SynEva: Evaluating ML Programs by Mirror Program Synthesis

Yi Qin[†], Huiyan Wang[§], Chang Xu[*‡], Xiaoxing Ma[‡], Jian Lu[‡]
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
Department of Computer Science and Technology, Nanjing University, Nanjing, China
[†]borakirchies@163.com, [§]cocowhy1013@gmail.com, [‡]{changxu, xxm, lj}@nju.edu.cn

***Abstract*—Machine learning (ML) programs are being widely used in various human-related applications. However, their testing always remains to be a challenging problem, and one can hardly decide whether and how the existing knowledge extracted from training scenarios suit new scenarios. Existing approaches typically have restricted usages due to their assumptions on the availability of an oracle, comparable implementation, or manual inspection efforts. We solve this problem by proposing a novel program synthesis based approach, SynEva, that can systematically construct an oracle-alike mirror program for similarity measurement, and automatically compare it with the existing knowledge on new scenarios to decide how the knowledge suits the new scenarios. SynEva is lightweight and fully automated. Our experimental evaluation with real-world data sets validates SynEva's effectiveness by strong correlation and little overhead results. We expect that SynEva can apply to, and help evaluate, more ML programs for new scenarios.**

***Index Terms*—Machine learning, program synthesis, mirror program, similarity measurement.**

I. Introduction

Machine learning (ML) is gaining increasing popularity nowadays. Various ML applications are developed for diverse tasks, including self-driving vehicle [1], recommendation system [2], social network analysis [3], image classification [4], and natural language processing [5]. Generally, such applications follow a similar workflow: (1) extracting knowledge from existing scenarios (a.k.a., *training scenarios*), and (2) then using the extracted knowledge in new scenarios (a.k.a., *predicting scenarios*). However, such extracted knowledge may not always work in predicting scenarios, if they differ substantially from training scenarios. In this case, undesirable consequences [6] can occur. For example, a Knightscope K5 security robot recently drowns itself into a fountain, because its predicting scenario is water surface, quite different from its training scenarios of road surfaces [8].

Regarding this, an important question is: "*how does one decide whether the knowledge extracted from training scenarios still suits predicting scenarios*?" Knowing the answer, one can then decide to use such extracted knowledge reliably for predicting scenarios, or need to take additional training to improve the knowledge for predicting scenarios.

One straightforward approach is to examine the performance of the extracted knowledge directly in predicting scenarios, i.e., examining whether the performance is satisfactory from the point of view of its users. Typically, to obtain satisfactory performance, there are two ways: (1) restricting predicting scenarios equal to training scenarios, and (2) validating the performance for predicting scenarios by human users. For (1), it brings heavy restrictions and may not be possible. This is because predicting scenarios typically refer to practical environments, usually more complex and sometimes quite different, as compared to training scenarios, which commonly refer to laboratory environments or contain only partial practical data. For (2), manual validation is inevitable and this process can be time-consuming and error-prone. Its underlying reason is the lack of *oracle* (i.e., an automated mechanism that can accurately tell what is the expected result given any input). One example is the expected driving action by human drivers given any scenario in front of a vehicle. Only human drivers know what to do in their minds.

To address the lack-of-oracle problem, another approach is to deploy differential testing. Differential testing has been widely used to alleviate the oracle problem [9], [10]. It, rather than referring to the oracle for predicting scenarios that may not be available, instead compares the outputs of different implementations for the same specification/requirement (e.g., multiple trained models for self-driving vehicles) to find out their behavioral differences when facing the same inputs [11]. Then any difference can potentially disclose latent problems in the knowledge under examination. However, differential testing assumes the availability of multiple implementations, which may not always be available in practice, and even if they are available, they may not follow exactly the same specification/requirement. Then the disclosed problems could be simply differences in features among these implementations. Besides, such behavioral differences still require manual analysis for a decision. For example, in a self-driving vehicle application, the difference of "30 degrees to right" vs. "32 degrees to right" can hardly be decided to be significant or not automatically.

As such, answering our earlier question requires new research efforts. We conjecture that a desirable approach should: (1) not rely on the oracle for predicting scenarios, (2) not require multiple implementations for the same specification/requirement, and (3) not involve manual efforts.

In this paper, we present a novel approach, SynEva, to meet the preceding three requirements, such that it can automati-

[*]Corresponding author

cally decide how previous knowledge extracted from training scenarios (e.g., trained model) suits predicting scenarios. To our best knowledge, SynEva, is the first approach to address this problem without relying on any oracle, existing implementation, or manual effort. Our key insight behind SynEva is that, it constructs an oracle-alike structure (named *mirror*) with respect to the existing knowledge, such that the mirror and existing knowledge behave almost exactly the same on the training scenarios but do not have any guarantee on predicting scenarios. Then if the mirror and existing knowledge behave similarly on predicting scenarios, one can decide that the knowledge can also work satisfactorily on predicting scenarios. Otherwise, they behave quite differently, and one can decide that the knowledge will not work satisfactorily on predicting scenarios. By doing so, SynEva transforms the challenging problem into an easy one of deciding whether the existing knowledge behaves similarly on predicting scenarios as the mirror, which can be constructed automatically.

The key challenge of SynEva is how to automatically generate a mirror for existing knowledge, such that the mirror and knowledge behave almost exactly the same on training scenarios, i.e., the mirror can represent the existing knowledge on training scenarios. SynEva addresses this challenges by program synthesis techniques [12], [13], which systematically explore the behavior of existing knowledge on training scenarios and construct a program to mimic the behavior. SynEva makes this process fully automated, and the constructed mirror program can behave as closely to the existing knowledge as possible. This makes that SynEva does not have to rely on any oracle or existing implementation.

With the constructed mirror program, SynEva can then decide how the previously extracted knowledge from training scenarios suits new predicting scenarios. To evaluate its effectiveness, we applied SynEva to the K-means clustering algorithm [14] with its five program variants of different settings on four real-world data sets of various sizes. Our experimental results show that SynEva effectively evaluated how the existing knowledge extracted from training scenarios suits predicting scenarios, with a strong correlation (-0.81) with the oracle (i.e., actual suiting extents). Besides, the results also show that SynEva's key design, i.e., synthesizing mirror programs by incorporating every detail from the extracted knowledge, directly contributed to its effectiveness, which brings a much stronger correlation as compared to some straightforward design, e.g., simulating the overall behavior of the extracted knowledge. Moreover, SynEva incurred very little overhead, no more than 5.1 seconds for its whole synthesis and evaluation process, suggesting its practicability.

In summary, this paper makes the following contributions:

- Proposed a program synthesis technique to systematically construct an oracle-alike mirror program with respect to the knowledge extracted from training scenarios.
- Proposed a similarity measurement technique to decide how existing knowledge for training scenarios suits new scenarios based on the mirror program's performance.
- Evaluated the SynEva approach with ML applications on real-world data sets.

The remainder of this paper is organized as follows. Section II introduces background of machine learning and program synthesis techniques. Section III elaborates on our SynEva approach, including a program synthesis technique for constructing mirror programs and a similarity measurement technique for evaluating how existing knowledge suits new scenarios. Section IV experimentally evaluates SynEva's performance with ML applications with real-world data sets. Section V discusses related work in recent years, and finally Section VI concludes this paper.

II. Backgrounds

In this section, we introduce some background knowledge about machine learning and program synthesis techniques.

A. Machine Learning (ML)

Typically, ML programs extract knowledge from training scenarios in terms of trained models (e.g., trees, networks, and other graphs), and then use them in new scenarios for various applications.

