A Machine Learning Framework for Programming by Example

Aditya Krishna Menon[1] AKMENON@UCSD.EDU
Omer Tamuz[2] OMER.TAMUZ@WEIZMANN.AC.IL
Sumit Gulwani[3] SUMITG@MICROSOFT.COM
Butler Lampson[4] BUTLER.LAMPSON@MICROSOFT.COM
Adam Tauman Kalai[4] ADUM@MICROSOFT.COM

[1]University of California, San Diego, 9500 Gilman Drive, La Jolla CA 92093, USA
[2]Faculty of Mathematics and Computer Science,The Weizmann Institute of Science, Rehovot Israel
[3]Microsoft Research, One Microsoft Way, Redmond, WA 98052
[4]Microsoft Research, One Memorial Drive, Cambridge MA 02142

Abstract

Learning programs is a timely and interesting challenge. In Programming by Example (PBE), a system attempts to infer a program from input and output examples alone, by searching for a composition of some set of base functions. We show how machine learning can be used to speed up this seemingly hopeless search problem, by learning weights that relate textual features describing the provided input-output examples to plausible sub-components of a program. This generic learning framework lets us address problems beyond the scope of earlier PBE systems. Experiments on a prototype implementation show that learning improves search and ranking on a variety of text processing tasks found on help forums.

1. Introduction

An interesting challenge is that of learning *programs*, a problem with very different characteristics to the more well-studied goal of learning real-valued functions (regression). Our practical motivation is *Programming by Example* (PBE) (Lieberman, 2001; Cypher et al., 1993), where an end user provides a machine with examples of a task she wishes to perform, and the machine infers a program to accomplish this. The study of PBE is timely; the latest version of Microsoft Excel ships with "Flash Fill," a PBE algorithm for simple string manipulation in spreadsheets, such as splitting and concatenating strings (Gulwani, 2011). We are interested in using learning for PBE to enable richer classes of programs such as, for example, text-processing tasks that might normally be accomplished in a programming language such as PERL or AWK.

Learning programs poses two key challenges. First, users give very few examples per task. Hence, one must impose a strong bias, learned across multiple tasks. In particular, say for task t the user provides n_t examples of pairs of strings $(\bar{x}_1^{(t)}, \bar{y}_1^{(t)}), \ldots, (\bar{x}_{n_t}^{(t)}, \bar{y}_{n_t}^{(t)})$ that demonstrate the task (in our application we will generally take $n_t = 1$). There are infinitely many consistent functions f such that $f(\bar{x}_i^{(t)}) = \bar{y}_i^{(t)}$. Hence, one requires a good *ranking* over such f, learned across the multiple tasks. Simply choosing the shortest consistent program can be improved upon using learning. For one, the shortest program mapping $\bar{x}$ to $\bar{y}$ may very well be the trivial constant function $f(x) = \bar{y}$.

Second, and perhaps even more pressing, is searching over arbitrary compositions of functions for consistent candidate functions. In many cases, finding *any* (non-trivial) consistent function can be a challenge, let alone the "best" under some ranking. For any sufficiently rich set of functions, this search problem defies current search techniques, such as convex optimization, dynamic programming, or those used in Flash Fill. This is because program representations are wildly unstable – a small change to a program can completely change the output. Hence, local heuristics that rely on progress in terms of some metric such as edit distance, will be trapped in local minima.

We introduce a learning-based framework to address both the search and ranking problems. Of particu-

Proceedings of the 30th International Conference on Machine Learning, Atlanta, Georgia, USA, 2013. JMLR: W&CP volume 28.

lar interest, machine learning *speeds up* search (inference). This is unlike earlier work on string processing using PBE, which restricted the types of programs that could be searched through so that efficient search would be possible using so-called version space algebras (Lau et al., 2000). It is also unlike work in structured learning that uses dynamic programming for efficient search. The types of programs we can handle are more general than earlier systems such as SMARTedit (Lau et al., 2000), LAPIS (Miller, 2002), Flash Fill (Gulwani, 2011), and others (Nix, 1985; Witten & Mo, 1993). In addition, our approach is more extensible and broadly applicable than earlier approaches.

How can one improve on brute force search for combining functions from a general library? In general, it seems impossible to speed up search if all one can tell is whether a program correctly maps example $\bar{x}$ to $\bar{y}$. The key idea in our approach is to augment the library with certain telling *textual features* on example pairs. These features suggest which functions are more likely to be involved in the program. As a simple example beyond the scope of earlier PBE systems, consider sorting a list of names by last name. Say the user gives just one example pair: $\bar{x}$ is a list of a few names, one per line, where each line is in the form FirstName LastName, and $\bar{y}$ is the same list sorted by last name. One feature of $(\bar{x}, \bar{y})$ is that the lines are permutations of one another, which is a clue that the desired program may involve sorting. We *learn* the reliability of such clues, and use these to dramatically speed up search and inference in complex examples.

