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

Andrea Passerini and P. Frasconi
Dipartimento di Sistemi e Informatica, Università di Firenze – Italy
http://www.dsi.unifi.it/neural/

Luc De Raedt
Institute for Computer Science, Albert-Ludwigs Universität, Freiburg – Germany
http://www.informatik.uni-freiburg.de/~ml/
Combining Symbolic and Statistical Learning

- Symbolic approaches allow to express domain knowledge in a very natural and expressive way.
- Statistical learning is very effective for high dimensional and noisy data and complex hypotheses.
- Desirable to combine advantages of the two approaches
Domain Knowledge as Trace of Programs

- Background knowledge can be encoded in programs to be run on examples:
  - A finite state automata accepting strings with a certain set of characteristics.
  - A relational database and a set of interfacing queries.
  - A set of clauses in first order logic.
  - In principle any Turing machine executed on the data.
- The trace of a program contains information on the reasons for a certain output.
- Several approaches exist that learn from example-traces (Bierman76, Mitchell83, Shapiro83, Zelle93).
Using Trace of Programs in Kernel Machines

- Learning algorithm accepting traces can be fed with all possible background knowledge.
- Kernel machine algorithms all rely on a similarity measure (kernel).
- e.g. Kernel machines for binary classification:

\[ f(x) = \sum_{i=1}^{N} y_i \alpha_i K(x, x_i) \]

- Similar examples should produce similar traces.
- Kernels between program traces \(\Rightarrow\) plugging background knowledge into statistical learning algorithms.
Kernel Machines in the ILP Setting

- Data are represented as a collection of facts.
- Background knowledge is represented as a collection of clauses.
- A program is represented as one or more special visitor predicates used to probe examples.
- The Kernel measures the similarity between examples as similarity between visit traces.
  - Full trace of steps passed for proving a goal contains more information than simply the success/failure of the goal.
  - The representation of instances is enriched by background knowledge.
Example: a simple molecule

\[
\begin{align*}
\text{atm}(d26, d26_1, c, 22, -0.093). & \quad \text{bond}(d26, d26_1, d26_2, 7). \\
\text{atm}(d26, d26_2, c, 22, -0.093). & \quad \text{bond}(d26, d26_2, d26_3, 7). \\
\text{atm}(d26, d26_3, c, 22, -0.093). & \quad \text{bond}(d26, d26_3, d26_4, 7). \\
\text{atm}(d26, d26_4, c, 22, -0.093). & \quad \text{bond}(d26, d26_4, d26_5, 7). \\
\text{atm}(d26, d26_5, c, 22, -0.093). & \quad \text{bond}(d26, d26_5, d26_6, 7). \\
\text{atm}(d26, d26_6, c, 22, -0.093). & \quad \text{bond}(d26, d26_6, d26_1, 7). \\
\text{atm}(d26, d26_7, h, 3, 0.167). & \quad \text{bond}(d26, d26_1, d26_7, 1). \\
\text{atm}(d26, d26_8, h, 3, 0.167). & \quad \text{bond}(d26, d26_3, d26_8, 1). \\
\text{atm}(d26, d26_9, h, 3, 0.167). & \quad \text{bond}(d26, d26_6, d26_9, 1). \\
\text{atm}(d26, d26_{10}, c1, 93, -0.163). & \quad \text{bond}(d26, d26_{10}, d26_5, 1). \\
\text{atm}(d26, d26_{11}, n, 38, 0.836). & \quad \text{bond}(d26, d26_4, d26_{11}, 1). \\
\text{atm}(d26, d26_{12}, n, 38, 0.836). & \quad \text{bond}(d26, d26_2, d26_{12}, 1). \\
\text{atm}(d26, d26_{13}, o, 40, -0.363). & \quad \text{bond}(d26, d26_{13}, d26_{11}, 2). \\
\text{atm}(d26, d26_{14}, o, 40, -0.363). & \quad \text{bond}(d26, d26_{11}, d26_{14}, 2). \\
\text{atm}(d26, d26_{15}, o, 40, -0.363). & \quad \text{bond}(d26, d26_{15}, d26_{12}, 2). \\
\text{atm}(d26, d26_{16}, o, 40, -0.363). & \quad \text{bond}(d26, d26_{12}, d26_{16}, 2).
\end{align*}
\]
Example: Inserting Background Knowledge

- Background knowledge

1 : \texttt{cycle}(E,X):- 
   \texttt{path}(E,X,Y,[X]), 
   \texttt{bond}(E,Y,X,\_).

