Higher Income, Larger Loan? Monotonicity Testing of Machine Learning Models

Arnab Sharma
Paderborn University
Paderborn, Germany
arnab.sharma@uni-paderborn.de

Heike Wehrheim
Paderborn University
Paderborn, Germany
wehrheim@uni-paderborn.de

ABSTRACT

Today, machine learning (ML) models are increasingly applied in decision making. This induces an urgent need for *quality assurance* of ML models with respect to (often domain-dependent) requirements. *Monotonicity* is one such requirement. It specifies a software as "learned" by an ML algorithm to give an increasing prediction with the increase of some attribute values. While there exist multiple ML algorithms for *ensuring* monotonicity of the generated model, approaches for *checking* monotonicity, in particular of *black-box* models are largely lacking.

In this work, we propose *verification-based testing* of monotonicity, i.e., the formal computation of test inputs on a white-box model via verification technology, and the automatic inference of this approximating white-box model from the black-box model under test. On the white-box model, the space of test inputs can be systematically explored by a directed computation of test cases. The empirical evaluation on 90 black-box models shows that verification-based testing can outperform adaptive random testing as well as property-based techniques with respect to effectiveness and efficiency.

CCS CONCEPTS

• **Software and its engineering**;

KEYWORDS

Machine Learning Testing, Monotonicity, Decision Tree.

ACM Reference Format:
Arnab Sharma and Heike Wehrheim. 2020. Higher Income, Larger Loan? Monotonicity Testing of Machine Learning Models. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3395363.3397352

1 INTRODUCTION

Today, machine learning (ML) is increasingly employed to take decisions previously made by humans. This includes areas as diverse as insurance, banking, law, medicine or autonomous driving. Hence, quality assurance of ML applications becomes of prime importance. Consequently, researchers have started to develop methods checking various sorts of requirements. Depending on the domain of application, such methods target safety, security, fairness, robustness or balancedness of ML algorithms and models (e.g., [9, 14, 18, 30]).

A requirement frequently expected in application domains is *monotonicity* with respect to dedicated attributes. Monotonicity requires an increase in the value of some attribute(s) to lead to an increase in the value of the prediction (class attribute). For instance, a loan-granting ML-based software might be required to give larger loans whenever the value of attribute "income" gets higher, potentially even when other attribute values are changed.

Monotonicity requirements occur in numerous domains like economic theory (house pricing, credit scoring, insurance premium determination), medicine (medical diagnosis, patient medication) or jurisdiction (criminal sentencing). In particular, monotonicity is often a requirement for ML software making acceptance/rejection decisions as it supports justification of a decision ("she gets a larger loan because she has got a higher income"). However, even if the training data used to generate the predictive model is monotone, the ML software itself might not be [26]. Hence, there are today a number of specialized ML algorithms which provide learning techniques guaranteeing monotonicity constraints on models (e.g. [26, 34, 36]).

Less studied is, however, the *validation* of monotonicity constraints, i.e., methods for answering the following question:

> Given some black-box predictive model, does it satisfy a given monotonicity constraint?

The term "black-box model" states the independence on the ML-technology employed in the model (i.e., the technique should be equally applicable to e.g. neural networks, random forests or support vector machines). We aim at a *model-agnostic* solution.

In this paper, we present the first approach for automatic monotonicity testing of black-box machine learning models. Our approach systematically explores the space of test inputs by (1) the inference of a white-box model *approximating* the black-box model under test (MUT), and (2) the computation of counter examples to monotonicity on the white-box model via established verification technology. We call this approach *verification-based testing*. The computed counter examples serve as starting points for the generation of further test inputs by variation. If confirmed in the black-box model, they get stored as counter examples to monotonicity. If unconfirmed, they serve as input to an improvement of the approximation quality of the white-box model.

More detailedly, our approach comprises the following key steps:

- **White-box model inference.** A white-box model is generated by training a decision tree with data instances of the black-box model under test.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397352

Table 1: Example banking data set

No.	income	children	contract	loan
1	100.0	1	20	high
2	25.0	0	2	no
3	17.8	3	5	no
4	25.5	2	15	medium
5	39.0	0	11	medium

- **Monotonicity computation.** The decision tree is translated to a logical formula on which we use an SMT-solver for monotonicity verification.
- **Variation.** The computed counter examples are systematically varied (similar to strategies used in symbolic execution [20]) in order to increase the size of the test suite and its coverage of the test space.
- **White-box model improvement.** In case none of the counter examples are valid for the black-box model under test, we employ the data instances together with the MUT's prediction to re-train the decision tree and thereby restart with an improved white-box model.

We have implemented our approach and have experimentally evaluated it using standard benchmark data sets and both monotonicity aware and ordinary ML algorithms. Our experimental results suggest that our directed generation of test cases outperforms (non-directed) techniques like property-based testing wrt. the effectiveness of finding monotonicity failures.

Summarizing, this paper makes the following contributions:

- We formally define monotonicity of ML models.
- We present a novel approach to monotonicity testing via the usage of verification technology on an approximating white-box model.
- We systematically evaluate our approach on 90 black-box models and compare it to state-of-the-art property-based and adaptive random testing.
- For the implementation of adaptive random testing as a baseline (to compare against), we design a distance metric specific to monotonicity testing on ML models.

The paper is structured as follows. In the next section, we define monotonicity of machine learning models. In Section 3 we describe verification-based testing and the way we have implemented adaptive random testing. Section 4 presents the results of our experimental evaluation. We discuss related work in Section 5 and conclude in Section 6.

2 MONOTONICITY

We start by introducing the basic terminology in machine learning and defining two notions of monotonicity.

A typical *supervised* machine learning (ML) algorithm works in two steps. Initially, it is presented with a set of data instances called *training data*. In the first (learning) phase, the ML algorithm generates a function (the *predictive model*), generalising from the training data by using some statistical techniques. The generated *predictive model* (short, model) is then used in the second (prediction) phase to predict classes for unknown data instances.

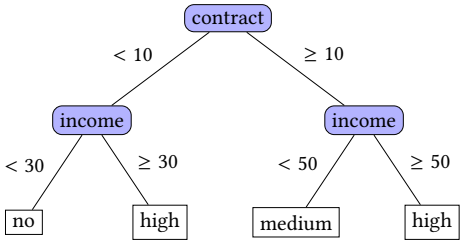

Figure 1: A decision tree for the banking data set

Formally, the generated model is a function

$$M : X_1 \times \ldots \times X_n \to Y \,,$$

where X_i is the value set of *feature i* (or attribute or characteristics i), $1 \leq i \leq n$, and Y is the set of *classes*. We define $\vec{X} = X_1 \times \ldots \times X_n$. The training data consists of elements from $\vec{X} \times Y$, i.e., data instances with known associated classes. During the prediction, the generated predictive model assigns a class $y \in Y$ to a data instance $(x_1, \ldots, x_n) \in \vec{X}$ (which is potentially not in the training data). We assume all X_i and the set of classes Y to be equipped with a total order $\preceq_i$ and $\preceq_Y$, respectively.

In this work, we check whether a given model is *monotone* with respect to a specific feature i.

