PAPER

Formal Verification of a Decision-Tree Ensemble Model and Detection of Its Violation Ranges

Naoto SATO[†a)], **Hironobu KURUMA**[†], **Yuichiroh NAKAGAWA**[†], *Nonmembers*, *and* **Hideto OGAWA**[†], *Member*

SUMMARY As one type of machine-learning model, a "decision-tree ensemble model" (DTEM) is represented by a set of decision trees. A DTEM is mainly known to be valid for structured data; however, like other machine-learning models, it is difficult to train so that it returns the correct output value (called "prediction value") for any input value (called "attribute value"). Accordingly, when a DTEM is used in regard to a system that requires reliability, it is important to comprehensively detect attribute values that lead to malfunctions of a system (failures) during development and take appropriate countermeasures. One conceivable solution is to install an input filter that controls the input to the DTEM and to use separate software to process attribute values that may lead to failures. To develop the input filter, it is necessary to specify the filtering condition for the attribute value that leads to the malfunction of the system. In consideration of that necessity, we propose a method for formally verifying a DTEM and, according to the result of the verification, if an attribute value leading to a failure is found, extracting the range in which such an attribute value exists. The proposed method can comprehensively extract the range in which the attribute value leading to the failure exists; therefore, by creating an input filter based on that range, it is possible to prevent the failure. To demonstrate the feasibility of the proposed method, we performed a case study using a dataset of house prices. Through the case study, we also evaluated its scalability and it is shown that the number and depth of decision trees are important factors that determines the applicability of the proposed method.

key words: *machine learning, formal verification, decision-tree ensemble model*

1. Introduction

Recently, software developed by machine learning has been used in various systems. Deep learning using deep neural networks (DNNs) is widely used for predicting and classifying image data, audio data [1], [2], and so on. For structured data, ensemble learning methods using decision trees, such as random forests [3] and gradient-boosting decision trees [4], are also effective [5]–[11], [13].

A decision-tree ensemble model (DTEM) is represented as a set of decision trees. A DTEM takes a vector of values—called an "attribute vector"—as input, and an element of the vector is called "attribute value." The DTEM returns a value as output, which is called a "prediction value." The prediction value is calculated as the sum or average value of the scores associated with the leaves of the decision trees. The DTEM is expected to be generalized, namely, to return the appropriate prediction value even if the attribute vector is not included in the training data.

Manuscript received May 2, 2019.
Manuscript revised August 28, 2019.
Manuscript publicized November 20, 2019.

[†]The authors are with the Research & Development Group, Hitachi, Ltd., Yokohama-shi, 244–0817 Japan.
a) E-mail: naoto.sato.je@hitachi.com

DOI: 10.1587/transinf.2019EDP7120

However, in general, it is difficult to train a DTEM to return the appropriate prediction value for every attribute vector, that is, a DTEM returns an inappropriate prediction value with a certain probability. Therefore, in particular, when a DTEM is used in a mission-critical system, whose behavior significantly affects business and society, it is important to comprehensively detect attribute vectors that lead to system failures during development, and take appropriate countermeasures to avoid such failures. Retraining or additional training of the DTEM are possible countermeasures; however, for the reason mentioned above, it is difficult to completely eliminate the possibility of failures occurring. Accordingly, as a practical measure, it is possible to create an input filter to control the attribute vector input into the DTEM and to use separate software to process the filtered attribute vector that leads to failures (Fig. 1). The implementation of the separate software is arbitrary. For example, the separate software might reject the attribute vector and return an error message.

A policing function [12] is also useful for preventing failures. Separated from the DTEM, it checks the prediction value of the DTEM against a certain property at runtime. If the prediction value does not satisfy the property, that is, the prediction value leads to a failure, the policing function rejects the prediction value. However, comparing the input filter and the policing function reveals that the input filter is more efficient because it detects and controls failures before the DTEM runs.

To create the input filter, it is necessary to specify which attribute values should be filtered as the filtering condition. In this paper, we therefore propose a method to formally verify whether a DTEM meets a certain property and, if that property is not met, extracts the range (part of the attribute-vector space) in which all attribute vectors violating the property are included. By setting the range extracted by the proposed method as the filtering condition of the input filter, it is possible to prevent the failure of the system due to the DTEM. A property is defined as a predicate for

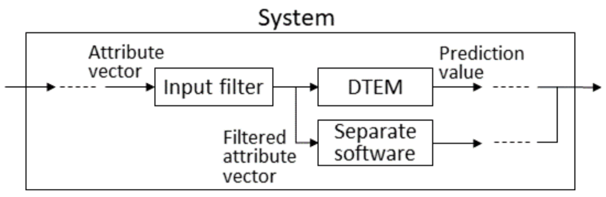


Fig. 1 Input filter

an attribute vector and its corresponding prediction value, or only for the prediction value. A property is thus either violated or satisfied by an attribute vector and its corresponding prediction value, or by the prediction value only. Hereafter, to simplify description, this statement is abbreviated as "a property is violated/satisfied by an attribute vector."

The feasibility of the proposed method is evaluated by showing a case study in which the method was applied experimentally. Moreover, the scalability of the proposed method is evaluated by changing the dimension of the attribute vector, the number of decision trees constituting the DTEM, and the maximum value of the depth of those decision trees. Hereafter, an attribute vector that violates the property is referred to as a "violating instance." Similarly, an attribute vector that satisfies the property is referred to as a "satisfying instance." The range in which a violating instance exists is called the "violation range." It should be noted that both violating instances and satisfying instances are included in the violation range.

As for the proposed method, the decision trees that compose the DTEM are encoded as a formula, and the formula is verified by using a satisfiability modulo theories (SMT) solver. Although this approach has mainly been applied to DNNs [14]–[19], to the authors' knowledge, this approach has not been applied to a DTEM. Given that situation, the first contribution of this paper is to demonstrate that our approach, namely, logically encoding a machine-learning model and verifying the model by solving the resulting formula with an SMT solver, is also applicable to a DTEM.

When a property is violated, as a result of the verification, an example of a violating instance (and its corresponding prediction value)—called a "counterexample"—is obtained. As a naive way to define the filtering condition, all violating instances are detected by repeating the verification. If an attribute vector input into the DTEM matches any violating instances, it is filtered. However, a large number of similar violating instances may exist around a certain violating instance. In particular, if each attribute value is represented by a continuous numerical variable (not a categorical variable), attribute vectors obtained by slightly changing the attribute values of the violating instance are also likely to violate the property. In this case, since the verification is repeated as many times as the number of violating instances, detecting all the violating instances is not practical in terms of calculation time.

Targeting a DTEM taking multi-dimensional attribute vectors whose elements (that is, attribute values) are continuous numerical variables, the proposed method extracts the violation range by searching around the origin at which a violating instance was first detected and gradually expanding the search range until the violating instance is not detected. This method makes it possible to include the violating instances around the origin within the range. However, satisfying instances are also included in the range. Even though they do not violate the property, they are filtered as well as the violating instances.

In consideration of the above-described condition, it is desirable to prevent as many satisfying instances as possible from being included in the violation range. Therefore, as for the proposed method, the extracted violation range is divided into a number of smaller ranges. Then, for each divided range, whether a violating instance exists within the range is checked. As a result, it is possible to narrow down the original violation range. As for the second contribution of this paper, under the assumption that a DTEM takes multi-dimensional attribute vectors of continuous numerical values as input, a method to extract the violation range and narrow it down is proposed. Moreover, the feasibility and scalability of the proposed method are evaluated through a case study.

The rest of this paper is organized as follows. In Sect. 2, a decision tree and a DTEM are formally defined. In Sect. 3, the proposed method is overviewed. In Sect. 4, among the procedures that compose the proposed method, the procedure for verifying the DTEM is explained. In Sect. 5, the procedure for extracting the violation range on the basis of the verification result is explained. In Sect. 6, the procedure for narrowing down by dividing the extracted violation range is explained. In Sect. 7, the feasibility and scalability of the proposed method are evaluated through a case study using a data set of house prices. In Sect. 8, the usefulness and applicability of the proposed method are verified. In Sect. 9, related work is described, and in Sect. 10, the conclusions drawn from this study are presented.

2. Preliminaries

An attribute vector is denoted by x, and a prediction value is denoted by y. A DTEM M can be defined as a function from X to Y, which is denoted by $M : X \longrightarrow Y$. X represents a set of attribute vectors, and Y represents a set of prediction values. If x is included in X, its corresponding y is also included in Y. It is assumed that x is represented by a vector of length $s \geq 2$ such that $[x[0], \ldots, x[k], \ldots, x[s-1]]$. It is also assumed that all the elements of x, that is, $x[0], \ldots, x[k], \ldots, x[s-1]$, are continuous numerical variables although original decision trees can deal with discrete, categorical, or Boolean values as input. It is also supposed that X gives the maximum and minimum values of $x[k]$ ($0 \leq k \leq s-1$), which are denoted by $X[upper][k]$ and $X[lower][k]$, respectively. When x is considered as an input to M, $x[k]$ is upper and lower bounded by X.

First, in Sect. 2.1, the decision trees that compose M are formally defined. Next, in Sect. 2.2, M is formally defined.

2.1 Decision Tree

A decision tree represents a procedure for determining the class to which a given attribute vector belongs [20], [21]. A decision tree consists of decision nodes, edges, and leaf nodes. The conditional expression for an attribute value is associated with the decision node (Fig. 2). In this paper, the conditional expression of the decision node is called an "attribute test." Each decision node has multiple child nodes, and each child node is connected by an edge. Each edge is labeled with the evaluation result of the attribute test. When an attribute value is given, the edge with the same label as the result of evaluating the attribute test is selected. Hereafter, the evaluation result of the attribute test is called the "test value." If the child node connected to the selected edge is a decision node, the same procedure as described above is performed for that decision node. If the child node is a leaf node, the value associated with that leaf node becomes the name of the class to which the attribute vector belongs. As for the decision trees making up the DTEM, numerical values—called "decision values"—are used as class names.