The contributions of this work are: (i) a framework for learning general programs that *speeds up search* (inference) and is also used for ranking. Previous work addressed ranking (Liang et al., 2010) and used restricted classes of programs that could be efficiently searched but not extended; (ii) the use of features relating input and output examples in PBE. Previous work such as Liang et al. (2010) use features of the target *programs* in the training set; (iii) advancing the state of the art in PBE by introducing a general, extensible framework, that can be applied to tasks beyond the scope of earlier systems (as discussed shortly). While our discussion is in the context of text processing, the approach could be adapted for different domains; and (iv) experiments with a prototype system on data extracted from help forums. To clarify matters, we step through a concrete example of our system's operation.

1.1. Example of our system's operation

Imagine a user has a long list of names with some repeated entries (say, the Oscar winners for Best Actor), and would like to create a list of the unique names, each annotated with their number of occurrences. Following the PBE paradigm, in our system, the user illustrates the operation by providing an example, which is an input-output pair of strings. Figure 1 shows one possible such pair, which uses a subset of the full list (in particular, the winners from '91–'95) the user possesses.

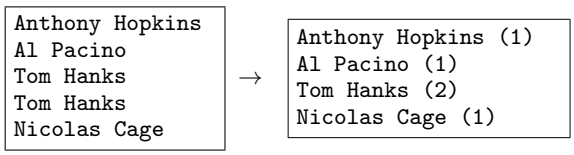


Figure 1. Input-output example for the desired task.

One way to perform the above transformation is to first generate an intermediate list where each element of the input list is appended with its occurrence count – which would look like `[ "Anthony Hopkins (1)", "Al Pacino (1)", "Tom Hanks (2)", "Tom Hanks (2)", "Nicolas Cage (1)"]` – and then remove duplicates. The corresponding program $f(\cdot)$ may be expressed as the composition

$$f(x) = \texttt{dedup}(\texttt{concatLists}(x, \text{“ ”}, \texttt{concatLists}(\text{“(”}, \texttt{count}(x, x), \text{“)”}))). \quad (1)$$

The argument x here represents the list of input lines that the user wishes to process, which may be much larger than the input provided in the example. We assume here a base language comprising (among others) a function `dedup` that removes duplicates from a list, `concatLists` that concatenates lists of strings elementwise, implicitly expanding singleton arguments, and `count` that finds the number of occurrences of the elements of one list in another.

While conceptually simple, this example is out of scope for existing text processing PBE systems. Most systems support a restricted, pre-defined set of functions that do not include natural tasks like removing duplicates; for example (Gulwani, 2011) only supports functions that operate on a line-by-line basis. These systems perform inference with search routines that are hand-coded for their supported functionality, and are thus not easily extensible. (Even if an exception could be made for specific examples like the one above, there are countless other text processing applications we would like to solve.)

Notice that certain textual features can help bias our search by providing clues about which functions may be relevant: in particular, (a) there are duplicate lines in the input but not output, suggesting

Table 1. Example of grammar rules generated for task in Figure 1.

Production	Probability	Production	Probability
P→join(LIST,DELIM)	1	CAT→LIST	0.7
LIST→split(x,DELIM)	0.3	CAT→DELIM	0.3
LIST→concatList(CAT,CAT,CAT)	0.1	DELIM→"\n"	0.5
LIST→concatList("(",CAT,")")	0.2	DELIM→" "	0.3
LIST→dedup(LIST)	0.2	DELIM→"("	0.1
LIST→count(LIST,LIST)	0.2	DELIM→")"	0.1

that `dedup` may be useful, (b) there are parentheses in the output but not input, suggesting the function `concatLists("(",`L`,")")` for some list L, (c) there are numbers on each line of the output but none in the input, suggesting that `count` may be useful, and (d) there are many more spaces in the output than the input, suggesting that `" "` may be useful. Our claim is that by *learning* weights that tell us the reliability of these clues – for example, how confident can we be that duplicates in the input but not the output suggests `dedup` – we can significantly speed up the inference process over brute force search.

In more detail, a clue is a function that generates rules in a probabilistic context free grammar based on features of the provided example. Dynamic programming common in structured learning approaches with PCFGs (Rush et al., 2011) does not apply here – it would be relevant if we were given a program and wanted to parse it. Instead we generate programs according to the PCFG and then evaluate them directly.

Each rule corresponds to a function[1] (possibly with bound arguments) or constant in the underlying programming language. The rule probabilities are computed from weights on the clues that generate them, which in turn are learned from a training corpus of input-output examples. To learn $f(\cdot)$, we now search through derivations of this grammar in order of decreasing probability. Table 1 illustrates what the grammar may look like for the above example. Note that the grammar rules and probabilities are *example specific*; we do not include a rule such as `DELIM→ "$"`, say, because there is no instance of `"$"` in the input or output. Further, compositions of rules may also be generated, such as `concatList("(",LIST,")")`.

Table 1 is of course a condensed view of the actual grammar our prototype system generates, which is based on a large library of about 100 features and clues. With the full grammar, a naïve brute force search over compositions takes 30 seconds to find the right solution to the example of Figure 1, whereas with learning the search terminates in just 0.5 seconds.

[1]When we describe clues as suggesting functions, we implicitly mean the corresponding grammar rule.

