MLCHECK– Property-Driven Testing of Machine Learning Classifiers

Arnab Sharma
University of Oldenburg
Oldenburg, Germany
arnab.sharma@uol.de

Caglar Demir
Paderborn University
Paderborn, Germany
caglar.demir@upb.de

Axel-Cyrille Ngonga Ngomo
Paderborn University
Paderborn, Germany
axel.ngonga@upb.de

Heike Wehrheim
University of Oldenburg
Oldenburg, Germany
heike.wehrheim@uol.de

***Abstract*—An increasing amount of software with machine learning components is being deployed. This poses the question of quality assurance for such components: how can we validate whether specified requirements are fulfilled by a machine learned software? Current testing and verification approaches either focus on a single requirement (e.g., fairness) or specialize in a single type of machine learning model (e.g., neural networks). We propose the property-driven testing of machine learning models. Our approach MLCHECK encompasses (1) a language for property specification, and (2) a technique for systematic test case generation. The specification language is comparable to property-*based* testing languages. The test case generation employs an elaborate verification method for a systematic, property-dependent construction of test suites, without additional user-supplied generator functions. We evaluate MLCHECK using requirements and data sets from three different application areas (software discrimination, learning on knowledge graphs and security). Our evaluation shows that in addition to its generality, MLCHECK can outperform specialised testing approaches while having a comparable runtime.**

***Index Terms*—Machine Learning Testing, Decision Tree, Neural Network, Property-Based Testing.**

I. INTRODUCTION

The importance of quality assurance for applications developed using machine learning (ML) increases steadily as they are deployed in a growing number of domains. Still, developers need to make sure that their software—whether learned or programmed—satisfies certain specified requirements. Currently, two orthogonal approaches can be followed to achieve this goal: (A) employing an ML algorithm guaranteeing some requirement per design, or (B) validating the requirement on the model generated by the ML algorithm.

Both approaches have shortcomings: Approach A is only available for a handful of requirements (e.g., fairness, monotonicity, robustness) [1]–[3]. Moreover, such algorithms cannot ensure the complete fulfillment of the requirement. For example, Galhotra et al. [4] have found fairness-aware ML algorithms to generate unfair predictions, and Sharma et al. [5] detected non-monotonic predictions in supposedly monotone classifiers. For robustness to adversarial attacks, the algorithms can only reduce the attack surface. Approach B, on the other hand, is only possible if a validation technique exists which is applicable to (1) the specific *type* of machine learning classifier under consideration (i.e., neural network, SVM, etc.) and (2) the specific *property* to be checked. Current validation techniques are restricted to either a single model type or a single property (or even both).

We propose *property-driven testing* as a validation technique for ML models overcoming the shortcomings of approach B. Our technique allows developers to specify the property under interest and—based on the property—performs a targeted generation of test cases. The target is to find test cases *violating* the property. The approach is applicable to arbitrary types of *non-stochastic* properties and model agnostic. We consider the model under test (MUT) as a black-box of which we just observe the input-output behaviour. To achieve a systematic generation of test cases specific to both MUT and property, we train a second *white-box model* approximating the MUT by using its predictions as training data. Knowing the white box's internal structure, we apply state-of-the-art verification technology to *verify* the property. A verification result of "failure" (property not satisfied) is then accompanied by (one or more) counterexamples, which we subsequently store as test inputs whenever they are failures for the MUT as well.

We currently employ two types of ML models as approximating white-boxes: decision trees and neural networks (NNs). While no prior knowledge is required pertaining to the internal structure of the model under test, the internals of the white-box model are accessible to verification. Test generation proceeds by (1) encoding both property and white-box model as logical formulae and (2) using an SMT (Satisfiability Modulo Theories) solver to check their satisfiability. Counterexamples in this case directly come in the form of satisfying assignments to logical variables which encode feature and class values. Due to the usage of a proxy white-box model, test generation is an iterative procedure: whenever a counterexample on the white-box model is found which is not valid for the MUT, the white-box model gets *retrained*. This way, the approximation quality of the white-box model is successively improved.

We have implemented our approach in a tool called MLCHECK and evaluated it on requirements of three different application areas:

- **Software discrimination** studies whether ML models give predictions which are (un)biased with respect to some attributes. Different definitions of such fairness requirements exist (see [6]); we exemplarily use *individual discrimination* [4].

- **Knowledge graphs** are a family of knowledge representation techniques. We consider learning classifiers for entities based on embeddings [7] and exemplarily consider the properties of *class disjointness* and *subsumption*.
- **Security** of ML applications investigates if ML models are vulnerable to attacks, i.e., can be manipulated as to give specific predictions. We exemplarily study vulnerability to *trojan attacks* [8].

Overall, this paper makes the following contributions:

- we present a language for specifying properties on machine learning models,
- we propose a method for systematic test case generation, driven by the property and ML model under consideration, and
- we systematically evaluate our approach in three different application areas employing 56 models under test generated from 24 data sets.[1]

II. Foundations

We start by introducing some basic notation and formally defining the problem of property-driven testing of ML models.

The model generated by a supervised ML approach is a function

$$M : X_1 \times \ldots \times X_n \to Z_1 \times \ldots \times Z_m \ ,$$

where X_i is the value set of *feature* i, $1 \leq i \leq n$, and every Z_j, $1 \leq j \leq m$, contains the *classes* for the jth *label*. Instead of numbering features and labels, we also use feature names $F_1, \ldots, F_n$ and label names $L_1, \ldots, L_m$, and let $F = \{F_1, \ldots, F_n\}$, $L = \{L_1, \ldots, L_m\}$. We freely mix numbers and names in our formalizations. When $m > 1$, the learning problem is a *multilabel* classification problem; when $|Z_i| > 2$ for some i, the learning problem is a *multiclass* classification problem. In case $|Z_i| = 2$ for all i, it is a *binary* classification problem.

