Learning Natural Programs from a Few Examples in Real-Time

Nagarajan Natarajan[1] **Danny Simmons**[2] **Naren Datha**[1] **Prateek Jain**[1] **Sumit Gulwani**[2]
[1]Microsoft Research, India [2]Microsoft Corporation, Redmond

Abstract

Programming by examples (PBE) is a rapidly growing subfield of AI, that aims to synthesize user-intended programs using input-output examples from the task. As users can provide only a few I/O examples, capturing user-intent accurately and ranking user-intended programs over other programs is challenging even in the simplest of the domains. Commercially deployed PBE systems often require years of engineering effort and domain expertise to devise ranking heuristics for real-time synthesis of accurate programs. But such heuristics may not cater to new domains, or even to a different segment of users from the same domain. In this work, we develop a novel, real-time, ML-based program ranking algorithm that enables synthesis of natural, user-intended, personalized programs. We make two key technical contributions: 1) a new technique to embed programs in a vector space making them amenable to ML-formulations, 2) a novel formulation that interleaves program search with ranking, enabling real-time synthesis of accurate user-intended programs. We implement our solution in the state-of-the-art PROSE framework. The proposed approach learns the intended program with just *one* I/O example in a variety of real-world string/date/number manipulation tasks, and outperforms state-of-the-art neural synthesis methods along multiple metrics.

1 Introduction

Programming by examples (PBE) is an important and emerging subfield of AI (Parisotto et al., 2016; Balog et al., 2017; Devlin et al., 2017; Bunel et al., 2018; Kalyan et al., 2018), where a user-intended program is synthesized automatically with the help of a few input-output examples (I/O specification, or spec for short). A large fraction of computer users are not experts in programming, and synthesizing programs automatically enables them to be more productive. Table 1 describes a typical PBE task.

Input	Output
Missing page numbers, 1993	1993
64-67, 1995	?
2002 (1-27)	?

Table 1: I/O spec provided to a PBE system. Goal is to find a program that is: a) *consistent* (maps the first input into the corresponding output), b) *generalizable* or accurate (computes desired output on last two inputs). While millions of programs in the DSL in Figure 1 are *consistent*, only a handful of them *generalize* well to *unseen* inputs.

PBE is essentially a needle-in-haystack problem where the goal is to search for a *consistent* program (i.e. one that satisfies given I/O spec) in a certain *Domain Specific Language* (DSL) that might contain infinitely many programs. The problem becomes significantly more difficult due to user-centric focus of the systems — the PBE system has to be real-time and should be able to synthesize non-trivial programs; and often in under-specified situations as one cannot expect a user to provide a large number of I/O examples. Unfortunately, these requirements are somewhat contradictory. That is, if the DSL is rich and can support complicated programs, then a small number of I/O examples might not be able to uniquely identify a program in the DSL. For example, for the specification in Table 1, we can generate several consistent programs in the DSL of Figure 1, such as “extract the first number” or “extract the last token”. However, human programmers are typically able to figure out the correct program using a few I/O examples. So, the key question is: *can we synthesize rich user-intended programs using a small number of I/O examples, in real-time?*

Starting with the FlashFill PBE system (Gulwani, 2011) that was commercially deployed in MS Excel (PCWorld, 2012), there has been tremendous progress in this domain over the past few years. Typical PBE techniques search for a program in a carefully-designed DSL and can be categorized into: a) symbolic deduction based techniques (Polo-

Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics (AISTATS) 2019, Naha, Okinawa, Japan. PMLR: Volume 89. Copyright 2019 by the author(s).

zov and Gulwani, 2015; Gulwani et al., 2017; Alur et al., 2017; Le et al., 2017), b) neural computation based techniques (Parisotto et al., 2016; Balog et al., 2017; Devlin et al., 2017; Bunel et al., 2018; Kalyan et al., 2018).

Most neural synthesis (Parisotto et al., 2016; Balog et al., 2017; Devlin et al., 2017; Bunel et al., 2018) models are trained on synthetic data and hence in general, do not capture user-intended programs with a small number of I/O examples. In contrast, symbolic computation based PBE systems handcode the structure of programs and domain knowledge tightly leading to significantly more accurate programs in certain cases. However, manual engineering of the system makes it challenging to extend the solution for even slightly different scenario or a new domain.

Our work alleviates concerns with both the approaches by carefully combining ML techniques with the symbolic search techniques well-understood by the PL community.
(**1**) Our first contribution addresses the fundamental question of embedding heterogeneous programs/expressions in a vector-space which can make programs more amenable to standard learning techniques (Section 3). In the context of PBE, a few key learning tasks that are enabled by program embeddings are: clustering programs/expressions (Padhi et al., 2017), predicting correct programs (Singh and Gulwani, 2015; Ellis and Gulwani, 2017), and ranking programs (Polozov and Gulwani, 2015; Balog et al., 2017). Defining program embedding is challenging because programs are inherently *recursive* and can be composed of heterogeneous sub-expressions. Furthermore, semantically different programs can often behave equivalently on a given I/O spec, so the embedding should take I/O spec into account as well.
(**2**) We show how the proposed embedding can be leveraged for learning to rank programs, a crucial component of PBE systems. However, we cannot apply standard ranking techniques — we cannot even enumerate all the candidate programs to rank as there can be millions of consistent programs. So, we need to interleave synthesis and ranking for real-time synthesis, which in turn requires comparing heterogeneous programs, subprograms, expressions, etc. The problem is further complicated by unavailability of supervision for such intermediate subprograms, and by biased training data that the bootstrapping process induces. We propose three novel and successively refined formulations to address the above mentioned challenges (Section 4).
(**3**) Finally, we integrate our ranking solution with the state-of-the-art PROgram Synthesis using Examples, PROSE (2015) framework. In particular, we show that on real-world data wrangling tasks, the proposed ranking approach outperforms baselines, as well as state-of-the-art neural-synthesis approaches significantly. Our solution is competitive wrt. the ranker tuned over two expert-years that currently ships in Microsoft products (MS Excel, Powershell, Azure ML). Using just one I/O example, our method synthesizes a desired program for about 67% of the tasks while baselines are successful only in at most 44%.

```
@start program ≔ tr | If(cond) Then(tr)
    Else(program);
bool cond := Matches(input, r);
string tr ≔ atom | Concat(atom, tr);
string atom ≔ ConstStr(s) | let string x
    : input in SubStr(x, pp) | input;
Tuple<int, int> pp ≔ Pair(pos, pos) |
    RegexOccurrence(x, r, k);
int pos ≔ AbsPos(x, k);
@input string input; string s; int k;
    Regex r; //Terminals
```

Figure 1: An illustrative subset of the FlashFill DSL (Gulwani, 2011). A program takes a string $input$, and returns a string, a concatenation of $atoms$. The operators are self-explanatory. See Appendix B for the full DSL.

2 Background

In this section, we define the PBE problem formally, introduce various aspects of PBE systems and terminology/notation used in the rest of the paper.

The goal of a PBE system is to generate *user-intended* program(s) where the user intention is specified using input-output examples (I/O spec): $\zeta = \{\sigma_i \mapsto \psi_i\}_{i=1}^m \cup \{\sigma_i\}_{i=m+1}^n$. σ_i is the i-th example's input and ψ_i is the corresponding output (when available). Unlabeled inputs are often available and can be used for doing simple validation checks on synthesized programs (See Remark 1).

Typically, PBE systems restrict the search for a program to a domain specific language $\mathcal{L}$ that is powerful enough for solving critical tasks in a certain domain, but is still restrictive and structured enough for efficient program synthesis. A DSL $\mathcal{L}$ is a represented as a context-free grammar (CFG) consisting of *terminal* symbols T, *non-terminal* symbols N, *rules* that govern how non-terminals are expanded, and *operators* $F : (N \cup T)^* \to N$ that make the production rules. As an example, consider the popular *FlashFill* DSL meant for data wrangling tasks in spreadsheets (Gulwani, 2011; Polozov and Gulwani, 2015). The core DSL is captured in Table 1 (Appendix B has the full DSL).

A program or an expression $\mathcal{L} \ni P : \sigma \to \psi$ is a structured entity with precise syntax and semantics defined by the DSL.

Remark 1 (Unlabeled inputs)**.** *Using unlabeled inputs (i.e. $\{\sigma_i\}_{i=m+1}^n$) can be often helpful in characterizing program behavior; for example, programs that map many of the unlabeled inputs to* `nulls` *or empty strings can be indicative of unintended behavior.*

For a PBE system to be usable in an interactive setting, it

should satisfy three key requirements:
(**R1**) be *consistent* (see Definition 1), i.e., return program(s) that satisfy the user-provided I/O spec,
(**R2**) be *generalizable*, i.e., the synthesized program(s) should give desired output on *unseen* inputs; for severely underspecified problems (say $m = 1$ I/O example) there can be millions of consistent programs (see Table 1), and
(**R3**) be *real-time*, i.e., the synthesise generalizable programs on consumer-class devices.