1.2. Comparison to previous learning systems

Most previous PBE systems for text processing handle a relatively small subset of natural text processing tasks. This is in order to admit efficient representation and search over consistent programs, e.g. using a version space (Lau et al., 2003), thus sidestepping the issue of searching for programs using general classes of functions. To our knowledge, every system designed for a library of arbitrary functions searches for appropriate compositions of functions either by brute force search, a similarly intractable operation such as invoking a SAT/SMT solver (Jha et al., 2010), or by using A*-style heuristics (Gulwani et al., 2011). (Gulwani, 2012) presents a survey of such techniques. Our learning approach based on textual features is thus more general and flexible than previous approaches.

This said, our goal in this paper is *not* to compete with existing PBE systems in terms of functionality. Instead, we wish to show that the fundamental PBE inference problem may be attacked by learning with textual features. This idea could in fact be applied in *conjunction* with prior systems. A specific feature of the data, such as the input and output having the same number of lines, may be a clue that a function corresponding to a system like Flash Fill (Gulwani, 2011) will be useful.

2. Formalism of our approach

We begin a formal discussion of our approach by defining the learning problem in PBE.

2.1. Programming by example (PBE)

Let $\mathcal{S}$ denote the set of strings. Suppose the user has some text processing operation in mind, in the form of some *target function* or *program* $f \in \mathcal{S}^{\mathcal{S}}$, from the set of functions that map strings to strings. For example, in Figure 1, f could be the function in Equation 1. To describe this program to a PBE system, at *inference* (or *execution*) time, the user provides a *system input* $z := (x, \bar{x}, \bar{y}) \in \mathcal{S}^3$, where x represents the data to be processed, and $(\bar{x}, \bar{y})$ is an example input-output pair that represents the string transformation the user

wishes to perform, so that $\bar{y} = f(\bar{x})$. In the example of the previous section, $(\bar{x}, \bar{y})$ is the pair of strings represented in Figure 1, and x is the list of *all* Oscar winners. Typically, but not necessarily, $\bar{x}$ is some prefix of x.[2] Our goal is to recover $f(\cdot)$ based on $(\bar{x}, \bar{y})$.

Our goal is complicated by the fact that while the user has one particular $f \in \mathcal{S}^{\mathcal{S}}$ in mind, there may be another function g that also explains $(\bar{x}, \bar{y})$. For example, the constant function $g(\cdot) = \bar{y}$ will always explain the example transformation, but will almost always not be the function the user has in mind. Let $\mathcal{F}(z) \subseteq \mathcal{S}^{\mathcal{S}}$ be the set of *consistent* functions for a given system input, so that for each $f \in \mathcal{F}(z)$, $\bar{y} = f(\bar{x})$. We would like to find a way of ranking the elements in $\mathcal{F}(z)$ based on some notion of "plausibility". For example, in Figure 1, one consistent function may involve removing the 4th line of the input. But this is intuitively a less likely explanation than removing duplicates.

We look to *learn* such a ranking based on *training* data. We do so by defining a probability model $\Pr[f|z;\theta]$ over programs, parameterized by some θ. Given θ, at inference time on input z, we pick the most likely program under $\Pr[f|z;\theta]$ which is also consistent with z. We do so by invoking a *search function* $\sigma_{\theta,\tau} : \mathcal{S}^3 \to \mathcal{S}^{\mathcal{S}}$ that depends on θ and an upper bound τ on search time. This produces our conjectured program $\hat{f} = \sigma_{\theta,\tau}(z)$ computing a string-to-string transformation, or a trivial failure function $\perp$ if the search fails in the allotted time.

The θ parameters are *learned* at training time, where the system is given a corpus of T training quadruples, $\{(z^{(t)}, y^{(t)})\}_{t=1}^{T}$, with $z^{(t)} = (x^{(t)}, \bar{x}^{(t)}, \bar{y}^{(t)}) \in \mathcal{S}^3$ representing the actual data and the example input-output pair, and $y^{(t)} \in \mathcal{S}$ the correct output on $x^{(t)}$. We also assume each training example $z^{(t)}$ is annotated with a "correct" program $f^{(t)} \in \mathcal{S}^{\mathcal{S}}$ such that $f^{(t)}(\bar{x}^{(t)}) = \bar{y}^{(t)}$ and $f^{(t)}(x) = y$. (In Section 3.1, we describe a workaround when such annotations are not available.) From these examples, the system chooses the parameters θ that maximize the likelihood $\Pr[f^{(1)}, \ldots, f^{(T)}|z^{(1)}, \ldots, z^{(T)};\theta]$.

Note that each quadruple $(z^{(t)}, y^{(t)})$ represents a different *task*; for example, one may represent the Oscar winners example of the previous section, another an email processing task, and so on. Put another way, for $t_1 \neq t_2$, it is often the case that $f^{(t_1)} \neq f^{(t_2)}$. Ostensibly then, we have a single example from which to estimate the value of $\Pr[f|z]$ for some fixed f! However, we note that while $f^{(t_1)} \neq f^{(t_2)}$, it is often the case that these functions *share* common subroutines. For example, many functions may involve splitting the input based on the "`\n`" character. This tells us that, all else being equal, we should favour splitting on newline characters "`\n`" over splitting on `"z"`, say. Our learning procedure exploits the shared structure amongst each of the T tasks to more reliably estimate $\Pr[f|z;\theta]$.