Definition 1. *A model M is* strongly monotone[1] *with respect to a feature i if for any two data instances $x = (x_1, \ldots, x_n)$, $x' = (x'_1, \ldots, x'_n) \in \vec{X}$ we have $x_i \preceq_i x'_i$ implies $M(x) \preceq_Y M(x')$.*

Note that the feature values apart from the one with respect to which we are checking monotonicity can differ in an arbitrary way. Definition 1 can be weakened as to only require an increasing prediction when all features values apart from the chosen one are kept.

Definition 2. *A model M is* weakly monotone *with respect to a feature i if for any two data instances $x = (x_1, \ldots, x_n), x' = (x'_1, \ldots, x'_n) \in \vec{X}$, we have $(x_i \preceq_i x'_i) \wedge (\forall j, j \neq i . x_j = x'_j)$ implies $M(x) \preceq_Y M(x')$.*

In the literature, the term monotonicity most often refers to our weak version. We also say that a training data set is strongly/weakly monotone if the above requirements hold for all elements in the set. As an example take the training data in Table 1 for a software making decisions about the granting of loans (inspired by [26]): The features of a person are the income (in thousand dollars), the (number of) children and the (duration of) current contract. The class "loan" can take four values: 'no', 'low', 'medium' and 'high', with total order 'no' $<_Y$ 'low' $<_Y$ 'medium' $<_Y$ 'high'. This data set is monotone[2] for features "contract" and "income", but not for feature "children".

Figure 1 gives a potential model (in the form of a decision tree) which the training on this data set could yield. It can correctly predict all instances in the training data. However, this model is

[1] Note that "strong" here does not refer to a strong increase in values, i.e., a definition with $<$ instead of $\preceq$.

[2] Note that we cannot apply our formal definitions here since the data set does not give us classes for all data instances.

not weakly monotone in feature "contract" anymore: Take e.g. the following two data instances

income=30.0, children=0, contract=9
income=30.0, children=0, contract=10

While the prediction for the first instance is 'high', it is 'medium' for the latter.

We next define *group monotonicity* which extends Definitions 1 and 2 to a set of features (called *monotone features*).

Definition 3. *A predictive model M is said to be* strongly group monotone *with respect to a set of features $F = \{i_1, i_2, \ldots, i_m\} \subseteq \{1, \ldots, n\}$ if for any two data instances $x = (x_1, \ldots, x_n), x' = (x'_1, \ldots, x'_n) \in \vec{X}$ we have $\forall j \in F : x_j \preceq_j x'_j$ implies $M(x) \preceq_Y M(x')$.*

Similarly, it is weakly group monotone *with respect to F if for all x, x' we have $(\forall j \in F : x_j \preceq_j x'_j \wedge \forall j \notin F : x_j = x'_j)$ implies $M(x) \preceq_Y M(x')$.*

Strong group monotonicity sits in between weak and strong (single feature) monotonicity in that it allows some feature's values to change in an arbitrary way while other values may only increase or stay as they are[3]. We see this as being a practically relevant case and have thus included it in our definitions.

Finally, note that test cases for (both strong and weak) monotonicity are by these definitions *pairs* of data instances (x, x'), and during test execution these need to be checked for the property $M(x) \preceq_Y M(x')$. If the precondition of the respective monotonicity version holds for x and x' but $M(x) \not\preceq_Y M(x')$, we call the pair (x, x') a *counter example* to monotonicity.

3 TESTING APPROACH

We conduct black-box testing to check monotonicity of the predictive model under test. Hence, in the following we assume the type of the MUT (i.e., which ML algorithm has been used for training) to be unknown.

3.1 Adaptive Random Testing

For the purpose of comparison, we have designed and implemented an *adaptive random testing* (ART) [10] approach for monotonicity testing which we describe first. Adaptive random testing aims at (randomly) computing test cases which are more evenly distributed among the test input space. To this end, it compares new candidates with the already computed test cases, and adds the one "furthest" away. The implementation of "furthest" requires the definition of a *distance metric*. For numerical inputs, this is often the Euclidean distance. For monotonicity, our test cases are however *pairs* (x, x').

Assume we are given two such pairs (x, x') and (z, z') and want to define how "different" they are. Assume furthermore that all the elements only contain *numerical* values[4]. Every element $x = (x_1, \ldots, x_n)$ can then be considered to be a point in an n-dimensional space. We let $Euc(x, x')$ be the Euclidean distance between x and x', and $m_{x,x'}$ be the point laying at the middle of x and x'. The metric which we employ captures two aspects: we see two pairs (x, x') and (z, z') as being very different if (a) their Euclidean distances are far apart (e.g., x is very close to x', but z far away from z') and (b) the middle of (x, x') is far away from the middle of (z, z'). Formally, we define

$$dist\big((x, x'), (z, z')\big) = \frac{|Euc(x, x') - Euc(z, z')|}{2} + \frac{Euc(m_{x,x'}, m_{z,z'})}{2}$$

Note that *dist* is positive-definite, symmetric and subadditive, i.e., indeed a metric. We use this distance function in the test case generation for ART within Algorithm 1 (inspired by a definition of ART algorithms in [35]).

Algorithm 1 *artGen* (Test Generation for ART)

Input: F ▷ *set of monotone features*
Output: set of test cases

```
1:  ts := ∅; count := 0;
2:  while count < INI_SAMPLES do ▷ randomly generate start set
3:      x := random(X⃗);
4:      x' := random({x' | ∀i ∈ F : x_i ⪯ x'_i, ∀j ∉ F : x_j = x'_j});
5:      if (x, x') ∉ ts then
6:          ts := ts ∪ {(x, x')}; count++;
7:  while |ts| < MAX_SAMPLES do ▷ extend start set
8:      cand := ∅; count :=0;
9:      while count < POOL_SIZE do ▷ generate candidates
10:         x := random(X⃗);
11:         x' := random({x' | ∀i ∈ F : x_i ⪯ x'_i, ∀j ∉ F : x_j = x'_j});
12:         if (x, x') ∉ cand then
13:             cand := cand ∪ {(x, x')}; count++;
14:     c_fur := oneOf(cand); ▷ initialize with arbitrary cand.
15:     maxDist := 0;
16:     for c ∈ cand do ▷ determine "furthest away" cand.
17:         dist := minDistance(c, ts);
18:         if dist > maxDist then
19:             c_fur := c; maxDist := dist;
20:     ts := ts ∪ {c_fur};
21: return ts;
```

The first loop in Algorithm 1 randomly computes a set of pairs (x, x') to start with. Note that all these pairs already satisfy the precondition of — in this case – weak monotonicity by construction (line 4). The second (outermost) loop extends this test set until it contains `MAX_SAMPLES` pairs. It starts in line 9 by generating a set of candidates. The loop starting in line 16 then determines the candidate "furthest away" from the current test set *ts*. It uses the function `minDistance` which computes the minimal distance between c and elements of *ts* using the metric *dist*. The furthest away candidate is put into the test set *ts* in line 20 and the algorithm returns with the entire test set in line 21. This test set is then subject to checking all pairs (x, x') for monotonicity (not given as algorithm here).

For checking strong monotonicity we follow a similar approach with the only exception being the generation of test input pairs (x, x'). In that case, the changes are in lines 4 and 11 which become $x' := \text{random}(\{x' \mid x \neq x', (\forall i \in F : x_i \preceq x'_i)\})$.

[3]Note that all $\preceq$ orders are reflexive.
[4]This can easily be achieved by some preprocessing step converting categorical to numerical values.

