turcotte et al. [75][5] that consists of the five most populated folds from each of the four main Scop classes. This setting yields 20 binary classification problems. The data sets for each of the 20 problems are relatively small (from about 30 to about 160 examples per fold, totaling 1143 examples).



```
visit_global(X):-
    normlen(X,Len),
    normnb_alpha(X,NumAlpha),
    normnb_beta(X,NumBeta).
```

```
visit_adjacent(X):-
    adjacent(X,A,B,PosA,TypeA,TypeB),
    normcoil(A,B,LenCoil),
    unit_features(A),
    unit_features(B).
```

```
visit_unit(X):-
    sec_struc(X,A),
    unit_features(A)
```

```
unit_features(A):-
        normsst(A,_,_,_,_,_,_,_,_,_,_),
        has_pro(A).

unit_features(A):-
        normsst(A,_,_,_,_,_,_,_,_,_,_),
        not(has_pro(A)).
```

```
leaf(adjacent(_,_,_,_,_,_)).
leaf(normcoil(_,_,_)).
```

Fig. 10. Visitors for the protein fold classification problem

We relied on the background knowledge provided in [75], to design a set of visitors managing increasingly complex information. Visitors are shown in Figure 10. The "global" visitor `visit_global/1` is meant to extract protein level information, such as its length and the number of its α or β secondary structure segments. A "local" visitor `visit_unit/1` explores the details of each of these segments. In particular, after determining the secondary structure element, it explores the general features of the element using `normsst/11` and checks for the presence of proline (an amino acid that is known to have important effects on the secondary structure). Note that since traces are recorded as proof trees, the first clause of the predicate `unit_features/1` above produces information only in the case a proline is present. Finally, the "connection" visitor `visit_adjacent/1` inspects pairs of adjacent segments within the protein.

Numerical values were normalized within each top level fold class. The kernel configuration mainly consisted of type signatures aiming to ignore identifiers

[5] Available at `http://www.bmm.icnet.uk/ilp/data/ml_2000.tar.gz`

Table 2. Protein fold classification: 10-fold cross validation accuracy (%) for Tilde, Progol and SVM for the different classification tasks, and micro averaged accuracies with 95% confidence intervals. Results for Progol are taken from [75].

	Tilde	Progol	SVM
All-α:			
Globin-like	97.4	95.1	94.9
DNA-binding 3-helical bundle	81.1	83.0	88.9
4-helical cytokines	83.3	70.7	86.7
lambda repressor-like DNA-binding domains	70.0	73.4	83.3
EF Hand-like	71.4	77.6	85.7
All-β:			
Immunoglobulin-like beta-sandwich	74.1	76.3	85.2
SH3-like barrel	91.7	91.4	93.8
OB-fold	65.0	78.4	83.3
Trypsin-like serine proteases	95.2	93.1	93.7
Lipocalins	83.3	88.3	92.9
α/β:			
beta/alpha (TIM)-barrel	69.7	70.7	73.3
NAD(P)-binding Rossmann-fold domains	79.4	71.6	84.1
P-loop containing nucleotide triphosphate hydrolases	64.3	76.0	76.2
alpha/beta-Hydrolases	58.3	72.2	86.1
Periplasmic binding protein-like II	79.5	68.9	79.5
$\alpha + \beta$:			
Interleukin 8-like chemokines	92.6	92.9	96.3
beta-Grasp	52.8	71.7	88.9
Ferredoxin-like	69.2	83.1	76.9
Zincin-like	51.3	64.3	79.5
SH2-like	82.1	76.8	66.7
Micro average:	75.2 $\pm$2.5	78.3 $\pm$2.4	83.6 $\pm$2.2

and treat some of the numerical features as categorical ones. A functor equality kernel was employed for those nodes of the proofs which did not contain valuable information in their arguments.

