Kernels on Prolog Proof Trees: Statistical Learning in the ILP Setting

A. Passerini PASSERINI@DSI.UNIFI.IT
P. Frasconi P-F@DSI.UNIFI.IT
Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze

L. De Raedt DERAEDT@INFORMATIK.UNI-FREIBURG.DE
Institute for Computer Science, Albert-Ludwigs Universität, Freiburg

Abstract

We develop kernels for measuring the similarity between relational instances using background knowledge expressed in first-order logic. The method allows us to bridge the gap between traditional inductive logic programming representations and statistical approaches to supervised learning. Logic programs will be used to generate proofs of given visitor programs which exploit the available background knowledge, while kernel machines will be employed to learn from such proofs. We report positive empirical results on Bongard-like and M-of-N problems that are difficult or impossible to solve with traditional ILP techniques, as well as on a real data set.

1. Introduction

Within the field of automated program synthesis, inductive logic programming and machine learning, several approaches exist that learn from example-traces. An example-trace is a sequence of steps taken by a program on a particular example input. For instance, Alan Bierman (Biermann & Krishnaswamy, 1976) has sketched how to induce Turing machines from example-traces; Mitchell et al. have developed the LEX system (Mitchell et al., 1983) that learned how to solve symbolic integration problems by analyzing traces (or search trees) for particular example problems; Ehud Shapiro's Model Inference System (Shapiro, 1983) inductively infers logic programs by reconstructing the proof-trees and traces corresponding to particular facts; and Zelle and Mooney (Zelle & Mooney, 1993) show how to speed-up the execution of logic programs by analyzing example-traces of the underlying logic program. The diversity of these applications as well as the difficulty of the learning tasks considered clearly illustrate the power of learning from example-traces for a wide range of applications.

In the present paper, we generalize the idea of learning from example-traces. Rather than explicitly learning a target program from positive and negative example traces, we assume that a particular – so-called *visitor* program – is given and that our task consists of learning from the associated traces. The advantage is that in principle any programming language can be used to model the visitor program and that any machine learning system able use traces as an intermediate representation can be employed. In particular, this allows us to combine two frequently employed frameworks within the field of machine learning: inductive logic programming and kernel methods. Logic programs will be used to generate traces corresponding to specific examples and kernels will be employed for quantifying the similarity between traces. The combination yields an appealing and expressive framework for tackling complex learning tasks involving structured data in a natural manner. We call *trace kernels* the resulting broad family of kernel functions obtainable as a result of this combination. The visitor program is a set of clauses that can be seen as the *interface* between the available background knowledge and the kernel itself. Intuitively, visitors are employed to specify a set of useful features and in this sense play a role similar to `rmode`s in ILP.

Starting from the seminal work of Haussler (Haussler, 1999), several researchers have already proposed kernels on discrete data structures such as sequences (Lodhi et al., 2000; Jaakkola & Haussler, 1998; Leslie et al., 2002; Cortes et al., 2004), trees (Collins & Duffy, 2002; Vishwanathan & Smola, 2002), annotated graphs (Gärtner, 2003; Schölkopf & Warmuth, 2003), and complex individuals defined using higher order logic abstractions (Gärtner et al., 2004). Constructing kernels on structured data types, however, is not the only aim of the proposed framework. In many

symbolic approaches to learning, logic programs allow us to define background knowledge in a very natural way. Similarly, in the case of kernel methods, the notion of similarity between two instances expressed by the kernel function is the main tool for exploiting the available domain knowledge. It seems therefore natural to seek a link between logic programs and kernels, also as a mean for embedding knowledge into statistical learning algorithms in a *principled* and *flexible* way. This aspect is an important contribution of this paper as few alternatives exist to achieve this goal. Propositionalization, for example, transforms a relational problem into one that can be solved by an attribute-value learner by mapping data structures into a finite set of features (Kramer et al., 2000). Although it is known that in many practical applications propositionalization works well, its flexibility is generally limited. A remarkable exception is the method proposed in (Cumby & Roth, 2002) that uses description logic to specify features and that has been subsequently extended to specify kernels (Cumby & Roth, 2003).

The guiding philosophy of trace kernels is very different from the above approaches. Intuitively, rather than defining a kernel function that compares two given instances, we define a kernel function that compares the execution traces of a program (that expresses background knowledge) run over the two given instances. Similar instances should produce similar traces when probed with programs examining characteristics they have in common. Clearly 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 convolution and decomposition kernels. In particular, *any* program to be executed on data can be exploited within the present framework to form a valid kernel function, provided one can give a suitable definition of the *visitor* program to specify how to obtain relevant traces and proofs to compare examples. In addition, although in this paper we only study trace kernels for logic programs, similar ideas could be used in the context of different programming paradigms and in conjunction with alternative models of computation such as finite state automata or Turing machines.

In this paper, we focus on a specific learning framework for Prolog programs. Prolog execution traces consist of sets of search trees (see e.g. (Sterling & Shapiro, 1994)) associated with goals in the visitor program; these traces can be conveniently represented as Prolog ground terms. Thus, in this case, kernels over traces reduce to Prolog ground terms kernels (PGTKs) (Passerini & Frasconi, 2005). These kernels (which are briefly reviewed in Section 3.3) can be seen as a specialization to Prolog of the kernels between higher order logic individuals earlier introduced in (Gärtner et al., 2004).

The paper is organized as follows. In Section 2 we revise the classic ILP framework and describe the structure of visitor programs. In Section 3 we describe the general form of the kernel on logical objects and, in particular, Prolog proof trees, in Section 4 we give some implementation details, and finally in Section 5 we report an empirical evaluation of the methodology on some classic ILP benchmarks including Bongard problems, M of N problems on sequences, and mutagenesis.

