Testing Machine Learning Algorithms for Balanced Data Usage

Arnab Sharma
Department of Computer Science
Paderborn University
Paderborn, Germany
Email: arnab.sharma@uni-paderborn.de

Heike Wehrheim
Department of Computer Science
Paderborn University
Paderborn, Germany
Email: wehrheim@uni-paderborn.de

***Abstract*—With the increased application of machine learning (ML) algorithms to decision-making processes, the question of *fairness* of such algorithms came into the focus. Fairness testing aims at checking whether a classifier as “learned” by an ML algorithm on some training data is biased in the sense of discriminating against some of the attributes (e.g. gender or age). Fairness testing thus targets the *prediction* phase in ML, not the learning phase.**

In this paper, we investigate fairness for the *learning* phase. Our definition of fairness is based on the idea that the learner should treat all data in the training set equally, disregarding issues like names or orderings of features or orderings of data instances. We term this property *balanced data usage*. We consequently develop a (metamorphic) testing approach called TILE for checking balanced data usage. TILE is applied on 14 ML classifiers taken from the `scikit-learn` library using 4 artificial and 9 real-world data sets for training, finding 12 of the classifiers to be unbalanced.

***Keywords*-ML classifier, fairness, balancedness, metamorphic testing**

I. Introduction

In recent decades, machine learning (ML) has emerged as one of the most important concepts applied in decision-making processes. This emergence is particularly due to the vast amount of data available for training. Today, machine learning has applications in many domains ranging from logistics and transportation to healthcare or even law (see e.g. [1], [2], [3]). As ML systems are being used in many critical applications, the correctness guarantees of such systems became a major concern.

One property of ML algorithms which recently came into focus is *fairness*. Fairness is studied in the context of *supervised* machine learning where the input to the learning algorithm is a set of labelled training data. The ML algorithm (in its learning phase) learns a predictive model generalizing from this training data as to make predictions of labels (classes) for unknown data instances. While fairness definitions for supervised machine learning are numerous (see e.g. [4] for an overview), they all refer to the same basic property: a predictor is unfair (or discriminating) if a change to the value of a feature in a data instance leads to a change in the prediction. For instance, a loan-granting software is discriminating against “address” if a change of the address of a person while keeping all of the person’s other data flips the decision from “yes” to “no”. Fairness testing [5] aims at detecting such sort of discrimination in ML predictors by systematically investigating how flipping feature values in data changes the prediction outcome.

Most often, the reasons for unfairness lie in the data set used for training. Biased training data almost inevitably leads to unfair predictors as it is the explicit purpose of ML algorithms to find a good generalization of its training data. Hence, fairness testing can only show unfairness in the predictor but does not allow for any conclusions about the fairness in the *learning* phase.

In this paper, we study fairness of the learning phase. Our objective is to determine whether the learner is only learning what is in the training data or is (potentially by design) biased. However, it is essentially unclear what the “correct” outcome of the learning process is, i.e. unclear what it means to “learn what is in the training data”. The ground truth to compare the learned predictor against is unknown, or – in terms of testing terminology – the *test oracle* is missing. Instead of comparing to some ground truth, we thus define “learning what’s in the data” by *using all training data in the same way*. We call this concept *balanced data usage*. Balanced data usage thus requires to disregard aspects like names chosen for features or orderings of data instances during learning. Technically, we formalize balanced data usage by defining a number of *metamorphic transformations* [6] on training data and specify the intended change (namely, no change) in the outcome of learning when applying these transformations (the *metamorphic relations*). Our metamorphic transformations capture (a) permutation of training data instances, (b) permutation of feature ordering, (c) shuffling of feature names and (d) renaming of feature values. An ML algorithm is *balanced* when an application of a metamorphic transformation on training data results in equivalent predictors.

