Predicting a Correct Program in Programming by Example

Rishabh Singh(✉) and Sumit Gulwani

Microsoft Research, Redmond, USA
`risin@microsoft.com`

We study the problem of efficiently predicting a correct program from a large set of programs induced from few input-output examples in Programming-by-Example (PBE) systems. This is an important problem for making PBE systems usable so that users do not need to provide too many examples to learn the desired program. We first formalize the two classes of sharing that occurs in version-space algebra (VSA) based PBE systems, namely set-based sharing and path-based sharing. We then present a supervised machine learning approach for learning a hierarchical ranking function to efficiently predict a correct program. The key observation of our learning approach is that ranking any correct program higher than all incorrect programs is sufficient for generating the correct output on new inputs, which leads to a novel loss function in the gradient descent based learning algorithm. We evaluate our ranking technique for the FlashFill PBE system on over 175 benchmarks obtained from the Excel product team and help forums. Our ranking technique works in real-time, reduces the average number of examples required for learning the desired transformation from 4.17 to 1.48, and learns the transformation from just one input-output example for 74 % of the benchmarks. The ranking scheme played a pivotal role in making FlashFill usable for millions of Excel users.

1 Introduction

Millions of computer end users need to perform repetitive tasks, but unfortunately lack the programming expertise required to do such tasks automatically. Example-based program synthesis techniques have the potential to enhance the productivity of such end users by enabling them to create small scripts using examples [8,9]. These techniques have been developed for a wide variety of domains including repetitive text-editing [14], syntactic string transformations [7], semantic string transformations [23], table transformations [11], and number transformations [24]. FlashFill [1,7] is a recent system in Excel 2013 that learns syntactic string transformation programs from examples.

Many recent Programming-By-Example (PBE) techniques use version-space algebra (VSA) [14] based methodology of computing the set of *all* programs in an underlying domain-specific language (DSL) that are consistent with a given set of input-output examples. The number of such programs is huge; but they are all succinctly represented using appropriate data-structures that share common program fragments. Given a representative set of input-output examples

for a task, all synthesized programs would be *correct*, i.e. the programs would correspond to the intended task. However, if only a few input-output examples are given (i.e. the task is under-specified), the set of synthesized programs will include both correct and incorrect programs. The user would then need to refine the specification by providing additional input-output examples to avoid learning an incorrect program. The number of representative input-output examples required to learn a desired task is a function of the underlying DSL and has also been referred to as the *learning dimension* [6] of the DSL. A more expressive DSL makes the synthesizer more useful (since it can assist users with a larger variety of tasks), but it also makes the synthesizer less usable (since users now need to provide more examples).

We study the problem of predicting a correct program from a huge set of programs in an expressive DSL that have been induced by a *small* number of examples. We propose a machine learning based ranking technique to rank the induced programs by assigning them a likelihood score based on their features. While machine learning has been used in the past to improve the efficiency of heuristic-based enumerative search in program synthesis [17], we leverage machine learning in a different manner: the VSA based programming-by-example techniques set up the space of programs (that are consistent with the user-provided examples) over which machine-learning based ranking is performed to predict a correct program. There are two key challenges that our technique addresses, namely that of automatically learning the ranking function, and that of efficiently identifying the highest ranked program from a large set of induced programs in a VSA representation.

We formalize the problem of learning a ranking function as a machine learning problem and present a novel solution to it. Traditional learning-to-rank approaches [2–4,12] either aim to rank all relevant documents over all non-relevant documents or rank the most relevant document at the top. We, instead, study the problem of ranking *some* correct program over *all* incorrect programs as any correct program would be sufficient to generate the desired outputs on new inputs. Our solution involves two key ideas: (a) we present a gradient descent based approach to learn the coefficients (weights) of a linear ranking function with the goal of ranking some correct program over all incorrect programs. (b) we also provide an automated method to obtain the labeled training data for our learning algorithm from training benchmark tasks.

A key challenge in using any ranking methodology for VSA based PBE systems is that of efficiency. The naïve approach of explicitly computing the rank for each induced program does not scale because the number of induced programs is often huge (more than 10^{20} [23]). These programs are represented using succinct data structures that allow sharing of expressions across different levels. We formalize two general classes of sharing that occurs in these data-structures [7,11,23,24], namely *set-based sharing* and *path-based sharing*. We learn a separate ranking function for each level of sharing—this enables us to apply the ranking methodology efficiently in practice.

We instantiate our ranking technique for the FlashFill synthesis algorithm [7]. The VSA based data-structure in FlashFill involves two levels of sharing.

We learn a separate ranking function for each level over corresponding efficient features (Sect. 5). We present the evaluation of our ranking technique on over 175 string manipulation tasks obtained from Excel product team and help-forums. The ranking scheme works in real-time and reduces the average number of examples required per benchmark to 1.48 as compared to 4.17 examples needed by a manually defined ranking scheme based on Occam's razor [7]. Our machine-learning based ranking scheme played a pivotal role in making FlashFill successful and usable for millions of Excel users.

This paper makes the following contributions.

- We formalize the two different classes of sharing used in VSA based representations, namely set-based sharing and path-based sharing(Sect. 3).
- We describe a machine-learning based technique to rank *some* correct program over all incorrect programs for most benchmarks in the training set (Sect. 4.3).
- We demonstrate the efficacy of our ranking technique for FlashFill on over 175 real-world benchmarks (Sect. 5.2).

2   Motivating Examples

In this section, we present a few motivating examples from FlashFill that show three observations: (i) there are multiple correct programs in the set of programs induced from an input-output example, (ii) simple features such as size are not sufficient for preferring a correct program over incorrect programs, and (iii) there are huge number of programs induced from a given input-output example.

