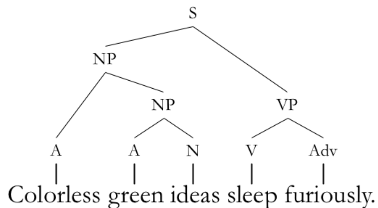
Bayesian program learning: Prospects for building more human-like AI systems

Joshua Tenenbaum and Kevin Ellis

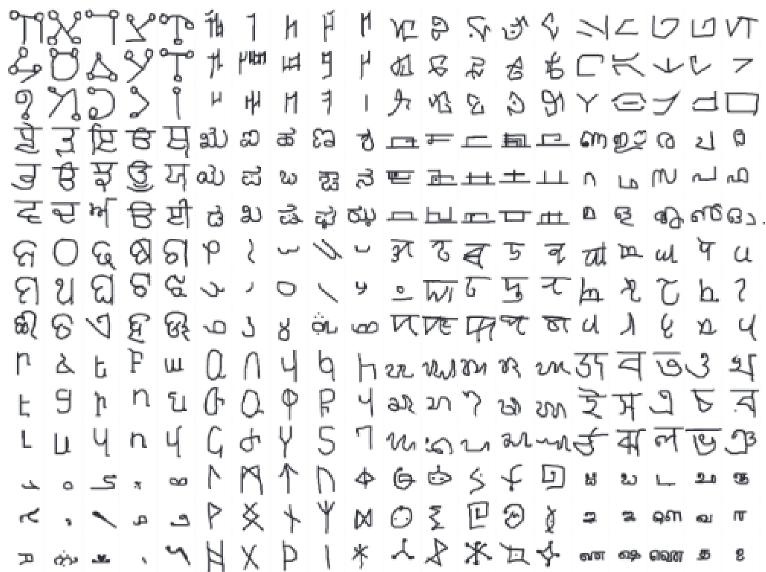
MIT

December 10, 2016

Programs in language

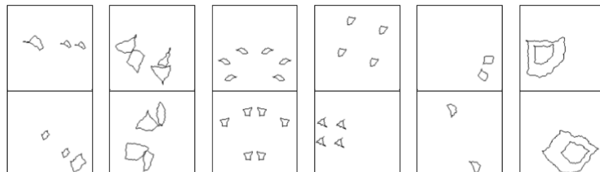


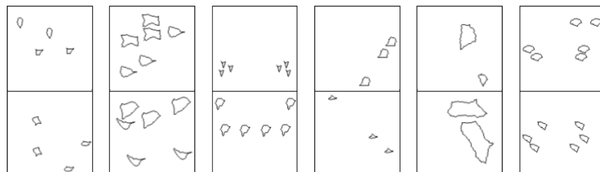
Programs that make signs

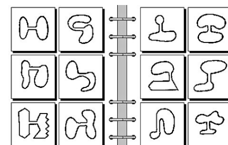


Lake, Salakhutdinov, Tenenbaum. *Science* 2015.

Programs in vision







The story

Where we want to be: how do we learn to drive, talk, do math

Where we are: geometric concepts, linguistic rules, simple programming

The story

Key ingredients:

- Learning from few examples
- Modeling your uncertainty
- Coming up with explanations (generative/causal models)

The story

Key ingredients:

- Learning from few examples
- Modeling your uncertainty
- Coming up with explanations (generative/causal models)

The toolkit:
Programming Languages meets Bayes

"Bayesian Program Learning":
[Lake et al, 2015]; [Liang et al, 2010]

One-shot learning

`Humans can learn programs` $\longrightarrow$ `H's can`

One-shot learning

`Humans can learn programs` $\longrightarrow$ `H's can`
`Robots don't eat meat` $\longrightarrow$???

One-shot learning

`Humans can learn programs` $\longrightarrow$ `H's can`
`Robots don't eat meat` $\longrightarrow$ `R's don't`

Simple algorithms from few examples

`(9 3 5 2 4)` $\longrightarrow$ `(4 2 5 3 9)`

Program-synthesis style problems: eg [Feser et al. 2015], [Solar-Lezama 2008]

Simple algorithms from few examples

`(9 3 5 2 4)` $\longrightarrow$ `(4 2 5 3 9)`

`(1 2 9 9 2 5)` $\longrightarrow$ `(1 2 9 5)`
`(9 7 9 0)` $\longrightarrow$ `(9 7 0)`

Program-synthesis style problems: eg [Feser et al. 2015], [Solar-Lezama 2008]

Simple algorithms from few examples

(9 3 5 2 4) $\longrightarrow$ (4 2 5 3 9)

(1 2 9 9 2 5) $\longrightarrow$ (1 2 9 5)
(9 7 9 0) $\longrightarrow$ (9 7 0)

(0 9) $\longrightarrow$ (443 0 9)
(1 1 1 1) $\longrightarrow$ (443 1 1 1 1)

Program-synthesis style problems: eg [Feser et al. 2015], [Solar-Lezama 2008]

Simple algorithms from few examples

(9 3 5 2 4) $\longrightarrow$ (4 2 5 3 9)

(1 2 9 9 2 5) $\longrightarrow$ (1 2 9 5)
(9 7 9 0) $\longrightarrow$ (9 7 0)

(0 9) $\longrightarrow$ (443 0 9)
(1 1 1 1) $\longrightarrow$ (443 1 1 1 1)

(4 9 5 2) $\longrightarrow$ (4 0 9 0 5 0 2)

Program-synthesis style problems: eg [Feser et al. 2015], [Solar-Lezama 2008]

