Learning to Learn Programs from Examples: Going Beyond Program Structure

Content areas: program synthesis, machine learning, programming by examples

Abstract

Programming-by-example technologies let end users construct and run new programs by providing examples of the intended program behavior. But, the few provided examples seldom uniquely determine the intended program. Previous approaches to picking a program used a bias toward shorter or more naturally structured programs. Our work here gives a machine learning approach for learning to learn programs that departs from previous work by relying upon features that are independent of the program structure, instead relying upon a learned bias over program behaviors, and more generally over program execution traces. Our approach leverages abundant unlabeled data for semisupervised learning, and incorporates simple kinds of world knowledge for common-sense reasoning during program induction. These techniques are evaluated in two programming-by-example domains, improving the accuracy of program learners.

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 anyone can 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. 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.

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.

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, 2015; 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 in two different PBE domains: a *string transformation* domain, which enriches Flash Fill-style problems (eg Table 1) with semantic transformations, like the ability to parse and transform times and dates [Singh and Gulwani, 2012a] and numbers [Singh and Gulwani, 2012b]; and a *text extraction* domain, where the goal is to learn a program that extracts structured tables out of a log file [Le and Gulwani, 2014]. Flash Fill, now a part of Microsoft Excel, motivated a series of other PBE systems, which coalesced into a software library called PROSE [Polozov and Gulwani, 2015]. In PROSE, one provides the hypothesis space (programming language) and gets a PBE tool for free. But PROSE does not solve the ambiguity problem, instead using a hand-engineered inductive bias over programs. Our work integrates into PROSE and provides a better inductive bias. Although we worked with existing PROSE implementations of the string transformation and text extraction domains, the broad approach is domain-agnostic. 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.

1.1 Our contribution: picking the correct program

We develop two new contributions to PBE technology:

Predictive features

Predicting program correctness based on its syntactic structure is perhaps the oldest and most successful idea in program induction [Solomonoff, 1964]. This general family of approaches use what we call **program features** to bias the learner. But the correctness of a program goes beyond its appearance. We develop two new classes of features that are invariant to program structure:

Output features. Some sets of outputs are apriori 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.

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.

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.

Machine learning framework

We develop a framework for learning to learn programs from a corpus of problems that improves on prior work as follows:

Weak supervision. We require no explicitly provided ground-truth programs, in contrast with eg [Menon *et al.*, 2013; Liang *et al.*, 2010]. This helps automate the engineering of PBE systems because the engineer need not manually annotate solutions to potentially hundreds of problems.

The modeling paradigm. We introduce a *discriminative probabilistic model*, in contrast with [Liang *et al.*, 2010; Menon *et al.*, 2013; Singh and Gulwani, 2015]. A discriminative approach leads to higher predictive accuracy, while a probabilistic framing lets us learn with simple and tractable gradient-guided search.

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.

1.2 Notation

We consider PBE problems where the program, written p, is drawn from a domain specific language (DSL), written $\mathcal{L}$. We have one DSL for string transformation and a different DSL for text extraction. DSLs are described using a grammar that constrains the ways in which program components may be combined. We learn a $p \in \mathcal{L}$ consistent with L labeled input/output examples, with inputs $\{x_i\}_{i=1}^{L}$ (collectively X_L) and user labeled outputs $\{y_i\}_{i=1}^{L}$ (collectively Y_L). We write $p(x)$ for the output of p on input x, so consistency with the labeled examples means that $y_i = p(x_i)$ for $1 \leq i \leq L$. We write N for the total number of inputs on which the user intends to run the program, so that means $N \geq L$. All of these inputs are written $\{x_i\}_{i=1}^{N}$ (collectively X). When a program $p \in \mathcal{L}$ is clear from context, we write $\{y_i\}_{i=1}^{N}$ (collectively Y) for the outputs p predicts on the inputs X. We write $\{y_i\}_{i=L+1}^{N}$ for the predictions of p on the unlabeled inputs (collectively Y_U).

