Inducing Domain Specific Languages for Bayesian Program Learning

Anonymous Authors[1]

Abstract

This document provides a basic paper template and submission guidelines. Abstracts must be a single paragraph, ideally between 4–6 sentences long. Gross violations will trigger corrections at the camera-ready phase.

1. Introduction

Imagine an agent faced with a suite of new problems totally different from anything it has seen before. It has at its disposal a basic set of primitive actions it can compose to build solutions to these problems, but it is no idea what kinds of primitives are appropriate for which problems nor does it know the higher-level vocabulary in which solutions are best expressed. How can our agent get off the ground?

The AI and machine learning literature contains two broad takes on this problem. The first take is that the agent should come up with a better representation of the space of solutions, for example, by inventing new primitive actions: see *options* in reinforcement learning (Stolle & Precup, 2002), the EC algorithm in program synthesis (Dechter et al., 2013), or predicate invention in inductive logic programming (Muggleton et al., 2015). The second take is that the agent should learn a discriminative model mapping problems to a distribution over solutions: for example, policy gradient methods in reinforcement learning or neural models of program synthesis (Devlin et al., 2017; Balog et al., 2016). Our contribution is a general algorithm for fusing these two takes on the problem: we propose jointly inducing a representation language, called a *Domain Specific Language* (DSL), alongside a bottom-up discriminative model that regresses from problems to solutions. We evaluate our algorithm on four domains: building Boolean circuits; symbolic regression; FlashFill-style (Gulwani, 2011) string processing problems; and Lisp-style programming problems. We show that EC2.0 can construct a set of basis primitives suitable for discovering solutions in each of these domains

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

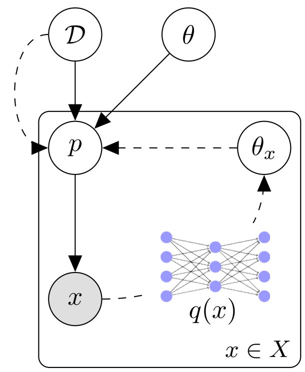


Figure 1: DSL $\mathcal{D}$ generates programs p by sampling DSL primitives with probabilities θ (Algorithm 2). We observe program outputs x. A neural network $q(\cdot)$ called the *recognition model* regresses from program outputs to a distribution over programs ($\theta_x = q(x)$). Solid arrows correspond to the top-down generative model. Dashed arrows correspond to the bottom-up recognition model.

We cast these problems as *Bayesian Program Learning* (BPL; see (Lake et al., 2013; Ellis et al., 2016; Liang et al., 2010)), where the goal is to infer from an observation x a posterior distribution over programs, $\mathbb{P}[p|x]$. A DSL $\mathcal{D}$ specifies the vocabulary in which programs p are written. We equip our DSLs with a *weight vector* θ; together, $(\mathcal{D}, \theta)$ define a probabilistic generative model over programs, $\mathbb{P}[p|\mathcal{D}, \theta]$. In this BPL setting, $\mathbb{P}[p|x] \propto \mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D}, \theta]$, where the likelihood $\mathbb{P}[x|p]$ is domain-dependent. The solid lines in Fig. 1 the diagram this generative model. Alongside this generative model, we infer a bottom-up recognition model, $q(x)$, which is a neural network that regresses from observations to a distribution over programs.

Our key observation is that the generative and recognition models can bootstrap off of each other, greatly increasing the tractability of BPL.

2. The EC2.0 Algorithm

2.1. Motivation and Overview

EC2.0 takes as input a set of *tasks*, written X, each of which is a program induction problem. It has at its disposal a *likelihood model*, written $\mathbb{P}[x|p]$, which scores the likelihood of a task $x \in X$ given a program p. Its goal is to solve each of the tasks by writing a program, and also to infer a DSL $\mathcal{D}$ that distills the commonalities across all of the programs that solve the tasks.