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

3 : \texttt{path}(E,X,Y,M):- 
   \texttt{atm}(E,X,\_,\_,\_,\_), 
   \texttt{bond}(E,X,Z,\_), 
   \texttt{\_+ member}(Z,M), 
   \texttt{\_+ member}(Y,M), 
   \texttt{path}(E,Z,Y,[Z\mid M]).

- Visitor

4 : \texttt{visit}(E): 
   \texttt{cycle}(E,X).
SLD-Resolution Tree

- Represents the full trace of the Prolog execution in trying to satisfy a given goal.
- Contains all the information leading to the proof (or failure) of the goal, with successful as well as unsuccessful tries.
- It’s structure is strongly dependent on the specific search strategy used by Prolog to prove goals.
- It can be very long and complex.
**Proof Tree**

- Represents only the successful path leading to the goal proof.
- Is empty if the goal cannot be proved.
- Contains only the subset of information contributing positively to the (eventual) success of the goal.
- It is like representing an object by the ‘attributes’ it owns, while ignoring the absence of the others.
- AND Tree as a compact representation of the Proof Tree.
Example: Proof Tree of visit over the molecule

4: visit(d26)

1: cycle(d26, d26_1)

3: path(d26, d26_1, d26_6, [d26_1])

atm(d26, d26_1, c, 22, -0.093)

bond(d26, d26_1, d26_2, 7)

\+member(d26_2, [d26_1])

bond(s26, d26_6, d26_1, 7)

3: path(d26, d26_2, d26_6, [d26_2, d26_1])

•

•

•
Proof Trees Representation

- A goal can often be satisfied in a number of alternative ways.
- A visit predicate generates a (possibly empty) set of proof trees.
- 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.
Proof Tree Pruning

- Not all proofs contain valuable information apart from their success/failure.
- Proof Trees can be very big for complex proofs (e.g. functional groups in molecules)
- It is possible to explicitly ask to ignore the proof for particular predicates.

leaf(benzene(_,_)).
Kernels over Visit Programs

- A Visit program made of \( n \geq 1 \) visitors run on instance \( x \) outputs:

\[
P(x) = [P_1(x), \ldots, P_n(x)] \quad \text{where} \quad P_i(x) = \{t_{i,1}(x), \ldots, t_{i,m_i}(x)(x)\}
\]

- The kernel between instances is defined as:

\[
K(x, z) = K_P(P(x), P(z)) = \sum_{i=1}^{n} K_i(P_i(x), P_i(z))
\]

- We do not compare proofs resulting from different visitors.
- We allow to specify different kernels for different visitors.
Kernels over Visitors

- Each visitor $v_i$ produces a (possibly empty) set of Proof trees:

$$P_i(x) = \{t_{i,1}(x), \ldots, t_{i,m_i}(x)\}$$

- We can use the definition of Kernel between sets:

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

- Kernels will be typically normalized at different levels.
Proof Tree as a Single Prolog Ground Term

\[
\text{visit}(d26, \text{cbody4}(\text{cycle}(d26, d26_1, \\
\text{cbody1}(\text{path}(d26, d26_1, d26_6, [d26_1], \\
\text{cbody3}(\ldots)), \text{bond}(d26, d26_6, d26_1, 7))))).
\]

4 : visit(d26)

1 : cycle(d26, d26_1)

3 : path(d26, d26_1, d26_6, [d26_1])

\bullet

\bullet

\bullet

\bullet

bond(s26, d26_6, d26_1, 7)
Examples of default type signatures in Proof Trees

- \( \text{cat} \times \text{cat} \times \text{body} \rightarrow \text{clause} \)
  
  \( \text{cycle}(d26, d26_1, \text{cbody1}(...)) \)

- \( \text{cat} \times \text{cat} \times \text{cat} \times \text{num} \times \text{num} \rightarrow \text{fact} \)
  
  \( \text{atm}(d26, d26_5, c, 22, -0.093) \)
Kernel between Typed Ground Terms (1)

- if \( s \in C, \ t \in C, \ s : \tau, \ t : \tau \) then

\[
K(s, t) = \kappa_\tau(s, t)
\]

where \( \kappa_\tau : C \times C \rightarrow \mathbb{R} \) is a valid kernel on constants of type \( \tau \);