An arbitrary decision tree can be expressed in the form $(N_d, N_l, n_1, E, attr, tv, dv)$, where N_d represents a set of decision nodes, N_l represents a set of leaf nodes, and n_1 represents a root node included in N_d. E is a set of edges, each of which edge is represented by a pair consisting of a connection-source node and a connection-destination node. That is, it can be defined as $E \subseteq N_d \times (N_l \cup (N_d \setminus \{n_1\}))$. Here, $attr : N_d \longrightarrow A$ is a function that associates an attribute test with a decision node. A represents a set of arbitrary expressions for attribute $x[0], \ldots, x[k], \ldots, x[s-1]$. Similarly, $tv : E \longrightarrow V$ is a function that associates a test value with an edge, where V represents a set of test values. In the example in Fig. 2, $V = \{True, False\}$. Here, $dv : N_l \longrightarrow \mathbb{R}$ is a function that associates a decision value with a leaf node. The decision tree is acyclic and defined as $\forall n \in N_d \cup N_l \cdot \langle n, n\rangle \notin E^+$ by using transitive closure E^+ of E.

It is assumed that the decision tree that constitutes the DTEM, which is the subject of this paper, has more than one decision node, and branches from a decision node to at least two child nodes. Therefore, if the number of nodes included in N_d is expressed as $card(N_d)$, $card(N_d) \geq 1$ holds. In a similar manner, $card(N_l) \geq 2$ also holds. As for the number of edges, $card(E)$, $card(E) \geq 2$ holds.

2.2 Decision-Tree Ensemble Model (DTEM)

Arbitrary M can be expressed as $(T, ensem)$. T represents

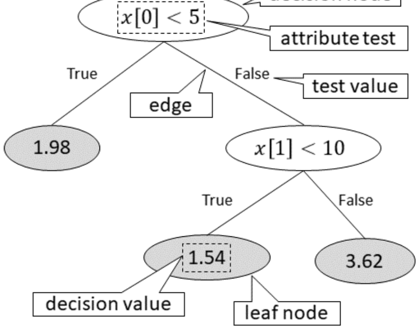


Fig. 2 Structure of decision tree

a set of decision trees constituting M. If the number of decision trees included in T is taken as $card(T)$, the decision trees included in T can be expressed as $t_1, \ldots, t_i, \ldots, t_{card(T)}$. The decision values of trees $t_1, \ldots, t_i, \ldots, t_{card(T)}$ are represented as $y_1, \ldots, y_i, \ldots, y_{card(T)}$, respectively.

The decision values are used to calculate prediction value y. Thus, a function $ensem$ that takes $y_1, \ldots, y_{card(T)}$ as arguments and returns y is introduced here. The specification of $ensem$ depends on the implementation of DTEM. For example, in the case of random forests, the average of the decision values is y. Alternatively, in the case of a gradient-boosting decision tree, the sum of the decision values is y. In this paper, function $ensem$ is not specified.

3. Outline of Proposed Method

The proposed method is outlined in Algorithm 1. φ denotes a property of DTEM M. Parameters r_a, r_b, and r_c are described later. Property φ is defined as a predicate for x and y, or for y only. Algorithm 1 returns the value of variable $vranges$, which is a set of violation ranges for φ. On line 1, function dom returns domain X of M. On line 3, function $within$ returns formula ρ, meaning that "x is included in X." On line 5, procedure $formal\_verification$ verifies whether there is an attribute vector that satisfies constraint ρ and violates φ. If such an attribute vector is detected, it is assigned to variable ce as a violating instance. If no such attribute vector is detected, "None" is assigned to ce. Procedure $formal\_verification$ is described in detail in Sect. 4. On line 7, procedure $range\_extraction$ extracts the violation range for φ from around ce. The extracted range is assigned to variable vio. Moreover, a range in which no violating instance exists is extracted along with vio; therefore, a set of such ranges is returned as variable $novios$. Hereafter, a

Algorithm 1 Violation Ranges Detection

```
Input: $M, \varphi, r_a, r_b, r_c$
Output: $vranges$

1:  $X \leftarrow dom(M)$
2:  $vranges \leftarrow \emptyset$
3:  $\rho \leftarrow within(X)$
4:  while True do
5:      $ce \leftarrow formal\_verification(M, \varphi, \rho)$
6:      if $ce \neq$ None then
7:          $vio, novios \leftarrow range\_extraction(ce, M, \varphi, \rho, r_a, X)$
8:          $core \leftarrow vio$
9:          while $continue\_division(core, X, r_b)$
10:                                          $\wedge\ novios \neq \emptyset$ do
11:             $iv \leftarrow novios$.pop()
12:             $core, surrds \leftarrow range\_division(core, iv, \rho, r_c, M, \varphi)$
13:             $vranges \leftarrow vranges \cup surrds$
14:         end while
15:         $vranges \leftarrow vranges \cup core$
16:         $\rho \leftarrow \rho \wedge outside(vio)$
17:     else
18:         break
19:     end if
20: end while
21: return $vranges$
```

range in which no violating instance exists is referred to as a "no-violation range." Procedure *range_extraction* is shown in detail in Sect. 5.

The extracted violation range, *vio*, is divided as follows. First, on line 8, *vio* is copied to variable *core*. On line 9, *continue_division* implements the condition to stop dividing the violation range. As long as it returns "True", *range_division* is executed iteratively. Parameter r_b is used as a reference value to decide whether to execute *range_division* again. Although the details of *continue_division* are not specified here, for example, it can be implemented so as to return "True" if the volume of *core* is equal to or more than r_b percent of the volume of X. On line 12, procedure *range_division* divides *core* into multiple pieces. This division is performed on the basis of a no-violation range, which is an element of *novios*. Of the ranges obtained by division, the inner range including *ce* (which is the starting point of range extension) is taken as a new *core*, and the set of other outer ranges is taken as *surrds*. The *range_division* is explained in detail in Sect. 6.

If *novios* contains a number of no-violation ranges, the range pushed to *novios* last (which is the outermost no-violation range in terms of *ce*) is used for the division. Therefore, the ranges stored in *surrds* do not include no-violation ranges because they are outside the outermost no-violation range. Accordingly, the elements of *surrds* are added to *vranges* without dividing them further (line 13). On the contrary, the new *core* might include a no-violation range. Therefore, on line 9, whether *core* will be divided further is evaluated by *continue_division*. If *core* is not divided any more, it is added to *vranges*.

On line 16, function *outside* is used to create a constraint representing "x is out of the range indicated by *vio*.", and that constraint is conjunctively appended to ρ. After that, by re-executing *formal_verification* on line 5 in the while loop, whether there is a violating instance outside *vio* is verified. If a violating instance is detected as a result of the re-execution, another violation range is extracted and divided by the same procedure. In a similar manner, several different violation ranges are extracted until the while loop is ended. When *formal_verification* returns "None" on line 5, the while loop is interrupted by the break statement on line 18. This means that no violating instance is found outside the violation ranges extracted until then. Therefore, it is guaranteed that the extracted violation ranges include all the violating instances in X when Algorithm 1 terminates. *vranges* extracted by Algorithm 1 represents the range in which a violating instance exists. Therefore, by creating an input filter (shown in Fig. 1) based on *vranges*, it is possible to filter the violating instance.

4. Formal Verification of Decision-Tree Ensemble Model

The procedure of *formal_verification* in Algorithm 1 is explained in detail hereafter. For any decision tree $t_i \in T$ that constitutes DTEM M, a set of all paths extractable from t_i is denoted by P_i. As stated in Sect. 2.1, since a decision tree is assumed to be acyclic, the elements of P_i are paths of finite length. Arbitrary path $p \in P_i$ can be represented by a sequence of nodes, $n_1^p, n_2^p, \ldots, n_d^p, n_{d+1}^p$. Among these nodes, $n_1^p, \ldots, n_d^p$ represent decision nodes, and n_{d+1}^p represents a leaf node. Since t_i has one or more decision nodes, $1 \leq d$ holds. Here, the specification of arbitrary path p of t_i is defined as f_i^p by using functions *attr*, *tv*, and *dv* (defined in Sect. 2.1) as follows:

$$f_i^p \stackrel{\text{def}}{=} \left(\bigwedge_{j=1}^{d} attr(n_j^p)(x) = tv(\langle n_j^p, n_{j+1}^p \rangle) \right) \\ \rightarrow (y_i = dv(n_{d+1}^p)) \tag{1}$$

The argument $\langle n_j^p, n_{j+1}^p \rangle$ of function *tv* represents an edge between nodes n_j^p and n_{j+1}^p. The specification of t_i is represented by F_i as

$$F_i \stackrel{\text{def}}{=} \bigwedge_{p \in P_i} f_i^p \tag{2}$$

As explained in Sect. 2.2, prediction value y of M is calculated by *ensem* from decision values $y_1, \ldots, y_i, \ldots, y_{card(T)}$. Therefore, the specification of M is expressed by F_M as

$$F_M \stackrel{\text{def}}{=} \left(\bigwedge_{i=1}^{card(T)} F_i \right) \\ \wedge (y = ensem(y_1, \ldots, y_i, \ldots, y_{card(T)})) \tag{3}$$

Furthermore, F is defined by using φ (i.e., a property of M) and constraint ρ (which indicates that x is included in X) as

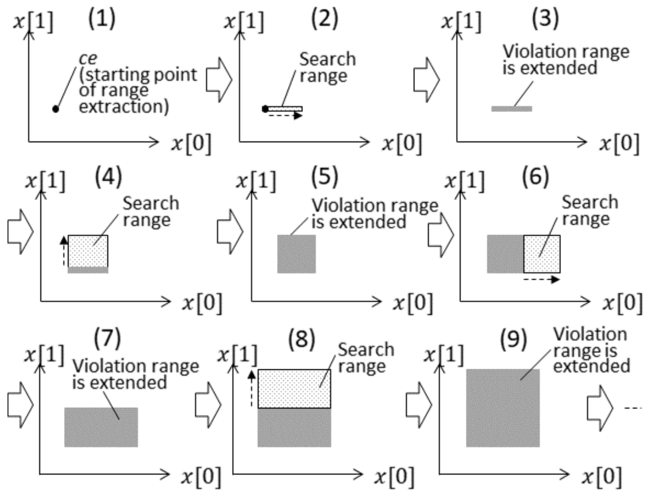
$$F \stackrel{\text{def}}{=} F_M \wedge \neg\varphi \wedge \rho \tag{4}$$

F is input into the SMT solver to determine its satisfiability. If F is unsatisfiable, it is guaranteed that there is no violating instance in the range constrained by ρ. If F is satisfiable, the SMT solver detects x and y satisfying F. Then, procedure *formal_verification* returns detected x as the violating instance, *ce*. It is thus possible to verify whether M meets φ by encoding M as formula F and solving the satisfiability problem by using the SMT solver.