We frame this problem as maximum a posteriori (MAP) inference in the generative model diagrammed in Fig. 1. From Fig. 1, the MAP probability of $(\mathcal{D}, \theta)$ is

$$\mathbb{P}[\mathcal{D}, \theta | X] \propto \mathbb{P}[\mathcal{D}, \theta] \prod_{x \in X} \sum_{p} \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta] \quad (1)$$

If we had a $(\mathcal{D}, \theta)$ maximizing Eq. 1, then we could recover the most likely program for task x by maximizing $\mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D}, \theta]$. Through this lens we now take as our goal to maximize Eq. 1. But even *evaluating* Eq. 1 is intractable because it involves summing over the infinite set of all possible programs. In general, programs are hard-won: finding even a single program that explains a given observation presents a daunting combinatorial search problem.

With this fact in mind, we will instead maximize the following tractable lower bound on Eq. 1, which we call J:

$$J = \mathbb{P}[\mathcal{D}, \theta] \prod_{x \in X} \log \sum_{p \in \mathcal{F}_x} \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta] \quad (2)$$

This lower bound depends on sets of programs, $\{\mathcal{F}_x\}_{x \in X}$:

Definition. The *frontier of task* x, written $\mathcal{F}_x$, is the set of programs discovered by EC2.0 where $\mathbb{P}[x|p] > 0$ for all $p \in \mathcal{F}_x$.

We maximize J by alternatingly maximizing it w.r.t. the DSL and the frontiers:

Program Synthesis: Maximizing J w.r.t. the frontiers. Here we want to find new programs to add to the frontiers so that J increases the most. Adding new programs to the frontiers means attempting to synthesize new programs for each of the tasks. So interleaved with the DSL induction steps are program induction steps. Section 2.2 explains how EC2.0 synthesizes new programs and given a DSL.

DSL Induction: Maximizing J w.r.t. the DSL. Here $\{\mathcal{F}_x\}_{x \in X}$ is held fixed and so we can evaluate J. Now the problem is that of searching the discrete space of DSLs and finding one maximizing J. Section 2.4 explains this step of the algorithm.

In practice, synthesizing programs is extremely difficult because of how large the search space is. Learning the DSL eases the difficulty of synthesis by exposing a domain-specific basis for constructing programs. Another complementary means of meeting search is to learn a bottom-up *recognition model*

2.2. Synthesizing Programs from a DSL

Describe the enumerator, depth first search on the random choices made by the generative model with iterative deepening

Limitation: has a hard time learning arbitrary constants. Solution: solve those constants using specialized solvers.

Algorithm 1 Grammar Induction Algorithm

Input: Set of frontiers $\{\mathcal{F}_x\}$
Hyperparameters: Pseudocounts α, regularization parameter λ, AIC coefficient a
Output: DSL $\mathcal{D}$, weight vector θ
Define $\log \mathbb{P}[\mathcal{D}] \stackrel{+}{=} -\lambda \sum_{p \in \mathcal{D}} \text{size}(p)$
Define $L(\mathcal{D}, \theta) = \prod_x \sum_{z \in \mathcal{F}_x} \mathbb{P}[z|\mathcal{D}, \theta]$
Define $\theta^*(\mathcal{D}) = \arg\max_\theta \text{Dir}(\theta|\alpha) L(\mathcal{D}, \theta)$
Define $\text{score}(\mathcal{D}) = \log \mathbb{P}[\mathcal{D}] + L(\mathcal{D}, \theta^*) - a|\mathcal{D}|$
$\mathcal{D} \leftarrow$ every primitive in $\{\mathcal{F}_x\}$
while true **do**
  N $\leftarrow \{\mathcal{D} \cup \{s\} | x \in X, z \in \mathcal{F}_x, s$ a subtree of $z\}$
  $\mathcal{D}' \leftarrow \arg\max_{\mathcal{D}' \in N} \text{score}(\mathcal{D}')$
  if $\text{score}(\mathcal{D}') > \text{score}(\mathcal{D})$ **then**
    $\mathcal{D} \leftarrow \mathcal{D}'$
  else
    return $\mathcal{D}, \theta^*(\mathcal{D})$
  end if
end while