Based on this definition, we develop a framework called TILE for balancedness testing of machine learning algorithms. A challenge in this approach is the equivalence checking of predictors. To the best of our knowledge no previous work has targeted systematic equivalence checking

978-1-7281-1736-2/19/$31.00 ©2019 IEEE
DOI 10.1109/ICST.2019.00022

of ML predictive models. Here, we propose two techniques neither of which is unfortunately sound *and* complete: (1) *Computing* equality on the models' representations and (2) *testing* equality of models. The first technique might give false negatives (the representation is different but the learned model still the same), the second technique false positives (no tests have revealed a difference though there is some). We use both techniques in combination to achieve a high degree of trustworthiness.

We then use TiLe to evaluate state-of-the-art ML classifiers. Our major finding is that 12 out of 14 classifiers of the `scikit-learn` ML repository are unbalanced.

Summarizing, this paper makes the following contributions:

- We develop a formalization of balanced data usage via metamorphic relations.
- We develop a testing approach for checking for balanced data usage. This includes compiling sets of training data for the learning phase.
- We present an approach for checking equivalence of ML predictive models, which encompasses both equivalence *computation* and *testing*.
- We systematically evaluate our approach on ML algorithms taken from the `scikit-learn` library [7].

The paper is structured as follows: We next introduce our definition of balancedness of ML classifiers. In Section III we explain our approach for testing ML classifiers for balancedness. Afterwards we present the results of our experimental evaluation. We discuss related work in Section V and conclude in Section VI.

II. Balanced data usage

We start by introducing some basic terminology in machine learning and then proceed to define our concept of balanced data usage. Supervised machine learning aims at "learning" certain tasks from *given* training data (hence, supervised), most often by using some statistical techniques. In this paper, we only consider *classification*, i.e., the assignment of a class to some given input data.

More formally, the purpose of an ML algorithm is to learn a function (the *predictive model* or short model)

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

where X_i is the value set of *feature* i (or attribute or characteristics i), $1 \le i \le n$, and Y is the set of *classes*. We write $\vec{X}$ for $X_1 \times \ldots \times X_n$. Thus, during prediction the learned model assigns a class $y \in Y$ to a data instance $(x_1, \ldots, x_n) \in X_1 \times \ldots \times X_n$. We let $\mathcal{M}$ be the set of such models (over arbitrary value sets and classes). Two models M_1, M_2 are said to be *equivalent* (denoted $M_1 \equiv M_2$) if $M_1(x_1, \ldots, x_n) = M_2(x_1, \ldots, x_n)$ for all $(x_1, \ldots, x_n) \in X_1 \times \ldots \times X_n$.

To learn such a model, a supervised ML algorithm is presented with *labelled training data* $T = ((x_{11}, \ldots, x_{1n}), y_1), \ldots, ((x_{k1}, \ldots, x_{kn}), y_k)$ consisting of a number of data instances (here k) and their class labels y_i, $1 \le i \le k$, and in practice also with a list $F = (\xi_1, \ldots, \xi_n)$ of feature *names*. We let $\mathcal{T}$ be the set of all training data and $\mathcal{F}$ the set of all feature name vectors. In the following we will also view training data as a *matrix* (two-dimensional array) where the rows are the instances in the training data, the columns give values for one feature and the last column $n+1$ is the vector of class labels appearing in the training data. The ML algorithm tries to generalize from the training data as to be able to later predict classes for unknown data instances. A *learner* is thus a function

$$learn : \mathcal{F} \times \mathcal{T} \to \mathcal{M}$$

(assuming without formalization that the number of features fits the number of columns in the training data).