We now describe how we model the distribution $\Pr[f|z;\theta]$ using a probabilistic context-free grammar.

2.2. PCFGs for programs

We maintain a distribution over programs with a *Probabilistic Context-Free Grammar* (PCFG) $\mathcal{G}$, as discussed in (Liang et al., 2010). The grammar is defined by a set of non-terminal symbols $\mathcal{V}$, terminal symbols Σ (which may include strings $s \in \mathcal{S}$ and also other program-specific objects such as lists or functions), and rules $\mathcal{R}$. Each rule $r \in \mathcal{R}$ has an associated probability $\Pr[r|z;\theta]$ of being generated given the system input z, where θ represents the unobserved parameters of the grammar. WLOG, each rule r is also associated with a function $f_r : \Sigma^{\mathrm{NArgs}(r)} \to \Sigma$, where $\mathrm{NArgs}(r)$ denotes the number of arguments in the RHS of rule r. A program[3] is a derivation of the start symbol $\mathcal{V}_{\mathrm{start}}$. The probability of any program $f(\cdot)$ is the probability of its constituent rules $\mathcal{R}_f$ (counting repetitions):

$$\Pr[f|z;\theta] = \Pr[\mathcal{R}_f|z;\theta] = \prod_{r \in \mathcal{R}_f} \Pr[r|z;\theta]. \quad (2)$$

We now describe how the distribution $\Pr[r|z;\theta]$ is parameterized using clues.

2.3. Features and clues for learning

The learning process exploits the following simple fact: the chance of a rule being part of an explanation for a string pair $(\bar{x}, \bar{y})$ depends greatly on certain characteristics in the structure of $\bar{x}$ and $\bar{y}$. For example, one interesting binary *feature* is whether or not every line of $\bar{y}$ is a substring of $\bar{x}$. If true, it may suggest that the `select_field` rule should receive higher probability in the PCFG, and hence will be combined with other rules more often in the search. Another binary feature indicates whether or not "Massachusetts" occurs repeatedly as a substring in $\bar{y}$ but not in $\bar{x}$. This suggests that a rule generating the string "Massachusetts" may be useful. Conceptually, given a training corpus, we would like to learn the relationship between such

[2] This is more general than the setup of e.g. (Gulwani, 2011), which assumes $\bar{x}$ and $\bar{y}$ have the same number of lines, each of which is treated as a separate example.

[3] Two *programs* from different derivations may compute exactly the same *function* $f : \mathcal{S} \to \mathcal{S}$. However, determining whether two programs compute the same function is undecidable in general. Hence, we abuse notation and consider these to be different functions.

features and the successful rules. However, there are an infinitude of such binary features as well as rules (e.g. a feature and rule corresponding to every possible constant string), but of course limited data and computational resources. So, we need a mechanism to estimate the relationship between the two entities.

We connect features with rules via *clues*. A clue is a function $c : \mathcal{S}^3 \to 2^{\mathcal{R}}$ that states, for each system input z, which subset of rules in $\mathcal{R}$ (the infinite set of grammar rules), may be relevant. This set of rules will be based on certain features of z, meaning that we search over compositions of instance-specific rules[4]. For example, one clue might return $\{\texttt{E} \to \texttt{select\_field}(\texttt{E}, \texttt{Delim}, \texttt{Int})\}$ if each line of $\bar{y}$ is a substring of $\bar{x}$, and $\emptyset$ otherwise. Another clue might recognize the input string is a permutation of the output string, and generate rules $\{\texttt{E} \to \texttt{sort}(\texttt{E}, \texttt{COMP}), \texttt{E} \to \texttt{reverseSort}(\texttt{E}, \texttt{COMP}), \texttt{COMP} \to \texttt{alphaComp}, \ldots\}$, i.e., rules for sorting as well as introducing a nonterminal along with corresponding rules for various comparison functions. A single clue can suggest a multitude of rules for different z's (e.g. $\texttt{E} \to s$ for every substring s in the input), and "common" functions (e.g. concatenation of strings) may be suggested by multiple clues.

We now describe our probability model that is based on the clues formalism.