Definition 1 (Consistent Program)**.** *A program $P \in \mathcal{L}$ is "consistent" on a given input-output specification $\{\sigma_i, \psi_i\}_{i=1}^m$, if $P(\sigma_i) = \psi_i$, for $i = 1, 2, \ldots, m$. Otherwise, P is inconsistent.*

While consistency (**R1**) is essentially a search problem, (**R2**) is more critical and interesting from a machine learning perspective — often there can be millions of programs that satisfy (**R1**), but the user would find most of the consistent programs unusable because they do not generalize to new inputs. It is not possible to formally specify "naturalness" of programs with symbolic logic. Typically, (**R2**) is addressed by means of a ranking function that can help choose the "best" program from possibly many consistent programs. One way to address this is to first synthesize all the consistent programs, and *then* rank them (Ellis and Gulwani, 2017). Unfortunately, the naïve approach cannot be done in real-time — it can take hours to even enumerate the consistent programs, thus contradicting (**R3**). State-of-the-art neural-network based synthesis approaches are trained on synthetic datasets/programs, so they fail to capture the structure in the domain. As a result, neural synthesis approaches suffer in the quality of synthesized programs, especially for underspecified synthesis tasks (See Section 5).

It is therefore crucial to look at the search and the ranking problem as a whole (i.e., (**R1**)-(**R3**)). Successful, commercially-deployed PBE systems (Gulwani, 2011; Gulwani et al., 2015; Alur et al., 2013) use symbolic logic and deductive synthesis techniques to efficiently address (**R1**) and (**R3**). In particular, the symbolic PBE systems use a top-down deductive synthesis strategy based on the divide-and-conquer paradigm. Here, the search problem for a given I/O spec is reduced into smaller subproblems with suitably modified specs[1]. For e.g., the synthesis problem $\zeta = \{$"New York" $\mapsto$ "NY"$\}$ is broken down into finding a set of subprograms $\mathcal{P}_1$ with spec $\zeta_1 = \{$"New York" $\mapsto$ "N"$\}$ and a set of subprograms $\mathcal{P}_2$ with spec $\zeta_2 = \{$"New York" $\mapsto$ "Y"$\}$, i.e., programs in $\mathcal{P}_1$ generating "N" and those in $\mathcal{P}_2$ generating "Y". Then the final program set is given by $\mathcal{P} = \{\texttt{Concat}(P_1, P_2)\ s.t.\ P_1 \in \mathcal{P}_1, P_2 \in \mathcal{P}_2\}$. Each of the synthesis subproblems is solved recursively using the same strategy.

[1]it is beyond the scope of the paper to describe how spec for the subproblems are obtained. See Polozov and Gulwani (2015) for details of search. The key idea is to leverage *inverse semantics* of the involved operators.

However, the aforementioned PBE systems rely on heuristics for (**R2**), i.e. ranking (Polozov and Gulwani, 2015; Rolim et al., 2017; Wang et al., 2017) (such as choosing smaller programs/expressions over larger ones). Simple heuristics may result in bad failures even in simple cases. For illustration, consider the data formatting task with just one I/O example: $\{$"[CCC-0001" $\mapsto$ "[CCC-0001]"$\}$. Adopting naive heuristics such as "prefer programs with fewer constants" or "prefer shorter programs" leads to the incorrect program: `Concat(`*input*`, ConstStr("]"))`, which would fail on an already formatted input, say "[CCC-002]". On the other hand, developing carefully-tuned ranking heuristics often takes one to two expert-years; and requires continual effort to keep up with domain changes, let alone scaling to new domains. Also, it can be challenging to *personalize* the heuristics to user segments with unique biases/preferences.

The primary goal of our work is to develop an ML-based ranking solution for real-time synthesis of natural programs. Programs are difficult objects to analyse/rank, so we need to be able to embed them in a suitable feature space. To this end, we first address the problem of embedding *heterogeneous* programs/expressions in a common vector space. Defining an embedding that handles the heterogeneity is non-trivial, and it turns out that we need to *learn* the embeddings themselves. Existing embedding techniques (Ellis and Gulwani, 2017) do not work because they are defined for homogeneous programs. We address the embedding challenges and our solution in Section 3. Subsequently, we consider the problem of doing program ranking and search jointly. Apriori, it is unclear how to set up/formulate the machine learning problem, or what loss function to optimize. Ranking programs/expressions is challenging for multiple reasons: 1) classical ranking techniques (Liu et al., 2009) do not work, as we do not even have a clean supervised dataset to begin with, and 2) search for user-intended programs is a sequential decision making problem, therefore a mistake at any point in the search may be irrevocable; this necessitates a novel ranking formulation that admits *interleaved* search and ranking during synthesis. We address these challenges and propose ranking solutions in Section 4.

3 Program-Spec Embedding

Informally, the problem is to find a representation for programs/expressions $P \in \mathcal{L}$ *together* with the I/O spec ζ, such that the embedding captures syntactic and semantic structure (defined by DSL), as well as behavioral properties (defined by I/O spec). Defining a feature vector for programs/expressions that captures the complex structure/properties is not obvious. Simple techniques like using the abstract syntax tree (AST) directly do not suffice. Programs with very similar ASTs can differ arbitrarily in their semantics. Consider two programs from the *FlashFill*

DSL for the task in Table 1, P_1 = `let` x : *input* `in SubStr(`x`, RegexOccurrence(`x`, "Number", 1))` and P_2 = `let` x : *input* `in SubStr(`x`, RegexOccurrence(`x`, "Number", -1))`; P_1 and P_2 have identical ASTs but different semantics (extracting the first number vs the last number in the input). On the other hand, two programs with very different ASTs can produce identical outputs on given inputs.

(1) It is crucial to embed I/O spec along with the program/expression itself. The utility of a program can vary drastically based on the I/O spec. For e.g., the program P = `let` x : *input* `in SubStr(`x`, Pair(1,3))` has the outcome of extracting first three digits of SSN in $\zeta_1 = \{$"123-45-6789" $\mapsto$ "123", "555-21-9012" $\mapsto$ "555"$\}$ vs an undesirable outcome of extracting first three letters of name in $\zeta_2 = \{$"Joe Smith" $\mapsto$ "Joe"$\}$. So the embedding must be defined on the tuple (P, ζ) rather than just P.

(2) The embedding should facilitate comparisons between expressions and programs of different sizes, types and complexities. For e.g., we want the expressions `Concat(Concat(ConstStr("@"), ConstStr("gmail")), ConstStr(".com"))` and `ConstStr("@gmail.com")` to yield similar representations. This is highly non-trivial; existing embedding techniques do not impose/satisfy such a requirement.

(3) Programs are compositional, e.g. `Concat(Concat(`P_1`, `P_2`), `P_3`)`. We want the embedding to be *recursive*, thereby preserving the compositional structure. The embedding of a program should respect and conform to the embeddings of its *subprograms/expressions*.

Often domain knowledge can help us define features for individual operators in the DSL. Concretely, let $\Phi_{Op}(P, \zeta) \in \mathbb{R}^{d_{Op}}$ be the set of *given* d_{Op} features for an operator Op. For e.g., for the Concat operator, the length of its prefix string argument is a feature (note that the feature may depend on the spec ζ). See Appendix B.3 for features in FlashFill DSL.

Define the dimensionality d to be $d = \sum_{Op \in CFG(\mathcal{L})} d_{Op}$.

Definition 2 (Program-Spec embedding)**.** *For any given program/expression $P \in \mathcal{L}$, operator features $\Phi_{Op} \in \mathbb{R}^{d_{Op}}$ for all operators $Op \in CFG(\mathcal{L})$, and I/O spec ζ, we want an embedding $\Phi(P; \zeta) \in \mathbb{R}^d$ that satisfies the aforementioned three requirements.*

To handle the recursive nature of programs (in the requirement (3) above), and the grammar itself, we critically exploit the fact that $\mathcal{L}$ is represented as an unambiguous grammar that has a unique parse $\mathcal{T}(P)$ for P. Let $\mathsf{Op}(P)$ be the operator at the top of $\mathcal{T}(P)$, and let $\mathcal{C}(P)$ denote the immediate children nodes of P in $\mathcal{T}(P)$. We obtain embedding for P by combining the *given* features for the top operator in $\mathcal{T}(P)$ with a weighted combination of embeddings of each child node of P in $\mathcal{T}(P)$. We define embedding

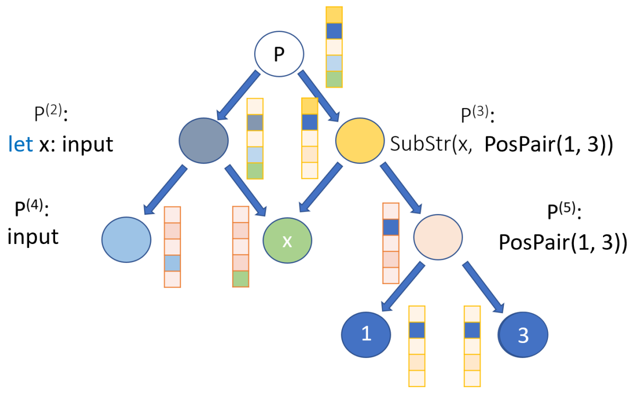


Figure 2: Parse-tree and embedding for the program: `let` x : *input* `in SubStr(`x`, PosPair(1, 3))`. The types of nodes (variables/operators) are color-coded.

$\Phi(P)$ of P *recursively* as:

$$\Phi(P; \zeta) = \Phi_{Op(P)}(P, \zeta; w) + \sum_{P' \in \mathcal{C}(P)} w(P')\Phi(P', \zeta_{P'}; w) \,, \tag{1}$$

where $\Phi_{Op(P)}(P; \zeta)$ are the given features for the root operator $Op(P)$ of P, $\zeta_{P'}$ is the spec for subprogram P' defined as $\{\sigma_i \mapsto P'(\sigma_i) \mid \sigma_i \in \zeta\}$, and $w(P')$ is the weight assigned to the operator at child P', i.e. $w(P') := w(Op(P'))$, in the parse tree of P (see Figure 2). Thus, in addition to the given features, the embeddings are characterized by children operator weights $w(P')$ as well, i.e., $w \in \mathbb{R}^{|\mathcal{L}_{Op}|}$ where $|\mathcal{L}_{Op}|$ is the number of operators in $\mathcal{L}$.