Following [75], we measured prediction accuracy by 10-fold cross-validation, micro-averaging the results over the 20 experiments by summing contingency tables. The proof-tree kernel was combined with a Gaussian kernel (see Eq. (9)) in order to model nonlinear interactions between the features extracted by the visitor program. Model selection (i.e., choice of the Gaussian width γ and the SVM regularization parameter C) was performed for each binary problem with a LOO procedure before running the 10-fold cross validation. Table 2 shows comparisons between the best setting for Progol (as reported by [75]), which uses both propositional and relational background knowledge, results for Tilde using the same setting, and SVM with our kernel over proof trees. The difference between Tilde and Progol is not significant, while our SVM achieves significantly higher overall accuracy with respect to both methods. The only task where our predictor performed worse than both ILP methods was the SH2-like one (the last one in Table 2). It is interesting to note that a simple global visitor would achieve 84.6% accuracy on this task, while in most other tasks full relational features produce consistently better results. This can suggest that even if SVMs are capable of effectively dealing with huge feature spaces, great amounts of uninformative or noisy features can also degrade performance, especially if only few examples are available.

Multiclass Classification. We additionally evaluated our proof tree kernels on the multiclass setting of the protein fold prediction task as described in [76]. The problem is based on the same 20 SCOP folds previously used for binary classification, but the data set contains only the chains considered as positive examples for one of the SCOP folds in the binary classification problems. Four independent multiclass classification problems are defined, one for each of the main fold classes in SCOP. A single multiclass problem consists of discriminating between chains belonging to the same fold class, by assigning each of them to one of the five main folds in the fold class. The statistics of the dataset are reported in Table 3, and show the unbalancing of the distribution of examples between folds. We employed a one-vs-all strategy to address each multiclass classification task: we trained a number of binary classifiers equal to the number of classes, each trained to discriminate between examples of one class and examples of all other classes; during testing, we presented each example to all binary classifiers, and assigned it to the class for which the corresponding binary classifier was the most confident, as measured by the margin of the prediction for the example. We employed the same 5-fold cross validation procedure as reported in [76] and

Table 3. Number of examples for each multiclass problem (fold class) both divided by single class (fold) and overall

fold class	fold$_1$	fold$_2$	fold$_3$	fold$_4$	fold$_5$	overall
all-α	13	30	10	10	14	77
all-β	90	32	40	42	28	116
α/β	55	21	14	12	13	115
$\alpha+\beta$	9	12	26	13	13	73

Table 4. Microaveraged accuracies with standard errors for the four multiclass problems and overall accuracy microaveraged over problems: comparison between SVM, SLP and ILP with majority class

fold class	SVM	SLP	ILP + majority class
all-α	80.5±4.5 (62/77)	76.6±4.8 (59/77)	71.4±5.2 (55/77)
all-β	87.1±3.1(101/116)	81.0±3.6 (94/116)	69.8±4.3 (81/116)
α/β	61.7±4.5 (71/115)	51.3±4.7 (59/115)	44.4±4.6 (51/115)
$\alpha+\beta$	60.3±5.7 (44/73)	82.2±4.5 (60/73)	80.8±4.6 (59/73)
overall	73.0±2.3 (278/381)	71.4±2.3 (272/381)	64.6±2.5 (246/381)

used exactly the same CV folds. Model selection (Gaussian width and regularization parameter) was conducted by a preliminary LOO procedure as in the case of binary classification, but using the F_1 measure (the harmonic mean of precision and recall) as a guiding criterion. We kept the same visitors developed for the binary setting. Table 4 reports accuracies with standard errors for the four multiclass problems, microaveraged on the CV folds, and overall accuracy microaveraged on CV folds and multiclass problems. Reported results include our SVM with proof tree kernels together to the results obtained by a stochastic logic program (SLP) and ILP with majority class prediction as reported in [76].

Results show that both the SLP and the kernel machine outperform the non-probabilistic ILP approach. In three out of four SCOP folds the SVM obtained a higher microaveraged accuracy than the SLP although the data sets have small size and the standard deviation is rather high. Interestingly, the SLP seems to perform better on smaller data set, which might indicate a faster rate of convergence of the SLP to its asymptotic error.

8 Conclusions

In this chapter we have pursued the construction of a bridge between two very different paradigms of machine learning: statistical learning with kernels, and inductive logic programming. In particular, we have shown in several ways that the use of stable (robust to noise) machine learning techniques are applicable in the ILP setting without resorting to propositionalization. This is especially interesting in cases where the feature space needed for representing the solution has a dimension that is not known in advance. The artificial Bongard data set shows this clearly. Of course one could have solved the Bongard problem even with traditional ILP techniques by adding to the background theory a predicate counting the number of polygons nested one inside another, but kernels on program traces can effectively discover this concept without the additional hint.