We write $\vec{X}$ for $X_1 \times \ldots \times X_n$, $\vec{Z}$ for $Z_1 \times \ldots Z_m$ and use an index (like in x_i) to access the i-th component. The training data consists of elements from $\vec{X} \times \vec{Z}$, i.e., data instances with known associated class labels. During the prediction, the generated predictive model assigns classes $z \in \vec{Z}$ to a data instance $x \in \vec{X}$. Based on this formalization, we define properties relevant to our three application areas.

Software discrimination. *Fairness* of predictive models refers to the absence of discrimination of individuals due to certain feature values. More precisely, a model has no individual discrimination [4] if flipping the value of a single, so called *sensitive* feature, while keeping the values of other features does not change the prediction.

Definition 1. *A predictive model M is* individually fair *with respect to a sensitive feature $s \in \{1, \ldots, n\}$ if for any two data instances $x, y \in \vec{X}$ the following holds:*

$$(x_s \neq y_s) \wedge (\forall i, i \neq s, x_i = y_i) \Rightarrow M(x) = M(y) \ .$$

[1]The tool and all data to replicate the results mentioned in this paper can be found at https://github.com/arnabsharma91/MlCheck

TABLE I: Characteristics of properties

	Hyperproperty	Binary	Multiclass	Multilabel
Fairness	✔	✔	✘	✘
Subsumption	✘	✔	✘	✔
Disjointness	✘	✔	✘	✔
Trojan attack	✘	✘	✔	✘

Fairness is (most often) a requirement for applications which perform binary classification.

Knowledge graphs. We aim to learn categorizations of entities according to concepts given in a fixed ontology. This is a multilabel classification problem–every concept is a label name–and for every label, we perform a binary classification (instance x is or is not a dog). In the following, we treat the two classes 0 and 1 of every label as boolean values.

Definition 2. *A* concept relationship *is a boolean expression over the label names L. A predictive model M is* respecting concept relationship *φ if for any data instance x the formula*

$$\varphi[L_i := M(x)_i, 1 \leq i \leq m] \textit{ is true.}$$

Here, $\varphi[L_i := M(x)_i, 1 \leq i \leq m]$ stands for the formula φ in which label names are replaced by the corresponding (boolean) values obtained from a prediction. Of frequent interest are the concept relationships *subsumption*, and *disjointness*. For an animal ontology, desired concept relationships might for instance be described by formulae $\varphi_1 : \mathit{dog} \Rightarrow \mathit{animal}$ (dogs are animals) or $\varphi_2 : \mathit{dog} \Rightarrow \neg \mathit{cat}$ (a dog is not a cat).

Security. Our third application area is security. Here, we exemplarily consider *trojan attacks*. Trojan attacks are input patterns for which—when present in a data instance—the attacker expects to yield a certain prediction.

Definition 3. *Let $T \subseteq \{i_1, \ldots, i_\ell\}$ be a set of* trigger features*, $\mathbf{t} \in \vec{X}$ a* trigger *vector and $\mathbf{z} \in \vec{Z}$ a target prediction. A predictive model M is* vulnerable to attack *$(T, \mathbf{t}, \mathbf{z})$ if for any data instance $x \in \vec{X}$ the following holds:*

$$\forall t \in T : x_t = \mathbf{t}_t \Rightarrow M(x) = \mathbf{z} \ .$$

Trojan attacks are often run on image classifiers. There are specific training as well as manipulation techniques for ML models which make models vulnerable to trojan attacks [9]. Note that these are different from adversarial attacks [10].

These three areas and their properties have complementary characteristics (see also Table I), and in the evaluation can as such demonstrate the versatility of our specification and testing approach. Fairness is a *hyperproperty* [11] as it requires comparing the prediction of the model on *two* inputs. Disjointness, subsumption and trojan vulnerability are trace properties; their violation can be checked on a single input. Furthermore, the required classifiers differ (Table I).

III. Property-Driven Testing

Our objective is the development of a property-driven tester for ML models. Our approach comprises the following core contributions:

```
s = ...                                          # Sensitive feature
for i in range(0, f_size - 1):                   # Assumption
  if(i == s):
    Assume('x[i] != y[i]', i)
  else:
    Assume('x[i] == y[i]', i)
Assert('M.predict(x) == M.predict(y)')           # Assertion
```

Fig. 1: Property specification for fairness

- a *language* for property specification and
- a *method* for targeted test case generation.

Alike property-based testing, we provide a simple domain-specific language for specifying non-stochastic properties. Contrary to property-based testing, we supply *property-driven* test suite generation *without* the user needing to write test case generator functions (strategies) herself.

A. Property Specification

In property-based testing [12], software developers specify properties about functions (of their programs), and the testing tool generates inputs for checking such properties. Often, properties are specified in an *assume/assert* style. The assert statement defines the conditions to be satisfied by a function's output; the assume statement specifies conditions on inputs. The property is violated if a test input can be found which satisfies the assume statement but where the output of the function applied to this input violates the assert statement.

Our domain-specific language follows this assume/assert style and uses Python as base language. Assume and assert statements are calls to functions `Assume` and `Assert`. These functions can be used within arbitrary Python code. To allow this Python code to refer to characteristics of the current model under test, the developer can use predefined variables and functions: (1) `f_size` (the number of features n), `F` (set of all feature names) and its elements, (2) similarly `l_size` (the number of labels m), `L` (set of all label names) and its elements plus (3) the function `predict` (the prediction of some model M).

