Automatic Fairness Testing of Machine Learning Models

Arnab Sharma, Heike Wehrheim

Automatic Fairness Testing of Machine Learning Models

Arnab Sharma, Heike Wehrheim

Paderborn University, Paderborn, Germany
{arnab.sharma,wehrheim}@uni-paderborn.de

Abstract. In recent years, there has been an increased application of machine learning (ML) to decision making systems. This has prompted an urgent need for validating requirements on ML models. *Fairness* is one such requirement to be ensured in numerous application domains. It specifies a software as "learned" by an ML algorithm to not be biased in the sense of discriminating against some attributes (like gender or age), giving different decisions upon flipping the values of these attributes.
In this work, we apply *verification-based testing* (VBT) to the fairness checking of ML models. Verification-based testing employs verification technology to generate test cases potentially violating the property under interest. For fairness testing, we additionally provide a specification language for the formalization of different fairness requirements. From the ML model under test and fairness specification VBT automatically generates test inputs specific to the specified fairness requirement. The empirical evaluation on several benchmark ML models shows verification-based testing to perform better than existing fairness testing techniques with respect to effectiveness.

Keywords: Fairness · Machine learning testing · SMT solving.

1 Introduction

Machine Learning (ML) is gradually being used in software systems and replacing humans in decision making. The application area of such software systems now ranges from social and economical domains to even law [12]. Therefore, the quality assurance of these systems is of utmost importance. In the past few years, a significant amount of research works have been performed for checking various sorts of requirements arising in different application domains (e.g., robustness [10], security [4], balancedness [18]).

One such requirement which often needs to be ensured by the "learned" software system is *fairness*. Although there exists multiple definitions of fairness in the literature [21], the basic idea of the property is always the same. Fairness requires that changing the values of only the *protected* attributes should not change the *prediction* of an ML classifier. For example, a loan granting software which decides whether a person gets a loan, is discriminating against "gender", if it gives a different decision (i.e., prediction) for male and female applicants when

all other values of features besides "gender" are equal. Here, gender is considered to be the *protected* attribute.

Most often the use of biased training data results in such unfair predictors. However, even if the training data does not contain any unfairness, the generated ML model can still be biased. Hence, today there exists a number of specialized algorithms aiming at the generation of fair ML models (e.g. [3], [22]). Galhotra et al. [8] have nevertheless shown that even the use of these fair algorithms cannot guarantee fair predictive models. They have proposed a random testing technique called THEMIS which works by generating a number of test cases testifying if and how much unfairness exists in the ML model. They have furthermore proposed *individual discrimination* as the definition of fairness, as their work suggests that the existing fairness definitions can hide unfairness in an ML model. Later, Udeshi et al. have proposed AEQUITAS [20] which generates a higher number of test cases in comparison to THEMIS in checking *individual discrimination*. More recently, Aggarwal et al. [1] have proposed a technique combining dynamic symbolic execution and local explanation which outperforms the previous two techniques in checking fairness.

Although these approaches perform quite well in detecting unfairness of a given ML model, they mostly focus on testifying one specific type of fairness. Hence, these approaches cannot be directly used to check a model for any fairness definition. As there exists several such fairness definitions depending on the application domain, a unified approach to test an ML model for a given fairness property is required.

In an earlier work, we have introduced a novel black-box testing approach [17] to test the monotonicity of a given ML model. This approach works by systematically exploring the input space of the given model by firstly inferring a white-box model *approximating* the black-box model under test (MUT), and computing the counter examples to monotonicity on the white-box model via an established verification technique. The computed counter examples on the white-box model are then checked with the black-box model. The confirmed ones are stored as test cases violating the property and further varied to generate more test cases. If unconfirmed, they serve as input to an improvement of the approximation quality of the white-box model. Here, we extend this approach to test the fairness of a given machine learning model. Our idea is to develop a property-based testing mechanism for fairness checking where the specific fairness requirement can be specified using an `assume-assert` construct. Test cases are then automatically generated attempting to violate the specified fairness property. The underlying mechanism remains the same as before and comprises four key steps: 1) White-box model inference, 2) fairness computation, 3) variation and 4) white-box model improvement.

We have implemented our approach and have experimentally evaluated it by applying it on standard benchmark ML models. Our experimental results suggest that our *verification-based testing* technique performs better than existing fairness testing techniques ([20] and [1]) for most of the test cases. Summarizing, this paper makes the following contributions:

- We provide a specification language for formulating fairness requirements on ML Models.
- We employ our verification-based-testing technique to test fairness of ML models.
- We systematically evaluate our approach on several standard benchmarks and compare our results with existing fairness testing approaches.

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

2 Fairness

We begin with the basic terminologies in machine learning and then give definitions of fairness.

Our interest is in testing models obtained by *supervised* machine learning. Such algorithms typically have two steps. Initially, in the *learning* phase, the algorithm is presented with a set of data instances called *training data*. The ML algorithm then generates a function (the *predictive model*), generalising from the *training data* by using some statistical techniques. The generated *predictive model* (short, model) is used in the second (prediction) phase to predict classes for unknown data instances.