The algorithmic stability achieved by combining ILP with regularization can be seen as an interesting alternative to fully fledged probabilistic ILP where structure and parameters of a stochastic program are learned from data. Empirical evidence on real-world problems such as the protein fold classification task demonstrates that proof tree kernels can achieve better accuracy than non probabilistic ILP and similar accuracy as learning stochastic logic programs. Of course the solution found by the kernel machine in this case lacks interpretability. However, computational efficiency is another factor that in some cases needs to be taken into account. For example, problems like the information extraction task presented in Section 7.1 can be solved in a fraction of the time required by a state-of-the-art ILP system.

kFOIL is perhaps the less developed and tested method but at the same time very promising. Its approach to propositionalization is effectively dynamic and can be interpreted in close connection to methods that attempt to learn the kernel matrix from data [77,78,79]. Moreover, the solution found by kFOIL combines the advantages of kernel machines and ILP systems. It consists of both a robust decision function and a kernel function defined by interpretable first-order clauses. This is a direction of research that certainly deserves more investigation since interpretability is one of the weak aspects of many statistical learning methods.

Finally, some of the ideas seeded here may deserve more study from an engineering perspective, as the availability of languages for feature description (like declarative kernels) can help leveraging machine learning into routine software development. One of the main obstacles towards this goal is maybe the difficulty of integrating machine learning capabilities in a typical programming environment. The recent growth of interest in knowledge representation and ontologies suggests that logic-based representations may be much more widespread in the future, and that the attention to learning modules that can take advantage of data and existing knowledge with minimal programmer intervention could increase.

Acknowledgments

We would like to thank Luc De Raedt, Niels Landwehr, and Stephen Muggleton, with whom we collaborated in the development of the methods presented in this chapter (in particular, we collaborated with LDR on kernels on Prolog proof trees, with SM on declarative kernels, with LDR and NL on kFOIL; NL also carried out the experiments reported in Section 6.3). A special thanks goes to Alessio Ceroni, Fabrizio Costa, Kristian Kersting, Sauro Menchetti, and Jan Ramon, who helped us in numerous occasions with fruitful discussions. This work was supported by the European Union, contract number FP6-508861, Applications of Probabilistic Inductive Logic Programming II.

References

1. Schölkopf, B., Smola, A.: Learning with Kernels. The MIT Press, Cambridge (2002)
2. Shawe-Taylor, J., Cristianini, N.: Kernel Methods for Pattern Analysis. Cambridge University Press, Cambridge (2004)
3. Lodhi, H., Saunders, C., Shawe-Taylor, J., Cristianini, N., Watkins, C.: Text classification using string kernels. J. Mach. Learn. Res. 2, 419–444 (2002)
4. Jaakkola, T., Haussler, D.: Exploiting generative models in discriminative classifiers. In: Advances in Neural Information Processing Systems 11, pp. 487–493. MIT Press, Cambridge (1999)
5. Leslie, C.S., Eskin, E., Noble, W.S.: The spectrum kernel: A string kernel for svm protein classification. In: Pacific Symposium on Biocomputing, pp. 566–575 (2002)
6. Cortes, C., Haffner, P., Mohri, M.: Rational kernels: Theory and algorithms. Journal of Machine Learning Research 5, 1035–1062 (2004)
7. Collins, M., Duffy, N.: New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In: Proceedings of the Fortieth Annual Meeting on Association for Computational Linguistics, Philadelphia, PA, USA, pp. 263–270 (2002)