2.4. Probability model

Suppose the system has n clues $c_1, c_2, \ldots, c_n$. For each clue c_i, we keep an associated parameter $\theta_i \in \mathbb{R}$. Let $\mathcal{R}_z = \cup_{i=1}^{n} c_i(z)$ be the set of *instance-specific* rules (wrt z) in the grammar. While the set of all rules $\mathcal{R}$ will be infinite in general, we assume there are a finite number of clues suggesting a finite number of rules, so that $\mathcal{R}_z$ is finite. For each rule $r \notin \mathcal{R}_z$, we take $\Pr[r|z] = 0$, i.e. a rule that is not suggested by any clue is disregarded. For each rule $r \in \mathcal{R}_z$, we model

$$\Pr[r \mid z; \theta] = \frac{1}{Z_{\text{LHS}(r)}} \exp\left(\sum_{i: r \in c_i(z)} \theta_i\right). \qquad (3)$$

where $\text{LHS}(r) \in \mathcal{V}$ denotes the nonterminal appearing appearing on the left hand side of rule r, and for each nonterminal $V \in \mathcal{V}$, the normalizer Z_V is

$$Z_V = \sum_{r \in \mathcal{R}_z : \text{LHS}(r) = V} \exp\left(\sum_{i: r \in c_i(z)} \theta_i\right).$$

[4] As long as the functions generated by our clues library include a Turing-complete subset, the class of functions being searched amongst is always the Turing-computable functions, though having a good bias is probably more useful than being Turing complete.

This is a log-linear model for the probabilities, where each clue has a *weight* e^{θ_i}, which is intuitively its *reliability*[5], and the probability of each rule is proportional to the *product* of the weights generating that rule. An alternative model would be to make the probabilities be the (normalized) *sums* of corresponding weights, but we favor products for two reasons. First, as described shortly, maximizing the log-likelihood is a convex optimization problem in θ for products, but not for sums. Second, this formalism allows clues to have positive, neutral, or even *negative* influence on the likelihood of a rule, based upon the sign of θ_i.

Our framework overcomes difficulties faced by classical approaches for this problem. Consider for example K-class logistic regression, which for input $x \in \mathbb{R}^d$ and label $y \in \{0,1\}^K$ such that $1^T y = 1$ models

$$\Pr[y \mid x; w] = \frac{1}{Z_x} \exp\left(w^T \phi(x, y)\right)$$

for $\phi(x, y) = x \otimes y$. In our problem, assuming $\mathcal{R}$ is countable we can represent r by the bitvector $e_r \in \{0,1\}^{|\mathcal{R}|}$, and z by a bitvector $f_z \in \{0,1\}^F$ indicating which of F *features* it possesses (e.g. whether it has numbers, or not). Using the representation $\phi(r, z) = e_r \otimes f_z$ would require the estimation of $|\mathcal{R}| \cdot F$ parameters, i.e. a parameter measuring the relationship between every pair of rule and feature. This means that we would for example end up learning separate parameters for all constant strings that occur in a training set. This is not scalable, and also does not exploit the fact that all else being equal, we expect the probabilities for two constant strings to be similar. By contrast, the model of Equation 3 is

$$\Pr[r \mid z; \theta] = \frac{1}{Z_{\text{LHS}(r)}} \exp\left(\theta^T \phi(z, r)\right),$$

where $\phi(r, z) \in \mathbb{R}^n$ is such that $\phi(z, r)_i = \mathbf{1}[r \in c_i(z)]$. If $\mathcal{R}$ is enumerable, this may further be thought of as $\phi(z, r) = C(z) \cdot e_r$, where $C(z) \in \{0,1\}^{n \times |\mathcal{R}|}$ is a binary matrix whose (i, r)th element indicates whether the clue i suggests rule r on input z. The key is that $C(z)$ is sparse, as each clue suggests only a small subset of all rules. This injects prior knowledge to the problem: a human looking at the example of Figure 1 would not think of performing date operations as part of the transformation, and so we encode this fact in our clues. Further, we can keep a single (data-dependent) clue that suggests every constant string that appears in the input or output, which encodes our belief that the probabilities for these rules are tied together.

[5] With these weights, if a clue is too lax in how it suggests rules – for example, suggesting a `date_to_string` operation every time there is a number in the input – its suggestions will be discounted.

3. System training and usage

We are now ready to describe in full the operation of the training and inference phases.

3.1. Training phase: learning θ

At training time, we wish to learn the parameter θ that characterizes the conditional probability of a program given the input, $\Pr[f|z;\theta]$. Recall that we assume each training example $z^{(t)}$ is also annotated with the "correct" program $f^{(t)}$ that explains both the example and actual data pairs. In this case, we choose θ so as to minimize the negative log-likelihood of the data, plus a regularization term:

$$\theta = \operatorname*{argmin}_{\theta' \in \mathbb{R}^n} - \log \Pr[f^{(t)}|z^{(t)};\theta'] + \lambda\Omega(\theta'),$$

where $\Pr[f^{(t)}|z^{(t)};\theta]$ is defined by equations (2) and (3), the regularizer $\Omega(\theta)$ is the ℓ_2 norm $\frac{1}{2}||\theta||_2^2$, and $\lambda > 0$ is the regularization strength which may be chosen by cross-validation. If $f^{(t)}$ consists of rules $r_1^{(t)}, r_2^{(t)}, \ldots, r_{k^{(t)}}^{(t)}$ (possibly with repetition), then

$$\log \Pr[f^{(t)}|z^{(t)};\theta] = \sum_{k=1}^{k=k^{(t)}} \log\left(Z_{\mathrm{LHS}(r_k^{(t)})}\right) - \sum_{i:r_k^{(t)} \in c_i(z^{(t)})} \theta_i$$

The convexity of the objective follows from the convexity of the regularizer and the log-sum-exp function. The parameters θ are optimized by gradient descent.