Formally, the generated model (M) can be defined as 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 $\vec{\mathbf{X}} = X_1 \times \ldots \times X_n$. The training data consists of elements from $\vec{\mathbf{X}} \times Y$, 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 $(x_1, \ldots, x_n) \in \vec{\mathbf{X}}$.

In the literature, a large number of fairness definitions for ML models can be found (see [21] for a survey). They basically all build on the same concept of fairness: an ML model (or algorithm in general) is unfair if it is discriminating some individuals (i.e., data instances) on the basis of values of some of their features. This discrimination is however formally captured in completely different ways: while some works (e.g. [7], [5]) employ statistical measures (probability distributions on certain outcomes of M), others employ similarity-based measures (e.g. [8], [1]). Our objective here is to develop an approach for testing *black-box* models, i.e., what our approach can observe about a model are just the outputs (predictions) for certain inputs (data instances). We thus focus on similarity-based measures.

Fairness definitions first of all fix so-called protected or *sensitive* attributes which are supposed to not lead to discrimination. As an example, consider Figure 1 which gives a decision tree for loan granting based on attributes “gender”,

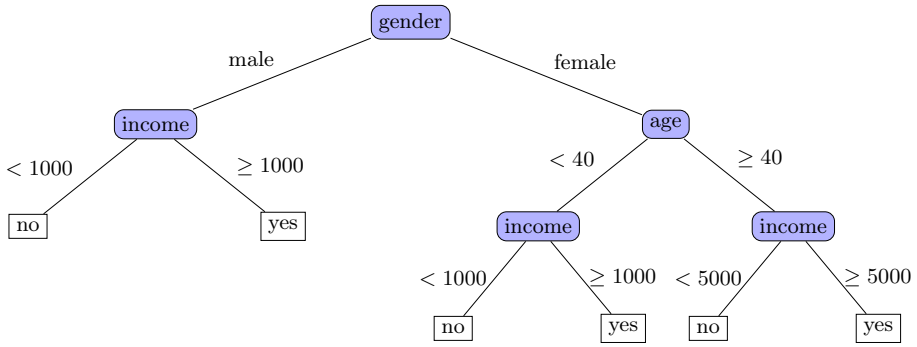

Fig. 1. A decision tree for predicting who gets a loan

"income" and "age". A fairness requirement on loan granting software might for instance state that the software should not discriminate against "gender", hence "gender" would be the sensitive attribute.

Definition 1. *A predictive model M is* fair *with respect to a sensitive feature i if for any two data instances $x = (x_1, \ldots, x_n), x' = (x'_1, \ldots, x'_n) \in \vec{\mathbf{X}}$, we have $(x_i \neq x'_i) \wedge (\forall j, j \neq i . x_j = x'_j)$ implies $M(x) = M(x')$.*

This fairness definition is termed *individual discrimination* and was introduced by Galhotra et al. [8]. According to this definition, all the feature values except for the protected one should have the same value. The idea is to find out whether changing only the protected feature value leads to the change of the prediction. Our decision tree model in Figure 1 is not fair wrt. Definition 1 and sensitive feature "gender". Take for instance the following pair of data instances

income=1000, age=40, gender=female
income=1000, age=40, gender=male

For the first instance, the result of the prediction is 'no', while for the second data instance it will be 'yes'.

We next define *group discrimination* which extends Definition 1 to a set of protected features.

Definition 2. *A predictive model M is said to be* fair *with respect to a set of sensitive features $F = \{i_1, i_2, \ldots, i_m\} \subseteq \{1, \ldots, n\}$ if for any two data instances $x = (x_1, \ldots, x_n), x' = (x'_1, \ldots, x'_n) \in \vec{\mathbf{X}}$ we have $(\forall j \in F : x_j \neq x'_j \wedge \forall j \notin F : x_j = x'_j)$ implies $M(x) = M(x')$.*

In case of the tree model depicted in Figure 1, it is evident that the decision tree is discriminating those female candidates who have age more than 40. The applicants with age less than 40 are treated equally regardless of their gender. But those who are above 40 should have a higher income than the rest.

Another similarity-based measure is *fairness through unawareness*. This requires that the protected attributes have no influence on the classification at all. Next, we directly define the group version.

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

Whenever two data instances coincide on all features values except possibly for the protected attributes, they should get the same prediction.

Finally, there is also a similarity-based measure called *fairness through awareness*. This relaxes the strict equality of (some) feature values required by the definitions so far to *similar* values. The similarity is therein captured by some distance metric $d : \vec{\mathbf{X}} \times \vec{\mathbf{X}} \to [0, 1]$ on data instances[1].