Knowing when you don't know

When one example doesn't suffice

Knowing when you don't know

When one example doesn't suffice

`(7 4 3)` $\longrightarrow$ `(3 4 7)`

Knowing when you don't know

When one example doesn't suffice

(7 4 3) $\longrightarrow$ (3 4 7)
(5 2 3) $\longrightarrow$ (2 3 5)

Knowing when you don't know

When one example doesn't suffice

$$
\begin{aligned}
\texttt{(7 4 3)} &\longrightarrow \texttt{(3 4 7)}\\
\texttt{(5 2 3)} &\longrightarrow \texttt{(2 3 5)}\\
\texttt{(1 6 4)} &\longrightarrow \texttt{(1 4 6)}
\end{aligned}
$$

Knowing when you don't know

When one example doesn't suffice

```
(7 4 3) ⟶ (3 4 7)
(5 2 3) ⟶ (2 3 5)
(1 6 4) ⟶ (1 4 6)
```

Input	Output
P Smith	P

A possible program	English interpretation
`substr(0,pos('',' ',0))`	"first word"
`substr(0,1)`	"first character"
`const('P')`	"output P"

See Sumit Gulwani's FlashFill system: Gulwani 2011. POPL.

Program induction algorithms for learning from few examples

Inductive biases over programs

Bayes: Learn from few examples $\implies$ Strong inductive bias
Programming Languages: Domain Specific Language $\mathcal{L}$

$$\mathbb{P}[\text{program}|\text{examples}] \propto \mathbb{P}[\text{program}]\mathbb{P}[\text{examples}|\text{program}]$$

$$\text{program} \in \mathcal{L}$$

Inductive biases over programs

Bayes: Learn from few examples $\implies$ Strong inductive bias
Programming Languages: Domain Specific Language $\mathcal{L}$

Transforming strings

```
Program ::= Term
          | Program + Term
Term    ::= String
          | substr(Pos,Pos)
Pos     ::= Number
          | pos(Str,Str,Num)
Num     ::= 0 | 1 | 2 | ...
          | -1 | -2 | ...
Str     ::= Character
          | Character + String
Character ::= a | b | c | ...
```

Transforming sequences

```
Bool ::= (<= Int) | (>= Int)
       | (= Int)
Int  ::= 0
       | (+1 Int) | (-1 Int)
       | (length List)
       | (head List)
List ::= nil | X
       | (filter Bool List)
       | (tail List)
       | (list Int)
```

ProgramSample

Sample from posterior $\mathbb{P}[\text{program}|\text{examples}]$

PL techniques for searching

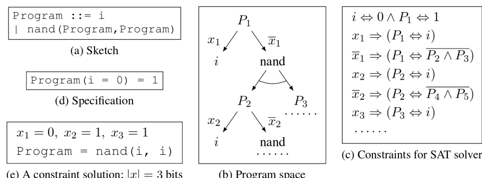
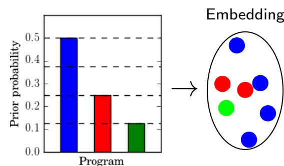
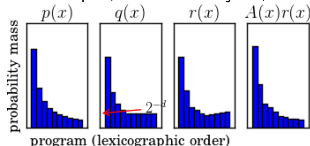


Figure 2: Synthesizing a program via sketching and constraint solving. `Typewriter font` refers to pieces of programs or sketches, while math font refers to pieces of a constraint satisfaction problem. The variable `i` is the program input.

ML techniques for sampling



"XORSample", "Embed & Project", ...



Proposition 3. *Let $Ar(x)$ be proportional to $A(x)r(x)$. Then $D(p||Ar) < \log\left(1 + \frac{1+2^{-\gamma}}{1+2^{\Delta}}\right)$ where $\Delta = \log|E| - K$ and $\gamma = d - \log|X| - |x_*|$.*

Proposition 5. *The expected number of calls to the solver per sample is bounded above by $\frac{1+2^{\Delta}}{(1+2^{-\gamma})^{-1}(1+2^{-\Delta})^{-1}-2^{-\Delta}}$.*

ProgramSample learning to sort

Examples	MDL	Sampled correct?	Posterior is...
(7 4 3)→(3 4 7)	reverse	✗	diffuse
(7 4 3)→(3 4 7) (5 2 3)→(2 3 5) (1 6 4)→(1 4 6)	sort	✓	peaky
(3 2 4 1)→(1 2 3 4)	count up to list length	✗	peaky
(3 2 4 1)→(1 2 3 4) (1 6 2 0)→(0 1 2 6)	sort	✓	peaky

See *Sampling for Bayesian Program Learning*, Ellis, Solar-Lezama, Tenenbaum. NIPS 2016.

ProgramSample list experiments

Reverse:

```
(if (<= (+1 0) (length X))
    X
    (append (recurse (tail X))
            (list (head X))))
```

ProgramSample list experiments

Reverse:

```
(if (= (+1 0) (length X))
    X
    (append (recurse (filter (<= (+1 0)) (tail X)))
            (list (head X))))
```

ProgramSample list experiments

Reverse:

```
(if (<= (+1 0) (length X))
    X
    (append (recurse (tail X))
            (list (head X))))
```

Count: # times list head occurs in list tail

```
(length (filter (= (head X)) (tail X)))
```

ProgramSample list experiments

Reverse:

```
(if (<= (+1 0) (length X))
    X
    (append (recurse (tail X))
            (list (head X))))
```

Count: # times list head occurs in list tail

```
(-1 (length (filter (= (head X)) X)))
```

ProgramSample list experiments

Reverse:

```
(if (<= (+1 0) (length X))
    X
    (append (recurse (tail X))
            (list (head X))))
```