The assumption that every training example has the annotated "correct" program may be unrealistic, as annotation requires human effort. However, we may attempt to discover the correct annotations automatically by bootstrapping: the reason is that the transformation by definition must explain both $(x^{(t)}, y^{(t)})$ and $(\bar{x}^{(t)}, \bar{y}^{(t)})$. We start with a uniform parameter estimate $\theta^{(0)} = 0$. In iteration $j = 1, 2, 3, \ldots$, we select $f^{(j,t)}$ to be the most likely program, based on $\theta^{(j-1)}$, consistent with the system data. (If no program is found within the timeout, the example is ignored.) Then, parameters $\theta^{(j)}$ are learned, as described above. This is run until convergence.

3.2. Inference phase: evaluating on new input

At inference time, we are given system input $z = (x, \bar{x}, \bar{y})$, n clues $c_1, c_2, \ldots, c_n$, and parameters $\theta \in \mathbb{R}^n$ learned from the training phase. We are also given a timeout τ. The goal is to infer the most likely program $\hat{f}$ that explains the data under a certain PCFG. This is done as follows:

(i) We evaluate each clue[6] on the system input z. The underlying PCFG $\mathcal{G}_z$ consists of the union of all suggested rules, $\mathcal{R}_z = \bigcup_{i=1}^n c_i(z)$.

(ii) Probabilities are assigned to these rules via Equation 3, using the learned parameters θ.

(iii) We enumerate over $\mathcal{G}_z$ in order of decreasing probability, and return the first discovered $\hat{f}$ that explains the $(\bar{x}, \bar{y})$ string transformation, or $\perp$ if we exceed the timeout.

To find the most likely consistent program, we enumerate all programs of probability at least $\eta > 0$, for any given η. We begin with a large η, gradually decreasing it and testing all programs until we find one which outputs $\bar{y}$ on $\bar{x}$ (or we exceed the timeout τ). (If more than one consistent program is found, we just select the most likely one.) Due to the exponentially increasing nature of the number of programs, this approach imposes a negligible overhead due to redundancy – the vast majority of programs are executed just once.

To compute all programs of probability at least η, a dynamic program first computes the maximal probability of a full trace from each nonterminal. Given these values, it is simple to compute the maximal probability completion of any partial trace. We then iterate over each nonterminal expansion, checking whether applying it can lead to any programs above the threshold; if so, we recurse.

4. Results on prototype system

To test the efficacy of our framework, we report results on a prototype web app implemented using client-side JavaScript and executed in a web browser on an Intel Core i7 920 processor. Our aim is to test whether learning weights using textual features – which has not been studied in any prior system, to our knowledge – can speed up inference. It is *not* to claim that our prototype is "better" than existing systems in terms of functionality or richness. Nonetheless, we attempt to construct a reasonably functional system so that our results can be indicative of what we might expect to see in a real-world text processing system.

4.1. Details of base functions and clues

As discussed in Section 2.2, we associated the rules in our PCFG with a set of base functions. In total we created around 100 functions, such as `dedup`, `concatLists`, and `count`, as described in Section 1.1. For clues to connect these functions to features of the examples, we had one set of base clues that suggested functions we believed to be common, regardless of the system in-

[6] Since a clue c is just a function that outputs a list of rules, evaluating $c(z)$ amounts to a single function call.

Table 2. Examples of clues and examples used in our prototype.

(a) Sample of clues used. `LIST` denotes a list-, `E` a string-nonterminal in the grammar.

Feature	Suggested rule(s)
Substring s appears in output but not input?	`E` → "`s`", `LIST` → {`E`}
Duplicates in input but not output?	`LIST` → `dedup`(`LIST`)
Numbers on each input line but not output line?	`LIST` → `count`(`LIST`)

(b) Sample of test-cases used to evaluate the system.

Input	Output
`Adam Ant\n1 Ray Rd.\nMA\n90113`	`90113`
`28/6/2010`	`June the 28th 2010`
`612 Australia`	`case 612: return Australia;`

put z (e.g. string concatenation). Other clues were designed to support common formats that we expected, such as dates, tabular and delimited data. Table 2(a) gives a sample of some of the clues in our system, in the form of grammar rules that certain textual features are connected to; in total we had approximately 100 clues. The full list of functions and clues is available at `http://cseweb.ucsd.edu/~akmenon/pbe`.

4.2. Training set for learning

To evaluate the system, we compiled a set of 280 examples with both an example pair $(\bar{x}, \bar{y})$ and evaluation pair (x, y) specified. These examples were partly hand-crafted, based on various common usage scenarios the authors have encountered, and partly based on examples used in (Gulwani, 2011). The latter examples were derived from manual crawling of Microsoft Excel help forums, and distilling common text processing questions that arise on these forums. Table 2(b) gives a sample of some of the scenarios we tested the system on. To encourage future research on this problem, our suite of training examples is available at `http://cseweb.ucsd.edu/~akmenon/pbe/`.