A training instance denotes one line of data from training scenarios. It consists of a sequence of *features* (e.g., weather condition, road condition, and driving speed of a vehicle) and a corresponding *label* specifying what category this instance should be classified into (e.g., a suggested next action for self-driving). It can be represented as $<f_1, f_2, \ldots, f_m, label>$. Each f_i denotes a specific feature value (e.g., "rainy" for the "weather condition" feature) and $label$ denotes its category (e.g., "braking").

A predicting instance denotes one line of data from predicting scenarios. It consists of a sequence of features only, waiting for predicting its corresponding label. It can be represented as $<f_1, f_2, \ldots, f_m>$. Then, based on the knowledge extracted from training scenarios, each predicting instance can find its corresponding label. This process is known as *classification*. A good ML algorithm is expected to have a high classification accuracy.

Within the scope of this paper, the ML programs/algorithms we refer to are such classification-based ones. Besides, their processes rely on labels, and thus they belong to supervised learning. For ease of presentation, we in subsequent discussions directly name them ML programs when there is no ambiguity.

B. Program Synthesis

Program synthesis aims to automatically construct a program satisfying a given correctness specification [15], [16]. It can be inductive or deductive. Inductive synthesis constructs a program based on a finite set of execution instances from the specification, and solves the construction problem by finite space searching [17]. Deductive synthesis constructs a program by directly refining the given specification [13].

In this paper, we use inductive program synthesis. It requests for a infinite set of execution instances or input-output examples [18], [19], without requiring any formal specification.

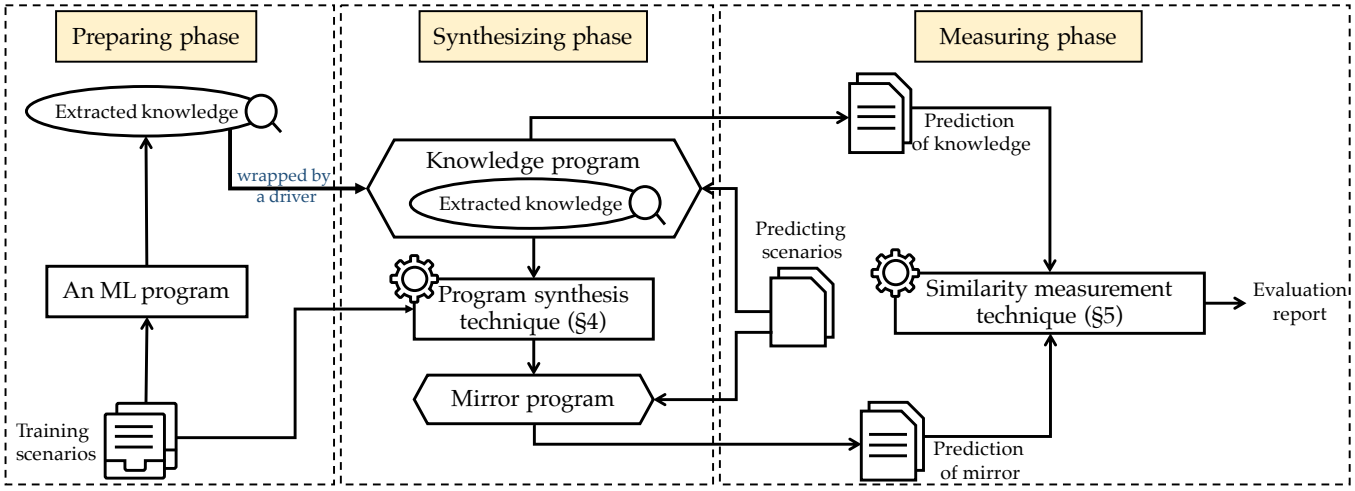


Fig. 1. SynEva overview.

Formally, the inductive program synthesis is as follows:

Problem. *Inductive program synthesis*

Given: (1) *A set of inputs $\mathcal{I}$, and (2) a set of corresponding outputs $\mathcal{O}$*;

To find: *A program P, such that when P takes any input from $\mathcal{I}$, it returns the corresponding output in $\mathcal{O}$.*

In the following, we refer to inductive program synthesis by program synthesis for ease of presentation. It well suites ML programs since they typically come with plenty of examples for training.

III. The SynEva Approach

In this section, we give our SynEva approach. We first give an overview of the whole approach, and then elaborate on the details of its two techniques. Finally, we take the K-means clustering algorithm [14] as the subject and explain how to apply SynEva to it.

A. Overview

We give our SynEva overview in Fig. 1. It consists of three phases: (1) *preparing*, (2) *synthesizing*, and (3) *measuring*. Phase (1) makes preparations by extracting knowledge from training scenarios via the given ML program, and wrapping the extracted knowledge for future use via an executable driver (together known as the *knowledge program*). Then phase (2) constructs a mirror program by program synthesis from the knowledge program and training scenarios, such that the mirror program behaves as closely to the knowledge program as possible on the training scenarios. After that, phase (3) compares the behaviors of the knowledge and mirror programs on predicting scenarios, and measures their similarity to compose a report that evaluates how the extracted knowledge suits the given predicting scenarios.

The whole process of the SynEva approach is automated. This facilitates its application. Among the three phases, preparing is essentially the traditional training. Therefore, we elaborate on synthesizing and measuring in the following.

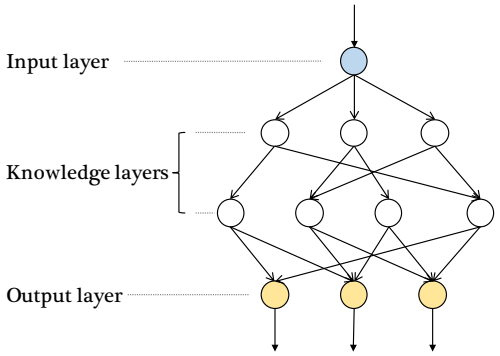


Fig. 2. Knowledge structure.

B. Program Synthesis

SynEva uses program synthesis to construct a mirror program that behaves as closely to the existing knowledge program as possible. We first explain more about the knowledge program. Its contained knowledge is extracted from trained scenarios, and is in the form of various trained models (e.g., trees, networks, and other graphs), depending on which ML algorithm the training has used. We show the general structure of such knowledge in Fig. 2.

Conceptually, the knowledge structure consists of three kinds of layers, *input layer*, *knowledge layer*, and *output layer*. The input layer accepts a given instance's feature values for prediction. The output layer reports the prediction about which category (i.e., a label) this instance should belong to. Multiple knowledge layers may exist, which store key information for making (intermediate and final) predictions. On a knowledge layer, each node can have its own prediction logic (e.g., a condition in a decision tree (DT) [20], or a perception function in a deep neuron network (DNN) [21]).

Based on this knowledge structure, to construct a mirror program that behaves as closely to a given knowledge program,

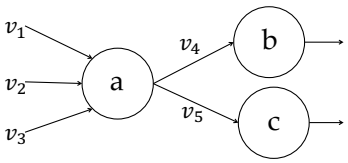


Fig. 3. A node's structure.

one needs to consider the logics of both its whole structure and individual nodes. In the following, we start with some necessary concepts.

Variable. For each node in the knowledge structure, it has own inputs (from its input edges). We consider these inputs as values of the *variables* associated with a node, e.g., three variables for the a node in Fig. 3.