Count: # times list head occurs in list tail

```
(length (filter (= (head X)) (tail X)))
```

Sort:

```
(if (<= (+1 0) (length X))
    X
    (append (recurse (filter (<= (head X)) (tail X)))
            (list (head X))
            (recurse (filter (>= (head X)) (tail X)))))
```

ProgramSample list experiments

Reverse:

```
(if (<= (+1 0) (length X))
    X
    (append (recurse (tail X))
            (list (head X))))
```

Count: # times list head occurs in list tail

```
(length (filter (= (head X)) (tail X)))
```

Sort:

```
(if (= (+1 0) (length X))
    X
    (append (recurse (tail X))
            (list (length X))))
```

ProgramSample text edit experiments

Training examples		Program
Input	Output	
12/31/13	12.31	`SubStr(0,Pos('','/',0))+`
1/23/2009	1.23	`Constant('.')+`
4/12/2023	4.12	`SubStr(Pos('/','',0),`
6/23/15	6.23	`Pos('','/',-1))`
7/15/2015	7.15	

17 problems adapted from:
Lin et al. Bias reformulation for one-shot function induction. ECAI 2014.

The last ingredient

Programs as Generative Models

From toy languages to real languages

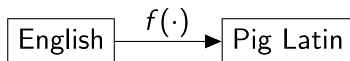
Pig Latin:

`pig` $\longrightarrow$ `ig-pay`

`latin` $\longrightarrow$ `atin-lay`

`program` $\longrightarrow$ `rogram-pay`

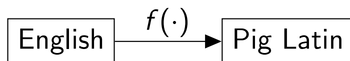
`induction` $\longrightarrow$ `induction-ay`



From toy languages to real languages

Pig Latin:

`pig` ⟶ `ig-pay`
`latin` ⟶ `atin-lay`
`program` ⟶ `rogram-pay`
`induction` ⟶ `induction-ay`



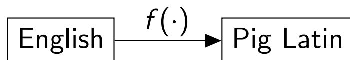
Latin: Nominative ∼ Genitive

`noks` ∼ `noktis` ("night")
`frons` ∼ `frontis` ("brow")
`frons` ∼ `frondis` ("leaf")

From toy languages to real languages

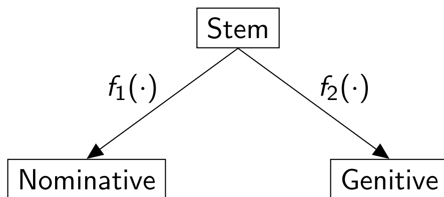
Pig Latin:

`pig` ⟶ `ig-pay`
`latin` ⟶ `atin-lay`
`program` ⟶ `rogram-pay`
`induction` ⟶ `induction-ay`



Latin: Nominative ∼ Genitive

`noks` ∼ `noktis` (“night”)
`frons` ∼ `frontis` (“brow”)
`frons` ∼ `frondis` (“leaf”)



UNSUPERVISED program induction!

Learning to count

7 Tibetan

Numbers between 11 and 19 are formed by placing the appropriate digit after the number 10, and multiples of 10 are formed by placing the appropriate multiplier before the number 10. What are the underlying forms of the basic numerals, and what phonological rule is involved in accounting for these data?

ǰu	'10'	ǰig	'1'	ǰugǰig	'11'
ši	'4'	ǰubši	'14'	šibǰu	'40'
gu	'9'	ǰurgu	'19'	gubǰu	'90'
ŋa	'5'	ǰuŋa	'15'	ŋabǰu	'50'

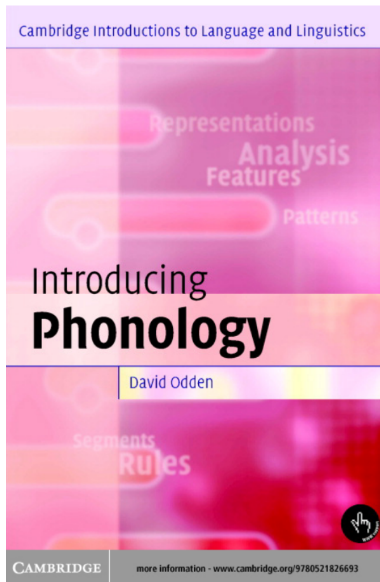
Learning to count

Counting in Tibetan:

Number	Pronunciation
1	ǰig
4	ši
10	ǰu
11	ǰu+gǰig (10+1)
40	ši+bǰu (4+10)

Explanation:
ǰig ("1") is really gǰig
ǰu ("10") is really bǰu
Program:
$C \rightarrow \varnothing / \# \; C \;_$
(Delete initial cluster of consonants)

Learning to make words



Timothy O'Donnell

Learning to make words

Cambridge Introducti

Introducing Phonol

David Odd

Patterns

Segments

Rules

CAMBRIDGE | more information - www.cambridge.org/9780521826693

Exercises

1 Axininca Campa

Provide underlying representations and a phonological rule which will account for the following alternations:

toniro	'palm'	notoniroti	'my palm'
yaarato	'black bee'	noyaaratoti	'my black bee'
kanari	'wild turkey'	noyanariti	'my wild turkey'
kosiri	'white monkey'	noyosiriti	'my white monkey'
pisiro	'small toucan'	nowisiroti	'my small toucan'
porita	'small hen'	noworitati	'my small hen'

p → w / [ ] _

k → y / [ ] _

kosiri, no-yosiri-ti

porita, no-worita-ti

yaarato, no-yaarato-ti

...