All examples are expressible as (possibly deep) compositions of our base functions; the median depth of composition on most examples is 4. Like any classical learning model, we assume these are iid samples from the distribution of interest, namely over "natural' text processing examples. It is hard to justify this independence assumption in our case, but we are not aware of a good solution to this problem in general; even examples collected from a user study, say, will tend to be biased in some way.

4.3. Does learning help?

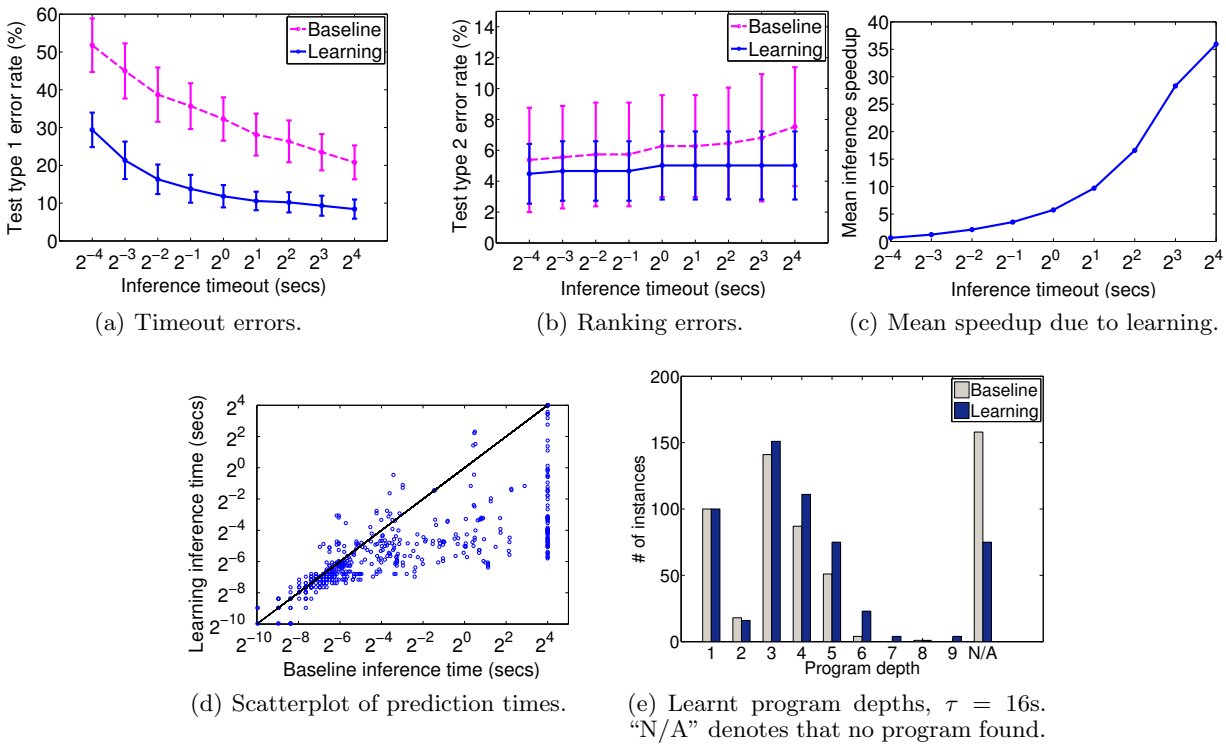
The learning procedure aims to allow us to find the correct program in the shortest amount of time. We compare this method to a baseline, hoping to see quantifiable improvements in performance.

Baseline. Our baseline is to search through the grammar in order of increasing program size, attempting to find the shortest grammar derivation that explains the transformation. The grammar does use clues to winnow down the set of relevant rules, but does not use learned weights: we let $\theta_i = 0$ for all i, i.e. all rules that are suggested by a clue have the same constant probability. This method's performance lets us measure the impact of learning. Note that pure brute force search would not even use clues to narrow down the set of feasible grammar rules, and so would perform strictly worse. Such a method is infeasible for the tasks we consider, because some of them involve e.g. constant strings, which cannot be enumerated.

Measuring performance. To assess a method, we look at its *accuracy*, as measured by the fraction of correctly discovered programs, and *efficiency*, as measured by the time required for inference. As every target program in the training set is expressible as a composition of our base functions, there are two ways in which we might fail to infer the correct program: (a) the program is not discoverable within the timeout set for the search, or (b) another program (one which also explains the example transformation) is wrongly given a higher probability. We call errors of type (a) *timeout errors*, and errors of type (b) *ranking errors*. Larger timeouts lead to fewer timeout errors.

Evaluation scheme. To ensure that the system is capable of making useful predictions on new data, we report the test error after creating 10 random 80–20 splits of the training set. For each split, we compare the various methods as the inference timeout τ varies from $\{1/16, 1/8, \ldots, 16\}$ seconds. For the learning method, we performed 3 bootstrap iterations (see Section 3.1) with a timeout of 8 seconds to get annotations for each training example.