Balanced data usage requires the learner to use all training data in the same way. It should neither treat some features differently from others nor should the position of a data instance in the training data influence the learned model. Formally, we require ML algorithms to be *invariant* under certain transformations carried out on their inputs. More precisely, we require ML algorithms to return the same model when trained on inputs which are related in a specific way. In the context of metamorphic testing, such relations on inputs are called *metamorphic relations*. Here, we call them *metamorphic transformations* since we construct new inputs by transformation of existing ones. Our metamorphic transformations are defined via permutations, i.e. bijective functions $\pi : \mathbb{N}_m \to \mathbb{N}_m$ where $\mathbb{N}_m = \{1, \ldots, m\}$. The set of all such permutations form the group S_m. Note that S_m has $m!$ elements. We consider the following metamorphic transformations:

Permutation of rows This transformation changes the ordering of rows in the training data. Formally, given a permutation $\pi \in S_k$ and training data $T = ((x_{11}, \ldots, x_{1n}), y_1), \ldots, ((x_{k1}, \ldots, x_{kn}), y_k)$ (i.e. with $n+1$ columns and k rows), we get $\pi(T)$ is $((x_{\pi(1)1}, \ldots, x_{\pi(1)n}), y_{\pi(1)}), \ldots, ((x_{\pi(k)1}, \ldots, x_{\pi(k)n}), y_{\pi(k)})$.

Permutation of columns Permutation of columns changes the ordering of n columns (excluding the class column) according to a permutation $\pi \in S_n$, i.e., the training data $\pi(T)$ for the T given above is $((x_{1\pi(1)}, \ldots, x_{1\pi(n)}), y_1), \ldots, ((x_{k\pi(1)}, \ldots, x_{k\pi(n)}), y_k)$.

Shuffling of feature names The third transformation simply permutes the feature names. We apply a permutation $\pi \in S_n$ on the input $F = (\xi_1, \ldots, \xi_n)$ and get $\pi(F) = (\xi_{\pi(1)}, \ldots, \xi_{\pi(n)})$. Note that this permutation is applied on features names only, not on the training data.

Renaming of feature values Finally, we replace categorical values of features by numerical ones (like a string "male" as the value of a feature "gender" by the number 1). To this end, we assume all value sets to be finite and let $|X_i| = \ell_i$, $1 \leq i \leq n$. The replacement is given by bijective functions $num_i : X_i \to \mathbb{N}_{\ell_i}$ (where $\mathbb{N}_m = \{1, \ldots, m\}$) and its application on the training set yields $((num_1(x_{11}), \ldots, num_n(x_{1n})), y_1), \ldots, ((num_1(x_{k1}), \ldots, num_n(x_{kn})), y_k)$.

Note that the first three types of transformations define a whole set of functions (e.g. the set of all permutations). We call all these functions *mm-transformations* and use $\mathcal{G}$ to denote this set. We furthermore let $Types = \{rp, cp, fs, fr\}$ be the set of mm-transformation *types* (i.e., row and column permutation, feature shuffling and feature renaming), and let $type(g) \in Types$ be the type of an mm-transformation g.

To have a unified way of applying mm-transformations, we lift all functions to the entire input $(F, T) \in \mathcal{F} \times \mathcal{T}$ to a learner. This finally lets us define balancedness as invariance under all mm-transformations.

Definition 1. *An ML algorithm learn* $: \mathcal{F} \times \mathcal{T} \to \mathcal{M}$ *is* balanced *if for all mm-transformation* $g \in \mathcal{G}$*, feature vectors and training data* $(F, T) \in \mathcal{F} \times \mathcal{T}$*,* $learn(F, T) \equiv learn(g(F, T))$[1].

If an ML algorithm is not invariant under a certain mm-transformation, we say that it is *sensitive* to the mm-transformation.

The mm-transformations that we use here are domain independent. Thus we expect all ML classifiers to be invariant under these transformations irrespective of the training algorithm. Domain specific mm-transformations can be found by having a deeper look at the algorithm, like done in [8]. One could use such transformations while considering some specific ML classifiers, but not for all of them.

We believe that an ordinary software developer (i.e., without special training in machine learning) taking some standard ML classifier from a library for use in his/her own software intuitively expects the classifier to not be sensitive to any of these mm-transformations. The next section presents an approach for checking balancedness of ML classifiers via testing.

III. Testing approach

As our definition of balancedness is given in terms of metamorphic relations, we use metamorphic testing to check for balancedness. Metamorphic testing compares the outputs of different executions of a program with respect to some fixed relation (in our case model equivalence), where the program is executed on inputs obtained by applying metamorphic transformations on some given fixed input.