Calls to the assume and assert functions take the following form:

```
Assume('<condition>',<arg1>, ...)
Assert('<condition>',<arg1>, ...)
```

The first parameter is a string containing the logical condition (on either inputs or outputs) which our tool parses to translate it to code for the SMT solver used for verification. The remaining arguments supply the values of variables occurring in the condition. The model under test has to be defined (trained or supplied as input) beforehand and can be referred to in the assert by a variable name (in our example, `M`). Figure 1 shows the specification of the property of Definitions 1.

B. Test Data Generation

For test data generation, we employ a technique called *verification-based testing*, first proposed in [5]. Verification-based testing performs formal *verification* of the property to be checked via SMT (satisfiability modulo theories) solving. Since we treat the model under test (MUT) as black-box (and since we aim at a testing technique applicable to *any* kind of machine learning model), verification first of all requires the existence of a verifiable *white-box* model. To this end, we train a white-box model *approximating* the MUT using predictions of the MUT as training data. On the white-box model, we verify the property, and use counterexamples to the property as test inputs. As the white-box model is only an approximation of the MUT, not all such counterexamples must be valid in the MUT. In case of counterexamples being invalid for the MUT, we do not include them in the test suite and instead retrain the white-box model to enhance its approximation quality.

The overall workflow of test data generation is depicted in Figure 2. Inputs are the model under test (MUT) and the property specification, the output is a test suite. We briefly discuss all steps in the sequel.

White-Box Model Training. The white-box model on which we *verify* the property is generated from predictions of the black-box model (MUT). To this end, we generate training data for white-box model from randomly chosen instances together with the MUT's predictions on these instances. Currently, our approach employs two types of white-box models which the user can choose from: decision trees and NNs. During evaluation, we compare them with respect to efficiency and effectiveness in generating test inputs (see Section V).

Formula Generation. Property and white-box model are translated to logical formulae. The construction guarantees that these formulae in conjunction are satisfiable if and only if the property does *not* hold for the white-box model. Our aim is the generation of test inputs *violating* the property. The translation itself is detailed in Section IV.

SMT Solving and Augmentation. Next, the SMT solver Z3 [13] is used to check satisfiability. When the formula is satisfiable, we extract the *logical model* of the formula. This logical model is a counterexample to the property, i.e., gives us values of data instances and predicted classes violating the property. Such a counterexample serves as a test input and thus becomes part of the *test suite* (unless it is no counterexample for the MUT). To generate several test inputs, we furthermore use an *augmentation* phase and let the SMT solver construct further logical models.

Retraining. As verification takes place on the white-box model, not every thus computed counterexample is a valid counterexample for the MUT (which is only approximated by the white-box model). Therefore, we compare the prediction of the white-box model on generated test inputs with that of the MUT. In case of differences, the test input plus MUT prediction is added to the training set for the white-box model. After having collected several such invalid counterexamples, the white-box model is retrained.

These steps are repeated until a user-definable maximum number of samples has been reached.

IV. Encodings

The generation of the logical formula requires an encoding of the white-box model and of the specified property.

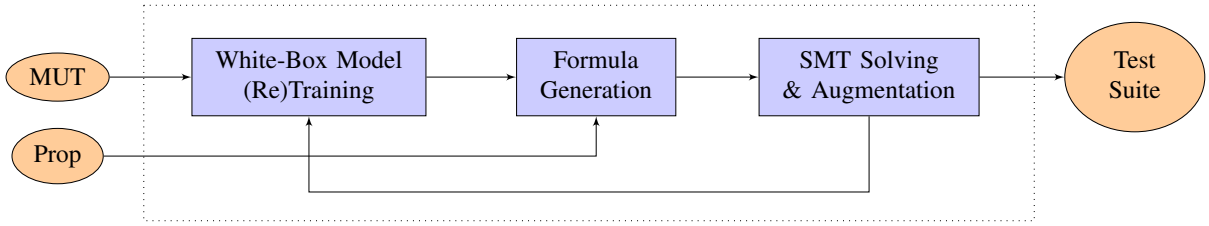

Fig. 2: Workflow of Test Data Generation

A. White-Box Model Encoding

Our approach currently involves two sorts of white-box models for verification-based testing, decision trees and neural networks. We briefly formalize their encodings as a number of logical constraints next.

Decision trees. A decision tree is a tree in which every edge between a node and its children is labelled with a boolean condition on features values, and every leaf is labelled with a prediction giving class values for all labels. Formally, for every level i in the tree, we let $s_j^{(i)}$ be the j-th node and $s_{pre(j)}^{(i-1)}$ be its predecessor on level $i-1$. We let $cond_{pre(j)}^{(i)}$ be the condition on the edge from $s_{pre(j)}^{(i-1)}$ to $s_j^{(i)}$, and $pred_j^{(i)}$ be the prediction associated to a leaf node $s_j^{(i)}$. We assume predictions to take the form $\bigwedge_{\ell \in L}(\ell = c)$ where $c \in Z_\ell$.