2.3. Learning a Neural Recognition Model

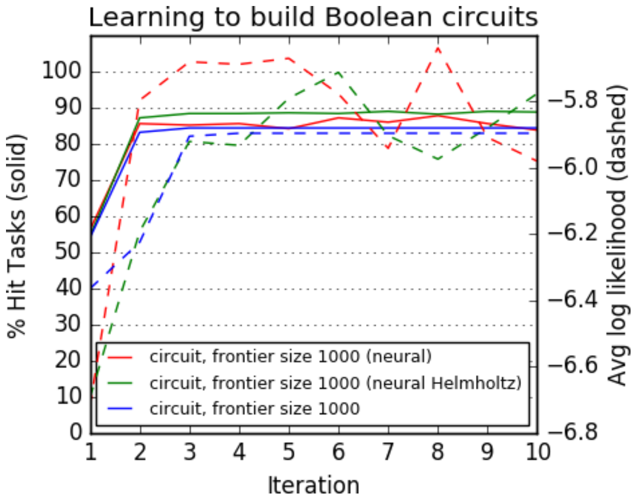
The key idea here is that if the DSL is suitably fit to the domain then we can get away with a simple, low-capacity model.

$$\mathcal{L}_{\text{RM}} = \mathcal{L}_{\text{AE}} + \mathcal{L}_{\text{HM}} \quad (3)$$
$$\mathcal{L}_{\text{AE}} = \mathbb{E}_{x \sim X}\left[\sum_p Q_x(p) \log \mathbb{P}[p|\mathcal{D}, q(x)]\right]$$
$$\mathcal{L}_{\text{HM}} = \mathbb{E}_{p \sim (\mathcal{D}, \theta)}\left[\log \mathbb{P}[p|\mathcal{D}, q(x)]\right], \ p \text{ evaluates to } x$$

2.4. Inducing the DSL from the Frontiers

3. Program Representation

We choose to represent programs using λ-calculus (Pierce, 2002). A λ-calculus expression is either:
A *primitive*, like the number 5 or the function `sum`.
A *variable*, like x, y, z
A λ-*abstraction*, which creates a new function. λ-abstractions have a variable and a body. The body is a λ-calculus expression. Abstractions are written as λvar.body.
An *application* of a function to an argument. Both the function and the argument are λ-calculus expressions. The application of the function f to the argument x is written as $f\ x$.



For example, the function which squares the logarithm of a number is $\lambda x.\texttt{square}(\texttt{log}\ x)$, and the identity function $f(x) = x$ is $\lambda x.x$. The λ-calculus serves as a spartan but expressive Turing complete program representation, and distills the essential features of functional languages like Lisp.

However, many λ-calculus expressions correspond to ill-typed programs, such as the program that takes the logarithm of the Boolean `true` (i.e., `log true`) or which applies the number five to the identity function (i.e., $5\ (\lambda x.x)$). We use a well-established typing system for λ-calculus called *Hindley-Milner typing* (Pierce, 2002), which is used in programming languages like OCaml. The purpose of the typing system is to ensure that our programs never call a function with a type it is not expecting (like trying to take the logarithm of `true`). Hindley-Milner has two important features: Feature 1: It supports *parametric polymorphism*: meaning that types can have variables in them, called *type variables*. Lowercase Greek letters are conventionally used for type variables. For example, the type of the identity function is $\alpha \to \alpha$, meaning it takes something of type α and return something of type α. A function that returns the first element of a list has the type $\texttt{list}(\alpha) \to \alpha$. Type variables are not the same has variables introduced by λ-abstractions. Feature 2: Remarkably, there is a simple algorithm for automatically inferring the polymorphic Hindley-Milner type of a λ-calculus expression (Damas & Milner, 1982). A detailed exposition of Hindley-Milner is beyond the scope of this work.