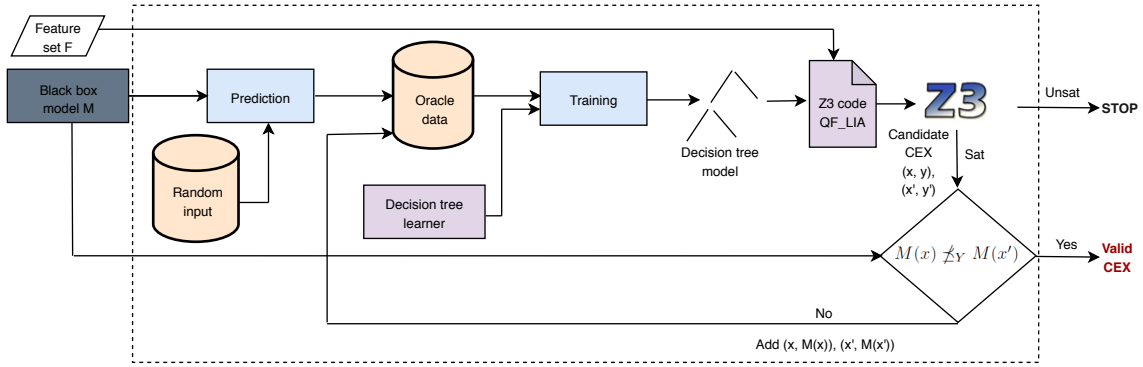

Figure 2: Basic workflow of verification-based testing

3.2 Verification-Based Testing

Next, we present our novel approach for the generation of test sets for monotonicity testing of a black-box model. The key idea therein is to approximate the MUT by a white-box model on which verification techniques can *compute* counter examples to monotonicity (in case these exist). These counter examples then serve as test inputs for the MUT, thereby achieving a target-oriented generation of test suites. We call this technique *verification-based testing* (VBT).

The idea of approximating an unknown predictive model by a white-box model is already employed in the areas of interpretability of AI as well as testing of (non-ML) software. In AI (Guidotti et al. [17]) an unknown black-box predictor is converted to an explainable model out of which explanations for humans can be constructed. In the field of testing, Papadopoulos and Walkinshaw [24] use the inference of a predictive model from test sets to further extend this set. None use it to compute counter examples to properties.

In our work, we approximate the black-box MUT by a decision tree. The choice of it is driven by the desire to employ verification-based testing: counter examples to monotonicity are *automatically computable* for decision trees via SMT solvers. Figure 2 depicts the basic workflow; the exact interplay of components is detailed in Algorithm 5. Our approach is composed of four parts.

White-box model inference. The inputs to our approach are the predictive model M (MUT) and a set of features F. In the first step, we train the decision tree. To this end, we generate so called *oracle data*, containing the predictions of the MUT for some randomly chosen input instances. This set of data instances are currently not used for testing monotonicity (but could be), rather they are employed for the purpose of generating the white-box model. A decision tree learner is then trained on the oracle data giving a decision tree model.

Monotonicity computation. Once we have generated the decision tree, the next step is to compute (non-)monotonicity. To this end, we use state of the art verification technology, namely the SMT solver Z3 [12]. First, we translate the decision tree into a logical formula describing how the classes are predicted for inputs x and x'. Figure 3 shows the Z3 code for the decision tree in Figure 1. Therein, variables `contract1` and `income1`[5] describe test input x

[5]There is no variable for "children" since the decision tree is not using this feature.

```
; Declaring components of x and x' and their classes
(declare-fun contract1 () Int) (declare-fun income1 () Real)
(declare-fun contract2 () Int) (declare-fun income2 () Real)
(declare-fun class1 () Int) (declare-fun class2 () Int)
; Specifying prediction of decision tree (no=0, medium=1, high=2)
(assert (=> (and (< contract1 10) (< income1 30)) (= class1 0)))
(assert (=> (and (< contract1 10) (>= income1 30)) (= class1 2)))
(assert (=> (and (>= contract1 10) (< income1 50)) (= class1 1)))
(assert (=> (and (>= contract1 10) (>= income1 50)) (= class1 2)))
(assert (=> (and (< contract2 10) (< income2 30)) (= class2 0)))
(assert (=> (and (< contract2 10) (>= income2 30)) (= class2 2)))
(assert (=> (and (>= contract2 10) (< income2 50)) (= class2 1)))
(assert (=> (and (>= contract2 10) (>= income2 50)) (= class2 2)))
; Non-monotonicity constraint
(assert (and (<= contract1 contract2) (= income1 income2)))
(assert (not (<= class1 class2 )))
; Satisfiable ?
(check-sat)
; Logical model extraction
(get-model)
```

Figure 3: Z3 code of the decision tree with strong monotonicity constraint

and `contract2` and `income2` describe x'. The code also contains the non-monotonicity query for weak monotonicity wrt. "income". The last four lines of the code ask Z3 to check for satisfiability of all assertions and – if yes (Sat) – to return a logical model. The logical model gives an evaluation for the variables such that all assertions are fulfilled. For our example, it can be found in Figure 4. It matches the counter example of Section 2.

The counter example consists of a pair of data instances and their respective classes $((x, y), (x', y'))$ *as predicted by the decision tree.* As the approximation of the MUT by the decision tree will typically be imprecise, this counter example might not be valid for the MUT: it is a *candidate* counter example (candidate CEX). Hence, we next check the validity of $M(x) \not\approx_Y M(x')$. If it holds, a true counter example to monotonicity of the MUT (valid CEX) has been found. If not, $(x, M(x))$ and $(x', M(x'))$ are added to the oracle data in order to increase precision of the approximation in later steps.

When the output of Z3 is 'Unsat', we conclude that the generated decision tree is monotone wrt. F (but not necessarily the MUT M) and thus verification-based testing has been unable to compute a test case for non-monotonicity.

```
sat (model
  (define-fun contract1 () Int 9)
  (define-fun income1 () Real 30.0)
  (define-fun class1 () Int 2)
  (define-fun contract2 () Int 10)
  (define-fun income2 () Real 30.0)
  (define-fun class2 () Int 1))
```

Figure 4: Logical model for the query of Fig. 3

Algorithm 2 *prunInst* (Pruning data instances)

Input: (x, x') ▷ *A candidate pair*
φ ▷ *Logical formula*
Output: set of candidate pairs
1: cand-set:= $\emptyset$;
2: **for** i := 1 to n **do** ▷ *n: number of features*
3: $\psi := \varphi \wedge \neg(\text{name}_i 1 = x_i)$;
4: **if** SAT(ψ) **then**
5: cand-set := cand-set ∪ get-model(ψ);
6: **for** i := 1 to n **do**
7: $\psi := \varphi \wedge \neg(\text{name}_i 2 = x'_i)$;
8: **if** SAT(ψ) **then**
9: cand-set := cand-set ∪ get-model(ψ);
10: **return** cand-set;

Variation. The basic workflow of Figure 2 is complemented by *variation* techniques. Whenever we obtain a counter example which is not confirmed in the MUT, we need to make the approximating decision tree more precise. This is achieved by re-training. As this is a rather costly operation, we would like to avoid re-training for a single unconfirmed counter example and only re-train once we have collected a number of test pairs. This calls for a systematic generation of counter examples for which we need Z3 to produce several different logical models for the same logical query. To this end, we employ two pruning techniques cutting off certain parts of the search space of Z3 when computing logical models.

