Maximal Multi-layer Specification Synthesis*

Yanju Chen
University of California Santa Barbara
Santa Barbara, California, USA
yanju@cs.ucsb.edu

Ruben Martins
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
rubenm@cs.cmu.edu

Yu Feng
University of California Santa Barbara
Santa Barbara, California, USA
yufeng@cs.ucsb.edu

ABSTRACT

There has been a significant interest in applying programming-by-example to automate repetitive and tedious tasks. However, due to the *incomplete nature* of input-output examples, a synthesizer may generate programs that pass the examples but do not match the user intent. In this paper, we propose Mars, a novel synthesis framework that takes as input a *multi-layer specification* composed by input-output examples, textual description, and partial code snippets that capture the user intent. To accurately capture the user intent from the noisy and ambiguous description, we propose a hybrid model that combines the power of an LSTM-based sequence-to-sequence model with the apriori algorithm for mining association rules through unsupervised learning. We reduce the problem of solving a multi-layer specification synthesis to a Max-SMT problem, where hard constraints encode well-typed concrete programs and soft constraints encode the user intent learned by the hybrid model. We instantiate our hybrid model to the data wrangling domain and compare its performance against Morpheus, a state-of-the-art synthesizer for data wrangling tasks. Our experiments demonstrate that our approach outperforms Morpheus in terms of running time and solved benchmarks. For challenging benchmarks, our approach can suggest candidates with rankings that are an order of magnitude better than Morpheus which leads to running times that are 15x faster than Morpheus.

CCS CONCEPTS

• **Software and its engineering → Programming by example**; **Automatic programming**.

KEYWORDS

program synthesis, machine learning, neural networks, Max-SMT

ACM Reference Format:
Yanju Chen, Ruben Martins, and Yu Feng. 2019. Maximal Multi-layer Specification Synthesis. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338951

1 INTRODUCTION

In today's data-centric world, data analytics has become one of the key elements in our daily life, including science, politics, business, and international relations. On the other hand, due to the messy nature of data in different application domains, data scientists spend close to 80% [8] of their time performing data wrangling tasks, which are considered to be the "janitor work" of data science.

To mitigate this problem, in recent years, there has been significant interest in end-user program synthesis for data science, in which the goal is to automate tedious data analytics tasks from informal specifications, such as input-output examples [13, 16] or natural language [27, 32]. For instance, programming-by-example (PBE) has been used to automate tedious tasks such as string manipulations in Excel [16], data wrangling tasks on tabular and hierarchical data [13, 31], and SQL queries [30]. Despite significant progress in PBE systems, expressing the user intent still remains a major challenge. As a result, due to the *incomplete nature* of input-output examples, a synthesizer may generate programs that pass the examples but do not match the user intent. In that case, the user has to provide additional examples to refine the results generated by the synthesizer, which imposes a huge burden to the end-user as it is tricky to figure out the root cause of the wrong candidates [22] and come up with new examples to refine the output of the synthesizer.

To address the above limitation, this paper aims to design a synthesis framework that *accurately* captures the user intent. By looking at hundreds of relevant data analytics questions from StackOverflow, we observe that an end-user typically describes her problem in *a combination of* input-output examples, natural language description, partial code snippet, etc. To give readers our insight, consider an example from StackOverflow in Figure 1. Here, the user has an input table and wants to transform it into an output table with a different shape. As shown in Figure 1, the correct solution (on the right) requires merging two column (i.e., `unite`), aggregating (i.e., `group_by, summarise`) the sum of another column, and finally pivoting (`spread`) the returning table. To solve this benchmark, it takes Morpheus [13], the state-of-the-art synthesizer for data wrangling tasks, around five minutes. Moreover, if the program found by Morpheus does not match the user intent, she has to refine the input-output examples and rerun the synthesizer.

In a lot of cases, the information provided by the end-user typically goes beyond input-output examples. In most helper forums (e.g., StackOverflow), we observed that people usually describe problems in the combination of natural language and input-output examples. For instance, looking at the example in Figure 1, the user not only provides input-output examples, but also indicates a rough "sketch" of the solution through natural language. For instance, the "reshape" and "count" keywords indicate that the solution should use library functions that perform pivoting (i.e. `spread` or `gather`) and aggregation (i.e., `group_by + summarise`), respectively. Other

*This work was sponsored by the National Science Foundation under agreement number of 1908494 and 1762363, and by a CMU-Portugal grant under agreement number of CMU/AIR/0022/2017.

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338951

keywords such as "total found" suggest that `sum` should be used together with `summarise`, and the keyword "Sp_B_pos" that the function call `unite` should be used. If we use arrows to visualize the connection between text description and function calls from data-wrangling libraries, we can observe a strong connection between the user intent and the solution.

However, real-world textual information is inherently noisy and ambiguous. As a result, it is very challenging to derive the right mapping from the textual information to their corresponding function calls. Second, even if we have the right mapping, it is still unclear how to integrate this information into most existing PBE systems [5, 16, 30, 31, 33], which typically rely on their efficient search algorithms by leveraging the syntax or semantics of the input-output examples.

In this paper, we propose Mars, a novel synthesis framework that takes as input a *multi-layer specification* that appears in a large class of applications. Here a *multi-layer specification* is composed of input-output examples, textual description, and partial code snippets that express the user intent. To solve a multi-layer specification synthesis (MSS) problem, Mars encodes input-output examples as *hard constraints* which have to be satisfied, and denotes additional preferences (e.g., textual description, partial code snippet, etc) as *soft constraints* which are *preferably* satisfiable. After that, the MSS problem is reduced to the *maximum satisfiability modulo theories* (Max-SMT) problem which can be efficiently solved by an off-the-shelf SMT solver [6, 9]. The Max-SMT encoding of the MSS problem aims to satisfy the input-output constraints and maximize the user intent that is obtained from natural language, partial code snippet, and intermediate results.

To accurately capture the user intent from the noisy and ambiguous description, we propose a hybrid neural architecture that combines the power of an LSTM-based sequence-to-sequence(i.e., *seq2seq*) [29] model and the *apriori algorithm* [2] for mining association rules. In particular, our *seq2seq* model encodes the probability of a *symbolic program* (i.e., a program of which constants are unknown.) given its corresponding textual description. However, like other deep learning applications, the performance of a *seq2seq* model heavily relies on the quality and quantity of the training data. Therefore, as shown in Section 7, for benchmarks of which solutions are complicated and rarely appear in the training set, our *seq2seq* model may not suggest the right candidates. To mitigate this problem, we leverage the *apriori algorithm* for mining the extra hidden information that cannot be covered by the *seq2seq* model. Intuitively, through unsupervised learning, the *apriori algorithm* is used to mine association rules that indicate the hidden connections between words and individual functions. After that, we use the association rules for refining the original rankings of the *seq2seq* model.

To evaluate the effectiveness of our technique, we instantiate Mars into the data wrangling domain and compare it against Morpheus [13], the state-of-the-art PBE synthesizer for data wrangling tasks. We evaluate both approaches on the 80 benchmarks from the Morpheus paper[14], and show that Mars outperforms Morpheus in terms of running time and number of benchmarks being solved. For challenging benchmarks, our approach is on average 15x faster than the Morpheus tool.