5. Extraction of Violation Range

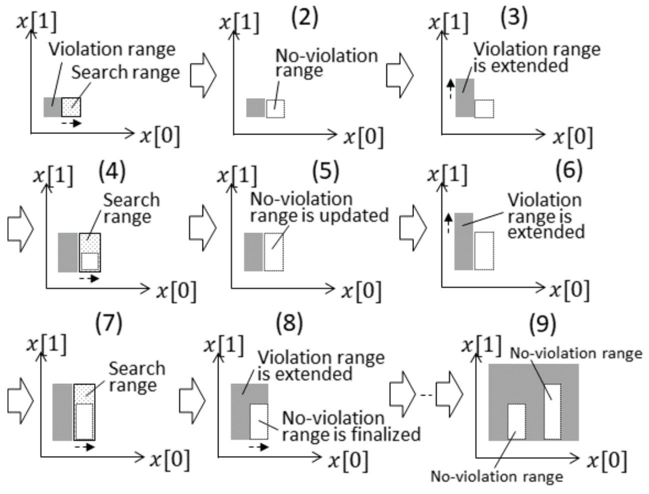
5.1 Overview

As for *range_extraction*, to find a violating instance as a starting point, the range around violating instance *ce* is searched by using *formal_verification*. If a violating instance is detected, the range is extracted as the violation range. Here, to promote intuitive understanding, it is assumed that variable x is represented by a two-dimensional vector, namely, $[x[0]], x[1]]$. The method for extracting the violation range by *range_extraction* is outlined in Fig. 3. The violation range is extended by alternately increasing the values of $x[0]$ and $x[1]$. In step (1), the violating instance obtained on line 5 of Algorithm 1 is set as the initial value of



(1)
ce (starting point of range extraction)
(2)
Search range
(3)
Violation range is extended
(4)
Search range
(5)
Violation range is extended
(6)
Search range
(7)
Violation range is extended
(8)
Search range
(9)
Violation range is extended
x[0]
x[1]

Fig. 3 Extraction of violation range by *range_extraction*



Violation range
Search range
(2)
No-violation range
(3)
Violation range is extended
(4)
Search range
(5)
No-violation range is updated
(6)
Violation range is extended
(7)
Search range
(8)
Violation range is extended
No-violation range is finalized
(9)
No-violation range
No-violation range
x[0]
x[1]

Fig. 4 Extraction of no-violation range by *range_extraction*

the violation range. In step (2), a search range is set from the initial value in the upward direction of $x[0]$, and whether a violating instance exists within that search range is verified by the method described in Sect. 4. If a violating instance is detected in the search range, as shown in step (3), the search range is imported into the violation range. In step (4), the search range is set in the upward direction of $x[1]$. From then on, to extend the violation range, the same procedure is repeated. Although omitted in Fig. 3, in a similar manner as described above, the violation range is also extended in the downward directions of $x[0]$ and $x[1]$.

As for *range_extraction*, when the violation range is extracted, no-violation ranges existing in that range are also extracted. The method of extracting no-violation ranges is outlined in Fig. 4. In step (1) of Fig. 4, the search range is set in the upward direction of $x[0]$. It is assumed that no violating instance exists in the search range. In that case, the no-violation range is created as shown in step (2). After the violation range is extended in the upward direction of $x[1]$ in step (3), the search range is reset in the upward direction of $x[0]$ in step (4). If a violating instance still does not exist in the search range, the no-violation range set in step (2) is extended by step (5). It is assumed that the violation range can be extended in steps (6), (7), and (8). In that case, the extension of the no-violation range is ended. Accordingly, when no violating instance exists in the search range, the search range is extracted as the no-violation range. As shown in step (9), multiple no-violation ranges may be created finally. One of the features of the proposed method is to extract this no-violation range. The extracted no-violation range is the utilized in *range_division* described in Sect. 6.

As for the strategy for extending the violation range, the values of $x[0]$ and $x[1]$ are alternately increased. Other strategies for extending the violation range, however, may be considered. For example, the values of $x[0]$ and $x[1]$ can be increased simultaneously. Strategies for extending the violation range are discussed in Sect. 8.

5.2 Algorithm

The procedure of *range_extraction* is shown in detail in Algorithm 2. Algorithm 2 returns *vio*, which represents a violation range for φ, and *novios*, which represents a range in which no violating instance exists. The violation range can be defined by the lower-limit values and the upper-limit values of variables $x[0], \ldots, x[s-1]$. It is therefore supposed that violation range *vio* is composed of *vio*[*lower*] and *vio*[*upper*], where *lower* and *upper* represent 0 and 1 respectively. The lower-limit value of each variable $x[k]$ ($0 \leq k \leq s-1$) is represented by *vio*[*lower*][k]. Similarly, the upper-limit value is represented by *vio*[*upper*][k].

On lines 2 and 3, *vio* is initialized with *ce*. On line 10, whether *vio* has been extended in the previous loop is checked, and if *vio* has been extended, lines 13 and 14 try to extend *vio* further. On line 12, the variable for which the range is to be expanded, $x[k]$, is selected. For the selected $x[k]$, line 13 attempts to expand the upper limit of the variable, and line 14 tries to expand the lower limit of the variable. If neither the upper limit nor the lower limit can be expanded for all variables $x[0], \ldots, x[s-1]$, $cont\_flag =$ False is returned, and the procedure ends.

On lines 13 and 14, the upper and lower limits are extended by EXPAND. The procedure of EXPAND is described below in the case that the upper limit is expanded. Parameter *dir* of EXPAND represents the direction of expansion. That is, in this example, *upper* is passed as an argument. On line 22, whether the violation range can be expanded in the upward direction of $x[k]$ is checked. As described in Sect. 2, $X[upper][k]$ is the maximum value of $x[k]$. If *vio*[*upper*][k] reaches $X[upper][k]$, the upper limit of the violation range is not further expanded in the upward direction of $x[k]$. In that case, line 23 returns "False". On line 25, variable *sr* is initialized with *vio*, where *sr* represents the search range. On line 27, *sr*[*upper*][k] is updated to *vio*[*upper*][k] + $mgn(r_a)[k]$, where *mgn* is a function that accepts parameter r_a as an argument and returns a vector of the same length as x. Since the search range

Algorithm 2 Algorithm of *range_extraction*

```
Input: $ce, M, \varphi, \rho, r_a$
Output: $vio, novios$

1: $X \leftarrow dom(M)$
2: $vio[lower] \leftarrow ce$
3: $vio[upper] \leftarrow ce$
4: $novios \leftarrow \emptyset$
5: for each $k$ in $\{0, \dots, s-1\}$ do
6:     $tmp\_nv[lower][k] \leftarrow$ None
7:     $tmp\_nv[upper][k] \leftarrow$ None
8: end for
9: $cont\_flag \leftarrow$ True
10: while $cont\_flag$ = True do
11:     $cont\_flag \leftarrow$ False
12:     for each $k$ in $\{0, \dots, s-1\}$ do
13:         $rtn\_upper \leftarrow$ EXPAND($upper, k, vio, tmp\_nv, \rho, M, \varphi, X$)
14:         $rtn\_lower \leftarrow$ EXPAND($lower, k, vio, tmp\_nv, \rho, M, \varphi, X$)
15:         if $rtn\_upper$ = True $\vee$ $rtn\_lower$ = True then
16:             $cont\_flag \leftarrow$ True
17:         end if
18:     end for
19: end while
20: return $vio, novios$

21: procedure EXPAND($dir, k, vio, tmp\_nv, \rho, M, \varphi, X$)
22:     if $vio[dir][k] = X[dir][k]$ then
23:         return False
24:     end if
25:     $sr \leftarrow vio$
26:     if $dir = upper$ then
27:         $sr[dir][k] \leftarrow vio[dir][k] + mgn(r_a)[k]$
28:         if $sr[dir][k] > X[dir][k]$ then
29:             $sr[dir][k] \leftarrow X[dir][k]$
30:         end if
31:     else // $dir = lower$
32:         $sr[dir][k] \leftarrow vio[dir][k] - mgn(r_a)[k]$
33:         if $sr[dir][k] < X[dir][k]$ then
34:             $sr[dir][k] \leftarrow X[dir][k]$
35:         end if
36:     end if
37:     $sr[opposite(dir)][k] \leftarrow vio[dir][k]$
38:     $\rho' \leftarrow \rho \wedge within(sr)$
39:     $ce' \leftarrow formal\_verification(M, \varphi, \rho')$
40:     if $ce' \neq$ None then
41:         $vio[dir][k] \leftarrow sr[dir][k]$
42:         if $tmp\_nv[dir][k] \neq$ None then
43:             $novios$.push($tmp\_nv[dir][k]$)
44:             $tmp\_nv[dir][k] \leftarrow$ None
45:         end if
46:         return True
47:     else
48:         $tmp\_nv[dir][k] \leftarrow sr$
49:         return False
50:     end if
51: end procedure
```

is bounded by X, $sr[upper][k]$ should be smaller than or equal to $X[upper][k]$. Therefore, $sr[upper][k]$ is changed to $X[upper][k]$ if it is greater than $X[upper][k]$ (lines 28 and 29). On line 37, $sr[lower][k]$ is updated to $vio[upper][k]$. *opposite* is a function that returns the direction opposite to the extension direction indicated by *dir*. Accordingly, the search range of $x[k]$ is set to the range $vio[upper][k] < x[k] \leq (vio[upper][k] + mgn(r_a)[k])$ (if $vio[upper][k] + mgn(r_a)[k]$ is smaller than or equal to $X[upper][k]$). Because the search range of variable $x[k']$ ($k' \neq k$) is not updated, it is given as $vio[lower][k'] \leq x[k'] \leq vio[upper][k']$. As a result, *sr* is created on the upper bound of *vio* in the $x[k]$ direction.

The upper-limit or lower-limit value of *sr*, that is, $sr[upper][k]$ or $sr[lower][k]$ is sometimes not included in the search range. Accordingly, it is necessary to hold that information about whether each $sr[upper][k]$ and $sr[lower][k]$ are included in the search range. To simplify the following explanation, how to keep that information in *sr* is omitted.

On line 38, ρ' is created by conjunctively appending the constraint "x is included in *sr*" to ρ. Line 39 verifies whether M meets φ under constraint ρ'. As a result of that verification, if a violating instance is detected within *sr*, the upper limit of the violation range, $vio[upper]$, is extended to $sr[upper]$ (line 41).