Definition 4. *Let $d : \vec{\mathbf{X}} \times \vec{\mathbf{X}} \to [0, 1]$ be a distance metric on data instances and ε a threshold. A predictive model M is said to be* fair *with respect to a set of sensitive features $F = \{i_1, i_2, \ldots, i_m\} \subseteq \{1, \ldots, n\}$ if for any two data instances $x = (x_1, \ldots, x_n), x' = (x'_1, \ldots, x'_n) \in \vec{\mathbf{X}}$ we have $d(x, x') \leq \varepsilon$ implies $M(x) = M(x')$.*

For our loan-granting example, we could for instance define the distance metric as

$$d(x, x') = \begin{cases} 1 & \text{if } x_{gender} = x'_{gender} \\ \frac{|x_{age} - x'_{age}|}{100} & \text{else} \end{cases}$$

and let $\varepsilon = 0.1$. This would require that all pairs of data instances with different gender and difference in age of less than or equal 10 get the same prediction.

Next, we describe an approach for testing arbitrary given ML models with respect to some given fairness definition.

3 Testing approach

In this section, we describe the language for specifying fairness and also briefly discuss our *verification-based-testing* (VBT) approach which we employ for testing fairness. Note that fairness testing is an instance of metamorphic testing [16] in that we cannot check violation of the tested property on a single test input, but just on *pairs* of test inputs.

3.1 Fairness specification

Usually, approaches for fairness testing employ a fixed fairness definition and allows users to specify the set of protected features only. Our testing approach in addition allows the software engineer to state the fairness requirement itself. Our language is inspired by *property-based testing* approaches [6]. The user can specify *assumptions* on the inputs, in our case pairs of data instances, and *assertions*

[1] Note that the definition given here differs from that of [21] as we again do not consider probability distributions of outcomes here.

```
# Train the ML algorithm with Loan data containing 3 features: income, age, gender
df = pd.read_csv('LoanData.csv')
data = df.values
x_train = data[:, :-1]
y_class = data[:, -1]
model = LogisticRegression() # Using Logistic regression classifier to train
model = model.fit(x_train, y_class)
# Setting the parameters of fairCheck
fc = fairCheck(no_of_instance = 2, XML_file = 'input.xml', model = model,
instance_list = (x,y))
# Assumptions: index '0' is income, '1' is age and '2' is gender
for i in range(0, 3):
  if(i == 2):
    assume('x[i] != y[i]', i)
  else:
    assume('x[i] = y[i]', i)
# Assertion
Assert('model.predict(x) = model.predict(y)')
```

Fig. 2. Python code snippet corresponding to Definition 1

on the outcome (the prediction). The testing technique then tries to generate inputs satisfying the assumptions and violating the assertions.

For the specification of assumptions, the user needs to know the attributes of data instances. These are fixed in a schema definition given in an `XML` configuration file (attributes and their types). Such schema files are often used to describe the format of training data in machine learning. In the assumptions, attributes of a data instance `x` can be accessed like arrays, i.e., `x[0]` is the value of the first feature of instance `x` and so on. Assumptions are then arbitrary boolean expressions over the attributes of two data instances, called `x` and `y` (instead of x and x'). The model itself can be referred to by the result of training a specific classifier on a given training data set. All parts of the fairness specification are written in Python.

As an example, consider the loan-granting setting of the previous section. The assumptions and assertion for individual discrimination (Def. 1) can be found in Figure 2. First, we train a classifier as to get the `model` (M). Before specifying the property, some input parameters need to be fixed for our approach to work. For example, the `no_of_instance` defines number of data instances required to specify the property which is in this case 2 and the `instance_list` contains the instance variables (in this case is `x` and `y`). Also, the `XML` schema file and ML model to be checked need to be specified. There are several other optional parameters with default values.

For example, the default value of parameter `no_of_cex` is set to 'single', indicating the generation of a single counter example if the assertion is violated. If the user assigns 'multi' to this parameter, then multiple counter examples

```
for i in range(0, 3):
  if(i == 2):
    assume('x[i] != y[i]', i)
  else:
    assume('0.01*(abs(x[i] - y[i])) <= 0.1', i)
# Assertion
Assert('model.predict(x) = model.predict(y)')
```

Fig. 3. Python code snippet corresponding to Definition 3

would be generated. After fixing the input parameters, we state assumptions and assertion. The testing approach will subsequently generate test inputs which fulfill the assumptions and violate the assertion (if possible).

Similarly, Figure 3 shows the code snippet for checking fairness with respect to Definition 3. Here we have elided the part on parameter setting, training etc. The only difference to the previous example is the specification of the distance function. The `assume` statement describes the distance metric we have defined earlier. The absolute value operator ($|..|$) is specified by `abs`. The expression to describe distance metric *dist* (i.e. the expression of `assume`) can be of the following form.

$$dist := d_{x,y} \bowtie \epsilon, \qquad d_{x,y} := k \oplus (x[i] \oplus y[i]) \mid k \oplus (|x[i] \oplus y[i]|)$$

where $\oplus \in \{+, -, *, /\}$, $\bowtie \in \{=, \leq, \geq, >, <, \neq\}$ and $\epsilon, k \in \mathbb{R}$. Currently, our approach can deal with arbitrary distance metrics, which can be defined by using standard arithmetic operators. Although, the distance metric containing multiplication or division operations might lead to undecidability of the satisfiability question. Hence, using those operators might cause a significant slow down of our approach.

