Textual Features for Programming by Example

Aditya Krishna Menon
University of California, San Diego,
9500 Gilman Drive, La Jolla CA 92093
akmenon@ucsd.edu

Omer Tamuz
Faculty of Mathematics and Computer Science,
The Weizmann Institute of Science, Rehovot Israel
omer.tamuz@weizmann.ac.il

Sumit Gulwani
Microsoft Research,
One Microsoft Way, Redmond, WA 98052
sumitg@microsoft.com

Butler Lampson
Microsoft Research,
One Memorial Drive, Cambridge MA 02142
butler.lampson@microsoft.com

Adam Tauman Kalai
Microsoft Research,
One Memorial Drive, Cambridge MA 02142
adum@microsoft.com

September 19, 2012
To appear in ICML '13

Abstract

In Programming by Example, a system attempts to infer a program from input and output examples, generally by searching for a composition of certain base functions. Performing a naïve brute force search is infeasible for even mildly involved tasks. We note that the examples themselves often present *clues* as to which functions to compose, and how to rank the resulting programs. In text processing, which is our domain of interest, clues arise from simple *textual features*: for example, if parts of the input and output strings are permutations of one another, this suggests that sorting may be useful. We describe a system that *learns* the reliability of such clues, allowing for faster search and a principled ranking over programs. Experiments on a prototype of this system show that this learning scheme facilitates efficient inference on a range of text processing tasks.

1 Introduction

Programming by Example (PBE) [10, 1] is an attractive means for end-user programming tasks, wherein the user provides the machine *examples* of a task she wishes to perform, and the machine infers a program to accomplish this. This paradigm has been used in a wide variety of domains; [4] gives a recent overview. We focus on *text processing*, a problem most computer users face (be it reformatting the contents of an email or extracting data from a log file), and for which several complete PBE systems have been designed, including LAPIS [11], SMARTedit [8], QuickCode [3, 13], and others [12, 14]. Such systems aim to provide a simpler alternative to the traditional solutions to the problem, which involve either tedious manual editing, or esoteric computing skills such as knowledge of `awk` or `emacs`.

A fundamental challenge in PBE is the following *inference* problem: given a set of base functions, how does one quickly search for programs composed of these functions that are consistent with the user-provided examples? One way is to make specific assumptions about the nature of the base functions, as is done by many existing PBE systems [11, 8, 3], but this is unsatisfying because it restricts the range of tasks a user can perform. The natural alternative, brute force search, is infeasible for even mildly involved programs [2]. Thus, a basic question is whether there is a solution possessing both generality and efficiency.

This paper aims to take a step towards an affirmative answer to this question. We observe that there are often telling *features* in the user examples suggesting which functions are likely. For example, suppose that a user demonstrates their intended task through one or more input-output pairs of strings $\{(x_i, y_i)\}$, where each y_i is a permutation of x_i. This feature provides a *clue* that when the system is searching for the $f(\cdot)$ such that $f(x_i) = y_i$, sorting functions may be useful. Our strategy is to incorporate a library of such clues, each suggesting relevant functions based on textual features of the input-output pairs. We *learn* weights telling us the reliability of each clue, and use this to bias program search. This bias allows for significantly faster inference compared to brute force search. Experiments on a prototype system demonstrate the effectiveness of feature-based learning.

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.

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 corre-

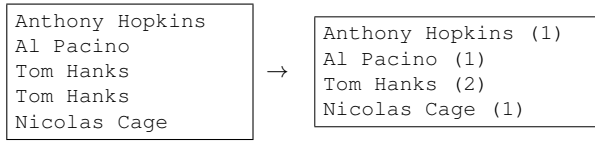


Figure 1: Input-output example for the desired task.

sponding program $f(\cdot)$ may be expressed as the composition

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

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 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 [3] 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.) Systems with richer functionality are inapplicable because they perform inference with brute force search (or a similarly intractable operation [5]) over all possible function compositions. Such a naïve search over even a moderate sized library of base functions is unlikely to find the complex composition of our example. Therefore, a more generic framework is needed.

Our basic observation is 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 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. 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

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

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,")")`.

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

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

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.