4. Experiments

4.1. Boolean circuits

pedagogical example; easy domain

4.2. Symbolic Regression

We show how to use EC2.0 to infer programs containing both discrete structure and continuous parameters. The high-level idea is to synthesize programs with unspecified-real-valued parameters, and to fit those parameters using gradient descent. Concretely, we ask the algorithm to solve a set of 1000 symbolic regression problems, each a polynomial of degree 0, 1, or 2, where our observations x take the form of N input/output examples, which we write as $x = \{(i_n, o_n)\}_{n\leq N}$. For example, one task is to infer a program calculating $3x+2$, and the observations are the input-output examples $\{(-1,-1),(0,2),(1,5)\}$.

We initially equip our DSL learner with addition and multiplication, along with the possibility of introducing real-valued parameters, which we write as $\mathcal{R}$. We define the likelihood of an observation x by assuming a Gaussian noise model for the input/output examples and integrate over the real-valued parameters, which we collectively write as $\vec{\mathcal{R}}$:

$$\log \mathbb{P}\left[\{(i_n, o_n)\}|p\right] = \log \int \mathrm{d}\vec{\mathcal{R}}\ P_{\vec{\mathcal{R}}}(\vec{\mathcal{R}}) \prod_{n\leq N} \mathcal{N}(p(i_n, \vec{\mathcal{R}})|o_n)$$

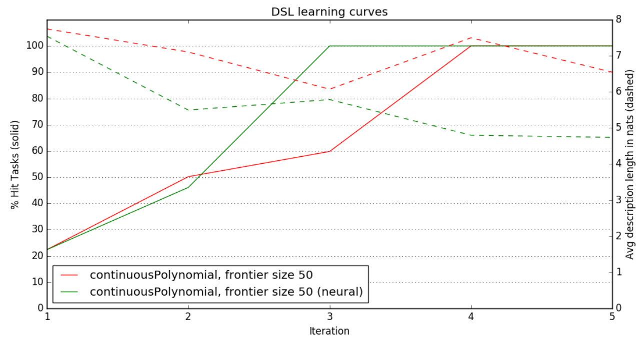
where $\mathcal{N}(\cdot|\cdot)$ is the normal density and $P_{\vec{\mathcal{R}}}(\cdot)$ is a prior over $\vec{\mathcal{R}}$. We approximate this marginal using the BIC (Bishop, 2006):

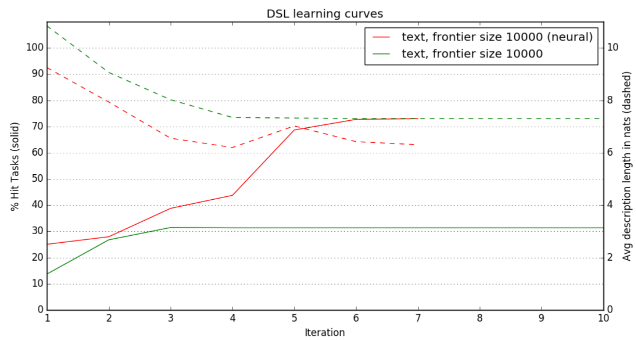
$$\log \mathbb{P}[x|p] \approx \sum_{n\leq N} \log \mathcal{N}(p(i_n, \vec{\mathcal{R}}^*)|o_n) - \frac{D\log N}{2}$$

where $\vec{\mathcal{R}}^*$ is an assignment to $\vec{\mathcal{R}}$ found by performing gradient ascent on the likelihood of the observations w.r.t. $\vec{\mathcal{R}}$.

What DSL does EC2.0 learn? The learned DSL contains templates for quadratic and linear functions, which lets the algorithm quickly hone in on the kinds of functions that are most appropriate to this domain. Examining the programs themselves, one finds that the algorithm discovers representations for each of the polynomials that minimizes the number of continuous degrees of freedom: for example, it represents the polynomial $8x^2 + 8x$