For the encoding, we introduce one boolean variable per node in the tree and one variable $class_\ell$ for every label $\ell \in L$. The constraints are as follows. We get one constraint for the root of the node on level 0: $C_{root} \equiv s_1^{(0)}$. Thus, the boolean variable for the root node is always true. For every further *inner* node $s_j^{(i)}$ we get one constraint

$$\begin{aligned} C_j^{(i)} \quad \equiv \quad & (s_{pre(j)}^{(i-1)} \wedge cond_{pre(j)}^{(i)} \wedge s_j^{(i)}) \\ & \vee ((\neg s_{pre(j)}^{(i-1)} \vee \neg cond_{pre(j)}^{(i)}) \wedge \neg s_j^{(i)}) \end{aligned}$$

Thus, node variables become true when their successor node is true and the condition on the edge holds. For every *leaf* $s_j^{(i)}$ with prediction $\bigwedge_{\ell \in L}(\ell = c)$ we get the constraint[2]:

$$C_j^{(i)} \equiv \bigwedge_{\ell \in L}(class_\ell = c)$$

Neural networks. The second option is to train a neural network as white-box model. We assume training to supply us with a feed forward neural network with ReLU (Rectified Linear Unit) activation functions modelling the function $M : \vec{X} \to \vec{Z}$ with $n = |\vec{X}|$ input nodes, $m = |\vec{Z}|$ output nodes (in case of a multilabel classifier), $m = |Z_1|$ (in case of a single label), and k hidden layers with n_i neurons each, $1 \leq i \leq k$. We set $n_0 = n$, and $n_{k+1} = m$. Attached to each connection from neuron j in layer i to neuron l in layer $i+1$ is a *weight* $w_{jl}^{(i)}$. Every neuron is equipped with a *bias* $b_j^{(i)}$.

The encoding of such neural networks is in spirit similar to other encodings, e.g., by Bastani et al. [14]. We use two real-valued variables $in_l^{(i)}$ and $out_l^{(i)}$ for neuron l on layer i and a boolean variable $class_\ell$ for every label $\ell \in L$.

For every hidden layer i, $1 \leq i \leq k$, we generate two constraints, one describing conditions about the inputs:

$$C_{in}^{(i)} \quad \equiv \quad \bigwedge_{l=1}^{n_i}(in_l^{(i)} = \Sigma_{j=1}^{n_{i-1}} w_{jl}^{(i-1)} out_j^{(i-1)} + b_l^{(i)})$$

the other about the outputs:

$$\begin{aligned} C_{out}^{(i)} \quad \equiv \quad & \bigwedge_{l=1}^{n_i}(in_l^{(i)} < 0 \wedge out_l^{(i)} = 0) \\ & \vee (in_l^{(i)} \geq 0 \wedge out_l^{(i)} = in_l^{(i)}) \end{aligned}$$

Basically, C_{out} encodes the ReLU activation function, and C_{in} fixes the input as the weighted sum over all outputs from nodes of the previous layer plus the bias term. For the output layer $k+1$, we just require constraint $C_{in}^{(k+1)}$. In case of a single label classifier, the predicted class is determined by the output neuron with the maximal input, and we therefore (disjunctively) add a constraint about the class for the single label ℓ to model this arg-max function[3].

$$C_{out}^{(k+1)}(c) \quad \equiv \quad \Big(\bigwedge_{c' \neq c}(in_c^{(k)} \geq in_{c'}^{(k)}) \wedge class_\ell = c \Big)$$

Here c, c' are the classes of Z_1. In case of a multilabel classifier, an additional *threshold* value th is learned and the constraint for label ℓ is

$$\begin{aligned} C_{out}^{(k+1)}(\ell) \quad \equiv \quad & \bigwedge_{\ell=1}^{n_{k+1}}(in_\ell^{(k)} \geq th \wedge class_\ell = 1) \\ & \vee (in_\ell^{(k)} < th \wedge class_\ell = 0) \end{aligned}$$

The generated formulae employ real numbers and multiplication operations. This often impairs the performance of the SMT solver. For better scalability, we employ some form of quantization: we parametrize training as to obtain weights and biases in the interval [-10,10] only and with 3 decimal places.

B. Property Encoding

On the encoding of the white-box model, we *verify* the specified property. Our properties take the form

$$assume \Rightarrow assert$$

[2]We assume that the decision tree makes deterministic predictions, i.e. only one leaf node is chosen.

[3]For simplicity, the translation given here ignores ties.

i.e., if the assume condition holds on the inputs, the outputs should satisfy the assert condition. For verification we basically generate a logical formula $assume \wedge \neg assert$ and check the satisfiability of its conjunction with the white-box model encoding. If the conjunction is satisfiable, its logical model is a counterexample to the property (for the white-box model).

Connecting white-box model and property. First, we need to generate one copy of the white-box model formula for every data instance x occurring as parameter to `predict` in the property. We use a simple numbering scheme on variables to distinguish these copies. Second, every copy needs to be connected to the parameter x of `predict`. In the decision tree encoding, this means that we replace every feature name occurring in a condition on an edge by its appropriately numbered version. In the neural network, we add a constraint equating the feature values of parameter x with the output of layer 0 (fixing $out^{(0)}$), again using the appropriate version.

Translating property. For the property itself, we *execute* the Python code containing assume and assert statements. Every execution of `Assume` and `Assert` generates one logical formula. The conjunction of all these formulae presents the encoding of the property.

V. Evaluation

We have implemented this technique in a tool called MLCheck. and evaluated our approach within the already mentioned three application areas. In the evaluation, we were interested in the following three research questions:

RQ1 How effective is MLCheck in constructing test cases violating properties compared to existing approaches?

RQ2 How efficient is MLCheck in constructing test cases violating properties compared to existing approaches?

RQ3 How do the white-box models compare to each other?

A. Setup

Evaluation requires to have (1) models under test (obtained by training on some data sets), (2) properties to be checked (already given) and (3) tools to compare MLCheck to.