Timothy O'Donnell

Learning to make words

Cambridge Introducti

Introducing
Phonology

Patterns

CAMB

Exercises

1 Axininca Campa

Provide underlying representations and a phon
for the following alternations:

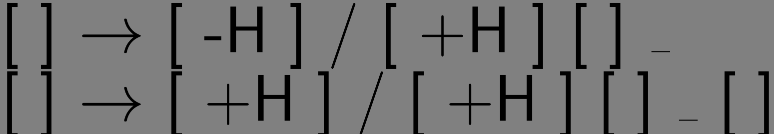
toniro	'palm'	notoni
yaarato	'black bee'	noyaar
kanari	'wild turkey'	noyana
kosiri	'white monkey'	noyosi
pisiro	'small toucan'	nowisi
porita	'small hen'	nowori

1 Kerewe

What two tone rules are motivated by the following data? Explain what order the rules apply in.

to V	*to V e.o*	*to V for*	*to V for e.o.*	
kubala	kubalana	kubalila	kubalilana	'count'
kugaya	kugayana	kugayila	kugayilana	'despise'
kugula	kugulana	kugulila	kugulilana	'buy'
kubála	kubálána	kubálíla	kubálílana	'kick'
kulúma	kulúmána	kulúmíla	kulúmílana	'bite'
kusúna	kusúnána	kusúníla	kusúnílana	'pinch'
kulába	kulábána	kulábíla	kulábílana	'pass'

to V us	*to V it*	*to V for us*	*to V it for us*	
kutúbála	kukíbála	kutúbálila	kukítúbalila	'count'
kutúgáya	kukígáya	kutúgáyila	kukítúgayila	'despise'
kutúgúla	kukígúla	kutúgúlila	kukítúgulila	'buy'
kutúbála	kukíbála	kutúbálila	kukítúbalila	'kick'
kutúlúma	kukílúma	kutúlúmila	kukítúlumila	'bite'
kutúsúna	kukísúna	kutúsúnila	kukítúsunila	'pinch'
kutúlába	kukílába	kutúlábila	kukítúlabila	'pass'



ku-láb-a, ku-láb-́ana, ku-láb-íla, ku-láb-ílana, ..., kukítú-lab-ila

ku-sún-a, ku-súnán-a, ku-sún-íla, ku-sún-ílana, ..., kukítú-sun-ila

...

Learning to make words

Cambridge Introducti

Exercises

1 Axininca Campa

Provide underlying representations and a phon

1 Kerewe

What two tone rules are motivated by the following data? Explain what order the rules apply in.

to V	*to V e.o*	*to V for*	*to V for e.o.*	
kubala	kubalana	kubalila	kubalilana	'count'
			ana	'despise'
			ana	'buy'
			ana	'kick'
			lana	'bite'
			lana	'pinch'
			ana	'pass'
			for us	
			alila	'count'
			avila	'despise'
				'buy'
				'kick'
				'bite'
				'pinch'
				'pass'

$$[\ +\text{tense}\ ] \rightarrow [\ \text{-hi}\ +\text{mid}\ ]\ /\ _\ [\ \text{-central}\ ]^*\ e$$

siind-a, seend-era

huut-a, hoot-era

suk-a, sok-era

tig-a, teg-era

…

5 Kuria

Provide appropriate underlying representations and phonological rules which will account for the following data:

Verb	*Verb for*	
suraaŋga	suraaŋgera	'praise'
taaŋgata	taaŋgatera	'lead'
baamba	baambera	'fit a drum head'
reenda	reendera	'guard'
rema	remera	'cultivate'
hoora	hoorera	'thresh'
roma	romera	'bite'
sooka	sookera	'respect'
tačora	tačorera	'tear'
siika	seekera	'close'
tiga	tegera	'leave behind'
ruga	rogera	'cook'
suka	sokera	'plait'
huuta	hootera	'blow'
riiŋga	reeŋgera	'fold'
siinda	seendera	'win'