- else if \( s = f(s_1, \ldots, s_n) \) and \( t = g(t_1, \ldots, t_m) \) are compound terms that have the same type \( \tau' \) but different arities, functors, or signatures, then

\[
K(s, t) = \iota_{\tau'}(f, g)
\]

where \( \iota_{\tau'} : \mathcal{F} \times \mathcal{F} \rightarrow \mathbb{R} \) is a valid kernel on functors that construct terms of type \( \tau' \)
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

$$K(s, t) = \begin{cases} 
\kappa_{\tau_1 \times, \ldots, \times \tau_n \mapsto \tau'}(s, t) & \text{if } (\tau_1 \times, \ldots, \times \tau_n \mapsto \tau') \in \mathcal{D} \\
\nu_{\tau'}(f, f) + \sum_{i=1}^{n} K(s_i, t_i) & \text{otherwise}
\end{cases}$$

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

• Sum can always be replaced by product.
Functor equality Kernel

- **functor** equality for clauses. Given two compound ground terms \( s = f(s_1, \ldots, s_n) \) and \( t = g(t_1, \ldots, t_n) \):

\[
K_f(s, t) = \begin{cases} 
0 & \text{if } \text{type}(s) \neq \text{type}(t) \\
\delta(f, g) & \text{if } s, t : \text{fact} \\
\delta(f, g) \ast K(s_n, t_n) & \text{if } s, t : \text{clause} \\
K(s, t) & \text{if } s, t : \text{body}
\end{cases}
\]

- The operator \( \ast \) can be either sum or product.
Algorithmic Implementation: input files

- A collection of facts describing the data.
- A collection of clauses describing the background knowledge.
- A collection of clauses describing one or more visitor predicates (plus possible leaf statements).
- A collection of clauses specifying the kernel configuration.
**Bongard: triangle-$$X^n$$-triangle**

```
bongard(1, pos).
triangle(1,o1).
circle(1,o2).
triangle(1,o3).
in(1,o1,o2).
in(1,o2,o3).

bongard(4, neg).
triangle(4,o1).
rectangle(4,o2).
circle(4,o3).
triangle(4,o4).
in(4,o1,o2).
in(4,o2,o3).
in(4,o3,o4).
```
**Bongard: Inserting Background Knowledge**

- **Background knowledge**

  \[
  \text{inside}(E,X,Y) :- \quad \text{in}(E,X,Y). \quad \% \text{clause nr 1}
  \]

  \[
  \text{inside}(E,X,Y) :- \quad \% \text{clause nr 2}
  
  \quad \text{in}(E,X,Z), \quad \text{inside}(E,Z,Y).
  \]

  \[
  \text{polygon}(E,X) :- \quad \text{triangle}(E,X). \quad \% \text{clause nr 3}
  \]

  \[
  \text{polygon}(E,X) :- \quad \text{rectangle}(E,X). \quad \% \text{clause nr 4}
  \]

  \[
  \text{polygon}(E,X) :- \quad \text{circle}(E,X). \quad \% \text{clause nr 5}
  \]

- **Visitor**

  \[
  \text{visit}(E) :- \quad \% \text{clause nr 6}
  
  \quad \text{inside}(E,X,Y), \text{polygon}(E,X), \text{polygon}(E,Y).
  \]
Bongard: Proof Trees for visit on Bongard 1

1 : inside(1,o1,o2)
   
   in(1,o1,o2)
   
   triangle(1,o1) circle(1,o2)

3 : polygon(1,o1)

5 : polygon(1,o2)
   
   in(1,o2,o3)
   
   circle(1,o2) triangle(1,o3)

6 : visit(1)

2 : inside(1,o1,o3)
   
   in(1,o1,o2) 1 : inside(1,o2,o3) triangle(1,o1) triangle(1,o3)

3 : polygon(1,o1)

3 : polygon(1,o3)

5 : polygon(1,o2)

6 : visit(1)
Bongard: Proof Trees Kernel

compound_kernel(product).

term_kernel(X,Y,K):-
    functor_equality_kernel(X,Y,K).
Comparison between SVM (loo error) and Tilde (empirical error) in learning the $triangle-X^n-triangle$ for different values of $n$, for datasets corresponding to $m = 10$ (left) and $m = 50$ (right).
Ordering: Detect if a List is Mostly Ordered

- Each example is a list of integers \( i \in [0, 9] \).

\[
\text{string}(4-146, \ [4 \ 3 \ 8 \ 8]).
\]

- Two consecutive elements \( i, j \) are ordered iff \( i \leq j \).

- A list is positive iff more than a half of its pairs of consecutive elements is ordered.

- Kind of \( M \ of \ N \) problem.

- Training set is made of 150 randomly generated lists of length 4 and 150 lists of length 5.