Step. We define a *step* as the basic granularity during the prediction. For a given instance, predicting its label can take multiple steps, starting with $Step_0$ and ending with $Step_F$. $Step_0$ denotes the step that the input layer accepts feature values from the given instance, i.e., initializing values for variables of nodes on the input layer. $Step_F$ denotes the step that values of variables have been propagated to those on the output layer. Each intermediate step denotes a point in the propagation flow along the knowledge structure. Suppose that for the example in Fig. 3, $Step_x$ denotes that the propagation has reached the a node. Then the effect of the next step $Step_{x+1}$ would depend on node a's logic, with node b, c, or both of them propagated with a value along a's output edges associated with variables v_4 and v_5 (represented as $\mathsf{a} \rightarrow \mathsf{b}$ and $\mathsf{a} \rightarrow \mathsf{c}$).

Sequence of steps. Which nodes/edges are involved in a propagation step can differ in different knowledge structures, e.g., by checking whether a node's condition is satisfied in a DT, or whether a node's perception function is activated in a DNN. We represent the whole prediction for a given instance by *a sequence of steps*, $<Step_0, Step_1, \ldots, Step_F>$, with each step involves a single node/edge or multiple nodes/edges.

With these concepts, we now explain how to construct the mirror program for a given knowledge program. We make the mirror program to have the same structure as the knowledge program, with each step as close to each other in the predictions for training scenarios as possible. To do so, we in the mirror program: (1) simulate all involved structures during the propagation in order to predict for training scenarios, and (2) synthesize each simulated node's logic to be as close to its corresponding one in the knowledge program as possible. We explain them in turn.

(1) Simulating structures. We consider all nodes/edges involved in the predictions for training scenarios, and simulate them in the mirror program. This part is straightforward by node/edge mappings.

(2) Mirroring logics. This part is critical in SynEva, which uses program synthesis to construct the logics for all nodes in the mirror program. We require that for any given inputs (variables and their values) encountered in the predictions for training scenarios by the knowledge program, the mirroring program should make its corresponding simulated node to generate the same outputs (activating follow-up nodes/edges), i.e., the same steps.

We use Supported Vector Machine (SVM) [22] as the logic learner for the mirror program due to the efficiency concern. Note that SynEva can also choose other efficient learners here. Then for each node in the mirror program, its logic is learned from all propagation history of its corresponding node in the knowledge program on the training scenarios. To make the learning simple, we adopt multi-label SVM [23], i.e., learning whether to activate one or more follow-up nodes/edges. For the example in Fig. 3, SynEva learns to choose one of four possibilities, i.e., activating edge $\mathsf{a} \rightarrow \mathsf{b}$, edge $\mathsf{a} \rightarrow \mathsf{c}$, both edges, or none of them, for the a node. Besides, to make our logic learner more flexible, SynEva also allows considering input variables of nodes involved in earlier steps. That is, when learning the logic for a node involved in $Step_x$, one can include in the learning the values of input variables for nodes involved in $Step_{x-1}$, ..., $Step_{x-k}$, with a *depth* of k.

Then for each node in the mirror program, SynEva synthesizes its logic by training its SVM-based logic learner using values of input variables from corresponding nodes in the knowledge program involved in its current and earlier k steps. This covers all instances in the training scenarios, so that the mirror program can behave as close to the knowledge program as possible for each simulated node. In particular, these trained logic learners in the mirror program simulate the knowledge program's behavior in choosing the same propagations (i.e., activating nodes/edges) when given the same inputs.

With the logic learners trained for the mirror program, SynEva has one remaining issue: how to obtain proper values for variables when the variables' associated edges are activated. Some ML algorithms, e.g., DT, only choose activated edges to execute. In this case, there is no need to obtain additional variable values for activated edges. However, other ML algorithms, e.g., DNN, need additional variable values propagated along activated edges. In this case, SynEva needs to obtain proper values for these variables associated with activated edges. To do so, SynEva copies original value-calculation functions (e.g., perception function in a DNN) in each node in the knowledge program to its corresponding node in the mirror program. As such, when a specific edge is activated for a given instance, SynEva can invoke its corresponding copied function to obtain its proper value.

With the synthesized mirror program, it can predict instances for new scenarios. In the following, we explain the use of the mirror program to help decide how the knowledge program suits new scenarios.

C. Similarity Measurement

The key insight behind SynEva is that: (1) if the knowledge extracted from training scenarios suits given new scenarios, then both the knowledge and mirror programs should behave similarly satisfactory for the new scenarios, since the training

and new scenarios should belong to the same type; (2) on the other hand, if the extracted knowledge does not suit given new scenarios, then the knowledge and mirror programs should behave differently for the new scenarios, since they agree only on training scenarios, not guaranteeing any performance for other scenarios, and thus the likelihood that they behave similarly satisfactory/unsatisfactory is low. Therefore, one can quantify the extent of how the existing knowledge suits new scenarios by measuring the behavioral similarity between the knowledge and mirror programs.

Consider new scenarios that contain multiple instances for prediction. With the given knowledge program and mirror program, each instance can be predicted for its label individually by them. For each prediction, it comes along with two sequences of steps, S_K and S_M, from their propagations during the prediction by the two programs, respectively.

In order to measure the similarity between two sequences, S_K and S_M, SynEva considers both similarities of their contained edges and edge orders.

(1) **Similarity of contained edges.** For S_K and S_M, SynEva derives two sets of contained edges associated with them, respectively, i.e., $EdgeSet_K$, $EdgeSet_M$. SynEva calculates the Degree of Similarity (DoS) value for the two sets of edges. Specifically, SynEva uses the Jaccard similarity index [24] for the calculation. Formally,

$$DoS(EdgeSet_K, EdgeSet_M) = \frac{|EdgeSet_K \cap EdgeSet_M|}{|EdgeSet_K \cup EdgeSet_M|}.$$

This calculation measures how much percentage of all contained edges are overlapping between two sequences, S_K and S_M.

(2) **Similarity of edge orders.** For S_K and S_M, SynEva also derives two sets of *forwarding edge pairs* associated with them, respectively, i.e., $PairSet_K$, $PairSet_M$. Each forwarding edge pair takes one edge from a step in the given sequence and connects it to another edge in this step's next step. Here, we omit the initial step $Step_0$ and final step $Step_F$, since they are always the same for S_K and S_M. For example, consider a sequence, $<Step_0, Step_1, Step_2, Step_F>$, where $Step_1$ involves $edge_1$ and $edge_2$, and $Step_2$ involves $edge_3$ and $edge_4$. Then, the set of all forwarding edge pairs would be: $\{<edge_1, edge_3>, <edge_1, edge_4>, <edge_2, edge_3>, <edge_2, edge_4>\}$. Similarly, SynEva calculates the Degree of Similarity (DoS) value for the two sets of forwarding edge pairs. Formally,

$$DoS(PairSet_K, PairSet_M) = \frac{|PairSet_K \cap PairSet_M|}{|PairSet_K \cup PairSet_M|}.$$

To make it simple, we consider the two similarity measurements equally important in this work, although it can be subject to change to cater for special considerations. Combining the two measurements together, SynEva calculates the behavioral similarity between the knowledge and mirror programs for one particular instance (incurring two sequences S_K and S_M) from new scenarios as follows:

$$DoS(S_K, S_M) = 0.5 \times DoS(EdgeSet_K, EdgeSet_M) + 0.5 \times DoS(PairSet_K, PairSet_M).$$

The above formula calculates the degree of similarity between a pair of sequence of steps for a given instance. Suppose that the given new scenarios contain multiple predicting instances $S : \{I_1, I_2, ..., I_n\}$. Let $DoS(S_K(I_i)/S_M(I_i))$ be the calculated degree of similarity for instance I_i. Then SynEva decides how the knowledge extracted from training scenarios suits new scenarios by a metric of the average degree of similarity for all these instances. Formally,

$$Similarity(S) = \frac{\sum_{I_i \in S} DoS(S_K(I_i), S_M(I_i))}{|S|}.$$

The value of the similarity metric falls in the range of [0, 1]. According to our earlier analyzed insights, when the similarity has a higher value, it indicates that the existing knowledge suits new scenarios well (i.e., would obtain a similarly satisfactory accuracy as for training scenarios). Otherwise, when the similarity has a lower value, it indicates that the knowledge does not suit the new scenarios well (i.e., would obtain a different, probably lower, accuracy as compared to the training scenarios).