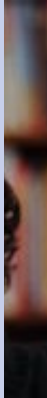
Segments

Rules

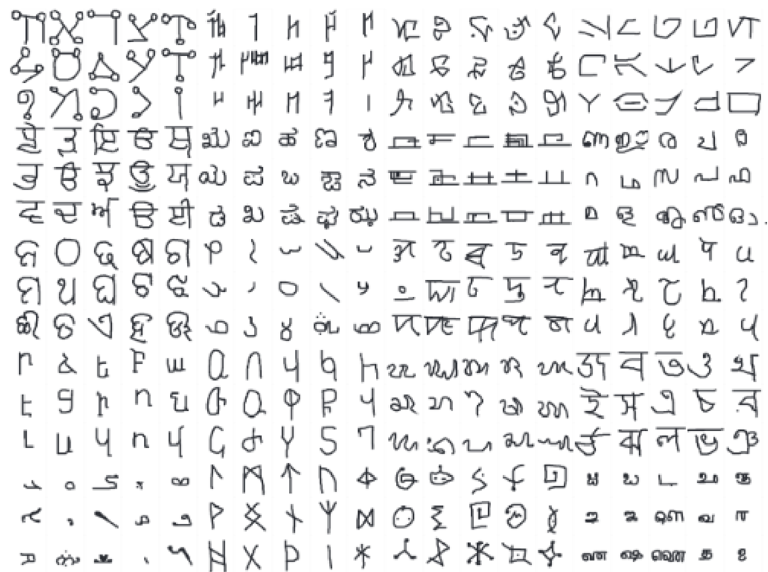
CAMBRIDGE

more information - www.cambridge.org/97805218266

ell

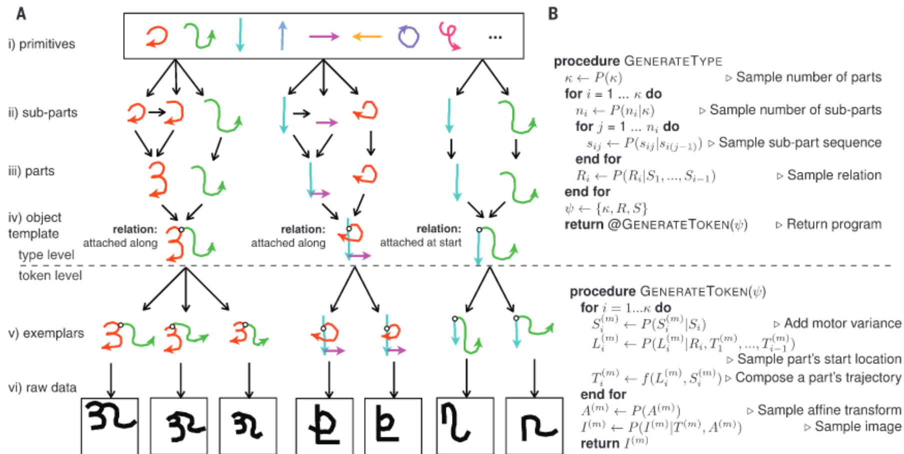


Learning to Draw Characters



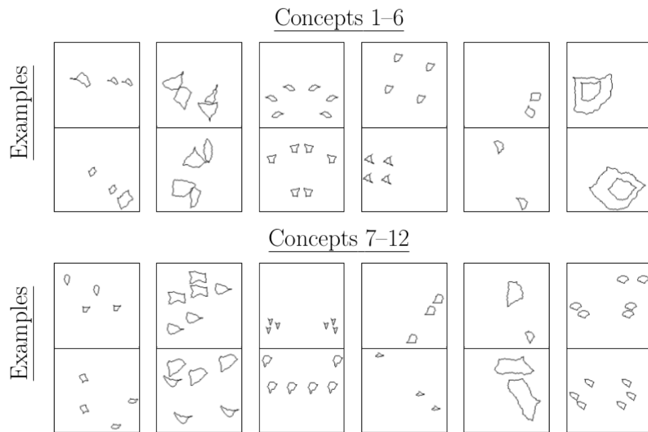
Lake, Salakhutdinov, Tenenbaum. *Science* 2015.

Learning to Draw Characters



Lake, Salakhutdinov, Tenenbaum. *Science* 2015.

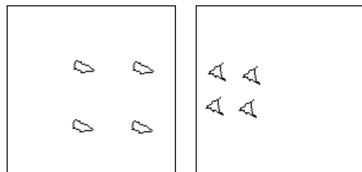
Learning Geometric Concepts



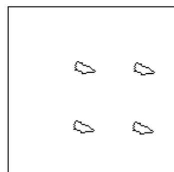
People are really good at these concepts! Fleuret et al. *PNAS* 2011.

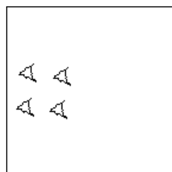
Innateness of geometric concepts: Dehaene et al. *Science* 2006.

Toolkit: Unsupervised Bayesian Program Learning



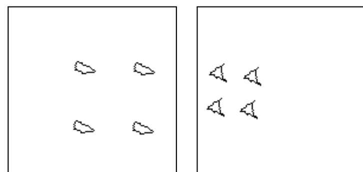
Toolkit: Unsupervised Bayesian Program Learning





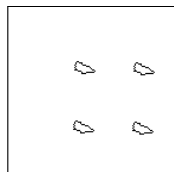
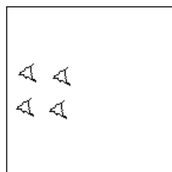
draw(shape[0]); move(ℓ);
draw(shape[0]); move($\ell, 90^\circ$);
draw(shape[0]); move($\ell, 90^\circ$);
draw(shape[0]);

Toolkit: Unsupervised Bayesian Program Learning

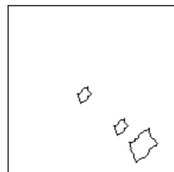
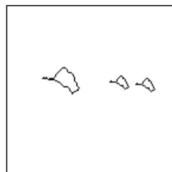


```
draw(shape[0]); move(ℓ);
draw(shape[0]); move(ℓ, 90°);
draw(shape[0]); move(ℓ, 90°);
draw(shape[0]);
```

Toolkit: Unsupervised Bayesian Program Learning

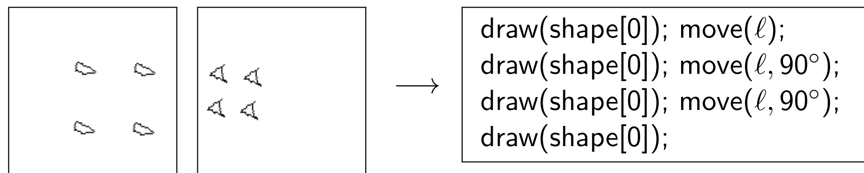



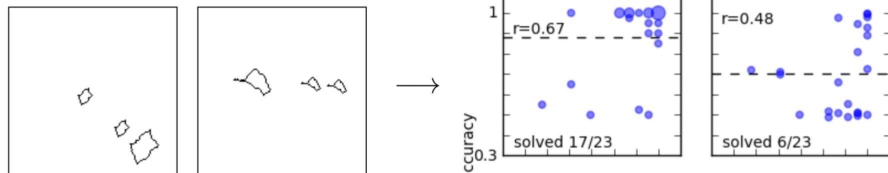
draw(shape[0]); move(ℓ);
draw(shape[0]); move($\ell, 90^\circ$);
draw(shape[0]); move($\ell, 90^\circ$);
draw(shape[0]);

goto(initPosition, initOrientation)
draw(shape[0]); move(ℓ);
draw(shape[0], scale=z); move(ℓ);
draw(shape[0], scale=z);