Lines 42, 44, and 48 create an element of *novios*. If no violating instance is detected in the search range as a result of the verification on line 39, the search range is saved in $tmp\_nv[upper][k]$ (line 48). $tmp\_nv[upper][k]$ stores a tentative no-violation range in the upward direction of $x[k]$. The range stored in $tmp\_nv[upper][k]$ will be extended as long as a violating instance is not detected in the upward direction of $x[k]$ (line 48). If a violating instance is detected in that direction, the latest range stored in $tmp\_nv[upper][k]$ is finalized as a no-violation range and added to *novios* (line 43).

6. Division of Violation Ranges

6.1 Overview

By *range_extraction* shown as Algorithm 2, violation range *core* can be extracted. However, it may include satisfying instances as well as violating instances. In other words, the violation range may be redundant. If an input filter based on the violation range is created, it unnecessarily filters satisfying instances. It is thus desirable to prevent as many satisfying instances as possible from being included in the violation range. To reduce the redundancy of the violation range, an additional procedure is required. Procedure *range_division* aims to narrow down the violation range by dividing it. It has not been theoretically proved that every no-violation range to be removed from the violation range can always be found. Neither is it guaranteed that the violation range can always be minimized. Even if the violation range is redundant, it is possible to create the input filter because the violation range includes all violating instances. At that time, it can be said that the smaller the violation range, the fewer satisfying instances are filtered. Hence, it is better to try to narrow down the violation range than do nothing. Through the case study described in Sect. 7, it is shown that *range_division* is useful for narrowing down the violation range.

As for *range_division*, violation range *core* extracted by *range_extraction* is divided into a number of smaller

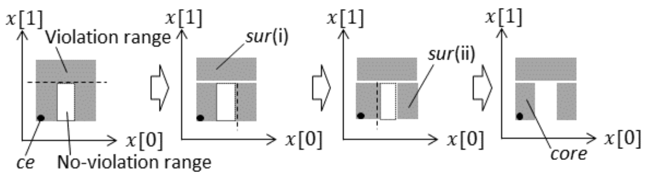


Fig. 5 Violation range divided by *range_division*

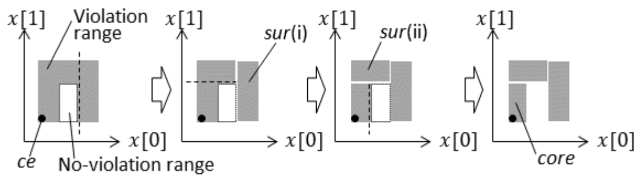


Fig. 6 Another order of dividing based on no-violation range

ranges. The division is based on a no-violation range, which is an element of *novios*. Among the divided ranges, the range including violating instance *ce* (which is the starting point of the range extension) is set as a new *core*, and the other ranges are called *sur*s. If *sur* includes a violating instance, it is added to *surrds*. Since it is possible to repeatedly divide *core*, the number of ranges in *surrds* may be increased with each division. For example, as for the violation range shown in step (9) in Fig. 4, the method of *range_division* is outlined in Fig. 5. In Fig. 5, first, the violation range is divided at the upper-limit value of the no-violation range on the $x[1]$ axis. Next, the range is divided at the upper-limit value of the no-violation range on the $x[0]$ axis. Finally, the range is further divided by using the lower-limit value of the no-violation range on the $x[0]$ axis. As a result of this division, the violation range is divided into three ranges: *sur*(i), *sur*(ii), and *core*.

It is clear that there exists a violating instance in *core* because *core* includes violating instance *ce* at least. *sur*(i) also includes another violating instance since the violation range was extended in step (8) in Fig. 4. However, *sur*(ii) may not include a violating instance. Whether *sur*(ii) includes a violating instance can be checked by using *formal_verification*. If *sur*(ii) does not include a violating instance, it is removed from the violation range.

The method of *range_division* aims to narrow the violation range by dividing it and verifying whether each outer range *sur* created by the division include violating instances. This is based on the assumption that the outer ranges of the no-violation range are not likely to include a violating instance. In relation to the starting point of the range extension, the outer ranges are defined as "adjacent to" and "beyond" the no-violation range. Accordingly, an attribute vector in the outer ranges is assumed to be "similar to" the attribute vector in the no-violation range and "more different" from the attribute vectors around the starting point than the attribute vector in the no-violation range. It is therefore considered that outer ranges created by the division on the basis of the no-violation range are not likely to include a violating instance.

When the violation range is divided on the basis of the no-violation range, the order of the division can be changed. For example, the division order shown in Fig. 6 can be considered. In this case, the division is performed in the following order: at the upper-limit value of the no-violating range on the $x[0]$ axis, at the upper-limit value on the $x[1]$ axis, and at the lower-limit value on the $x[0]$ axis.

It is said that the division order that makes the violation range narrower is more effective. However, the extent to which the violation range can be narrowed by a certain division order depends on the DTEM and property to be verified. It is therefore not known which division order is effective unless the division is actually performed. Accordingly, as for the proposed method, multiple divisions are tried at random, and the order that makes the violation range the narrowest is adopted.

The violation range is divided by the surface of the no-violation range (corresponding to the dotted lines in Fig. 5 and Fig. 6). Since the no-violation range is an s-dimensional hyperrectangle, the number of dividing planes is $2s$. Therefore, the number of ways to divide the violation range is $(2s)!$ As for the proposed method, divisions to be tried, whose number equals parameter r_c, are randomly selected and the results of the divisions are compared according to the sum of the hypervolumes of the ranges in *surrds*. The division that minimizes the sum of the hypervolumes is adopted

Algorithm 3 Algorithm of *range_division*

```
Input: $core, iv, \rho, r_c, M, \varphi$
Output: $core, surrds$

1:  $planes \leftarrow calc\_planes(iv)$
2:  $orders \leftarrow permutate(plane, r_c)$
3:  $results \leftarrow \emptyset$
4:  for each $od$ in $orders$ do
5:      $core \leftarrow core$
6:      while $od \neq \emptyset$ do
7:          $plane \leftarrow od$.pop()
8:          $core, sur \leftarrow slice\_core(core, plane)$
9:          $\rho' \leftarrow \rho \wedge within(sur)$
10:         $ce' \leftarrow formal\_verification(M, \varphi, \rho')$
11:         if $ce' \neq$ None then
12:             $surrds \leftarrow surrds \cup sur$
13:         end if
14:     end while
15:     $volume \leftarrow calc\_volume\_sum(core, surrds)$
16:     $results \leftarrow results \cup \{((core, surrds), volume)\}$
17: end for
18: $(core, surrds) \leftarrow min\_volume(results)$
19: return $core, surrds$
```

6.2 Algorithm

The procedure of *range_division* is shown in details in Algorithm 3. In Algorithm 3, violation range *core* is divided on the basis of no-violation range *iv*. Then, as a result of the division, the new *core* and *surrds* are returned. On line 1, *calc_planes* extracts the $(s-1)$-dimensional hyperplanes constituting hyperrectangle *iv* and creates set *planes*. The number of elements in *planes* is $2s$. On line 2, *permutate* creates as many arbitrary division orders by using elements of *planes* as r_c, and the set of the created division orders is assigned to *orders*. Line 4 picks out arbitrary element *od* of *orders*. *od* is an array (with length of $2s$) having the hyperplanes constituting *iv* as elements. On line 7, the top element of *od* is assigned to *plane*. By *slice_core* on line 8, *core* is divided with *plane* as the dividing plane. Of the two ranges created by the division, the range that includes *ce* is taken as the new *core*, and the other is taken as *sur*. *formal_verification* on line 10 verifies whether the violating instance is included in the range of that *sur*. If a violating instance exists within the range of *sur*, *sur* is added to *surrds* (line 12). The division with *plane* is repeated for each element of *od*. On line 15, *calc_volume_sum* is used to calculate the sum of the hypervolumes of the violation range represented by the elements of *core* and *surrds*. The procedures from lines 5 to 16 are repeated according to the number of elements of *orders*, that is, r_c times. In this manner, for each division order randomly created on line 2, *core* and *surrds*, which are the results of the division, and *volume*, which is the sum of the hypervolumes, are obtained. On line 18, the result of the division that gives the smallest *volume* is extracted by function *min_volume* and used as the final return value.

In this way, the violation range is divided by *range_division* on the basis of the no-violation range created by *range_extraction*. It should be noted that *range_division* is not executed if the no-violation range is not created by *range_extraction*.

7. Case Study

7.1 Setup

As the subject of the case study, DTEM M is created by using a dataset of house prices[†] as a training data set. To implement M, XGBoost [13] was used. M receives a 7-dimensional attribute vector, $x = [x[0], \dots, x[6]]$, and returns a house price as prediction value y, where $x[0], \dots, x[6]$ are, respectively, grade of house, condition of house, number of bedrooms, size of living room, size of parking space, size of ground floor, and size of basement. The number of decision trees constituting M is 100, and the maximum depth of each decision tree is 3. As properties to be verified, φ_1, φ_2, and φ_3 are defined as follows:

$$\varphi_1 \overset{\text{def}}{=} x[0] \geq 7000 \Rightarrow y \geq 500000 \tag{5}$$

$$\varphi_2 \overset{\text{def}}{=} y > 50000 \tag{6}$$

$$\varphi_3 \overset{\text{def}}{=} y < 10000000 \tag{7}$$

Here, $x[0]$ in φ_1 represents size of living room. The larger the living room, the higher the price. Therefore, it is defined as φ_1 that if the size of the living room is 7000 or more, the price is 500,000 or more. Moreover, the price output from M is expected to be realistic in regard to a house price. Accordingly, φ_2 is defined as the property that the price is higher than 50,000. Similarly, φ_3 is taken as the property that the price is less than 10,000,000.

In this case study, X is defined on the basis of the maximum and minimum values of the training dataset. For each variable $x[k]$ $(0 \leq k \leq 6)$, the maximum value included in the training dataset is represented as max_k, and the minimum value is similarly represented as min_k. These values are used to define X as follows:

$$\begin{aligned} X \overset{\text{def}}{=} \{&[x[0], \dots, x[k], \dots, x[s-1]] \\ &\mid \forall k \cdot min_k \leq x[k] \leq max_k\} \end{aligned} \tag{8}$$

Function *mgn* (which determines the search range in *range_extraction*) is implemented with parameter r_a as an argument as follows:

$$mgn(r_a) \overset{\text{def}}{=} [m[0], \dots, m[k], \dots, m[s-1]], \text{ such that} \\ \begin{cases} \forall k \cdot m[k] = (max_k - min_k)/r_a & (x[k] \in \mathbb{R}) \\ \forall k \cdot m[k] = ceil((max_k - min_k)/r_a) & (x[k] \in \mathbb{Z}) \end{cases} \tag{9}$$

Here, $mgn(r_a)[k]$ is created on the basis of the width of X (i.e., the difference between max_k and min_k), where function *ceil* rounds up a real number to an integer. In this case study, $r_a = 100$ is supposed. *continue_division* shown in Algorithm 1 is implemented so as to return "True" if the hypervolume of *core* is r_b percent or more of the hypervolume of X. In this case study, $r_b = 10$ is also supposed. Furthermore, parameter r_c for determining the number of elements of *orders* in *range_division* is taken as $r_c = 10$.