Results. Figures 2(a) and 2(b) plot the timeout and ranking error rates respectively. As expected, for both methods, most errors arise due to timeout when the τ is small. To achieve the same timeout error rate, learning saves about two orders of magnitude in τ compared to the baseline. Learning also achieves lower mean



(a) Timeout errors. (b) Ranking errors. (c) Mean speedup due to learning.

(d) Scatterplot of prediction times.

(e) Learnt program depths, $\tau = 16$s. "N/A" denotes that no program found.

Figure 2. Comparison of baseline versus learning approach.

ranking error, but this difference is not as pronounced as for timeout errors. This is not surprising, because the baseline generally finds few candidates in the first place (recall that the ranking error is only measured on examples that do not timeout); by contrast, the learning method opens the space of plausible candidates, but introduces a risk of some of them being incorrect.

Figure 2(c) shows the relative speedup due to learning as τ varies. We see that learning manages to cut down the prediction time by a factor of almost 40 over the baseline with $\tau = 16$ seconds. (The system would be even faster if implemented in a low-level programming language such as C instead of Javascript.) The trend of the curve suggests there are examples that the baseline is unable to discover with 16 seconds, but learning discovers with far fewer. Figure 2(d) is a scatterplot of the times taken for both methods with $\tau = 16$ over all 10 train-test splits, confirms this: in the majority of cases, learning finds a solution in much less time than the baseline, and solves many examples the baseline fails on within a fraction of a second. (In some cases, learning slightly increases inference time. Here, the test example involves functions insufficiently represented in the training set.)

Finally, Figure 2(e) compares the depths of programs (i.e. number of constituent grammar rules) discovered by learning and the baseline over all 10 train-test splits, with an inference timeout of $\tau = 16$ seconds. As expected, the learning procedure discovers many more programs that involve deep (depth ≥ 4) compositions of rules, since the rules that are relevant are given higher probability.

5. Conclusion and future work

We propose a PBE system for repetitive text processing based on exploiting certain clues in the input data. We show how one can learn the utility of clues, which relate textual features to rules in a context free grammar. This allows us to speed up the search process, and obtain a meaningful ranking over programs. Experiments on a prototype system show that learning with clues brings significant savings over naïve brute force search. As future work, it would be interesting to learn correlations between rules and clues that did *not* suggest them, although this would necessitate enforcing some strong parameter sparsity. It would also be interesting to incorporate ideas like adaptor grammars (Johnson et al., 2006) and learning program structure as in (Liang et al., 2010). Finally, (Gulwani, 2012; Gulwani et al., 2012) describe several application domains for PBE where it would be interesting to consider applying the techniques proposed in this paper.

References

Cypher, Allen, Halbert, Daniel C., Kurlander, David, Lieberman, Henry, Maulsby, David, Myers, Brad A., and Turransky, Alan (eds.). *Watch what I do: programming by demonstration*. MIT Press, Cambridge, MA, USA, 1993. ISBN 0262032139.

Gulwani, Sumit. Automating string processing in spreadsheets using input-output examples. In *POPL*, pp. 317–330, 2011.

Gulwani, Sumit. Synthesis from examples: Interaction models and algorithms. *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2012.

Gulwani, Sumit, Korthikanti, Vijay Anand, and Tiwari, Ashish. Synthesizing geometry constructions. In *PLDI*, pp. 50–61, 2011.

Gulwani, Sumit, Harris, William R., and Singh, Rishabh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, 2012.

Jha, Susmit, Gulwani, Sumit, Seshia, Sanjit A., and Tiwari, Ashish. Oracle-guided component-based program synthesis. In *ICSE*, pp. 215–224, 2010.

Johnson, Mark, Griffiths, Thomas L., and Goldwater, Sharon. Adaptor grammars: A framework for specifying compositional nonparametric bayesian models. In *NIPS*, pp. 641–648, 2006.

Lau, Tessa, Wolfman, Steven A., Domingos, Pedro, and Weld, Daniel S. Programming by demonstration using version space algebra. *Mach. Learn.*, 53:111–156, October 2003. ISSN 0885-6125.

Lau, Tessa A., Domingos, Pedro, and Weld, Daniel S. Version space algebra and its application to programming by demonstration. In *ICML*, pp. 527–534, 2000.

Liang, Percy, Jordan, Michael I., and Klein, Dan. Learning programs: A hierarchical Bayesian approach. In *ICML*, pp. 639–646, 2010.

Lieberman, H. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, 2001.

Miller, Robert C. *Lightweight Structure in Text*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, May 2002. URL http://www.cs.cmu.edu/~rcm/papers/thesis/.

Nix, Robert P. Editing by example. *TOPLAS*, 7(4): 600–621, 1985.

Rush, Alexander M., Collins, Michael, and Kaelbling, Pack. A tutorial on dual decomposition and Lagrangian relaxation for inference in natural language processing. http://www.cs.columbia.edu/~mcollins/acltutorial.pdf, 2011.

Witten, Ian H. and Mo, Dan. *TELS: learning text editing tasks from examples*, pp. 183–203. MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-03213-9.