For each DSL $\mathcal{L}$, we assume a simple hand-crafted scoring function that assigns higher scores to shorter or simpler programs. PROSE can enumerate the top K programs under this scoring function, where K is large but manageable (for K up to 10^4). We call these top K programs the **frontier**, and we write $\mathcal{F}_K(X, Y_L)$ to mean the frontier of size K for inputs X and labeled outputs Y_L (so this means that if $p \in \mathcal{F}_K(X, Y_L)$ then $p(x_{i \leq L}) = y_{i \leq L}$). The existing PROSE approach is to predict the single program in $\mathcal{F}_1(X, Y_L)$. We write $\phi(\cdot)$ to mean some kind of feature extractor, and use the variable θ to mean weights placed on those features.

For ease of exposition we will draw our examples from string transformation, where the goal is to learn a program that takes as input a vector of strings (so x_i is a vector of strings) and produces as output a string (so y_i is a string).

2 Extracting predictive features

2.1 Features of program structure

A common intuition in the program induction literature is that one should prefer short, simple programs over long, complicated programs. Many old and modern approaches [Solomonoff, 1964; Liang *et al.*, 2010; Polozov and

Gulwani, 2015; Lau, 2001] realize this intuition by first modeling the set of all programs consistent with the examples, and then picking the program in that set maximizing a measure of program simplicity. The only way in which the examples participate in these program induction approaches is by excluding impossible programs.

We model these **program feature**-style approaches by defining a feature extractor for programs, $\phi_{\text{program}}(p)$. The learner predicts the program p^* (consistent with examples) maximizing a linear combination of these features:

$$p^* = \underset{p \text{ consistent with examples}}{\arg\max} \; \theta \cdot \phi_{\text{program}}(p) \tag{1}$$

This framework models several lines of work: (1) if the scoring function is likelihood under a probabilistic grammar, then $\phi_{\text{program}}(p)$ are counts of the grammar productions used in p and θ are log production probabilities (eg, [Menon *et al.*, 2013]); (2) if the grammar's structure is unknown then $\phi_{\text{program}}(p)$ are counts of all program fragments used (eg, [Liang *et al.*, 2010]); or (3) if the scoring function is the size of the program then $\phi_{\text{program}}(p)$ is the one-dimensional count of the size of the syntax tree (eg, [Lin *et al.*, 2014]).

Our ϕ_{program} counted occurrences of different program primitives, so our model could mimic the inductive bias of a probabilistic grammar. It also detected the presence of domain-specific code templates, for example counting the number of times that a prefix of the input is extracted, or the number of times that an input is parsed as a date. These domain specific choices are motivated by past models that learn a bias towards useful code fragments [Liang *et al.*, 2010], an idea which has been usefully deployed in string transformation domains [Singh and Gulwani, 2015]. But, our contribution is not a more sophisticated preference over programs. Instead, we go beyond this approach by turning to features of program *behaviors*, as the next two sections describe.

2.2 Features of program trace

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 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.

This failure is not an artifact of the fact that PROSE's inductive bias was written by hand rather than being learned from data. With a learned prior over program structures, the model made the exact same error. The program structure alone simply does not provide a strong signal that Oliver should be Dr. Oliver, rather than Do. Oliver.

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.

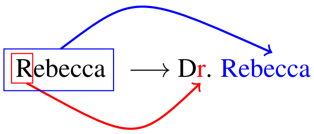

Figure 1: Execution trace for erroneous program with the behavior shown in Table 4. Notice the overlapping substring extractions.

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. Simply introducing an inductive bias against accessing overlapping regions of the input is enough to disprefer the erroneous program in Table 4.

More generally one can learn an inductive bias for execution traces by fitting a probabilistic model to traces from intended programs. This scheme could work for any DSL, with the system using the model to steer the learner towards intended programs.

