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

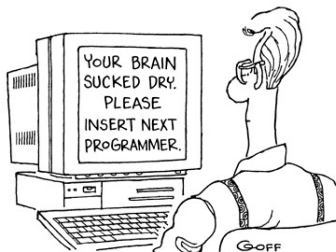
Inductive Programming

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

Program Synthesis

Automa*g*ic 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

last(l) $\Leftarrow$ *find z such that for some y, l = y* $\circ$ *[z]*
where islist(l) and l $\neq$ *[ ]* (Manna & Waldinger)

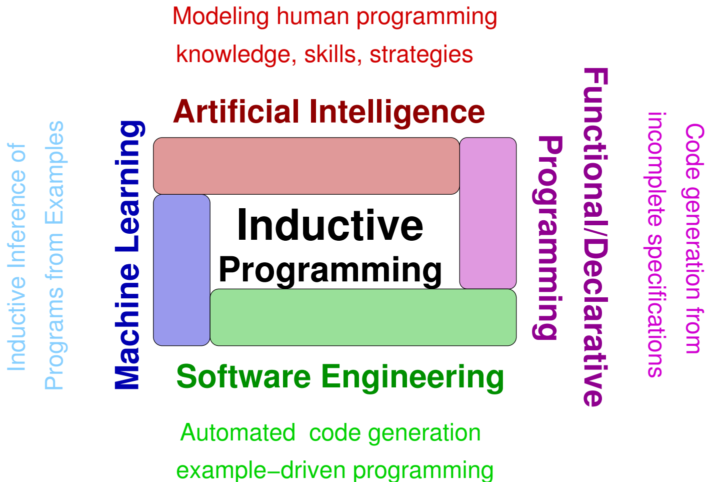
Approaches to Program Synthesis

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



Inductive Programming Example

Learning `last`

I/O Examples

```
last [a]       =  a
last [a,b]     =  b
last [a,b,c]   =  c
last [a,b,c,d] =  d
```

Generalized Program

```
last [x]      =  x
last (x:xs)   =  last xs
```

Some Syntax

```
-- sugared
[1,2,3,4]
-- normal infix
(1:2:3:4:[])
-- normal prefix
((:) 1
   ((:) 2
      ((:) 3
         ((:) 4
              [])))))
```

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 (recursivly) 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: MAGICHASKELLER (Katayama)
- also in functional/generic programming context: automated generation of instances for data types in the model-based test tool G∀st (Koopmann & Plasmeijer)

Inductive Programming – Approaches

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

$$\text{nil} \rightarrow \text{nil}$$
$$(A) \rightarrow ((A))$$
$$(A\ B) \rightarrow ((A)\ (B))$$
$$(A\ B\ C) \rightarrow ((A)\ (B)\ (C))$$

Traces

$$
\begin{aligned}
F_L(x) \leftarrow \quad & (\text{atom(x)} \rightarrow \text{nil},\\
& \text{atom(cdr(x))} \rightarrow \text{cons(x, nil)},\\
& \text{atom(cddr(x))} \rightarrow \text{cons(cons(car(x), nil), cons(cdr(x), nil))},\\
& \text{T} \rightarrow \text{cons(cons(car(x), nil), cons(cons(cadr(x),nil),}\\
& \qquad\qquad \text{cons(cddr(x),nil))))}
\end{aligned}
$$

Example: Deriving Fragments

Unique Expressions for Fragment (A B)

(x, (A B)),
(car[x], A),
(cdr[x], (B)),
(cadr[x], B),
(cddr[x], ( ))

Combining Expressions

((A) (B)) = cons[(A); ((B))] = cons[cons[A, ()];cons[(B), ( )]].

Replacing Values by Functions

cons[cons(car[x]; ( )];cons[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 recusion
- Traces are *k*th unfolding of an unknown program following the program scheme
- Identify differences, detect recurrence

$$\begin{aligned} F(x) \leftarrow \quad & (p_1(x) \rightarrow f_1(x), \\ & \ldots, \\ & p_k(x) \rightarrow f_k(x), \\ & T \rightarrow C(F(b(x)), x)) \end{aligned}$$

Example: Fold Traces

kth unfolding

$$
\begin{aligned}
F_L(x) \leftarrow \quad & (\text{atom(x)} \rightarrow \text{nil}, \\
& \text{atom(cdr(x))} \rightarrow \text{cons(x, nil)}, \\
& \text{atom(cddr(x))} \rightarrow \text{cons(cons(car(x), nil), cons(cdr(x), nil))}, \\
& \text{T} \rightarrow \text{cons(cons(car(x), nil), cons(cons(cadr(x),nil),} \\
& \qquad\qquad \text{cons(cddr(x),nil))))}
\end{aligned}
$$

Differences:
$p_2(x) = p_1(\text{cdr(x)})$
$p_3(x) = p_2(\text{cdr(x)})$
$p_4(x) = p_3(\text{cdr(x)})$

$f_2(x) = cons(x, f_1(x))$
$f_3(x) =$
$cons(cons(car(x), nil), f_2(cdr(x)))$
$f_4(x) =$
$cons(cons(car(x), nil), f_3(cdr(x)))$

Recurrence Relations:
$p_1(x) = \text{atom(x)}$
$p_{k+1}(x) = p_k(\text{cdr(x)})$ for $k = 1, 2, 3$

$f_1(x) = nil$
$f_2(x) = cons(x, f_1(x))$
$f_{k+1}(x) = cons(cons(car(x), nil), f_k(cdr(x)))$
for $k = 2, 3$