3.2 Verification-based testing

For generating test inputs, we employ verification-based testing which we have introduced for testing monotonicity of ML models in [17]. The basic idea is to approximate the black-box model under test (MUT) by a white-box model, essentially a decision tree. The intention therein is to be able to apply verification techniques for property checking once we have a white-box model. Decision trees are a good candidate for such a white-box model since they are easily convertible into logical formulae so that we can use an SMT solver to verify properties on the tree.

The idea of using a white-box approximator for a given black-box model is not new and studied in the areas of *interpretable* AI and testing of non-ML software. In AI, Guidotti et al. [9] uses an explainable white-box model to explain an unknown black-box model. In the testing domain, Papadopoulos and Walkinshaw [14] use the inference of a predictive model from test sets to further

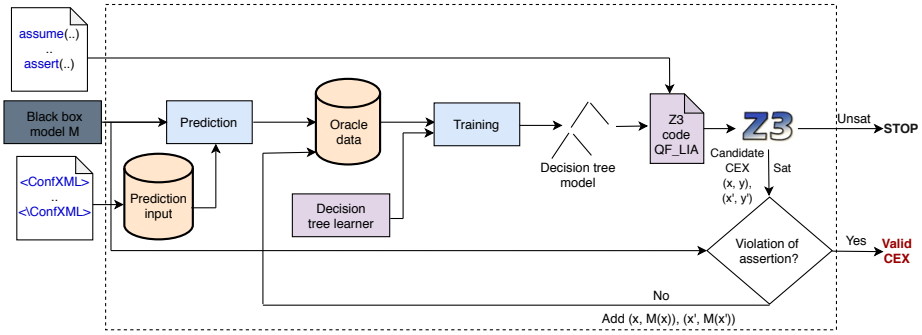

Fig. 4. Basic workflow of verification-based testing

extend this set. We use the inference of a black-box model to a white-box model to compute counter examples to specified properties.

Figure 4 depicts the basic workflow of our approach. The inputs to our approach are the predictive model M (MUT), the schema describing the set of features F and their types (e.g. `int, float`) used in the training of the model as an `XML` configuration file and an `assume-assert` fairness specification.

Our approach consists of four steps. Below, we describe the steps of our technique and explain the workflow in detail.

1. White-box model inference. First, we train a decision tree with a set of data instances called *oracle data*. We generate this oracle data (i.e., training data for the tree) by using the predictions of the MUT for some randomly chosen input instances (referred to as 'Prediction input' in Fig. 4). A decision tree learner is trained on the oracle data which results in a decision tree model approximating the given unknown black-box model.

2. Test generation. Once we have generated the decision tree, the next step is to verify the given fairness property and thereby generate test inputs. For this, we use the state of the art SMT solver Z3 [13]. First, we translate the decision tree into a logical formula describing how the classes are predicted for inputs x and x'. Figure 5 shows the Z3 code for the decision tree in Figure 1. The data instance x (i.e. test input) is described by the features `gender1`, `age1` and `income1` whereas, test input x' is described by `gender2`, `age2` and `income2`.

The Z3 code of the tree is then complemented with code for the fairness specification. The assumptions of the fairness requirement become `assert` statements in Z3, the assertion becomes an `assert` statement in its *negated* form. The resulting formula is then checked for satisfiability, i.e., we check whether the decision tree allows for a prediction satisfying the assumptions in the fairness constraint, but not the assertion. If the formula is satisfiable, the SMT solver returns a satisfying assignment which is a counter example to this fairness constraint (however, not necessarily for the black-box model, so far just for the decision tree).

```
; Declaring components of x and x' and their classes
(declare-fun gender1 () Int)  (declare-fun income1 () Real)
(declare-fun gender2 () Int)  (declare-fun income2 () Real)
(declare-fun age1() Int) (declare-fun age2() Int)
(declare-fun class1 () Int) (declare-fun class2 () Int)
; Specifying prediction of decision tree (no=0, yes=1)
(assert (=> (and (= gender1 0) (< income1 1000)) (= class1 0)))
(assert (=> (and (= gender1 0) (>= income1 1000)) (= class1 1)))
(assert (=> (and (= gender1 1) (< age1 40) (< income1 1000)) (= class1 0)))
(assert (=> (and (= gender1 1) (< age1 40) (>= income1 1000)) (= class1 1)))
(assert (=> (and (= gender1 1) (>= age1 40) (< income1 5000)) (= class1 0)))
(assert (=> (and (= gender1 1) (>= age1 40) (>= income1 5000)) (= class1 1)))
(assert (=> (and (= gender2 0) (< income2 1000)) (= class2 0)))
(assert (=> (and (= gender2 0) (>= income2 1000)) (= class2 1)))
(assert (=> (and (= gender2 1) (< age2 40) (< income2 1000)) (= class2 0)))
(assert (=> (and (= gender2 1) (< age2 40) (>= income2 1000)) (= class2 1)))
(assert (=> (and (= gender2 1) (>= age2 40) (< income2 5000)) (= class2 0)))
(assert (=> (and (= gender2 1) (>= age2 40) (>= income2 5000)) (= class2 1)))
; Unfairness constraint
(assert (and (not(= gender1 gender2)) (= income1 income2) (= age1 age2)))
(assert (not (= class1 class2 )))
;  Satisfiable ?
(check-sat)
;  Logical model extraction
(get-model)
```