Remark 2. *Note that although the definition of the embedding is recursive, we can compute it once weights w are fixed. Observe that the leaf nodes in $\mathcal{T}(P)$ have only the given features and hence the embeddings are well-defined and immediately obtained; thus, the embedding for the program P can be computed* efficiently *in a bottom-up fashion.*

Thus we have a homogeneous embedding $\Phi(P, \zeta; w)$ of the program P in the same d-dimensional space as that of its constituent expressions. The weights $w(.)$ can be learned based on the end task that the embedding will be used for, addressed in Section 4.

Remark 3. *Our program embedding technique is also an independent technical contribution, as it enables key learning tasks such as clustering programs/expressions (Padhi et al., 2017), predicting correct programs (Singh and Gulwani, 2015; Ellis and Gulwani, 2017), and ranking programs in code-completion task (Balog et al., 2017).*

4 Program Ranking

The goal of program ranking is to learn a ranking function s that provides the highest score to user-intended programs; and to facilitate synthesis of a user-intended program from a few I/O examples. However, as mentioned in Section 2, a standard approach (Ellis and Gulwani, 2017) of generating all the *consistent* programs and then ranking them using standard formulations (Liu et al., 2009) is not feasible for real-time systems.

Instead, a key motivating observation for our solution is that the search process of the synthesis algorithm partitions the program generation into multiple *smaller* program synthesis sub-problems. So, the ranking algorithm should be able to generate "correct" subprograms for each of the smaller sub-problems as well; we call a program *correct* if it produces the desired output on unseen inputs as well.

That is, say a program $P = Op(P_1, \dots, P_r), \{P_j \in \mathcal{P}_j\}$ is generated for specification ζ with operator Op in the DSL $\mathcal{L}$. Each $P_j \in \mathcal{P}_j,\ 1 \le j \le r$, is in turn generated by solving a smaller PBE problem with "refined" specification ζ_j (discussed briefly in Section 2). Now, we require the ranking function s to be such that it not only scores P higher than other programs $P' \in \mathcal{L}$ for specification ζ but it also scores each P_j above other programs $P'_j \in \mathcal{L}$ for specification ζ_j. That is the ranking function is *monotonic*.

Definition 3 (Program ranking)**.** *Let* $\zeta = \{\sigma_i \mapsto \psi_i\}_{i=1}^m \cup \{\sigma_i\}_{i=m+1}^n$ *denote I/O spec given to the PBE system. We want to learn a ranking function* $s : \mathbb{R}^d \to \mathbb{R}$ *as well as the embedding function* $\Phi(\cdot,\cdot)$ *such that below hold:*
1. Correctness: $s(\Phi(P;\zeta)) > s(\Phi(P';\zeta))$, *for* **correct programs** $P \in \mathcal{L}$ *and incorrect programs* $P' \in \mathcal{L}$.
2. Monotonicity: *Let* $\mathcal{P}_1, \dots, \mathcal{P}_r$ *denote the top-K programs returned for each subproblem with specification* ζ_j, $1 \le j \le r$ *and let the final set of programs be* $\mathcal{P} = \{P, s.t. P = Op(P_1, \dots, P_r), P_j \in \mathcal{P}_j\}$. *Then, the following holds:*

$$\forall P \in \mathcal{P}, \forall P' \in \mathcal{L} \setminus \mathcal{P}, s(\Phi(P;\zeta)) \ge s(\Phi(P';\zeta)) \Rightarrow$$
$$\forall P_j \in \mathcal{P}_j, \forall P'_j \in \mathcal{L} \setminus \mathcal{P}_j, s(\Phi(P_j;\zeta_j)) \ge s(\Phi(P'_j;\zeta_j)).$$

To learn the ranking function, we use a benchmark of real-world programming tasks that should capture the typical user-intent. Each task has a set of input-output examples; while we provide a small number of them for synthesizing the program, the remaining I/O examples are used for testing if a synthesized program succeeds on the task. Designing such a function requires further solving the following two key challenges:

(1) **Biased training data**: Learning a ranking function requires generating data from the PBE system itself (by applying it to a few tasks in the benchmark). To bootstrap and to generate training data, we supply the PBE system with a baseline ranker s_0 (e.g., a ranker that generates random scores); generated training data is used to learn a new ranker s_1. When we deploy s_1, the distribution of the subprograms generated itself changes based on s_1's rankings, hence the accuracy can be arbitrarily poor as s_1 was trained on data generated from s_0.
(2) **Distant supervision**: Though the ranking function s is applied to rank smaller subprograms as well as the final programs, the feedback (correctness label) is available only for final programs; i.e., we can apply the final set of synthesized programs on unseen inputs to measure their accuracy, but we cannot get similar feedback for their subprograms.

Algorithm 1 Algorithm for training ML-PROSE.

```
function ML-PROSE($\mathcal{L}, \theta_0, T = \{\zeta_i, i \in [|T|]\}, \Gamma$)
1:  $w(P')_0 = 1$ for all $P' \in CFG(\mathcal{L})$
2:  for all $0 \le \tau \le \Gamma$ do
3:      $\mathcal{P}_j = Synthesis(h_\tau, \zeta_j), 1 \le j \le |T|$, Synthesized programs by applying $s_\tau$ to spec $\zeta_j$
4:      Assign $y_P = 1$ for each correct $P \in \mathcal{P}_j, \forall j$
5:      Assign $y_P = -1$ for each incorrect $P \in \mathcal{P}_j, \forall j$
6:      $\theta = \theta_\tau, w = w_\tau$
7:      while not converged do
8:          Compute $\Phi(P, \zeta_j; w_\tau)$ using (1), $P \in \mathcal{P}_j, \forall j$
9:          Update $\theta$ by solving (3) with fixed $w$
10:         Update $w$ by solving (3) with fixed $\theta$ and $s(P, \zeta; w)$ computed recursively using (2)
11:     $w_{\tau+1} = w, \theta_{\tau+1} = \theta$
12: return $s_\Gamma = (w_\Gamma, \theta_\Gamma)$
```

4.1 Learning to Rank Programs/Subprograms

In this section, we describe three methods to generate ranking problems; successive methods capture the problem structure better and try to address the above mentioned challenges more directly. In this work, we focus on linear scoring functions over the embedding Φ (parameterized by w) defined in (1) (see Remark 4 for discussion on non-linear functions), i.e., the score $s(P,\zeta;w)$ for program P with spec ζ is given by: $s(P,\zeta;w) := \theta^T\Phi(P,\zeta;w)$, θ are the weights. If $P = Op(P_1, \dots, P_r)$ then,

$$s(P,\zeta;w) = \theta_{Op}^T \Phi_{Op}(P,\zeta;w) + \sum_j w(P_j) s(P_j, \zeta_j; w), \tag{2}$$

is a recursive scoring (ranking) function as desired, where θ_{Op} is the projection of θ onto given features for operator Op. Note that $w(P_j) \ge 0$ is a necessary condition for satisfying monotonicity (Definition 3). Now, we want to learn weights $w(.) \ge 0$ as well as θ in (2) such that the ranking problem in Definition 3 is feasible and can be solved accurately. For a DSL $\mathcal{L}$, let $T = \{\zeta^1, \zeta^2, \dots, \zeta^{|T|}\}$ denote a set of tasks, where each task corresponds to an I/O spec $\zeta^\tau = \{\sigma_i^\tau \mapsto \psi_i^\tau\}_{i=1}^{m_\tau} \cup \{\sigma_i^\tau\}_{i=m_\tau+1}^{n_\tau}$. For task ζ^τ, let $\mathcal{P}_{\zeta^\tau}$ denote the set of programs synthesized. It is *always* possible to generate at least one correct program for *offline* training tasks by providing sufficiently many I/O examples (as search returns only consistent programs). Note that correctness of a program (if it produces the desired output on all unseen inputs as well) can be easily determined for *training data.* Let $y(P) = 1$ if $P \in P_{\zeta^\tau}$ is correct for task ζ^τ, else $y(P) = -1$.

(I) Basic formulation (ML-PROSE): In the first formulation, we avoid the challenges mentioned in the previous section by starting with a random ranker and by comparing only the final programs. That is, the goal is to learn a scoring function that ranks any correct program above all incorrect programs, i.e. $\theta^T \Phi(P_a, \zeta; w) > \theta^T \Phi(P_b, \zeta; w)$, for programs $P_a, P_b \in \mathcal{P}(\zeta^\tau)$ generated for a task τ, such that $y(P_a) = 1$ and $y(P_b) = -1$. More generally, we want to penalize the difference between their scores using a suitable loss function ℓ. The corresponding optimization problem is written as:

$$\min_{\theta,w} \sum_{\tau=1}^{|T|} \sum_{\substack{P\in\mathcal{P}_{\zeta^\tau},\\ y(P)=1}} \sum_{\substack{P'\in\mathcal{P}_{\zeta^\tau},\\ y(P')=-1}} \ell\big(s(P,\zeta^\tau;w) - s(P',\zeta^\tau;w)\big) \tag{3}$$

$$+C_1\big\|\theta\big\|_2^2 + C_2 \sum_{Op\in CFG(\mathcal{L})} w_{Op}^2, \quad s.t. \quad w_{Op} \geq 0, \forall Op,$$

where $s(.)$ is defined in Equation (2) and the loss function $\ell(a)$ penalizes negative a; we use standard hinge loss for ℓ in our experiments. We solve the above given problem by alternating over θ and w; note that each of the sub-problems for θ and w is individually convex and easy to optimize.