With these intuitions in hand, we now want an inductive bias over execution traces that strongly penalizes these pathological behaviors. An inductive bias based only on three features sufficed: *Feature 1*: did substring extractions overlap? Correct programs usually pull out the intended data only once, so this feature strongly predicted program incorrectness. *Feature 2*: were substring extractions repeated? This is a weaker signal of incorrectness. *Feature 3*: were substring extractions adjacent? Intended programs often split adjacent inputs, so this weakly signals correctness. We packed these features up into an execution trace feature extractor, $\phi_{\text{trace}}(p, X_L)$, which maps a program and its inputs to the vector of these binary features. Although ϕ_{trace} is tailored to string transformation domains, we stress that the idea of learning an inductive bias over execution traces is more general. Our ϕ_{trace} is just a special case of one such bias.

2.3 Features of program outputs

Users typically expect 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, we calculate the “smoothness” of a program's outputs by first finding a good description of the outputs, called a **descriptor**. We then score a descriptor using a scheme described below. Table 5 gives examples of program outputs paired with their descriptor.

“[CPT-” · Digits · “]”	Name $\vee$ Name · Digits
[CPT-00350]	Mary
[CPT-00340]	John
[CPT-115]	Sue0481

Table 5: We prefer programs whose outputs (bottom rows) have good descriptions (top row), called descriptors. The left descriptor is more likely to correspond to the outputs of a valid program than the right descriptor.

We formalize a preference for “smooth programs” in terms of a regularization-like penalty on programs whose outputs are too dissimilar. For now we assume only that (1) the descriptor is a probabilistic generative model over strings, so we can write $\mathbb{P}(y|D)$ for the probability of descriptor D generating string y; and (2) we can model prior probabilities of descriptors for (in)correct program's outputs, writing $\mathbb{P}(D|\text{correct})$ for the probability of D describing intended outputs, and writing $\mathbb{P}(D|\text{incorrect})$ for unintended outputs.

We consider the log odds ratio of two hypotheses: (1) the candidate program is correct, and so all Y's are the result of the intended program; and (2) the candidate program is incorrect, and so Y_L are the result of a correct program and Y_U are the result of unintended program. This log odds ratio will be our regularizer-like preference for smooth programs. This log ratio is $\log \mathbb{P}(Y|\text{correct}) - \log\left(\mathbb{P}(Y_L|\text{correct})\mathbb{P}(Y_U|\text{incorrect})\right)$. As $\mathbb{P}(Y_L|\text{correct})$ contributes a term independent of the program, we drop it, giving $\log \mathbb{P}(Y|\text{correct}) - \log \mathbb{P}(Y_U|\text{incorrect})$.

We now make some simplifying approximations: if D is the descriptor for Y, then we approximate $\mathbb{P}(Y|\text{correct})$ by a lower bound $\mathbb{P}(D|\text{correct}) \prod_{y \in Y} \mathbb{P}(y|D)$. We similarly approximate $\mathbb{P}(Y_U|\text{incorrect})$ by $\mathbb{P}(D|\text{incorrect}) \prod_{y \in Y_U} \mathbb{P}(y|D)$. Assume a log linear prior over D, so $\mathbb{P}(D|k) \propto \exp(\phi(D) \cdot \theta_k)$ where $\phi(\cdot)$ is a feature extractor for descriptors, θ_k is a weight vector, and $k \in \{\text{correct}, \text{incorrect}\}$. These approximations give the final expression for our inductive bias over program outputs:

$$\theta \cdot \phi(D) + \sum_{y \in Y_L} \log \mathbb{P}(y|D) \tag{2}$$

The first term in Eq. 2 says to prefer outputs whose descriptor D has certain features - for example, not containing outliers or not containing empty strings or containing common sense categories like names or cities. The second term says to prefer outputs whose descriptor D puts high probability mass on the outputs the user actually provided. In summary, smooth programs have “smooth” descriptors and the labeled outputs are typical instances of something sampled from the descriptor.

2.4 Representing and Scoring Descriptors

Representing descriptors. We want descriptors to encode typical patterns within program outputs. To achieve this goal, we model descriptors as mixtures (disjunctions) of regular expressions. We restrict the allowed regular expressions to be sequences of expressions chosen from a predefined set of elements called **tokens**. For example, Table 5 shows the descriptor Name $\vee$ Name · Digits, which is a mixture of Name and Name · Digits regular expressions, the latter of which is the concatenation of the Name and Digits tokens. We built in about 30 tokens. Because descriptors also serve as probabilistic generative models over strings, we equip each token T with a likelihood model $\mathbb{P}_{y|T}$ over strings y. See Table 6.