2. Visitors and proof trees in First Order Logic

In traditional inductive logic programming approaches, the learner is given a set of positive and negative examples P and N (in the form of definite clauses that are (resp. are not) entailed by the target theory), and a background theory BK (a set of definite clauses), and has to induce a hypothesis H (a set of definite clauses) such that $BK \cup H$ covers all positive examples and none of the negative ones. More formally, $\forall p \in P : BK \cup H \models p$ and $\forall n \in N : BK \cup H \not\models n$. In practice, rather than working with ground clauses of the form $e \leftarrow f_1, ..., f_n$ as examples, inductive logic programming systems often employ e as the example and add the facts f_i to the background theory BK. As an illustration, consider the famous mutagenicity benchmark by (Srinivasan et al., 1996). There the examples are of the form mutagenic(id) where id is a unique identifier of the molecule and the background knowledge contains information about the atoms, bonds and functional groups in the molecule. A hypothesis in this case could be

mutagenic(ID) ← nitro(ID,R), lumo(ID,L), L<-1.5.

It entails, i.e., covers, the molecule listed in Fig. 1. For the purposes of this paper, it will be convenient to look at examples as objects and to consider the clausal notation $h(x) \leftarrow f_1, ..., f_n$ where x is a unique identifier of the example. Furthermore, where necessary, we will refer to the head of the example as $h(x)$ and the set of facts in the body as $F(x)$.

We can now introduce the framework of learning from trace kernels. The key difference with the traditional inductive logic programming setting is that the learner is given a set of so-called *visitor* clauses V, which de-

```
mutagenic(225).
molecule(225).
logmutag(225,0.64).
lumo(225,-1.785).
logp(225,1.01).
nitro(225,[f1_4,f1_8,f1_10,f1_9]).
atom(225,f1_1,c,21,0.187).
atom(225,f1_2,c,21,-0.143).
atom(225,f1_3,c,21,-0.143).
atom(225,f1_4,c,21,-0.013).
atom(225,f1_5,o,52,-0.043).
...
```

Figure 1. An example from the mutagenesis domain

fine *visitor* predicates and which replace the hypothesis H. So rather than having to find a set of clauses H, the learner is given a set of clauses V. The idea then is that for each example x, the proofs of the visitor predicates are computed. These proofs then constitute the representation employed by the kernel, which has to learn how to discriminate the set of proofs for a positive example from those of a negative example. The rationale behind the use of the program trace is the idea that not only the success or failure of the goal is of interest in order to characterize a given instance, but also the full trace of steps passed in order to produce such a result. Different visitors can be conceived in order to explore different aspects of the examples and include multiple sources of information.

This idea can be formalized as follows: for each example $(h(x), F(x))$, background theory BK and visitor clauses V defining visitor predicates v_i, we compute the set of proofs $P_i(x) = \{p \mid p \text{ is a proof such that } BK \cup F(x) \cup V \models v_i(x)\}$.

So far, we have not detailed which type of proof or trace is employed. At this point, there are several possibilities. One could employ the SLD-tree, which would not only contain information about succeeding proofs but also about failing ones. The SLD-tree is however a very complex and rather unstructured representation. It is much more convenient to work with *and*-trees for the visitor facts.

An *and-tree* for a query v for an example $(h(x), F(x))$, a background theory BK and visitor clauses V for which $F(x) \cup BK \cup V \models v$ is a tree such that

- v is the root of the tree and
- if v is a fact in $F(x) \cup BK \cup V$ then v is a leaf
- otherwise there must be a clause $w \leftarrow b_1, ..., b_n \in BK \cup V$ and a substitution θ grounding it such that $w\theta = v$ and $BK \cup V \models b_i\theta \forall i$ and there is a subtree of v for each $b_i\theta$ that is an and-tree for $b_i\theta$

The simplest visitor we can imagine just ignores the background knowledge and extracts the ground facts concerning a given example (or a subset of them). Note that visitors actually allow us to expand the example representation as described in (Lloyd, 2003) by naturally including information derived from the background knowledge.

As an example, consider again the mutagenicity benchmark. The following is the atom bond representation of the simple molecule in Figure 2. By looking at the molecule as a graph where atoms are nodes and bonds are edges, we can introduce the common notions of *path* and *cycle*:

```
1 : cycle(E,X):-
      path(E,X,Y,[X]),
      bond(E,Y,X,_).

2 : path(E,X,Y,M):-
      atm(E,X,_,_,_),
      bond(E,X,Y,_),
      atm(E,Y,_,_,_),
      \+ member(Y,M).

3 : path(E,X,Y,M):-
      atm(E,X,_,_,_),
      bond(E,X,Z,_),
      \+ member(Z,M),
      path(E,Z,Y,[Z|M]).
```

A possible visitor in such context would be the one simply looking for a cycle in the molecule, which can be written as:

```
4 : visit(E):
      cycle(E,X).
```

Note that we numbered each clause in $BK \cup V$ (but not in $F(e)$[1]) with a unique identifier. This will allow us to take into account information about the clauses that are used in a proof.