Fig. 5. Z3 code of the decision tree with individual discrimination constraint

For example, consider the Python code for checking individual discrimination (Def. 1) described in Figure 2. The `assume` statements in Python code are translated to `(assert(and(not(= gender1 gender2))(= income1 income2) (= age1 age2)))` as Z3 code in Figure 5. The `assert` condition is then added as `(assert (not (= class1 class2)))` to Z3. The assertion specified is basically negated in the logical formula in an attempt to generate a counter example violating the specified property. The last two 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 6.

The counter example returned by the SMT solver is a pair of data instances and their respective classes $((x, y), (x', y'))$. The corresponding classes are essentially the prediction given by the approximating decision tree. As the tree is only an approximation, the counter example produced might not be valid for the MUT. Hence, we consider it as a *candidate* counter example (candidate CEX in Fig. 4). In the next step we check whether this is a valid counter example by using the prediction of the MUT. If yes, we add the counter example as a test input to the test suite. If not, $(x, M(x))$ and $(x', M(x'))$ are added to the or-

```
sat (model
(define-fun income1 () Real 1000.0)
(define-fun age1 () Int 40)
(define-fun gender1 () Int 0)
(define-fun class1 () Int 1)
(define-fun income2 () Real 1000.0)
(define-fun age2 () Int 40)
(define-fun gender2 () Real 1)
(define-fun class2 () Int 0))
```

Fig. 6. Logical model for the query of Fig. 5

acle data in order to increase precision of the approximation in later steps by re-training the decision tree.

When Z3 cannot find a counter example, hence giving 'Unsat' as output, the approximated decision tree is fair wrt. the given fairness constraint. Still, it might be the case that the MUT itself is not fair. Then our approach was unable to generate test inputs. In all our experiments this has however not occurred so far.

3. White-box model improvement. Once we have collected a larger set of test pairs (candidate counter examples, see below on how to get multiple counter examples), we examine the validity of those for the MUT M. If any of the candidate counter examples turns out to be also a valid one for the MUT, we have found a test case violating the specified fairness property. If we cannot find such a case, then the decision tree's prediction obviously differs from the MUT for this input. These cases will then be added to the oracle data to re-train the decision tree and thereby improve the approximation. With this new tree, test generation will be started once more.

4. Variation. The basic workflow depicted in Figure 4 is enhanced by one more step. As discussed in the previous step, if a counter example found by Z3 on computing fairness of the decision tree is invalid, the approximating tree needs to be made more precise. This can be easily achieved by adding the counter example to the oracle data and train the decision tree algorithm again. But training is a costly operation and hence it would be more efficient to train the tree only after gathering a sufficient number of invalid counter examples. To this end, we apply two strategies to produce several different logical models (i.e., counter examples) for the same logical query by using Z3 repetitively, namely *branch pruning* and *data instance pruning*. Basically, branch pruning inserts additional constraints into the logical formulae which tell the SMT solver not to generate counter examples following the same branches of the tree while data instance pruning tells Z3 to generate different data instances.

The detailed algorithms of our pruning techniques can be found in our earlier work [17]. We use these two pruning techniques to generate multiple counter examples. In the previous works of fairness testing, the effectiveness of a technique is measured in terms of the number of valid test cases generated, i.e., test cases

violating the fairness constraint. Hence, we run test generation until we reach a specified limit or no further test cases can be found.

4 Evaluation

We have implemented our technique in Python. The implementation of our approach is available online at https://github.com/arnabsharma91/fairCheck. While evaluating our approach we have focused on the following two research questions.

RQ1. How does verification based testing compare to existing fairness testing approaches?
RQ2. Which pruning strategy performs better in computing unfairness?

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

RQ1. We intend to compare how our approach performs in detecting unfairness compared to existing fairness testing techniques. Note however that there does not exist a fairness checking mechanism so far which can validate a black-box model with respect to a user given fairness specification. Hence, we compare our approach only to fairness testing techniques with fixed fairness definitions. For this, we have chosen AEQUITAS [20] and Symbolic Generation (SG) algorithm [1] which have both been designed to test a given black-box model for individual discrimination. We do not consider THEMIS [8] for our comparison as it has already been shown to be less effective than the other two approaches as stated by Zhang et al. [24]. The aforementioned techniques use the number of valid unfair cases generated within a specified limit of total generated test cases as their evaluation metric. The comparison between the three techniques is thus performed on this basis. We employ both data instance and branch pruning strategies for this experiment.