The proposed method was implemented in Python[††]. The Z3 Theorem Prover [36] was used as a SMT solver for *formal_verification*. Moreover, this case study was performed on a Windows 10® PC equipped with two Intel® Core™ i7-8700 3.2-GHz processors with six cores and with 16-GB memory.

7.2 Results and Evaluation

The results of applying the proposed method are listed in Table 1. The violation range detected by the proposed method is shown in column (b) for each property shown in column (a). Column (b) shows the violation range before division by *range_division* and the violation range after division. Column (c) shows the hypervolume of each violation range. Column (d) shows the value obtained by dividing the sum of hypervolumes of the violation range after division by the sum of hypervolumes of the violation range before division. Violation ranges include both violating instances and satisfying instances. Since all the violating instances are included in any violation range and their number is fixed, the sum of hypervolumes indicates

[†]https://www.kaggle.com/harlfoxem/housesalesprediction

[††]Available at https://github.com/hitachi-rd-yokohama/deep_saucer/tree/master/xgb_encoding

Table 1 Violation ranges extracted by the proposed method

(a) Property	(b) Violation range									(c) Hyper volume	(d) Hypervolume sum after division / hypervolume sum before division	(e) Total run time (s)	(f) SMT solver run time (s)	(g) Number of executions of SMT solver	(h) Average execution time per run of SMT solver (s)
φ_1	Before division	#1-1	6977 ≤ x[0] ≤ 13540	0 ≤ x[1] ≤ 33	520 ≤ x[2] ≤ 1651359	1 ≤ x[3] ≤ 4	1 ≤ x[4] ≤ 9	290 ≤ x[5] ≤ 9410	0 ≤ x[6] ≤ 4820	3.77E+20	0.74	1063.66	1061.14	822	1.29
	After division	#1-1-1	6977 ≤ x[0] ≤ 13540	0 ≤ x[1] ≤ 33	520 ≤ x[2] ≤ 1651359	3 < x[3] ≤ 4	1 ≤ x[4] ≤ 9	6542 < x[5] ≤ 9410	0 ≤ x[6] ≤ 4820	3.95E+19					
		#1-1-2	6977 ≤ x[0] < 7110	0 ≤ x[1] ≤ 33	520 ≤ x[2] ≤ 1651359	1 ≤ x[3] ≤ 3	1 ≤ x[4] ≤ 9	6542 < x[5] ≤ 9410	0 ≤ x[6] ≤ 4820	1.60E+18					
		#1-1-3	7110 ≤ x[0] ≤ 13540	0 ≤ x[1] ≤ 33	520 ≤ x[2] ≤ 1651359	1 ≤ x[3] ≤ 3	1 ≤ x[4] ≤ 9	290 ≤ x[5] ≤ 9410	2459 < x[6] ≤ 4820	1.21E+20					
		#1-1-4	7110 ≤ x[0] ≤ 13540	0 ≤ x[1] ≤ 33	520 ≤ x[2] ≤ 1651359	1 ≤ x[3] ≤ 3	8 < x[4] ≤ 9	5990 < x[5] ≤ 9410	0 ≤ x[6] ≤ 2459	5.89E+18					
		#1-1-5	7110 ≤ x[0] ≤ 13540	0 ≤ x[1] ≤ 33	520 ≤ x[2] ≤ 1651359	1 ≤ x[3] ≤ 3	1 ≤ x[4] ≤ 8	290 ≤ x[5] ≤ 9410	0 ≤ x[6] ≤ 2459	1.10E+20					
φ_2	Before division	#2-1	290 ≤ x[0] ≤ 2101	0 ≤ x[1] ≤ 33	520 ≤ x[2] ≤ 39838	1 ≤ x[3] ≤ 5	1 ≤ x[4] ≤ 7	290 ≤ x[5] ≤ 1812	2299 ≤ x[6] ≤ 4504	1.89E+17		710.31	706.65	1097	0.64
		#2-2	7283 ≤ x[0] ≤ 7948	7 ≤ x[1] ≤ 33	38221 ≤ x[2] ≤ 1651359	1 ≤ x[3] ≤ 3	1 ≤ x[4] ≤ 7	6585 ≤ x[5] ≤ 9410	2495 ≤ x[6] ≤ 4504	1.90E+18					
	After division	Division not performed													
φ_3	No violation range detected											1.53	0.94	1	0.94

how many satisfying instances are included in the violation ranges. Hence, the smaller the sum of hypervolumes is, the fewer satisfying instances are filtered by the input filter. That is, a smaller hypervolume is more practical. The value in column (d) is therefore used to evaluate the effect of narrowing the violation range by *range_division*. Column (e) shows the time elapsed by executing the proposed method. Column (f) shows the time taken to execute the SMT solver. Column (g) shows the number of times of the SMT solver is executed (i.e., number of calls), and column (h) shows the result of dividing (f) by (g), that is, the average execution time per run of the SMT solver. As a result of executing $formal\_verification$ for φ_1, a violating instance was detected. Then, starting from the detected violating instance, *range_extraction* was executed, and the violation range shown in #1-1 was extracted. Moreover, by dividing this violation range by *range_division*, the violation ranges shown in #1-1-1 to #1-1-5 were obtained. This division reduced the hypervolume of the violation range to about 74 percent of the hypervolume before the division.

Similarly, with respect to φ_2, the violation ranges shown in #2-1 and #2-2 were extracted by *range_extraction*. In this case, they were not divided by *range_division* because the hypervolume of each violation range was less than r_b = 10 % of the hypervolume of X. However, it was confirmed that when r_b is 0.1, *range_division* reduces the hypervolume of the violation range to about 33% of the violation range before the division. As for φ_3, no violating instance was detected. That is, it was confirmed that M meets φ_3.

The above results demonstrate that the DTEM can be formally verified by the method described as $formal\_verification$, that is, encoding the DTEM as a formula and solving it with an SMT solver. Moreover, it was confirmed that the violation range can be extracted by *range_extraction*. It was also confirmed that using *range_division* makes it possible to narrow the violation range. It can therefore be concluded that the feasibility of the proposed method was demonstrated.

Scalability of the proposed method is evaluated next. If total execution time and execution time of the SMT solver are focused on, it becomes clear that the time taken to execute the SMT solver occupies most of the total execution time. In other words, the execution time of the proposed method is considered to be largely dependent on the execution time of the SMT solver. As a factor that affects the execution time of the SMT solver, for example, the number and depth of the decision trees constituting M can be considered. However, various heuristics are implemented and black-boxed in the SMT solver, so it is difficult to predict how much these factors affect the execution time of the SMT solver. Accordingly, in this study, the value of the factor considered to affect the execution time of the SMT solver was changed, and the execution time was measured. The following experiment uses the same settings as described in Sect. 7.1 unless otherwise stated. Moreover, a practically acceptable execution time was assumed as 24 hours, and the execution was aborted if it exceeds 24 hours.

7.2.1 Number of Decision Trees

The number of decision trees constituting M is represented by *n_est*. In proportion to the increase of *n_est*, the number of paths of M increases, and the length of formula F_M increases in proportion to the number of paths of M. That is, as *n_est* increases, the length of the formula F_M increases by $O(n\_est)$. Therefore, it is considered that the execution time of the SMT solver tends to increase as *n_est* increases. The execution times of the proposed method when the value of *n_est* was changed are listed in Table 2. The results in Table 2 show that the execution time of the SMT solver tends to increase as *n_est* increases. As for the verification of φ_2, a timeout occurs when $n\_est \geq 190$. If the number of executions of the SMT solver is focused on, it turns out that the SMT solver was executed frequently in the case of φ_2. In

Table 2 Execution time when *n_est* is changed

n_est	φ_1				φ_2				φ_3			
	Total run time (s)	SMT solver run time (s)	Number of executions of SMT solver	Average execution time per run of SMT solver (s)	Total run time (s)	SMT solver run time (s)	Number of executions of SMT solver	Average execution time per run of SMT solver (s)	Total run time (s)	SMT solver run time (s)	Number of executions of SMT solver	Average execution time per run of SMT solver (s)
100	1063.66	1061.14	822	1.29	710.31	706.65	1097	0.64	1.53	0.94	1	0.94
110	1586.40	1584.13	771	2.05	1205.49	1197.99	2458	0.49	1.81	1.05	1	1.05
120	2523.68	2521.52	731	3.45	3029.89	3010.17	7986	0.38	2.11	1.22	1	1.22
130	1124.10	1121.76	757	1.48	3258.12	3233.20	10147	0.32	1.89	0.97	1	0.97
140	2809.93	2807.18	777	3.61	5982.13	5941.71	15372	0.39	1.98	1.16	1	1.16
150	5769.97	5766.97	830	6.95	12910.27	12857.07	17887	0.72	2.22	1.30	1	1.30
160	7559.52	7556.19	825	9.16	37227.36	37196.13	9202	4.04	3.08	2.11	1	2.11
170	13223.68	13219.80	717	18.44	53506.85	53480.13	8759	6.11	12.90	11.56	1	11.56
180	23162.16	23158.84	594	38.99	84575.47	84564.18	2878	29.38	58.44	57.36	1	57.36
190	22555.33	22550.98	712	31.67	timeout				219.17	218.15	1	218.15
200	25035.25	25030.94	724	34.57					649.38	648.49	1	648.49
210	21008.32	21004.44	709	29.63					1206.68	1205.65	1	1205.65
220	27452.39	27449.03	621	44.20					2991.70	2990.65	1	2990.65
230	38842.29	38837.71	771	50.37					4696.83	4695.73	1	4695.73
240	45709.01	45704.40	798	57.27					4954.07	4953.40	1	4953.40
250	54722.09	54717.69	796	68.74					6966.38	6965.68	1	6965.68
260	timeout								10061.54	10060.76	1	10060.76
270									20043.85	20043.04	1	20043.04
280									25163.02	25162.18	1	25162.18
290									30264.30	30263.52	1	30263.52
300									52134.28	52133.47	1	52133.47
310									58858.78	58857.78	1	58857.78
320									72110.07	72109.17	1	72109.17
330									timeout			