In many situations, the and-tree for a given goal will be unnecessary complex in that it may contain several uninteresting subtrees. To account for this situation, we will often work with *pruned* and-trees, which are trees where subtrees rooted at specific predicates (declared as *leaf* predicates by the user) are turned into leafs. This will allow the kernel to ignore the way atoms involving these predicates are proved. For instance, consider again the molecule in Figure 2, and suppose we have the background knowledge of functional groups as described in (Srinivasan et al., 1996). A potential visitor could look for a benzene ring within the molecule, and eventually find out the details of the

[1] These numbers would change from example to example and hence, would not carry any useful information.

```
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).
```

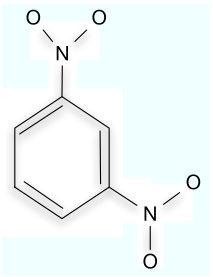

Figure 2. Simple molecule from the mutagenicity benchmark.

atoms involved. In this case it could be convenient to ignore the details of the proof of the ring, provided the atoms involved are extracted. This would be implemented by the predicate `visit_benzene` as follows:

```
1 : atoms(E,[]).

2 : atoms(E,[H|T]):-
      atm(E,H,_,_,_),
      atoms(E,T).

3 : visit_benzene(E):-
      benzene(E,Atoms),
      atoms(E,Atoms).

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

It is important to note that in general a goal can be satisfied in a number of alternative ways. Therefore, a visitor predicate actually generates a (possibly empty) set of proof trees. Furthermore, as we already underlined, different visitors can be conceived in order to analyse different characteristics of the data. An example is thus represented as a tuple of sets of proof trees, obtained by running all the available visitors on it. Given such a representation, we are now able to develop kernels over pairs of examples.

3. Bridging the Gap: Kernels over Logical Objects

Having defined the program traces generated by the visitors, in this section we detail how traces are compared by a kernel over tuples of sets of proof trees.

3.1. Kernels for Discrete Structures

A very general formulation of kernels on discrete structures is that of convolution kernels (Haussler, 1999). Suppose $x \in \mathcal{X}$ is a composite structure made of "parts" $x_1, \ldots, x_D$ such that $x_d \in \mathcal{X}_d$ for all $i \in [1, D]$. This can be formally represented by a relation R on $\mathcal{X}_1 \times \cdots \times \mathcal{X}_D \times \mathcal{X}$ such that $R(x_1, \ldots, x_D, x)$ is true iff $x_1, \ldots, x_D$ are the parts of x. Given a set of kernels $K_d : \mathcal{X}_d \times \mathcal{X}_d \to \mathbb{R}$, one for each of the parts of x, the *R-convolution* kernel is defined as

$$(K_1 \star \cdots \star K_D)(x, z) = \sum_R \prod_{d=1}^{D} K_d(x_d, z_d), \quad (1)$$

where the sum runs over all the possible decompositions of x and z. For finite relations R, this can be shown to be a valid kernel (Haussler, 1999).

A special case of convolution kernel, which will prove useful in defining kernels between proof trees, is the set kernel (Shawe-Taylor & Cristianini, 2004). Provided an object can be represented as a set of simpler objects, we define the part-of relation to be the set-membership, and the kernel reduces to the sum of all pairwise kernels between members:

$$K_{set}(x, z) = \sum_{\xi \in x, \zeta \in z} K_{member}(\xi, \zeta). \quad (2)$$

In order to reduce the dependence on the dimension of the objects, kernels over discrete structures are often normalized. A commmon choice is that of using normalization in feature space, given by:

$$K_{norm}(x, z) = \frac{K(x, z)}{\sqrt{K(x, x)}\sqrt{K(z, z)}}. \quad (3)$$

In the case of set kernels, an alternative is that of dividing by the size of the two sets, thus computing

the mean value between pairwise comparisons:

$$K_{mean}(x,z) = \frac{K_{set}(x,z)}{|x||z|}. \quad (4)$$

This formalism allows us to define a kernel over logical objects as the convolution kernel over the parts in which the objects can be decomposed according to the background knowledge available, provided we are able to define appropriate kernels between individual parts.

3.2. Kernels over Visit Programs

Assume we have a visiting program V made of a number $n \geq 1$ of visitor predicates $v_1, \ldots, v_n$, each producing a (possibly empty) set of proof trees $t_{i,j}(x)$ when tested over an example x. The proof tree representation of x can be written as:

$$P(x) = [P_1(x), \ldots, P_n(x)] \quad (5)$$

where

$$P_i(x) = \{t_{i,1}(x), \ldots, t_{i,h_i(x)}(x)\} \quad (6)$$

and $m_i(x) \geq 0$ is the number of alternative proofs of visitor v_i for example x. Assuming that we do not want to compare proof trees derived from different visitors (but it is straightforward to include such a case), we can define the kernel between examples as:

$$\begin{aligned} K(x,z) &= K_P(P(x), P(z)) \\ &= \sum_{i=1}^{n} K_i(P_i(x), P_i(z)). \end{aligned} \quad (7)$$

By using the definition of set kernel introduced in Section 3.1, we further obtain:

$$K_i(P_i(x), P_i(z)) = \sum_{j=1}^{m_i(x)} \sum_{\ell=1}^{m_i(z)} K(t_{i,j}(x), t_{i,\ell}(z)) \quad (8)$$

The problem boils down to defining the kernel between individual proof trees. Note that we can define different kernels for proof trees originating from different visitors, thus allowing for the greatest flexibility.

At the highest level of kernel between visit programs, we will employ a feature space normalization (eq. (3)). However, it is still possible to normalize lower level kernels, in order to rebalance contributions of individual parts. We will employ a mean normalization (eq. (4)) for the kernel between visitors, and possibly further normalize kernels between individual proof trees, thus reducing the influence of the dimension of proofs.

3.3. Kernels over Proof Trees

Proof trees are discrete data structures and, in principle, existing kernels on trees could be applied (e.g. (Collins & Duffy, 2002; Vishwanathan & Smola, 2002)). However, we can gain more expressiveness by representing individual proof trees as typed Prolog ground terms. In so doing we can exploit type information on constants and functors so that different sub-kernels can be applied to different object types. In addition, while traditional tree kernels would typically compare *all* pairs of subtrees between two proofs, the kernel on ground terms presented below results in a more selective approach that compares certain parts of two proofs only when reached by following similar inference steps, (a distinction that would be difficult to implement with traditional tree kernels).

We will use the following procedure to represent a proof tree as a ground term:

- Nodes corresponding to facts are already ground terms.
- Consider a node corresponding to a clause, with n arguments in the head, and the conjunction of m terms in the body, which correspond to the m children of the node.
  - Let the ground term be a compund term with $n+1$ arguments, and functor equal to the head functor of the clause.
  - Let the first n arguments be the arguments of the clause head.
  - Let the last argument be a compound term, with functor equal to the clause number[2], and m arguments equal to the ground term representations of the m children of the node.