1.2 Contributions

To the best of our knowledge, ours is the first PBE system to exploit textual features for inference, which we believe is a step towards achieving the desiderata of efficiency and generality. The former will be demonstrated in an empirical evaluation of our learning scheme. For the latter, while the learning component is discussed in the context of text processing, the approach could possibly be adapted for different domains. Further, the resulting system is highly extensible. Through the use of clues, one only considers broadly relevant functions during the search for a suitable program: one is free to add functionality to process addresses, e.g. , without fear of it adversely affecting the performance of processing dates. Through the use of learning, we further sift amongst these broadly relevant functions, and determine which of them is likely to be useful in explaining the given data. A system designer need only write clues for any new functionality, and add relevant examples to the training corpus. Our system then automatically learns weights associated with these clues.

1.3 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 [7], 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, or a similarly intractable operation such

as invoking a SAT solver [5].[2] Our learning approach based on textual features is thus more general and flexible than previous approaches.

Having said this, 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 QuickCode [3] 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. At *inference 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 the example input-output pair that represents the string transformation the user wishes to perform. 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. While a typical choice for $\bar{x}$ is some prefix of x, this is not required in general[3]. We assume that $\bar{y} = f(\bar{x})$, for some unknown *target function* or *program* $f \in \mathcal{S}^{\mathcal{S}}$, from the set of functions that map strings to strings. Our goal is to recover $f(\cdot)$.

We do so by defining a probability model $\Pr[f|z;\theta]$ over programs, parameterized by some θ. Given some 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 $x^{(t)}$. Note that each quadruple here represents a different *task*; for example, one may represent the Oscar winners example of the previous section, another a generic email processing task, and so on. From these examples, the system chooses the parameters θ that maximize the likelihood $\Pr[f|z;\theta]$. We now describe how we model the conditional distribution $\Pr[f|z;\theta]$ using a probabilistic context-free grammar.

[2] One could consider employing heuristic search techniques such as Genetic Programming. However, this requires picking a metric that captures meaningful search progress. This is difficult, since functions like sorting cause drastic changes on an input. Thus, standard metrics like edit distance may not be appropriate.

[3] This is more general than the setup of e.g. [3], which assumes $\bar{x}$ and $\bar{y}$ have the same number of lines, each of which is treated as a separate example.

2.2 PCFGs for programs

We maintain a probability distribution over programs with a *Probabilistic Context-Free Grammar* (PCFG) $\mathcal{G}$, as discussed in [9]. 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 $f_r : \Sigma^{\text{NArgs}(r)} \to \Sigma$, where $\text{NArgs}(r)$ denotes the number of arguments in the RHS of rule r. A program[4] is a derivation of the start symbol $\mathcal{V}_{\text{start}}$. The probability of any program $f(\cdot)$ is the probability of its constituent rules $\mathcal{R}_f$ (counting repetitions):

$$\Pr[f|z;\theta] = \Pr[\mathcal{R}_f|z;\theta] = \prod_{r \in \mathcal{R}_f} \Pr[r|z;\theta]. \tag{1}$$

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 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[5]. 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 $\{\texttt{E} \to \texttt{sort}(\texttt{E}, \texttt{COMP}), \texttt{E} \to$

[4]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.

[5]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.

$\texttt{reverseSort}(\texttt{E}, \texttt{COMP}), \texttt{COMP} \to \texttt{alphaComp}, \ldots\}$, i.e., rules for sorting as well as introducing a nonterminal along with corresponding rules for various comparison functions. Note that 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 use the probability model

$$\Pr[r \mid z; \theta] = \frac{1}{Z_{\mathrm{LHS}(r)}} \exp\left(\sum_{i: r \in c_i(z)} \theta_i\right). \tag{2}$$

where for each nonterminal V, the normalizer Z_V ensures we get a valid probability distribution:

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

This is a log-linear model for the probabilities, where each clue has a *weight* e^{θ_i}, which is intuitively its *reliability*, and the probability of each rule is proportional to the *product* of the weights generating that rule. An alternative 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.

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, $\Pr[f|z; \theta]$. 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. We may attempt to discover these annotations automatically by bootstrapping: we start with a uniform parameter estimate $\theta^{(j)} = 0$. 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 below. This is run until convergence.

Fix a single iteration j. For notational convenience, we write target programs $f^{(t)} = f^{(j,t)}$ and parameters $\theta = \theta^{(j)}$. We choose θ so as to minimize the negative log-likelihood of the data, plus a regularization term:

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

where $\Pr[f^{(t)}|z^{(t)}; \theta]$ is defined by equations (1) and (2), 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

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

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.

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 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 2, 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 decreasing threshold 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 proposed system, 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 goal with the experiments is *not* to claim that our prototype system is "better" than existing systems in terms of functionality or richness. (Even if we wished to compare functionality, this would be difficult since all existing text processing systems that we are aware of are proprietary.) Instead, our aim is to evaluate whether learning weights using textual features – which has not been studied in any prior system, to our knowledge – can speed up inference. Nonetheless, we do 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 input 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 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 as part of our supplementary material.