Remark 4 (Non-linearity). *We can capture non-linearity in the ranking model by generating polynomial features for the local features Φ_{Op}. This enables learning complex scoring functions like the one in Figure 4 (See Appendix).*

(II) Handling distant supervision (ML-PROSE-SubPRG): The above formulation ignores the fact that subprograms are generated by solving smaller synthesis problems. So, even if the scoring function s is accurate for final programs, it can be arbitrarily poor for the subprograms. We alleviate this issue partially by sampling subprograms in the training data to solve Problem (3). We use a baseline ranker to generate both the final programs as well as the subprograms and include a sample from the subprograms in Problem (3). We address the issue of distant supervision by fixing "correctness" of a subprogram P' as follows: $y(P^{'}) = 1$ if P' appears as part of at least one correct program for a given task, or else we assign $y(P') = -1$. Table 2 clearly shows that the ranking function can be improved significantly by inclusion of subprograms when solving (3).

(III) MinMax formulation: Problem (3) does not directly address the crucial requirement of deductive program synthesis (even if we include sampled subprograms as in ML-PROSE-SubPRG) — we want *all* the subprograms of a *given* correct program to be ranked correctly during synthesis. Furthermore, it suffices to rank *any one* correct program above *all* incorrect programs for a given task. For the subprograms of given correct program P for task ζ, i.e., $P_j \in \mathcal{T}(P)$, let $\zeta_j, j = 1, 2, \ldots, |\mathcal{T}(P)|$ denote their respective subproblem specification. Let $\mathcal{P}_{\zeta_j}$ denote the set of all programs in $\mathcal{L}$ that satisfy ζ_j, for each j. Of course, it is impossible to enumerate the entire set, but we can sample many such subprograms for each subproblem specification. We determine the label for the subprograms in $\mathcal{P}_{\zeta_j}$ as before (+1 if the subprogram is part of at least one correct program for the task, or else -1). Define the loss on a correct program P as the max over the losses of all the comparisons during its synthesis:

$$\Delta(P) = \max_{P_j\in\mathcal{T}(P)} \max_{\substack{P'_j\in\mathcal{P}_{\zeta_j}\\ y(P'_j)=-1}} \ell\big(s(P_j,\zeta_j;w) - s(P'_j,\zeta_j;w)\big). \tag{4}$$

We solve:

$$\min_{\theta,w\geq 0} \sum_{\tau=1}^{|T|} \min_{\substack{P\in\mathcal{P}_{\zeta^\tau},\\ y(P)=1}} \Delta(P) + C_1\big\|\theta\big\|_2^2 + C_2 \sum_{Op} w_{Op}^2. \tag{5}$$

The above optimization problem is non-convex even in θ, however, we can still define sub-gradient for the problem. In particular, we implement stochastic sub-gradient descent method for this problem using the widely-used Tensorflow framework (`www.tensorflow.org`).

4.2 Iterative Training

The above formulations still do not address the biased training data challenge. In fact, even if we have a good ranker s_0 to bootstrap with, the bias of baseline ranker still persists. To alleviate this concern, we use an iterative scheme to ensure that the train-test distribution for our ranking function matches while we improve the ranking function itself. Using a base ranker we synthesize programs for the training tasks, sample programs and solve problem (3) (or (5), for training the MinMax model). We then deploy the learned ranker in the PBE system, synthesize (possibly different set of) programs for the tasks, sample programs afresh again to re-learn the ranking model, and repeat. The iterative procedure is described in Algorithm 1 (and in Figure 3 of Appendix) and is able to handle the biased training data issue effectively. In order to ensure *smooth* refinement of s, we combine data from a few recent iterations $\mathcal{D}_1 \cup \mathcal{D}_2 \ldots \cup \mathcal{D}_\Gamma$; here $\mathcal{D}_\tau$ is the training dataset generated using s_τ. This also helps us avoid poor local minima and helps the ranker converge to a reasonable stationary point.

5 Experiments

We have implemented our learning approach in the PROSE (2015) framework, which is the state-of-the-art PBE system for data wrangling tasks, and is publicly available for academic use.

Benchmark tasks. We use 740 real-world string/date/time manipulation tasks obtained from Polozov and Gulwani (2015). Each task in the benchmark consists of a list of input strings and their corresponding outputs (See Appendix C). The available number of I/O examples per task varies from two to a few hundreds. We use 100 tasks for training, and the remaining 640 for testing. Permuting the order of I/O examples in each *training* task, and varying the spec size m, we get several variants of a single training task. Results are reported on the 640 test tasks used as-is from the benchmark.

Performance Metrics. We want the PBE system to get an intended program in top-K. We report results for $K = 1$ (ACC@1) as well as $K = 10$ (ACC@10).

Initial Ranking Model. In our experiments, we use the ranking function that prefers shorter programs as the initial ranking function in Algorithm 1. Natural programs tend to be terse and often short, so this is a reasonable starting point. Here, $s(P;\zeta) = \frac{1}{|\mathcal{C}(P)|}\sum_{P' \in \mathcal{C}(P)} s(P';\zeta) - 1$.

Training Data. At each iteration of Algorithm 1, we take the top-1000 programs for each task generated with the ranking model of the previous iteration. This ensures we have good mix of correct and incorrect programs to sample from. With 100 tasks (and their variants) in the training set, sampling about 40 correct and 40 incorrect programs from each task results in about 1.2M data points in total per iteration for learning the ML-PROSE model (3). To train the MinMax model, we sample 50 correct programs from each training task, to compute the inner $\min$ in (5); on average there are about 20 subprograms per program corresponding to the outer $\max$ in (4) and about 20 (incorrect) subprograms corresponding to the inner $\max$ in (4); this sampling strategy leads to about 2M training data points. ACC@1 for training tasks flattens after about 5 iterations, as shown in Figure 5 (in Appendix A); so we use the ranking function at the end of 6 iterations to report results on test data.

5.1 Results on FlashFill benchmark

Compared methods. Our three proposed ranking algorithms are (i) ML-PROSE where we use only top-level programs for training, (ii) ML-PROSE-SubPRG where we use both programs as well as subprograms for learning the ranking model in the objective (3), and (iii) MinMax model that uses the more directed objective in (5). We compare our methods against four baseline ranking functions: (i) RANDOM ranking function where each weight $\theta_i \sim \text{Uniform}([-1, 1])$; (ii) the initial ranking model outlined earlier, that prefers shorter programs, which we call SHORTEST-PROGRAM (Wang et al., 2017; Osera and Zdancewic, 2015); here, we discard trivial ConstStr programs (which is by definition the shortest program, when the I/O spec has only one example), (iii) a ranking score model that prefers fewer and shorter constants, which we call FEWER-CONSTANTS; good constants like delimiters tend to be short, so this is a reasonable heuristic; (iv) combining the ranking models of SHORTEST-PROGRAM and FEWER-CONSTANTS (i.e. prefer programs that are short *as well as* with fewer, shorter constants).

Accuracy. The results for accuracy at top-1 and at top-10 for the different methods are presented in Table 2 (columns 1-4). The best performing method in terms of ACC@1 is ML-PROSE-SubPRG, which retrieves the intended program at the top in 67% test tasks, *using just one I/O example*. Note that the hand-designed PROSE ranker (that comes with PROSE (2015) SDK, and is shipped as part of Microsoft products including MS Excel, Powershell, and Azure ML), tuned using the entire benchmark, i.e. training *as well as* test tasks, achieves 0.72 top-1 accuracy with $m = 1$. However, its top-10 accuracies are comparable to ML-PROSE-SubPRG. In terms of ACC@10, the MinMax model is the clear winner, in both $m = 1$ and $m = 2$ cases; this suggests that the $\Delta(P)$ loss (4) effectively captures the synthesis-time "competitions" among potential subprograms. Another important takeaway from the results is that the synthesis problem becomes significantly easier with $m = 2$ compared to $m = 1$. This is evident from observing the lift in performance of all the baseline methods, especially the fourth one.

Synthesis time. Our ranking models are competitive compared to the optimized PROSE ranker in terms of synthesis times (i.e. elapsed CPU time to synthesize top-1 program for a given I/O spec). See Figure 5 in Appendix A.

5.2 Comparison to state-of-the-art ML methods

Two important neural synthesis techniques in PBE context are the **RobustFill** framework (Devlin et al., 2017) and the **DeepCoder** framework (Balog et al., 2017). For fair comparison, we conduct experiments on a simpler DSL that Devlin et al. (2017) use. In particular, we use 73 tasks from the FlashFill benchmark, which is an *exact* subset of our 640 test tasks, on which the results are reported in Kalyan et al. (2018). We summarize the results in Table 1 of Kalyan et al. (2018) as well as present comparisons to our method in Table 3 of Appendix A. We find that even the SHORTEST-PROGRAM baseline achieves 32% ACC@1 with $m = 1$, about 7% better than RobustFill with $m = 1$, on the *exact* 73 tasks. The simple baseline performs reasonably well because, in this subset of tasks, 2 or 3 I/O examples are sufficient for the search strategy to find *consistent programs that also generalize very well*; on the other hand, RobustFill cannot even guarantee consistent programs. Our ranker ML-PROSE-SubPRG performs the best on the 73 tasks, achieving 70% ACC@1 with $m = 1$. We exclude comparisons to Menon et al. (2013) as it requires additional information beyond I/O spec for synthesis.