Toolkit: Unsupervised Bayesian Program Learning





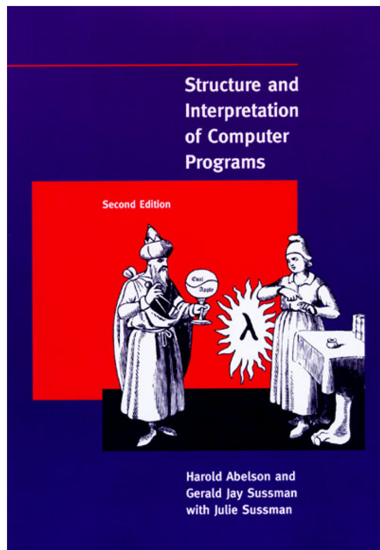
46 concepts, 23 problems:
Human (avg 6.27 examples): **19.85/23**
Best baseline (10000 examples): 11/23
Our baseline (6 examples): 11/23
BPL (ours) (6 examples): **17/23**

Ellis, Solar-Lezama, Tenenbaum. *NIPS* 2015.

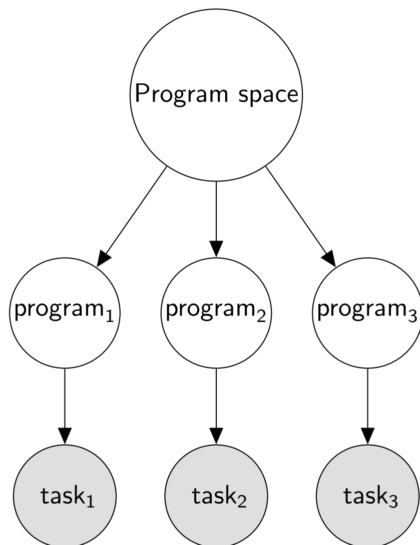
The Future

Curriculum Learning

How do we learn complicated behaviors, like programming?



Hierarchical Bayes & Curriculum Learning



[Liang et al, 2010]; [Lake et al, 2015]

Bootstrap Learning & Hierarchical Bayes

EC Algorithm. Starting with NAND, build complicated Boolean circuits

	func.	CL expression	schematic
(a)	NOT	(S NAND I)	x ≡
	AND	(C B NAND) (B (S NAND I)) → (C B NAND) (B NOT)	y x
	OR	((B (C (B (S NAND) NAND))) (S NAND I) → ((B (C (B (S NAND) NAND))) NOT	y x
(b)	E_1	S B (S NAND)	x $f(\cdot)$ $f(\cdot)$
	E_2	(B (C B NAND) S)	x $f(\cdot)$ $g(\cdot)$
(c)	TRUE	(S NAND) (S NAND I) → (S NAND) NOT	x
	FALSE	(S B (S NAND)) (S NAND I) → E_1 NOT	E_1 ( ) = x

Bootstrap Learning via Modular Concept Discovery. Dechter et al., 2013. *IJCAI*.

Eyal Dechter

Similar Ideas

Learning Programs: A Hierarchical Bayesian Approach

Percy Liang PLIANG@CS.BERKELEY.EDU
Computer Science Division, University of California, Berkeley, CA 94720, USA

Michael I. Jordan JORDAN@CS.BERKELEY.EDU
Computer Science Division and Department of Statistics, University of California, Berkeley, CA 94720, USA

Dan Klein KLEIN@CS.BERKELEY.EDU
Computer Science Division, University of California, Berkeley, CA 94720, USA

Abstract

We are interested in learning programs for multiple related tasks given only a few training examples per task. Since the program for a single task is underdetermined by its data, we introduce a nonparametric hierarchical Bayesian prior over programs which shares statistical strength across multiple tasks. The key challenge is to parametrize this multi-task sharing. For this, we introduce a new representation of programs based on combinatory logic and provide an MCMC algorithm that can perform safe program transformations on this representation to reveal shared inter-program substructures.

1. Introduction

A general focus in machine learning is the estimation of functions from examples. Most of the literature focuses on real-valued functions, which have proven useful in many classification and regression applications. This paper explores the learning of a different but also important class of functions—those specified most naturally by computer programs.

To motivate this direction of exploration, consider programming by demonstration (PBD) (Cypher, 1993). In PBD, a human demonstrates a repetitive task in a few contexts; the machine then learns to perform the task in new contexts. An example we consider in this paper is text editing (Lau et al., 2003). Suppose a user wishes to italicize all occurrences of the word *statistics*. If the user demonstrates italicizing two occurrences of *statistics*, can we generalize to the others? The solution to this italicization task can be represented compactly by a program: (1) move the cursor to the next occurrence of *statistics*, (2) insert <1>, (3) move to the end of the word, and (4) insert </1>.

From a learning perspective, the main difficulty with PBD is that it is only reasonable to expect one or two training examples from the user. Thus the program is underdetermined by the data: Although the user moved to the beginning of the word *statistics*, an alternate predicate might be after a space. Clearly, some sort of prior or complexity penalty over programs is necessary to provide an inductive bias. For real-valued functions, many penalties based on smoothness, norm, and dimension have been studied in detail for decades. For programs, what is a good measure of complexity (prior) that facilitates learning?

We often want to perform many related tasks (e.g., in text editing, another task might be to italicize the word *logic*). In this multi-task setting, it is natural to define a hierarchical prior (a joint measure of complexity) over multiple programs, which allows the sharing of statistical strength through the joint prior.