Token	Regular expression	Likelihood $\mathbb{P}_{y\|T}(\cdot\|\cdot)$
Ampersand	`&`	1
Lowercase	`[a-z]`	$\frac{1}{26}$
EnglishWord	`(a|the|by|...)`	$\propto$(word frequency)

Table 6: Descriptors are mixtures of regular expressions. Each regex is a sequence of “tokens”, some of which are shown above. We built in about 30 tokens.

Inferring descriptors. We treat the problem of computing the descriptor as one of probabilistic inference: given some program outputs, what is the most likely descriptor? This is an unsupervised clustering problem. Conditioned on strings $Y = \{y_i\}_{i=1}^N$, we find the most likely a posteriori regular expressions (written $\{r_j\}$) and cluster assignments (written $\{z_i\}_{i=1}^N$, where z_i indexes the cluster for y_i).

Unlike some mixture models, we don't know ahead of time the number of mixture components (i.e. regular expressions). So we borrow a key model from Bayesiana nonparametrics called the Chinese Restaurant Process (CRP) [Gershman and Blei, 2012], a generative model over cluster assignments that does not assume a fixed number of clusters.

Our strategy for inference is to first marginalize over the regular expressions and (approximately) maximize the joint likelihood of the outputs and the cluster assignments:

$$\log \text{CRP}(\{z_i\}_{i=1}^N) + \sum_z \log \sum_r \mathbb{P}(r) \prod_{i:z_i=z} \mathbb{P}(y_i|r) \tag{3}$$

The marginal probability $\sum_r \mathbb{P}(r) \prod_{y \in Y} \mathbb{P}(y|r)$ can be calculated using a dynamic program that recurses on suffixes of r and Y. This dynamic program lets us efficiently integrate out the regular expressions and evaluate the likelihood of a clustering assignment. Unfortunately there is no similar trick for finding the most likely cluster assignments, so we performed a greedy agglomerative search to locally maximize Equation 3. In practice, this inference strategy allows us to compute most descriptors in a handful of milliseconds - a prerequisite for our system's use in real-world PBE applications.

Extracting features from a descriptor. We can now compute the descriptor for a program's outputs and use Eq. 2 to pick a program with “smooth” outputs. Here, we bring these ideas together to define a feature extractor, $\phi_{\text{output}}(p, X)$.

We extract features of D that distinguish the descriptors of correct and incorrect outputs. Returning to the derivations in Section 2.3, these correspond to the ways in which priors over (in)correct descriptors differ. About a dozen features of D were useful; see Table 7.

The $\log \mathbb{P}(y_i|D)$ term of Eq. 2 is $\log \mathbb{P}(y_i|r_{z_i}) + \log \mathbb{P}(z_i|\{z_{\neq i}\})$. Exploiting the exchangability of the CRP, $\mathbb{P}(z_i|\{z_{\neq i}\}) \propto |\{j : z_j = z_i, j \neq i\}|$. In other words, the

Feature	Intuition
# clusters	Fewer clusters $\Longrightarrow$ fewer outliers
# empty regexes	Failure to produce output $\Longrightarrow$ incorrect
# constant strings	Correct programs have variable outputs

Table 7: Some features of the descriptor that predict program (in)correctness. About a dozen features used.

log likelihood in Eq. 2 breaks down into two terms: one is the probability of a user-labeled output given its regex in D, and another is proportional to the size of the cluster containing the user labeled outputs. This allows us to prefer descriptors that put labeled outputs in larger clusters, which captures the intuition that the labeled outputs should not be "outliers". In practice we found it useful to break these two terms up as separate features. The form of $\phi_{\text{output}}(p, X)$ is

$$\left[\phi(D);\ \sum_{i=1}^{L} \log \mathbb{P}(y_i|r_{z_i});\ \sum_{i=1}^{L} \log(\text{ClusterSize}(z_i) - 1)\right]$$