Example 1. An Excel user had a series of names in a column and wanted to add the title `Mr.` before each name. She gave the input-output example as shown in the table to express her intent. The intended program concatenates the constant string `"Mr."` with the input string in column v_1.

	Input v_1	Output
1	Roger	Mr. Roger
2	Simon	
3	Benjamin	
4	John	

The challenge for FlashFill to learn the desired transformation in this case is to decide which substrings in the output string "`Mr. Roger`" are constant strings and which are substrings of the input string "`Roger`". We use the notation $s[i..j]$ to refer to a substring of s of length $j - i + 1$ starting at index i and ending at index j. FlashFill infers that the substring $\mathtt{out}_1[0..0] \equiv$ "`M`" has to be a constant string since "`M`" is not present in the input string. On the other hand, the substring $\mathtt{out}_1[1..1] \equiv$ "`r`" can come from two different substrings in the input string ($\mathtt{in}_1[0..0] \equiv$ "`R`" and $\mathtt{in}_1[4..4] \equiv$ "`r`"). FlashFill learns more than 10^3 regular expressions to compute the substring "`r`" in the output string from the input string, some of which include: 1st capital letter, 1st character, 5th character from end, 1st character followed by a lower case string etc. Similarly, FlashFill learns more than 10^4 expressions to extract the substring "`Roger`" from the input string, thereby learning more than 10^7 programs from just one input-output example. All programs in the set of learnt programs that include

an expression for extracting "r" from the input string are incorrect, whereas programs that treat "r" as a constant string are correct. Some hints than can help FlashFill rank constant expressions for "r" higher are:

- Length of substring: Since the length of substring "r" is 1, it is less likely to be an input substring.
- Relative length of substring: The relative length of substring "r" as compared to the output string is small $\frac{1}{9}$.
- Constant neighboring characters: The neighboring characters "M" and "." of "r" are both constant expressions.

Example 2. An Excel user had a list of names consisting of first and last names, and wanted to format the names such that the first name is abbreviated to its first initial and is followed by the last name as shown in the table.

	Input v_1	Output
1	Mark Sipser	M.Sipser
2	Louis Johnson	
3	Edward Davis	
4	Robert Mills	

This example requires the output substring $\text{out}_1[0..0] \equiv$ "M" to come from the input string instead of it being the constant string "M". The desired behavior in this example of preferring the substring "M" to be a non-constant string is in conflict with the desired behavior of preferring smaller substrings as constant strings in Example 1. Some hints that can help FlashFill prefer the substring expression for "M" over the constant string expression are:

- Output Token: The substring "M" of the output string is a Capital token.
- String case change: The case of the substring does not change from input.
- Regular expression Frequency: The regular expression to extract 1st capital letter occurs frequently in practice.

Example 3. An Excel user had a series of addresses in a column and wanted to extract the city names from them. The user gave the input-output example shown in the table.

	Input v_1	Output
1	243 Flyer Dr,Cambridge, MA 02145	Cambridge
2	512 Wir Ave,Los Angeles, CA 78911	
3	64 128th St,Seattle, WA 98102	
4	560 Heal St,San Mateo, CA 94129	

FlashFill learns more than 10^6 different substring expressions to extract the substring "Cambridge" from the input string "243 Flyer Drive,Cambridge, MA 02145", some of which are listed below.

- p_1: Extract the 3rd alphabet token string.
- p_2: Extract the 4th alphanumeric token string.
- p_3: Extract substring between 1st and 2nd comma tokens.
- p_4: Extract substring between 3rd capital and the 1st comma.
- p_5: Extract substring between 1st and last comma tokens.

The problem with learning the substring expression p_1 is that on the input string "512 Wright Ave, Los Angeles, CA 78911", it produces the output string "Los" that is not the desired output. On the other hand, the expression p_3 (or p_5) generates the desired output string "Los Angeles". Some features that can help FlashFill rank the expression p_3 higher are:

- Same left and right position logics: The regular expression tokens for left and right position logics for p_3 are similar (comma).
- Match Id: The match count of substring between two comma tokens is 1 as compared to 3 for the alphabet token of p_1.

3   Domain-Specific Languages (DSLs) for PBE in VSA

There have been many recent proposals for DSLs for PBE systems in the domains of string [1,7], table [23], numbers [24], and layout manipulations [11]. The key idea in designing these DSLs is to make them expressive enough to capture majority of the desired tasks, but concise enough for amenable learning from examples. Since the specification mechanism of input-output examples is inherently incomplete and ambiguous, there are typically a huge number of expressions in these expressive languages that conform to the provided examples. These large number of consistent expressions are represented succinctly using VSA based data structures that allow for sharing expressions. In this section, we describe an abstract language $\mathcal{L}_a$ that captures two major kinds of expressions that allow for such sharing, namely *fixed arity* expressions and *associative* expressions. We then present the syntax and semantics of the VSA based data structure and the algorithm to efficiently compute the highest ranked expression.

$$
\begin{aligned}
\text{Expr } e &:= v \mid c \\
&\mid e_f \mid e_h \\
\text{Fixed Arity Expr } e_f &:= f(e_1, \cdots, e_n) \\
\text{Associative Expr } e_h &:= h(e_1, \cdots, e_k) \\
&\text{(a)}
\end{aligned}
\qquad
\begin{aligned}
\text{Union Expr } \tilde{e} &:= \{c_i, v_j, \cdots, \tilde{e}_f, \tilde{e}_h\} \\
\text{Join Expr } \tilde{e}_f &:= f(\tilde{e}_1, \cdots, \tilde{e}_n) \\
\text{DAG Expr } \tilde{e}_h &:= \mathcal{D}(\tilde{\eta}, \eta^s, \eta^t, W), \text{where} \\
& W : (\eta_1, \eta_2) \to \tilde{e}, |\tilde{\eta}| = k + 1 \\
&\text{(b)}
\end{aligned}
$$

Fig. 1. (a) Syntax for a general abstract language $\mathcal{L}_a$ for a VSA based PBE system, and (b) a data structure for succinctly representing a set of $\mathcal{L}_a$ expressions.