3.2.1 Pruning data instances. Our first strategy is to call the SMT solver Z3 several times and simply disallow it to return the same counter example again. For our running example, we can simply add `(assert (not (= contract1 9)))` to our query and re-run the SMT solver. We can similarly do so for the values of the other features. This way we can often generate a large number of *similar* counter examples. E.g., for our example Z3 then returns an instance with `contract1` being 8, then 7, and so on. Algorithm 2 describes how we generate new candidate pairs this way. Therein, $\text{name}_i 1$ ($\text{name}_i 2$) stands for the name of feature i in instance x (x', respectively), e.g. `income1`.

The disadvantage of this approach is that Z3 will give counter examples considering only a few number of branches in the decision tree. Hence, a major part of the decision tree paths will remain unexplored. Next we define a second strategy which achieves better coverage of the tree and thereby an improved *test adequacy*.

3.2.2 Pruning branches. In this case we use a global approach to traverse as many paths as possible. Once a test pair (x, x') is found, we identify the paths in the tree which this pair takes. Then we toggle the conditions on x's path and on x''s path, one after the other.

Algorithm 3 *prunBranch* (Pruning branches)

Input: (x, x') ▷ *A candidate pair*
tree ▷ *Decision tree*
φ ▷ *Logical formula*
Output: set of candidate pairs
1: cand-set:= $\emptyset$;
2: $(c_1, \ldots, c_m)$:= getPath(tree,x); ▷ *Path of x in tree*
3: **for** i := 1 to m **do** ▷ *toggle path conditions*
4: $\psi := \varphi \wedge \neg c_i$;
5: **if** SAT(ψ) **then**
6: cand-set := cand-set ∪ get-model(ψ);
7: $(c_1, \ldots, c_k)$:= getPath(tree,x'); ▷ *Path of x' in tree*
8: **for** i := 1 to k **do** ▷ *toggle path conditions*
9: $\psi := \varphi \wedge \neg c_i$;
10: **if** SAT(ψ) **then**
11: cand-set := cand-set ∪ get-model(ψ);
12: **return** cand-set;

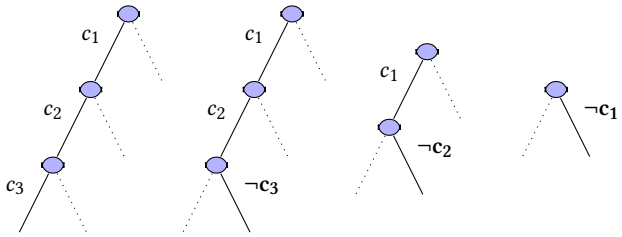

Figure 5: Illustration of condition toggling

Other toggling strategies are possible. This "condition toggling" follows the established strategy of determining path conditions in symbolic execution and systematically negating conditions for concolic testing [16, 29]. Figure 5 illustrates it on one path (the conditions in bold are added to the Z3 code) and Algorithm 3 gives the algorithm.

While the first pruning strategy tries to find new pairs in the local neighbourhood of a counter example, the second strategy globally searches for counter examples and thus achieves a better coverage. Algorithm 4 summarizes one such pass of counter example candidate generation from the decision tree.

White-box model improvement. Once we have collected a larger set of test pairs (all candidate counter examples), we test whether they are also counter examples to monotonicity for the MUT M. We furthermore check whether M's prediction differs from the decision tree's prediction on the candidates. If yes, they will be added to the oracle data to re-train the decision tree.

Overall algorithm. Algorithm 5 summarizes all these steps in one algorithm. It interleaves generation of test cases with monotonicity checking, i.e., after one call to *veriGen* it first checks all candidates and only if none of these are counter examples to monotonicity, it starts retraining. Two constants play a rôle in this algorithm: the constant `MAX_ORCL` fixes the number of data instances to be generated for initially training the decision tree. and (again) constant `MAX_SAMPLES` limits the number of samples we generate[6].

[6]Note that due to variation, *ts* might become slightly larger than `MAX_SAMPLES`.

Algorithm 4 *veriGen* (Test Generation for VBT)

Input: F ▷ *Set of features*
tree ▷ *trained decision tree*
Output: set of test cases
1: $cs := \emptyset$;
2: $\varphi :=$ tree2Logic(tree);
3: $\varphi := \varphi \wedge$ nonMonConstr(F);
4: **if** UNSAT(φ) **then** ▷ *No CEX cand. found*
5: **return** $\emptyset$;
6: $((x, y), (x', y')) :=$ getModel(φ); ▷ *Gen. candidates*
7: $cs := \{((x, y), (x', y'))\}$;
8: $cs := cs \cup$ prunInst(x, x', φ) $\cup$ prunBranch(x, x',tree,φ);
9: **return** cs;

Algorithm 5 *veriTest* (Verification-based testing)

Input: M ▷ *Model under test*
F ▷ *Set of features*
Output: counter example to monotonicity or empty
1: $orcl\_data := \emptyset$; $ts := \emptyset$; $cs := \emptyset$;
2: **while** $|orcl\_data| <$ MAX_ORCL **do**
3: $x :=$ random($\vec{X}$);
4: **if** $(x, M(x)) \notin orcl\_data$ **then**
5: $orcl\_data := orcl\_data \cup \{(x, M(x))\}$;
6: **while** $|ts| <$ MAX_SAMPLES **do**
7: tree := trainDecTree($orcl\_data$);
8: $cs :=$ veriGen(F,tree);
9: **for** $((x, y), (x', y')) \in cs$ **do** ▷ *Duplicates?*
10: **if** $(x, x') \notin ts$ **then**
11: $ts := ts \cup \{(x, x')\}$;
12: **else**
13: $cs := cs \setminus \{((x, y), (x', y'))\}$;
14: **if** $cs = \emptyset$ **then** ▷ *No new candidates?*
15: **return** Empty;
16: **for** $((x, y), (x', y')) \in cs$ **do**
17: **if** $M(x) \not\simeq_Y M(x')$ **then** ▷ *Valid CEX?*
18: **return** $((x, y), (x', y'))$
19: **if** $y \neq M(x)$ **then** ▷ *Different prediction?*
20: $orcl\_data := orcl\_data \cup \{(x, M(x))\}$;
21: **if** $y' \neq M(x')$ **then**
22: $orcl\_data := orcl\_data \cup \{(x', M(x'))\}$;
23: **return** Empty; ▷ *No counter example found*

4 EVALUATION

We have implemented adaptive random as well as verification-based testing for monotonicity in Python and have comprehensively evaluated VBT. The following research questions guided our evaluation. We broadly divide these questions into two categories. The first category concerns the comparison of VBT with existing techniques with respect to effectiveness and efficiency. The second category evaluates the performance of VBT itself.

RQ1. Effectiveness
How does VBT compare to existing testing approaches with respect to the error detection capabilities?

Table 2: Data sets and their characteristics

Name	#Features	#Group	#Instances	#TreeNodes
Adult	13	4	32561	4673
Diabetes	8	5	768	267
Mammographic	6	3	961	481
Car-evaluation	6	4	1728	167
ESL	4	2	488	295
Housing	13	3	506	107
Automobile	24	10	205	53
Auto-MPG	7	5	392	117
ERA	4	2	1000	87
CPU	6	5	209	11

RQ2. Efficiency
How does VBT compare to existing testing approaches with respect to the effort for error detection?

