Lecture 13: Inductive Program Synthesis
Cognitive Systems - Machine Learning

Part III: Learning Programs and Strategies

Inductive Programming, Functional Programs, Program Schemes, Traces, Folding

last change January 14, 2015
Outlook

- Learning complex rules over symbolic structures
- Program Synthesis
- Inductive Programming: Summers’ Thesys
- Igor2
- Application of Igor2 to Cognitive Problems
Knowledge Level Learning

- opposed to low-level (statistical) learning
- learning as generalization of symbol structures (rules) from experience
- “white-box” learning: learned hypotheses are verbalizable, can be inspected, communicated

Examples:

**Classification/Concepts:**

- IF odor=almond THEN poisonous
- IF cap-shape=conical & gill-color=grey THEN poisonous

**Recursive Concepts:**

- ancestor(X,Y) :- parent(X,Y).
- ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).

**Simple action rules:**

- IF distance-to-object < threshold THEN STOP

**Recursive action rules:**

- e.g., Tower of Hanoi (to be seen later)
Approaches to Symbolical Learning

Machine Learning Approaches
- Grammar Inference
- Decision Tree Algorithms
- Inductive Logic Programming
- Evolutionary Programming
- Inductive (functional) Programming

Inductive Programming
- Very special branch of machine learning
- Learning programs from *incomplete* specifications, typically I/O examples or constraints
Program Synthesis

Automatic Programming

- Let the computer program itself
- Automatic code generation from (non-executable) specifications very high level programming
- Not intended for software development in the large but for semi-automated synthesis of functions, modules, program parts
Approaches to Program Synthesis

Deductive and transformational program synthesis

- Complete formal specifications (vertical program synthesis)
- e.g. KIDS (D. Smith)
- High level of formal education is needed to write specifications
- Tedious work to provide the necessary axioms (domain, types, ...)
- Very complex search spaces

\[ \forall x \exists y \ p(x) \rightarrow q(x, y) \]
\[ \forall x \ p(x) \rightarrow q(x, f(x)) \]

Example

\( \text{last}(l) \iff \text{find } z \text{ such that for some } y, \ l = y \circ [z] \text{ where islist}(l) \text{ and } l \neq [ ] \) (Manna & Waldinger)
Approaches to Program Synthesis

Inductive program synthesis

- Very special branch of machine learning
- Learning programs from *incomplete* specifications, typically I/O examples or constraints
- Inductive programming (IP) for short

(Flener & Schmid, AI Review, 29(1), 2009; Encyclopedia of Machine Learning, 2010; Schmid, Kitzelmann & Plasmeijr, AAIP 2009)
**IP – Contributing Areas**

- **Artificial Intelligence**
  - Modeling human programming knowledge, skills, strategies

- **Inductive Programming**
  - Incomplete specifications
  - Code generation from Inductive Programming

- **Software Engineering**
  - Automated code generation
  - Example-driven programming

- **Machine Learning**
  - Programs from Examples

- **Functional/Declarative Programming**
  - Code generation from incomplete specifications

Inductive Programming Example

Learning \texttt{last}

I/O Examples

\begin{align*}
\text{last } [a] & = a \\
\text{last } [a, b] & = b \\
\text{last } [a, b, c] & = c \\
\text{last } [a, b, c, d] & = d \\
\end{align*}

Generalized Program

\begin{align*}
\text{last } [x] & = x \\
\text{last } (x:xs) & = \text{last } xs \\
\end{align*}

Some Syntax

\begin{itemize}
\item \texttt{-- sugared} \texttt{[1,2,3,4]}
\item \texttt{-- normal infix} \texttt{(1:2:3:4:[])}
\item \texttt{-- normal prefix} \texttt{((:) 1}
\begin{itemize}
\item \texttt{((:) 2}
\begin{itemize}
\item \texttt{((:) 3}
\begin{itemize}
\item \texttt{((:) 4}
\item \texttt{[]])})
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
Inductive Programming – Basics

IP is search in a class of programs (hypothesis space)

Program Class characterized by:

Syntactic building blocks:

- **Primitives**, usually data constructors
- **Background Knowledge**, additional, problem specific, user defined functions
- **Additional Functions**, automatically generated