3.1   An Abstract Language $\mathcal{L}_a$ for PBE Systems

An abstract language $\mathcal{L}_a$ that captures the major kinds of expression sharing in DSLs of several VSA based PBE systems is shown in Fig. 1(a). The top-level expression e in $\mathcal{L}_a$ can either be a constant string c, a variable v, a fixed arity expression e_f, or an associative expression e_h.

Definition 1 (Fixed Arity Expression). *Let f be any constructor for n independent expressions ($n \geq 1$). We use the notation $f(e_1, \ldots, e_n)$ to denote a fixed arity expression with n arguments.*

Example 4. The position pair expression in the FlashFill language $\texttt{SubStr}(v_i, p_1, p_2)$ is a fixed arity expression that represents the left and right position logic expressions p_1 and p_2 independently. The Boolean expression predicate $(C_1 = e_t \wedge \cdots \wedge C_k = e_t)$ for a candidate key of size k in the lookup transformation language [23], and the decimal and exponential number formatting expressions $\texttt{Dec}(u, \eta_1, f)$ and $\texttt{Exp}(u, \eta_1, f, \eta_2)$ in the number transformation language [24] are also examples of fixed arity expressions with independent arguments.

Definition 2 (Associative Expression). *Let h be a binary associative constructor for independent expressions. We use the simplified notation $h(e_1, \ldots, e_k)$ to denote the associative expression $h(e_1, h(e_2, h(e_3, \ldots, h(e_{k-1}, e_k) \ldots)))$ for any $k \geq 1$ (where $h(e)$ simply denotes e).*

Example 5. The $\texttt{Concatenate}(f_1, .., f_n)$ expression in FlashFill is an an associative expression with $\texttt{Concatenate}$ as the associative constructor. The top-level select expression $e_t := \texttt{Select}(C, T, C_i = e_t)$ in the lookup transformation language [23] and the associative program $\texttt{Assoc}(F, s_0, s_1)$ in the table layout transformation language [11] are also examples of associative expressions.

Associative expressions involve applying an associative operator with input and output type T to an unbounded sequence of expressions of type T. They differ from the fixed arity expressions in two ways: (i) they have unbounded arity, and (ii) their input and output types are restricted to be the same.

$$\llbracket c \rrbracket_\sigma := c$$
$$\llbracket v \rrbracket_\sigma := \sigma(v)$$
$$\llbracket \tilde{e} \rrbracket_\sigma := \{e_j \mid e_j \in \llbracket e_i \rrbracket_\sigma, e_i \in \tilde{e}\}$$
$$\llbracket \tilde{e}_f \rrbracket_\sigma := \{f(e_1, \cdots e_n) \mid e_i \in \llbracket \tilde{e}_i \rrbracket_\sigma\}$$
$$\llbracket \tilde{e}_h \rrbracket_\sigma := \{h(e_1, \cdots, e_k) \mid (\eta_0, \cdots, \eta_k) \in \tilde{\eta},$$
$$\eta_0 = \eta^s, \eta_k = \eta^t,$$
$$e_i \in \llbracket W(\eta_{i-1}, \eta_i) \rrbracket_\sigma\}$$

(a)

$$\mathcal{R}(\{\tilde{e}_1, \cdots, \tilde{e}_n\}, \sigma) := r_u(e_1, \cdots, e_n, \sigma)$$
$$e_i = \mathcal{R}(\tilde{e}_i, \sigma)$$
$$\mathcal{R}(f(\{\tilde{e}_{11}, \cdots, \tilde{e}_{1n}\}, := r_f(e_{11}, \cdots, e_{2m}, \sigma)$$
$$\{\tilde{e}_{21}, \cdots, \tilde{e}_{2m}\}), \sigma) \quad e_{ij} = \mathcal{R}(\tilde{e}_{ij}, \sigma)$$
$$\mathcal{R}(\mathcal{D}(\tilde{\eta}, \eta^s, \eta^t, W), \sigma) := r_g(e_{12}, \cdots, e_{ij}, \cdots, \sigma)$$
$$e_{ij} = \mathcal{R}(W(\eta_i, \eta_j), \sigma)$$

(b)

Fig. 2. (a) Semantics of the VSA based data structure for $\mathcal{L}_a$ expressions, and (b) Ranking functions for efficiently identifying the top-ranked expressions.

3.2 Data Structure for Representing a Set of $\mathcal{L}_a$ Expressions

The data structure to succinctly represent a huge number of $\mathcal{L}_a$ expressions is shown in Fig. 1(b). The Union Expression $\tilde{e}$ represents a set of top-level expressions as an explicit set without any sharing. The Join Expression $\tilde{e}_f$ represents a

set of fixed arity expressions by maintaining independent sets for its arguments $e_1, \cdots, e_n$. The DAG expression $\tilde{e}_h$ represents a set of associative expressions using a DAG $\mathcal{D}$, where the edges correspond to a set of expressions $\tilde{e}$ and each path from the start node η^s to the end node η^t represents an associative expression. The semantics of the data structure is shown in Fig. 2(a).