To summarize, this paper makes the following key contributions:

- We design a customized deep neural network architecture for learning the user's preference using an aligned corpus that maps the user's textual information to the desired solutions.
- We design a novel multi-layer specification that allows the end-user to specify her intent using soft and hard constraints.
- We propose a Max-SMT based synthesis framework that takes as input a multi-layer specification and enumerates solutions that are close to the user's intent. Our framework is parameterized with the underlying neural networks and the DSL, which can be easily instantiated to different domains.
- We integrate Mars's hybrid model into the Morpheus tool and empirically evaluate our approach in the data wrangling domain by showing that Mars outperforms the state of the art in running time and number of benchmarks solved.

2 OVERVIEW

In this section, we give an overview of our approach with the aid of the motivating example in Figure 1. Specifically, as shown in Figure 2, we use a simplified domain-specific language (DSL) based on `dplyr` and `tidyr`, which are two popular libraries for data wrangling tasks in *R*.

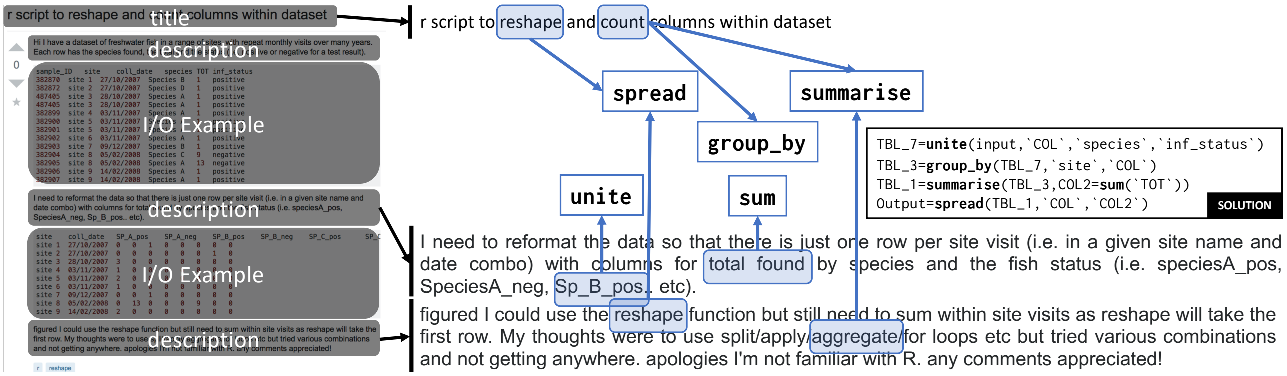
In this example, the user wants to perform a complex data wrangling task which requires concatenating two columns (i.e., `unite`), aggregation (i.e., `summarise`), and table pivoting (i.e., `spread`). We now explain the key ideas that enable Mars to solve this complex problem. We use abstract syntax trees (AST) to represent programs. For example, Figure 3 shows an AST that represents a *symbolic program* where some of the nodes are still unknown. A symbolic program can be instantiated in many ways and can generate several thousand concrete programs. For instance, the concrete program represented in Figure 4 corresponds to the following assignment:

$$\{N_1 \mapsto \texttt{select},\ N_2 \mapsto \texttt{gather}, N_3 \mapsto \texttt{[1,2]}, \\ N_4 \mapsto x_0, N_5 \mapsto \texttt{[1,3]}\}$$

This approach, while being general, has several drawbacks. First, since input-output examples are imprecise specifications, a synthesizer may generate a candidate that does not match the *user intent*, which requires the user to provide additional examples to refine the result [16, 30]. Second, given a specific task, there can be many candidates satisfying the input-output examples but only a few of them match the user intent. In this case, a synthesizer typically enumerates solutions according to some heuristic, such as the size of AST [15], or keywords provided by the user [30]. None of the previous work proposes a systematic solution for unifying the user intent from different sources.

Mars takes a different step by proposing a *multi-layer specification* that combines input-output examples with additional hints from the user. For instance, looking at the StackOverflow example in Figure 1, in addition to the input-output tables, the user also provides extra hints using natural language and intermediate results. Specifically, the word "reshape" in the title indicates that the solution should use either `spread` or `gather`, and "count" suggests the occurrence of aggregate functions(i.e.,`summarise, group_by`).

To incorporate the additional information, we propose a novel *hybrid neural architecture* by leveraging the advantages of a *seq2seq* [29] model and the *apriori algorithm* for learning association rules [1].



r script to reshape and count columns within dataset

```
TBL_7=unite(input,`COL`,`species`,`inf_status`)
TBL_3=group_by(TBL_7,`site`,`COL`)
TBL_1=summarise(TBL_3,COL2=sum(`TOT`))
Output=spread(TBL_1,`COL`,`COL2`)
```

I need to reformat the data so that there is just one row per site visit (i.e. in a given site name and date combo) with columns for total found by species and the fish status (i.e. speciesA_pos, SpeciesA_neg, Sp_B_pos.. etc).

figured I could use the reshape function but still need to sum within site visits as reshape will take the first row. My thoughts were to use split/apply/aggregate/for loops etc but tried various combinations and not getting anywhere. apologies I'm not familiar with R. any comments appreciated!

Figure 1: A motivating example from StackOverflow.[1]

$$
\begin{array}{rl}
T \rightarrow & x_i \mid \texttt{spread(T, COL,COL)} \mid \texttt{unite(T, COL, COL)} \\
 & \texttt{group\_by(T, LIST)} \mid \texttt{summarise(T, AG, COL)} \\
 & \texttt{gather(T, LIST)} \mid \texttt{select(T, LIST)} \\
\texttt{LIST} \rightarrow & \texttt{[1] | [1,2] | ... | [4,5]} \\
\texttt{COL} \rightarrow & \texttt{0 | ... | 10} \\
\texttt{AG} \rightarrow & \texttt{sum | mean | max | min}
\end{array}
$$

Figure 2: The grammar of a DSL for data wrangling tasks in `dplyr` and `tidyr`.

In particular, the *seq2seq* model takes as input the text description and returns the most likely symbolic program according to a statistical model trained from a corpus. For the example in Figure 1, our seq2seq model suggests some of the following candidates:

$$
\begin{array}{r}
\{\texttt{mutate,group\_by,summarise,spread}\}(92) \\
\{\texttt{group\_by,summarise,mutate,select}\}(91) \\
\ldots \\
\{\texttt{unite,group\_by,summarise,spread}\}(79) \\
\ldots
\end{array}
$$

Each item in the list is a pair $(\overline{\mathcal{P}}, w_i)$ where $\overline{\mathcal{P}}$ represents a symbolic program that we learn from the data, and w_i denotes the likelihood of being part of the solution. By leveraging the additional description from the user, the *seq2seq* model is able to suggest candidates that are close to the user intent. However, due to the size and quality of the training data, for complex solutions which *rarely appear* in the corpus, the *seq2seq* model is unlikely to suggest the correct symbolic program. As a result, a synthesizer may still spend a significant amount of time enumerating wrong candidates. For instance, by following the ranking generated from the *seq2seq* model, a synthesizer has to explore 130 symbolic programs before finding the right candidate.