Datasets. We use different data sets to construct MUTs in the different application areas. Some statistics about the data sets can be found in Table II. For the *fairness experiments*, we use the Adult and German credit datasets from the UCI machine learning repository.[4] We used "gender" as sensitive feature for checking individual discrimination. For testing *concept relationships* (dataset CR), we employ Pyke embeddings [7] to map entities from the DBpedia knowledge graph (version 3.6)[5] to real vectors in 50 dimensions. We used 6 fragments of DBpedia, which each contain embeddings from 3 classes (our labels). We test for *disjointness* with three of the datasets, which each contain instances of 2 classes known to be disjoint (e.g., persons and places). The other three datasets are used analogously for subsumption (e.g., persons and actors). For testing on *trojan attacks*, we employ MNIST[6] dataset which is also used by Baluta et al. [15] for quantitatively verifying NNs wrt. trojan attacks. To obtain models which are vulnerable to trojan attacks, we further extended this training set with additional "poisened" data instances[7], i.e., instances in which some trigger $\mathbf{t}$ is present and the specific target prediction $\mathbf{z}$ is given.

TABLE II: Data sets and their characteristics

Name	#Features	#Instances	#NoClasses
Adult	13	32,561	2
German credit	22	1000	2
CR	50	450	3
MNIST	100	60,000	10

ML algorithms. Out of the training sets, we generate ML models using `scikit-learn` and PyTorch, the latter for all NNs as it provides more sophisticated configuration options for training NNs. For fairness testing, we train a random forest, a logistic regression classifier, a naive Bayes classifier and a decision tree. Moreover, we employ two *fair-aware* classifiers [1], [16], i.e., classifiers which are supposed to generate non-discriminating models. For concept relationship testing, we train an NN and a random forest. Finally, for trojan attacks we train an NN since this is the main classifier used on images. We use two different architectures for NN: one with 1 hidden layer of 100 neurons (called NN1 in Tables VI and VII) and one with 2 hidden layers with 64 neurons (NN2).

Baselines. For fairness testing, there are specialized tools for testing for individual discrimination. We compared our tools with the Symbolic Generation (SG) algorithm of Aggarwal et al. [17][8] and with AEQUITAS [19]. We do not consider THEMIS [4] for our comparison as this has already been shown to be less effective in comparison to SG and AEQUITAS as stated by Zhang et al. [18].

For concept relationships and trojan attacks, there are no specialized testing tools available. Here, we used the Python implementation (Hypothesis) of the property-based testing tool QuickCheck as the baseline approach to compare against.

Note that the *ground truth* about the models under test is unknown in all the experiments, i.e., we do not a priori know whether the trained classifiers do or do not satisfy the property. Further details of the setup can be found in [20].

B. Results

For **RQ1**, we compared the effectiveness of the tools by generating test inputs violating the property under interest. We report on the results separately for every application area. Due to the randomness in ML algorithms, we ran every experiment 20 times. Whenever we generated multiple counterexamples (i.e., for fairness), we give the mean over the 20 runs as well as

[4]https://archive.ics.uci.edu/ml

[5]http://dbpedia.org

[6]http://yann.lecun.com/exdb/mnist/

[7]Another option to obtain a "trojaned" model is to employ specific trojaning algorithms which however requires manipulating the model itself, i.e., requires a white-box model.

[8]We got the implementation of SG from the authors of [18].

TABLE III: Mean (± SEM) for Adult dataset

Classifiers	MLC_DT	MLC_NN	SG	AEQUITAS
Logistic Regress.	**102.30** (±16.36)	65.21 (±7.78)	30.20 (±3.27)	90.80 (±31.46)
Decision Tree	214.00 (±20.16)	64.30 (±1.36)	**225.48** (±4.23)	112.00 (±25.14)
Naive Bayes	38.40 (±5.53)	**69.60** (±3.93)	23.83 (±1.68)	0.00 (±0.00)
Random Forest	**166.14** (±22.12)	50.60 (±2.47)	19.82 (±5.59)	158.00 (±4.35)
Fair-Aware1	0.00 (±0.00)	**5.70** (±1.38)	-	-
Fair-Aware2	**80.91** (±2.67)	1.25 (±0.76)	3.87 (±0.56)	0.89 (±0.50)

TABLE IV: Mean (± SEM) for Credit dataset

Classifiers	MLC_DT	MLC_NN	SG	AEQUITAS
Logistic Regress.	**144.71** (±13.62)	78.60 (±7.97)	63.43 (±2.27)	63.00 (±8.65)
Decision Tree	**396.17** (±28.16)	17.75 (±1.36)	239.25 (±4.71)	18.72 (±8.98)
Naive Bayes	3.00 (±1.03)	**39.40** (±8.76)	3.00 (±0.00)	0.00 (±0.00)
Random Forest	154.57 (±22.12)	69.43 (±5.91)	**251.42** (±9.74)	10.20 (±9.12)
Fair-Aware1	0.00 (±0.00)	**19.89** (±1.38)	-	-
Fair-Aware2	**120.87** (±7.98)	0.00 (±0.00)	2.54 (±0.56)	1.78 (±0.50)

the standard error of the mean[9]. In the cases of a single counterexample (i.e., for concept relationships and trojan attacks) the probability of finding a counterexample across the 20 runs is given. Tables III and IV show the measures for the number of detected *unfair* test cases (i.e., test input pairs) for Adult and Credit dataset, respectively. The classifiers used for training the MUT are given in the first column (Fair-Aware1 and Fair-Aware2 are the algorithms of [1] and [16]). The next columns give the numbers for MLCHECK (MLC_DT with decision tree and MLC_NN with NN as white-box) as well as SG and AEQUITAS. The largest number is shown in bold. An entry 0.00 stands for no counterexamples found, the entry - (for Fair-Aware1) describes the fact that SG and AEQUITAS could not work on the MUT generated by this algorithm because of the format of the model. We see that MLCHECK always generates the largest number of counterexamples except for a single one (Random Forest with Adult dataset).