Join Expressions (Set-based Sharing): There can often be a huge number of fixed-arity expressions that are consistent with a given example(s). Consider the input-output example pair (u, v). Suppose v_1, v_2, v_3 are values such that $v = f(v_1, v_2, v_3)$. Suppose E_1, E_2, and E_3 are sets of expressions that are respectively consistent with the input-output pairs (u, v_1), (u, v_2), and (u, v_3). Then, $f(e_1, e_2, e_3)$ is consistent with (u, v) for any $e_1 \in E_1$, $e_2 \in E_2$, and $e_3 \in E_3$. The number of such expressions is $|E_1| \times |E_2| \times |E_3|$. However, these can be succinctly represented using the data-structure $f(E_1, E_2, E_3)$, which denotes the set of expressions $\{f(e_1, e_2, e_3) \mid e_1 \in E_1, e_2 \in E_2, e_3 \in E_3\}$, using space that is proportional to $|E_1| + |E_2| + |E_3|$.

Example 6. The position pair expressions $\texttt{SubStr}(v_i, \{\tilde{p}_j\}_j, \{\tilde{p}_k\}_k)$ in FlashFill represents the set of left and right position logic expressions $\{\tilde{p}_j\}_j$ and $\{\tilde{p}_j\}_j$ independently. The generalized Boolean conditions in the select expression $\texttt{Select(C,T,B)}$ of the lookup transformation language [23] also exhibit set-based sharing. The data structure for representing a set of decimal and exponential number formatting expressions in the number transformation language $\texttt{Dec}(u, \tilde{\eta}_1, \tilde{f})$ and $\texttt{Exp}(u, \tilde{\eta}_1, \tilde{f}, \tilde{\eta}_2)$ represents integer formats $(\tilde{\eta}_1)$, fractional formats $(\tilde{f})$, and exponent formats $(\tilde{\eta}_2)$ as independent sets.
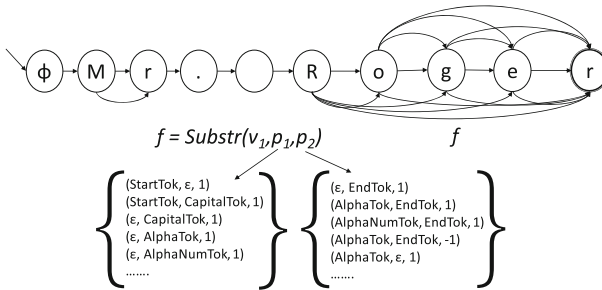

Fig. 3. The DAG data structure for representing the induced programs in Example 1.

DAG Expressions (Path-based Sharing): There can often be a huge number of associative expressions that can be consistent with a given example(s). Consider the input-output example pair (u, v). Suppose $v_1, \ldots, v_n$ are n values such that $v = h(v_1, \ldots, v_n)$ and let $e_{i,j}$ be an expression that evaluates to the value $v_{i,j} \equiv h(v_i, \ldots, v_j)$ on input u $(1 \leq i < j \leq n)$. Let $\sigma = [\sigma_0, \ldots, \sigma_m]$ be a subsequence of $[0, \ldots, n]$ such that $\sigma_0 = 0$ and $\sigma_m = n$ and e_σ be the expression

$h(e'_1, \ldots, e'_m)$, where $e'_i = e_{\sigma_{i-1},\sigma_i}$. Note that the number of such subsequences σ is exponential in n, and for any such subsequence σ, e_σ evaluates to $v_{1,n}$. Such an exponential sized set of associative expressions can be represented succinctly as a DAG whose nodes correspond to $0, \ldots, n$ and an edge between two nodes i and j corresponds to the value $v_{i,j}$ and is labeled with $e_{i,j}$. A path in the DAG from source node 0 to sink node n is some subsequence $[\sigma_1, \ldots, \sigma_m]$ of $[0, \ldots, n]$ where $\sigma_1 = 0$ and $\sigma_m = n$, and it represents the expression $F(e'_1, \ldots, e'_m) = v$, where $e'_i = e_{\sigma_{i-1},\sigma_i}$. An example DAG data structure representing all programs consistent with the input-output example in Example 1 is shown in Fig. 3. The graph data structure for generalized expression nodes for representing select expressions [23] also uses such path-based sharing for succinctly representing exponential number of expressions.

3.3   Ranking the Set of $\mathcal{L}_a$ Expressions

Given an input-output example, the PBE system learns a huge number of conforming expressions and represents them succinctly using the data structure shown in Fig. 1(b). Some of these learnt expressions are correct (desired) and others are incorrect (undesired). A user typically needs to provide more input-output examples to refine their intent until the set of expressions learnt by the system consists of only correct expressions. Our goal is to learn the desired expression from minimal number of examples (preferably 1). We formulate this problem as learning a ranking function that can rank the correct expression as the highest ranked expression.

We need to define the ranking function such that it can identify the top-ranked expression without explicitly enumerating the constituent sets. The ranking function $\mathcal{R}$ (shown in Fig. 2(b)) takes a set of $\mathcal{L}_a$ expressions and the set of input-output examples , as input, and returns the highest ranked expression. For maintaining the version-space algebra based sharing, the ranking function is defined hierarchically in terms of individual ranking functions at different levels, namely r_u, r_f, and r_h. The ranking function r_u computes the highest ranked expression from a Union Expression. It first recursively computes the top-ranked expression e_i for each of its constituent expression $\tilde{e}_i$, and then computes the highest ranked expression amongst them.

The ranking function r_f computes the highest-ranked expression from a Join expression $f(E_1, .., E_n)$. Since we assume the ranking function to be a linear weighted function of features, if all features depended on only one column (say E_i), we can easily enumerate the expressions individually for each column ($e \in E_i$) and compute the highest ranked expression $f(e_1, .., e_n)$ by selecting the highest ranked expression e_i for each individual column E_i. But often times the features depend on multiple columns, which leads to challenges in efficiently identifying the highest ranked expression. A key observation we use for computing such features is that these features typically do not depend on all concrete values of other columns, but only on a few abstract values (defined as the *abstract dimension* of the feature). For a given set of features, the columns can

be extended to a set whose size is bounded by the product of abstract dimensions of features such that a feature now depends on only one column.