To mitigate the above limitation, we leverage the *apriori* algorithm [2] for mining association rules. Intuitively, an association rule, which is learned from a corpus of data through unsupervised learning, aims to identify the hidden connections among the keywords. For instance, given the text description in Figure 1, our algorithm is able to discover the following rule which suggests that spread has a high chance to appear in the solution:

$$\{\texttt{reshape, count}\} \Rightarrow \{\texttt{spread}\}$$

and the following rule indicates that unite should also appear in the solution:

$$\{\_, \texttt{reshape}\} \Rightarrow \{\texttt{unite}\}$$

Using our *refinement algorithm* discussed in Section 4.3, our system is able to incorporate the hints from the association rules to adjust the distribution of the *seq2seq* model. For instance, after running the refinement algorithm, the previous ranking is adjusted to:

$$
\begin{array}{r}
\ldots \\
\{\texttt{unite,group\_by,summarise,spread}\}(109) \\
\ldots \\
\{\texttt{mutate,group\_by,summarise,spread}\}(96) \\
\{\texttt{group\_by,summarise,mutate,select}\}(94)
\end{array}
$$

Observe that the score of all three candidates get increased as they are connected to association rules learned from data. The score of the correct candidate increases more as this candidate matches *more rules* than others. As a result, a synthesizer only needs to explore less than 30 symbolic programs before reaching the right one.

To incorporate the above ranking from our statistical model, Mars provides *soft constraints* in the form of $(f(s_1, \ldots, s_k), w_i)$ where f is a k-nary predicate over DSL constructs with likelihood weight w_i. For instance, the symbolic program of the correct candidate can be expressed with the following *soft constraints*:

$$
\begin{array}{r}
(\texttt{occurs(unite), 109}) \wedge (\texttt{occurs(group\_by), 109}) \wedge \\
(\texttt{occurs(summarise), 109}) \wedge (\texttt{occurs(spread), 109}) \wedge \\
(\texttt{hasChild(group\_by,unite), 109}) \wedge \\
(\texttt{hasChild(summarise,group\_by), 109}) \wedge \\
(\texttt{hasChild(spread,summarise), 109})
\end{array}
$$

Here, $\texttt{hasChild}(s_i, s_j)$ is a binary predicate which indicates that the DSL construct s_i should be the parent of s_j in the solution. Similarly, $\texttt{occurs}(s_i)$ is a unary predicate asserting that s_i should occur in the solution. Given the soft constraints generated by the hybrid model, the underlying Max-SMT solver in Mars can enumerate candidates in a way that $\sum \omega_i$ is maximized. In other words, Mars always prioritizes candidates that not only pass the input-output

[1] https://stackoverflow.com/questions/39369502

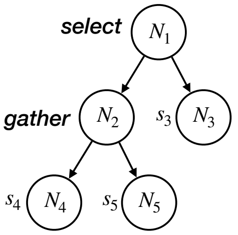


Figure 3: An example of a symbolic program.

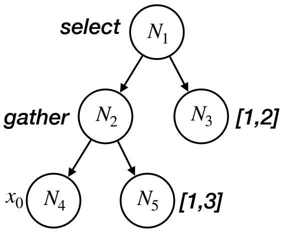


Figure 4: An example of a concrete program.

examples, but are also consistent with the user intent expressed in natural language.

3 PROBLEM FORMALIZATION

This section proposes a general setting for our synthesis problem, and formally states the definitions of our multi-layer specification and maximal synthesis.

Given a domain-specific language (DSL) described by a context-free grammar $\mathcal{G}$, our synthesis framework searches the space of all possible programs up to a given depth. A DSL is a tuple (Σ, R, S), where Σ, R, and S represent the set of symbols, productions, and the start symbol, respectively. Each symbol $\chi \in \Sigma$ corresponds to our built-in DSL construct (e.g., +, spread, gather, select etc), constants, and variables. Program inputs are expressed as symbols $x_1, \dots, x_k \in \Sigma$. Every production $p \in R$ has the form $p = (A \rightarrow \chi(A_1, \dots, A_k))$, where $\chi \in \Sigma$ is a DSL construct and $A_1, \dots, A_k \in \Sigma$ are symbols for the arguments. *Symbolic* and *concrete* programs are defined using symbols from the DSL.

Definition 1. **Symbolic Program.** *A symbolic program $\overline{\mathcal{P}}$ is an abstract syntax tree (AST) where some labels of the AST nodes are represented as* symbolic variables *yet to be determined.*

Example 1. *Figure 3 shows a symbolic program with depth of size two. Here, s_3, s_4, and s_5 denote symbolic variables which corresponds to unknown symbols. This symbolic program corresponds to* `select(gather(`*?, ?), ?), where the "?" denotes symbolic variables that still need to be determined.*

Intuitively, a symbolic program $\overline{\mathcal{P}}$ represents partial programs where some of the symbols are unknown. In Section 4, we will introduce a neural architecture for learning the most likely symbolic programs from a corpus of data.

Definition 2. **Concrete Program.** *A concrete program $\mathcal{P}$ is an AST where each node is labeled with a symbol from the DSL.*

Example 2. *Figure 4 shows an AST which corresponds to the concrete program:* `select(gather(`*x_0, [1,3]), [1,2]).*

Definition 3. **Hard Specification.** *The hard specification expresses a set of constraints that the symbolic program $\overline{\mathcal{P}}$ has to satisfy. In classical PBE systems, we often refer to the input-output examples as the hard specification. In particular, $\overline{\mathcal{P}}(\mathcal{E}_{in}) = \mathcal{E}_{out}$.*

Example 3. *In* Mars, *the hard specification is used to encode the input-output requirement from the end-user. E.g., in figure 1, the input and output tables are translated into hard constraints in* Mars.

Definition 4. **Soft Specification.** *The soft specification denotes a set of constraints that the symbolic program $\overline{\mathcal{P}}$* preferably *satisfies. In particular, each soft constraint is denoted by a pair $(pr(\chi_1, \dots, \chi_k), \omega)$ where $pr(\chi_1, \dots, \chi_k)$ is a k-ary predicate over the DSL constructs and ω represents the predicate confidence.*

Example 4. *In* Mars, *the soft specification is used to encode the user preference in the form of natural language. For instance, the unary predicate* (`occurs`$(\chi_i), \omega_i$) *encodes that a DSL construct χ_i should appear in the program with confidence ω_i. Similarly, the binary predicate* (`hasChild`$(\chi_i, \chi_j), \omega_j$) *denotes that a DSL construct χ_i should appear as the parent of χ_j in the program with confidence ω_j. Note that the weight of each predicate is automatically learned from a corpus of data.*

Now we are ready to formally state our synthesis problem.

Definition 5. **Maximal Multi-layer Specification Synthesis.** *Given specification $(\mathcal{E}, \Psi, \Sigma)$ where $\mathcal{E} = (T_{in}, T_{out})$, $\Psi = \bigcup (\chi_i, \omega_i)$, and Σ represents all symbols in the DSL, the Maximal Multi-Specification Synthesis problem is to infer a program $\mathcal{P}$ such that:*

- *$\mathcal{P}$ is a well-typed expression over symbols in Σ.*
- *$\mathcal{P}(T_{in}) = T_{out}$.*
- *$\sum \omega_i$ is maximized.*

4 NEURAL ARCHITECTURE

In this section, we propose a hybrid neural architecture for inferring the most promising symbolic programs given the user description. In particular, our architecture incorporates a *sequence-to-sequence (seq2seq)* model and the *apriori* algorithm for discovering association rules through *unsupervised learning.* While the *seq2seq* model is for estimating the initial score of a symbolic program, the association rules are further used to adjust the initial score by mining hidden information that can not be identified by the *seq2seq* model.

4.1 Sequence-to-Sequence Model

The problem of inferring the most promising symbolic programs from user description can be viewed as a translation between two different languages. In particular, our goal is to translate from natural language to symbolic programs expressed in our DSL. Inspired by the recent success in natural language processing, we apply a *seq2seq* model with *Long Short-Term Memory (LSTM)* [17] cells.