Restriction Bias

- syntactic restrictions of programs in a given language

Result influenced by:

Preference Bias

- choice between syntactically different hypotheses
Inductive Programming – Approaches

- Typical for declarative languages (LISP, PROLOG, ML, HASKELL)
- Goal: finding a program which covers all input/output examples correctly (no PAC learning) and (recursively) generalizes over them
- Two main approaches:
  - **Analytical, data-driven:**
    detect regularities in the I/O examples (or traces generated from them) and generalize over them (folding)
  - **Generate-and-test:**
    generate syntactically correct (partial) programs, examples only used for testing
Inductive Programming – Approaches

Generate-and-test approaches

- ILP (90ies): FFOIL (Quinlan) (sequential covering)
- evolutionary: ADATE (Olsson)
- enumerative: MAGICHASKEILLER (Katayama)
- also in functional/generic programming context: automated generation of instances for data types in the model-based test tool G∀st (Koopmann & Plasmeijer)
Analytical Approaches

- Classical work (70ies–80ies):
  THESYS (Summers), Biermann, Kodratoff
  learn linear recursive Lisp programs from traces

- ILP (90ies):
  Golem, Progol (Muggleton), Dialogs (Flener)
  inverse resolution, θ-subsumption, schema-guided

- IGOR1 (Schmid, Kitzelmann; extension of THESYS)
  IGOR2 (Kitzelmann, Hofmann, Schmid)
Summers’ Thesys

Summers (1977), A methodology for LISP program construction from examples, Journal ACM

Two Step Approach

- Step 1: Generate traces from I/O examples
- Step 2: Fold traces into recursion

Generate Traces

- Restriction of input and output to nested lists
- Background Knowledge:
  - Partial order over lists
  - Primitives: `atom`, `cons`, `car`, `cdr`, `nil`

Rewriting algorithm with unique result for each I/O pair: characterize `I` by its structure (lhs), represent `O` by expression over `I` (rhs)

→ restriction of synthesis to structural problems over lists (abstraction over elements of a list) not possible to induce `member` or `sort`
Example: Rewrite to Traces

I/O Examples

- nil → nil
- (A) → ((A))
- (A B) → ((A) (B))
- (A B C) → ((A) (B) (C))

Traces

\[ F_L(x) \leftarrow (\text{atom}(x) \rightarrow \text{nil}, \right. \]
\[ \text{atom}(\text{cdr}(x)) \rightarrow \text{cons}(x, \text{nil}), \]
\[ \text{atom}(\text{cddr}(x)) \rightarrow \text{cons}(\text{cons}(\text{car}(x), \text{nil}), \text{cons}(\text{cdr}(x), \text{nil})), \]
\[ T \rightarrow \text{cons}(\text{cons}(\text{car}(x), \text{nil}), \text{cons}(\text{cons}(\text{cadr}(x), \text{nil}), \text{cons}(\text{cddr}(x), \text{nil})))) \]
Example: Deriving Fragments

Unique Expressions for Fragment \((A \ B)\)

\[(x, (A \ B)), \]
\[(\text{car}[x], A), \]
\[(\text{cdr}[x], (B)), \]
\[(\text{cadr}[x], B), \]
\[(\text{cddr}[x], ( ))\]

Combining Expressions

\[((A) (B)) = \text{cons}[(A); ((B))] = \text{cons}[	ext{cons}[A, ()];\text{cons}[(B), ( )]].\]

Replacing Values by Functions

\[\text{cons}[	ext{cons}(\text{car}[x]; ( ));\text{cons}(\text{cdr}[x]; ( ))]\]
Folding of Traces

- Based on a program scheme for linear recursion (restriction bias)
- Synthesis theorem as justification
- Idea: inverse of fixpoint theorem for linear recursion
- Traces are $k$th unfolding of an unknown program following the program scheme
- Identify differences, detect recurrence

$$F(x) \leftarrow (p_1(x) \rightarrow f_1(x), \ldots, p_k(x) \rightarrow f_k(x), T \rightarrow C(F(b(x)), x))$$
Example: Fold Traces

kth unfolding

\[ F_L(x) \leftarrow \begin{align*}
\text{atom}(x) & \rightarrow \text{nil,} \\
\text{atom}(\text{cdr}(x)) & \rightarrow \text{cons}(x, \text{nil}), \\
\text{atom}(\text{cddr}(x)) & \rightarrow \text{cons}(\text{cons}(\text{car}(x), \text{nil}), \text{cons}(\text{cdr}(x), \text{nil})), \\
T & \rightarrow \text{cons}(\text{cons}(\text{car}(x), \text{nil}), \text{cons}(\text{cons}(\text{cad}(x), \text{nil}), \text{cons}(\text{cddr}(x), \text{nil}))))
\end{align*}\]