D. Applying SynEva to K-means Clustering

SynEva is a general approach for evaluating ML programs (supervised, classification-based). In this paper, we take the K-means clustering algorithm as the subject to explain how to apply SynEva to it. We are also working on SynEvas application to other ML programs but consider them as our future work. In the following, we explain how to use K-means clustering for ML, how to define its knowledge structure for program synthesis, and how to collect sequences of steps for similarity measurement.

First, the K-means algorithm is originally for clustering, but can also be used for ML by deriving a trained model from training scenarios and later using the model to predict labels in new scenarios. To do so, one needs to store some key information, such as the number of clusters classified for instances from training scenarios and the clustered instances for each cluster. Besides, each cluster is assigned with a label in a greedy way according to label information in the training scenarios. Later, one can use the stored information to predict labels for instances from new scenarios. For example, when classifying a given instance, one can calculate the sum of squared errors as in K-means if classifying this instance into a cluster. After comparing all sums, the instance will be classified into the cluster that corresponds to the minimal sum. This decides its predicted label, which is associated with this cluster. After examining all instances from the new scenarios, the prediction accuracy can also be calculated if their actual labels (i.e., the oracle) are available.

Second, in defining the knowledge structure for program synthesis, SynEva can be used in an easier way for the K-means clustering algorithm. The reason is that the knowledge structure in Fig. 2 essentially simulates a traditional program's control flow graph, and the preceding logic of using K-means clustering for ML is already a program. Therefore, instead of constructing another knowledge structure for K-means clustering, we directly let SynEva synthesize a program to

TABLE I
DESCRIPTION OF THE FOUR REAL-WORLD DATA SETS

Data set	# total instances	# categories	# training instances	# predicting instances
IRIS Flower	150	3	100	50
Balance Scale Weight & Distance	625	3	425	200
DSRC Vehicle Communications	1,000	2	900	100
IRIS Waveform	5,000	3	4,000	1,000

simulate this K-means clustering program's behavior. The only thing worth mentioning is that all branch points in the program are considered as nodes in the original knowledge structure. Besides, in program synthesis, a depth value for k needs setting. A larger value implies that more nodes participate into the learning, and thus a higher accuracy is expected but with a correspondingly higher cost. We set the k value according to existing work on simulating model behavior by LSTM [12], which suggests $k = 2$.

Third, measuring the degree of similarity between the knowledge and mirror programs concerns collecting sequences of steps in making predications for given instances. With the preceding program structure as the knowledge structure, a sequence of steps corresponds to an execution trace (concerning branch points only), and each step corresponds to one propagation edge only. This is simple and can be realized easily in SynEva's application.

IV. EVALUATION

In this section, we experimentally evaluate our SynEva approach on its effectiveness in evaluating how the existing knowledge extracted from training scenarios suits new scenarios, as well as studying its internal characteristics, e.g., use of SVM logic learners, impact of the number of evaluated instances from new scenarios, and overhead in program synthesis and similarity measurement. We implemented SynEva as a prototype tool in Java 8, and aim to study based on the implementation the following four research questions:

- **RQ1.** *How effective is SynEva in evaluating how existing knowledge from training scenarios suits new scenarios by similarity measurement for knowledge and mirror programs?*
- **RQ2.** *How does SynEva (simulating every detail of a knowledge structure by multiple SVM-based logic learners) compare to its simple version (simulating the whole logic by a single SVM learner)?*
- **RQ3.** *What is the impact of the number of evaluated instances from new scenarios on SynEva's effectiveness?*
- **RQ4.** *What is SynEva's overhead in program synthesis and similarity measurement?*

A. Experimental Subjects

ML programs. As mentioned earlier, we applied SynEva to the K-means clustering algorithm [14]. Its corresponding program contains 320 LOC and four classes. The effectiveness of K-means clustering depends on its parameter settings, and we deploy it with five different settings, which either concern different distance measuring methods or termination conditions. As such, they form five program variants, represented as K1, K2, ..., and K5, respectively, and cal lead to different levels of effectiveness in ML, as we show later.

Data sets. We selected four real-world data sets for testing the five ML program variants: (1) IRIS Flower Data [26] provided by Incorporated Research Institutions for Seismology (IRIS), which includes 150 data instances of three different categories (labels), (2) Balance Scale Weight & Distance Database [27] provided by the University of California, Irvine (UCI), which includes 625 data instances of three different categories, (3) DSRC Vehicle Communications Data [28] provided by UCI, which includes 1,000 data instances of two categories, and (4) IRIS Waveform Data [29] also provided by IRIS, which includes 5,000 data instances of three different categories. We denote the four data sets as S1, S2, ..., and S4, respectively. For each data set, we randomly separated it into a training subset (as SynEva's training scenarios) and a predicting subset (as SynEva's new/predicting scenarios). The former is relatively larger and the latter is smaller. Their relative size ratio follows ML conventions. We give the data set details in Table I.

Configurations. Combining the five ML program variants (K1, K2, ..., and K5) and four real-world data sets (S1, S2, ..., and S4), we obtain 20 different configurations (5 $\times$ 4). SynEva was deployed with these 20 configurations for evaluation and comparison. Note that SynEva aims to evaluate how the knowledge extracted from training scenarios suits new scenarios (predicting scenarios). Under our experimental settings, according to how SynEva is applied to K-means clustering, an extracted knowledge is decided by both the given ML program and given training scenarios. As such, the extent that the extracted knowledge suits given predicting scenarios can vary under different configurations. This makes the 20 configurations and their associated predicting scenarios qualify for our SynEva's evaluation, since they have different suiting extents.

Oracle. Since predicting scenarios contain label information already, one can calculate an ML program's accuracy for given predicting scenarios accordingly, as for training scenarios. Then, by comparing the accuracy difference for an ML program between a pair of training scenarios and predicting scenarios, one can actually tell how the knowledge extracted from the training scenarios suits the predicting scenarios. This serves as the oracle for evaluating our SynEva's effectiveness in experiments. That is, we will see whether SynEva can accurately evaluate such differences by its reported similarity metric values.

B. Experimental Setup and Design

To answer the four research questions, we evaluate SynEva under the aforementioned 20 different configurations.