As shown in Figure 5, given a *question-solution* pair (D, S), where a *question* is a user description composed by word tokens d:

$$D = (d_1, d_2, \dots, d_n),$$

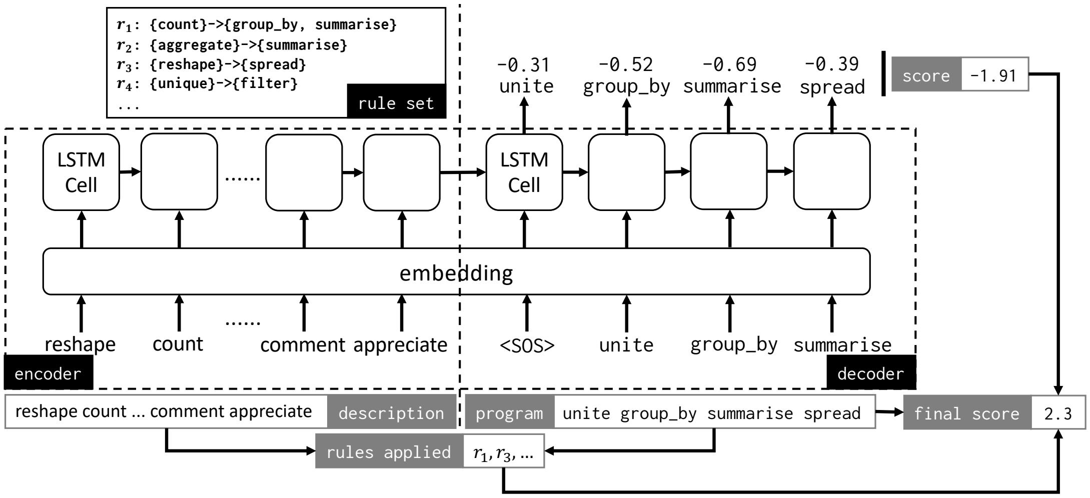


Figure 5: The hybrid neural architecture in Mars

and a *solution* is a symbolic program composed by a sequence of functions s_i: [2]

$$S = (s_1, s_2, \dots, s_m),$$

the *seq2seq* model is used to estimate the probability of $P(S|D)$, which is then given by:

$$\begin{aligned} P(S|D) &= P(s_1, s_2, \dots, s_m | d_1, d_2, \dots, d_n) \\ &= \prod_{t=1}^{m} P(s_t | v, s_1, s_2, \dots, s_{t-1}), \end{aligned}$$

where v is a fixed-dimensional vector representation of the user description generated by the *encoder*.

Internally, the *seq2seq* model is composed by two components: the *encoder* and the *decoder*. The *encoder* is an *LSTM* cell that takes as input a *question* D and generate its corresponding vector representation v. At every time step t, we feed each token d_t from the *question* to the *encoder* and compute the following functions as given by the *LSTM* mechanism:

$$\begin{aligned} z_t &= \sigma(W_z \cdot [h_{t-1}, d_t]) \\ r_t &= \sigma(W_r \cdot [h_{t-1}, d_t]) \\ \tilde{h_t} &= \tanh(W \cdot [r_t * h_{t-1}, d_t]) \\ h_t &= (1 - z_t) * h_{t-1} + z_t * \tilde{h_t}, \end{aligned}$$

where at time step t, h_t is the hidden state, W_* are network parameters that will later be learned from data, $[,]$ is the vector concatenation operation, $\cdot$ is matrix multiplication, and σ (sigmoid) and tanh are both activation functions that are given by:

$$\begin{aligned} \sigma(x) &= \frac{1}{1 + e^{-x}} \\ \tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \end{aligned}$$

The final vector representation of a *question* is given by the last hidden state: $v = h_n$.

Similar to the *encoder*, the *decoder* is also composed by an *LSTM* cell which takes as input a symbolic program represented by a sequence of functions. The output of the *decoder* is a distribution of functions given the current hidden state h_i:

$$u_i = W_u \cdot h_i + b_u,$$

where W_u and b_u are both learnable parameters, and the probability for a specific function (for example, the jth function) at time step i is estimated by:

$$\begin{aligned} P(s_{i,j}) &= P(s_{i,j} | v, s_1, s_2, ..., s_{i-1}) \\ &= \frac{\exp(u_{i,j})}{\sum_j \exp(u_{i,j})}, \end{aligned}$$

where $u_{i,j}$ is the jth element of the vector.

Finally, we use the *back propagation* method with *negative log likelihood* loss to learn the parameters of the neural network. The probability of a *symbolic program* given a *question* is computed by estimating the product of the probability at each time step. We take logarithm of every time step to prevent underflow of the final result, which gives the equation of the probability score as follows:

$$\begin{aligned} P(S|D) &= P(s_1, s_2, ..., s_m | d_1, d_2, ..., d_n) \\ &= \sum_{t=1}^{m} \log P(s_t | v, s_1, s_2, ..., s_{t-1}), \end{aligned}$$

where the most promising symbolic programs have higher scores.

4.2 Learning Association Rules

As shown later in section 7, due to the quality of the training data, our *seq2seq* model alone does not always achieve good performance. Specifically, for complex benchmarks of which solutions rarely appear in the training data, it is difficult for the *seq2seq* model to suggest the right candidates. On the other hand, even though the user cannot figure out the exact solution for her problem, she may still indicate partial information of the desired solution using some keywords or phrases. In order to discover hidden information that can not be inferred by the *seq2seq* model, we leverage the *apriori*

[2] Each symbolic program ignores all constant variables and only preserves the name of each function.

algorithm to mine association rules that will later be used to adjust the rankings from the *seq2seq* model.

As shown in figure 5, let Q be the union of all tokens that appear in the questions and all functions that appear in a *solution*:

$$Q = q_1, q_2, ..., q_c,$$

and let E be the set of all tokens in a *question-solution* pair (S, D):

$$E = \bigcup (s_i, d_j) \text{ where } s_i \in S, d_j \in D$$

An *association rule* r of a given set E is defined by:

$$r : X \Rightarrow Y,$$

where $X, Y \subseteq Q$. For example,

$$\{\text{unite}, \text{wide}\} \Rightarrow \{\texttt{spread}\}$$

indicates that if the two keywords "unite" and "wide" appear in the *question*, then the function `spread` is also appearing in the corresponding *solution*. Also, rules can apply on functions:

$$\{\texttt{filter}, \texttt{summarise}\} \Rightarrow \{\texttt{group\_by}\}$$

which means if both `filter` and `summarise` appear in the *solution*, then the function `group_by` also appears in the same *solution*.

To learn the association rules, we run the *apriori* algorithm on more than 30,000 answers[3] from Stackoverflow. Since the *apriori* algorithm is based on *unsupervised learning*, it may generate rules that are not useful. To address this issue, we further filter out the association rules of which confidence are low according to the following formulas:

$$\text{supp}(X) = \frac{|\{e \in E, X \subseteq e\}|}{|E|}$$

$$\text{conf}(X \Rightarrow Y) = \frac{\text{supp}(X \cup Y)}{\text{supp}(X)}.$$

Here, `supp` indicates the frequency of X that appears in the dataset, and `conf` represents how often the rule holds.

4.3 Score Refinement Algorithm

In this section, we describe an algorithm that refines the score of the *seq2seq* model using the association rules in section 4.2.