The key conceptual question is how to allow sharing between programs. Here, we can take inspiration from good software engineering principles: Programs should be structured modularly so as to enable code reuse. However, it is difficult to implement this intuition since programs typically have many internal dependencies; therefore, transforming programs safely into a modular form for statistical sharing without disrupting the program semantics requires care. Our solution is to build on *combinatory logic* (Schönfinkel, 1924), a simple and elegant formalism for building complex programs via composition of simpler subprograms. Its simplicity makes it conducive to probabilistic modeling.

Appearing in *Proceedings of the 27th International Conference on Machine Learning*, Haifa, Israel, 2010. Copyright 2010 by the author(s)/owner(s).

Similar Ideas

Learning Programs: A Hierarchical Bayesian Approach

Automatic Invention of Functional Abstractions

Robert J. Henderson and Stephen H. Muggleton

Department of Computing, Imperial College London, United Kingdom
{rjh09,shm}@doc.ic.ac.uk

Abstract. We investigate how new elements of background knowledge can be abstracted automatically from patterns in programs. The approach is implemented in the *KANDINSKY* system using an algorithm that searches for common subterms over sets of functional programs. We demonstrate that KANDINSKY can invent higher-order functions such as *map*, *fold*, and *sumBy* from small sets of input programs. An experiment shows that KANDINSKY can find high-compression abstractions efficiently, with low settings of its input parameters. Finally we compare our approach with related work in the inductive logic programming and functional programming literature, and suggest directions for further work.

1 Introduction

Can background knowledge be learned automatically through problem-solving experience? This would be a form of *meta-learning* [7], distinct from *base learning* which is concerned simply with solving problem instances. We propose that a general strategy for acquiring new background knowledge can be found in the *abstraction principle* of software engineering. *Abstractions* [1] are re-usable units obtained by separating out and encapsulating patterns in programs. We define *abstraction invention* as the process of formulating useful abstractions in an inductive programming context, and when these abstractions take the form of functions, *Functional Abstraction Invention* (FAI). Some forms of *predicate invention* may be regarded as FAI (since predicates are functions).

We have implemented *KANDINSKY*[1], a system which performs FAI over sets of functional (λ-calculus) programs by *Inverse β-Reduction* (IβR), an analogue of inverse resolution [4,5]. This move from first-order logic to λ-calculus is crucial because it allows our system to invent *higher-order* functional abstractions, an ability that is necessary in order to generalise on arbitrary patterns in programs. See Fig. 1 for an example where first-order methods fail.

```
incElems([],[]).                      doubleElems([],[]).
incElems([H|T],[H1|T1]) :-            doubleElems([H|T],[H1|T1]) :-
    inc(H,H1), incElems(T,T1).            times(2,H,H1), doubleElems(T,T1).
```

Fig. 1. To abstract over the commonality manifest in the above two programs requires quantification over predicate symbols, which is impossible in first-order logic.

[1] Source code available at: http://ilp.doc.ic.ac.uk/kandinsky

Similar Ideas

Learning Programs: A Hierarchical Bayesian Approach

Automatic Invention of Functional Abstractions

Bias reformulation for one-shot function induction

Dianhuan Lin[1] and **Eyal Dechter**[1] and **Kevin Ellis**[1] and **Joshua Tenenbaum**[1] and **Stephen Muggleton**[2]

Abstract. In recent years predicate invention has been under-explored as a bias reformulation mechanism within Inductive Logic Programming due to difficulties in formulating efficient search mechanisms. However, recent papers on a new approach called Meta-Interpretive Learning have demonstrated that both predicate invention and learning recursive predicates can be efficiently implemented for various fragments of definite clause logic using a form of abduction within a meta-interpreter. This paper explores the effect of bias reformulation produced by Meta-Interpretive Learning on a series of Program Induction tasks involving string transformations. These tasks have real-world applications in the use of spreadsheet technology. The existing implementation of program induction in Microsoft's FlashFill (part of Excel 2013) already has strong performance on this problem, and performs one-shot learning, in which a simple transformation program is generated from a single example instance and applied to the remainder of the column in a spreadsheet. However, no existing technique has been demonstrated to improve learning performance over a series of tasks in the way humans do. In this paper we show how a functional variant of the recently developed Metagol$_D$ system can be applied to this task. In experiments we study a regime of layered bias reformulation in which size-bounds of hypotheses are successively relaxed in each layer and learned programs re-use invented predicates from previous layers. Results indicate that this approach leads to consistent speed increases in learning, more compact definitions and consistently higher predictive accuracy over successive layers. Comparison to both FlashFill and human performance indicates that the new system, Metagol$_{DF}$, has performance approaching the skill level of both an existing commercial system and that of humans on one-shot learning over the same tasks. The induced programs are relatively easily read and understood by a human programmer.

1 Introduction

A remarkable aspect of human intelligence is the ability to learn a general principle, concept, or procedure from a single instance. Suppose you were told a computer program outputs "BOB" on input "bob." What will it produce on input "alice"? Will it return "BOB" again, ignoring the input? Or perhaps it will return "BOLICE", postpending to "BO" the all-caps transform of the input minus the first two characters. Is it reasonable to think it returns the all-caps palindrome formed by all but the last letter of the input, so "alice" maps to "ALICILA"? In practice most people will predict the program will return "ALICE", and not any of the above possibilities. Similarly, guessing the program associated with any of the input-output pairs in the rows of Figure 1 seems straightforward, but the space of possible consistent transformations is deceptively large. The reason these

	Input	*Output*
Task1	miKe dwIGHT	Mike Dwight
Task2	European Conference on Artificial Intelligence	ECAI
Task3	My name is John.	John

Figure 1. Input-output pairs typifying string transformations in this paper

problems are easy for us, but often difficult for automated systems, is that we bring to bear a wealth of knowledge about which kinds of programs are more or less likely to reflect the intentions of the person who wrote the program or provided the example.