We are now able to employ kernels on Prolog ground terms as defined in (Passerini & Frasconi, 2005) to compute kernels over individual proof trees. Let us briefly recall the definition of the kernel for typed Prolog ground terms.

We denote by $\mathcal{T}$ the ranked set of type constructors, which contains at least the nullary constructor $\perp$. 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 of a function is the type of its result. 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, and is thus a special case of type signatures of functions. We write $t : \tau$ to assert that

[2] Actually the number will be prefixed by 'cbody' because Prolog does not allow to use numbers as functors.

t is a term of type τ. We denote by $\mathcal{B}$ the set of all typed ground terms, by $\mathcal{C} \subset \mathcal{B}$ the set of all typed constants, and by $\mathcal{F}$ the set of typed functors. Finally we introduce a (possibly empty) set of *distinguished* type signatures $\mathcal{D} \subset \mathcal{T}$ that can be useful to specify ad-hoc kernel functions on certain compound terms.

Definition 3.1 (Sum Kernels on typed terms) *The kernel between two typed terms t and s is defined inductively as follows:*

- *if $s \in \mathcal{C}$, $t \in \mathcal{C}$, $s : \tau$, $t : \tau$ then $K(s,t) = \kappa_\tau(s,t)$ 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 arities, functors, or signatures, i.e. $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$, $f : \sigma_1 \times, \ldots, \times \sigma_n \mapsto \tau'$, $g : \tau_1 \times, \ldots, \times \tau_m \mapsto \tau'$, then*
$$K(s,t) = \iota_{\tau'}(f,g) \tag{9}$$
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 type, arity, and functor, i.e. $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_n)$, and $f : \tau_1 \times, \ldots, \times \tau_n \mapsto \tau'$, then*
$$K(s,t) = \begin{cases} \kappa_{\tau_1 \times, \ldots, \times \tau_n \mapsto \tau'}(s,t) \\ \qquad \text{if } (\tau_1 \times, \ldots, \times \tau_n \mapsto \tau') \in \mathcal{D} \\ \iota_{\tau'}(f,f) + \sum_{i=1}^{n} K(s_i, t_i) \quad \text{otherwise} \end{cases} \tag{10}$$
- *in all other cases $K(s,t) = 0$.*

By replacing Equation (10) with

$$K(s,t) = \begin{cases} \kappa_{\tau_1 \times, \ldots, \times \tau_n \mapsto \tau'}(s,t) \\ \qquad \text{if } (\tau_1 \times, \ldots, \times \tau_n \mapsto \tau') \in \mathcal{D} \\ \iota_{\tau'}(f,f) \prod_{i=1}^{n} K(s_i, t_i) \quad \text{otherwise} \end{cases} \tag{11}$$

we obtain the *Product Kernel* on typed ground terms. In order to employ such kernels on proof trees, we need a typed syntax for them. We will assume the following default types for constants: `num` (numerical) and `cat` (categorical). Types for compounds terms will be either `fact`, corresponding to leaves in the proof tree, `clause` in the case of internal nodes, and `body` when containing the body of a clause. Note that regardless of the specific implementation of kernels between types, such definitions imply that we actually compare the common subpart of proofs starting from the goal (the visitor clause), and stop whenever the two proofs diverge.

A number of special cases of kernels can be implemented with appropriate choices of the kernel for compound and atomic terms. The *equivalence* kernel outputs one iff two proofs are equivalent, and zero otherwise:

$$K_{equiv}(s,t) = \begin{cases} 1 & \text{if } s \equiv t \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

We say that two proof trees s and t are equivalent iff they have the same number of nodes, and each node is equivalent to its partner in the perfect matching relation between the trees. This can be implemented using the Product Kernel in combination with binary valued kernels, such as the matching one, for kernels on constants and functors , thus implementing the notion of equivalence between individual nodes.

In many cases, we will be interested in ignoring some of the arguments of a pair of ground terms when computing the kernel between them. As an example, consider the atom bond representation in the mutagenicity benchmark, and the background knowledge in the example at the end of Section 2: the argument denoted by `E` indicates the unique identifier of a given molecule, and we would like to ignore its value when comparing two molecules together. This can be implemented using a special *ignore* type for arguments that should be ignored in comparisons, and a corresponding *constant* kernel which always outputs a constant value:

$$K_\eta(s,t) = \eta \tag{13}$$

It is straightforward to see that K_η is a valid kernel provided $\eta \geq 0$. The constant η should be set equal to the neutral value of the operation which is used to combine results for the different arguments of the term under consideration, that is $\eta = 0$ for the sum kernel and $\eta = 1$ for the product one.

The extreme use for this kernel is that of implementing the notion of *functor* equality for nodes, where two nodes are the same iff they share the same functor (and number of arguments), regardless the specific values taken by their arguments. Given two ground terms $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_n)$ the functor equality kernel is given by:

$$K_f(s,t) = \begin{cases} 0 & \text{if } type(s) \neq type(t) \\ \delta(f,g) & \text{if } s,t : \texttt{fact} \\ \delta(f,g) \star K(s_n, t_n) & \text{if } s,t : \texttt{clause} \\ K(s,t) & \text{if } s,t : \texttt{body} \end{cases} \tag{14}$$

where in the internal node case the comparison proceeds on the children, and the operator $\star$ can be either sum or product.

Moreover, it will often be useful to define custom kernels for specific terms, being them clauses or facts, by using distinguished type signatures.

4. Algorithmic Implementation

The algorithm we implemented allows for a high flexibility in customizing the behaviour to match the requirements of the specific task at hand. Four different files should be filled in order to provide the following information:

- The knowledge base describing the data.
- The background knowledge.
- The visit program to be run on the data.
- The specific implementation of kernel over proof trees, as a combination of default behaviours and possibly customized ones.

The first two files are standard in the ILP setting. The visit program is represented as a collection of clauses implementing one or more visitors, together with possible *leaf* statements aimed at pruning resulting proof trees (see the example at the end of Section 2). Note that it is not necessary to explicitly specify numeric identifiers for clauses, as the program will use the ones automatically provided by Prolog interpreters.

The kernel specification defines the way in which data and knowledge should be treated. The default way of treating compound terms can be declared to be either *sum* or *product*, by writing `compound_kernel(sum)`or `compound_kernel(product)`respectively.

The default atomic kernel is the matching one for symbols, and the product for numbers. Such behaviour can be modified by directly specifying the type signature of a given clause or fact. As an example, the following definition overrides the default kernel between *atm* terms for the mutagenicity problem:

```
type(atm(ignore,ignore,cat,cat,num)).
```

allowing to ignore identifiers for molecule and atom, and change the default behaviour for atom type (which is a number) to categorical.

Default behaviours can also be overriden by defining specific kernels for particular clauses or facts. This corresponds to specifying distinguished types together to appropriate kernels for them. Thus, the kernel between atoms could be equivalently specified by writing[3]:

```
term_kernel(atm(_,_,Xa,Xt,Xc),
            atm(_,_,Ya,Yt,Yc),K):-
  delta_kernel(Xa,Ya,Ka),
  delta_kernel(Xt,Yt,Kt),
  dot_kernel(Xc,Yc,Kc),
  K is Ka + Kt + Kc.
```

A useful kernel which can be selected is the *functor_equality* kernel as defined in Equation (14). For example, by writing

```
term_kernel(X,Y,K):-
  functor_equality_kernel(X,Y,K).
```

at the end of the configuration file it is possible to force the default behaviour for all remaining terms to functor equality, where the combination operator employed for internal nodes will be the one specified with the *compound_kernel* statement.

Finally, hyperparameters must be provided for the particular kernel machine to be run. We employed *gist-svm* [4] as it permits to separate kernel calculation from training by accepting the complete kernel matrix as input. Note that in this phase it is possible to specify kernels other than the linear one (e.g. Gaussian) on top of the visit program kernel, in order to further enlarge the feature space.

In the next section, we will provide a number of experiments showing how to customize the program to the task at hand and providing evidence of the possibilities and limitations of the proposed method.

5. Experiments

5.1. Bongard problems

In order to provide a full basic example of visit program construction, algorithm configuration and exploitation of the proof tree information, we created a very simple Bongard problem (Bongard, 1970). The concept to be learned can be represented with the simple 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 3 shows a pair of examples of such scenes with their representa-

[3] Actually, this also allows to possibly override the kernel combination operator specified by the `compound_kernel` statement.

[4] available at `http://microarray.genomecenter.columbia.edu/gist/`

tion 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 as follows:

```
                        polygon(E,X) :-
inside(E,X,Y):-           triangle(E,X).
  in(E,X,Y).
                        polygon(E,X) :-
inside(E,X,Y):-           rectangle(E,X).
  in(E,X,Z),
  inside(E,Z,Y).        polygon(E,X) :-
                          circle(E,X).
```

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(E):-
  inside(E,X,Y),polygon(E,X),polygon(E,Y).
```

Figure 4 shows the proofs trees obtained running such a visitor on the first Bongard problem in Figure 3.

A very simple kernel can be employed to solve such a task, namely an equivalence kernel with functor equality for nodewise comparison. This can be implemented with the following kernel configuration file:

```
compound_kernel(product).

term_kernel(X,Y,K):-
  functor_equality_kernel(X,Y,K).
```

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 background knowledge would not provide sufficient information for the task.

The data set was generated by creating m scenes each containing a series of n randomly chosen objects nested one into the other, and repeating the procedure for n varying from 1 to 19. 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 $n \in [1, 15]$. In each setting, positive examples where scenes containing the pattern *triangle-X^n-triangle*. We run an SVM with the above mentioned proof trees kernel and a fixed value $C = 10$ for the regulatization parameter, being the data set noise free. We evaluated its performance with a leave-one-out procedure, and compared it to *Tilde* (Blockeel & Raedt, 1997) trained from the same data and background knowledge (including the visitor).

Results are plotted in Figure 5(a) and 5(b) for $m = 10$ and $m = 50$ respectively. Both methods obtained better performance for bigger data sets, but SVM performance was very stable when increasing the nesting level corresponding to positive examples, whereas Tilde was not able to learn the concept for $n > 5$ when $m = 10$, and $n > 9$ when $m = 50$.

5.2. Strings

The possibility to plug background knowledge into the kernel allows to address problems which are notoriously hard for ILP approaches. An example of such concepts is the *M of N* one, which expects the model to be able to count and make the decision according to the result of such count.

We represented this kind of tasks with a toy problem. Examples are strings of integers $i \in [0, 9]$, and a string is positive iff more than a half of its pairs of consecutive elements is ordered, where we employ the partial ordering relation $\leq$ between numbers. In this task, M and N are example dependent, while their ratio is fixed.