Table 3 Execution times before and after changing parameters

n_est	φ_2									
	Before changing parameters ($r_a = 100,\ r_b = 10,\ r_c = 10$)					After changing parameters ($r_a = 20,\ r_b = 30,\ r_c = 5$)				
	Total run time (s)	SMT solver run time (s)	Number of executions of SMT solver	Average execution time per run of SMT solver (s)	Hypervolume sum (after division)	Total run time (s)	SMT solver run time (s)	Number of executions of SMT solver	Average execution time per run of SMT solver (s)	Hypervolume sum (after division)
100	710.31	706.65	1097	0.64	2.09E+18	477.02	475.84	292	1.63	2.61E+18
110	1205.49	1197.99	2458	0.49	5.82E+18	1062.60	1060.70	689	1.54	1.29E+19
120	3029.89	3010.17	7986	0.38	1.47E+19	2186.94	2184.44	907	2.41	2.98E+19
130	3258.12	3233.20	10147	0.32	7.34E+19	1992.35	1990.15	750	2.65	2.37E+20
140	5982.13	5941.71	15372	0.39	7.36E+19	3694.32	3689.32	1648	2.24	2.39E+20
150	12910.27	12857.07	17887	0.72	7.94E+19	10233.05	10228.46	1208	8.47	2.53E+20
160	37227.36	37196.13	9202	4.04	3.21E+20	15290.46	15284.93	1387	11.02	3.98E+20
170	53506.85	53480.13	8759	6.11	2.47E+20	21506.59	21504.97	177	121.50	7.85E+20
180	84575.47	84564.18	2878	29.38	2.66E+20	27538.12	27536.06	248	111.03	7.58E+20
190	timeout					42610.96	42608.63	283	150.56	8.05E+20
200						38449.99	38447.89	243	158.22	6.57E+20
210						68205.25	68202.88	295	231.20	7.64E+20
220						55449.52	55447.78	168	330.05	1.01E+21
230						timeout				

such a case, changing parameters r_a, r_b, and r_c may reduce the number of executions of the SMT solver and, thereby, shorten the execution time.

Parameter r_a determines the size of the search range in *range_extraction*. In the implementation shown in Eq. (9), as the value of r_a gets smaller, the search range gets larger. The number of times *formal_verification* is called can therefore be reduced if the value of r_a is decreased. Note that in that case, it is highly likely that the extracted violation range is wider than before r_a was decreased. That is, the extracted violation range includes more satisfying instances. r_b is referred to as a reference value for determining whether or not to execute *range_division*. In the implementation shown in Sect. 7.1, as the value of r_b increases, the possibility of dividing the violation range decreases. The number of times *formal_verification* is called is thus reduced if the value of r_b is increased. In that case, however, the possibility of narrowing the violation range is reduced. r_c determines how many division orders are tried in *range_division*. Therefore, if the value of r_c is reduced, the number of times *formal_verification* is called is also reduced. Accordingly, the values of these parameters were changed to $r_a = 20$, $r_b = 30$, and $r_c = 5$, and the verification of φ_2 was retried. Execution times before and after changing the parameters are listed in Table 3. As shown in Table 3, by changing the values of r_a, r_b, and r_c, even in the case of $190 \leq n\_est \leq 220$, execution could be completed. This result demonstrates that when execution cannot be completed within a practical time, due to the number of executions of the SMT solver, the execution time can be shortened by adjusting the values of these parameters. It should, however, be noted that when these parameters are adjusted to reduce

Table 4 Execution times in case of changing *max_d*

max_d	φ_1				φ_2				φ_3			
	Total run time (s)	SMT solver run time (s)	Number of executions of SMT solver	Average execution time per run of SMT solver (s)	Total run time (s)	SMT solver run time (s)	Number of executions of SMT solver	Average execution time per run of SMT solver (s)	Total run time (s)	SMT solver run time (s)	Number of executions of SMT solver	Average execution time per run of SMT solver (s)
2	8.77	5.44	1652	0.003	30.03	24.54	1641	0.01	0.53	0.22	1	0.22
3	1063.66	1061.14	822	1.29	710.31	706.65	1097	0.64	1.53	0.94	1	0.94
4	16985.42	16982.48	738	23.01	54262.33	54254.26	2299	23.60	4.81	3.36	1	3.36
5									17.71	14.87	1	14.87
6									66.70	61.89	1	61.89
7									214.57	206.94	1	206.94
8									700.18	688.26	1	688.26
9									2485.35	2463.96	1	2463.96
10	timeout				timeout				4579.15	4544.48	1	4544.48
11									7644.85	7601.93	1	7601.93
12									15235.65	15192.93	1	15192.93
13									29206.68	29148.72	1	29148.72
14									55403.9	55320.072	1	55320.07
15									timeout			

Table 5 Execution times in case of changing dimension *s*

s	φ_1				φ_2				φ_3			
	Total run time (s)	SMT solver run time (s)	Number of executions of SMT solver	Average execution time per run of SMT solver (s)	Total run time (s)	SMT solver run time (s)	Number of executions of SMT solver	Average execution time per run of SMT solver (s)	Total run time (s)	SMT solver run time (s)	Number of executions of SMT solver	Average execution time per run of SMT solver (s)
2	0.31	0.03	1	0.03	1.27	0.67	1	0.67	0.89	0.36	1	0.36
3	0.66	0.33	1	0.33	6.75	6.07	1	6.07	1.05	0.44	1	0.44
4	0.77	0.43	1	0.43	12.25	11.62	1	11.62	1.12	0.47	1	0.47
5	62.37	61.08	543	0.11	25.35	24.75	1	24.75	1.16	0.50	1	0.50
6	553.87	549.83	2063	0.27	216.19	215.32	105	2.05	1.19	0.56	1	0.56
7	1063.66	1061.14	822	1.29	710.31	706.65	1097	0.64	1.53	0.94	1	0.94
8	2941.19	2938.35	937	3.14	3708.19	3695.22	4407	0.84	1.37	0.72	1	0.72
9	9289.89	9286.22	1096	8.47	7898.07	7879.84	6260	1.26	1.33	0.72	1	0.72
10	4714.45	4710.71	1045	4.51	6000.43	5982.00	5926	1.01	1.50	0.89	1	0.89
11	9997.78	9992.27	1041	9.60	6199.39	6187.48	3201	1.93	1.37	0.81	1	0.81
12	9183.41	9178.07	1184	7.75	19357.53	19351.55	1108	17.47	1.41	0.84	1	0.84
13	34333.31	34324.08	1758	19.52	53627.70	53619.83	1464	36.63	1.50	0.92	1	0.92
14	448.88	385.35	15431	0.02	2797.57	2709.76	19109	0.14	1.42	0.77	1	0.77
15	897.81	897.29	1	897.29	10366.73	10365.96	1	10365.96	1.39	0.77	1	0.77
16	4584.89	4574.87	1747	2.62	24870.13	24859.50	1861	13.36	1.61	0.95	1	0.95
17	11995.09	11982.71	1669	7.18	62240.87	62228.50	1553	40.07	1.16	0.55	1	0.55
18	17112.35	17099.96	1639	10.43	57923.14	57887.69	6440	8.99	1.14	0.58	1	0.58

the execution time, the extracted violation range may become wider. In fact, according to Table 3, the total volume of the violation range is increased by changing the value of the parameters. On the contrary, it may be possible to narrow the violation range more strictly by changing these parameters if it is acceptable that the execution time becomes longer. That is, by changing these parameters, it is possible to adjust the balance between the fineness of the violation range and the execution time.

7.2.2 Depth of Decision Trees

The maximum depth of a decision tree is taken as *max_d*. When the depth increases by 1, the number of paths extracted from one decision tree is doubled. (Moreover, the length of each path also increases in proportion to *max_d*.) Therefore, the length of formula F_M increases by $O(2^{max\_d})$. Accordingly, if *max_d* increases, it is considered that the execution time of the SMT solver increases. The execution times of the proposed method when the value of *max_d* was changed are listed in Table 4. It is clear from the results in Table 4 that the average execution time of the SMT solver increases as *max_d* increases. In particular, as for verification of φ_1 and φ_2, the execution time increases by 10 to 100 times each time *max_d* increases by one. When a timeout is invalidated and φ_1 is verified with *max_d* = 5, it takes about 70 hours to complete execution, and the average execution time of SMT solver is about 46 seconds. From these results, it can be said that *max_d* significantly influences the execution time of the proposed method.

7.2.3 Dimension of Attribute Vector

The loop in line 12 of *range_extraction* is executed the same number of times as the number of dimensions s. In addition, as s increases, the number of variables appearing in F_M also increases; that is, F_M becomes more complicated. For the reasons above, it is considered that the execution time of the SMT solver tends to increase with increasing s. In the house-price dataset used in this study, we can use up to 18 attributes as elements of x. These attributes were therefore used to change dimension s of x in the range of $2 \leq s \leq 18$. The execution times of the proposed method when the value of s was changed are listed in Table 5. It is clear from the results in Table 5 that execution time tends to increases as s increases. However, the execution time sometimes decreases even though s increases, for example, in the verification of φ_1, when $s = 14$. In other cases, for example, when

$s = 13$, execution time increases significantly. Therefore, the correlation of s with the execution time is considered to be weaker than that of *n_est* and *max_d* with execution time.

According to the results in Tables 3, 4, and 5, it can be concluded that *n_est*, *max_d*, and s are factors that increase the execution time of the proposed method. In particular, *n_est* and *max_d* are important factors that determines the applicability of the proposed method. Specifically, the proposed method is practical if *n_est* is less than around 200 and *max_d* is less than 5 at least.

8. Discussion

As for the proposed method, the search range is set around the first violating instance detected, and if a violating instance exists within the search range, the search range is defined as the violation range. Furthermore, by setting the search range around the violation range, the violation range is expanded in the same manner. After the violation range is extracted, it is confirmed whether another violating instance exists outside the violation range. If another violating instance exists, the violation range is extracted in the same way. In this manner, it is assured that all violating instances fall within any violation range. Therefore, by creating the input filter shown in Fig. 1 based on the violation ranges extracted by the proposed method, all violating instances leading to system failures can be filtered.