Primitives	$+, \times : \mathbb{R} \to \mathbb{R} \to \mathbb{R}$ $\mathcal{R} : \mathbb{R}$ (real valued parameter)	
Observation x	N input/output examples: $\{(i_n, o_n)\}_{n\leq N}$	
Likelihood $\mathbb{P}[x\|p]$	$\propto \exp\left(-D \log N\right) \prod_{n\leq N} \mathcal{N}(p(i_n)\|o_n)$	
Subset of Learned DSL	$\lambda x.\mathcal{R} \times x + \mathcal{R}$ $\lambda x.\mathcal{R} + x$ $\lambda x.x \times (\text{linear } x)$ $\lambda x.\text{increment } (\text{quadratic}_0\ x)$	linear increment quadratic_0 quadratic





4.3. String editing

4.4. List problems

5. Model

6. Estimating θ

We write $c(e,p)$ to mean the number of times that primitive e was used in program p; $R(p)$ to mean the sequence of types input to sample in Alg.2. Jensen's inequality gives an intuitive lower bound on the likelihood of a program p:

$$\log \mathbb{P}[p|\theta] \stackrel{+}{=} \sum_{e\in\mathcal{D}} c(e,p)\log\theta_e - \sum_{\tau\in R(p)} \log \sum_{\substack{e:\tau'\in\mathcal{D}\\ \text{unify}(\tau,\tau')}} \theta_e$$

$$\stackrel{+}{\geq} \sum_{e\in\mathcal{D}} c(e,p)\log\theta_e - c(p)\log \sum_{\tau\in R(p)} \sum_{\substack{e:\tau'\in\mathcal{D}\\ \text{unify}(\tau,\tau')}} \theta_e$$

$$= \sum_{e\in\mathcal{D}} c(e,p)\log\theta_e - c(p)\log\sum_{e\in\mathcal{D}} r(e,p)\theta_e$$

where $c(p) = \sum_{e\in\mathcal{D}} c(e,p)$ and $r(e:\tau',p) = \sum_{\tau\in R(p)} \mathbb{1}[\text{canUnify}(\tau,\tau')]$.

Differentiate with respect to θ_e and set to zero

$$\frac{c(x)}{\theta_x} = N\frac{a(x)}{\sum_y a(y)\theta_y} \tag{4}$$

Algorithm 2 Generative model over programs

function sample($\mathcal{D},\theta,\mathcal{E},\tau$):
Input: DSL $\mathcal{D}$, weight vector θ, environment $\mathcal{E}$, type τ
Output: a program whose type unifies with τ
if $\tau = \alpha\to\beta$ **then**
  var $\leftarrow$ an unused variable name
  body $\sim$ sample($\mathcal{D},\theta,\{\text{var}:\alpha\}\cup\mathcal{E},\beta$)
  return λvar. body
end if
primitives $\leftarrow \{p|p:\alpha\to\cdots\to\beta\in\mathcal{D}\cup\mathcal{E}$
  if canUnify$(\tau,\beta)\}$
Sample $e\sim$ primitives, w.p. $\propto \theta_e$ if $e\in\mathcal{D}$
  w.p. $\propto \frac{\theta_{var}}{|\text{variables}|}$ if $e\in\mathcal{E}$
Let $e:\alpha_1\to\alpha_2\to\cdots\to\alpha_K\to\beta$. Unify τ with β.
for $k=1$ **to** K **do**
  $a_k\sim$ sample($\mathcal{D},\theta,\mathcal{E},\alpha_k$)
end for
return $e\ a_1\ a_2\ \cdots\ a_K$

This equality holds if $\theta_x = c(x)/a(x)$:

$$\frac{c(x)}{\theta_x} = a(x). \tag{5}$$

$$N\frac{a(x)}{\sum_y a(y)\theta_y} = N\frac{a(x)}{\sum_y c(y)} = N\frac{a(x)}{N} = a(x). \tag{6}$$

If this equality holds then $\theta_x \propto c(x)/a(x)$:

$$\theta_x = \frac{c(x)}{a(x)} \times \underbrace{\frac{\sum_y a(y)\theta_y}{N}}_{\text{Independent of } x}. \tag{7}$$

Now what we are actually after is the parameters that maximize the joint log probability of the data+parameters, which I will write J:

$$J = L + \log \text{D}(\theta|\alpha) \tag{8}$$

$$\stackrel{+}{\geq} \sum_x c(x)\log\theta_x - N\log\sum_x a(x)\theta_x + \sum_x(\alpha_x - 1)\log\theta_x \tag{9}$$

$$= \sum_x (c(x)+\alpha_x-1)\log\theta_x - N\log\sum_x a(x)\theta_x \tag{10}$$

So you add the pseudocounts to the *counts* ($c(x)$), but not to the *possible counts* ($a(x)$).