Ranking Method	Acc@1		Acc@10	
	$m = 1$	$m = 2$	$m = 1$	$m = 2$
RANDOM	0.22	0.60	0.38	0.67
(A) SHORTEST PROGRAM	0.37	0.69	0.49	0.80
(B) FEWER CONSTANTS	0.38	0.60	0.59	0.80
(A) and (B)	0.44	0.72	0.60	0.87
ML-PROSE	0.63	0.78	0.73	0.87
ML-PROSE-SubPRG	**0.67**	**0.83**	0.75	0.89
MinMax	0.65	0.81	**0.79**	**0.92**

Table 2: Performance on the FlashFill benchmark. The number of I/O examples given to the PBE system for each of the 640 test tasks is m. The proposed methods, especially ML-PROSE-SubPRG and MinMax, perform significantly better than the baselines. The expert-designed ranker, currently shipped as part of several Microsoft products, tuned using training *as well as* test tasks, gets 0.72 Acc@1 with $m = 1$, and 0.85 with $m = 2$; though its Acc@10 is worse than MinMax.

5.3 Personalization

A single ranking function may not cater to all types of users, even within the same domain. A significant advantage of our ranking solution is that we can re-train the scoring model in order to capture the unique biases/preferences for different user segments. For e.g, geography often determines date/time formats; we want the ranking function to prefer the default formatting style for the specific user locale, unless additional I/O examples overrule the assumed preferences. One simple and effective way to capture these biases is to repeat the task, on which the ranker deviates from the desired behavior, multiple times (or equivalently, weigh the loss associated with this task higher). Below, we present two scenarios for personalized ranking.

Rounding Numbers. Say we want to induce the following preference for rounding a number: "Nearest" $>$ "TowardsZero" $>$ "Down". The preference that our method (using MinMax ranking formulation) learns from the training data is "TowardsZero" $>$ "Nearest" $>$ "Down" (See Figure 7, Appendix D). Learning this preferential order from the randomly sampled training data is likely because in many number transformation tasks where "Nearest" rounding operation applies, "TowardsZero" also leads to correct programs (and "Down" is the least representative rounding operation in the entire benchmark). By replicating three training tasks that induce the preferred rounding behavior 10 times and re-training, the (MinMax) ranking model learns "Nearest" as the most-preferred rounding operation (See Figure 8 and Example 1, Appendix D).

Formatting Dates. In many tasks, the intended output format is ambiguous unless one looks at several I/O examples. Say, some users prefer "m/d" to "M/dd" (2/3 vs 02/03 for 3rd Feb) for date, or "h:mm:ss" to "hh:mm:ss" for time. Our (MinMax) ranker learns a preference towards "mm/dd" and "hh:mm:ss" formats which are representative of the training data. By replicating 2 tasks that induce the desired formatting behavior in the training data and retraining, the ranking model learns the desired formatting preferences (See Example 2, Appendix D).

Remark 5 (Maintenance and Debugging)**.** *The personalization scenarios above also imply another significant advantage of our ML-based ranking solution over neural synthesis approaches — transparency. It is crucial for an ML-based PBE system to be maintainable and debuggable.*

6 Related Work

As mentioned in Section 1, there are two lines of work on program synthesis, symbolic and ML/neural-synthesis based approaches. For symbolic techniques, Gulwani (2010) and Gulwani et al. (2017) provide extensive surveys. State-of-the-art neural program synthesis techniques have already been mentioned/discussed earlier. See Gulwani and Jain (2017) for recent results that are at the intersection of ML and PL. Statistical learning techniques for PBE have also received some attention. Ellis and Gulwani (2017) try to improve the accuracy of existing PROSE implementations, by learning to re-rank the top K consistent programs for given I/O spec, assuming a "good" ranking function is already in place unlike our approach. The statistical learning framework of Menon et al. (2013) employs a log-linear model for inferring likelihood of consistent programs from a probabilistic CFG. In addition to I/O spec, it also needs "clues" to be able to narrow down the rules to consider for enumeration, so that synthesis time is not prohibitive. Singh and Gulwani (2015) learn a ranking function (that prefers generalizable programs) using only top-level programs but apply the learned function recursively to rank subprograms during synthesis; their method has not been implemented in a PBE system to demonstrate real-time synthesis. Raychev et al. (2016) focus on the synthesis setting where one has access to many, and potentially noisy, I/O examples. Christakopoulou and Kalai (2017) specify intent through a "glass-box" scoring program that evaluates candidate programs; they do not use any I/O spec.

References

Alur, R., Bodík, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. (2013). Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–8.

Alur, R., Radhakrishna, A., and Udupa, A. (2017). Scaling enumerative program synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 319–336. Springer.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. (2017). Deepcoder: Learning to write programs. *International Conference on Learning Representations (ICLR)*.

Bunel, R., Hausknecht, M., Devlin, J., Singh, R., and Kohli, P. (2018). Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*.

Christakopoulou, K. and Kalai, A. T. (2017). Glass-box program synthesis: A machine learning approach. *arXiv preprint arXiv:1709.08669*.

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.-r., and Kohli, P. (2017). Robustfill: Neural program learning under noisy i/o. In *International Conference on Machine Learning*, pages 990–998.

Ellis, K. and Gulwani, S. (2017). Learning to learn programs from examples: Going beyond program structure. *IJCAI*.

Gulwani, S. (2010). Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 13–24. ACM.

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

Gulwani, S., Hernández-Orallo, J., Kitzelmann, E., Muggleton, S. H., Schmid, U., and Zorn, B. (2015). Inductive programming meets the real world. *Communications of the ACM*, 58(11):90–99.

Gulwani, S. and Jain, P. (2017). Programming by examples: PL meets ML. In *Asian Symposium on Programming Languages and Systems*, pages 3–20. Springer.

Gulwani, S., Polozov, O., Singh, R., et al. (2017). Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119.

Kalyan, A., Mohta, A., Polozov, O., Batra, D., Jain, P., and Gulwani, S. (2018). Neural-guided deductive search for real-time program synthesis from examples. *International Conference on Learning Representations (ICLR)*.

Le, X.-B. D., Chu, D.-H., Lo, D., Le Goues, C., and Visser, W. (2017). S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 593–604. ACM.

Liu, T.-Y. et al. (2009). Learning to rank for information retrieval. *Foundations and Trends® in Information Retrieval*, 3(3):225–331.

Menon, A., Tamuz, O., Gulwani, S., Lampson, B., and Kalai, A. (2013). A machine learning framework for programming by example. In *International Conference on Machine Learning*, pages 187–195.

Osera, P.-M. and Zdancewic, S. (2015). Type-and-example-directed program synthesis. In *ACM SIGPLAN Notices*, volume 50, pages 619–630. ACM.

Padhi, S., Jain, P., Perelman, D., Polozov, O., Gulwani, S., and Millstein, T. (2017). Flashprofile: Interactive synthesis of syntactic profiles. *arXiv preprint arXiv:1709.05725*.

Parisotto, E., Mohamed, A.-r., Singh, R., Li, L., Zhou, D., and Kohli, P. (2016). Neuro-symbolic program synthesis. In *International Conference on Learning Representations (ICLR)*.

PCWorld (2012). Microsoft Office 2013 Preview: Hands On.

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

PROSE (2015). Microsoft SDK.

Raychev, V., Bielik, P., Vechev, M., and Krause, A. (2016). Learning programs from noisy data. In *ACM SIGPLAN Notices*, volume 51, pages 761–774. ACM.

Rolim, R., Soares, G., D'Antoni, L., Polozov, O., Gulwani, S., Gheyi, R., Suzuki, R., and Hartmann, B. (2017). Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, pages 404–415. IEEE Press.

Singh, R. and Gulwani, S. (2015). Predicting a correct program in programming by example. In *International Conference on Computer Aided Verification*, pages 398–414. Springer.

Wang, X., Dillig, I., and Singh, R. (2017). Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages*, 2(POPL):63.

A Ranking: Details

A.1 Iterative Training

Training the ranking model in our ML-based PBE system (Algorithm 1) is depicted in Figure 3.

A.2 Training Accuracy with Increasing Iterations

Figure 5 shows how the training accuracy (at top 1) improves and stabilizes with the number of iterations when training ML-PROSE. The first iteration corresponds to the initial ranking function SHORTEST-PROGRAM described in Section 5. In the second iteration, the accuracy on the training tasks increases significantly. After about 5 iterations, the accuracy stabilizes (we also observe that other metrics stabilize as well — such as percentage of correct programs in the top-1000 programs retrieved for each task, though not shown in the plot). We deploy the ranking function obtained after 6 iterations for evaluating the performance of our methods on test tasks, in Section 5.

A.3 Hand-Tuned PROSE Ranker

Figure 4 serves to illustrate the complexity of the ranking function in the PROSE implementation PROSE (2015). This manually-devised ranking heuristic scoring function for the AbsPos operator in the FlashFill grammar is provided in the opensource PROSE framework. Clearly the heuristic has been carefully tuned manually with domain insights. The engineering effort required to design such scoring functions for the entire grammar is conceivably quite high, and requires re-engineering the solution for every new operator or new domain.

A.4 Comparison to State-of-the-Art Neural Synthesis Methods

Here, we report results on a subset of 73 tasks from the Flashfill benchmark used in Kalyan et al. (2018). This is a strict subset of our 640 test tasks that we report results on in Section 5. For neural synthesis methods, we simply quote results presented in Table 1 of Kalyan et al. (2018). We run the baseline SHORTEST-PROGRAM and our method ML-PROSE-SubPRG on the exact 73 test tasks and present comparisons in Table 3.