[1]Note that when applying column permutations, we also need to apply the same permutation on the inputs to the learned model when comparing for equivalence. For simplicity, we did not include this aspect in the formalization.

A. Overview

Figure 1 depicts the overall workflow of our approach. The classifiers to be checked for balancedness are taken from a repository of ML algorithms. In our case, the classifiers in the ML repository are expected to be written in Python. In addition, there is a repository of training data to be used for testing which is part of our framework TiLe. During one test run, a classifier *learn* is taken from the ML repository and features and training data set (F, T) from our training data repository. The latter is used to train the classifier directly (Training 1) as well as train the classifier after performing a metamorphic transformation g on it (Training 2). The resulting models M_1 and M_2 are then given to the equivalence checker. The equivalence checker also obtains (F, T) as input as to generate test data for equivalence checking when necessary (see below).

For every classifier in the ML repository this procedure (in principle) needs to be carried out for

- every mm-transformation $g \in \mathcal{G}$ and
- every set (F, T) in the training data repository TD.

Unbalancedness of the classifier is testified when a single run outputs "no". Besides in the occurrence of a single yes/no answer, we are also interested in the *relative* unbalancedness, individually per type of mm-transformation and in summary. For an mm-transformation g and training input (F, T), we let

$$diff(F, T, g, learn) = \begin{cases} 1 & \text{if } learn(F, t) \not\equiv learn(g(F, T)) \\ 0 & \text{else} \end{cases}$$

and define $G_t(F, T) = \{g \in \mathcal{G} \mid type(g) = t \wedge g \text{ is applicable to } (F, T)\}$. Here, applicability simply means that the transformation matches the training data, e.g. that a permutation fits to the number of rows. The *transformation specific balancedness indicator* for an ML-classifier *learn* wrt. to a transformation type t, $bi_t(learn)$, is then

$$\begin{aligned} bi_t(learn, F, T) &= \frac{\Sigma_{g \in G_t(F,T)} diff(F, T, g, learn)}{|G_t(F, T)|} \\ bi_t(learn) &= \frac{\Sigma_{(F,T) \in TD} bi_t(learn, F, T)}{|TD|} \end{aligned}$$

and the *overall balancedness indicator* is

$$bi(learn) = \frac{\Sigma_{t \in Types} bi_t(learn)}{|Types|}$$

Note that both indicators of course depend on the training data. Furthermore, we cannot exactly compute these indicators in practice as the number of transformations is often too large to be exhaustively checked (e.g., the number of permutations of n rows is $n!$ and our training data contains sets with up to 30,000 data instances). Our tool is thus approximating the indicators. We will detail below how to choose e.g. permutations.

To put our testing concept into practice, all parts of the workflow need to be instantiated.