References

Balog, Matej, Gaunt, Alexander L, Brockschmidt, Marc, Nowozin, Sebastian, and Tarlow, Daniel. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

Algorithm 3 DSL Learner

Input: Initial DSL $\mathcal{D}$, set of tasks X, iterations I
Hyperparameters: Frontier size F
Output: DSL $\mathcal{D}$, weight vector θ, bottom-up recognition model $q(\cdot)$
Initialize $\mathcal{D}_0 \leftarrow \mathcal{D}$, $\theta_0 \leftarrow$ uniform, $q_0(\cdot) = \theta_0$
for $i = 1$ **to** I **do**
  for $x : \tau \in X$ **do**
    $\mathcal{F}_x \leftarrow \{z | z \in \text{enumerate}(\mathcal{D}_{i-1}, q_{i-1}(x), F) \cup \text{enumerate}(\mathcal{D}_{i-1}, \theta_{i-1}, F) \text{ if } \mathbb{P}[x|z] > 0\}$
  end for
  $\mathcal{D}_i, \theta_i \leftarrow$induceGrammar$(\{\mathcal{F}_x\}_{x \in X})$
  Define $Q_x(z) \propto \begin{cases} \mathbb{P}[x|z]\mathbb{P}[z|\mathcal{D}_i, \theta_i] & x \in \mathcal{F}_x \\ 0 & x \notin \mathcal{F}_x \end{cases}$
  $q_i \leftarrow \arg\min_q \sum_{x \in X} \text{KL}(Q_x(\cdot) || \mathbb{P}[\cdot | \mathcal{D}_i, q(x)])$
end for
return $\mathcal{D}^I, \theta^I, q^I$

Bishop, Christopher M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387310738.

Damas, Luis and Milner, Robin. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 207–212. ACM, 1982.

Dechter, Eyal, Malmaud, Jon, Adams, Ryan P., and Tenenbaum, Joshua B. Bootstrap learning via modular concept discovery. In *IJCAI*, pp. 1302–1309. AAAI Press, 2013. ISBN 978-1-57735-633-2. URL http://dl.acm.org/citation.cfm?id=2540128.2540316.

Devlin, Jacob, Uesato, Jonathan, Bhupatiraju, Surya, Singh, Rishabh, Mohamed, Abdel-rahman, and Kohli, Pushmeet. Robustfill: Neural program learning under noisy i/o. *arXiv preprint arXiv:1703.07469*, 2017.

Ellis, Kevin, Solar-Lezama, Armando, and Tenenbaum, Josh. Sampling for bayesian program learning. In *Advances in Neural Information Processing Systems*, 2016.

Gulwani, Sumit. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pp. 317–330. ACM, 2011.

Lake, Brenden M, Salakhutdinov, Ruslan R, and Tenenbaum, Josh. One-shot learning by inverting a compositional causal process. In *Advances in neural information processing systems*, pp. 2526–2534, 2013.

Liang, Percy, Jordan, Michael I., and Klein, Dan. Learning programs: A hierarchical bayesian approach. In Fürnkranz, Johannes and Joachims, Thorsten (eds.), *ICML*, pp. 639–646. Omnipress, 2010. ISBN 978-1-60558-907-7.

Muggleton, Stephen H, Lin, Dianhuan, and Tamaddoni-Nezhad, Alireza. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, 100(1):49–73, 2015.

Pierce, Benjamin C. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

Stolle, Martin and Precup, Doina. Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation*, pp. 212–223. Springer, 2002.