A.5 Synthesis Times

A scatter plot of synthesis times of test tasks, comparing our ML-PROSE method and the highly-optimized PROSE ranker are presented in Figure 5. By synthesis time, we mean the wall-clock time elapsed between invoking the PBE system with an I/O spec and the system returning the synthesized (top-1) program. The ranking functions learned using ML-PROSE-SubPRG and MinMax formulations yield very similar synthesis times, and are not presented. It is extremely important to be able to do synthesis in real-time in several end-user applications, especially in an interactive setting. We observe from the figure that the times taken to synthesize the top-1 program for the two ranking functions are comparable. Furthermore, both the ranking functions take less than a second for a majority of the tasks.

B FlashFill DSL

The complete FlashFill DSL we work with is presented in Table 6. In the following, we give descriptions of the operators and the data types of the DSL.

B.1 Operators

The operators IfThenElse, Concat, ConstStr, SubStr, ToUppercase, ToLowercase, ToTitlecase are self-explanatory. Kth simply returns the kth string in the input array $inputs$.

- Matches checks if the given string s is generated by the given regular expression r.
- RegexOccurrence finds the k^{th} occurrence of regex r in x and returns its boundaries. e.g. RegexOccurrence(x,“number”, 2), on input x = “12th Ave, Seattle 98003” returns the beginning and the ending indices of the substring “98003”.
- AbsPos directly indexes into a given position of the input string.
- RegexPosition indexes into the k^{th} occurrence of a pair of regular expressions in the input string. e.g. RegexPosition(x, std.Pair(“CommaOrWhiteSpace”, “NumberOrWhiteSpace”), 1), on input x = “12th Ave, Seattle 98003” returns the beginning and the ending indices of the substring “Seattle”.
- Lookup searches for the input string in the specified dictionary. e.g. Lookup(x, { “Male”: “M”, “Female”: “F” }) returns “M” or “F” depending on the input string, and null if the input string is neither “Male” nor “Female”.
- ParsePartialDateTime parses the input string using the specified date/time format(s), and instantiates an object of type `PartialDateTime`.

  e.g. ParsePartialDateTime(x, [“d MMM yyyy”, “dd MMM yyyy”, “d-MMM-yyyy”, “dd-MMM-yyyy”]) parses input strings like “17 Dec 1999”, “1-Jan-2001” as expected, but not “17 12 1999” or “1-Jan-01”.

  (See Appendix B.2 for a description of `PartialDateTime` type)

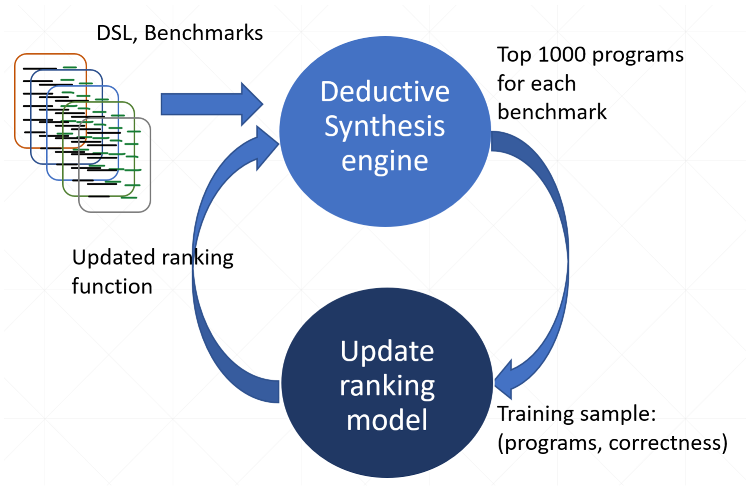

DSL, Benchmarks
Deductive Synthesis engine
Top 1000 programs for each benchmark
Updated ranking function
Update ranking model
Training sample: (programs, correctness)

Figure 3: End-to-End training of the PBE system: At each iteration, new programs are synthesized based on the ranking function learned during the previous iteration, which are then fed into training the ranking model and so on.

RANKING METHOD	ACC@1
PROSE $(m = 1)$	0.67
DC $(m = 1)$	0.36
DC $(m = 2)$	0.47
DC $(m = 3)$	0.63
RF $(m = 1)$	0.25
RF $(m = 2)$	0.40
RF $(m = 3)$	0.56
NGDS $(m = 1)$	0.68
SHORTEST-PROGRAM $(m = 1)$	0.32
ML-PROSE-SubPRG $(m = 1)$	**0.70**

Table 3: Performance on a subset of FlashFill benchmark consisting of 73 tasks used in Kalyan et al. (2018), which is a strict subset of our 640 test tasks. The results corresponding to rows 1 through 8 are quoted from Table 1 of Kalyan et al. (2018); RF stands for RobustFill (Devlin et al., 2017), and DC stands for DeepCoder (Balog et al., 2017). The last two rows correspond to results we obtain by running the baseline and our method on the exact set of 73 tasks. Note that when $m = 1$, i.e. only 1 I/O example case, even the SHORTEST-PROGRAM baseline is comparable to or outperforms neural synthesis methods. Our method ML-PROSE-SubPRG achieves the best accuracy in this setting (we get one more test task correct compared to NGDS). See Section 5 for discussion.

```
static double Score_AbsPos(double k)
{
k = 1 / k - 1;
// Prefer absolute positions to regex positions if k is small
return Math.Max(10 * Token.MinScore - (k
     - 1) * 3 * Token.MinScore, 1 / k);
}
```

Figure 4: Scoring function for the AbsPos operator of the FlashFill DSL presented in Table 1. This is part of Microsoft's PROSE (2015) SDK, made available for academic use. The scoring function is opaque from a software maintenance pespective, and can be very challenging to engineer even for relatively straight-forward operators.

- RoundPartialDateTime performs the specified rounding on the input `PartialDateTime` object and returns a new instance of the same type.

  e.g. RoundPartialDateTime(ParsePartialDateTime(x, ["d MMM yyyy HH:mm", "dd MMM yyyy HH:mm"]), "(, 30, Minute, Down, , 0)") first parses input string "17 Dec 1999 03:55" into appropriate PartialDateTime object; and rounds the minutes part from "55" to "30", which is the closest multiple of 30 that is less than or equal to the specified minutes, and returns the updated object.

  (See Appendix B.2 for a description of `DateTimeRoundingSpec` type)

- FormatPartialDateTime formats the input `PartialDateTime` object in the specified output format.

  e.g. FormatPartialDateTime(ParsePartialDateTime(x, ["d MMM yyyy", "dd MMM yyyy", "d-MMM-yyyy", "dd-MMM-yyyy"]), "dd/MM/yyyy") returns "01/01/2001" for the input string "1-Jan-2001".

- FormatDateTimeRange formats the input `PartialDateTime` object as a range: the lower end of the range is specified in the first `DateTimeRoundingSpec` argument and the upper end of the range is specified in the second `DateTimeRoundingSpec` argument, and s is used as the range delimiter string in the output.

  e.g. FormatDateTimeRange(ParsePartialDateTime(x, ["htt"]), "h:mmtt", "-",

  "(,2,Hour,Down,,0)", "(,2,Hour,UpOrNext,Minute,1)") formats the input string "2PM" as "1:00PM-2:59PM".

- AsDecimal casts the input as `decimal` type (has more precision and smaller range, and is appropriate for computations arising in spreadsheets such as in the financial domain).

- ParseNumber parses the input string using the given number format specification, and returns the number as `decimal` type. e.g. ParseNumber(x, "(',', , ,'.',)") parses the input string as a number using '.' as the separator between the decimal and the integral parts and ',' as the separator for segments before the decimal (e.g. $x =$ "4, 999.99").

  (See Appendix B.2 for a description of `NumberFormatDetails` type)

- RoundNumber performs the specified rounding operation (of type `RoundingSpec`) on the input `decimal` number.

  e.g. RoundNumber(ParseNumber(x, "(',', , ,'.',)"), "(0, 1000, Nearest)"), on input $x =$ "1, 954" returns number 2000, and on $x =$ "458" returns 0.

  (See Appendix B.2 for a description of `RoundingSpec` type)

- FormatNumber formats the input number in the specified output format (of type `NumberFormat`). e.g. FormatNumber(ParseNumber(x, "(',', , ,'.',)"), "(2U, 2U, , , , (, , ,'.',))"), on input $x =$ "123.4567" returns "123.46", and on $x =$ "102" returns "102.00" (i.e. the format ensures a minimum and a maximum of 2 digits in the decimal part).

  (See Appendix B.2 for a description of `NumberFormat` type)

- FormatNumericRange formats the input number as a range: the lower end of the range is specified in the first `RoundingSpec` argument and the upper end of the range is specified in the second `RoundingSpec` argument, and s is used as the range delimiter string in the output.

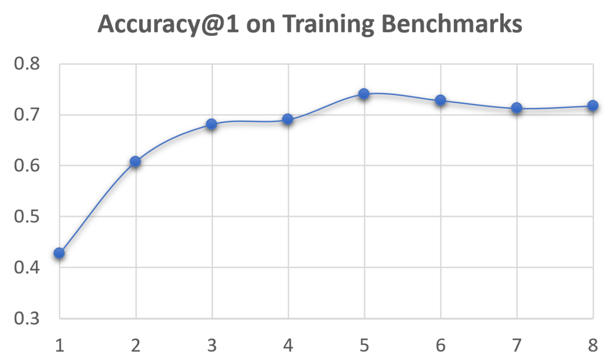
  e.g. FormatNumericRange(ParseNumber(x, "(',', , ,'.',)"), "(2U, 2U, , , , (, , ,'.',))", "-", "(1,25,Down)", "(0,25,Up)") formats the input string "46" as the numeric range "26.00-50.00".