To analyse VBT itself we have focused on the following research questions:

RQ3. Approximation quality
Can the decision trees adequately represent black-box models?

RQ4. Strategy selection
Which pruning strategy performs better in computing non-monotonicity?

We have carried out the following experiments to evaluate these research questions.

RQ1. As there are no specific approaches for computing monotonicity of a given black-box model, we use property-based testing (i.e., a variant of QuickCheck [11] for Python) and (our own implementation of) adaptive random testing as baseline approaches to compare against. Our intention is to measure how well a technique is able to generate test cases revealing non-monotonicity of a given model. To this end, we have taken 8 ML algorithms from the state of the art ML library `scikit-learn` plus one monotonicity aware classifier [2], and trained them on ten data sets (see below) to generate 90 different predictive models. For these models it is first of all unknown whether they are monotone or not, so we lack a ground truth. The comparison is thus performed on the basis of just counting the number of models in which non-monotonicity is detected[7]. We perform the evaluation for weak (group) monotonicity as it is the standardly employed concept.

RQ2. To answer RQ2, we have carried out experiments in the same setting as considered for RQ1. We wanted to evaluate how efficient our verification-based testing approach is compared to adaptive random and property-based testing. To this end, we (a) determine the run time needed for test generation and checking, and (b) the number of generated test cases necessary for finding the first error or, in case of a failure in error detection, just the number of generated tests. In the latter case, this is the maximal number of samples to be considered by the approach, which is configurable for property-based testing and which is `MAX_SAMPLES` for ART and VBT (see below for values used).

[7]Note that none of the techniques produce false positives since they all perform a dynamic analysis.

RQ3. As we employ decision trees for computing candidate counter examples, the performance of VBT crucially depends on decision trees to adequately approximate the black-box model. "Adequately" here means adequate for the task of computing counter examples. In general, the decision tree model and the black-box will differ on some predictions. The inadequacy of the decision tree shows up whenever we need to re-train it several times in order to find a proper counter example. Hence, for RQ3 we determine the number of re-trainings for all 90 models and weak monotonicity.

RQ4. For test generation (Algorithm *veriGen*) we have implemented two different pruning strategies to achieve better coverage of the decision tree. So we wanted to find out which strategy is better in terms of finding counter examples. For the evaluation, we slightly change the setting. First, instead of only using weak monotonicity, we also check for strong monotonicity since we had the impression in initial experiments that the pruning strategies might behave differently for the weak and strong version. Second, we modify VBT such that it generates several counter examples (simply by not stopping it on the first one) and compute the achieved *detection rate*, i.e. number of detected errors divided by number of overall test cases $\frac{\#\text{errors}}{\#\text{test cases}}$.

4.1 Setup

We have collected our 10 data sets from the UCI machine learning repository[8] and the OpenML data repository[9]. These training data sets have also been used in existing works [21, 34] on monotonicity. Table 2 shows the data sets and their characteristics, i.e., the number of features, the size of the group (number of features in the group) and the number of data instances in the set. We have also computed the number of nodes of the decision tree model (column *#TreeNodes*) when being trained on the corresponding dataset to give a rough idea about the size of decision trees generated in VBT.

The *monotone features* are chosen based on our own assumptions about the domain and previous works. For instance, in case of the Adult data set, where a model predicts whether a person's income is at least $50,000, we check monotonicity with respect to the group consisting of age, weekly working hours, capital-gain and education level [36].

The eight classification algorithms which have been taken from `scikit-learn` are kNN, Neural Networks (NN), Random Forests (RF), Support Vector Machines (SVM), Naive Bayes (NB), AdaBoost, GradientBoost and Logistic Regression. We have used a linear kernel for SVM and a Gaussian version of NB for our experiments. There are several other ML algorithms which can be found in the library but the eight algorithms chosen here belong to the most basic family of ML classifiers. The 9th ML classifier is the monotonicity aware algorithm LightGBM [2] which has been specifically designed to construct models being monotone with respect to a given set of features. The monotonicity constraints can be enforced during the training phase of this algorithm. This should guarantee monotonicity but – as initial experiments have revealed – LightGBM does not entirely manage to rule out non-monotonicity. This classifier is an excellent benchmark for the three approaches since there are only a very few erroneous input pairs and the challenge is to generate exactly these as test cases.

[8]https://archive.ics.uci.edu/ml
[9]https://www.openml.org

Table 3: Number of non-monotonicity detections

Classifiers	*VBT*	*ART*	*PT*
k-NN	9	9	7
Logistic Regression	8	8	6
Naive Bayes	7	4	5
SVM	9	8	5
Neural Network	8	6	4
Random Forest	9	9	5
AdaBoost	8	7	5
GradientBoost	8	7	5
LightGbm	2	0	0
Overall	68	58	42

We have evaluated the accuracy score while generating predictive models and used the score to adjust the hyperparameters of the learning algorithms. While generating our white-box model (i.e. decision tree), we use the default hyperparameter settings from `scikit-learn`. This essentially does not fix the depth of the tree and hence causes the tree to overfit to the training data (which in our case is the oracle data). As we try to bring the decision tree as close to the black box model as possible, this is actually an advantage, not a disadvantage.

The input parameters of *artGen* and *veriTest* algorithms have been chosen based on execution time of some initial experiments. The parameter `POOL_SIZE` of the *artGen* algorithm is set to half of `INI_SAMPLES` which is 100; for `MAX_SAMPLES` we use 1000.

We have created oracle data in the verification-based testing approach by generating random data instances (90%) and also taking training data instances randomly (10%)[10]. This choice is influenced by the work of Johansson et al. [19] who found that using random data instances to approximate a model gives the best result.

We have used Hypothesis [1], a Python version of QuickCheck [11] as our property-based testing tool. Property-based testing allows users to specify the property to be tested. The parameters of this tool have been set in accordance with the *artGen* and *veriTest* algorithms (parameter for upper bound of test cases is 1000 like `MAX_SAMPLES`).

Finally, because all three approaches involve some sort of randomness, every experiment was carried out ten times. The results give the arithmetic mean over these ten runs. The experiments were run on a machine with 2 cores Intel(R) Core(TM) i5-7300U CPU with 2.60GHz and 16GB memory using Python version 3.6.

4.2 Results

Next, we report on the findings of our experiments while evaluating the research questions. All the necessary code and the datasets required to replicate the results reported here are available at https://github.com/arnabsharma91/MonotonicityChecker.

RQ1 - Effectiveness. Table 3 shows the results of the experiments for RQ1. It gives the number of models (out of 90 in total) for which our approach verification-based testing (VBT) and the two baselines adaptive random testing (ART) and property-based

[10]Note that for simplicity the stated algorithm does not include the training part.