- Test set is made of 1455 randomly generated lists of length from 6 to 100.
Ordering: Inserting Background Knowledge

- Background knowledge

  \[\text{substr}([A,B], [A,B|_T]).\]
  \[\text{substr}([A,B], [_H|T]):=\]
  \[\text{substr}([A,B], T).\]

  \[\text{comp}(A,B):= A @> B.\]
  \[\text{comp}(A,B):= A @=< B.\]

- Visitor

  \[\text{visit}(E):=\]
  \[\text{string}(E,S), \text{substr}([A,B], S), \text{comp}(A,B).\]

  \[\text{leaf}(	ext{substr}(\_,\_)).\]
**Ordering: Proof Trees Kernel**

compound_kernel(sum).

term_kernel(X,Y,K):-
    functor_equality_kernel(X,Y,K).
Ordering: Experimental Results

- SVM obtains an area under the ROC curve equal to 1.
- Tilde was totally unable to learn the concept.
Mutagenesis: Learning Task

- Data are 188 small aromatic molecules
- atom bond representation
- Classify each molecule as either mutagenic or not
- Binary classification task
Mutagenesis: Inserting Background Knowledge (1)

- Background knowledge

\[
\text{path}(\text{Drug}, 1, X, Y, M) : - \\
\quad \text{atm}(\text{Drug}, X, \_, \_, \_), \text{bond}(\text{Drug}, X, Y, \_), \\
\quad \text{atm}(\text{Drug}, Y, \_, \_, \_), \text{\texttt{\textbf{\texttt{\texttt{\texttt{\texttt{\texttt{\texttt{\texttt{\texttt{\texttt{- memb}}}}}}}}}}}}er (Y, M).
\]

\[
\text{path}(\text{Drug}, L, X, Y, M) : - \\
\quad \text{atm}(\text{Drug}, X, \_, \_, \_), \text{bond}(\text{Drug}, X, Z, \_), \\
\quad \text{\texttt{\texttt{\texttt{\texttt{\texttt{\texttt{\texttt{\texttt{\texttt{\texttt{\texttt{- memb}}}}}}}}}}}}er (Z, M), L1 \text{ is } L - 1, \\
\quad \text{path}(\text{Drug}, L1, Z, Y, [Z | M]).
\]

- Visitors

\[
\text{visit1}(\text{Drug}) : - ... \quad \text{visit5}(\text{Drug}) : - \\
\quad \text{path}(\text{Drug}, 1, X, \_, [X]). \quad \text{path}(\text{Drug}, 5, X, \_, [X]).
\]
compound_kernel(sum).

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

term_kernel(X,Y,K):-
    functor_equality_kernel(X,Y,K).
Mutagenesis: Inserting Background Knowledge (2)

• Background knowledge

\[\text{atoms}(\text{Drug},[]). \quad \text{consult('ring\_theory.pl')}\].
\[\text{atoms}(\text{Drug},[\text{H}|\text{T}]):-\]
\[\text{atm}(\text{Drug},\text{H},\_\_\_\_\_),\text{atoms}(\text{Drug},\text{T})].\]

• Visitors

\[\text{visit\_anthracene}(\text{Drug}):-\]
\[\text{anthracene}(\text{Drug},[\text{Ring1},\text{Ring2},\text{Ring3}]),\]
\[\text{atoms}(\text{Drug},\text{Ring1}),\]
\[\text{atoms}(\text{Drug},\text{Ring2}),\]
\[\text{atoms}(\text{Drug},\text{Ring3}).\]
\[\text{visit\_benzene}(\text{Drug}):-\]
\[\text{benzene}(\text{Drug},\text{Atoms}),\]
\[\text{atoms}(\text{Drug},\text{Atoms}).\]

leaf(anthracene(_,_)).
leaf(benzene(_,_)).
........
compound_kernel(sum).

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

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

• Adding global attributes

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

type(lumo(ignore,num)).
type(logp(ignore,num)).
Mutagenesis: Experimental Results

![Graph showing LOO Accuracy vs Regularization parameter and Gaussian gamma]
SCOP: protein fold classification task

- Classifying proteins into SCOP folds.
- 20 binary classification tasks (Turcotte:MLJ01).
- Proteins represented by their secondary structure segments, encoded in both propositional and relational form.
\textbf{SCOP: Writing visitors}

\begin{verbatim}
visit_global(X):-
    normlen(X,Len),
    normnb_alpha(X,NumAlpha),
    normnb_beta(X,NumBeta).