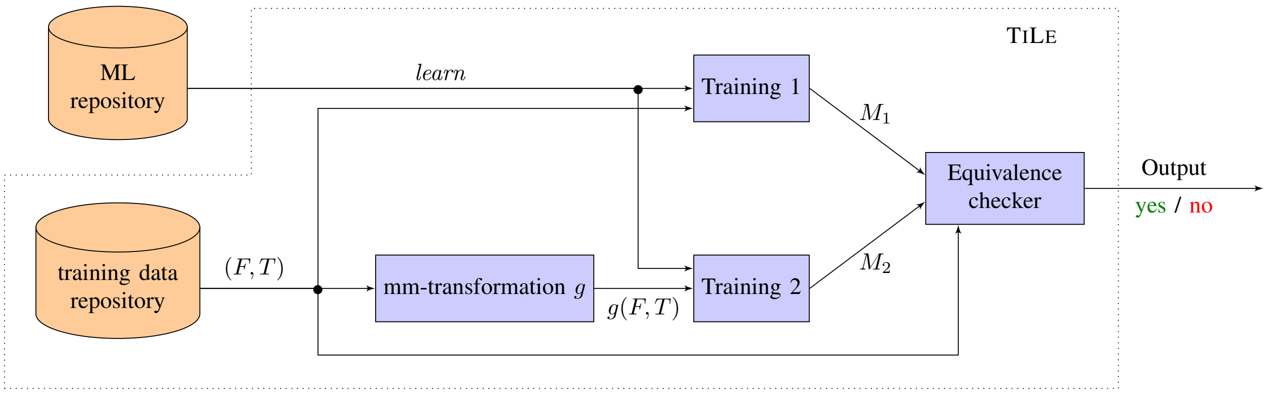

Figure 1. Workflow of our testing approach

- For the ML repository we have chosen `scikit-learn` as example because it is frequently used for various machine learning applications and our approach currently requires the classifiers to be written in Python.
- For the training data repository, we have assembled a set of real-world training data as well as constructed some training data ourselves. This will be explained in more detail in Section IV.
- As explained above, we need to choose the mm-transformations we actually check.
- Somewhat surprisingly, equivalence checks for ML models seem to not have been studied so far. We propose two techniques here, one based on *computing* equivalence by comparing the model representations and the other on *testing* for equivalence.

We discuss the latter two aspects in more detail.

B. Metamorphic transformations

As we cannot exhaustively apply all permutations on rows and columns, we use the following strategy for permutation selection. For column permutations only the first strategy is used.

- **Random**: we sample `PERM_RATIO` percentage of the number of all permutations uniformly at random, where `PERM_RATIO` is user configurable,
- **Ordering**: we give the classes in Y an ordering and *sort* the training data in an ascending and descending order (keeping the ordering of the training data within classes),
- **Alternating**: we reorder the training set so that it alternates between classes,
- **Reversing**: we invert the order of training data,
- **Batch-flipping**: we take a batch of 100 training data instances and move them to the end of the training data.

C. Model equivalence checking

One of the difficulties in designing a model equivalence checker is the fact that (a) a model itself can be a very complex entity and (b) the models learned by different classifiers vary significantly. To understand this, we explain three classifiers in more detail.

A *support vector machine* (SVM) [9] in the simplest case learns a hyperplane used for separating data instances[2]. The hyperplane is (in case of two classes) defined as a function $h : X_1 \times \ldots \times X_n \to \mathbb{R}$ of the following form

$$h(x_1, \ldots, x_n) = w_1 x_1 + \ldots + w_n x_n + b$$

In this, the values w_i are *weights* given to different features. A comparison of two models in this case is straightforward: we simply compare the weights plus the offset b.

Second, a *decision tree classifier* [10] – as the name says – builds a tree of (often binary) decisions in which the branches are tagged with conditions on feature values and the leafs are the classes. The left of Figure 2 gives an example of a decision tree stating the connection between a person's gender and age and the retirement[3]. At first sight, equivalence checking could be achieved by simply comparing the tree's branches and leafs. However, the decision tree on the right hand side of Figure 2 defines exactly the same predictive model. Thus decision trees are an example of a model type in which *different* representations define the same model. One way to still determine equivalence on decision trees is to translate both trees into logical formulae and check their logical equivalence. The translation proceeds by a path condition like construction, equating a class value $y \in Y$ with the disjunction over the path conditions for all branches leading to this class value. For example, for the

[2]In case of linearly separable data.

[3]Actually, the tree represents the retirement rules of a european country.