As background knowledge, we introduced the concepts of length two substring and ordering between pairs of elements:

```
substr([],_):-fail.          comp(A,B):-
substr(_,[]):-fail.            A @> B.
substr([A,B],[A,B|_T]).      comp(A,B):-
substr([A,B],[_H|T]):-         A @=< B.
  substr([A,B],T).
```

while the visitor actually looks for a substring of length two in the example, and compares its elements:

```
visit(E):-
  string(E,S),substr([A,B],S),comp(A,B).

leaf(substr(_,_)).
```

Note that we state *substr* is a leaf, because we are not interested in where the substring is located within the example.

The kernel we employed for this task is a sum kernel with functor equality for nodewise comparison. This can be implemented with the following kernel configuration file:

```
compound_kernel(sum).
```

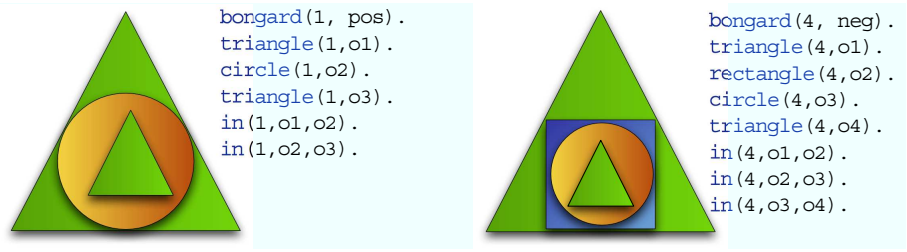

Figure 3. Graphical and Prolog facts representation of two Bongard scenes. The left and right examples are positive and negative, respectively, according to the pattern *triangle-X-triangle*.

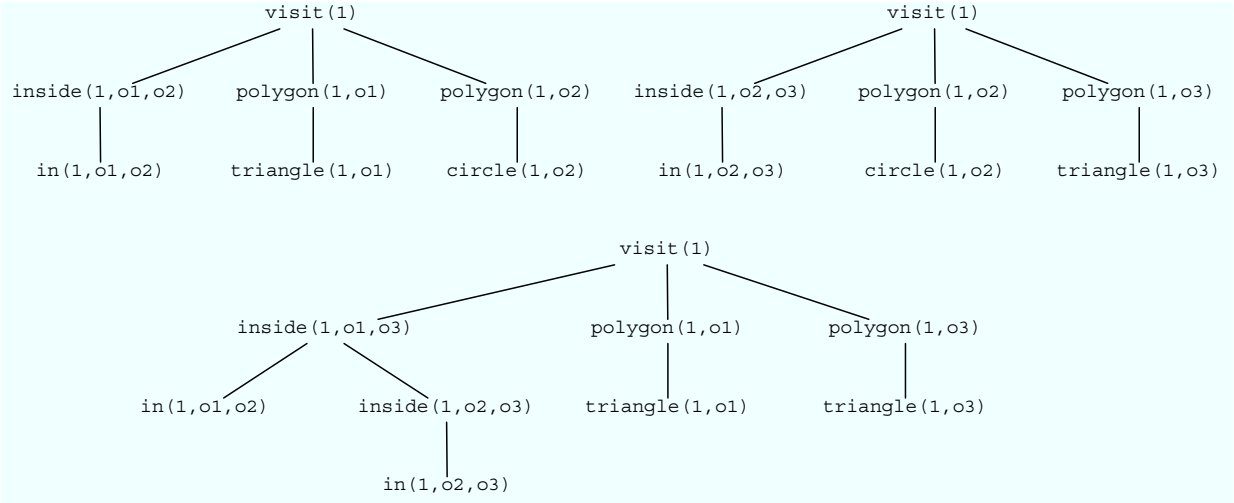

Figure 4. Proof trees obtained by running the visitor on the first Bongard problem in Figure 3.

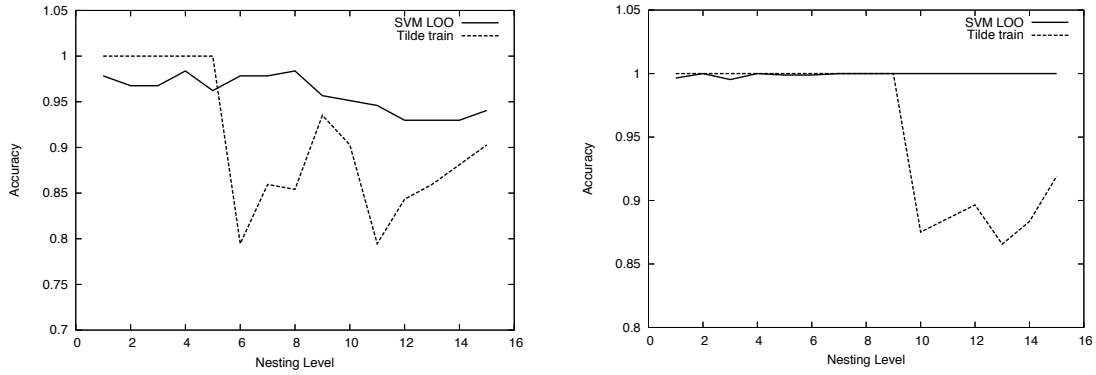

Figure 5. Comparison between SVM and Tilde in learning the *triangle-X^n-triangle* for different values of n, for data sets corresponding to $m = 10$ (left) and $m = 50$ (right).

```
term_kernel(X,Y,K):-
  functor_equality_kernel(X,Y,K).
```

The data set was created in the following way: the training set was made of 150 randomly generated lists of length 4 and 150 lists of length 5; the test set was made of 1455 randomly generated lists of length from 6 to 100. This allowed to verify the generalization performances of the algorithm for lengths very different from the ones it was trained on. The area under the ROC curve (Bradley, 1997) on the test set was equal to 1, showing that the concept had been perfectly learned by the algorithm.