TABLE II
CORRELATION BETWEEN ACCURACY DIFFERENCES AND SYNEVA'S REPORTED SIMILARITY METRICS

Data set	ML program variant	Training accuracy	Predicting accuracy	Accuracy difference	SynEva's reported similarity metric
S1	K1	75.37%	72.50%	2.87%	87.08%
	K2	72.50%	67.50%	5.00%	83.91%
	K3	82.50%	81.05%	1.45%	91.05%
	K4	87.50%	82.50%	5.00%	86.69%
	K5	90.00%	87.25%	2.75%	92.13%
S2	K1	67.14%	60.26%	6.88%	90.81%
	K2	66.63%	60.00%	6.63%	79.18%$^{+}$
	K3	64.71%	55.42%	9.29%*	67.23%$^{+}$
	K4	60.26%	56.27%	3.99%	82.22%
	K5	46.68%	42.16%	4.52%	85.54%
S3	K1	73.60%	70.24%	3.36%	85.30%
	K2	86.55%	83.85%	2.70%	87.80%
	K3	70.26%	63.27%	6.99%*	78.05%$^{+}$
	K4	77.24%	64.85%	12.39%*	69.64%$^{+}$
	K5	74.32%	70.64%	3.68%	88.45%
S4	K1	84.23%	77.04%	7.19%*	79.85%
	K2	83.50%	79.01%	4.49%	84.83%
	K3	88.00%	79.01%	8.99%*	78.25%$^{+}$
	K4	84.01%	69.08%	14.93%*	73.80%$^{+}$
	K5	87.40%	80.83%	6.57%	85.81%

* Top 30% unsuitable cases ranked according to the accuracy difference
$^{+}$ Top 30% unsuitable cases ranked according to SynEva's reported similarity metric

To answer RQ1, for each configuration, we compare the oracle (i.e., actual accuracy differences for an ML program between a pair of training scenarios and predicting scenarios) to SynEva's reported similarity metric values. We consider that a larger accuracy difference implies a lower suiting extent, and vice versa. SynEva should be able to disclose such a relationship. Therefore, we study the correlation between reported accuracy differences (by the oracle) and reported similarity metric values (by SynEva). If a strong correlation can be found. SynEva would be validated to be effective in evaluating how the knowledge extracted from training scenarios suits new scenarios.

To answer RQ2, we actually want to compare SynEva to existing work. However, we are not aware of any existing work that can evaluate how the knowledge extracted from training scenarios suits new scenarios without relying on any oracle, comparable implementation, or manual effort. As such, we compare SynEva to its simple version that also represents an intuitive idea of using ML to capture the behavioral logic of a program in a black box way. SynEva simulates every detail of a knowledge structure by multiple SVM-based logic learner. For comparison, we implemented its simple version by simulating the whole knowledge structure by a single SVM learner, which is named SvmEva. Other parts are the same for SynEva and SvmEva. By comparing their effectiveness under the 20 configurations, we plan to validate the necessity of SynEva's program synthesis on both the knowledge structure and its every logic detail.

To answer RQ3, we reduce the number of evaluated instances from predicting scenarios with a pace of 25%, (i.e., 100%, 75%, 50%, and 25%), to study its impact on SynEva's effectiveness in evaluating how the knowledge extracted from training scenarios suits new scenarios. We conjecture that less evaluated predicting instances could possibly lead to more unstable similarity metric values. Therefore, this should negatively affect the studied correlation, which directly relates to SynEva's effectiveness. We plan to validate this in experiments.

To answer RQ4, we measure SynEva's time costs in synthesizing the mirror programs and making similarity measurements. The costs should be acceptable.

All experiments were conducted on a commodity PC with an Intel(R) Core(TM) i7 CPU @4.2GHz and 32GB RAM.

C. Experimental Results and Analyses

In the following, we answer the preceding four research questions in turn.

RQ1. We conducted experiments with the aforementioned 20 configurations and collected SynEva's reported similarity metric values. All data are listed in Table II, aligned with these configurations' corresponding training accuracy, predicting accuracy, and accuracy difference data. We can roughly observe that when SynEva reports a high similarity metric value, say 91.05%, its corresponding difference is low, say 1.45% (suiting). On the other hand, when SynEva reports a low similarity metric value, say 69.64%, its corresponding accuracy difference is high, say 12.39% (unsuiting). To reach

a deeper understanding of such a relationship, we calculate the correlation between the two series of data (accuracy difference vs. similarity metric), and obtain a result of -0.81, which discloses a strong negative correlation. This result suggest that SynEva can effectively evaluate how the knowledge extracted from training scenarios suits new scenarios by its similarity measurement for knowledge and mirror programs.

Besides, we also observe that SynEva is good at identifying extremely unsuiting scenarios. For example, when considering the top 30% unsuiting cases ranked according to the accuracy difference (i.e., top large accuracy differences from the oracle), they concern six configurations, namely, S2-K3, S3-K3, S3-K4, S4-K1, S4-K3, and S4-K4. Among them, five out of six have been reported by SynEva as the top 30% unsuiting cases ranked according to the similarity metric (i.e., top small similarity metric values). The only missing one, S4-K1, is ranked #5 by the oracle and #7 by SynEva. Their ranks are actually very close to each other.

Therefore, we answer RQ1 as follows:

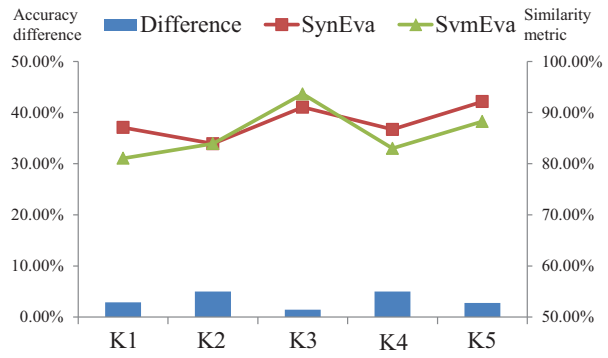
> *SynEva can effectively evaluate how an ML program's knowledge extracted from training scenarios suits new scenarios, especially for extremely unsuiting cases. Besides, its evaluation has a strong negative correlation (-0.81) with the oracle.*

RQ2. We then study how SynEva (simulating every detail of a knowledge structure by multiple SVM-based logic learners) compares to its simple version SvmEva (simulating the whole logic by a single SVM learner). Fig. 4 compares such two settings on the five program variants and four data sets (a–d).

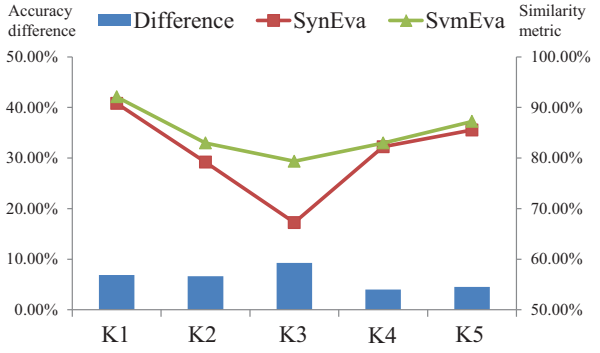
A rough observation may suggest that SynEva and SvmEva can still report close similarity metric values on some points (e.g., K2 for S1 and K4 for S2). However, an objective comparison should take into account all these performance data (covering all 20 configurations). Therefore, we also show their corresponding accuracy differences from the oracle in Fig. 4, and compare the correlations between SynEva/SvmEva data and the oracle data for each data set. Then for data set S1, the correlation for SynEva is -0.63, while for SvmEva, it is only -0.31 (50.8% less by absolute value). For S2, the correlation is -0.80 and -0.70 (12.5% less), respectively. For S3 and S4, the correlation is -0.91 vs. -0.74 (18.7% less) and -0.98 vs. -0.82 (16.2% less). If combining all data sets together (i.e., all 20 configurations), the correlation is -0.81 vs. -0.62 (23.5% less). We observe that no matter considering each data set or all data sets together, SynEva always outperforms SvmEva in the correlation. This result validates the necessity of SynEva's program synthesis on both the knowledge structure and its every logic detail.