The ranking function r_h efficiently computes the highest ranked expression from a DAG Expression by exploiting the notion of *associative* features. A feature g over associative expressions is said to be *associative* if there exists an associative monotonically increasing binary operator $\circ$ and a numerical feature h over expressions e_i such that $g(F(e_1, \ldots, e_n)) = g(F(e_1, \ldots, e_{n-1})) \circ h(e_n)$. The ranking function uses a dynamic programming algorithm similar to the Dijkstra's shortest path algorithm for computing the highest-ranked expression, where each DAG node maintains the highest-ranked path from the start node to itself, together with the corresponding edge feature values.

The key challenge now is to learn these ranking functions automatically at different levels. We present a supervised learning-to-rank approach for learning the ranking functions.

4   Learning the Ranking Function

Most previous approaches for *learning to rank* [2–4,12] aim at ranking all relevant documents above all non-relevant documents or ranking the most relevant document as highest. However, in our case, we want to learn a ranking function that ranks any correct program higher than all incorrect programs. We use a supervised learning approach to learn such a function, but it requires us to solve two main challenges. First, we need some labeled training data for the supervised learning. We present a technique to automatically generate labeled training data from a set of input-output examples and the corresponding set of induced programs. Second, we need to learn a ranking function based on this training data. We use a gradient descent based method to optimize a novel loss function that aims to rank *any* correct program higher than *all* incorrect programs.

4.1   Preliminaries

The training phase consists of a set of tasks $T = \{t_1, \cdots, t_n\}$. Each task t_i consists of a set of input-output examples $E^i = \{e_1^i, \cdots, e_{n(t_i)}^i\}$, where example $e_j^i = (\text{in}_j^i, \text{out}_j^i)$ denotes a pair of input (in_i) and output (out_i). We assume that for each training task t_i, sufficiently large number of input-output examples E^i are provided such that only correct programs are consistent with the examples. The task labels i on examples e_j^i are used only for assigning the training labels, and we will drop the labels to refer the examples simply as e_j for notational convenience. The complete set of input-output examples for all tasks is obtained by taking the union of the set of examples for each task $E = \{e_1, \cdots, e_{n(e)}\} = \cup_t E^t$. Let p_i denote the set of synthesized programs that are consistent with example e_i such that $p_i = \{p_i^1, \cdots, p_i^{n(i)}\}$, where $n(i)$ denotes the number of programs in the set p_i. We define positive and negative programs induced from an input-output example as follows.

Definition 3 (Positive and Negative Programs). *A program $p \in p_j$ is said to be a positive (or correct) program if it belongs to the set intersection of the set of programs for all examples of task t_i, i.e. $p \in p_1 \cap p_2 \cap \cdots \cap p_{n(t_i)}$. Otherwise, the program $p \in p_j$ is said to be a negative (or incorrect) program i.e. $p \notin p_1 \cap p_2 \cap \cdots \cap p_{n(t_i)}$.*

4.2   Automated Training Data Generation

We now present a technique to automatically generate labeled training data from the training tasks specified using input-output examples. Consider a training task t_i consisting of the input-output examples $E^i = \{(e_1, \cdots, e_{n(t_i)}\}$ and let p_j be the set of programs synthesized by the synthesis algorithm that are consistent with the input-output example e_j. For a task t_i, we construct the set of all positive programs by computing the set $p_1 \cap p_2 \cap \cdots \cap p_{n(t_i)}$. We compute the set of all negative programs by computing the set $\{p_k \setminus (p_1 \cap p_2 \cap \cdots \cap p_{n(t_i)}) \mid 1 \leq k \leq n(t_i)\}$. The version-space algebra based representation allows us to construct these sets efficiently by performing intersection and difference operations over corresponding shared expressions.

We associate a set of programs $p_i = \{p_i^1, \cdots, p_i^{n(i)}\}$ for an example e_i with a corresponding set of labels $y_i = \{y_i^1, \cdots, y_i^{n(i)}\}$, where label y_i^j denotes the label for program p_i^j. The labels y_i^j take binary values such that the value $y_i^j = 1$ denotes that the program p_i^j is a positive program for the task, whereas the label value 0 denotes that program p_i^j is a negative program for the task.

4.3   Gradient Descent Based Learning Algorithm

From the training data generation phase, we obtain a set of programs p_i associated with labels y_i for each input-output example e_i of a task. Our goal now is to learn a ranking function that can rank a positive program higher than all negative programs for each example of the task. We present a brief overview of our gradient descent based method to learn the ranking function for predicting a correct program by optimizing a novel loss function.

We compute a feature vector $x_i^j = \phi(e_i, p_i^j)$ for each example-program pair $(e_i, p_i^j), e_i \in E, p_i^j \in p_i$. For each example e_i, a training instance (x_i, y_i) is added to the training set, where $x_i = \{x_i^1, \cdots, x_i^{n(i)}\}$ denotes the list of feature vectors and $y_i = \{y_i^1, \cdots, y_i^{n(i)}\}$ denotes their corresponding labels. The goal now is to learn a ranking function f that computes the ranking score $z_i = (f(x_i^1), \cdots, f(x_i^{n(i)}))$ for each example such that a positive program is ranked as highest.