There are a number of difficulties associated with successfully completing such a task. One is inherent ambiguity: how should one choose from the vast number of consistent procedures? There is no clear objective function to minimize; nor is this objective function a subjective utility an intelligent agent can set arbitrarily since there is generally a consensus regarding the "right" answer. Finally, there is the difficulty of inductive programming in general: the space of such procedures is unruly. Syntactically similar procedures are not in general semantically similar.

It is often most effective and natural to teach another person a new behavior or idea by providing a few examples, and the ease with which someone extrapolates from a few examples seems to be a hallmark of intelligence in general and of expertise in specific domains. To produce intelligent robots and interactive software that can flexibly engage in novel tasks, we need to understand how such learning can be accomplished. The literature on Programming By Example has explored many of these questions, with the goal of producing end-user software that automates repetitive tasks without requiring a programmer's expertise. For the most part, the tools produced by these systems have not reached levels of accuracy, flexibility, and performance suitable for end-user adoption [9].

Recent work by Gulwani et al [8] demonstrates that a carefully engineered Domain Specific Language (DSL) for string transformations allows their system to induce string transformations from a single input-output example with speeds and accuracies suitable for commercial software. In further work [7], they demonstrate that careful crafting of a DSL results in impressive inductive programming in other domains such as intelligent tutoring.

The research presented here is intended as a first response to the challenge Gulwani et al's work seems to pose to AI: if carefully crafted DSLs are a key ingredient for competent one-shot induction of programs, then can we develop AI systems that attain such competence by automatically learning these DSLs?

Metagol$_D$ [13] is an Inductive Logic Programming (ILP) system

[1] Department of Brain and Cognitive Sciences, Massachusetts Institute of Technology, USA
[2] Department of Computing, Imperial College London, UK, email: s.muggleton@imperial.ac.uk

Similar Ideas

Learning Programs: A Hierarchical Bayesian Approach

Automatic Invention of Functional Abstractions

Bias reformulation for one-shot function induction

Optimal Ordered Problem Solver

JÜRGEN SCHMIDHUBER juergen@idsia.ch; www.idsia.ch/~juergen
IDSIA, Galleria 2, 6928 Manno-Lugano, Switzerland

Editors: Christophe Giraud-Carrier, Ricardo Vilalta and Pavel Brazdil

Abstract. We introduce a general and in a certain sense time-optimal way of solving one problem after another, efficiently searching the space of programs that compute solution candidates, including those programs that organize and manage and adapt and reuse earlier acquired knowledge. The Optimal Ordered Problem Solver (OOPS) draws inspiration from Levin's Universal Search designed for single problems and universal Turing machines. It spends part of the total search time for a new problem on testing programs that exploit previous solution-computing programs in computable ways. If the new problem can be solved faster by copy-editing/invoking previous code than by solving the new problem from scratch, then OOPS will find this out. If not, then at least the previous solutions will not cause much harm. We introduce an efficient, recursive, backtracking-based way of implementing OOPS on realistic computers with limited storage. Experiments illustrate how OOPS can greatly profit from metalearning or metasearching, that is, searching for faster search procedures.

Keywords: OOPS, bias-optimality, incremental optimal universal search, efficient planning and backtracking in program space, metasearching and metalearning, self-improvement

1. Introduction

New problems often are more easily solved by reusing or adapting solutions to previous problems. That's why we often train machine learning systems on sequences of harder and harder tasks.

Sometimes we are interested in strategies for solving *all* problems in a given problem sequence. For example, we might want to teach our learner a program that computes $\text{FAC}(n) = 1 \times 2 \times \cdots n$ for any given positive integer n. Naturally, the n-th task in our ordered sequence of training problems will be to find a program that computes $\text{FAC}(i)$ for $i = 1, \ldots, n$.

But ordered problem sequences may also arise naturally without teachers. In particular, new tasks may depend on solutions for earlier tasks. For example, given a hard optimization problem, the n-th task may be to find an approximation to the unknown optimal solution such that the new approximation is at least 1% better (according to some measurable performance criterion) than the best found so far.

In general we would like our learner to continually profit from useful information conveyed by solutions to earlier tasks. To do this in an optimal fashion, the learner may also have to improve the algorithm used to exploit earlier solutions. In other words, it may have to learn a better problem class-specific learning algorithm, that is, to *metalearn*. Is there a general yet time-optimal way of achieving such a feat?

Similar Ideas

Learning Programs: A Hierarchical Bayesian Approach

Automatic Invention of Functional Abstractions

Bias reformulation for one-shot function induction

Optimal Ordered Problem Solver

OPTIMUM SEQUENTIAL SEARCH

R.J. Solomonoff
Oxbridge Research, Box 559
Cambridge, Mass. 02238

REV., 1985

There are two theorems in Levin's "Universal Sequential Search Problems" (1973). The first states the now well-known principle of NP completeness and is followed by an outline of a proof.

The second gives a solution to a very broad class of mathematical problems, but, partly because no proof was suggested in the paper, its great importance is not widely appreciated. It is our purpose to give an outline of Levin's proof and a simple extension of the theorem to another broad class of problems.

This will be followed by a discussion of the significance of these problem solving methods and a technique by which they may be used to obtain practical solutions to problems.

The second theorem gives a method for inverting functions

Thank you!

Questions?