Therefore, we answer RQ2 as follows:

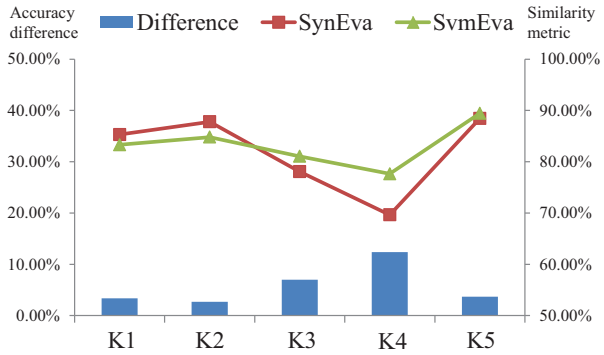
> *SynEva synthesizes a mirror program by considering both the knowledge structure and its every logic detail. This design directly contributes to its effectiveness. SynEva can thus produce a much stronger correlation, as compared to the simple design of simulating the whole logic by a single SVM learner.*



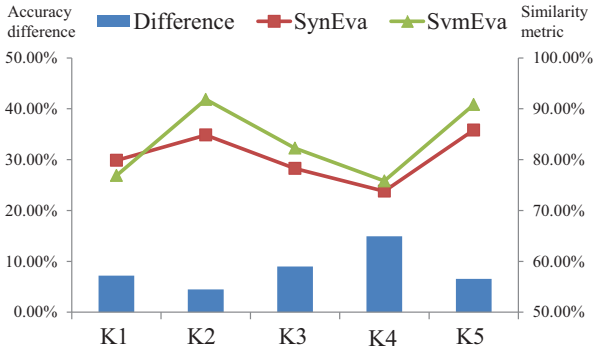
(a) On data set S1



(b) On data set S2

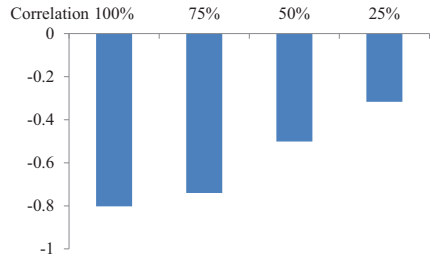


(c) On data set S3

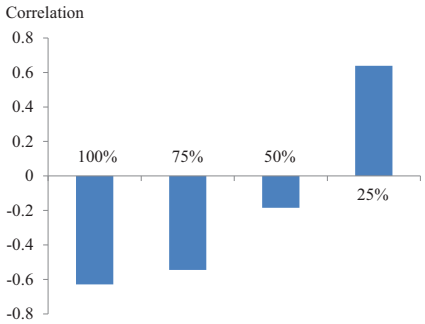


(d) On data set S4

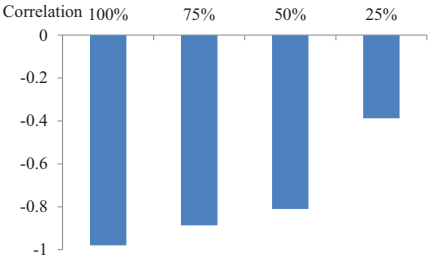
Fig. 4. Comparison of SynEva's program synthesis techniques (whole structure + logic detail vs. whole logic)



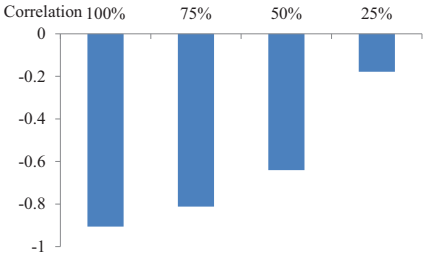
(a) On data set S1



(b) On data set S2



(c) On data set S3



(d) On data set S4

Fig. 5. Impact of the number of evaluated instances on SynEva's effectiveness

RQ3. We next study the impact of the number of evaluated instances from predicting scenarios on SynEva's effectiveness. So we controlled the number from 100% (i.e., all predicting instances) down to 25% with a pace of 25%, and study its impact on the correlation between the oracle and SynEva's evaluation results. Fig. 5 shows how the correlation changes with the decreasing of the number of evaluated instances on the four data sets (a–d).

We observe that in all but one cases, the absolute value of the correlation dramatically decreases, becoming closer to zero. This suggests that the correlation becomes weaker with the decreasing of the number of evaluated instances, clearly negatively affecting SynEva's effectiveness. When averaging all data sets, their average correlation is -0.83 if one uses 100% instances, but it drops (by absolute value) to -0.54 when using half instances (50%). For the only exceptional case (from S2-50% to S2-25%), the negative correlation of -0.18 unexpectedly becomes a positive correlation of 0.64. This directly denies SynEva's original purpose (i.e., potentially considering suiting/unsuiting scenarios as unsuiting/suiting ones). This suggests that in such an extremely case, SynEva's reported evaluation might no longer be trustworthy. Therefore, SynEva's effectiveness would rely on the number of evaluated instances from predicting scenarios. The more, the better. In practice, we consider that the ratio between the number of evaluated predicting instances and that of training instances can be a good indicator. Since our experiments have used a range of such ratios (11.1–50%) for RQ1 and they all correspond to satisfactory results, we suggest that a threshold of 20% can be a good choice. Nevertheless, the threshold should relate to specific ML programs and associated testing scenarios. This issue deserves future work and is not in the scope of this work.

Therefore, we answer RQ3 as follows:

> *The number of evaluated instances from predicting scenarios has an impact on SynEva's effectiveness. With the decreasing of this number, SynEva's reported evaluation results would become less trustworthy.*

TABLE III
OVERHEAD IN PROGRAM SYNTHESIS

Data set	Time cost (s)	# Training instances	Time cost per instance (ms)
S1	0.32	100	3.20
S2	0.71	425	1.67
S3	3.26	900	3.62
S4	4.82	4,000	1.20
Total	9.11	5,425	1.68

TABLE IV
OVERHEAD IN SIMILARITY MEASUREMENT

Data set	Time cost (ms)	# Predicting instances	Time cost per instance (ms)
S1	17.53	50	0.35
S2	179.64	200	0.90
S3	74.44	100	0.74
S4	283.40	1,000	0.28
Total	555.01	1,350	0.41

RQ4. Finally, we measure SynEva's overhead in terms of time costs on its program synthesis and similarity measurement. Table III and Table IV list these time cost data.

We observe that SynEva spent 0.32–4.82 seconds (average: 2.28 s) to synthesize mirror programs and 17.5–283.4 milliseconds (average: 138.75 ms) to measure similarity metric

values. We note that the time costs depend on the number of training/predicting instances. So we also calculate the average time costs per instance. Then the costs are 1.68 ms and 0.41 ms, respectively. We observe that no matter one considers all instances or one instance, the time costs are both small. In practice, one may apply SynEva to a data set with 10,000 training instances and 2,000 predicting instances (as a typical ML application), the total time cost would be: 16.8 s + 0.82 s = 17.62 s, which is clearly acceptable.

Therefore, we answer RQ4 as follows:

> *SynEva's overhead is small. Its time cost goes for program synthesis and similarity measurement. In our experiments, the former takes 0.32–4.82 seconds and the latter takes 17.5–283.4 milliseconds only.*

D. Discussions and Threat Analyses

There are some issues that deserve further discussions regarding SynEva's usage and its experimental validity.

First, we applied SynEva to K-means clustering and evaluated its effectiveness. As mentioned, K-means is originally for clustering, and does not strictly follow a traditional ML process (supervised, classification-based), i.e., first training a model and then using the model for prediction. As the first subject in our work, we carefully considered the application of K-means clustering for the ML purpose and explained it in Section III.D. This makes K-means clustering work as ML, and thus automatically suitable for our SynEva approach. Note that SynEva does not use any special feature of K-means clustering, and thus applicable to other ML programs as long as they have a training and prediction interface. We are working along this line to test SynEva on more other ML programs.