As shown in Algorithm 1, the key idea of our REFINEMENT procedure is to take as input a symbolic program S together with its original score c from the *seq2seq* model, and produce a new score c_r according to the association rules R discussed in section 4.2. Internally, the refined score c_r is computed based on an *accumulative boosting ratio* b that is initialized at line 4. Then for each association rule r_i, the algorithm updates the *accumulative boosting ratio* based on a weight function θ as well as a *match* function that decides whether the current rule $r_i = X \Rightarrow Y$ applies to the current symbolic program together with its description (D, S):

$$\text{match}(r, D, S) = \begin{cases} 1 & \forall e \in X \cup Y, e \in D \text{ or } e \in S \\ 0 & \text{otherwise} \end{cases}$$

Furthermore, the weight function θ is used to measure the quality of association rule r_i by taking several factors into account, including the confidence (i.e. `conf`) and support (i.e., `supp`) discussed in section 4.2, number of keywords that appear in rule r_i, and cost of the DSL construct (e.g. compared to `select`, `mutate` is more computationally intensive).

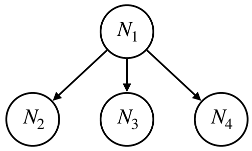


Figure 6: An example of a bounded symbolic program.

Algorithm 1 *Symbolic Program* Score Refinement Algorithm

```
1: procedure REFINEMENT($R$, $D$, $S$, $c$, $\theta$)
2:     input: association rule set $R$, question $D$, solution $S$ with its
       corresponding score $c$ and weight function $\theta$
3:     output: refined score $c_r$
4:     $b \leftarrow 0$                                ▷ accumulative boosting ratio
5:     for rule $r_i \in R$ do
6:         $b \leftarrow b + \theta(r_i) \cdot \text{match}(r_i, D, S)$
7:     $c_r \leftarrow c + b \cdot |c|$                ▷ update score
8:     return $c_r$
```

5 MAXIMAL SPECIFICATION SYNTHESIS

In this section, we describe how MARS leverages the statistical information (discussed in section 4) to enumerate programs that are close to user intent.

As we mentioned earlier, most PBE synthesizers [5, 13, 16, 30, 31, 33] perform program enumeration until they find a program that satisfies the input-output examples provided by the user. In order to perform program enumeration, we first need to represent the set of all possible programs up to a given depth. Consider a DSL $D = (\Sigma, R, S)$ where Σ_m represent DSL constructs with arity-m and m is the greatest arity between DSL constructs. A symbolic program $\overline{\mathcal{P}}$ represented by a tree of depth k where each node has exactly m children can represent all programs that use at most $k - 1$ production rules. Figure 6 shows a 3-ary tree with depth 2 that represents all programs that can be constructed using at most 1 production rule from the DSL shown in Figure 2. Note that $m = 3$ since the greatest arity between DSL constructs is 3.

EXAMPLE 5. *Assigning* $N_1 \mapsto$ *unite,* $N_2 \mapsto$ *input,* $N_3 \mapsto 1$*,* $N_4 \mapsto 2$ *corresponds to the program* "unite(input, 1, 2)" *which unites columns* 1 *and* 2 *from table input.*

Given a symbolic program $\overline{\mathcal{P}}$ and a DSL D, we encode the set of all possible concrete programs as an SMT formula φ. The Satisfiability Modulo Theories (SMT) problem is a decision problem for formulas that are composed of multiple theories. To encode symbolic programs, we use the quantifier-free fragment of the theory of Linear Integer Arithmetic (LIA). A model of φ can be mapped to a concrete program by assigning a symbol to each node in $\overline{\mathcal{P}}$.

Variables. For each node N_i, we use an integer variable with domain between 0 and r, where $r = |\Sigma|$. Assigning $N_i \mapsto k$ means that we assign to N_i the corresponding symbol. Let idx : $\Sigma \rightarrow \mathbb{N}_0$ be a mapping between a symbol and its position. Since some production rules p may have arity smaller than m, there may exist

[3] An answer towards a specific question is usually composed by some natural language description and solution code, which fits the prerequisits of association rules mining.

some children nodes N_j that are not assigned any symbols. To enforce the invariant that each node is assigned *exactly one* symbol, we introduce a special symbol p_e with index 0 that is assigned to nodes without symbols, i.e. $N_j \mapsto 0$.

Example 6. *Consider the DSL in Figure 2. idx maps each symbol to a corresponding integer that identifies its position. For example the input x_i is mapped to index 1, spread to index 2, unite to index 3, etc.*

Constraints. Let I, O correspond to all symbols that are consistent with the input and output examples, respectively. To guarantee that all models correspond to well-typed concrete programs we must enforce the following constraints.

(1) The root node N_1 of $\overline{\mathcal{P}}$ will be assigned a symbol that is consistent with the output type.

$$\bigvee_{p \in O} N_1 = \mathsf{idx}(p)$$

Example 7. *Let $O = \{x_i,\ spread,\ unite,\ group\_by,\ summarise,\ gather, select\}$. The following constraint enforces that the output type is consistent with the output example:*
$N_1 = idx(x_i) \vee N_1 = idx(spread) \vee N_1 = idx(unite) \vee idx(group\_by) \vee N_1 = idx(summarise) \vee N_1 = idx(gather) \vee N_1 = idx(select)$.

(2) Let N be the set of all nodes and Ch_{N_i} the set of children nodes of $N_i \in N$. Furthermore, let $C(p, N_i)$ be the set of production rules that are consistent with production p and can be assigned to N_i. If a production rule $p = (A \rightarrow \chi(A_1, \ldots, A_k))$ is assigned to node N_i then all m children $N_j, \ldots, N_{j+m}$ will have to be consistent with $A_1, \ldots, A_k$.

$$\bigwedge_{p \in \Sigma, N_i \in N} N_i = \mathsf{idx}(p) \implies \bigwedge_{N_j \in Ch_{N_i}} \bigvee_{p_j \in C(p, N_j)} N_j = \mathsf{idx}(p_j)$$

Example 8. *To guarantee that if production $p = unite$ is assigned to node N_1 then its children are consistent with p, we add the following constraints to φ:*
$N_1 = idx(unite) \implies \big(N_2 = idx(x_i) \vee N_2 = idx(spread) \vee N_2 = idx(unite) \vee idx(group\_by) \vee N_2 = idx(summarise) \vee N_2 = idx(gather) \vee N_2 = idx(select)\big)$.
Similar constraints are added to guarantee the consistency of N_3 and N_4 when unite is assigned to N_1.

(3) Let L the set of leaf nodes and T the set of terminal symbols. Only terminal symbols can be assigned to a leaf node.

$$\bigwedge_{N_i \in L} \bigvee_{p \in T} N_i = \mathsf{idx}(p)$$

Example 9. *Consider the leaf node N_2. To restrict the occurrence of terminals in N_2, we add the following constraints:*
$N_2 = idx(x_i) \vee N_2 = idx([1]) \vee N_2 = idx([1,2]) \vee \ldots \vee N_2 = idx([4,5]) \vee N_2 = idx(0) \vee \ldots \vee N_2 = idx(10)$.

5.1 Enumerating Maximal Programs

Enumerating models from the SMT formula φ described in Section 5 will correspond to concrete programs. However, this enumeration does not take into consideration the user intent captured by the neural network described in Section 4. To capture this information, we extend the SMT formula to a Max-SMT (Maximum Satisfiability Modulo Theories) formula. A Max-SMT formula is composed by a set of *hard* and *soft* constraints. The Max-SMT problem is to satisfy all hard constraints while maximizing the number of soft constraints that can be simultaneously satisfied. This problem can be further generalized to the weighted Max-SMT problem where each soft constraint c_i can be associated with a weight w_i. As hard constraints, we use the constraints described in Section 5 that guarantee all enumerated programs are well-typed. As soft constraints, we use the predicates occurs and hasChild encoded as follows.