RQ2. For test generation, we have implemented branch and data instance pruning strategies to achieve better coverage of the decision tree. We intend to find out which strategy is better in finding unfair test cases. We use a different evaluation metric for this comparison. Instead of counting the test cases generated we compute detection rate when each of the strategies is applied individually. We have also used a similar approach in our earlier work [17] to compare our pruning strategies. The detection rate is defined as $\frac{\#\text{Valid test cases}}{\#\text{Total test cases}}$, i.e., we measure how many of the generated test cases (candidate counter examples) are really test cases violating the fairness definition.

4.1 Setup

We have performed the evaluation on two data sets, namely Adult and German credit data from the UCI machine learning repository[2]. The Adult dataset contains 13 features and 32561 data instances, the German credit dataset has

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

Table 1. Comparison of number of test cases generated for Adult dataset

Classifier	Prot. feature	VBT	SG	AEQUITAS
Logistic Regression	Gender	**133**	37	80
Logistic Regression	Race	**66**	58	25
Random Forest	Gender	**456**	90	29
Random Forest	Race	**358**	338	129
Naive Bayes	Gender	**75**	18	6
Naive Bayes	Race	**43**	40	30
Decision tree	Gender	**684**	94	156
Decision tree	Race	**739**	320	134

21 features and 1000 data instances. We have chosen four classification algorithms for our experiments: Naive Bayes (NB), Random Forests (RF), Logistic Regression and Decision tree. All these have been taken from `scikit-learn`. The choice of these algorithms and datasets is driven by the fact that they all have been used in the previous works of fairness testing. These ML algorithms are also frequently being used in decision making systems.

We have evaluated the accuracy score while generating predictive models and used the score to adjust the hyperparameters of the learning algorithms. We have taken AEQUITAS from the GitHub repository[3]. It was hardcoded for working with Adult dataset only. We have modified its code to make it work for any dataset. We have obtained the implementation of SG from [24]. For AEQUITAS and SG we have chosen the setting which gives the best performance. We have created oracle data in the verification-based testing approach by using a combination of random data instances and training data instances.

Finally, because all three approaches involve some sort of randomness, every experiment was carried out ten times. The results give the average of test cases generated out of these ten turns. 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 experimental evaluation.

RQ1. Tables 1 and 2 show the results of the experiments for RQ1. They give the number of test cases generated by each of the technique while using the four ML models obtained by training on Adult and German credit dataset, respectively. It can be inferred from the results shown that in most of the cases our technique can generate a larger number of test cases in comparison to SG and AEQUITAS.

RQ2. Figure 7 represents our results for the two pruning strategies for Adult and Credit datasets. Here, we compute individual discrimination with respect to the feature 'Gender'. It is evident from the results that data instance (short,

[3] https://github.com/sakshiudeshi/Aequitas

Table 2. Comparison of number of test cases generated for German credit dataset

Classifier	Prot. feature	VBT	SG	AEQUITAS
Logistic Regression	Gender	**182**	60	38
Logistic Regression	Age	127	**164**	3
Random Forest	Gender	173	**211**	200
Random Forest	Age	**130**	118	4
Naive Bayes	Gender	**16**	3	0
Naive Bayes	Age	**27**	20	0
Decision tree	Gender	**453**	135	259
Decision tree	Age	**444**	155	3

instance) pruning performs better than branch pruning in computing individual discrimination.

We have also performed experiments to evaluate the efficiency of our approach. We have observed that for all the test cases we consider here, the maximum run time of our approach is 929.76 seconds. On average, our approach needs 450 seconds to generate test cases.

4.3 Limitations and Threats to Validity

Since we employ an SMT solver for computing fairness, verification-based testing is restricted to feature values and operations allowed by the solver. The datasets we consider in this work containing integer and real values. For categorical feature values, an encoding needs to be performed. But this is frequently done by ML algorithms in data preprocessing step.

A threat to the internal validity is the high degree of randomness involved in the techniques. First, the classifiers considered here use randomized algorithms for generating models. Thus, in principle we might get varying number of test cases to unfairness when training *with the same classifier on exactly the same data set*. To ensure a *fair* comparison, all three approaches were always started with the same model as input (training of MUTs is external to testing). All three approaches randomly generate data instances (AEQUITAS and SG for test data and VBT for oracle data). In addition to that, VBT and SG use a decision tree training algorithm which itself involves randomness. To mitigate these threats, all experiments were performed 10 times and the results give the average over these 10 runs.

