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 domain-specific language 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 (**?**), the EC algorithm in program synthesis (**?**), or predicate invention in inductive logic programming (**?**). 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 (**?**).

Our algorithm accomplishes the following:

- Learns from relatively small amounts of data and with weak supervision. We do not use ground truth programs – weak supervision. Our DSL learner does not need huge amounts of data – tens of examples suffice.
- Jointly infers a *generative model* along with a *recognition model*. The *generative model* is a probabilistic context-sensitive grammar over programs, and includes a DSL. The *recognition model* is a neural network that guides the agents use of the DSL.

[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.

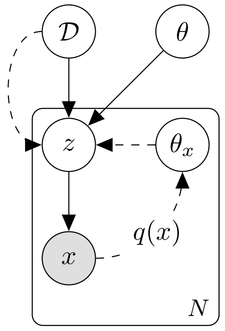


Figure 1: DSL $\mathcal{D}$ generates programs z by sampling DSL primitives with probabilities θ (Algorithm 1). We observe program outputs x. A neural network $q(\cdot)$ called the *recognition model* regresses from program outputs to a distribution over programs ($\theta_x = q(x)$). Solid arrows correspond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

The generative model and the recognition model bootstrap off of each other:

- The recognition model works by upweighting the probability of program components likely to be useful for a given problem. By learning a DSL, the recognition model gets more power because it can upweight new more powerful primitives that are better suited for the domain.
- The generative model (DSL) is learned from programs that the agent has found so far. Because the recognition model speeds up search, it generates more training data for the generative model.

2. 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 (**?**). A detailed exposition of Hindley-Milner is beyond the scope of this work.

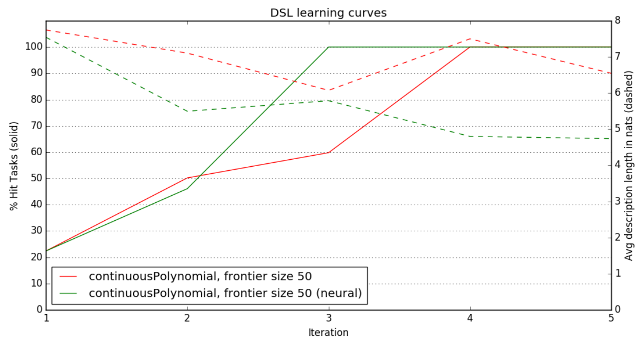
3. Experiments

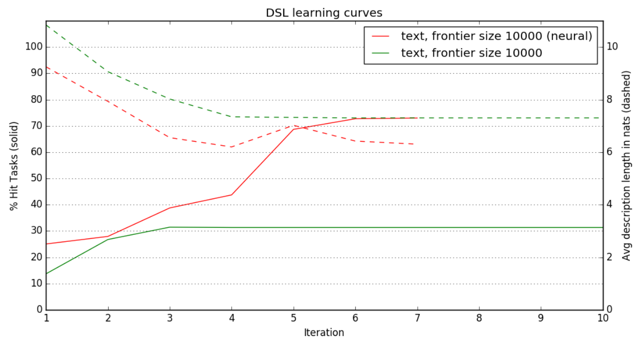
4. Model

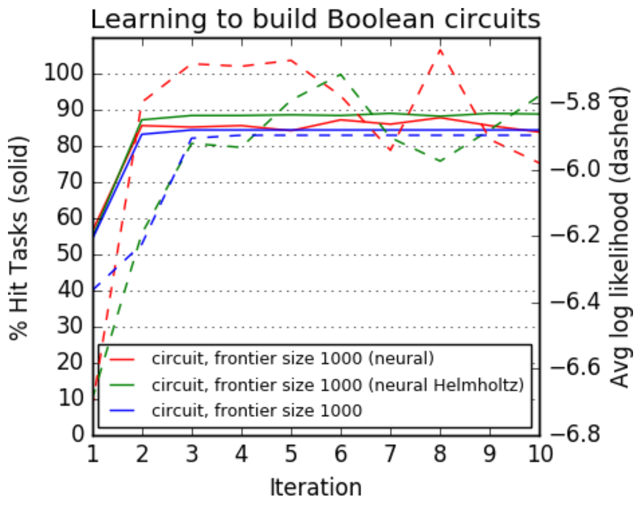
5. Implementation

6. Estimating the grammar parameters

I justify this estimator by proving that it maximizes a lower bound on the log likelihood of the data. Writing L for the log likelihood, θ for the parameters of the grammar, N for the number of random choices, A to range over the alternative choices for a random variable, $c(x)$ to mean the number of times that primitive x was used, and $a(x) = \sum_A \mathbb{1}[x \in A]$ to mean the number of times that primitive x could have