8. Viswanathan, S., Smola, A.J.: Fast kernels for string and tree matching. In: Becker, S.T., S., Obermayer, K. (eds.) Advances in Neural Information Processing Systems, vol. 15, pp. 569–576. MIT Press, Cambridge (2003)
9. Gärtner, T.: A survey of kernels for structured data. SIGKDD Explorations Newsletter 5(1), 49–58 (2003)
10. Smola, A.J., Kondor, R.: Kernels and Regularization on Graphs. In: Schölkopf, B., Warmuth, M.K. (eds.) COLT/Kernel 2003. LNCS (LNAI), vol. 2777, pp. 144–158. Springer, Heidelberg (2003)
11. Kashima, H., Tsuda, K., Inokuchi, A.: Marginalized kernels between labeled graphs. In: Proceedings of ICML 2003 (2003)
12. Mahé, P., Ueda, N., Akutsu, T., Perret, J.L., Vert, J.P.: Extensions of marginalized graph kernels. In: Greiner, R., D. Schuurmans, A.P. (eds.) Proceedings of the Twenty-first International Conference on Machine Learning, Banff, Alberta, Canada, pp. 552–559 (2004)
13. Horváth, T., Gärtner, T., Wrobel, S.: Cyclic pattern kernels for predictive graph mining. In: Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 158–167. ACM Press, New York (2004)
14. Menchetti, S., Costa, F., Frasconi, P.: Weighted decomposition kernels. In: Proceedings of the Twenty-second International Conference on Machine Learning, pp. 585–592. ACM Press, New York (2005)
15. Kramer, S., Lavrac, N., Flach, P.: Propositionalization approaches to relational data mining. In: Relational Data Mining, pp. 262–286. Springer, Heidelberg (2000)
16. Cumby, C.M., Roth, D.: Learning with feature description logics. In: Matwin, S., Sammut, C. (eds.) ILP 2002. LNCS (LNAI), vol. 2583, pp. 32–47. Springer, Heidelberg (2003)
17. Cumby, C.M., Roth, D.: On kernel methods for relational learning. In: Proceedings of ICML 2003 (2003)
18. Ramon, J., Bruynooghe, M.: A Framework for Defining Distances Between First-Order Logic Objects. In: Proc. of the 8th International Conf. on Inductive Logic Programming, pp. 271–280 (1998)
19. Kirsten, M., Wrobel, S., Horváth, T.: Distance based approaches to relational learning and clustering. In: Relational Data Mining, pp. 213–230. Springer, Heidelberg (2001)
20. Ramon, J.: Clustering and instance based learning in first order logic. AI Communications 15(4), 217–218 (2002)
21. Cortes, C., Vapnik, V.N.: Support vector networks. Machine Learning 20, 1–25 (1995)
22. De Raedt, L.: Logical and Relational Learning: From ILP to MRDM. Springer, Heidelberg (2006)
23. Vapnik, V.N.: The Nature of Statistical Learning Theory. Springer, New York (1995)
24. Herbrich, R., Graepel, T., Obermayer, K.: Support vector learning for ordinal regression. In: Artificial Neural Networks, 1999. ICANN 1999. Ninth International Conference on (Conf. Publ. No. 470), vol. 1 (1999)
25. Tax, D., Duin, R.: Support vector domain description. Pattern Recognition Letters 20, 1991–1999 (1999)
26. Ben-Hur, A., Horn, D., Siegelmann, H., Vapnik, V.: Support vector clustering. Journal of Machine Learning Research 2, 125–137 (2001)
27. Schölkopf, B., Smola, A., Müller, K.: Nonlinear component analysis as a kernel eigenvalue problem. Neural computation 10(5), 1299–1319 (1998)