(1) Let predicate (occurs(p_i), w_i) denote that a production rule p_i occurs with likelihood w_i in the final program. This predicate can be encoded into Max-SMT with the following soft constraints with weight w_i.

$$\bigwedge_{p_i \in \Lambda} \bigvee_{N_i \in N} N_i = \mathsf{idx}(p_i)$$

Example 10. *The predicate* (occurs(spread), 80) *is encoded by adding the following soft constraint to φ with weight 80:*
$N_1 = idx(spread) \vee N_2 = idx(spread) \vee N_3 = idx(spread) \vee N_4 = idx(spread)$.

(2) Let predicate *(hasChild(p_i, p_j), w_i)* denote that production p_i has production p_j as its children with likelihood w_i. This predicate is encoded as follows where all soft constraints have weight w_i.

$$\bigwedge_{p_i, p_j \in \Lambda, N_i \in N} N_i = \mathsf{idx}(p_i) \implies \bigwedge_{N_j \in Ch_{N_i}} N_j = \mathsf{idx}(p_j)$$

Example 11. *The predicate* (hasChild(summarise, group_by), 92) *is encoded by adding the following constraints to φ with weight 92:*

$$\big(N_1 = idx(summarise) \implies N_2 = idx(group\_by)\big) \wedge$$
$$\big(N_1 = idx(summarise) \implies N_3 = idx(group\_by)\big) \wedge$$
$$\big(N_1 = idx(summarise) \implies N_4 = idx(group\_by)\big)$$

Maximizing the satisfaction of these soft constraints will guarantee that we enumerate programs that are closer to the user intent. Note that even though the predicates occurs and hasChild suffice to capture the information extracted by the neural network, our approach is not limited to these predicates and can be extended by adding additional predicates (e.g., happens before).

6 IMPLEMENTATION

Data collection and preparation. We collect 20,640 pages from Stackoverflow [28] using the search keywords "tidyr" and "dplyr" (with testing benchmarks excluded), where each page contains a single *question* and multiple *solutions.* By removing duplicate contents and *questions* with no *solutions*, we obtain 16,459 *question-solution* pairs. Each *question* is pre-processed by a standard NLP pipeline that includes: stop word removal, lemmatization and tokenization, and a *solution* is represented as a sequence of DSL constructs (i.e., function names). The *question-solution* pairs are then used to train a *seq2seq* model. For the association rules mining, we extract *descriptions* from answers and their corresponding *solutions* and totally obtain 37,748 transactions as the input to the *Apriori* algorithm. To ensure the validity of our experiments, we remove all the benchmarks from the collected data.

Neural Network and Hybrid Architecture. We build a *seq2seq* neural network using the PyTorch framework [23]. The hyper-parameters (e.g., numbers of dimensions of the word/function embedding layer and LSTM hidden layer) are obtained through a simple grid search. For the *seq2seq* model in Mars, we set both the dimensions of word/function embedding layer and LSTM hidden layer to be 256, where the embedding layer maps 25,004 words and 14 functions[4] to vectors of the dimension 256. Furthermore, a single layer perceptron is connected to the hidden layer of each output time step in the decoder, mapping from a dimension of 512[5] to 14, which is used to predict the probability of each function given the previous hidden state and the current input.

As for the association rule mining, we apply the *Efficient-Apriori* [11] package to discover useful association rules that can be further applied to refine the original ranking generated by the *seq2seq* model. We then select *valid* rules according to the following criteria:

- confidence ≥ 0.9 or support ≥ 0.003.
- Each *valid* rule should have at least 1 *word* and 1 *function*. And the number of *functions* in the rules shall not exceed 2.
- Each *valid* rule should not contain any *stop* words, which builds upon the English stop words and includes additional *words* and *functions* that we consider less indicative.

By filtering out less relevant rules, we obtain 187 association rules.

Machine configuration. We train our *seq2seq* model on a machine from Google Cloud Platform with a 2.20GHz Intel Xeon CPU and an NVIDIA Tesla K80 GPU. All synthesis tasks were run on a laptop equipped with Intel Core i5 CPU and 16GB memory. Since the Morpheus tool is only available on a virtual machine [14], we used this virtual machine to run all program synthesis experiments. It took around 8 hours to train our hybrid model.

7 EVALUATION

We evaluated Mars by conducting experiments that are designed to answer the following questions:

- Q1: Do our multi-layer specification and neural architecture suggest candidates that are close to the user intent?
- Q2: What is the impact of the neural architecture in Mars on the performance of a state-of-the-art synthesizer for data wrangling tasks?
- Q3: How is the performance of Mars affected by the quality of the corpus?

7.1 Quality of Suggested Candidates

To evaluate the benefit of the multi-layer specification and neural architecture in Mars, we instantiate the tool to the data wrangling domain, where data scientists tend to spend about 80% of their time doing tedious and repetitive tasks. In particular, we use the data in section 6 to train the *n*-gram model from the Morpheus paper [13], the *seq2seq* model discussed in section 4.1, and the hybrid neural architecture described in Figure 5. Since the output of each model is a distribution of symbolic programs, we run all three models on the original benchmarks from Morpheus, which contains 80 data wrangling tasks using two popular R libraries, namely, `tidyr` and `dplyr`. In particular, the data wrangling DSL contains 60 production rules and can induce a gigantic search space of the symbolic programs, posing a challenge for state-of-the-art synthesizers. As shown in Morpheus' user study, data scientists solved on average two benchmarks in one hour. For each benchmark, we then use the *seq2seq* model and hybrid neural architecture to enumerate symbolic programs and record the ranking of the correct candidate that matches the user intent. Finally, we manually checked all solutions synthesized by MARS and made sure that they are semantically equivalent to the reference solutions. Because the *n*-gram model in Morpheus only considers programs in the posts on StackOverflow and ignore user description, it provides a global ranking shared by all benchmarks.

Results. As shown in Table 1, the average ranking and standard deviation of the *n*-gram model are 42 and 70, respectively. In other words, a synthesizer would need to explore 42 symbolic programs on average. Recall that a symbolic program may correspond to several thousand concrete programs. The standard deviation is used to quantify the stability of the model. In contrast, by incorporating the user descriptions, the *seq2seq* model achieves an average ranking of 25 and a standard deviation of 39. Finally, with the help of the association rules, the hybrid model obtains the best performance with an average ranking of 18 and a standard deviation of 26. The result shows that our hybrid model not only suggests candidates that are close to user intent (i.e., low average), and it is also more stable (i.e., low standard deviation) across different benchmarks.

Table 1: Statistics for different model rankings.

model	*n-gram*	*seq2seq*	*hybrid*
average[*]	42	25	18
std.[1]	70	39	26

[1] standard deviation.
[*] computed based on the rankings of the correct solutions.

Table 2: Counts of top-1s and top-3s in different models.

model	*n-gram*	*seq2seq*	*hybrid*
Top-1 total[*]	0	8	11
Top-3 total[*]	2	18	29

[*] computed based on the rankings of the correct solutions.