Moreover, the violation range can be narrowed by dividing the extracted violation range. Through a case study, the feasibility of the division of the violation range was confirmed. The extent to which the violation range can be narrowed depends on the DTEM or the property to be verified; thus, it is difficult to estimate the effectiveness of the division in advance. However, dividing the violation range is still useful for narrowing the violation range on a trial basis.

In the case study described in Sect. 7.2, it was shown that the number of decision trees constituting the DTEM, *n_est*, maximum depth of the decision trees, *max_d*, and dimension s are factors that increase the execution time of the proposed method. Among those factors, *max_d* has the greatest influence on the execution time. On the other hand, the correlation of s with execution time is considered to be relatively weak. Moreover, as mentioned above, F_M becomes longer exponentially with increasing *max_d*. On the contrary, the length of F_M does not change even if s increases. From these findings, it is inferred that the execution time of the proposed method strongly depends on the length of F_M.

The purpose of the proposed method is to create the filtering condition for the input filter. In that case, the proposed method is required to complete execution within a practical time. However, even if execution of the proposed method is not completed within a practical time, the proposed method can be utilized by returning the violation range extended up to that point when the execution was interrupted. The violation range extracted at the time execution is suspended does not include all violating instances. However, it is useful for determining where violating instances exist around. For example, it is possible to use the attribute vectors included in the violation range as training data for additionally training the DTEM. The proposed method is thus useful even in the case of a large-scale DTEM whose execution is not completed within the practical time.

As described in Sect. 1, as a method for obtaining the condition to filter violating instances, a method that finds all the violating instances can be considered. However, for a certain violating instance, a large number of similar violating instances might exist, and the values of some elements in those instances might be slightly changed. Accordingly, from the viewpoint of calculation time, it is not practical to detect all violating instances. Furthermore, as another approach, extracting the path condition [29] of the decision trees constituting the DTEM is considered hereafter.

In the same manner as the proposed method, the violating instance is detected by verification. If the detected violating instance is input into the DTEM, the execution path of each decision tree is uniquely determined. Here, the condition of the attribute values to execute these paths is called "path condition". The path condition can be extracted by executing the DTEM with the violating instance or by analyzing the decision trees. If an attribute vector satisfies the path condition and is input into the DTEM, the same paths are executed as they are when the violating instance is input. Moreover, if the same paths are executed, the same prediction value is returned. For all attribute vectors satisfying the path condition, the DTEM therefore returns the same prediction value. Here, a conditional expression for the attribute vector is created by assigning the prediction value to variable y appearing in the property to be verified and negating the assigned property. For example, when the prediction value is given as 499,999, the conditional expression of property φ_1 is $\neg(x[0] \geq 7000 \Rightarrow 499999 \geq 500000)$. If an attribute vector satisfies the conjunction of the path condition and the conditional expression of the property, the corresponding prediction value is 499,999, and at that time, the property is not satisfied. Hereafter, the conjunction is called the "violation condition." By repeating the above procedure, other violation conditions can be extracted. Finally, the set of extracted violation conditions can be used as the filtering condition of the input filter.

Since the extracted violation condition filters only attribute vectors that violate the property, this method based on the path condition creates the filtering condition more precisely than the proposed method. However, from the viewpoint of calculation time, this method is considered to be less practical than the proposed method because it is likely to extract a large number of violation conditions. For example, in the case of a DTEM composed of 100 decision trees with two paths each, there are 2^{100} combinations of paths, that is, about 10^{29} combinations. The violation condition extracted in a single verification is only one of the 10^{29} combinations, and it is highly likely that there will be a large number of similar violation conditions in which part of the path condition differs. In that case, since the number

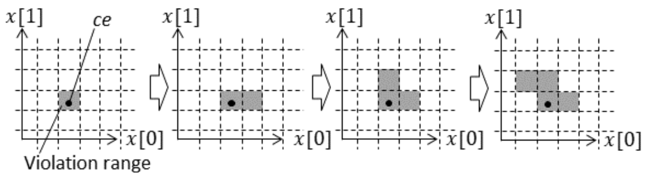


Fig. 7 Example of extending violation range by using a mesh strategy

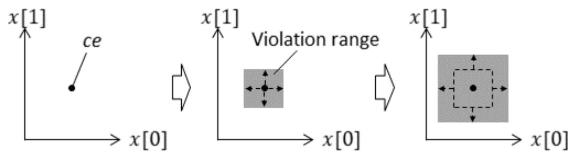


Fig. 8 Example of extending violation range by using a hypercube strategy

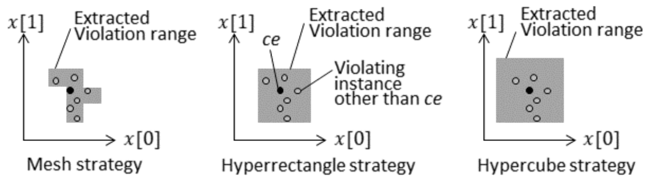


Fig. 9 Comparison of fineness of violation range

of verifications by the SMT solver is also enormous, this method is not practical.

In general, fineness of the violation range and execution time have a trade-off relationship. As for the strategy to extend the violation range, this trade-off relationship should be discussed. As for the proposed method, the violation range is extended in the upward and downward direction of $x[k]$ $(0 \leq k \leq s-1)$ in order. Hence, violation ranges are extracted in the form of hyperrectangles. The extension strategy of the proposed method is therefore called a "hyperrectangle strategy." Here, the mesh strategy described below can be considered as a strategy for extracting the violation range more precisely than the hyperrectangle strategy. An example of extending the violation range according to mesh strategy is shown in Fig. 7. As for this mesh strategy, first, the space of attribute vectors is divided into meshes. Then, the mesh including the violating instance *ce* is taken as the initial violation range. Next, whether a violating instance exists in the mesh adjacent to the violation range is verified by *formal_verification*. If a violating instance exists in the adjacent mesh, the adjacent mesh is taken into the violation range. Then, if no violating instance exists in any mesh adjacent to the violation range, the extension of the violation range is ended. When this strategy is adopted, the number of adjacent meshes increases as the violation range is extended. For example, if it is supposed that the violation range is an s-dimensional hypercube and the number of meshes on one side of the violation range is α, the number of adjacent meshes to the violation range is $2s{*}\alpha^{s-1}$. To extend the violation range by one round, *formal_verification* is executed the same number of times as the number of adjacent meshes. The amount of calculation to extend the violation range by one round therefore increases as the violation range is extended. On the other hand, in the case of the hyperrectangle strategy, to extend the violation range by one round in the same situation, *formal_verification* is executed only $2s$ times. That is, it is advantageous that the violation range can be extended with a fixed number of executions regardless of the size of the violation range.

Moreover, a strategy to increase the values of $x[1], \ldots, x[s-1]$ simultaneously can be considered. This strategy can extend the violation range in a shorter time than possible with the hyperrectangle strategy. In this case, the violation range is extracted in the form of a hypercube, so this strategy is called a "hypercube strategy." An example of extending the violation range on the basis of the hypercube strategy is shown in Fig. 8. When the hypercube strategy is adopted, the number of verifications for extending the violation range by one round is one. However, the violation range extracted by the hypercube strategy is coarser than that of other strategies. An example of the extracted violation range for each strategy is shown in Fig. 9. The hypercube strategy is highly likely to include a range in which no violating instance exists in the violation range. Therefore, if an input filter is created based on the violation range created by the hypercube strategy, many attribute vectors satisfying the property will be filtered.

In terms of execution time, mesh strategies are considered less practical than hyperrectangle strategies. The hypercube strategy is considered to be less practical in terms of the fineness of the violation ranges. A hyperrectangle strategy with a good balance between fineness and execution time was therefore adopted.

As mentioned in Sect. 7.2, changing the values of r_a, r_b, and r_c makes it possible to adjust the balance between fineness of the violation range and execution time. These parameters should be changed by the user according to the DTEM and property to be verified as well as allowable execution time. As for finding an appropriate parameter value, the following procedure can be considered. First, the parameter values are set so that execution time is short, and it is confirmed by actually executing the procedure that the execution time is shorter than the allowable time. After that, the parameters are gradually changed to improve the fineness of the violation range. This procedure improves the fineness of the violation ranges within the constraints of execution time.

Among the proposed methods, *range_extraction* and *range_division* are applicable to machine-learning models other than a DTEM. On the contrary, *formal_verification* is a procedure specialized for a DTEM, so it must be replaced with a verification method suitable for the target machine-learning model. For example, in the case of a DNN, methods for encoding and verifying the DNN in a formula have been proposed [14]–[19]. By replacing *formal_verification* in the proposed method with such a method, it would be possible to extract the violation range of the DNN.

9. Related Work

As described in Sect. 1, Bogdiukiewicz et al. proposed a method for developing a policing function for autonomous systems [12]. The policing function checks prediction values of intelligent function such as a machine-learning model at runtime. The policing function is useful for preventing failures in a similar manner to the input filter. However, the policing function works after the intelligent function is executed. That is, the policing function detects and controls the possible failure later than the input filter. The proposed method for creating the input filter is therefore more useful in development of systems that require quick handling of failures.

In $formal\_verification$ shown in Sect. 4, a rule-form formula composed of conditions and conclusions is extracted from the DTEM. Extracting a specification from a program in the form of rules has been studied [22]–[28]. In these studies, the control path of the program is extracted by static analysis or symbolic execution, and the rule is created on the basis of the branch condition that constitutes the control path. The method proposed in this paper extracts the decision tree path and creates a rule-form formula based on the attribute test associated with the decision node that composes the path. It can thus be said that it adopts the same approach.

As for the verification of the rule-form formula, methods for verifying properties such as consistency, redundancy, and completeness [30] have been established. Furthermore, a method for extracting "minimal unsatisfiable subsets," which are useful for causal analysis when the rule set does not satisfy consistency, has also been proposed [31]–[33]. Validation of rule programs used in a business-rule management system has also been studied [35]. As in the case of the method proposed in this paper, the verification of the rule-form formula is translated to checking the satisfiability problem.

As described in Sect. 1, an approach of encoding a DNN model as a formula and verifying it by determining the satisfiability of the formula has been proposed in recent years [14]–[19]. It can be said that the method for verifying the DTEM proposed in this paper also takes this approach. However, a study describing a specific method for verifying a DTEM has not been reported. One of the contributions of this paper is formally specifying a verification method for a DTEM and demonstrating its feasibility through a case study.