Differences:

\[
\begin{align*}
p_2(x) &= p_1(\text{cdr}(x)) \\
p_3(x) &= p_2(\text{cdr}(x)) \\
p_4(x) &= p_3(\text{cdr}(x))
\end{align*}
\]

Recurrence Relations:

\[
\begin{align*}
p_1(x) &= \text{atom}(x) \\
p_{k+1}(x) &= p_k(\text{cdr}(x)) \text{ for } k = 1, 2, 3
\end{align*}
\]

\[
\begin{align*}
f_2(x) &= \text{cons}(x, f_1(x)) \\
f_3(x) &= \text{cons}(\text{cons}(\text{car}(x), \text{nil}), f_2(\text{cdr}(x)))) \\
f_4(x) &= \text{cons}(\text{cons}(\text{car}(x), \text{nil}), f_3(\text{cdr}(x))))
\end{align*}
\]

\[
\begin{align*}
f_1(x) &= \text{nil} \\
f_2(x) &= \text{cons}(x, f_1(x)) \\
f_{k+1}(x) &= \text{cons}(\text{cons}(\text{car}(x), \text{nil}), f_k(\text{cdr}(x))) \text{ for } k = 2, 3
\end{align*}
\]
Example: Fold Traces

**kth unfolding**

\[ F_L(x) \leftarrow \begin{cases} 
\text{atom}(x) \rightarrow \text{nil}, \\
\text{atom}(\text{cdr}(x)) \rightarrow \text{cons}(x, \text{nil}), \\
\text{atom}(\text{cddr}(x)) \rightarrow \text{cons}(\text{cons}(\text{car}(x), \text{nil}), \text{cons}(\text{cdr}(x), \text{nil})), \\
T \rightarrow \text{cons}(\text{cons}(\text{car}(x), \text{nil}), \text{cons}(\text{cons}(\text{cadr}(x), \text{nil}), \text{cons}(\text{cddr}(x), \text{nil}))))
\end{cases} \]

**Folded Program**

\[ \text{unpack}(x) \leftarrow \begin{cases} 
\text{atom}(x) \rightarrow \text{nil}, \\
T \rightarrow \text{u}(x)
\end{cases} \]

\[ \text{u}(x) \leftarrow \begin{cases} 
\text{atom}(\text{cdr}(x)) \rightarrow \text{cons}(x, \text{nil}), \\
T \rightarrow \text{cons}(\text{cons}(\text{car}(x), \text{nil}), \text{u}(\text{cdr}(x))))
\end{cases} \]
Summers’ Synthesis Theorem

- Based on fixpoint theory of functional program language semantics.
  (Kleene sequence of function approximations: a partial order can be defined over the approximations, there exists a supremum, i.e. least fixpoint)

- Idea: If we assume that a given trace is the $k$-th unfolding of an unknown linear recursive function, than there must be regular differences which constitute the stepwise unfoldings and in consequence, the trace can be generalized (folded) into a recursive function
Illustration of Kleene Sequence

Defined for no input

\[ U^0 \leftarrow \Omega \]

Defined for empty list

\[ U^1 \leftarrow (\text{atom}(x) \rightarrow \text{nil}, \quad T \rightarrow \Omega) \]

Defined for empty list and lists with one element

\[ U^2 \leftarrow (\text{atom}(x) \rightarrow \text{nil}, \quad \text{atom}(\text{cdr}(x)) \rightarrow \text{cons}(x, \text{nil}), \quad T \rightarrow \Omega) \]

\[ \ldots \text{Defined for lists up to } n \text{ elements} \]
The IP System IGOR2

Inductive
- induces programs from I/O examples
- inspired by Summers’ THESYS system
- successor of IGOR1 (Schmid, 2001)