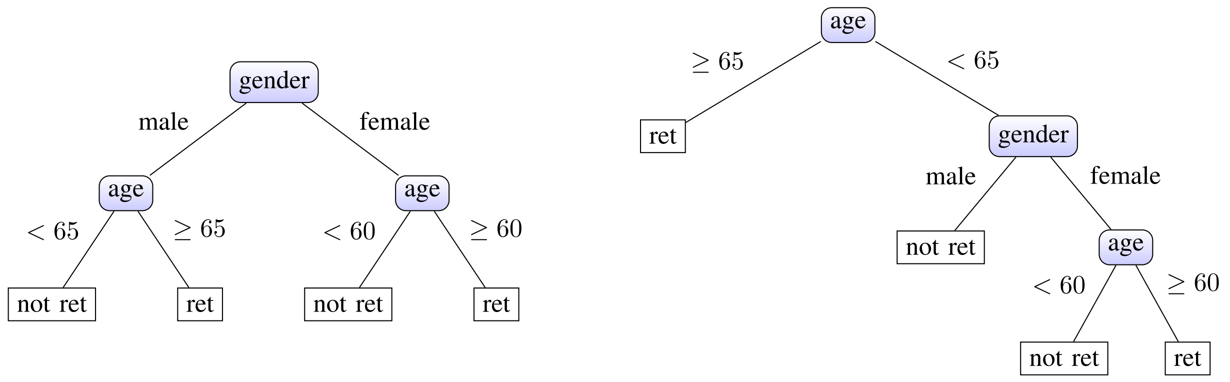

Figure 2. Two equivalent but syntactically different decision trees

Algorithm 1 *equiv* (testing equality of models M_1 and M_2 on given input data instance t)

```
Input: $M_1, M_2 \in \mathcal{M}$                ▷ models to be compared
       $t \in X_1 \times \ldots \times X_n$        ▷ test instance
Output: boolean                                  ▷ yes or no answer
1: $y_1 := M_1(t)$;
2: $y_2 := M_2(t)$;
3: return $(y_1 = y_2)$;
```

tree in the left-hand-side of Figure 2 we get the following:

$$\begin{aligned} ret &\Leftrightarrow \big((gender = male \wedge age \geq 65) \\ &\quad \vee (gender = female \wedge age \geq 60)\big) \wedge \\ non\_ret &\Leftrightarrow \big((gender = male \wedge age < 65) \\ &\quad \vee (gender = female \wedge age < 60)\big) \end{aligned}$$

This technique does however not scale to large decision trees. The trees constructed for our training data often have more than 10,000 nodes.

Third, a *k-nearest-neighbour* (k-NN) algorithm [11] does not build a model representation in its learning phase at all. Instead, it stores the training data and during prediction makes a majority vote on the k nearest neighbours of the test instance (where "nearest" is defined according to some distance function, typically Euclidean distance). Thus there are plainly no models on which we could compute equivalence.

Currently, we have methods for computing equivalence for the following classifiers (from the repository `scikit-learn` which we used): Neural Network, SVM, (partially for) Decision Tree, Logistic Regression and AdaBoost. This equivalence computation is summarized in a method called *computeEqui* and the method *equiComputable* checks on the type of a classifier whether a method for equivalence computation is available.

For all the other classifiers (and for decision trees when they get too large), we *test* for equivalence. Again, we cannot do exhaustive testing and thus employ some strategies for test input selection.

- **Random:** Like for permutations, some test inputs for equivalence checking are chosen randomly, where we take `INPUT-RATIO` percentage of all possible test inputs.
- **Training inputs:** We (also randomly) take `TRAIN-RATIO` percentage of the data instances from the training data T.
- **Min/Max/Median/Mean:** For every feature, we compute the minimal and maximal value of this feature in the training set as well as the median and arithmetic mean of all values. We then construct one test instance for each of these values.

We use the method `random` to generate a value of a set Z given as a parameter uniformly at random. We cache all computed test inputs as to avoid testing the same input twice. Algorithm 2 summarizes the procedure for equivalence testing. It uses the method *equiv* as given in Algorithm 1.

Finally, Algorithm 3 defines how equivalence computation and testing is combined. Therein, we use the method *type* which returns the type of a model (e.g., hyperplane or decision tree) as to determine the availability of an equivalence computation method.

IV. Evaluation

We have implemented our approach in a tool called TiLe (TestIng balancedness of machine LEarning). The tool is written in Python and relies on training data given in `csv` format. The evaluation aimed at checking the effectiveness of our own approach as well as evaluating existing ML classifiers with respect to balancedness.