To the authors' knowledge, no similar studies on extracting violation ranges as described in Sect. 5 and dividing violation ranges as described in Sect. 6 have been reported. One of the reasons for that situation is that the problems solved by these methods are unique to software developed by machine learning. In regard to conventional software, namely, algorithmic programs, when a counterexample is detected by verification, the fault that caused the counterexample is analyzed, and the fault is removed by correcting the algorithm. Verification and correction are then repeated until no counterexample is detected. On the other hand, in the case of a machine learning model, a possible method for handling such faults is retraining or additional training using the detected counterexample (and data similar to it). Although this approach may eliminate the fault, retraining and additional training may affect the entire model, and another new fault may be inserted. That is, "regression" occurs. Although this regression also occurs in the case of algorithmic programs, in that case, regression occurs due to a developer's mistake; therefore, such mistakes must be carefully corrected to avoid regression. On the contrary, in the case of a machine-learning model, it is difficult for the developer to control retraining and additional training so that regression does not occur. In other words, it can be said that it is inherently difficult to create a complete machine-learning model that always returns the expected prediction value. Accordingly, when a machine-learning model is implemented in a system, the proposed method is used to implement the input filter. That is, the proposed method solves a specific problem that occurs when machine-learning models are used.

10. Conclusion

When a DTEM is implemented in a system, the input filter can be effectively used to prevent system failures. As a means of creating the filtering condition for the input filter, a method for extracting the violation range of the DTEM, which takes multi-dimensional vectors whose elements are continuous numerical variables, was proposed. The proposed method consists of procedures for formally verifying the DTEM, extracting the violation range, and narrowing the extracted violation range. The violation range extracted by the proposed method includes all violating instances. The proposed method is therefore useful for creating the filtering condition. On the basis of the results of the case study using a dataset on the house prices, the feasibility of the proposed method was demonstrated. Through the case study, the scalability of the proposed method was also evaluated. The number of decision trees constituting the DTEM, the maximum depth of the decision trees, and the dimension of the attribute vector were shown to be factors that increase the execution time of the proposed method. Specifically, it is concluded that the proposed method is practical if the number of decision trees is less than around 200 and the maximum depth of the decision trees is less than 5 at least.

Future issues include improving the scalability of the proposed method. Since the form of formula F_M created by the proposed method is constant, the scalability of the proposed method may be improved by implementing heuristics specialized for that form in the SMT solver. Moreover, the procedure for finding appropriate values of parameters r_a, r_b, and r_c shown in Sect. 8 can be incorporated into the proposed method. Furthermore, the limitation of the proposed method can be evaluated in more detail by adding case studies using other datasets.

References

[1] A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks," Advances in Neural Information Processing Systems, pp.1097–1105, 2012.

[2] G. Hinton, L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury, "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups," IEEE Signal Process. Mag., vol.29, no.6, pp.82–97, 2012.

[3] L. Breiman, "Random forests," Machine Learning, vol.45, no.1, pp.5–32, 2001.

[4] J.H. Friedman, "Greedy function approximation: a gradient boosting machine," Annals of statistics, pp.1189–1232, 2001.

[5] P. Li, "Robust logitboost and adaptive base class (abc) logitboost," arXiv preprint arXiv:1203.3491, 2012.

[6] M. Richardson, E. Dominowska, and R. Ragno, "Predicting clicks: estimating the click-through rate for new ads," Proc. 16th international conference on World Wide Web, pp.521–530, ACM, 2007.

[7] C.J.C. Burges, "From ranknet to lambdarank to lambdamart: An overview," Microsoft Research Technical Report MSR-TR-2010-82, 2010.

[8] R.A.V. Rossel and T. Behrens, "Using data mining to model and interpret soil diffuse reflectance spectra," Geoderma, vol.158, pp.46–54, 2010.

[9] P.T. Sorenson, C. Small, M.C. Tappert, S.A. Quideau, B. Drozdowski, A. Underwood, and A. Janz, "Monitoring organic carbon, total nitrogen, and pH for reclaimed soils using field reflectance spectroscopy," Can. J. Soil Sci., vol.97, no.2, pp.241–248, 2017.

[10] A.M. Prasad, L.R. Iverson, and A. Liaw, "Newer classification and regression tree techniques: Bagging and random forests for ecological prediction," Ecosystems, vol.9, no.2, pp.181–199, 2006.

[11] H. Ishwaran, "Variable importance in binary regression trees and forests," Electron. J. Stat., vol.1, pp.519–537, 2007.

[12] C. Bogdiukiewicz, M. Butler, T.S. Hoang, M. Paxton, J.H. Snook, X. Waldron, and T. Wilkinson, "Formal Development of Policing Functions for Intelligent Systems," 2017 IEEE 28th International Symposium on Software Reliability Engineering, 2017.

[13] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," Proc. 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp.785–794, ACM, 2016.

[14] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety Verification of Deep Neural Networks," Computer Aided Verification 2017, Lecture Notes in Computer Science, vol.10426, pp.3–29, 2017.

[15] G. Katz, C. Barrett, D.L. Dill, K. Julian, and M.J. Kochenderfer, "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks," Computer Aided Verification 2017, pp.97–117, 2017.

[16] G. Katz, C. Barrett, D.L. Dill, K. Julian, and M.J. Kochenderfer, "Towards Proving the Adversarial Robustness of Deep Neural Networks," arXiv:1709.02802, 2017.

[17] D. Gopinath, G. Katz, C. Pasareanu, and C. Barrett, "DeepSafe: A data-driven approach for checking adversarial robustness in neural networks," arXiv:1710.00486, 2017.

[18] R. Ehlers, "Formal verification of piece-wise linear feed-forward neural networks," Automated Technology for Verification and Analysis 2017, 2017.

[19] R. Bunel, I. Turkaslan, P.H.S. Torr, P. Kohli, and M.P. Kumar, "A unified view of piecewise linear neural network verification," arXiv:1711.00455, 2018.

[20] P.E. Utgoff, "Incremental induction of decision trees," Machine Learning, vol.4, no.2, p.161, 1989. https://doi.org/10.1023/A:1022699900025

[21] A.M. Bhavitha S. and S. Madhuri, "A classification method using decision tree for uncertain data," International Journal of Computer Trends and Technology (IJCTT), V3(1), pp.102–107, 2012.

[22] T. Hatano, T. Ishio, J. Okada, Y. Sakata, and K. Inoue, "Extraction of Conditional Statements for Understanding Business Rules," Proc. 6th International Workshop on Empirical Software Engineering in Practice, 2014.

[23] V. Cosentino, J. Cabot, P. Albert, P. Bauquel, and J. Perronnet, "Extracting business rules from COBOL: A model-based framework," 20th Working Conference on Reverse Engineering (WCRE), pp.409–416, 2013.

[24] H.M. Sneed, "Extracting Business Logic from Existing COBOL Programs as a Basis for Redevelopment," Proc. 9th International Workshop on Program Comprehension, pp.167–175, 2001.

[25] X. Wang, J. Sun, and X. Yang, "Business rules extraction from large legacy systems," Proc. CSMR, 2004, pp.249–253, 2004.

[26] H. Huang and W. Tsai, "Business rule extraction from legacy code," Proc. COMPSAC, 1996, pp.162–167, 1996.

[27] V. Cosentino, J. Cabot, P. Albert, P. Bauquel, and J. Perronnet, "A Model Driven Reverse Engineering Framework for Extracting Business Rules out of a Java Application," Proc. RuleML, 2012, pp.17–31, 2012.

[28] J. Pichler, "Specification extraction by symbolic execution," Proc. 20th Working Conference on Reverse Engineering, pp.462–466, 2013.

[29] R. Baldoni, E. Coppa, D.C. D'elia, C. Demetrescu, and I. Finocchi, "A Survey of Symbolic Execution Techniques," ACM Computing Surveys (CSUR), vol.51, no.3, pp.1–39, 2018.

[30] A. Ligeza and G.J. Nalepa, "Rules verification and validation," Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches, pp.273–301, 2009.

[31] J. Bailey and P.J. Stuckey, "Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization," Practical Aspects of Declarative Languages, 7th International Symposium, PADL 2005, Lecture Notes in Computer Science, vol.3350, pp.174–186, 2005.

[32] M.H. Liffiton and A. Malik, "Enumerating infeasibility: Finding multiple MUSes quickly," Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2013, Lecture Notes in Computer Science, vol.7874, pp.160–175, 2013.

[33] M.H. Liffiton and K.A. Sakallah, "Algorithms for computing minimal unsatisfiable subsets of constraints," Journal of Automated Reasoning, vol.40, no.1, pp.1–33, 2008.

[34] T.S. Hoang, S. Itoh, K. Oyama, K. Miyazaki, H. Kuruma, and N. Sato, "Consistency Verification of Specification Rules," Formal Methods and Software Engineering (ICFEM) 2015, Lecture Notes in Computer Science, vol.9407, pp.50–66, 2015.

[35] B. Berstel and M. Leconte, "Using constraints to verify properties of rule programs," Third International Conference on Software Testing, Verifcation and Validation, ICST 2010, pp.349–354, 2010.

[36] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008, Lecture Notes in Computer Science, vol.4963, pp.337–340, 2008.

Naoto Sato is a senior researcher in the System Productivity Research Department, Research and Development Group, Hitachi, Ltd. He received an M.S. from Tokyo Institute of Technology in 2005 and a Ph.D. from the University of Electro-Communications in 2016. His research interests and expertise include methods and tools for improving software productivity and quality, especially formal modeling and verification.

Hironobu Kuruma received an M.S. in physics from Hiroshima University in 1983 and a Ph.D. from The Graduate University for Advanced Studies in 2006. He is a unit member of Research and Development Group of Hitachi, Ltd. His research interests are software engineering and formal methods. He is a member of ACM, IEEE, IPSJ, and JSSST.

Yuichiroh Nakagawa received a B.E. and an M.S. in aeronautics and astronautics from Tokai University in 1999 and 2001, respectively. He is a researcher in the Center for Technology Innovation - Systems Engineering, Research and Development Group, Hitachi, Ltd. His research interests include software testing techniques.

Hideto Ogawa received an M.S. in information engineering from Nagoya University in 1996 and a Ph.D. in information science from Japan Advanced Institute of Science and Technology in 2015. He is a chief researcher at the Center for Technology Innovation - Systems Engineering, Research and Development Group, Hitachi, Ltd. His research interests include formal methods, software testing, and software-development processes. He is a member of the IPSJ, IEICE, and JSSST.