Analytical
- data-driven
- finds recursive generalization by analyzing I/O examples
- integrates best first search

Functional
- learns functional programs
- first prototype in MAUDE by Emanuel Kitzelmann
- in HASKELL and extended (general fold) by M. Hofmann
Example

reverse

I/O Example

reverse \[\] = \[\] reverse \[a,b\] = \[b,a\]
reverse \[a\] = \[a\] reverse \[a,b,c\] = \[c,b,a\]

Generalized Program

reverse \[\] = \[
reverse (x:xs) = last (x:xs) : reverse(init (x:xs))

Automatically induced functions (renamed from \(f_1, f_2\))

last \[x\] = x init \[a\] = \[
last (x:xs) = last xs init (x:xs) = x:(init xs)
Data-type Definitions

data [a] = [] | a:[a]

Target Function

reverse :: [a]-> [a]
reverse [] = []
reverse [a] = [a]
reverse [a,b] = [b,a]
reverse [a,b,c] = [c,b,a]

Background Knowledge

snoc :: [a] -> a -> [a]
snoc [] x = [x]
snoc [x] y = [x,y]
snoc [x,y] z = [x,y,z]

- Input must be the first $k$ I/O examples (wrt to input data type)
- Background knowledge is optional
Output

Set of (recursive) equations which cover the examples

**reverse Solution**

\[
\text{reverse } [] = [] \\
\text{reverse } (x:x \text{xs}) = \text{snoc } (\text{reverse } \text{xs}) \ x
\]

**Restriction Bias**

- Subset of **HASKELL** or **MAUDE**
- Case distinction by *pattern matching*
- Syntactical restriction: patterns are not allowed to unify

**Preference Bias**

- Minimal number of case distinctions
Basic Idea

- Search a rule which explains/covers a (sub-) set of examples
- Initial hypothesis is a single rule which is the least general generalization (anti-unification) over all examples

Example Equations

reverse [a] = [a]
reverse [a,b] = [b,a]

Initial Hypothesis

reverse (x:xs) = (y:ys)

Hypothesis contains unbound variables in the body!
Basic Idea cont.

Initial Hypothesis

reverse \( (x:xs) = (y:ys) \)

Unbound variables are cue for induction.

Three Induction Operators (to apply simultaneously)

1. **Partitioning** of examples
   \( \rightsquigarrow \) Sets of equations divided by case distinction

2. Replace right-hand side by **program call** (recursive or background)

3. Replace sub-terms with unbound variables by to be induced **sub-functions**

Kitzelmann & Schmid, JMLR, 7, 2006; Kitzelmann, LOPSTR, 2008; Kitzelmann doctoral thesis 2010
## Some Empirical Results (Hofmann et al. AGI’09)

<table>
<thead>
<tr>
<th></th>
<th>isort</th>
<th>reverse</th>
<th>weave</th>
<th>shiftr</th>
<th>mult/add</th>
<th>allodds</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADATE</td>
<td>70.0</td>
<td>78.0</td>
<td>80.0</td>
<td>18.81</td>
<td>—</td>
<td>214.87</td>
</tr>
<tr>
<td>FLIP</td>
<td>×</td>
<td>—</td>
<td>134.24</td>
<td>448.55</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>FFOIL</td>
<td>×</td>
<td>—</td>
<td>0.4</td>
<td>&lt; 0.1</td>
<td>8.1</td>
<td>0.1</td>
</tr>
<tr>
<td>GOLEM</td>
<td>0.714</td>
<td>—</td>
<td>0.66</td>
<td>0.298</td>
<td>—</td>
<td>0.016</td>
</tr>
<tr>
<td>IGOR II</td>
<td>0.105</td>
<td>0.103</td>
<td>0.200</td>
<td>0.127</td>
<td>⊙</td>
<td>⊙</td>
</tr>
<tr>
<td>MAGH.</td>
<td>0.01</td>
<td>0.08</td>
<td>⊙</td>
<td>157.32</td>
<td>—</td>
<td>×</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>lasts</th>
<th>last</th>
<th>member</th>
<th>odd/even</th>
<th>multlast</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADATE</td>
<td>822.0</td>
<td>0.2</td>
<td>2.0</td>
<td>—</td>
<td>4.3</td>
</tr>
<tr>
<td>FLIP</td>
<td>×</td>
<td>0.020</td>
<td>17.868</td>
<td>0.130</td>
<td>448.90</td>
</tr>
<tr>
<td>FFOIL</td>
<td>0.7</td>
<td>0.1</td>
<td>0.1</td>
<td>&lt; 0.1</td>
<td>&lt; 0.1</td>
</tr>
<tr>
<td>GOLEM</td>
<td>1.062</td>
<td>&lt; 0.001</td>
<td>0.033</td>
<td>—</td>
<td>&lt; 0.001</td>
</tr>
<tr>
<td>IGOR II</td>
<td>5.695</td>
<td>0.007</td>
<td>0.152</td>
<td>0.019</td>
<td>0.023</td>
</tr>
<tr>
<td>MAGH.</td>
<td>19.43</td>
<td>0.01</td>
<td>⊙</td>
<td>—</td>
<td>0.30</td>
</tr>
</tbody>
</table>