We further look into the number of top-1 and top-3 candidates that are correctly suggested by each model. As shown in Table 2, without user descriptions, the *n*-gram model fails to predict any correct candidates in top-1 and only suggests correct candidates in top-3 for two benchmarks. By leveraging user descriptions, the *seq2seq* model is able to figure out the right top-1 and top-3 candidates for 8 and 18 benchmarks, respectively. Finally, our hybrid

[4] There are 25,000 natural language words in the word vocabulary and 10 functions in the function vocabulary. Each vocabulary contains 4 special helper tokens, namely namely "<PAD>" (empty placeholder), "<SOS>" and "<EOS>" (the start and end of a sequence), "<UKN>" (out-of-vocabulary word).

[5] Since we are using separate *seq2seq* structures for *title* and *question*, the concatenation of the hidden layers from both are of a dimension of 256×2=512.

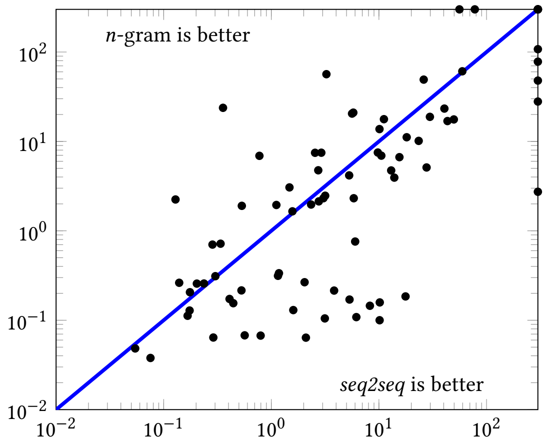


Figure 7: Comparison of run times (in seconds) between n-gram (x-axis, used in MORPHEUS) and *seq2seq* (y-axis, used in MARS) using a logarithmic scale.

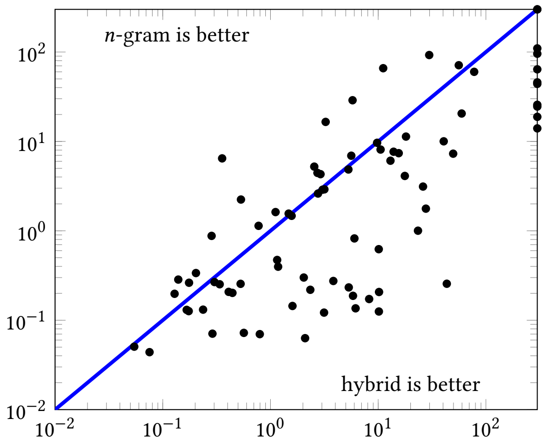


Figure 8: Comparison of run times (in seconds) between n-gram (x-axis, used in MORPHEUS) and hybrid (y-axis, used in MARS) using a logarithmic scale.

model successfully suggests top-1 and top-3 candidates for 11 and 29 benchmarks.

7.2 Effectiveness of Hybrid Neural Architecture

In this section, we further investigate the impact of a better ranking on the end-to-end performance of a synthesizer. Specifically, we integrate the previous three statistical models into MORPHEUS, a state-of-the-art synthesizer for data wrangling tasks.

Results. Figure 7 and Figure 8 show the results of running MORPHEUS on its original 80 benchmarks with three different models (namely n-gram model in original MORPHEUS and *seq2seq*/hybrid model in MARS) and a time limit of 300 seconds. In particular, each dot in the figure represents the pairwise running time of a specific benchmark under different models. As a result, the dots near the diagonal indicates that the performance of the two models is similar in those benchmarks. For instance, Figure 8 shows the comparison between the n-gram model and our hybrid model in terms of running time. Specifically, our hybrid model outperforms MORPHEUS' original n-gram model in 58 of 80 benchmarks. In the meantime, MORPHEUS times out on 11 benchmarks with the n-gram model, whereas it only times out on 2 benchmarks with our hybrid model. The performance of the *seq2seq* model is between the above two models by outperforming MORPHEUS n-gram model in 47 of 80 benchmarks and timing out on 8 benchmarks. Table 3 shows the average speedup for challenging benchmarks (i.e. > 3 library calls) with respect to the n-gram model for benchmarks that can be solved by both models. On average, the *seq2seq* model is 6x faster than the n-gram model and the hybrid model is 15x faster than the n-gram model. The result further confirms that a statistical model that accurately captures user intent tends to have a better performance in running time.

Table 3: Statistics of running time.

model	avg. speedup[1]	#timeouts*
ngram	1x	11
seq2seq	6x	8
hybrid	15x	2

[1] average speedup on challenging solved benchmarks.
* number of timeouts on all benchmarks.

Remarks. To understand the cases where our technique runs significantly faster, we manually look into some of the benchmarks. We notice that our technique performs especially well if the user states her problem in a clear way. For instance, in this post from StackOverflow, [6] although the user does not know the exact solution for her complex task, she is still able to convey the transformations using keywords (e.g., "count" and "unique") and partial code snippets. Even with these discrete signals, our hybrid model manages to guide MORPHEUS to the correct program in less than a second:

```
TBL_7=filter(p25_input1,`b` > 1)
TBL_3=unite(TBL_7,key_ab,`a`,`b`)
TBL_1=group_by(TBL_3,`key_ab`)
morpheus=summarise(TBL_1,e=n())
```

In contrast, MORPHEUS with its original n-gram model takes several minutes to find the right candidate.

7.3 Discussion

Like any other technique, our approach also has its own limitations. For instance, in Figure 8, there are still some benchmarks where n-gram performs better, we manually inspect all these cases and notice that the issue is caused by the following reasons:

Insufficient text. In this post, [7] the user only provides input-output examples but her description barely contains any useful signals that allow our hybrid model to make a good prediction.

Contextual text. In this post, [8] the user explicitly states that she does not want to use the `mutate` function:

[6] https://stackoverflow.com/questions/33549927
[7] https://stackoverflow.com/questions/26733449
[8] https://stackoverflow.com/questions/29447325

> *"... I can solve my problem using dplyr's* **mutate but** *it's a time-intensive, roundabout way to achieve my goal. ..."*

However, after tokenizing the natural description and removing all the stop words (e.g., *"but"*), our hybrid model loses the contextual information and takes mutate as the keyword.

Misleading text. In contrast to the previous example, in this post, [9] the user explicitly wants to use the mutate function:

> *"... I* **want to use mutate** *to make variable d which is mean of a,b and c. ..."*

However, since we directly adopt the DSL from Morpheus and the DSL does not support this special usage of mutate, our hybrid model proposes candidates that do not lead to the correct solution.

7.4 Threats to Validity

Quality of the corpus. Even though the hybrid neural architecture is more resilient to the limitation of the existing data set, the performance of Mars is still sensitive to the quality of the training data. To mitigate this concern, we train our statistical model using all relevant posts from StackOverflow. In the future, we also plan to leverage transfer learning to incorporate resources written in other languages (e.g., Python and Matlab).

Benchmark selection. Due to the expressiveness of the DSL, in terms of complexity, the benchmarks from Morpheus [13] may not represent the actual distribution of the questions on StackOverflow. While the comparison on the Morpheus benchmarks may not completely unveil the benefit of our hybrid neural architecture, and a representative test suite may provide a more comprehensive view, we believe our comparison is sufficient to show the strength of our technique. Furthermore, since both our neural architecture and the enumerator are designed in domain-agnostic way, we also believe our technique can generalize to other domains.

8 RELATED WORK

Program synthesis has been extensively studied in recent years. In this section, we briefly discuss prior closely-related work.