5 Related work

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

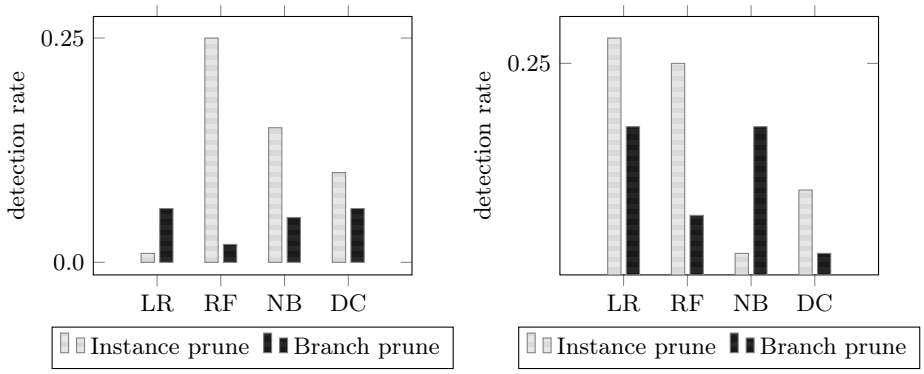

Fig. 7. Performance of data instance and branch pruning in computing individual discrimination for Adult (left) and Credit dataset (right)

Fairness testing. There exists a number of works discussing testing fairness of ML models. Galhotra et al. first propose [8] black-box testing of predictive model for fairness. They introduce individual discrimination as a definition of fairness and give a confidence driven random testing technique to check it. They also experimentally show why the existing fairness definitions are not enough to detect discrimination in the ML model. However, their approach is less effective as they solely rely on random test case generation.

Later, Udeshi et al. propose AEQUITAS [20] for individual discrimination detection. It first randomly searches for discriminatory input test cases. Once such test cases have been found, it tries to generate more test cases by perturbing the initial ones. They propose an automated technique to use the generated test cases for retraining the given model and then obtaining a fair one.

Our approach is closest to the work of Aggarwal et al. [1]. They generate a path of a decision tree from the black-box model under test by using a tool called LIME. This tool generates a small decision tree for *local* explanations of the predictions given by the model. After generating such a *partial* decision tree, they use dynamic symbolic execution to generate multiple test cases violating *individual discrimination*. In contrast, we approximate the entire black-box model by a decision tree. We then compute the test inputs on this tree. Also, we cater for checking the predictive model with respect to several fairness definitions instead of just individual discrimination.

In a more recent work, Zhang et al. [24] propose a fairness testing technique for Deep Neural Networks (DNN). They focus on individual discrimination by generating test cases through adversarial sampling. Like AEQUITAS, they also operate in two phases. Although their approach outperforms existing fairness testing approaches, they perform a white-box testing technique, hence are limited to DNNs.

Validating models. There exists a number of recent works which aim at validating properties of predictive models. One such important property of an ML model is *robustness*. In [10], Huang et al. first propose this 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. Pei et al. [15] later propose the first white-box testing technique based on differential testing approach to test DNNs for *robustness*. Sun et al. [19] propose a concolic testing approach to test DNNs.

In a very recent work, Lee et al. [11] propose a white-box testing technique for DNNs to generate test cases for checking *robustness*. They use an online algorithm to select the relevant neurons during the testing process and thus do not rely on a fixed strategy of selecting neurons for coverage. The experimental results show the effectiveness of their approach across diverse DNN models.

Recently, Sharma et al. [18] 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. Instead of checking the predictive model, this work focuses on testing the learning algorithm. A survey of different important properties arising in Machine Learning and their validation techniques can be found in [23].

Testing via model inference. The inference of a decision tree describing the behaviour of software has already been pursued by Papadopoulus and Walkinshaw [14] as well as Briand et al. [2].

The former work is related to ours as it also translates the decision tree to logical formula in Z3. However, they do not use the tree to compute counter examples to the property to be tested. Instead, they use Z3 to generate test inputs covering different branches of the tree. Our approach on the other hand use the white-box model to generate targeted test inputs by using an established verification technique. Briand et al. 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.

6 Conclusion

In this work, we have proposed a novel approach to test ML model for a user given fairness property. Our technique approximates the black-box model by a white-box model and then applies SMT solving techniques to compute fairness. It allows the user to specify the required fairness constraint herself, and with this goes beyond current fairness testing with hardcoded requirements.

We have evaluated the effectiveness of our approach by applying it to several ML models and found our approach to perform better than the existing fairness testing approaches in testifying a particular type of fairness in a large number of cases.

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 robustness, just by applying a different check on the generated SMT code. Also, we would like to improve our framework by incorporating additional techniques to check statistical measures of fairness.