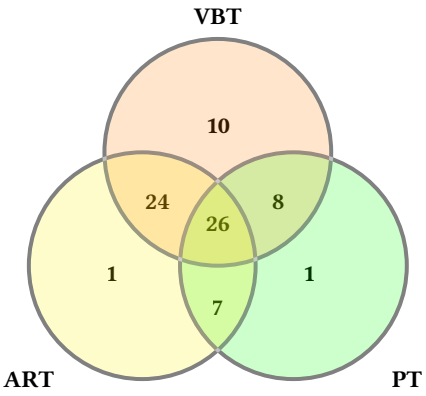

Figure 6: Venn diagram showing distribution of detected non-monotone models on approaches

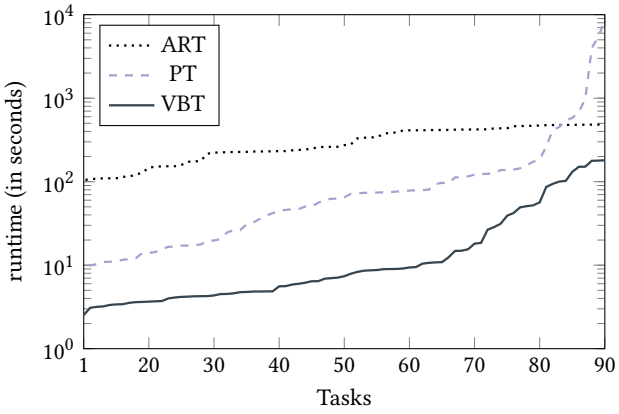

Figure 7: Run time in checking monotonicity

testing (PT) were able to detect non-monotonicity. Note again that the ground truth, i.e., which models are in fact non-monotone, is unknown, but all reported non-monotonicity cases are true positives. Per classifier 10 models were tested.

The results show that verification-based testing is more effective in detecting non-monotonicity than both adaptive random and property-based testing. It also shows that adaptive random testing can outperform (pure) property-based testing because – with the help of the distance metric – it more systematically generates test cases covering the test input space.

Another interesting result is the detection of non-monotonicity in two (out of 10) models generated by LightGBM (and the fact that ART and PT fail to detect it). LightGBM is a monotonicity-aware classifier which is supposed to just generate monotone models. For two of the training sets it has however failed to do so, resulting in a model which still has a small number of non-monotone pairs which VBT can find, but ART and PT cannot.

Given these differences in numbers, we also wanted to know whether the non-monotonicity detections of ART and PT are simply a subset of those of VBT. This is actually not the case. The Venn diagram in Figure 6 shows the distribution of counter examples onto the three approaches. 26 models are in the intersection of all three techniques. For the rest, only one or two of the approaches could detect non-monotonicity. The diagram also shows that there are 9 models for which both PT or ART can detect non-monotonicity, but VBT cannot. These models are all models trained on the ERA (6 models) and ESL (3 models) data sets. Looking at the models themselves, it turns out that their accuracy (wrt. the training data) is always very low (below 0.62 and for ERA even below 0.3). Hence, it seems that generalization from this training data is difficult for ML algorithms, and the decision trees in VBT seem to generalize in a different way than the black-box models and hence approximate them less well. This also shows in the results of RQ3 below concerning the ERA and ESL data sets.

Summarizing the findings of RQ1 in our experiments, we get

> On average, VBT is more effective than ART and PT in detecting non-monotonicity of black-box ML models.

RQ2 - Efficiency. For RQ2, we designed experiments to evaluate how efficient VBT is in comparison to ART and PT. Figure 7 shows the runtime of the three approaches for testing monotonicity. The x-axis enumerates the 90 tasks (i.e., models to be tested) where the tasks are sorted in ascending order of runtime per approach, and the y-axis gives the runtime for testing the task (in seconds, on a logarithmic scale). It shows that for all tasks VBT takes less time than both ART and PT even though our approach consists of several steps (including the training of a decision tree).

Adaptive random testing takes (on average) the same amount of time for all tasks. During ART, most of the time is needed for creating the input test cases which are "furthest" away from each other. As the size of test inputs is always the same for all the test cases, the time does not vary that much. On the other hand, property-based testing performs better than ART in most of the cases apart from some exceptions.

Second, for RQ2 we have determined the number of test cases generated during testing. All three approaches stop once they have detected the first counter example to monotonicity. Figure 8 shows the number of failed attempts (i.e., generated test cases before finding the first counter example) for each of the classifiers averaging over all the datasets. Our experimental results suggest that VBT always needs the least amount of test cases to testify non-monotonicity.

Note that our approach has two possible execution instances when not finding counter examples: (1) the SMT solver might continuously generate counter example candidates which all fail to be real counter examples in the model and thus VBT successively retrains the tree until `MAX_SAMPLES` candidates have been generated, or (2) it immediately stops because the first decision tree is already monotone and the SMT solver finds no counter example at all. In the latter case, the number of failed attempts is 0 (which is favourable for a low number of attempts). However, this only occurred in 7 out of the 90 models.

Summarizing the findings of RQ2 in our experiments, we get

> On average, VBT is more efficient than ART and PT in detecting non-monotonicity of black-box ML models.

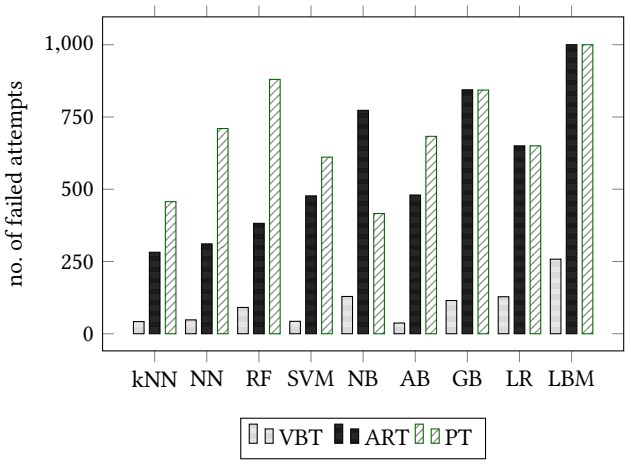

Figure 8: Number of failed attempts

For the next two research questions we look at verification-based testing only.

RQ3 - Approximation quality. Table 4 shows the mean number of re-trainings of the decision tree per classifier and data set (mean over 10 runs). For those cases where VBT could not find any non-monotonicity, we have written '-'. As the results show, the number of re-trainings is high for the monotonicity aware algorithm LightGBM (LBM), which is consistent with the results shown in Figure 8. Apart from the classifier, the data set seems to influence the number of retrainings (e.g., Adult needs a large number of re-trainings). In general, the numbers are – however – relatively low (≤ 10). Note that there are also 27 models for which *no* retraining at all is needed.

Hence, we conclude the following.

> On average, the approximation quality of decision trees in VBT is good enough to only require a small number of retrainings for non-monotonicity detection.

RQ4 - Strategy Selection. For RQ4, we modified VBT as to not stop upon the first valid counter example. Figures 9 and 10 show the *detection rates* (number of detected counter examples divided by number of test cases) of the two pruning strategies alone in computing strong (Fig. 9) and weak monotonicity (Fig. 10), respectively. Note the difference in the maximal detection rate which is 0.5 for strong and only 0.25 for weak monotonicity. Our experimental results suggest that on a large number of classifiers branch pruning is better or equal to feature pruning for strong monotonicity. On the other hand, for weak monotonicity this is the opposite. This can partly be explained on the decision tree itself: since weak monotonicity requires all but the values of monotone features to be the same in a pair, branch pruning cannot exhibit its full power.

The results suggest the following.