5.3. Mutagenicity

The mutagenicity problem described in (Srinivasan et al., 1996) is a standard benchmark for ILP approaches. Background theory is represented as number of clauses looking for functional groups, such as benzene or anthracene, within a molecule. As a baseline we used a visitor looking for paths of different lenghts within the molecule, thus ignoring the notion of functional groups:

```
path(Drug,1,X,Y,M):-
  atm(Drug,X,_,_,_),bond(Drug,X,Y,_),
  atm(Drug,Y,_,_,_),\+ member(Y,M).

path(Drug,L,X,Y,M):-
  atm(Drug,X,_,_,_),bond(Drug,X,Z,_),
  \+ member(Z,M),L1 is L - 1,
  path(Drug,L1,Z,Y,[Z|M]).

visit1(Drug):-
  path(Drug,1,X,_,[X]).
  .
  .
visit5(Drug):-
  path(Drug,5,X,_,[X]).
```

the kernel compared atoms and bonds in corresponding positions for paths of same length:

```
compound_kernel(sum).

type(atm(ignore,ignore,cat,cat,num)).
type(bond(ignore,ignore,ignore,cat)).

term_kernel(X,Y,K):-
  functor_equality_kernel(X,Y,K).
```

A more complex notion of similarity would be to compare atoms belonging to the same type of functional group, according to the background knowledge available. This was implemented with the following set of visitors:

```
atoms(Drug,[]).

atoms(Drug,[H|T]):-
  atm(Drug,H,_,_,_),atoms(Drug,T).

visit_benzene(Drug):-
  benzene(Drug,Atoms),
  atoms(Drug,Atoms).

visit_anthracene(Drug):-
  anthracene(Drug,[Ring1,Ring2,Ring3]),
  atoms(Drug,Ring1),atoms(Drug,Ring2),
  atoms(Drug,Ring3).
  .
  .
visit_ring_size_5(Drug):-
  ring_size_5(Drug,Atoms),
  atoms(Drug,Atoms).

leaf(benzene(_,_)).
leaf(anthracene(_,_)).
  .
  .
leaf(ring_size_5(_,_)).
```

and corresponding kernel configuration:

```
compound_kernel(sum).

type(atm(ignore,ignore,cat,cat,num)).

term_kernel(X,Y,K):-
  functor_equality_kernel(X,Y,K).
```

Note that we are not interested in the way the presence of a functional group is proved, but simply on the characteristics of the atoms belonging to it. Finally, an additional source of information is given by some non structural attributes, which were included using a visitor which simply reads them

```
visit_global(Drug):-
  lumo(Drug,_Lumo),
  logp(Drug,_Logp).
```

and a kernel configuration like

```
type(lumo(ignore,num)).
type(logp(ignore,num)).
```

to be added before the last statement for the default functor equality kernel.

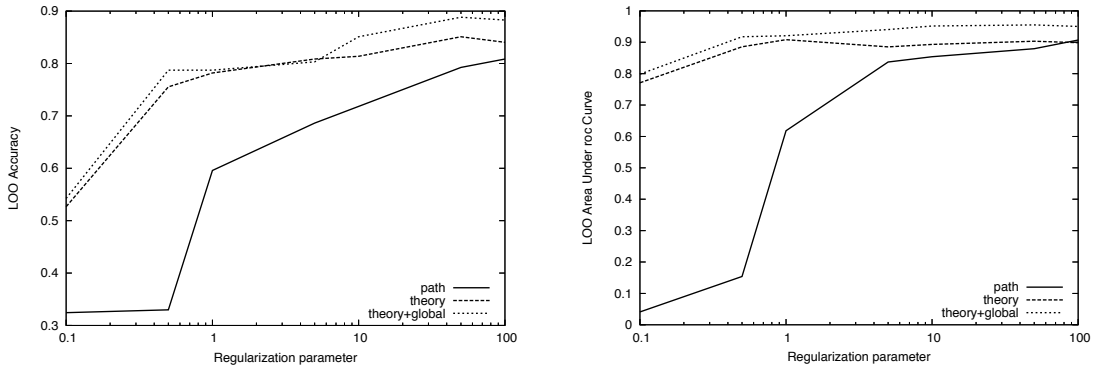

Figure 6. LOO accuracy (left) and AUC (right) for the regression friendly mutagenesis data set using different types of visitors/kernels.

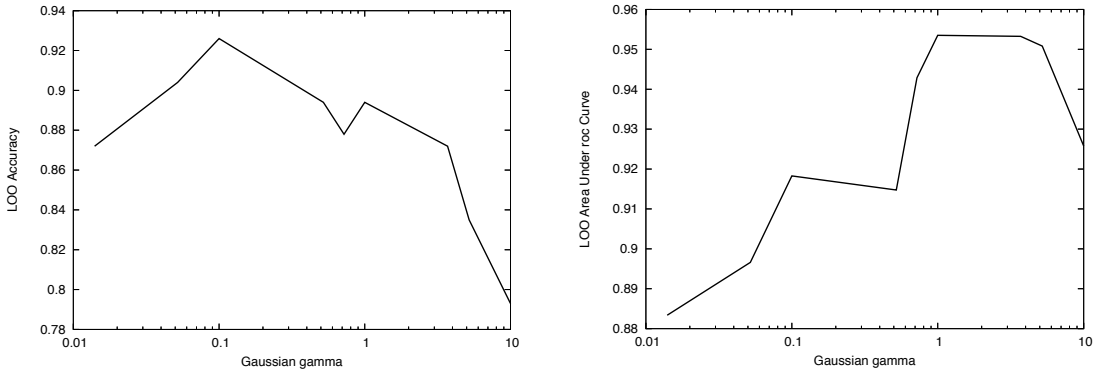

Figure 7. LOO accuracy (left) and AUC (right) for the regression friendly mutagenesis data set using the *theory+global* visitor/kernel, a Gaussian kernel on top of it, $C = 50$ and different values for the Gaussian width.