Next, Table V shows the result of testing concept relationships. We generated test cases for three properties (called S1, D1 and D2), one subsumption and two disjointness relationships. The rows show the results in probabilities per dataset (summarizing datasets with equal results) and model type (NN or random forest RF). We see that MLCHECK (in either DT or NN version) is able to find more or an equal number of falsifying test cases compared to property-based testing.

Tables VI and VII show the results of our experiments for trojan attacks. The tables again depict the probabilities with which the testing tool was or was not able to find a test input falsifying the property under interest. We considered two architectures for NNs models (NN1 and NN2), trained on the MNIST data set enhanced by 1,000 and 10,000 additional "poisened" instances, respectively for the two tables, using 4 different trigger features T1 to T4 and 2 different target predictions (classes 4 and 5). The triggers are hence named T1-4, T1-5 and so on.

It turned out that the property-based testing tool which we employed is not able to generate test cases.[10] We suspect that the reason for this failure is the high number of features (100)

[9]The Standard Error of the Mean (SEM) is obtained by dividing the standard deviation with the total number of samples which in our case is number of times we run our tool.

[10]On all instances, it stopped with the error message "hypothesis.errors.Unsatisfiable: Unable to satisfy assumptions of hypothesis", typically after trying to generate test inputs for around 40 minutes.

TABLE V: Probability of detected violations of subsumption/disjointness

Dataset	MLC_DT S1/D1/D2	MLC_NN S1/D1/D2	PBT S1/D1/D2
CR1 (NN)	**1.00**/0.00/**0.80**	**0.25**/0.00/**1.00**	0.00/0.00/0.00
CR1 (RF)	0.00/0.00/0.00	0.00/0.00/0.00	0.00/0.00/0.00
CR2-5 (NN)	**1.00/1.00/1.00**	**1.00/1.00/1.00**	**1.00/1.00/1.00**
CR2-5 (RF)	0.00/0.00/0.00	0.00/0.00/0.00	0.00/0.00/0.00
CR6 (NN)	**0.95/1.00/1.00**	**1.00/1.00/1.00**	**1.00/1.00/1.00**
CR6 (RF)	0.00/0.00/0.00	0.00/0.00/0.20	0.00/0.00/0.00

TABLE VI: Probability of detected violations of trojan attacks (data set with 1,000 poisened instances)

Trigger	MLC_DT NN1/NN2	MLC_NN NN1/NN2	PBT NN1/NN2	ART NN1/NN2
T1-4	0.00/0.00	**1.00/1.00**	err/err	**1.00/1.00**
T1-5	**0.10**/0.00	**1.00/1.00**	err/err	**1.00/1.00**
T2-4	**0.05**/0.00	**1.00/1.00**	err/err	0.00/**0.10**
T2-5	**0.20**/0.00	**1.00/1.00**	err/err	0.00/**0.25**
T3-4	0.00/0.00	**1.00/1.00**	err/err	**1.00/1.00**
T3-5	**0.20**/0.00	**1.00/1.00**	err/err	**1.00**/0.00
T4-4	0.00/0.00	**1.00/1.00**	err/err	**1.00/0.80**
T4-5	0.00/0.00	**1.00/1.00**	err/err	**1.00/0.50**

in this data set, i.e., the fact that Hypothesis has to generate inputs for a function with 100 arguments.

In order to be able to compare our technique to other methods, we hence decided to develop a prototype tool for adaptive random testing [21] with respect to trojan attacks. Tables VI and VII therefore also give the result for our prototype adaptive random tester (ART). Interestingly, ART is able to find counterexamples for a number of models trained on the data set enhanced with 1,000 poisened instances, even more often than MLCHECK with a decision tree. However, in the harder cases with models trained on the data set enhanced with 10,000 instances, ART also produces no test inputs at all.

In summary, MLCHECK outperforms other tools in almost all cases, even when they are specialised on the property to be tested.

For **RQ2**, we compared the efficiency (in terms of runtime) of the tools in generating test inputs violating the property under interest. Again, the given values are averaged over 20 runs.

Figures 3 (fairness), 4 (concept relationships) and 5 (trojan attacks) depict the runtimes of MLCHECK versus SG and Aequitas, property-based testing (PBT) and adaptive random testing (ART), respectively. The x-axis depicts the number of tasks solved, ordered by runtime per tool from fastest to slowest, the y-axis is the runtime in seconds[11].

Except for trojan attacks we see that the increased effectiveness of MLCHECK does not come at the prize of a much higher runtime.

[11]SG and AEQUITAS curves end at 10 tasks as they do not run on models generated by one of the fair-aware algorithms.

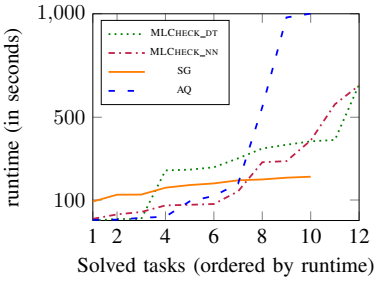

Fig. 3: Fairness

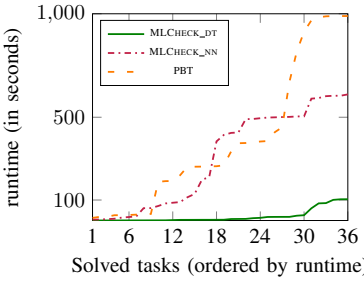

Fig. 4: Concept relationships

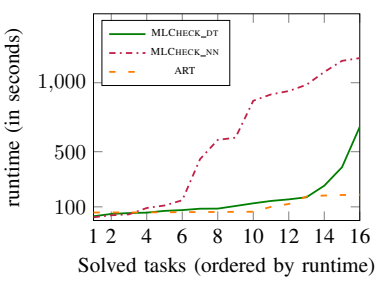

Fig. 5: Trojan attacks

TABLE VII: Probability of detected violations of trojan attacks (data set with 10,000 poisened instances).
All triggers yield the same result.