3 Learning to pick a program

3.1 Probabilistic model

Given our feature extractors, we want to learn a model that predicts which program outputs the user intended. We placed a log-linear probabilistic model over programs parameterized by a real-valued vector θ. So we define $\mathbb{P}(p|X;\theta) \propto \exp(\theta \cdot \phi(p, X))$ However, our main task is predicting the correct program *outputs*, and this is also where the actual supervision signal comes from. We model the probability of predicting outputs Y as the marginal probability of predicting any of the programs that produce those outputs:

$$\mathbb{P}(Y|X, Y_L;\theta) \propto \sum_{\substack{p:\\ p(x_i)=y_i}} \exp(\theta \cdot \phi(p, X)) \qquad (4)$$

At test time we predict the most likely outputs Y^* in the frontier $\mathcal{F}_K(X, Y_L)$:

$$Y^* = \arg\max_{Y} \sum_{\substack{p\in\mathcal{F}_K(X,Y_L)\\ p(x_i)=y_i}} \exp(\theta \cdot \phi(p, X))$$

3.2 Inferring the model parameters

Our goal now is to find model parameters θ so that the model usually predicts the intended program outputs. We assume a data set of PBE problems, each of which is a triple of inputs, labeled outputs, and all outputs: (X, Y_L, Y).

One could pick a θ maximizing the likelihood of the data set (ie $\mathbb{E}[\log \mathbb{P}(Y|X, Y_L;\theta)]$). However, since our true objective is to maximize the fraction of PBE problems we get correct, directly minimizing a loss function more closely matching this gave higher predictive accuracy. Specifically we maximize the expected number of problems the model gets correct, where the expectation is taken both over the problem (X, Y_L, Y) and the model prediction in Eq. 4. Intuitively this is a "softened" measurement of the model's accuracy that gets partial credit for almost getting problems correct. So we want the best model parameters θ^* according to:

$$\theta^* = \arg\max_{\theta} \mathbb{E}\left[\mathbb{P}(Y|X, Y_L;\theta)\right] \qquad (5)$$

Eq. 5 has no closed form solution and is nonconvex, but is differentiable. We locally maximize it using RMSProp.

4 Experimental results

We used a dataset of 447 string transformation and 488 text extraction problems. The number of examples given to each problem was increased until a correct program was in a size 1000 frontier. This strategy resulted in all of the text extraction problems having one example, while 91%, 8%, or 1% of the string transformation problems had 1, 2, or 3 examples.

4.1 Accuracy of the learned program

How often does our system predict a correct program? We considered four variants of our system; see Figure 2. (1) *Trace*, which predicts program correctness based only on its execution trace (applicable only to string transformation); (2) *Output*, which predicts based only on its outputs; (3) *Program*, which predicts based on its syntactic structure; and (4) *All*, which combines these features. Although our approach consistently improves upon PROSE, it is also helped out by PROSE, which provides the frontier. So we compare with a baseline which picks an output uniformly at random from the frontier (*Random baseline* in Figure 2). This baseline's poor performance shows that the structure of the frontier alone is not a strong signal from which to judge program correctness.

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 ± 2 %
Program (ours)	77.9%	57.9 ± 4 %
All (ours)	88.4%	$\mathbf{83.5 \pm 3}$**%**

(a) String transformation

Model	Training	Test
Random baseline	14.7%	14.7%
PROSE	65.8%	–
Output (ours)	70.5%	$\mathbf{68.2 \pm 1}$**%**
Program (ours)	63.9%	$49.9 \pm 1\%$
All (ours)	79.3%	$\mathbf{69.2 \pm 2}$**%**

(b) Text extraction

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

Program outputs provide a surprisingly strong signal. Output features are lower dimensional than program features because descriptors have simpler structures than programs; accordingly, predicting based on outputs is less prone to over fitting. Even on the test data, the program outputs can give a better signal than the program's structure (see Figure 2b).