This problem formulation is similar to the problem formulation of listwise approaches for learning-to-rank [2,25]. The main difference comes from the fact that while previous listwise approaches aim to rank most documents in accordance with their training scores or rank the most relevant document as highest, our approach aims to rank any one positive program higher than all negative

$$L(E) = \sum_{i=1}^{n(e)} L(y_i, z_i) = \sum_{i=1}^{n(e)} \mathtt{sign}(\mathtt{Max}(\{f(x_i^j) \mid y_i^j = 0\}) - \mathtt{Max}(\{f(x_i^k) \mid y_i^k = 1\}))$$

$$(1)$$

$$L(y_i, z_i) = \mathtt{tanh}(c_1 \times (\frac{1}{c_2} \times \mathtt{log}(\sum_{y_i^j=0} e^{c_2 \times f(x_i^j)}) - \frac{1}{c_2} \times \mathtt{log}(\sum_{y_i^k=1} e^{c_2 \times f(x_i^k)})))$$

$$(2)$$

programs. Therefore, our loss function counts the number of examples where a negative program is ranked higher than all positive programs, as shown in Eq. 1. For each example, the loss function compares the maximum rank of a negative program ($\mathtt{Max}(\{f(x_i^j) \mid y_i^j = 0\})$) with the maximum rank of a positive program ($\mathtt{Max}(\{f(x_i^k) \mid y_i^k = 1\})$), and adds 1 to the loss function if a negative program is ranked highest (and subtracts 1 otherwise).

The presence of $\mathtt{sign}$ and $\mathtt{Max}$ functions in the loss function in Eq. 1 makes the function non-continuous. The non-continuity of the loss function makes it u nsuitable for gradient descent based optimization as the gradient of the function can not be computed. We, therefore, perform smooth approximations of the $\mathtt{sign}$ and $\mathtt{Max}$ functions using the hyperbolic $\mathtt{tanh}$ function and softmax function respectively (with scaling constants c_1 and c_2) to obtain a continuous and differentiable loss function in Eq. 2.

We assume the desired ranking function $f(x_i^j) = \boldsymbol{w} \cdot x_i^j$ to be a linear function over the features. Let there be m features in the feature vector $x_i^j = \{g_1, \cdots, g_m\}$ such that $f(x_i^j) = w_0 + w_1 g_1 + \cdots + w_m g_m$. We use the gradient descent algorithm to the learn the weights w_i of the ranking function that minimizes the loss function from Eq. 2. Although our loss function is differentiable, it is not convex, and therefore the algorithm only achieves a local minima. We need to restart the gradient descent algorithm from multiple random initializations to avoid getting stuck in non-desirable local minimas.

5   Case Study: FlashFill

We instantiate our ranking method for the FlashFill synthesis algorithm [7]. We chose FlashFill because of the availability of several real-world benchmarks. FlashFill uses a version-space algebra based data-structure shown to succinctly represent a huge set of programs. The expressions in FlashFill are shared at three different levels: (i) set-based sharing of position pair expressions at the lowest level, (ii) union expressions for atomic expressions on the DAG edges,

and (iii) path-based sharing of concatenate expressions at the top level. We describe efficient features for expressions at each of the levels.

5.1 Efficient Expression Features

Position Pair Expression Features: The binary position pair expressions take two position logic expressions as arguments. The features used for ranking the position pair expressions are shown in Fig. 4(a) together with their low abstract-dimensions. These features include frequency-based features denoting frequencies of: token sequences of left and right position logic expression arguments (g_1, g_2, g_7, g_8), occurrence Id and the position logics (g_3, g_4, g_9, g_{10}),and length of token sequences of position logics (g_5, g_6, g_{11}, g_{12}). In addition to frequency-based features, there are also Boolean features that include whether the right token sequence of left position logic is equal to the left token sequence of the right position logic (g_{13}), the right token sequence (resp. left) of left position logic and left token sequence (resp. right) of right position logic are empty (g_{14}, g_{15}).

Fixed Arity Feature	Abs. Dim.
$g_1 : \nu(r_1^l), g_2 : \nu(r_2^l)$	1
$g_3 : \nu(c^l), g_4 : \nu((r_1^l, r_2^l))$	1
$g_5 : \nu(|r_1^l|), g_6 : \nu(|r_2^l|)$	1
$g_7 : \nu(r_1^r), g_8 : \nu(r_2^r)$	1
$g_9 : \nu(c^r), g_{10} : \nu((r_1^r, r_2^r))$	1
$g_{11} : \nu(|r_1^r|), g_{12} : \nu(|r_2^r|)$	1
$g_{13} : r_2^l = r_1^r$	$|\tilde{p}_k|$
$g_{14} : r_2^l = \epsilon \wedge r_1^r = \epsilon$	2
$g_{15} : r_1^l = \epsilon \wedge r_2^r = \epsilon$	2

(a)

Associative Feature	Binary Operator $\circ$	Numerical Feature h
g_1 : NumArgs	$+$	c(1)
g_2 : SumWeights	$+$	weight
g_3 : ProdWeights	$\times$	weight
g_4 : MaxWeight	Max	weight
g_5 : MinWeight	Min	weight

(b)

Fig. 4. (a) The set of features for ranking position pair expression $\texttt{SubStr}(v_i, \{\tilde{p}_j\}_j, \{\tilde{p}_k\}_k)$, where $\tilde{p}_j = \texttt{Pos}(r_1^l, r_2^l, c^l), \tilde{p}_k = \texttt{Pos}(r_1^r, r_2^r, c^r)$. (b) The set of associative features for ranking a set of $\texttt{Concatenate}(f_1, .., f_n)$ expressions.

Atomic Expression Features: An atomic expression corresponds to a substring of the output string, which can come from several positions in the input string in addition to being a constant string. This leads to multiple atomic expression edges between any two nodes of the DAG, which are represented explicitly using a Union expression. The features for ranking these expressions are: whether the left and right positions of output (input resp.) substring matches a token (g_1, g_2, g_3, g_4), expression is a constant string or a position pair (g_5, g_6), there is a case change (g_7), absolute and relative lengths of the substring as compared to input and output strings (g_8, g_9, g_{10}),the left and right expressions of the output

substring are constant expressions or not (g_{11}, g_{12}), and the rank of position pair expression obtained from the previous level (g_{13}).