Trigger	MLC_DT NN1/NN2	MLC_NN NN1/NN2	PBT NN1/NN2	ART NN1/NN2
Tn-m	0.00/0.00	**1.00/1.00**	err/err	0.00/0.00

For **RQ3**, we take another look at the tables of detected violations and figures of runtimes, now comparing the decision tree and NN version of MLCHECK.

With respect to the number of detected violations, we see that the NN as white-box model is–with a few exceptions–only able to outperform the decision tree in case of the MUT being an NN itself. The better performance in these cases does often not come at the prize of a (much) increased runtime. In such cases the NN white-box model can even outperform the decision tree, which can be seen in the trojan attack setting trained with 10,000 additional poisoned instances. For the trojan attacks with a high number of features, the NN white-box is much better in approximating the MUT. This confirmed our initial expectation that it does in fact make sense to employ two different white-box models with different generalization abilities in test case generation. Note that all three application areas contain *hard* benchmarks characterised by only few counterexamples, either generated by using specific ML algorithms (fair-aware algorithms) or by assembling specific data sets, obtained by embeddings (for concept relationships) or by flooding the training set with property-satisfying instances (trojan attacks). The results of our experiments on these benchmarks can be summarized as follows.

MLCHECK outperforms other tools on hard testing tasks.

VI. DISCUSSION

We briefly discuss some further aspects of our approach. First, our approach is *sound* in the sense of only generating test inputs which are counterexamples to the property *on the black-box model*. We might generate candidate counterexamples which are only valid counterexamples on the approximating white-box model, but such counterexamples do not get into the test suite. All candidate counterexamples are checked on the model under test. Furthermore, our approach supports adding *constraints* on feature values to the test input generation process. Such constraints can for instance reflect the "data semantics" (e.g., 'age' having to be less than 120).

Pertaining to *scalability*, we do not claim that our approach will in general be applicable to image classifiers, although image recognition is one of our case studies. Images with several thousands of features pose difficulties for the SMT solver. However, the number of hidden layers of the model is *not* a limiting factor as we employ a much simpler NN for approximation, and translate this NN into logical formulae.

VII. RELATED WORK

We briefly discuss other approaches to the validation of machine learning models. The most frequently studied type of models are deep neural networks (DNNs). Most often, DNNs are tested for *adversarial robustness*, i.e., their vulnerability of a DNN to adversarial attacks. Attack methods that aim at generating adversarial examples can be classified into white-box (e.g., [10], [22]) and black-box (e.g., [23], [24]) approaches. Recent works have also proposed methods for computing probabilistic guarantees on robustness [25].

Testing methods aim at generating counterexamples to properties. *Formal verification* on the other hand aims at correctness guarantees. For NNs a number of verification techniques have been developed, based on abstract interpretation [26]–[28], by layer-wise computations of safety constraints [29] or by a combination of SAT solving and linear programming [30].

More recently, Pham et al. [31] have proposed a framework to verify different types of NN models using *optimization-based falsification* and *statistical model checking* methods. Alike us, they also provide a specification language to specify several properties (for e.g. fairness, robustness etc.) to be checked on the model.

Baluta et al. [15] propose *quantitative* verification for NNs, i.e., verification which gives a quantitative account on the number of inputs violating some property. Similar to the encoding of our NN white-box models, they translate NNs to logical formulae on which approximate model counting can then provide estimates about the number of satisfying logical models. To make their approach scale, they apply it to binarized NNs only, and perform further quantization. Other verification approaches using logical encodings of DNNs together with SAT, SMT or MIP solvers for property checking have been proposed by Narodytska et al. [32] (studying various properties, in particular also adversarial robustness), Pulina et

al. [33] or Cheng et al. [34]. Katz et al. [35] in addition build a specific SMT solver for solving linear real arithmetic constraints arising from DNNs with ReLU activation functions.

A survey on testing techniques for ML models in general, i.e., not restricted to NNs, has recently been assembled by Zhang et al. [36]. Model-agnostic but property-specific approaches most often target *fairness testing*, more precisely testing for individual discrimination. Themis [4], [37] is an automated test case generation technique that uses random testing complemented by three optimization procedures. It allows for checking two types of fairness definitions, namely *causal* (i.e. individual) discrimination and group discrimination for a given black-box model. Aggarwal et al. [17] propose a symbolic approach (SG) to generate test cases for checking individual discrimination. They use LIME to generate a path of a decision tree from the MUT, on which they use dynamic symbolic execution to generate test cases. They show that their approach outperforms Themis and AEQUITAS [19]. In contrast, we approximate the entire MUT by a white-box model (either a decision tree or an NN) and then compute the test cases on this model. The idea of using a white-box model to approximate a given ML model is already discussed in the context of *explainable AI* (for a survey of such works see [38]). Moreover, our approach can also be used for checking other types of fairness (e.g. *fairness through awareness* [6]) as well as completely different properties.

VIII. Conclusion

We have proposed an approach for property-driven testing of machine learning models. The approach encompasses a language for property specification and a method for targeted generation of test cases falsifying the property. As future work, we intend to study the applicability of the more advanced verification techniques on DNNs (e.g. [31]) for generating counterexamples for our white-box model NN.

Acknowledgement

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

References

[1] M. B. Zafar, I. Valera, M. Gomez-Rodriguez, and K. P. Gummadi, "Fairness constraints: Mechanisms for fair classification," in *AISTATS*, 2017, pp. 962–970.