Algorithm 1 Generative model over programs

```
function sample($\mathcal{D}, \theta, \mathcal{E}, \tau$):
Input: DSL $\mathcal{D}$, weight vector $\theta$, environment $\mathcal{E}$, type $\tau$
Output: a program whose type unifies with $\tau$
if $\tau = \alpha \to \beta$ then
    var $\leftarrow$ an unused variable name
    body $\sim$ sample($\mathcal{D}, \theta, [\text{var} : \alpha] + \mathcal{E}, \beta$)
    return $\lambda$var. 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 $e \in \mathcal{D}$ and 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 $\tau$ with $\beta$.
for $k = 1$ to $K$ do
    $a_k \sim$ sample($\mathcal{D}, \theta, \mathcal{E}, \alpha_k$)
end for
return $e(a_1, a_2, \cdots, a_K)$
```

been used:

$$L = \sum_x c(x) \log \theta_x - \sum_A \log \sum_{x \in A} \theta_x \tag{1}$$

$$= \sum_x c(x) \log \theta_x - N \mathbb{E}_A \log \sum_{x \in A} \theta_x \tag{2}$$

$$\geq \sum_x c(x) \log \theta_x - N \log \mathbb{E}_A \sum_{x \in A} \theta_x, \text{ Jensen's inequality} \tag{3}$$

$$= \sum_x c(x) \log \theta_x - N \log \frac{1}{N} \sum_A \sum_x \mathbb{1}[x \in A] \theta_x \tag{4}$$

$$\overset{+}{=} \sum_x c(x) \log \theta_x - N \log \sum_x a(x) \theta_x. \tag{5}$$

Differentiate with respect to θ_x and set to zero:

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

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

$$\frac{c(x)}{\theta_x} = a(x). \tag{7}$$

$$N \frac{a(x)}{\sum_y a(y) \theta_y} = N \frac{a(x)}{\sum_y c(y)} = N \frac{a(x)}{N} = a(x). \tag{8}$$

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

$$\theta_x = \frac{c(x)}{a(x)} \times \underbrace{\frac{\sum_y a(y) \theta_y}{N}}_{\text{Independent of } x}. \tag{9}$$

Algorithm 2 DSL Learner

```
Input: Initial DSL $\mathcal{D}$, set of tasks $X$, iterations $I$
Hyperparameters: Frontier size $F$
Output: DSL $\mathcal{D}$, weight vector $\theta$, 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
        $\mathcal{F}_x \leftarrow \{z | z \in$ enumerate$(\mathcal{D}_{i-1}, q_{i-1}(x), F) \cup$
        enumerate$(\mathcal{D}_{i-1}, \theta_{i-1}, F)$ 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 $\mathcal{D}^I, \theta^I, q^I$
```

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

$$J = L + \log \text{D}(\theta | \alpha) \tag{10}$$

$$\overset{+}{\geq} \sum_x c(x) \log \theta_x - N \log \sum_x a(x) \theta_x + \sum_x (\alpha_x - 1) \log \theta_x \tag{11}$$

$$= \sum_x (c(x) + \alpha_x - 1) \log \theta_x - N \log \sum_x a(x) \theta_x \tag{12}$$

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

References

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

Algorithm 3 Grammar Induction Algorithm

```
Input: Set of frontiers $\{\mathcal{F}_x\}$
Hyperparameters: Pseudocounts $\alpha$, regularization parameter $\lambda$, AIC coefficient $a$
Output: DSL $\mathcal{D}$, weight vector $\theta$
Define $\log \mathbb{P}[\mathcal{D}] \overset{+}{=} -\lambda \sum_{p\in\mathcal{D}} \text{size}(p)$
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 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 $z\}$
  $\mathcal{D}' \leftarrow \arg\max_{\mathcal{D}'\in N}$ score$(\mathcal{D}')$
  if score$(\mathcal{D}')$ > score$(\mathcal{D})$ then
    $\mathcal{D} \leftarrow \mathcal{D}'$
  else
    return $\mathcal{D}, \theta^*(\mathcal{D})$
  end if
end while
```