We used the regression friendly data set of 188 molecules with a LOO procedure to evaluate the methods, and both accuracy and area under the ROC curve (AUC) as performance measures. Figures 6(a) and 6(b) report LOO accuracy and AUC for different values of the regularization parameter C, for *path*, *theory* and *theory+global* visitors and corresponding kernels. Note that performances could be further improved by composing additional kernels on top of the visit program one. As an example, Figure 7(a) and 7(b) report LOO accuracy and AUC when using a Gaussian kernel on top of the *theory+global* kernel, with a fixed parameter $C = 50$ (tuned on the non composed kernel), and different values for the Gaussian width.

6. Conclusions

We have introduced the general idea of kernels over program traces and specialized it to the case of Prolog proof trees in the logic programming paradigm. The theory and the experimental results that we have obtained indicate that this method can be seen as a successful attempt to bridge several important aspects of symbolic and statistical learning, including the ability of working with relational data, the incorporation of background knowledge in a flexible and principled way, and the use of kernel methods. Besides the case of classification that has been studied in this paper, other learning tasks could benefit from the proposed framework including regression, clustering, ranking, and novelty detection. One advantage of ILP as compared to the present work is the intrinsic ability of inductive logic programming to *generate* transparent explanations of the learned function. We are currently investigating the possibility to use the kernel in guiding program synthesis or refinement, for example by learning to change the default order of Prolog resolution looking at the traces of successful and unsuccessful proofs.

Acknowledgements

This research is supported by EU Grant APrIL II (contract n° 508861). PF and AP are also partially sup-

ported by MIUR Grant 2003091149_002.

References

Biermann, A., & Krishnaswamy, R. (1976). Constructing programs from example computations. *IEEE Transactions on Software Engineering*, *2*, 141–153.

Blockeel, H., & Raedt, L. D. (1997). *Top-down induction of logical decision trees* (Technical Report CW 247). Dept. of Computer Science, K.U.Leuven.

Bongard, M. (1970). *Pattern recognition*. Spartan Books.

Bradley, A. (1997). The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recognition*, *30*, 1145–1159.

Collins, M., & Duffy, N. (2002). New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. *Proceedings of ACL 2002* (pp. 263–270). Philadelphia, PA, USA.

Cortes, C., Haffner, P., & Mohri, M. (2004). Rational kernels: Theory and algorithms. *Journal of Machine Learning Research*, *5*, 1035–1062.

Cumby, C. M., & Roth, D. (2002). Learning with feature description logics. *Proc. of ILP'02* (pp. 32–47). Springer-Verlag.

Cumby, C. M., & Roth, D. (2003). On kernel methods for relational learning. *Proc. of ICML'03*.

Gärtner, T. (2003). A survey of kernels for structured data. *SIGKDD Explor. Newsl.*, *5*, 49–58.

Gärtner, T., Lloyd, J., & Flach, P. (2004). Kernels and distances for structured data. *Machine Learning*, *57*, 205–232.

Haussler, D. (1999). *Convolution kernels on discrete structures* (Technical Report UCSC-CRL-99-10). University of California, Santa Cruz.

Jaakkola, T., & Haussler, D. (1998). Exploiting generative models in discriminative classifiers. *Proc. of NIPS*.

Kramer, S., Lavrac, N., & Flach, P. (2000). Propositionalization approaches to relational data mining. In *Relational data mining*, 262–286. SV, NY.

Leslie, C., Eskin, E., & Noble, W. (2002). The spectrum kernel: a string kernel for svm protein classification. *Proc. of the Pac. Symp. Biocomput.* (pp. 564–575).

Lloyd, J. (2003). *Logic for learning: learning comprehensible theories from structured data*. Springer-Verlag.

Lodhi, H., Shawe-Taylor, J., Cristianini, N., & Watkins, C. (2000). Text classification using string kernels. *NIPS 2000* (pp. 563–569).

Mitchell, T. M., Utgoff, P. E., & Banerj, R. (1983). Learning by experimentation: Acquiring and refining problem-solving heuristics. In *Machine learning: An artificial intelligence approach*, vol. 1. Morgan Kaufmann.

Passerini, A., & Frasconi, P. (2005). Kernels on prolog ground terms. *Int. Joint Conf. on Artificial Intelligence (IJCAI'05)*. Edinburgh.

Schölkopf, B., & Warmuth, M. (Eds.). (2003). *Kernels and regularization on graphs*, vol. 2777 of *Lecture Notes in Computer Science*. Springer.

Shapiro, E. (1983). *Algorithmic program debugging*. MIT Press.

Shawe-Taylor, J., & Cristianini, N. (2004). *Kernel methods for pattern analysis*. Cambridge University Press.

Srinivasan, A., Muggleton, S., Sternberg, M. J. E., & King, R. D. (1996). Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, *85*, 277–299.

Sterling, L., & Shapiro, E. (1994). *The art of prolog: Advanced programming techniques*. MIT Press. 2nd edition.

Vishwanathan, S., & Smola, A. (2002). Fast kernels on strings and trees. *NIPS 2002*.

Zelle, J. M., & Mooney, R. J. (1993). Combining FOIL and EBG to speed-up logic programs. *Proc. of IJCAI-93* (pp. 1106–1111).