A. Research Questions and Evaluation Plan

For the evaluation, we were interested in the following research questions:

RQ1 Are state-of-the-art ML classifiers balanced?
RQ2 Is TiLe able to detect unbalancedness in classifiers?
RQ3 Is our equivalence testing approach able to detect non-equivalent predictive models?

Algorithm 2 *equiTest* (testing equivalence of models M_1 and M_2 obtained from training data (F, T))

```
Input: $M_1, M_2 \in \mathcal{M}$                ▷ models to be compared
       $(F, T) \in \mathcal{F} \times \mathcal{T}$         ▷ features and training data
Output: boolean                          ▷ yes or no answer

1: ts:= $\emptyset$; count :=0;
2: no-random := $\frac{\text{INPUT-RATIO} \times |\vec{X}|}{100}$;
3: while count < no-random do
4:     t := random($\vec{X}$); count := count + 1;
5:     if (t $\notin$ ts) then
6:         ts := ts $\cup$ {t};
7:         if $\neg equiv(M_1, M_2$,t) then
8:             return false;

9: no-train := $\frac{\text{TRAIN-RATIO} \times |T|}{100}$; count :=0;
10: while count < no-train do
11:     t := random($T$); count := count + 1;
12:     if (t $\notin$ ts) then
13:         ts := ts $\cup$ {t};
14:         if $\neg equiv(M_1, M_2$,t) then
15:             return false;
16: for f $\in \{min, max, median, mean\}$ do
17:     for $i$ from 1 to $|F|$ do
18:         t[i] := f($i, T$);      ▷ compute f-value of feature $i$
19:     if (t $\notin$ ts) then
20:         ts := ts $\cup$ {t};
21:         if $\neg equiv(M_1, M_2$,t) then
22:             return false;
23: return true;
```

Algorithm 3 *equi* (checking equivalence of models)

```
Input: $M_1, M_2 \in \mathcal{M}$                ▷ models to be compared
       $(F, T) \in \mathcal{F} \times \mathcal{T}$         ▷ features and training data
Output: boolean                          ▷ yes or no answer

1: equal := false;
2: if $equiComputable(type(M_1))$ then          ▷ compute equival.
3:     equal := $computeEqui_{type(M_1)}(M_1, M_2)$;
4: if (equal) then
5:     return true;
6: else                                    ▷ test equivalence
7:     equal:= $equiTest(F, T, M_1, M_2)$;
8: return equal;
```

RQ4 Which permutation strategy is most effective for finding non-balancedness?

To evaluate these four research questions, we planned for the following experiments.

RQ1. We chose one of the most frequently used machine learning libraries, namely `scikit-learn`, and applied our approach on all its 14 classifiers. This also required setting up a training data repository which we explain below. We then applied all sorts of mm-transformations as explained above with one exception: renaming of feature values. The transformation of categorical into numerical values has to be carried out for all classifiers from `scikit-learn` anyway (the classifiers require this format), and thus does not need to be checked. All classifiers are thus insensitive to renaming of feature values.

RQ2. For the second research question, we looked at ML classifiers which are known to be "unbalanced" in some sense. There are some proposals for *discrimination aware* ML algorithms. In the learning phase, these get a feature (or a group of features) as input and try to learn a model which is not discriminating against this feature (group). These algorithms inherently require an unbalanced treatment of training data. We applied our approach to the discrimination aware classifiers of Zafar et al. [12] and Calders et al. [13] (called Fa1 and Fa2, respectively, in our tables). As input, these two classifiers got one arbitrary feature of the data set.

A further sort of unbalanced classifiers are those using *window sizes* for their training data, i.e., only consider data instances in a training set up to the window size. As these ignore training data from out of the window, they are inherently unbalanced. We use the k-NN classifier from `scikit-learn` and call the version with window size taking effect k-NN$_{ws}$. We have fixed the window size for k-NN$_{ws}$ to be 500, which means it will only consider the last 500 training instances it has seen while predicting the class for each of the test instances.

