Learning to Learn Programs from Examples: Going Beyond Program Structure[1]

Kevin Ellis*
MIT
ellisk@mit.edu

Sumit Gulwani
Microsoft
sumitg@microsoft.com

1 Introduction

Billions of people own computers, yet vanishingly few know how to program. Imagine an end user wishing to extract the years from a table of data, like in Table 1. What would be a trivial regular expression for a coder is impossible for the vast majority of computer users. But in many cases, it is easy to show a computer what to do by giving examples – an observation that has motivated a long line of work on the problem of *programming by examples* (PBE), a paradigm where end users give examples of intended behavior and the system responds by inducing and running a program [Lieberman, 2001].

A core problem in PBE is determining which single program the user intended within the vast space of all programs consistent with the examples. Users would like to provide only one or a few examples, leaving the intended behavior highly ambiguous. Consider a user who provides just the first input/output example in Table 1. Did they mean to extract the first number of the input? The last number? The first number after a comma? Or did they intend to just produce "1993" for each input? In real-world scenarios we could encounter on the order of 10^{100} distinct programs consistent with the examples [Singh and Gulwani,]. Getting the right program from fewer examples means less effort for users and more adoption of PBE technology. This concern is practical: Microsoft refused to ship the recent PBE system Flash Fill [Gulwani, 2011] until common scenarios were learned from only one example.

We develop a new inductive bias for resolving the ambiguity that is inherent when learning programs from few examples. Prior inductive biases in PBE use features of the program's syntactic structure, picking either the smallest program consistent with the examples, or the one that looks the most natural according to some learned criterion [Liang *et al.*, 2010; Menon *et al.*, 2013; Singh and Gulwani, ; Lin *et al.*, 2014]. In contrast, we look at the outputs and execution traces of a program, which we will show can sometimes predict program correctness even better than if we could examine the program itself. Intuitively, we ask, "what do typically intended programs compute?" rather than "what do typically intended programs look like?" Returning to Table 1, we prefer the program extracting years because its outputs look like an intended behavior, even though extracting the first number is a shorter program. We apply our technique to a *string transformation* domain, which includes Flash Fill-style problems (eg Table 1); and a *text extraction* domain; see [Le and Gulwani, 2014]. We implement our algorithms within the PROSE [Polozov and Gulwani, 2015] library (https://microsoft.github.io/prose/). We take as a goal to improve PROSE's inductive bias, and use the phrase "PROSE" to refer to the current PROSE implementations of these domains, in contrast to our augmented system.

Input table	Desired output table
Missing page numbers, 1993	1993
64-67, 1995	1995
1992 (1-27)	1992
...	...

Table 1: An everyday computer task trivial for programmers but inaccessible for nonprogrammers: given the input table of strings, automatically extract the year to produce the desired output table on the right.

Predicting program correctness based on its syntactic structure is perhaps the oldest and most successful idea in program induction [Solomonoff, 1964]. But the correctness of a program goes beyond its appearance. We develop two new classes of features that are invariant to program structure, called **output features** and **execution trace features**.

2 Output features

Some sets of outputs are a priori more likely to be produced from valid programs. In PBE scenarios the user typically labels few inputs by providing outputs but has many unlabeled inputs; the candidate outputs on the unlabeled inputs give a semisupervised learning signal that leverages the typically larger set of unlabeled data. See Table 2 and 3. In Table 2, the system considers programs that either append a bracket (a simple program) or ensure correct bracketing (a complex program). PROSE opts for the simple program, but our system notices that program predicts an output too dissimilar from the labeled example. Instead we prefer the program without this "outlier" in its outputs. More generally users expect

*Work done during two internships at Microsoft with the PROSE team

[1]Extended abstract for: [Ellis and Gulwani, 2017]

Input	Output (PROSE)	Output (ours)
[CPT-00350	[CPT-00350]	[CPT-00350]
[CPT-00340	*[CPT-00340]*	*[CPT-00340]*
[CPT-115]	*[CPT-115]]*	*[CPT-115]*

Table 2: Learning a program from one example (top row) and applying it to other inputs (bottom rows, outputs italicized). Our semisupervised approach let us get the last row correct.

Input	Output (PROSE)	Output (ours)
Brenda Everroad	Brenda	Brenda
Dr. Catherine Ramsey	*Catherine*	*Catherine*
Judith K. Smith	*Judith K.*	*Judith*
Cheryl J. Adams and Binnie Phillips	*Cheryl J. Adams and Binnie*	*Cheryl*

Table 3: Learning a program from one example (top row) and applying it to other inputs (bottom rows, outputs italicized). Our semisupervised approach uses simple common sense reasoning, knowing about names, places, words, dates, etc, letting us get the last two rows correct.

programs to produce similarly formatted outputs, such as all being dates, natural numbers, or addresses. This is similar to the idea that programs should be well-typed, and so should predictably output data of a certain type. This is also an analogy to regularizers that prefer smooth functions: here, we might prefer "smooth" programs whose outputs are not too dissimilar.

Concretely, our system learns an inductive bias over program outputs in the form of a probabilistic model that assigns higher likelihood to programs whose outputs on unlabeled examples have similar statistics to the user labeled outputs. Here, "output statistics" is formalized by fitting a generative model to the program outputs – allowing us to insert simple kinds of common sense knowledge into the inductive bias (see Table 3).