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.

4.2 Overhead of the approach

Our approach confers greater accuracy at the expense of increased computation. PROSE need only find the top program in the frontier, but our approach needs to enumerate many programs, run them, and then get the descriptors of their outputs. This introduces a trade-off between performance and accuracy: enumerating larger frontiers increases the chance of discovering a correct program, but we have to wait longer.

How long do we spend computing descriptors? Figure 3 plots the relationship between time spent computing descriptors and fraction of problems solved, both of which increase with frontier size. To get most of the benefit of our approach, we need only spend about 1 second computing descriptors. The inference algorithm for descriptors is highly optimized so that this technique can be used in the real-world.

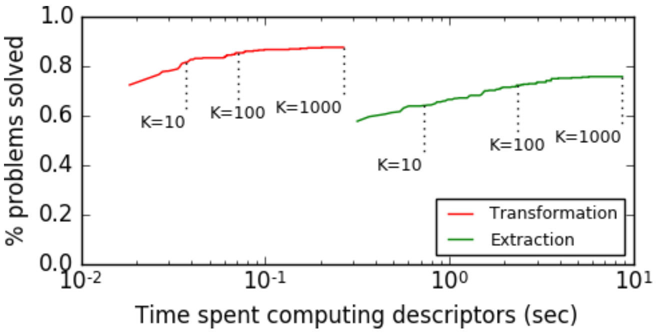

Figure 3: Enumerating more programs increases accuracy, because we get more chances at enumerating a correct program, but incurs additional overhead because we compute a descriptor for each distinct prediction. K = frontier size.

$K = 1$	$K = 10$	$K = 100$	$K = 1000$
449 ms	516 ms	2042 ms	2943 ms

Table 8: Overhead of enumerating top K string transformation programs. Compare with Figure 3.

How long do we spend enumerating frontiers? In practice, this was slowest for string transformation programs, which can involve relatively difficult to synthesize operations like number or date transformations. Table 8 shows that we need to spend a couple seconds on average enumerating frontiers to get the most of our approach.

We envision our system working in two regimes. One is where a data scientist is wrangling large data sets, and absolutely must get the program right. Here it is worth waiting an extra few seconds to get our best guess for the correct program. Alternatively, if an ordinary user is manipulating a smaller spreadsheet, text file, or webpage, we prefer responsiveness over accuracy, and so suggest a program based on a small frontier. In the background we could calculate our best predictions, so that if the user indicates that the program got it wrong, we can immediately respond with a better suggestion.

5 Discussion

5.1 Related work

Picking the right program is a key problem in PBE which has received attention from scientists in several research areas. Work in Human Computer Interaction has designed interfaces for letting users navigate the space of consistent programs and choose the intended behavior [Mayer *et al.*, 2015]. This complements our work: in practice users cannot explore the entire space, so it is important to propose only the most plausible candidates. Researchers in Inductive Logic Programming have used a bias towards more compressive logic programs [Muggleton *et al.*, 2015; Lin *et al.*, 2014]. Machine learning researchers explored similar inductive biases by learning priors over programs [Menon *et al.*, 2013; Liang *et al.*, 2010]. The programming languages community has put forth similar models [Le and Gulwani, 2014; Gulwani, 2011; Gulwani *et al.*, 2015], some of which learns from a corpus of problems [Singh and Gulwani, 2015].

The implementation details of our feature extractors would not generalize to learning, for example, graphics programs or dynamic programming algorithms, but the idea of moving beyond program appearance could be applied to these and other domains. Looking at *what* was computed has precedence in other fields: linguistics calls it *optimality theory* [Prince and Smolensky, 2008]; cognitive scientists used related ideas to model analogy [Hofstadter *et al.*, 1994]. Examining *how* the program computes has analogues in theoretical models of induction, like the *speed prior* [Schmidhuber, 2002].