Table 2: Sample of clues used. `LIST` denotes a list-, `E` a string-nonterminal.

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

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 [3]. All examples are expressible as (possibly deep) compositions of our base functions; the median depth of composition on most examples is around 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. Table 3 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 ready for public release, and is available as part of our supplementary material.

Table 3: 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;`

4.3 Does learning help?

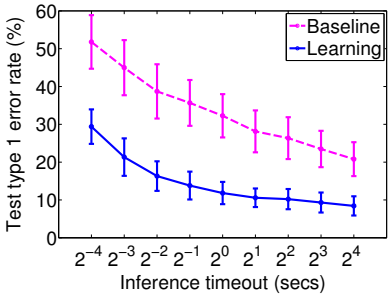
The learning procedure aims to allow us to find the correct program in the shortest amount of time. We compare this method to a baseline, hoping to see quantifiable improvements in performance.

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.

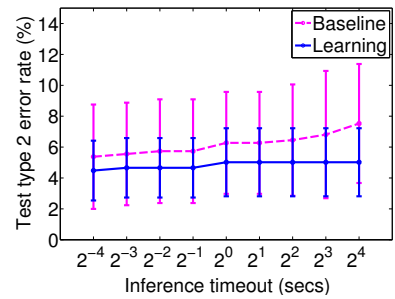
Measuring performance. To assess a method, we look at its *accuracy*, as measured by the fraction of correctly discovered programs, and *efficiency*, as measured by the time required for inference. As every target program in the training set is expressible as a composition of our base functions, there 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. One possible pitfall in an empirical evaluation is having an overly specific set of clues for the training set: an extreme case would be a single clue for each training example, which automatically suggested the correct rules to compose. 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.

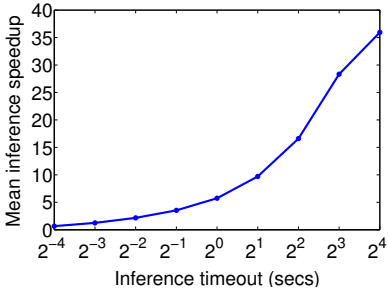
Results. Figures 2(a) and 2(b) plot the timeout and ranking error rates respectively. As expected, for both methods, most errors arise due to timeout when the τ is small. To achieve the same timeout error rate, learning saves about two orders of magnitude in τ compared to the baseline. Learning also achieves lower mean ranking error, but this difference is not as pronounced as for timeout errors. This is not surprising, because the



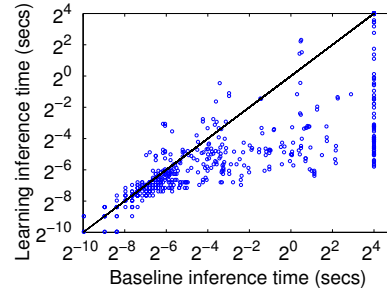
(a) Timeout errors.



(b) Ranking errors.



(c) Mean speedup due to learning.



(d) Scatterplot of prediction times.

Figure 2: Comparison of baseline versus learning approach.

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 3 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.

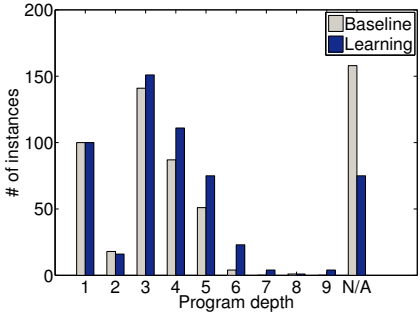


Figure 3: Learnt program depths, $\tau = 16$s. "N/A" denotes that no successful program is found.

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 [6] and learning program structure as in [9].

References

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

[2] Sumit Gulwani. Dimensions in program synthesis. In *PPDP*, 2010.

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

[4] Sumit Gulwani. Synthesis from examples. *WAMBSE (Workshop on Advances in Model-Based Software Engineering) Special Issue, Infosys Labs Briefings*, 10(2), 2012.

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

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

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

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

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

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

[11] Robert C. Miller and Brad A. Myers. Lightweight structured text processing. In *USENIX Annual Technical Conference, General Track*, pages 131–144, 1999.

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

[13] Rishabh Singh and Sumit Gulwani. Learning semantic string transformations from examples. *PVLDB*, 5(8), 2012.

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