3 Execution trace features

Going beyond the final outputs of a candidate program, we show how to consider the entire execution trace. Our model learns a bias over sequences of computations, which allows us to disprefer seemingly natural programs with pathological behavior on the provided inputs. Imagine a spreadsheet of professor names: Rebecca, Oliver, *etc*. One thing you might want a PBE system to do is put the title "Dr." in front of each of these names. So, you give the system an example of "Dr." being prepended to the string "Rebecca." This should be a trivial learning problem, and the system should induce a program that just puts the constant "Dr." in front of the input. However, PROSE failed on this simple case; see Table 4. Although the system can represent the intended program, it instead prefers a program that extracts the first character from "Rebecca" to produce the *r* in "Dr.", with unintended consequences for "Oliver."

Why does PROSE prefer a program that extracts the first character? In general, programs with more constants are less

Input	Output
Rebecca	Dr. Rebecca
Oliver	*Do. Oliver*

Table 4: A string transformation problem; the user provided the first output and an incorrect program produced the italicized second output.

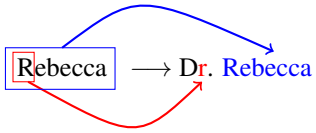

Figure 1: Execution trace for erroneous program with the behavior shown in Table 4. Notice the overlapping substring extractions.

plausible; this is related to the intuition that we should prefer programs with shorter description lengths. Furthermore, the first character of the input is very commonly extracted, so PROSE was tuned to prefer programs that extract prefixes. These two inductive biases conspired to steer the system toward the wrong program.

By looking at the execution trace of the program we discovered a new kind of signal for program correctness. Returning to our motivating example, the erroneous program first extracts a region of the input and then extracts an overlapping region (see Figure 1). Accessing overlapping regions of data is seldom intended: usually programs pull out the data they want and then do something with it, rather than extracting some parts of the data multiple times. More generally one can learn an inductive bias for execution traces by fitting a probabilistic model to traces from intended programs. For our domain we learned a probabilistic model over sequences of substring extractions.

4 Experimental results

We trained a log linear model to predict the intended program using either program, output, or trace features, or their combination. See Figure 2.

Model	Training	Test
Random baseline	13.7%	13.7%
PROSE	76.4%	–
Trace (ours)	56.6%	$46.1 \pm 2\%$
Output (ours)	68.2%	$66.5 \pm 2\ \%$
Program (ours)	77.9%	$57.9 \pm 4\ \%$
All (ours)	88.4%	$\mathbf{83.5 \pm 3\%}$

Figure 2: Accuracy (% test cases where all predicted outputs are correct) of different models. Test accuracies determined by 10-fold cross validation. 477 string transformation test cases.

Program outputs provide a surprisingly strong signal. Output features are lower dimensional than program features; accordingly, predicting based on outputs is less prone to over fitting. Our learned model beats PROSE, *even though PROSE was hand tuned to these particular data sets.* Yet our learned model has higher accuracy even on test cases it did not see than the old system does on the test cases that it did see (all

of them). However, note that success of our system relies on our new classes of features, as our learned model for program structure approximately matches PROSE's accuracy.

5 What does this mean for program induction?

Most program induction algorithms predict the program that jointly minimizes some measure of program cost, plus a term measuring agreement with input/output examples. This scheme is a close analogy to how regression is framed in machine learning.

But programs, whether they be deterministic or probabilistic, procedural or declarative (e.g., in inductive logic programming or grammar induction), expose structure beyond their syntax tree. The execution trace and predictions on unlabeled inputs offer alternative signals for program correctness. Exploiting these signals in other domains is a target for future research.

Acknowledgments

We gratefully acknowledge collaboration with all of the PROSE team at Microsoft, but especially, in no particular order, Vu Le, Daniel Perelman, Alex Polozov, Danny Simmons, Abhishek Udupa, and Adam Smith. We are grateful for feedback from Armando Solar-Lezama and our anonymous reviewers.

References

[Ellis and Gulwani, 2017] Kevin Ellis and Sumit Gulwani. Learning to learn programs from examples: Going beyond program structure. *IJCAI*, 2017.

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

[Le and Gulwani, 2014] Vu Le and Sumit Gulwani. Flashextract: a framework for data extraction by examples. In *ACM SIGPLAN Notices*, volume 49, pages 542–553. ACM, 2014.

[Liang *et al.*, 2010] Percy Liang, Michael I. Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In Johannes Fürnkranz and Thorsten Joachims, editors, *ICML*, pages 639–646. Omnipress, 2010.

[Lieberman, 2001] Henry Lieberman. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[Lin *et al.*, 2014] Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B. Tenenbaum, and Stephen Muggleton. Bias reformulation for one-shot function induction. In *ECAI 2014*, pages 525–530, 2014.

[Menon *et al.*, 2013] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *ICML*, pages 187–195, 2013.

[Polozov and Gulwani, 2015] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices*, 50(10):107–126, 2015.

[Singh and Gulwani,] Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example. In *CAV*.

[Solomonoff, 1964] Ray J Solomonoff. A formal theory of inductive inference. *Information and control*, 7(1):1–22, 1964.