B.2 Data Types

The custom-defined data types are as follows:

- `NumberFormatDetails` type has 5 members:

  1. `DecimalMarkChar` that separates decimal from integral part of a number,
  2. `SeparatorChar` that separates segments in the integral part of a number,
  3. `Scale` factor to apply before formatting a number (usually a power of 10); used for handling percentages



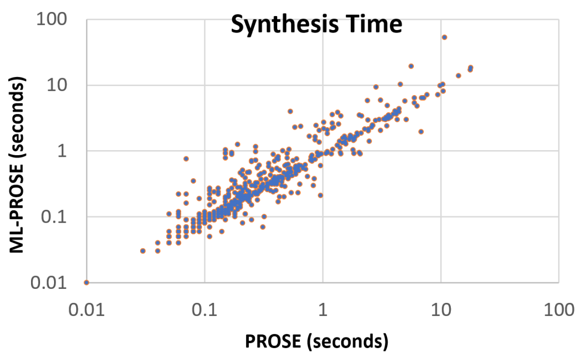


Figure 5: (Left) Mean percentage accuracy at top-1 for the *training* tasks with increasing iterations of Algorithm 1. The first iteration ranker is the SHORTEST-PROGRAM ranking function; with just one iteration, we see that the accuracy increases significantly. (Right) Comparison of synthesis time on test tasks for hand-tuned PROSE ranker for FlashFill grammar with the ranker learned using our approach ML-PROSE. We observe that ML-PROSE ranking model is also as efficient as the manually-optimized ranker, and is faster in certain cases. The ranking functions learned using ML-PROSE-SubPRG and MinMax formulations yield very similar synthesis times as ML-PROSE, hence not shown.

```
// Nonterminals
@start program := transform |
If(cond) Then(transform) Else(program)
bool cond := let s : Kth(inputs, k) in Matches(s, r)
string transform := atom | Concat(atom, transform)
string atom := ConstStr(s) | let string x : Kth(inputs, k) in conv
string conv := substring |
ToLowercase(substring) | ToUppercase(substring) |
ToTitlecase(substring) | Lookup(x, lookupDictionary) |
FormatPartialDateTime(datetime, oDtFormat) |
FormatNumber(number, oNumberFormat) |
FormatDateTimeRange(iDateTime, oDtFormat, s, dtRoundingSpec, dtRoundingSpec) |
FormatNumericRange(iNumber, oNumberFormat, s, roundingSpec, roundingSpec)

string substring := x | SubStr(x, pp)
Tuple<int, int> pp := Pair(pos, pos) | RegexOccurrence(x, r, k)
int pos := AbsPos(x, k) | RegexPosition(x, Pair(r, r), k)
PartialDateTime datetime := iDateTime | RoundPartialDateTime(iDateTime, dtRoundingSpec)
PartialDateTime iDateTime := ParsePartialDateTime(substring, iDtFormats)
decimal number := iNumber | RoundNumber(iNumber, roundingSpec)
decimal iNumber := AsDecimal(x) | ParseNumber(substring, numberFormatDetails)

// Terminals
@input string[] inputs; string s; int k; Regex r;
NumberFormatDetails numberFormatDetails;
NumberFormat oNumberFormat;
RoundingSpec roundingSpec;
DateTimeRoundingSpec dtRoundingSpec;
DateTimeFormat oDtFormat;
DateTimeFormat[] iDtFormats;
IReadOnlyDictionary<Optional<string>,string> lookupDictionary;
```

Figure 6: The FlashFill DSL Gulwani (2011). A program takes as input a list of strings *inputs*, and returns a string, a concatenation of *atoms*. See Appendix B.1 for description of the `Operators` and Appendix B.2 for `data types`. A simpler version of the DSL is given in Section 2, Figure 1 for clarity.

4. `SeparatedSectionSizes`, a list of the number of digits in the segments before the decimal (e.g. [1,3] in “4,022,111.22”); the first number is the section closest to the decimal point; the last number is used repeatedly for all the remaining sections, and
5. `CurrencySymbol`, an optional currency symbol to be output before the number but after the

sign (if any)

- `NumberFormat` type has the following members:
  1. `MinTrailingZeros`, mininum number of digits after the decimal,
  2. `MaxTrailingZeros`, maximum number of digits after the decimal,
  3. `MinLeadingZeros`, mininum number of digits before the decimal,
  4. `MaxTrailingZeros`, maximum number of digits after the decimal,
  5. `MinTrailingZerosAndWhitespace`, mininum number of characters after the decimal including whitespace padding,
  6. `MinLeadingZerosAndWhitespace`, minimum number of characters before the decimal including whitespace padding,
  7. `FormatDetails`, format specification of type `NumberFormatDetails`

- `RoundingSpec` has 3 members:
  1. `Zero`, the zero-point of the set of numbers to round to; numbers are rounded to some multiple of `Delta` from this value,
  2. `Delta`, the increment between numbers to round to
  3. `Mode`, one of {Nearest, Up, Down} (self-explanatory)

- `DateTimeRoundingSpec` has the following members:
  1. `Zero`, the zero-point of the set of numbers to round to; numbers are rounded to some multiple of `Delta` from this value,
  2. `Delta`, the increment between numbers to round to
  3. `Mode`, one of {Up, UpOrNext, Down} (where “UpOrNext” increases the value by `Delta`, even if already rounded),
  4. `Unit`, the unit of measurement for `Delta`

- `PartialDateTime` has the following members:
  1. [`Year, Month, Day`]
  2. `DayOfWeek` (one of {Sunday, Monday, ...})
  3. [`Hour, Minute, Second`]
  4. `Period` (one of {AM, PM})

- `DateTimeFormat`
  1. `Parts` (of type `DateTimeFormat`[])

- `DateTimeFormatPart` has the following members:
  1. `MatchedPart`, one of {Year, Month, Day, Hour, Minute, Second, Period, DayOfWeek}
  2. `FormatChar`, the character for this format part (repeated)
  3. `MinimumLength`, the minimum number of characters for a number this parses/outputs
  4. `MaximumLength`, the maximum number of characters for a number this parses/outputs
  5. `MinValue`, the least valid value this may parse/output
  6. `MaxValue`, the greatest valid value this may parse/output
  7. `StringLookup`, a dictionary encoding the correspondence between matched strings and parsed values

B.3 Domain-Specific Features

In Table 4, we give a comprehensive list of domain-specific features for the operators of the FlashFill DSL (i.e., the $\Phi_{Op(.)}$ in Equation (2)). The state-of-the-art implementation of the PROSE framework essentially uses the same set of features in the hand-crafted, heuristic ranking function for the FlashFill grammar.

C FlashFill Benchmarks

A few representative training and test tasks from the FlashFill benchmark are given in Tables 5 through 11. In each of the tasks, the benchmark specifies the order and the number of I/O examples (m) required or provided for synthesis (the remainder of the examples $m+1, m+2, \ldots, n$ are given to the system as unlabeled examples, i.e. without their output). In most cases, the number of I/O examples $m = 1$, and in some cases it varies all the way up to $m = 6$. In fact, tasks that require conditional programs (`if-then-else`) by definition need at least $m = 2$ examples to generate a desired program. We vary the order of examples and the size m within each training task, in order to create many variants of the same task, thereby increasing the training data as well as better capturing the distribution of programs.

C.1 Examples of synthesized programs

For the input-output specification { “52” $\mapsto$ “50”, “65” $\mapsto$ “70” }, the (correct) program generated at the top using the learned ranker is:

```
let x: Kth(inputs, 1) in
FormatNumber(RoundNumber(ParseNumber(x, "('
    ,', , , '.', )"), "(0, 10, Nearest)"),
    BuildNumberFormat(null, null, null,
    null, null, "(, , , '.', )"))
```

OPERATOR	FEATURE	DESCRIPTION
AbsPos	AllSameLength	Are all the input strings of same length?
RegexPosition	ProportionNull	What fraction of inputs is `null`?
	NullPenalty	A fixed penalty if ProportionNull is above a threshold
RegexOccurrence	RegexIsConstant	Is regex just a constant expression?
	ConstRegexPenaltyFactorBias	A fixed penalty for using regex extraction if RegexIsConstant is true
	RegexBonusBias	Fixed bonus when RegexIsConstant is false
	ProportionNull	What fraction of inputs is `null`?
	NotMatchedFactor	A fixed penalty if ProportionNull is above a threshold
AsDecimal	CastingInputStringToNumber	Casting a non-number to number?
Concat	BothSidesConstant	Concatenating constant strings?
	ConcatNumbers	Concatenating numbers?
	ValueLenLeft	Length of 1st argument
	ValueLenRight	Length of 2nd argument
	RepeatWholeColumnsCount	Is a whole column used multiple times?
	BothSidesConstant	Are both the arguments constant strings?
	ContainsCommonDelimLeft	Left argument has common delimiters?
	ContainsCommonDelimRight	Right arg. has common delimiters?
ConstStr	ConstStrLength	String length
	LogConstStrLength	Log of ConstStrLength
	IsCommonDelimiter	Part of a commonly-used delimiter set?
	ExamplesCount	How many I/O examples are provided?
	ConstantInInput	Is the constant string part of input?
FormatPartialDateTime	numInputDateFormats	Length of $iDtFormats$ array
	SameDateFormat	numInputDateFormats $== 1 \quad \wedge iDtFormats[0] == oDtFormat$?
	SameNumberPenalty	Are date formats used to reformat a number?
	ExtractionMatches	A relaxation of SameDateFormat (instead of exact equality, look for overlap)
FormatDateTimeRange	SepContainsDigit	Does the separator s contain a digit?
	SepIsSymbolsAndPunctuation	Does s contain a symbol (as defined by Unicode standards) or a punctuation?
	SepIsWhitespace	Does s contain a whitespace character?
	SepIsWrappedByWhitespace	Does s begin and end with whitespace characters?
	IsCommonDateTimeSeparator	Is s a commonly used range separator?
FormatNumericRange	RoundToMultipleOf5	Is `roundingSpec.Delta` a multiple of 5?