Programming by Example. Our technique is related to a line of work on programming-by-example (PBE) [5, 13, 16, 30, 31, 33]. PBE has been widely applied to different domains such as string manipulation [5, 16], data wrangling [13, 31], and SQL queries [30, 33]. Among these techniques, the Morpheus tool is directly related to the data wrangling client to which Mars is instantiated. However, unlike Morpheus that is specialized to table transformation, the techniques in Mars can be generalized to other synthesis tasks. Compared to existing PBE systems, Mars proposes a novel neural architecture that can learn user preferences from natural language.

Programming by Natural Language. Programming-by-natural-language (PBNL) is another paradigm [10, 18, 24, 27, 32] that is related to our approach. Specifically, the Sqlizer [32] tool takes input as natural language and generates its corresponding query in SQL. There are other PBNL systems that translate natural language into simple commands in smartphone [18], IFTTT scripts [24], and scripts for text editing [10, 27]. Compared to previous PBNL systems, our neural architecture can reasonably capture the user intent even in the presence of low-quality training data. Furthermore, in addition to natural language, the multi-layer specification in Mars also accepts input-output examples as hard constraints which provide a stronger correctness guarantee.

Machine Learning for Program Synthesis. The neural architecture in Mars is relevant to two major directions for applying machine learning to program synthesis. In particular, The first line of work is to directly generate programs from inputs in the form of natural language or input-output examples [20, 21], which is inspired by the *seq2seq* model in machine translation. Although we also incorporates a *seq2seq* model as part of the neural architecture, we further leverage the *apriori* algorithm for mining association rules to mitigate the quality of training data.

The second approach [19] incorporates statistical information to guide a program synthesizer. In other words, a statistical model is used to suggest the most promising candidates a synthesizer has to explore. For instance, DeepCoder [3] uses a deep neural network that can directly predict programs from input-output examples. The Morpheus tool [13] adopts an *n*-gram model for synthesizing data wrangling tasks. Similarly, the SLANG [26] tool integrates an *n*-gram model for code completion. Raychev et al. [25] extend the previous approach to obtain a statistical model that can guide a synthesizer in the presence of noisy examples. The Neo [12] synthesizer generalizes previous approaches by incorporating an arbitrary statistical model as its "decider" to guide the enumerative search. While Mars proposes a novel neural architecture to suggest the most promising candidates, it can also leverage advanced techniques from previous work, such as pruning infeasible candidates through deduction [13] and conflict-driven learning [12].

Interactive Program Synthesis. The goal of our technique is also aligned with tools in interactive program synthesis [4, 7, 22], where the goal is to iteratively refine user intent through incorporating user decision in the synthesizer loop. While our approach leverages natural language to capture the user intent, we believe the idea of interactive synthesis is complementary to our approach and can further refine the distribution of our statistical model.

9 CONCLUSION

We propose Mars, a novel synthesis framework that takes as input a *multi-layer* specification which combines input-output examples, textual description, and partial code snippets to capture the user intent. To solve a multi-layer specification synthesis (MSS) problem, Mars encodes input-output examples as *hard constraints* and denotes additional preferences (e.g., textual description, partial code snippet, etc) as *soft constraints*. The MSS problem is reduced to a Max-SMT formula which can be solved by an off-the-self solver [6, 9]. To accurately capture user intent from noisy and ambiguous descriptions, we propose a novel hybrid neural architecture that combines the power of a sequence-to-sequence model and the *apriori algorithm* for mining association rules. We instantiate our hybrid model to the data wrangling domain and compare its performance against Morpheus on its original 80 benchmarks. Our results show that our approach outperforms Morpheus and it is on average 15x faster for challenging benchmarks.

[9] https://stackoverflow.com/questions/33401788

REFERENCES

[1] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. 1993. Mining Association Rules between Sets of Items in Large Databases. In *Proc. International Conference on Management of Data*. ACM, 207–216.

[2] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast Algorithms for Mining Association Rules in Large Databases. In *Proc. International Conference on Very Large Data Bases*. ACM, 487–499.

[3] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. Deepcoder: Learning to write programs. In *Proc. International Conference on Learning Representations*. OpenReview.

[4] Shaon Barman, Rastislav Bodík, Satish Chandra, Emina Torlak, Arka Aloke Bhattacharya, and David Culler. 2015. Toward tool support for interactive synthesis. In *Proc. International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, 121–136.

[5] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin G. Zorn. 2015. FlashRelate: extracting relational data from semi-structured spreadsheets using examples. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 218–228.

[6] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. 2015. νZ - An Optimizing SMT Solver. In *Proc. Tools and Algorithms for Construction and Analysis of Systems*. Springer, 194–199.

[7] Rastislav Bodík, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. 2010. Programming with angelic nondeterminism. In *Proc. Symposium on Principles of Programming Languages*. ACM, 339–352.

[8] Tamraparni Dasu and Theodore Johnson. 2003. *Exploratory Data Mining and Data Cleaning*. John Wiley.

[9] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proc. Tools and Algorithms for Construction and Analysis of Systems*. Springer, 337–340.

[10] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. 2016. Program synthesis using natural language. In *Proc. International Conference on Software Engineering*. ACM, 345–356.

[11] efficient apriori. 2018. efficient-apriori 0.4.5. https://pypi.org/project/efficient-apriori/.

[12] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 420–435.

[13] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 422–436.

[14] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2018. Morpheus. https://utopia-group.github.io/morpheus/.

[15] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-output Examples. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 229–239.

[16] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proc. Symposium on Principles of Programming Languages*. ACM, 317–330.

[17] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[18] Vu Le, Sumit Gulwani, and Zhendong Su. 2013. SmartSynth: synthesizing smartphone automation scripts from natural language. In *Proc. International Conference on Mobile Systems, Applications, and Services*. ACM, 193–206.

[19] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. 2013. A machine learning framework for programming by example. In *Proc. International Conference on Machine Learning*. Proceedings of Machine Learning Research, 187–195.

[20] Arvind Neelakantan, Quoc V Le, Martin Abadi, Andrew McCallum, and Dario Amodei. 2017. Learning a natural language interface with neural programmer. In *Proc. International Conference on Learning Representations*. OpenReview.

[21] Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. 2016. Neural programmer: Inducing latent programs with gradient descent. In *Proc. International Conference on Learning Representations*. OpenReview.

[22] Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming not only by example. In *Proc. International Conference on Software Engineering*. ACM, 1114–1124.

[23] pytorch. 2018. The Pytorch Framework. https://pytorch.org/.

[24] Chris Quirk, Raymond J. Mooney, and Michel Galley. 2015. Language to Code: Learning Semantic Parsers for If-This-Then-That Recipes. In *Proc. Meeting of the Association for Computational Linguistics*. 878–888.

[25] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016. Learning programs from noisy data. In *Proc. Symposium on Principles of Programming Languages*. ACM, 761–774.

[26] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 419–428.

[27] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional Program Synthesis from Natural Language and Examples. In *Proc. International Joint Conference on Artificial Intelligence*. AAAI Press, 792–800.

[28] Stackoverflow. 2018. Stackoverflow. https://stackoverflow.com/.

[29] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Proc. Annual Conference on Neural Information Processing Systems*. 3104–3112.

[30] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 452–466.

[31] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing transformations on hierarchically structured data. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 508–521.

[32] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query Synthesis from Natural Language. In *Proc. International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 63:1–63:26.

[33] Sai Zhang and Yuyin Sun. 2013. Automatically synthesizing sql queries from input-output examples. In *Proc. International Conference on Automated Software Engineering*. IEEE, 224–234.