Concatenate Expression Features: At the top-level of DAG, we use associative features to compute the ranking of paths. The set of associative features together with their corresponding binary operator and numerical feature are shown in Fig. 4(b). These features include number of arguments in the Concatenate expression (g_1), the sum of weights of edges on the path (g_2), the product of weights of edges on the path (g_3), and the maximum (g_4) and minimum (g_5) weights of an edge on the path.

5.2   Experimental Evaluation

We now present the evaluation of our ranking scheme for FlashFill on a set of 175 benchmark tasks obtained from Excel product team and help forums. We evaluate our algorithm on three different train-test partition strategies, namely 20–80, 30–70 and 40–60. For each partition strategy, we randomly assign the corresponding number of benchmarks to the training and test set. For each benchmark problem, we provide 5 input-output examples. The experiments were performed on an Intel Core i7 3.20 GHz CPU with 32 GB RAM.

Training Phase: We run the gradient descent algorithm 1000 times with different random values for initialization of weights, while also varying the value of the learning rate α from 10^{-5} to 10^5 (in increments of multiples of 10). We learn the weights for the ranking functions for the initialization and α values for which best ranking performance is achieved on the training set.

(a)   (b)
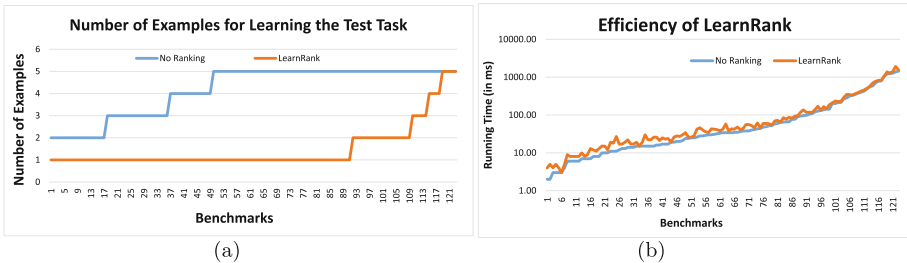
Fig. 5. Comparison of LearnRank with the Baseline scheme for a random 30–70 partition on (a) number of examples required for learning and (b) running time.

Test Phase: We compare the following two ranking schemes on the basis of number of input-output examples required to learn the desired task.

- **Baseline**: The manual ranking algorithm that chooses smallest and simplest program [7]. The algorithm prefers lesser number of arguments for the concatenate expressions, prefers simpler token expressions (such as Alphabets over AlphaNumeric), and ranks regular expression based position expression higher than constant position expressions.

- **LearnRank**: Our ranking scheme that uses the gradient descent algorithm to learn the ranking functions for position pair, atomic, and concatenate expressions in DAG.

Comparison with Baseline: The average number of input-output examples required to learn a test task for 10 runs of different train-test partitions is shown in the table. The LearnRank scheme performs much better than Baseline in terms of average number of examples required to learn the desired task (1.49 vs 4.17). For

Train-Test	Average Examples	
Partition	Baseline	LearnRank
20–80	4.19	**1.52** ± 0.07
30–70	4.17	**1.49** ± 0.06
40–60	4.18	**1.44** ± 0.07

a random 30–70 partition run, the number of input-output examples required to learn the 123 test benchmark tasks under the two ranking schemes is shown in Fig. 5(a). The LearnRank scheme learns the desired task from just 1 example for 91 tasks (74 %) as compared to 0 for Baseline, and from at most 2 examples for 110 tasks (89 %), as compared to only 18 tasks (14 %) for Baseline. Moreover, Baseline is not able to learn any program for 72 benchmarks (needing all 5 examples) as compared to 4 such benchmarks for LearnRank.

Efficiency of LearnRank: For evaluating the overhead of LearnRank scheme, we compare the running times of FlashFill with the Baseline ranking and Flash-Fill augmented with the LearnRank scheme over the same number of input-output examples for each test task. The running times of the two FlashFill versions is shown in Fig. 5(b). We observe that the overhead of LearnRank is small. The average overhead of LearnRank over Baseline is about 20 milliseconds (ms) per benchmark task whereas the median overhead is about 8 ms. This translates to an average overhead of about 29 % and a median overhead of 25 % in running times as compared to Baseline.

6 Related Work

In this section, we describe several work related to our technique which can be broadly divided into two areas: ranking techniques for program synthesis and machine learning for program synthesis.

Ranking in Program Synthesis: There have been several related work on using a manual ranking function for ranking of synthesized programs (or expressions). Gvero et al. [10] use weights to rank the expressions for efficient synthesis of likely program expressions of a given type at a given program point. These weights depend on the lexical nesting structure of declarations and also on the statistical information about the usage of declarations in a code corpus. PROSPECTOR [16] synthesizes jungloid code fragments (chain of objects and method calls from type τ_{in} to type τ_{out}) by ranking jungloids using the primary criterion of length, and secondary criteria of number of crossed package boundaries and generality of output type. Perelman et al. [20] synthesize hole values in

partial expressions for code completion by ranking potential completed expressions based on features such as class hierarchy of method parameters, depth of sub-expressions, in-scope static methods, and similar names. PRIME [18] uses relaxed inclusion matching to search for API-usage from a large collection of code corpuses, and ranks the results using the frequency of similar snippets. The SEMFIX tool [19] uses a manual characterization of components in different complexity levels for synthesizing simpler expression repairs. Our ranking scheme also uses some of these features, but we learn the ranking function automatically using machine learning unlike these techniques which need manual definition and parameter tuning for the ranking function.