[2] R. Potharst and A. J. Feelders, "Classification trees for problems with monotonicity constraints," *SIGKDD Explor. Newsl.*, vol. 4, no. 1, 2002.

[3] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial machine learning at scale," in *ICLR*. OpenReview.net, 2017.

[4] S. Galhotra, Y. Brun, and A. Meliou, "Fairness testing: testing software for discrimination," in *ESEC/SIGSOFT FSE*. ACM, 2017, pp. 498–510.

[5] A. Sharma and H. Wehrheim, "Higher income, larger loan? Monotonicity testing of machine learning models," in *ISSTA*. ACM, 2020.

[6] S. Verma and J. Rubin, "Fairness definitions explained," in *International Workshop on Software Fairness, FairWare@ICSE*, 2018, pp. 1–7.

[7] C. Demir and A. Ngonga Ngomo, "A physical embedding model for knowledge graphs," in *JIST*, ser. LNCS 12032, 2019.

[8] A. Geigel, "Neural network trojan," *J. Comput. Secur.*, vol. 21, no. 2, pp. 191–232, 2013.

[9] Y. Liu, S. Ma, Y. Aafer, W. Lee, J. Zhai, W. Wang, and X. Zhang, "Trojaning attack on neural networks," in *NDSS*, 2018.

[10] S. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: A simple and accurate method to fool deep neural networks," in *CVPR*. IEEE Computer Society, 2016, pp. 2574–2582.

[11] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, no. 6, pp. 1157–1210, 2010.

[12] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," in *(ICFP '00)*, 2000, pp. 268–279.

[13] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, 2008, pp. 337–340.

[14] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. V. Nori, and A. Criminisi, "Measuring neural net robustness with constraints," in *NIPS*, 2016, pp. 2613–2621.

[15] T. Baluta, S. Shen, S. Shinde, K. S. Meel, and P. Saxena, "Quantitative verification of neural networks and its security applications," in *CCS*. ACM, 2019, pp. 1249–1264.

[16] T. Calders, F. Kamiran, and M. Pechenizkiy, "Building classifiers with independency constraints," in *ICDM Workshops*, 2009, pp. 13–18.

[17] A. Aggarwal, P. Lohia, S. Nagar, K. Dey, and D. Saha, "Black box fairness testing of machine learning models," in *ESEC/SIGSOFT FSE*, 2019, pp. 625–635.

[18] P. Zhang, J. Wang, J. Sun, G. Dong, X. Wang, X. Wang, J. S. Dong, and T. Dai, "White-box fairness testing through adversarial sampling," in *ICSE*. ACM, 2020, pp. 949–960.

[19] S. Udeshi, P. Arora, and S. Chattopadhyay, "Automated directed fairness testing," in *ASE 2018*. ACM, pp. 98–108.

[20] A. Sharma, C. Demir, A. N. Ngomo, and H. Wehrheim, "Mlcheck-property-driven testing of machine learning models," *CoRR*, vol. abs/2105.00741.

[21] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *ASIAN*, 2004, pp. 320–329.

[22] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *ICLR*, 2015.

[23] F. Zhang, S. P. Chowdhury, and M. Christakis, "DeepSearch: A Simple and Effective Blackbox Attack for Deep Neural Networks," in *ESEC/FSE*, 2020.

[24] N. Papernot, P. D. McDaniel, I. J. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *AsiaCCS*. ACM, 2017, pp. 506–519.

[25] L. Cardelli, M. Kwiatkowska, L. Laurenti, and A. Patane, "Robustness guarantees for bayesian inference with gaussian processes," in *AAAI*. AAAI Press, 2019, pp. 7759–7768.

[26] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. T. Vechev, "AI2: safety and robustness certification of neural networks with abstract interpretation," in *IEEE Symposium on Security and Privacy, SP*, 2018, pp. 3–18.

[27] G. Singh, T. Gehr, M. Püschel, and M. T. Vechev, "An abstract domain for certifying neural networks," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 41:1–41:30, 2019.

[28] Y. Y. Elboher, J. Gottschlich, and G. Katz, "An abstraction-based framework for neural network verification," in *CAV*, ser. LNCS 12224, 2020.

[29] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety verification of deep neural networks," in *CAV*, 2017, pp. 3–29.

[30] R. Ehlers, "Formal verification of piece-wise linear feed-forward neural networks," in *ATVA*, ser. LNCS, vol. 10482, 2017.

[31] L. H. Pham, J. Li, and J. Sun, "SOCRATES: towards a unified platform for neural network verification," *CoRR*, vol. abs/2007.11206, 2020.

[32] N. Narodytska, S. P. Kasiviswanathan, L. Ryzhyk, M. Sagiv, and T. Walsh, "Verifying properties of binarized deep neural networks," in *(AAAI-18)*. AAAI Press, 2018, pp. 6615–6624.

[33] L. Pulina and A. Tacchella, "An abstraction-refinement approach to verification of artificial neural networks," in *CAV*, ser. LNCS 6174, 2010.

[34] C. Cheng, G. Nührenberg, and H. Ruess, "Maximum resilience of artificial neural networks," in *ATVA*, ser. LNCS 10482, 2017.

[35] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," in *CAV*, ser. LNCS 10426, 2017.

[36] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *CoRR*, vol. abs/1906.10742, 2019.

[37] R. Angell, B. Johnson, Y. Brun, and A. Meliou, "Themis: automatically testing software for discrimination," in *ESEC/FSE*. ACM, 2018.

[38] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi, "A survey of methods for explaining black box models," *ACM Comput. Surv.*, 2019.