Table 4: Mean number of re-trainings

Data \ *Classifier*	kNN	NN	RF	SVM	NB	AB	GB	LR	LBM
Adult	4	6	19	7	26	1	25	37	-
Auto	0	1	3	0	-	3	2	0	35
Car	0	0	0	0	0	-	1	5	-
CPU	0	0	1	0	26	0	9	1	-
Diabetes	3	0	1	0	4	8	2	7	-
ERA	1	-	-	-	-	0	-	-	-
ESL	-	-	1	0	-	-	-	-	-
Housing	0	0	1	0	1	0	1	0	-
Mammo	1	0	0	1	1	0	1	5	-
Mpg	0	10	0	1	0	0	0	5	18

> On average, branch pruning achieves a higher detection rate than feature pruning for strong monotonicity and vice versa for weak monotonicity.

4.3 Limitations and Threats to Validity

Since we employ an SMT solver for monotonicity computation, verification-based testing is restricted to feature values allowed by the solver. Currently, we have data sets with integer and real values. For other domains of feature values, an encoding would be necessary. This is however often done by ML algorithms within preprocessing steps anyway, so we could easily make use of existing techniques there.

Threats to the validity of results are the choice of data sets and ML algorithms and the choice of feature groups for monotonicity checking. For the ML algorithms, we are confident that we have covered all sorts of basic classifiers in usage today (of course, there are in addition numerous specialised classifiers which however often make use of the base techniques). As we have taken a number of different, publicly available data sets for machine learning, we are furthermore confident that our data sets are diverse enough to exhibit different properties of the approaches and reflect real data sets. In particular, the ERA data set exposes interesting properties of verification-based testing.

A threat to the internal validity is the high degree of randomness involved in the techniques. First, a number of classifiers use randomized algorithms for generating models. Thus, in principle we might get one monotone and one non-monotone model when training *with the same classifier on exactly the same data set.* To ensure fairness during comparison, all three approaches were always started with the same model as input (training of MUTs is external to testing). Second, all three approaches themselves randomly generate (at least some) data instances (ART and PT as potential test cases, VBT for oracle data). VBT in addition uses a decision tree training algorithm which itself involves randomness. Hence, our decision trees can vary from one run to the next, and this stays so even if we would fix the oracle data. To mitigate these threats, all experiments were performed 10 times and the results give the mean over these 10 runs.

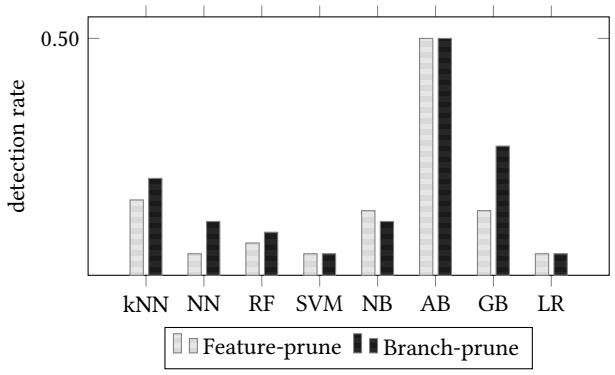

Figure 9: Performance of branch and feature pruning in computing strong monotonicity

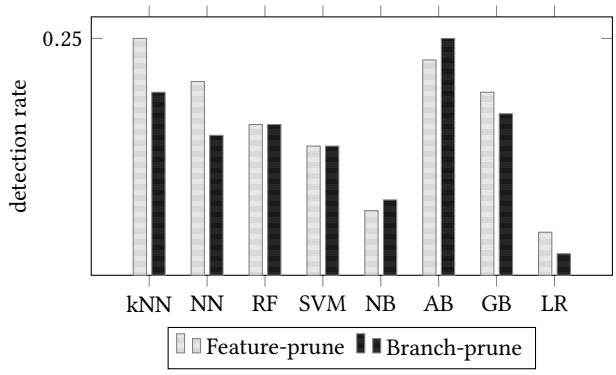

Figure 10: Performance of branch and feature pruning in computing weak monotonicity

5 RELATED WORK

We divide our discussion of related works in three parts. First, we discuss some works incorporating monotonicity in the predictive model, then mention some recent techniques for machine learning testing and third discuss approaches using model inference in testing.

Generating monotone models. The existing works in monotonicity focus on specific ML algorithms. In [6], Archer et al. first propose building a monotone neural network model by adjusting the contribution of training samples in the training process. There exists some follow up works which constrain the parameters of the neural network algorithm to enforce monotonicity [13, 32]. In a more recent work You et al. [36] propose an approach to generate a guaranteedly monotone deep lattice network with respect to a given set of features. Lauer et al. [22] enforce monotonicity in support vector machines (with linear kernels) by constraining the derivative to be positive within a specified range. Riihimäki et al. [28] build a Gaussian monotone model by using virtual derivative observations. In a follow up work, Siivola et al. [31] give an approach based on the same idea to detect monotonicity only for Gaussian distributions. Although using derivatives of the function can work for some ML algorithms, it cannot be generalized and is not possible to use in algorithms where the learned functions (i.e., models) are non-linear.

Validating models. There are a number of recent works which aim at validating properties of predictive models, none of which however have looked at monotonicity. In [18], Huang et al. propose *robustness* as a safety property and give a verification technique showing that a Deep Neural Network (DNN) guarantees postconditions to hold on its outputs when the inputs satisfy a given precondition. Gehr et al. [15] use abstract interpretation to verify robustness of DNN models. Pei et al. [25] propose the first white-box testing technique to test DNNs. They use neuron coverage as a criterion to generate the test cases for the predictive model. Sun et al. [33] propose concolic testing to test the robustness property of DNNs. The authors use a set of coverage requirements (such as neuron, MC/DC and neuron boundary coverage, Lipschitz continuity) to generate test inputs.

Recently, Sharma et al. [30] have proposed a property called *balancedness* on the learning algorithm. They perform specific transformations on the training data and check whether the learning algorithm generates a different predictive model after applying such transformations. This work thus focusses on testing the ML algorithm itself, not the model.

In [14], Galhotra et al. perform black-box testing to check *fairness* of the predictive model. Basically, they use random testing with confidence driven sampling. It has the drawback of generating completely random sets of test data without considering the structure of the model.

The work closest to us is that of Agarwal et al. [3]. They also study fairness testing, but compared to Galhotra et al. [14] they aim at a more systematic generation of test inputs. To this end, they employ LIME [27] (a tool for generating *local* explanations for predictions) to generate a *partial* decision tree (often just a path in the tree) from the black-box model. On this path, they use dynamic symbolic execution to generate multiple test cases, much alike we do. The difference to our work is that we generate a decision tree approximating the *entire* black-box model under test, and – more importantly – we use the generated white-box model for the *computation* of test inputs (potential counter examples to monotonicity). We thus achieve a targeted test case generation.

Testing via model inference. The use of learning in testing has long been considered in the field of model-based testing. Therein, learning is used to extract a model of the system under test. Such models most often are some sort of automaton (finite state machine) and learning is based on Angluin's L^* algorithm [5]. For a survey of techniques see [4, 23].

In contrast to this, we employ machine learning techniques to infer a model. The inference of a decision tree describing the behaviour of software has already been pursued by Papadopoulus and Walkinshaw [24] as well as Briand et al. [8]. The former – similar to us – translate the decision tree to logic in order to have Z3 generate test inputs covering different branches of the tree. However, they do not employ the tree to generate counter examples to the property to be tested. Thus, the advantage of having a verifiable white-box model for targeted test input generation is not utilized. Briand et

al. on the other hand use the decision tree in a semi-automated approach to the re-engineering of test suites. This approach requires the manual inspection of the decision tree by testers. A survey on inference-driven techniques is given in [35].