28. Kramer, S.: Structural regression trees. In: Proceedings of the Thirteenth National Conference on Artificial Intelligence, pp. 812–819 (1996)
29. Kramer, S.: Prediction of Ordinal Classes Using Regression Trees. Fundamenta Informaticae 47(1), 1–13 (2001)
30. Cucker, F., Smale, S.: On the mathematical foundations of learning. Bulletin (New Series) of the American Mathematical Society 39(1), 1–49 (2002)
31. Lin, Y.: Support Vector Machines and the Bayes Rule in Classification. Data Mining and Knowledge Discovery 6(3), 259–275 (2002)
32. Bartlett, P., Jordan, M., McAuliffe, J.: Large margin classifiers: Convex loss, low noise, and convergence rates. Advances in Neural Information Processing Systems 16 (2003)
33. Ng, A., Jordan, M.: On Discriminative vs. Generative classifiers: A comparison of logistic regression and naive Bayes. Neural Information Processing Systems (2001)
34. Passerini, A., Frasconi, P.: Kernels on prolog ground terms. In: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, pp. 1626–1627 (2005)
35. Gärtner, T., Lloyd, J., Flach, P.: Kernels for structured data. In: Matwin, S., Sammut, C. (eds.) ILP 2002. LNCS (LNAI), vol. 2583, pp. 66–83. Springer, Heidelberg (2003)
36. Passerini, A., Frasconi, P., De Raedt, L.: Kernels on prolog proof trees: Statistical learning in the ILP setting. Journal of Machine Learning Research 7, 307–342 (2006)
37. Landwehr, N., Passerini, A., Raedt, L.D., Frasconi, P.: kFOIL: Learning simple relational kernels. In: Gil, Y., Mooney, R. (eds.) Proc. Twenty-First National Conference on Artificial Intelligence (AAAI 2006), AAAI Press, Menlo Park (2006)
38. Quinlan, J.R.: Learning Logical Definitions from Relations. Machine Learning 5, 239–266 (1990)
39. Saunders, G., Gammerman, A., Vovk, V.: Ridge regression learning algorithm in dual variables. In: Proc. 15th International Conf. on Machine Learning, pp. 515–521 (1998)
40. Poggio, T., Smale, S.: The mathematics of learning: Dealing with data. Notices of the American Mathematical Society 50(5), 537–544 (2003)
41. Kimeldorf, G.S., Wahba, G.: A correspondence between Bayesian estimation on stochastic processes and smoothing by splines. The Annals of Mathematical Statistics 41, 495–502 (1970)
42. Freund, Y., Schapire, R.E.: Large margin classification using the perceptron algorithm. Machine Learning 37(3), 277–296 (1999)
43. Haussler, D.: Convolution kernels on discrete structures. Technical Report UCSC-CRL-99-10, University of California, Santa Cruz (1999)
44. Lodhi, H., Shawe-Taylor, J., Cristianini, N., Watkins, C.: Text classification using string kernels. Advances in Neural Information Processing Systems, 563–569 (2000)
45. Collins, M., Duffy, N.: Convolution kernels for natural language. In: NIPS 14, pp. 625–632 (2001)
46. Gärtner, T., Flach, P., Kowalczyk, A., Smola, A.: Multi-instance kernels. In: Sammut, C., Hoffmann, A. (eds.) Proceedings of the 19^{th} International Conference on Machine Learning, pp. 179–186. Morgan Kaufmann, San Francisco (2002)
47. Srinivasan, A., Muggleton, S., Sternberg, M.J.E., King, R.D.: Theories for mutagenicity: A study in first-order and feature-based induction. Artificial Intelligence 85(1-2), 277–299 (1996)
48. Lloyd, J.W.: Logic for learning: Learning comprehensible theories from structured data. Springer, Heidelberg (2003)