Second, when evaluating SynEva's effectiveness on new scenarios, it would be better if one uses already tagged scenarios that clearly state whether they match original training scenarios from which the knowledge is extracted. Due to the lack of such resources, we make both training instances and predicting instances from the same sources (i.e., the four data sets). Nevertheless, this does not mean that thus constructed predicting scenarios must match training scenarios, as we have explained in Section IV.A. In fact, constructed predicting scenarios have clearly varying suiting extents, as compared to their corresponding training scenarios. The oracle shows that the accuracy difference between a pair of training scenarios and predicting scenarios varies from 1.45% to 14.93% (in Table II), which specifies a significant range. Note that SynEva is unaware of such accuracy differences, but still able to evaluate how the knowledge extracted from training scenarios suits predicting scenarios, with a strong correlation with the oracle. This validates its effectiveness.

Third, we earlier implied that when an ML program obtains a satisfactory model from training scenarios, then one would be interested in whether the trained model suits new scenarios (i.e., our target problem). In our experiment, the column "Training accuracy" in Table II suggests the performance of the trained model on the training scenarios from which the model is trained. We observe that the performance is quantified by an accuracy value range from 46.68% to 90.00%, seemingly unable to fall into "satisfactory". Nevertheless, we should say that the decision on whether an accuracy value is "satisfactory" is application-specific and ML technique-specific. In experiments, we do not make any assumption that a "satisfactory" accuracy value must be over 80% or even 95%. Instead, we tested SynEva with such a large range of accuracy values, and it still produced good results. In fact, SynEva itself does not have such a requirement. This helps widen its applicability.

V. Related Work

In this section, we discuss the related work in recent years on three aspects: *testing ML programs*, *testing programs without oracle*, and *automatic program synthesis*.

Testing ML programs. The traditional practices for testing ML programs primarily involves accuracy measurement on randomly sampled test inputs from manually labeled datasets [30]. For example, Google [31] used both in-field testing driving and unguided simulations to test its Waymo self-driving cars. Different from such black-box testing approaches, some researchers use an ML program's internal knowledge (i.e., its trained model) to reveal the program's abnormal behaviors. Goodfellow et al. [32] introduced an approach to finding an ML model's adversarial examples based on the model's linear behaviors in high-dimensional spaces. Nguyen et al. [33] showed how to produce images totally unrecognizable to human eyes that well-trained ML programs believe with certainty are different objects. Pei et al. [6] proposed a white-box framework to generate corner cases that can result in an ML program's erroneous behaviors via differential testing. Tian et al. [7] presented a systematical tool to automatically detect erroneous behaviors of DNN-driving vehicles based on transformation-specific metamorphic relations. Different from these testing techniques, our SynEva does not aims to reveal possible abnormal behaviors of ML programs, but tries to provide a quantitative evaluation of how an ML program's trained model from training scenarios suits new scenarios. Besides, SynEva does not require any oracle, comparable implementation, or manual effort for inspection.

Testing programs without oracles. The oracle problem closely relates to the effectiveness of software testing techniques. A conventional approach to testing a program without its oracle is to use invariants that encode a program's behavior in passing executions as an oracle-alike artifact for testing. As a representative approach for automated invariant generation, Daikon [34] inferred preconditions and postconditions for methods ever executed in a program, by collecting program execution information and then using its built-in templates to synthesize invariants from such collected information. DySy [35] followed a similar way and used branch conditions to infer invariants. Eclat [36] took automated invariant generation one step further, by learning a model from assumed executions and identifying inputs that differ from the

learned model. For testing purpose, some techniques, such as Randoop [37] and Evosuite [38], can generate test cases that include assertions encoding their observed behaviors from program executions. The synthesized mirror program in our SynEva approach from given knowledge can be regarded as a special type of invariants that encode the concerned ML program's behaviors on training scenarios. However, instead of using the mirror program alone as an assertion or invariant-alike artifact, SynEva compares the program's behavior with the corresponding knowledge program's behavior on predicting scenarios, and evaluates it with a quantified similarity measurement, which suggests how the knowledge program suits new scenarios.

Differential testing is another representative approach to testing programs without oracles. It typically relies on several already implemented programs with the same specification as cross-reference oracles. Chen et al. [9] combined differential testing with mutation testing to focus testing efforts on representative classfiles of JVMs. Yang et al. [10] provided a tool, CSmith, to automatically generate C programs that cover a large subset of C features, and used such generated programs' executions on different open-source C compilers to reveal potential compiler bugs. Jung et al. [39] developed a differential testing technique, in which perturbations in a mobile app's inputs are mapped to perturbations in the app's outputs to discover likely private information leaks. Petsios et al. [11] introduced a notion of δ-diversity to summarizes the observed asymmetries between the behaviors of multi-version programs, similar to differential testing, and provided a domain-independent input generation mechanism based on δ-diversity. Similar to existing differential testing techniques, our SynEva approach also uses a cross-reference oracle for evaluating an ML program's performance. Nevertheless, SynEva does not assume the availability of such an oracle. Instead, it constructs it from the existing knowledge automatically.

Automatic program synthesis. Automatic program synthesis is considered to be one of the most central problems in the theory of programming [40], and attracts many research efforts in the communities of software engineering and programming languages. Manna et al. [42] proposed deductive synthesis, whose basic idea is that a program can be extracted from a constructive proof of the satisfiability of a given specification. Solar-Lezama [25] used a partial program to express the high-level structure of the target program to be synthesized, and proposed an SAT-based inductive approach to synthesizing the program's implementation from a small number of test cases. Wu et al. [41] emphasized on entity transformation tasks and proposed ENTER to automatically synthesize such entity transformations into a program from examples based on a domain-agnostic language. There are various of off-the-shelf systems that support automatic program synthesis, including NuPRL [43] and KIDS [44]. These systems allow a programmer to provide the insight about the implementation at a high level, in the form of axioms and theorems about the problem domain, and then use them to automatically derive a correct implementation from the high-level specification. Different from these pieces of existing work, our SynEva approach synthesizes a mirror program using the given ML program's execution traces (or its propagation steps in the corresponding knowledge structure), instead of using the program's specification directly, which is typically difficult to obtain.

The artificial intelligence community also makes efforts on automatic program synthesis in recent years. Graves et al. [45] developed a neural Turing machine, which can learn and execute simple program functionalities, such as repeating, copying, sorting and so on. Vinyals et al. [46] presented Pointer Networks that generalize the notion of encoder attention in order to provide a decoder for a variable-sized output space depending on the input sequence's length. Joulin et al. [47] augmented a recurrent network with a pushdown stack, allowing for generalization to longer input sequences than one during training for several algorithmic patterns. Redd et al. [12] proposed a neural programmer-interpreter, a recurrent and compositional neural network that learns to represent and execute programs. Our SynEva approach also synthesizes programs, inspired by these pieces of existing work. Nevertheless, SynEva moves one step further by synthesizing mirror programs to automatically evaluate how existing knowledge suits new scenarios.

VI. Conclusion

In this paper, we present a novel ML program evaluation approach, SynEva, to automatically evaluate how a given ML program's knowledge extracted from training scenarios suits its new scenarios. SynEva realizes this goal by a program synthesis technique that systematically constructs the mirror program that accurately captures expected behaviors of the existing knowledge on the training scenarios, and a similarity measurement technique that automatically compares how similarly the mirror program behaves on the new scenarios as the knowledge program. SynEva is fully automated, and does not require any oracle, comparable implementation, or manual effort. Besides, it produces satisfactory results by strong correlation and little overhead as our experimental evaluation reports.