References

1. Aggarwal, A., Lohia, P., Nagar, S., Dey, K., Saha, D.: 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. pp. 625–635 (2019). https://doi.org/10.1145/3338906.3338937, https://doi.org/10.1145/3338906.3338937
2. Briand, L.C., Labiche, Y., Bawar, Z., Spido, N.T.: Using machine learning to refine category-partition test specifications and test suites. Information & Software Technology **51**(11), 1551–1564 (2009). https://doi.org/10.1016/j.infsof.2009.06.006, https://doi.org/10.1016/j.infsof.2009.06.006
3. Calders, T., Kamiran, F., Pechenizkiy, M.: Building classifiers with independency constraints. In: ICDM Workshops 2009, IEEE International Conference on Data Mining Workshops, Miami, Florida, USA, 6 December 2009. pp. 13–18 (2009). https://doi.org/10.1109/ICDMW.2009.83, https://doi.org/10.1109/ICDMW.2009.83
4. Carlini, N., Wagner, D.A.: Towards evaluating the robustness of neural networks. In: 2017 IEEE Symposium on Security and Privacy, SP. pp. 39–57 (2017). https://doi.org/10.1109/SP.2017.49, https://doi.org/10.1109/SP.2017.49
5. Chouldechova, A.: Fair prediction with disparate impact: A study of bias in recidivism prediction instruments. Big Data **5**(2), 153–163 (2017). https://doi.org/10.1089/big.2016.0047, https://doi.org/10.1089/big.2016.0047
6. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. In: (ICFP '00). pp. 268–279 (2000). https://doi.org/10.1145/351240.351266, https://doi.org/10.1145/351240.351266
7. Dwork, C., Hardt, M., Pitassi, T., Reingold, O., Zemel, R.S.: Fairness through awareness. In: Goldwasser, S. (ed.) Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012. pp. 214–226. ACM (2012). https://doi.org/10.1145/2090236.2090255, https://doi.org/10.1145/2090236.2090255
8. Galhotra, S., Brun, Y., Meliou, A.: Fairness testing: testing software for discrimination. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 498–510. ACM (2017)
9. Guidotti, R., Monreale, A., Ruggieri, S., Turini, F., Giannotti, F., Pedreschi, D.: A survey of methods for explaining black box models. ACM Comput. Surv. **51**(5), 93:1–93:42 (2019). https://doi.org/10.1145/3236009, https://doi.org/10.1145/3236009
10. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Computer Aided Verification - 29th International Conference, CAV. pp. 3–29 (2017). https://doi.org/10.1007/978-3-319-63387-9_1, https://doi.org/10.1007/978-3-319-63387-9_1
11. Lee, S., Cha, S., Lee, D., Oh, H.: Effective white-box testing of deep neural networks with adaptive neuron-selection strategy. In: ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020. pp. 165–176. ACM, https://doi.org/10.1145/3395363.3397346
12. Liptak, A.: Sent to prison by a software program's secret algorithms. https://nyti.ms/2qoe8FC, accessed: 2017-05-01

13. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008. pp. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24, https://doi.org/10.1007/978-3-540-78800-3_24
14. Papadopoulos, P., Walkinshaw, N.: Black-box test generation from inferred models. In: RAISE. pp. 19–24 (2015). https://doi.org/10.1109/RAISE.2015.11, https://doi.org/10.1109/RAISE.2015.11
15. Pei, K., Cao, Y., Yang, J., Jana, S.: Deepxplore: Automated whitebox testing of deep learning systems. In: Proceedings of the 26th Symposium on Operating Systems Principles,. pp. 1–18 (2017). https://doi.org/10.1145/3132747.3132785, https://doi.org/10.1145/3132747.3132785
16. Segura, S., Fraser, G., Sánchez, A.B., Cortés, A.R.: A survey on metamorphic testing. IEEE Trans. Software Eng. **42**(9), 805–824 (2016). https://doi.org/10.1109/TSE.2016.2532875, https://doi.org/10.1109/TSE.2016.2532875
17. Sharma, A., Wehrheim, H.: Higher income, larger loan? monotonicity testing of machine learning models. In: ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, 2020. pp. 200–210. ACM, https://doi.org/10.1145/3395363.3397352
18. Sharma, A., Wehrheim, H.: Testing machine learning algorithms for balanced data usage. In: 12th IEEE Conference on Software Testing, Validation and Verification, ICST. pp. 125–135 (2019), https://doi.org/10.1109/ICST.2019.00022
19. Sun, Y., Wu, M., Ruan, W., Huang, X., Kwiatkowska, M., Kroening, D.: Concolic testing for deep neural networks. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE. pp. 109–119 (2018). https://doi.org/10.1145/3238147.3238172, https://doi.org/10.1145/3238147.3238172
20. Udeshi, S., Arora, P., Chattopadhyay, S.: Automated directed fairness testing. In: Huchard, M., Kästner, C., Fraser, G. (eds.) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018. pp. 98–108. ACM (2018). https://doi.org/10.1145/3238147.3238165, https://doi.org/10.1145/3238147.3238165
21. Verma, S., Rubin, J.: Fairness definitions explained. In: International Workshop on Software Fairness, FairWare@ICSE. pp. 1–7 (2018), http://doi.acm.org/10.1145/3194770.3194776
22. Zafar, M.B., Valera, I., Gomez Rodriguez, M., Gummadi, K.P.: Fairness constraints: Mechanisms for fair classification. arXiv preprint arXiv:1507.05259 (2017)
23. Zhang, J.M., Harman, M., Ma, L., Liu, Y.: Machine learning testing: Survey, landscapes and horizons. IEEE Transactions on Software Engineering pp. 1–1 (2020)
24. Zhang, P., Wang, J., Sun, J., Dong, G., Wang, X., Wang, X., Dong, J.S., TING, D.: White-box fairness testing through adversarial sampling. In: ICSE '20: 42nd ACM SIGSOFT International Conference on Software Engineering, Virtual Event, South Korea, 2020. ACM