SLANG [22] uses the regularities found in sequences of method invocations from large code repositories to synthesize likely method invocation sequences for code completion. It uses alias and history analysis to extract precise sequences of method invocations during the training phase, and then trains a statistical language model on the extracted data. CodeHint [5] is an interactive and dynamic code synthesis system that also employs a probabilistic model learnt over ten million lines of code to guide and prune the search space. The main difference in our technique is that it is based on a VSA based representation where it is possible to compute all conforming programs.

Machine Learning for Programming by Example: A recent work by Menon et al. [17] uses machine learning to bias the search for finding a composition of a given set of typed operators based on clues obtained from the examples. Raychev et al. [21] use A^* search based on a heuristic function of length of current refactoring sequence and estimated distance from target tree for efficient learning of software refactorings from few user edits. On the other hand, we use machine learning to identify an intended program from a given set of programs that are consistent with a given set of examples. Our technique is applicable to domains where it is possible to compute the set of all programs that are consistent with a given set of examples [8,9]. SMARTedit [14] is a PBD (Programming By Demonstration) text-editing system where a user presents demonstration(s) of the text-editing task and the system tries to generalize the demonstration(s) to a macro by extending the notion of version-spaces to model plausible macro hypotheses. The macro language of SMARTedit is not as expressive as FlashFill's, and furthermore the task demonstrations in SMARTedit reduce a lot of ambiguity in the hypothesis space. Liang et al. [15] introduce hierarchical Bayesian prior in a multi-task setting that allows sharing of statistical strength across tasks. Our underlying language and representation of string manipulation programs is different from the combinatory logic based representation used by Liang et al., which requires us to use a different learning approach.

7   Conclusion

Learning programs from few examples is an important problem to make PBE systems usable. In this paper, we presented a general approach for efficiently predicting a correct program from a large number of programs induced by few

examples. Our solution of using gradient descent based algorithm for learning the ranking function for VSA representations is at the intersection of machine learning and formal methods. We show the efficacy of our ranking technique for the FlashFill system. This machine-learning based ranking technique played a pivotal role in making FlashFill successful and usable for millions of Excel users.

References

1. Flash Fill (Microsoft Excel 2013 feature). http://research.microsoft.com/users/sumitg/flashfill.html
2. Cao, Z., Qin, T., Liu, T.-Y., Tsai, M.-F., Li, H.: Learning to rank: from pairwise approach to listwise approach. In: ICML (2007)
3. Cossock, D., Zhang, T.: Subset ranking using regression. In: Lugosi, G., Simon, H.U. (eds.) COLT 2006. LNCS (LNAI), vol. 4005, pp. 605–619. Springer, Heidelberg (2006)
4. Freund, Y., Iyer, R., Schapire, R.E., Singer, Y.: An efficient boosting algorithm for combining preferences. J. Mach. Learn. Res. **4**, 933–969 (2003)
5. Galenson, J., Reames, P., Bodík, R., Hartmann, B., Sen, K.: Codehint: dynamic and interactive synthesis of code snippets. In: ICSE, pp. 653–663 (2014)
6. Goldman, S.A., Kearns, M.J.: On the complexity of teaching. J. Comput. Syst. Sci. **50**, 303–314 (1992)
7. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: POPL (2011)
8. Gulwani, S.: Synthesis from examples: interaction models and algorithms. In: 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (2012)
9. Gulwani, S., Harris, W., Singh, R.: Spreadsheet data manipulation using examples. Commun. ACM **55**(8), 97–105 (2012)
10. Gvero, T., Kuncak, V., Kuraj, I., Piskac, R.: Complete completion using types and weights. In: PLDI, pp. 27–38 (2013)
11. Harris, W.R., Gulwani, S.: Spreadsheet table transformations from examples. In: PLDI (2011)
12. Herbrich, R., Graepel, T., Obermayer, K.: Large margin rank boundaries for ordinal regression. In: Smola, A. J., Bartlett, P. L., Scholkopf, B., Schuur-mans, D. (eds.) Advances in Neural Information Processing Systems, pp. 115–132 (1999)
13. Jha, S., Gulwani, S., Seshia, S., Tiwari, A.: Oracle-guided component-based program synthesis. In: ICSE (2010)
14. Lau, T., Wolfman, S., Domingos, P., Weld, D.: Programming by demonstration using version space algebra. Mach. Learn. **53**(1–2), 111–156 (2003)
15. Liang, P., Jordan, M.I., Klein, D.: Learning programs: a hierarchical bayesian approach. In: ICML (2010)
16. Mandelin, D., Xu, L., Bodík, R., Kimelman, D.: Jungloid mining: helping to navigate the api jungle. In: PLDI, pp. 48–61 (2005)
17. Menon, A., Tamuz, O., Gulwani, S., Lampson, B., Kalai, A.: A machine learning framework for programming by example. In: ICML (2013)
18. Mishne, A., Shoham, S., Yahav, E.: Typestate-based semantic code search over partial programs. In: OOPSLA, pp. 997–1016 (2012)
19. Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: Semfix: program repair via semantic analysis. In: ICSE (2013)

20. Perelman, D., Gulwani, S., Ball, T., Grossman, D.: Type-directed completion of partial expressions. In: PLDI, pp. 275–286 (2012)
21. Raychev, V., Schäfer, M., Sridharan, M., Vechev, M.T.: Refactoring with synthesis. In: OOPSLA, pp. 339–354 (2013)
22. Raychev, V., Vechev, M.T., Yahav, E.: Code completion with statistical language models. In: PLDI (2014)
23. Singh, R., Gulwani, S.: Learning semantic string transformations from examples. PVLDB **5**(8), 740–751 (2012)
24. Singh, R., Gulwani, S.: Synthesizing number transformations from input-output examples. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 634–651. Springer, Heidelberg (2012)
25. Xia, F., Liu, T.-Y., Wang, J., Zhang, W., Li, H.: Listwise approach to learning to rank: theory and algorithm. In: ICML (2008)