SynEva can potentially have boarder usages. In nowadays ML applications such as self-driving, driving logics are continuously trained and strengthened for future use. Such logics, as trained models from earlier scenarios, are becoming kernel program logics to self-driving. That is, they are not only outputs of ML programs, but also part of evolving application programs. Then the online validation on whether and how such logics suit future scenarios is a critical task, and our SynEva can potentially contribute to this, helping improve human life and protecting them from dangers.[1] Of course, researches and practitioners have a long way to go, and we are working along this line.

[1]The latest pedestrian-hitting event occurred on March 18, 2018: http://www.syracuse.com/us-news/index.ssf/2018/03/uber_self-driving_car_pedestrian_video_arizona.html

ACKNOWLEDGMENT

This work was supported in part by National Key R&D Program (Grant #2017YFB1001801), National Natural Science Foundation (Grants #61690204, #61472174) of China. The authors would also like to thank the support of the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

REFERENCES

[1] http://google.com/selfdrivingcar/

[2] J. Davidson, L. B. Liebald, J. Liu, P. Nandy, T. Van Vleet, U. Gargi, and S. Gupta, "The YouTube video recommendation system," in *proceedings of the fourth ACM Conference on Recommender Systems*, 2010, pp. 293-296.

[3] N.B. Ellison, C. Steinfield, and C. Lampe, "The benefits of Facebook 'friends:' Social capital and college students' use of online social network sites," in *Journal of Computer-mediated Communication*, vol. 12, no. 4, 2007, pp.1143-1168.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *proceedings of the 29th IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770C 778.

[5] G. Chowdhury, "Natural language processing," in *Annual Review of Information Science and Technology*, vol. 37, no. 1, 2003, pp. 51-89.

[6] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated whitebox testing of deep learning systems," in *proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1-18.

[7] Y. Tian, K. Pei, S. Jana, and B. Ray, "DeepTest: Automated testing of deep-neural-network-driven autonomous cars," arXiv preprint arXiv:1708.08559, 2017.

[8] http://time.com/4862263/security-robot-fountain-knightscope-k5/

[9] Y Chen, T.Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of JVM implementations," in *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation,* 2016.

[10] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011, pp. 283-294.

[11] T. Petsios, A. Tang, S. J. Stolfo, A. D. Keromytis, and S. Jana, "NEZHA: Efficient domain independent differential testing," in *proceedings of the 38th IEEE Symposium on Security and Privacy*, 2017, pp. 615-632.

[12] S. Reed and N. D. Freitas, "Neural programmer-interpreters," in *proceeding of the Fourth Internaional Conference on Learning Representations*, 2016.

[13] Z. Manna, and R. Waldinger. "A deductive approach to program synthesis," *Readings in Artificial Intelligence and Software Engineering*, vol. 2, no. 1, 1986, pp. 3C34.

[14] J. A. Hartigan, and M. A. Wong, "Algorithm AS 136: A k-means clustering algorithm," in *Journal of the Royal Statistical Society*, Series C (Applied Statistics) vol. 28, no. 1, 1979, pp. 100-108.

[15] R. Alur, R. Bodik, and G. Juniwal, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design IEEE*, 2013, pp. 1-8.

[16] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster, "Path-based inductive synthesis for program inversion," in *proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011, pp. 492-503.

[17] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv, "A simple inductive synthesis methodology and its applications," in *proceedings of the ACM International Conference on Object Oriented Programming Systems, Languages and Applications*, 2010, pp. 36-46.

[18] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, 2011, pp. 317-330.

[19] W. R. Harris and S. Gulwani, "Spreadsheet table transformations from examples," in *proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011, pp. 317-328.

[20] J. R. Quinlan, "Induction of decision trees," in *Machine learning*, vol. 1, no. 1, 1986, pp.81-106.

[21] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *proceedings of 30th International Conference on Machine Learning*, 2013, pp. 1139-1147.

[22] J. A. Suykens, and J. Vandewalle, "Least squares support vector machine classifiers," in *Neural Processing Letters*, vol. 9, no. 3, 1999, pp. 293-300.

[23] G. Tsoumakas, and I. Katakis, "Multi-label classification: An overview," in *International Journal of Data Warehousing and Mining*, vol. 3, no. 3, 2006.

[24] M. Levandowsky, and D. Winter, "Distance between sets." in *Nature*, vol. 234, no. 5323, 1971, pp. 34-35.

[25] A. Solar-Lezama, "Program Synthesis by Sketching", Ph.D thesis, University of California, Berkeley, 2008.

[26] http://ds.iris.edu/ds/nodes/dmc/tools/flowers/

[27] http://archive.ics.uci.edu/ml/datasets/Balance+Scalet/

[28] https://archive.ics.uci.edu/ml/datasets/DSRC+Vehicle+Communications/

[29] http://ds.iris.edu/ds/nodes/dmc/tools/waveforms/

[30] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical machine learning tools and techniques*, Morgan Kaufmann, 2016.

[31] Google auto Waymo disengagement report for autonomous driving. https://www.dmv.ca.gov/portal/ wcm/connect/946b3502-c959-4e3b-b119-91319c27788f/ GoogleAutoWaymodisengagerepport2016.pdf?MOD=AJPERES.

[32] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and Harnessing Adversarial Examples," in *proceedings of the 3rd International Conference on Learning Representations*, 2015.

[33] A. Nguyen, J. Yosinski, and J. Clune, "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images," in *proceedings of the 28th IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 427 - 436.

[34] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, 2001, pp. 99-123.

[35] C. Csallner, N. Tillmann, and Y. Smaragdakis, "DySy: Dynamic symbolic execution for invariant inference," in *proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 281-290.

[36] C. Pacheco, and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *proceedings of the European Conference on Object-Oriented Programming*, 2005, pp. 504-527.

[37] C. Pacheco, and M. D. Ernst, "Randoop: feedback-directed random testing for Java," in *proceedings of Companion to the Conference on Object-oriented Programming Systems and Applications*, 2007, pp. 815-816.

[38] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011, pp. 416-419.

[39] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno, "Privacy oracle: a system for finding application leaks with black box differential testing," in *proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008, pp. 279-288.

[40] A. Pnueli and R. Rosner, "On the synthesis of an asynchronous reactive module," in *proceedings of the 16th International Colloquium on Automata, Languages and Programming*, 1989, pp. 652-671.

[41] J. Wu, Y. Jiang, C. Xu, S. C. Cheung, X. Ma, and J. Lu, "Poster: Synthesizing relation-aware entity transformation by examples," in *proceedings of the 31st International Conference on Software Engineering*, 2018.

[42] Z. Manna and R. Waldinger, "A deductive approach to program synthesis," in *ACM Trans. Program. Lang. Syst.*, vol. 2, no.1, 1980, 90-121.

[43] R. L. Constable, "Implementing Mathematics with the NuPRL Proof Development System," Prentice Hall, 1986.

[44] D. R. Smith, "KIDS: A semiautomatic program development system," in *IEEE Trans. Soft. Eng.*, vol. 16, no. 9, 1990, pp. 1024-1043.

[45] A. Graves, G. Wayne, and I. Danihelka, "Neural Turing machines," arXiv preprint arXiv:1410.5401, 2014.

[46] O, Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Advances in Neural Information Processing Systems*, 2015, pp. 2692-2700.

[47] A. Joulin and T. Mikolov, "Inferring algorithmic patterns with stack-augmented recurrent nets," in *Advances in Neural Information Processing Systems*, 2015, pp. 190-198.

[48] http://archive.ics.uci.edu/ml/datasets.html/