We see semisupervised learning as the default regime to consider in future PBE systems. Although users label few examples, there is usually lots of unlabeled data. The new system BlinkFill [Singh, 2016] also leverages unlabeled data. We see their approach as complementary to ours: while we analyze the program outputs and traces, they analyze the inputs. Quantitative comparison is difficult as their tool is not yet public. We note that while our semisupervised signal is invariant to the program representation, BlinkFill's is closely tied to it, which limited its application to a restricted subset of Flash Fill, whereas we deployed our approach on both a superset of Flash Fill and a different domain (text extraction).

5.2 Future work

The applications of program induction are much wider than presented here: synthesis of smartphone scripts [Le *et al.*, 2013]; creating XML/tree transformers [Feng *et al.*, 2016]; systems that learn from natural language [Liang *et al.*, 2011; Raza *et al.*, 2015]; intelligent tutoring systems [Gulwani, 2014]; and induction of graphics programs [Cheema *et al.*, 2012; Šťava *et al.*, 2010]. Our motivating intuitions – learning an inductive bias over program behaviors and predictions; incorporating commonsense knowledge of the world – could be exploited in domains like these.

References

[Cheema *et al.*, 2012] Salman Cheema, Sumit Gulwani, and Joseph LaViola. Quickdraw: improving drawing experience for geometric diagrams. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1037–1064. ACM, 2012.

[Feng *et al.*, 2016] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. *PLDI*, 2016.

[Gershman and Blei, 2012] Samuel J Gershman and David M Blei. A tutorial on bayesian nonparametric models. *Journal of Mathematical Psychology*, 56(1):1–12, 2012.

[Gulwani *et al.*, 2015] Sumit Gulwani, Jose Hernandez-Orallo, Emanuel Kitzelmann, Stephen Muggleton, Ute Schmid, and Ben Zorn. Inductive programming meets the real world. *Commun. ACM*, 2015.

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

[Gulwani, 2014] Sumit Gulwani. Example-based learning in computer-aided stem education. *Communications of the ACM*, 57(8):70–80, 2014.

[Hofstadter *et al.*, 1994] Douglas R Hofstadter, Melanie Mitchell, et al. The copycat project: A model of mental fluidity and analogy-making. 1994.

[Lau, 2001] Tessa Lau. *Programming by demonstration: a machine learning approach*. PhD thesis, University of Washington, 2001.

[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.

[Le *et al.*, 2013] Vu Le, Sumit Gulwani, and Zhendong Su. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 193–206. ACM, 2013.

[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.

[Liang *et al.*, 2011] P. Liang, M. I. Jordan, and D. Klein. Learning dependency-based compositional semantics. In *Association for Computational Linguistics (ACL)*, pages 590–599, 2011.

[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.

[Mayer *et al.*, 2015] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. User interaction models for disambiguation in programming by example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software &#38; Technology*, UIST '15, pages 291–301, New York, NY, USA, 2015. ACM.

[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.

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

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

[Prince and Smolensky, 2008] Alan Prince and Paul Smolensky. *Optimality Theory: Constraint interaction in generative grammar*. John Wiley & Sons, 2008.

[Raza *et al.*, 2015] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[Schmidhuber, 2002] Jürgen Schmidhuber. The speed prior: a new simplicity measure yielding near-optimal computable predictions. In *International Conference on Computational Learning Theory*, pages 216–228. Springer, 2002.

[Singh and Gulwani, 2012a] Rishabh Singh and Sumit Gulwani. Learning semantic string transformations from examples. *Proceedings of the VLDB Endowment*, 5(8):740–751, 2012.

[Singh and Gulwani, 2012b] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *International Conference on Computer Aided Verification*, pages 634–651. Springer, 2012.

[Singh and Gulwani, 2015] Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example. In *International Conference on Computer Aided Verification*, pages 398–414. Springer, 2015.

[Singh, 2016] Rishabh Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proceedings of the VLDB Endowment*, 2016.

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

[Št'ava *et al.*, 2010] O Št'ava, B Beneš, R Měch, Daniel G Aliaga, and P Krištof. Inverse procedural modeling by automatic generation of l-systems. In *Computer Graphics Forum*, volume 29, pages 665–674. Wiley Online Library, 2010.