6 CONCLUSION

In this work, we have defined the property of monotonicity of ML models and have proposed a novel approach to testing monotonicity. Our technique approximates the black-box model by a white-box model and applies SMT solving techniques to compute monotonicity on the white-box model. We have evaluated the effectiveness and efficiency of our approach by applying it to several ML models and found our approach to outperform both adaptive random and property-based testing.

As future work, we plan to apply this scheme to validate other important properties of ML models. Our white-box model easily allows for checking other properties, like for instance fairness, just by applying a different check on the generated SMT code. Also, we would like to improve our framework by using incremental learning to avoid re-training of the entire decision tree.

ACKNOWLEDGEMENTS

We would like to thank Vitalik Melnikov and Eyke Hüllermeier for several discussions on monotonicity in machine learning, and Cedric Richter for his feedback on the experimental evaluation.

REFERENCES

[1] 2019. Hypothesis. https://github.com/HypothesisWorks/hypothesis. (2019).
[2] 2019. LightGBM. https://github.com/Microsoft/LightGBM. (2019).
[3] Aniya Aggarwal, Pranay Lohia, Seema Nagar, Kuntal Dey, and Diptikalyan Saha. 2019. Black box fairness testing of machine learning models. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*. 625–635. https://doi.org/10.1145/3338906.3338937
[4] Bernhard K. Aichernig, Wojciech Mostowski, Mohammad Reza Mousavi, Martin Tappler, and Masoumeh Taromirad. 2018. Model Learning and Model-Based Testing, See [7], 74–100. https://doi.org/10.1007/978-3-319-96562-8_3
[5] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75, 2 (1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6
[6] Norman P Archer and Shouhong Wang. 1993. Application of the back propagation neural network algorithm with monotonicity constraints for two-group classification problems. *Decision Sciences* 24, 1 (1993), 60–75.
[7] Amel Bennaceur, Reiner Hähnle, and Karl Meinke (Eds.). 2018. *Machine Learning for Dynamic Software Analysis*. Lecture Notes in Computer Science, Vol. 11026. Springer. https://doi.org/10.1007/978-3-319-96562-8
[8] Lionel C. Briand, Yvan Labiche, Zaheer Bawar, and Nadia Traldi Spido. 2009. Using machine learning to refine Category-Partition test specifications and test suites. *Information & Software Technology* 51, 11 (2009), 1551–1564. https://doi.org/10.1016/j.infsof.2009.06.006
[9] Nicholas Carlini and David A. Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *2017 IEEE Symposium on Security and Privacy, SP*. 39–57. https://doi.org/10.1109/SP.2017.49
[10] Tsong Yueh Chen, Hing Leung, and I. K. Mak. 2004. Adaptive Random Testing. In *ASIAN (Lecture Notes in Computer Science)*, Michael J. Maher (Ed.), Vol. 3321. Springer, 320–329. https://doi.org/10.1007/978-3-540-30502-6_23
[11] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *(ICFP '00)*, Martin Odersky and Philip Wadler (Eds.). ACM, 268–279. https://doi.org/10.1145/351240.351266
[12] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24
[13] Charles Dugas, Yoshua Bengio, François Bélisle, Claude Nadeau, and René Garcia. 2009. Incorporating Functional Knowledge in Neural Networks. *J. Mach. Learn. Res.* 10 (2009), 1239–1262. https://dl.acm.org/citation.cfm?id=1577111
[14] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. 2017. Fairness testing: testing software for discrimination. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 498–510.
[15] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *IEEE Symposium on Security and Privacy, SP*. 3–18. https://doi.org/10.1109/SP.2018.00058
[16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. 213–223. https://doi.org/10.1145/1065010.1065036
[17] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. 2019. A Survey of Methods for Explaining Black Box Models. *ACM Comput. Surv.* 51, 5 (2019), 93:1–93:42. https://doi.org/10.1145/3236009
[18] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In *Computer Aided Verification - 29th International Conference, CAV*. 3–29. https://doi.org/10.1007/978-3-319-63387-9_1
[19] Ulf Johansson and Lars Niklasson. [n. d.]. Evolving decision trees using oracle guides. In *Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining, CIDM 2009*. 238–244. https://doi.org/10.1109/CIDM.2009.4938655
[20] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252
[21] Wojciech Kotlowski and Roman Slowinski. 2009. Rule learning with monotonicity constraints. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009*. 537–544. https://doi.org/10.1145/1553374.1553444
[22] Fabien Lauer and Gérard Bloch. 2008. Incorporating prior knowledge in support vector regression. *Machine Learning* 70, 1 (2008), 89–118. https://doi.org/10.1007/s10994-007-5035-5
[23] Karl Meinke. 2018. Learning-Based Testing: Recent Progress and Future Prospects, See [7], 53–73. https://doi.org/10.1007/978-3-319-96562-8_2
[24] Petros Papadopoulos and Neil Walkinshaw. 2015. Black-Box Test Generation from Inferred Models. In *RAISE*, Rachel Harrison, Ayse Basar Bener, and Burak Turhan (Eds.). IEEE Computer Society, 19–24. https://doi.org/10.1109/RAISE.2015.11
[25] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles,*. 1–18. https://doi.org/10.1145/3132747.3132785
[26] R. Potharst and A. J. Feelders. 2002. Classification Trees for Problems with Monotonicity Constraints. *SIGKDD Explor. Newsl.* 4, 1 (June 2002), 1–10. https://doi.org/10.1145/568574.568577
[27] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1135–1144. https://doi.org/10.1145/2939672.2939778
[28] Jaakko Riihimäki and Aki Vehtari. 2010. Gaussian processes with monotonicity information. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS*. 645–652. http://proceedings.mlr.press/v9/riihimaki10a.html
[29] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 263–272. https://doi.org/10.1145/1081706.1081750
[30] Arnab Sharma and Heike Wehrheim. 2019. Testing Machine Learning Algorithms for Balanced Data Usage. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST*. 125–135. https://doi.org/10.1109/ICST.2019.00022
[31] Eero Siivola, Juho Piironen, and Aki Vehtari. 2016. Automatic monotonicity detection for Gaussian Processes. *arXiv preprint arXiv:1610.05440* (2016).
[32] Joseph Sill. 1997. Monotonic Networks. In *Advances in Neural Information Processing Systems*. 661–667. http://papers.nips.cc/paper/1358-monotonic-networks
[33] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic testing for deep neural networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE*. 109–119. https://doi.org/10.1145/3238147.3238172
[34] Ali Fallah Tehrani, Weiwei Cheng, Krzysztof Dembczynski, and Eyke Hüllermeier. 2011. Learning Monotone Nonlinear Models Using the Choquet Integral. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML*. 414–429. https://doi.org/10.1007/978-3-642-23808-6_27
[35] Neil Walkinshaw. 2018. Testing Functional Black-Box Programs Without a Specification, See [7], 101–120. https://doi.org/10.1007/978-3-319-96562-8_4
[36] Seungil You, David Ding, Kevin Robert Canini, Jan Pfeifer, and Maya R. Gupta. 2017. Deep Lattice Networks and Partial Monotonic Functions. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems*. 2985–2993. http://papers.nips.cc/paper/6891-deep-lattice-networks-and-partial-monotonic-functions