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

unit_features(A):-
    normsst(A,B,C,D,E,F,G,H,I,J,K),
    (has_pro(A); \+ has_pro(A)).

leaf(adjacent(_,_,_,_,_,_,_)).
leaf(normcoilm(_,_,_,_)).

visit_unit(X):-
    sec_struc(X,A),
    unit_features(A).
\end{verbatim}
compound_kernel(sum).

type(normlen(ignore,num)).
type(normnb_alpha(ignore,num)).
type(normnb_beta(ignore,num)).
type(normsst(ignore,ignore,ignore,ignore,ignore,
    ignore,num,ignore,num,num,num,ignore)).
type(adjacent(ignore,ignore,ignore,ignore,cat,cat,cat)).
type(normcoil(ignore,ignore,num)).

term_kernel(X,Y,K):-
    functor_equality_kernel(X,Y,K).
## SCOP: Experimental Results (1)

<table>
<thead>
<tr>
<th>Fold</th>
<th>Progol</th>
<th>SVM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>All-α</strong>:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hungin-like</td>
<td>95.1</td>
<td>94.9</td>
</tr>
<tr>
<td>DNA-binding 3-helical bundle</td>
<td>83.0</td>
<td>88.9</td>
</tr>
<tr>
<td>4-helical cytokines</td>
<td>70.7</td>
<td>86.7</td>
</tr>
<tr>
<td>Lambda repressor-like DNA-binding domains</td>
<td>73.4</td>
<td>83.3</td>
</tr>
<tr>
<td>EF Hand-like</td>
<td>77.6</td>
<td>85.7</td>
</tr>
<tr>
<td><strong>All-β</strong>:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Immunoglobulin-like beta-sandwich</td>
<td>76.3</td>
<td>85.2</td>
</tr>
<tr>
<td>SH3-like barrel</td>
<td>91.4</td>
<td>93.7</td>
</tr>
<tr>
<td>OB-fold</td>
<td>78.4</td>
<td>83.3</td>
</tr>
<tr>
<td>Trypsin-like serine proteases</td>
<td>93.1</td>
<td>93.6</td>
</tr>
<tr>
<td>Lipocalins</td>
<td>88.3</td>
<td>92.9</td>
</tr>
</tbody>
</table>
## SCOP: Experimental Results (2)

<table>
<thead>
<tr>
<th>Fold</th>
<th>Progol</th>
<th>SVM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>α/β:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beta/alpha (TIM)-barrel</td>
<td>70.7</td>
<td>73.3</td>
</tr>
<tr>
<td>NAD(P)-binding Rossmann-fold domains</td>
<td>71.6</td>
<td>84.1</td>
</tr>
<tr>
<td>P-loop containing nucleotide triphosphate hydrolases</td>
<td>76.0</td>
<td>76.2</td>
</tr>
<tr>
<td>alpha/beta-Hydrolases</td>
<td>72.2</td>
<td>86.1</td>
</tr>
<tr>
<td>Periplasmic binding protein-like II</td>
<td>68.9</td>
<td>79.5</td>
</tr>
<tr>
<td><strong>α + β:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interleukin 8-like chemokines</td>
<td>92.9</td>
<td>96.3</td>
</tr>
<tr>
<td>beta-Grasp</td>
<td>71.7</td>
<td>88.9</td>
</tr>
<tr>
<td>Ferredoxin-like</td>
<td>83.1</td>
<td>76.9</td>
</tr>
<tr>
<td>Zincin-like</td>
<td>64.3</td>
<td>79.5</td>
</tr>
<tr>
<td>SH2-like</td>
<td>76.8</td>
<td>66.7</td>
</tr>
<tr>
<td><strong>Micro average:</strong></td>
<td>78.3</td>
<td>83.6</td>
</tr>
<tr>
<td></td>
<td>±2.4</td>
<td>±2.2</td>
</tr>
</tbody>
</table>
**Summing Up**

- Proof tree kernels allow to naturally include background knowledge and relational features in kernel machine algorithms.

- They can successfully solve problems requiring counting (e.g. M of N problems), which are difficult for classic ILP methods.

- They take the advantages of statistical learning when dealing with real world data (e.g. SCOP classification).

- Interfacing to traditional ILP data and BK boils down to:
  - writing the *visitor* program
  - choosing/developing the appropriate kernel for its trace.
Possible extensions

- Apply kernel to other learning tasks (regression, clustering, ranking, novelty detection).
- Extend visitor to manage probabilistic proofs.
- use the kernel in guiding program synthesis or refinement:
  - change Prolog resolution order by looking at proofs.
  - Proof tree given by ILP algorithm \(\Rightarrow\) classifier refinement/regularization/rejection.
  - program verifier \(\Rightarrow\) verify if certain program output (ex. sorted string) is correct based on the program trace.