Example: Fold Traces

kth unfolding

$$
\begin{aligned}
F_L(x) \leftarrow \quad & (\text{atom(x)} \rightarrow \text{nil},\\
& \text{atom(cdr(x))} \rightarrow \text{cons(x, nil)},\\
& \text{atom(cddr(x))} \rightarrow \text{cons(cons(car(x), nil), cons(cdr(x), nil))},\\
& \text{T} \rightarrow \text{cons(cons(car(x), nil), cons(cons(cadr(x),nil),}\\
& \qquad\qquad \text{cons(cddr(x),nil))))}
\end{aligned}
$$

Folded Program

$$
\begin{aligned}
\text{unpack(x)} \leftarrow \quad & (\text{atom(x)} \rightarrow \text{nil},\\
& \text{T} \rightarrow \text{u(x))}\\
\text{u(x)} \leftarrow \quad & (\text{atom(cdr(x))} \rightarrow \text{cons(x, nil)},\\
& \text{T} \rightarrow \text{cons(cons(car(x), nil), u(cdr(x))))}
\end{aligned}
$$

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

$$U^1 \leftarrow \begin{array}{l}(atom(x) \rightarrow nil,\\ T \rightarrow \Omega)\end{array}$$

Defined for empty list and lists with one element

$$U^2 \leftarrow \begin{array}{l}(atom(x) \rightarrow nil,\\ atom(cdr(x)) \rightarrow cons(x, nil),\\ T \rightarrow \Omega)\end{array}$$

... Defined for lists up to n 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 $f1$, $f2$)

```
last [x]     = x             init [a]     = []
last (x:xs) = last xs        init (x:xs) = x:(init xs)
```

Input

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

```
reverse   []    = []
reverse (x:xs) = snoc (reverse 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)

	isort	*reverse*	*weave*	*shiftr*	*mult/add*	*allodds*
ADATE	70.0	78.0	80.0	18.81	—	214.87
FLIP	$\times$	—	134.24$^{\perp}$	448.55$^{\perp}$	$\times$	$\times$
FFOIL	$\times$	—	0.4$^{\perp}$	$< 0.1^{\perp}$	8.1$^{\perp}$	0.1$^{\perp}$
GOLEM	0.714	—	0.66$^{\perp}$	0.298	—	0.016$^{\perp}$
IGOR II	0.105	0.103	0.200	0.127	$\odot$	$\odot$
MAGH.	0.01	0.08	$\odot$	157.32	—	$\times$

	lasts	*last*	*member*	*odd/even*	*multlast*
ADATE	822.0	0.2	2.0	—	4.3
FLIP	$\times$	0.020	17.868	0.130	448.90$^{\perp}$
FFOIL	0.7$^{\perp}$	0.1	0.1$^{\perp}$	$< 0.1^{\perp}$	< 0.1
GOLEM	1.062	< 0.001	0.033	—	< 0.001
IGOR II	5.695	0.007	0.152	0.019	0.023
MAGH.	19.43	0.01	$\odot$	—	0.30

— not tested $\times$ stack overflow $\odot$ timeout $\perp$ wrong

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 → NP VP   NP → d n   VP → v NP | v S

Learning Tower of Hanoi

Input to IGOR2

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

Applying IGOR2 to Number Series Induction

- IGOR2 is designed as IP system
- Generalization over traces/streams of observations to productive rules
- Learning from few, small examples, generalization to *n*
- IGOR2 as a “cognitive rule acquisition device” (ct. 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

```
eq Plustwo((s 0) nil) = s^3 0
eq Plustwo((s^3 0) (s 0) nil) = s^5 0
eq Plustwo((s^5 0) (s^3 0) (s 0) nil) = s^7 0
```

(2) Input Position – Output List

```
eq Plustwo(s 0) = (s 0) nil
eq Plustwo(s^2 0) = (s^3 0)(s 0) nil
eq Plustwo(s^3 0) = (s^5 0)(s^3 0)(s 0) nil
eq Plustwo(s^4 0) = (s^7 0)(s^5 0)(s^3 0)(s 0) nil
```

(3) Input Position – Output Value

```
eq Plustwo(s 0) = s 0
eq Plustwo(s s 0) = s s s 0
eq Plustwo(s s s 0) = s s s s s 0
eq Plustwo(s s s s 0) = s s s s s s s 0
```

Series examples

1 2 3 12 13 14 23	$f(n-3)+11$	$+$, 1, small, large, linear, $n-3$, const
1 2 3 5 8	$f(n-1)+f(n-2)$	$+$, 1, small, small, comp, $n-1/n-2$, const
6 7 8 18 21 24 54	$f(n-3)\times 3$	$\times$, 1, small, small, linear, $n-3$, const
3 4 12 48 576	$f(n-1)\times f(n-2)$	$\times$, 1, small, sm/lrg, comp, $n-1/n-2$, const
5 10 30 120 600	$f(n-1)\times n$	$\times$, 1, large, large, linear, $n-1$, pos
15 15 16 15 15 16 15	$f(n-3)$	$=$, 1, large, large, linear, $n-3$, const

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

Summary

- 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

Thesys, Igor2

Supervised Learning	unsupervised learning

Approaches:

Concept / Classification	**Policy Learning**
symbolic	statistical / neuronal network
inductive	**analytical**

Learning Strategy:	**learning from examples**
Data:	**input/output examples or traces**
Target Values:	**Recursive rule sets**