49. Taskar, B., Abbeel, P., Koller, D.: Discriminative probabilistic models for relational data. In: Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence, Morgan Kaufmann, San Francisco (2002)
50. Neville, J., Jensen, D.: Collective classification with relational dependency networks. In: Proceedings of the Second International Workshop on Multi-Relational Data Mining, pp. 77–91 (2003)
51. Lakshman, T.K., Reddy, U.S.: Typed prolog: A semantic reconstruction of the mycroft-O'keefe type system. In: Saraswat, Vijay, Ueda, K. (eds.) Proceedings of the 1991 International Symposium on Logic Programming (ISLP 1991), pp. 202–220. MIT Press, San Diego (1991)
52. Gärtner, T., Lloyd, J., Flach, P.: Kernels and distances for structured data. Machine Learning 57(3), 205–232 (2004)
53. Ramon, J., Bruynooghe, M.: A polynomial time computable metric between point sets. Acta Informatica 37(10), 765–780 (2001)
54. Horváth, T., Wrobel, S., Bohnebeck, U.: Relational instance-based learning with lists and terms. Machine Learning 43(1/2), 53–80 (2001)
55. Passerini, A., Frasconi, P., De Raedt, L.: Kernels on prolog proof trees: Statistical learning in the ILP setting. Journal of Machine Learning Research 7, 307–342 (2006)
56. Bianucci, A., Micheli, A., Sperduti, A., Starita, A.: Application of cascade correlation networks for structures to chemistry. Appl. Intell. 12, 117–146 (2000)
57. Leśniewski, S.: Podstawy ogólnej teorii mnogości. Moscow (1916)
58. Leonard, H.S., Goodman, N.: The calculus of individuals and its uses. Journal of Symbolic Logic 5(2), 45–55 (1940)
59. Casati, R., Varzi, A.: Parts and places: The structures of spatial representation. MIT Press, Cambridge, MA and London (1999)
60. Joachims, T.: Making large-scale SVM learning practical. In: Schölkopf, B., Burges, C., Smola, A. (eds.) Advances in Kernel Methods – Support Vector Learning, pp. 169–185. MIT Press, Cambridge (1998)
61. Srinivasan, A.: The Aleph Manual. Oxford University Computing Laboratory (2001)
62. Biermann, A., Krishnaswamy, R.: Constructing programs from example computations. IEEE Transactions on Software Engineering 2(3), 141–153 (1976)
63. Mitchell, T.M., Utgoff, P.E., Banerji, R.: Learning by experimentation: Acquiring and refining problem-solving heuristics. In: Machine learning: An artificial intelligence approach, vol. 1, pp. 163–190. Morgan Kaufmann, San Francisco (1983)
64. Shapiro, E.Y.: Algorithmic program debugging. MIT Press, Cambridge (1983)
65. Zelle, J.M., Mooney, R.J.: Combining FOIL and EBG to speed-up logic programs. In: Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence, Chambéry, France, pp. 1106–1111 (1993)
66. De Raedt, L., Kersting, K., Torge, S.: Towards learning stochastic logic programs from proof-banks. In: Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI 2005), pp. 752–757 (2005)
67. Muggleton, S., Lodhi, H., Amini, A., Sternberg, M.: Support vector inductive logic programming. In: Hoffmann, A., Motoda, H., Scheffer, T. (eds.) DS 2005. LNCS (LNAI), vol. 3735, pp. 163–175. Springer, Heidelberg (2005)
68. Russell, S., Norvig, P.: Artifical Intelligence: A Modern Approach, 2nd edn. Prentice-Hall, Englewood Cliffs (2002)
69. Bongard, M.: Pattern Recognition. Spartan Books (1970)

70. Landwehr, N., Kersting, K., De Raedt, L.: nFOIL: Integrating Naïve Bayes and FOIL. In: Proc. of the 20th National Conf. on Artificial Intelligence, pp. 795–800 (2005)
71. Blockeel, H., Dzeroski, S., Kompare, B., Kramer, S., Pfahringer, B., Laer, W.: Experiments in Predicting Biodegradability. Applied Artificial Intelligence 18(2), 157–181 (2004)
72. Ray, S., Craven, M.: Representing sentence structure in hidden Markov models for information extraction. In: Proceedings of IJCAI 2001, pp. 1273–1279 (2001)
73. Goadrich, M., Oliphant, L., Shavlik, J.W.: Learning ensembles of first-order clauses for recall-precision curves: A case study in biomedical information extraction. In: Camacho, R., King, R., Srinivasan, A. (eds.) ILP 2004. LNCS (LNAI), vol. 3194, pp. 98–115. Springer, Heidelberg (2004)
74. Goadrich, M.: Personal communication (2005)
75. Turcotte, M., Muggleton, S., Sternberg, M.: The effect of relational background knowledge on learning of protein three-dimensional fold signatures. Machine Learning 43(1-2), 81–96 (2001)
76. Chen, J., Kelley, L., Muggleton, S., Sternberg, M.: Multi-class prediction using stochastic logic programs. In: Muggleton, S., Otero, R., Tamaddoni-Nezhad, A. (eds.) ILP 2006. LNCS (LNAI), vol. 4455, Springer, Heidelberg (2007)
77. Lanckriet, G.R.G., Cristianini, N., Bartlett, P., Ghaoui, L.E., Jordan, M.I.: Learning the kernel matrix with semidefinite programming. J. Mach. Learn. Res. 5, 27–72 (2004)
78. Ong, C.S., Smola, A.J., Williamson, R.C.: Hyperkernels. In: Adv. in Neural Inf. Proc. Systems (2002)
79. Micchelli, C.A., Pontil, M.: Learning the Kernel Function via Regularization. Journal of Machine Learning Research 6, 1099–1125 (2005)