Table 4: Domain-specific features for the operators in the FlashFill DSL (given in Table 6). For a description of the operators, see Appendix B.1. In addition to the features above, a) we include a bias feature for all the operators, and b) second and third order features (pairwise and triplet-wise products within each operator) to capture non-linearities.

Input	Output
John Smith\n111 Main St.\nBellevue\nWA\n90111	WA
Frank Thomas\n222 Main St.\nRedmond\nCA\n90112	CA
Mike Myers\n333 Main St.\nKirkland\nMA\n90113	MA

Table 5: Extract states from addresses.

Input	Output
2015-11-08T17:39:24Z	Nov 2015
U$Yvf#@tuy#@!eD3	Not a date.
2015-06-17T08:45:04Z	Jun 2015
2014-08-30T15:07:32Z	Aug 2014

Table 6: Format dates; and handle exceptions.

Input	Output
“col1”:“d”, “col2”:“e”, “col3”:“f”	def
“col1”:“x”, “col2”:“y”, “col3”:“z”	xyz

Table 7: Combine data from multiple columns.

Input	Output
syed e abbas	Abbas, S.
catherine r. abel	Abel, C.
kim abercrombie	Abercrombie, K.
kim b abercrombie	Abercrombie, K.
⋮	⋮
victoria c bailey	Bailey, V.

Table 8: Format names.

Input	Output	Input	Output
-234.52	-243.50	-243	-2.43
-12.5	-12.50	-12.5	-0.125
-2345.23292	-2345.20	-2345.23292	-23.4523292
-1202.3433	1202.30	-1202.3433	12.023433
1202.3433	1202.30	1202.3433	12.023433
23224.1	23224.10	23224.1	232.241

Table 9: (Left) Round numbers; (Right) Scale numbers.

Input	Output	Input	Output
6:54:00 PM	18:54	17:10:52	Between 4PM and 6PM
1:12:00 AM	1:12	16:10:52	Between 4PM and 6PM
4:18:00 AM	4:18	18:10:52	Between 6PM and 8PM
12:12:00 PM	12:12		
0:01:00 AM	0:01		
12:02:00 AM	0:02 AM		

Table 10: (Left) Normalize times; (Right) Format time range.

Input	Output	Input	Output
225-706-7709	225-706-7709	123 Privet Drive	123.Privet.Drive
(225) 706 7709	225-706-7709	31 Thomas Rd	31.Thomas.Rd
(425) 706 7709	425-706-7709	1600 Pennsylvania Ave	1600.Pennsylvania.Ave
325 123 4567	325-123-4567	2000 Spring Rd	2000.Spring.Rd

Table 11: (Left) Format phone numbers; (Right) Format spaces.

which rounds the number to the nearest multiple of 10. For the spec { “17:10:52” $\mapsto$ “4:45PM-5:15PM” }, an intended program that appropriately formats the given time in the right 30-minute range generated is:

```
let x: Kth(inputs, 1) in
    FormatDateTimeRange(
    ParsePartialDateTime(x, ["H\\:m\\:s", "
    H\\:m\\:ss", "H\\:mm\\:s", "H\\:mm\\:ss
    ", "HH\\:m\\:s", "HH\\:m\\:ss", "HH\\:
    mm\\:s", "HH\\:mm\\:ss"]), "h\\:mmtt",
    "-", "({Hour=0, Minute=15, Second=0,
    Millisecond=0, HourInPeriod=12, Period
    =0}, 30, Minute, Down, , 0)", "({Hour
    =0, Minute=15, Second=0, Millisecond=0,
     HourInPeriod=12, Period=0}, 30, Minute
    , UpOrNext, , 0)")
```

The details of the operators used in the above programs are presented earlier in this Appendix.

D Personalization: Details

The scoring functions for the RoundingSpec operator before and after re-learning as discussed in Section 5.3 are given in Figures 7 and 8 respectively.

Remark 6 (Personalized Rounding)**.** *A subtle point to note from the two scoring functions presented in Figures 7 and 8 is that other untoward changes are* not *introduced by re-training. In particular, the ordering “Nearest” > “Up” > “AwayFromZero” remains the same, as the replicated benchmark tasks are inconsequential to this ordering. This usecase highlights that the simple replication (or task-dependent weights in the objective) scheme is effective for introducing personalization without causing unintended consequences.*

Example 1 (Synthesized Rounding Programs)**.** *A relevant number formatting task in the FlashFill benchmark is* {“112” $\mapsto$ “110”, “117” $\mapsto$ “120”,

“11112” $\mapsto$ “11110”, “548” $\mapsto$ “550”}. *Using only the first I/O example, the top program generated with the learned ranker (that uses the scoring function in Figure 7) is:*

```
let x: Kth(inputs, 1) in
FormatNumber(RoundNumber(ParseNumber(x, "('
    ,', , , '.', )"), "(0, 10, TowardsZero)
    "), BuildNumberFormat(null, null, null,
     3, 3, "(, , , '.', )"))
```

which would fail on two I/O examples in the spec, as it rounds the input number towards zero. With two examples, the ranker indeed gets the correct program, but the point here is to be able to tailor the ranker to the type of tasks typically arising in the domain. The re-learnt ranker (Figure 8) produces the desired program with just the first example:

```
let x : Kth(inputs, 1) in
FormatNumber(RoundNumber(ParseNumber(x, "('
    ,', , , '.', )"), "(0, 10, Nearest)"),
    BuildNumberFormat(null, null, null, 3,
    3, "(, , , '.', )"))
```

Example 2 (Personalized Date/Time Formatting)**.** *Consider the task* {“23/12/2010” $\mapsto$ “2010 23 12”, “3/4/2010” $\mapsto$ “2010 3 4”, “1932 97” $\mapsto$ “1932 6 4”, “*$Monday$ #4 $December$* 1973” $\mapsto$ “1973 24 12”}. *With one example, the synthesized top program is*

```
let columnName = "0" in let x : ChooseInput
    (vs, columnName) in
    FormatPartialDateTime(
    ParsePartialDateTime(x, ["d\\/M\\/yyyy
    ", "yyyy\\ j", "dddd\" #\"i\\ MMMM\\
    yyyy"]), "yyyy\\ dd\\ M"),
```

which fails on the second example, producing the wrong output “2010 03 4”, as it formats the date with “`dd`*”. After re-training, the new ranker retrieves the correct program at the top:*

```
let columnName = "0" in let x : ChooseInput
    (vs, columnName) in
    FormatPartialDateTime(
    ParsePartialDateTime(x, ["d\\/M\\/yyyy
    ", "yyyy\\ j", "dddd\" #\"i\\ MMMM\\
    yyyy"]), "yyyy\\ d\\ M").
```

```
public double roundingSpec(double bias_roundingSpec, double base_Delta, double
    base_DeltaIsPowerOf10, double base_Zero, double base_ZeroIsZero, double
    base_RoundingMode, double base_RoundingModeIsNearest, double
    base_RoundingModeIsTowardZero, double base_RoundingModeIsAwayFromZero) {
return 1.885516E-09 * base_Delta +
0.1580201 * base_DeltaIsPowerOf10 +
-1.449533E-05 * base_RoundingModeIsAwayFromZero +
0.03198133 * base_RoundingModeIsNearest +
0.1897284 * base_RoundingModeIsTowardZero +
-0.0004875684 * base_Zero +
-0.05304068 * base_ZeroIsZero +
-0.05818909 * bias_roundingSpec +
2.835578E-07 * base_Delta * base_DeltaIsPowerOf10;
}
```

Figure 7: The learnt scoring function (using Algorithm 1 and MinMax formulation) for the RoundingSpec operator of the FlashFill DSL presented in Table 1.

```
public double roundingSpec(double bias_roundingSpec, double base_Delta, double
    base_DeltaIsPowerOf10, double base_Zero, double base_ZeroIsZero, double
    base_RoundingMode, double base_RoundingModeIsNearest, double
    base_RoundingModeIsTowardZero, double base_RoundingModeIsAwayFromZero) {
return -2.830293E-08 * base_Delta +
1.038527 * base_DeltaIsPowerOf10 +
-0.04912099 * base_RoundingModeIsAwayFromZero +
1.967019 * base_RoundingModeIsNearest +
0.9285749 * base_RoundingModeIsTowardZero +
-0.002863473 * base_Zero +
-0.6954886 * base_ZeroIsZero +
-0.7241547 * bias_roundingSpec +
2.73517E-07 * base_Delta * base_DeltaIsPowerOf10;
}
```

Figure 8: The re-learnt scoring function (using Algorithm 1 and MinMax formulation) for the RoundingSpec operator of the FlashFill DSL presented in Table 1, after replicating 4 pertinent rounding tasks in the training data. The weight for the "Nearest" rounding mode (`base_RoundingModeIsNearest`) has increased significantly (compared to Figure 7), thus letting it taking precedence over "TowardZero" mode as desired. See Appendix D.