— not tested   × stack overflow   ⊙ timeout   ⊥ wrong
all runtimes in seconds
Application of IGOR2 to Cognitive Problems

(Schmid & Kitzelmann, CSR, 2011)

**Problem Solving**

- **Clearblock** (4 examples, 0.036 sec)
- **Rocket** (3 examples, 0.012 sec)
- **Tower of Hanoi** (3 examples, 0.076 sec)
- **Car Park** (4 examples, 0.024 sec)
- **Blocks-world Tower** (9 examples, 1.2 sec)

**Recursive Concepts**

- **Ancestor** (9 examples, 10.1 sec)

**Syntactic Rules**

- **Phrase structure grammar** (3 examples, 0.072 sec)

\[
S \rightarrow NP \ VP \\
NP \rightarrow \text{d n} \\
VP \rightarrow v \ NP \mid v \ S
\]
Learning Tower of Hanoi

**Input to IGOR2**

```plaintext
eq Hanoi(0, Src, Aux, Dst, S) = 
  move(0, Src, Dst, S) .

eq Hanoi(s 0, Src, Aux, Dst, S) = 
  move(0, Aux, Dst,
      move(s 0, Src, Dst,
          move(0, Src, Aux, S))) .

eq Hanoi(s s 0, Src, Aux, Dst, S) = 
  move(0, Src, Dst,
      move(s 0, Aux, Dst,
          move(0, Aux, Src,
              move(s s 0, Src, Dst,
                  move(0, Dst, Aux,
                      move(s 0, Src, Aux,
                          move(0, Src, Dst, S)))))))) .
```

**Induced Tower of Hanoi Rules (3 examples, 0.076 sec)**

- \(Hanoi(0, \text{Src}, \text{Aux}, \text{Dst}, S) = \text{move}(0, \text{Src}, \text{Dst}, S)\)
- \(Hanoi(s \ D, \text{Src}, \text{Aux}, \text{Dst}, S) =
  Hanoi(D, \text{Aux}, \text{Src}, \text{Dst},
    \text{move}(s \ D, \text{Src}, \text{Dst},
        Hanoi(D, \text{Src}, \text{Dst}, \text{Aux}, S)))\)
Applying IGOR2 to Number Series Induction

- IGOR2 is designed as an IP system
- Generalization over tracesstreams of observations to productive rules
- Learning from few, small examples, generalization to $n$
- IGOR2 as a “cognitive rule acquisition device” (compare Chomsky’s LAD)?
- Detterman challenge 2011: An AI system should be able to solve a variety of different problems without being engineered to each special application
- IGOR2 can solve different cognitive problems “from the shelf”!
- Further example: IQ test problems – number series (J. Hofmann, Kitzelmann, Schmid, KI’14)
Applicability of IGOR2 to number series problems

- Crucial: How to represent number series problems as input for IGOR2

(1) Input List – Output Successor Value

\[
\begin{align*}
eq \text{Plustwo}((s \ 0) \ nil) &= s^3 \ 0 \\
eq \text{Plustwo}((s^3 \ 0) \ (s \ 0) \ nil) &= s^5 \ 0 \\
eq \text{Plustwo}((s^5 \ 0) \ (s^3 \ 0) \ (s \ 0) \ nil) &= s^7 \ 0
\end{align*}
\]

(2) Input Position – Output List

\[
\begin{align*}
eq \text{Plustwo}(s \ 0) &= (s \ 0) \ nil \\
eq \text{Plustwo}(s^2 \ 0) &= (s^3 \ 0)(s \ 0) \ nil \\
eq \text{Plustwo}(s^3 \ 0) &= (s^5 \ 0)(s^3 \ 0)(s \ 0) \ nil \\
eq \text{Plustwo}(s^4 \ 0) &= (s^7 \ 0)(s^5 \ 0)(s^3 \ 0)(s \ 0) \ nil
\end{align*}
\]

(3) Input Position – Output Value

\[
\begin{align*}
eq \text{Plustwo}(s \ 0) &= s \ 0 \\
eq \text{Plustwo}(s \ s \ 0) &= s \ s \ s \ 0 \\
eq \text{Plustwo}(s \ s \ s \ 0) &= s \ s \ s \ s \ s \ 0 \\
eq \text{Plustwo}(s \ s \ s \ s \ 0) &= s \ s \ s \ s \ s \ s \ s \ s \ s \ 0
\end{align*}
\]
Series examples

<table>
<thead>
<tr>
<th>Series</th>
<th>Formula</th>
<th>Size</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 12 13 14 23</td>
<td>$f(n - 3) + 11$</td>
<td>+, 1</td>
<td>small, large, linear, $n - 3$, const</td>
</tr>
<tr>
<td>1 2 3 5 8</td>
<td>$f(n - 1) + f(n - 2)$</td>
<td>+, 1</td>
<td>small, small, comp, $n - 1/n - 2$, const</td>
</tr>
<tr>
<td>6 7 8 18 21 24 54</td>
<td>$f(n - 3) \times 3$</td>
<td>$\times$, 1</td>
<td>small, small, linear, $n - 3$, const</td>
</tr>
<tr>
<td>3 4 12 48 576</td>
<td>$f(n - 1) \times f(n - 2)$</td>
<td>$\times$, 1</td>
<td>small, sm/lrg, comp, $n - 1/n - 2$, const</td>
</tr>
<tr>
<td>5 10 30 120 600</td>
<td>$f(n - 1) \times n$</td>
<td>$\times$, 1</td>
<td>large, large, linear, $n - 1$, pos</td>
</tr>
<tr>
<td>15 15 16 15 15 16 15</td>
<td>$f(n - 3)$</td>
<td>$=$, 1</td>
<td>large, large, linear, $n - 3$, const</td>
</tr>
</tbody>
</table>
Combining IP, Planning, Analogy

- Learning from planning: generate action traces with an AI planning system and learn a recursive rule set
- Use already known recursive rule sets to solve structurally similar problems (analogy)
- Generalization over base and target problems results in a hierarchy of program/problem solving schemes
Summary

- Learning can be high-level, on symbolic representations or low-level (e.g., feature vectors)
- Reinforcement learning addresses learning of control strategies/policies based on a statistical approach
- Inductive programming addresses learning of recursive rule sets from few examples or from traces
- IP is part of automated programming research, it complements deductive approaches
- The first IP system was Summers’s THESYS, Summers also provided the formal foundation (Synthesis Theorem)
- IGOR2 is a current approach to IP, it is more general than THESYS and incorporates aspects from ILP (use of background knowledge, function invention)
- IP approaches are often general enough to be applied to other strategy learning domains, e.g., from cognition
In contrast to classical ML, IP needs only few and only positive examples.

IP is nevertheless *supervised* (learning functions: negative examples are present implicitly).
# Learning Terminology

## Approaches:

<table>
<thead>
<tr>
<th>Concept / Classification</th>
<th>Policy Learning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>symbolic</strong></td>
<td>statistical / neuronal network</td>
</tr>
<tr>
<td><strong>inductive</strong></td>
<td>analytical</td>
</tr>
</tbody>
</table>

## Learning Strategy:

<table>
<thead>
<tr>
<th>Data:</th>
<th>learning from examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Values:</td>
<td>input/output examples or traces</td>
</tr>
<tr>
<td></td>
<td>Recursive rule sets</td>
</tr>
</tbody>
</table>