In addition, we have written an unbalanced classifier called UnBa ourselves. It basically ignores one feature (the last one in F).

RQ3. With respect to equivalence testing, we wanted to find out whether our testing approach is able to detect non-equivalent predictive models. To this end, we generated non-equivalent models and evaluated whether our equivalence checking procedure is able to detect the difference. As our testing approach employs the training data for test input selection, we needed models constructed by a classifier from training data. For this, we took training data from our repository and modified a fixed percentage of class labels. We then trained the classifier with the original and the modified data set thereby getting two models. These models are taken for the evaluation of the equivalence checker. We furthermore varied our two testing parameters `TRAIN_RATIO` and `INPUT-RATIO` to see how these affect equivalence testing.

RQ4. For research question 4, we took the experiments for RQ1 and recorded for the row permutations how often a certain strategy shows a sensitivity to row permutation, and whether there is some unbalancedness which can only

Table I
REAL-WORLD DATASETS

#Features	#Instances	Name
8	90	*Immuno-Therapy*
10	699	*Breast-Cancer*
7	20560	*Occupancy*
56	32	*Lung-Cancer*
20	1000	*German-Credit*
14	48842	*Census-Income*
102	74	*SE Data*
309	126	*Voice-Recognition*
128	1994	*Crime Data*

Table II
SENSITIVITY TO METAMORPHIC TRANSFORMATIONS (MEASURED ON ARTIFICIAL AND REAL-WORLD DATA SETS)

Classifiers (e-comp)	*FN shuffle*	*Row perm.*	*Col. perm.*
k-NN (No)	✗	✓	✗
Decision Tree (Yes)	✗	✓	✓
Naive Bayes (No)	✗	✗	✗
SVM (Yes)	✗	✓	✗
Neural Network (Yes)	✗	✓	✓
Logistic Regression (Yes)	✗	✓	✓
Random Forest (No)	✗	✓	✓
AdaBoost (Yes)	✗	✓	✓
Bagging Classifier (No)	✗	✓	✓
Extra Trees (No)	✗	✗	✓
Gradient Boosting (No)	✗	✓	✓
Gaussian (No)	✗	✗	✗
Linear Regression (No)	✗	✓	✓
Elastic Net (No)	✗	✗	✓

be detected by a single strategy. For the latter aspect, we excluded random permutations as these can also randomly generate the permutations of one of the other strategies. We furthermore varied the parameter `PERM_RATIO` to see how it influences the detection of unbalancedness.

B. Setup

For our training data repository, we first of all developed 4 *artificial* training sets by hand. These are explicitly designed to exploit unbalancedness properties which we have found during our first experiments. For instance, some k-NN classifiers carry out *tie-breaking* according to the order of occurrence of a data instance in the training set. Permutation of rows can then give a different outcome. A training set exploiting this property thus needs to contain ties. Our synthetic training sets all contain 3 features and up to 6000 data instances.

To see whether unbalancedness also occurs on real-world data, we have in addition taken 9 training sets from the *UCI machine learning repository*[4]. These real-world data sets are chosen to cover a variation of number of features and data instances. Table I shows the characteristics of the data sets.

Finally, to ensure that exactly the same classifier is used in phase **Training 1** and **Training 2**, we fixed the hyperparameters of classifiers to default values. Hyperparameters are parameters of the learning algorithm itself (e.g., number of hidden layers in a neural network). Sometimes, this requires explicitly using a configuration in which no hyperparameter values are set by the classifier itself.

C. Results

Next, we report on the results for all research questions.

RQ1. Our first result is that 12 out of 14 classifiers show some form of unbalancedness. In our experiments, the Gaussian classifier is used with RBF kernel and the logistic link function and the AdaBoost classifier is used with decision tree stumps. Table II shows the sensitivity of a classifier's learning phase to the three metamorphic transformations *shuffling of feature names* (FN shuffle), *row permutation* and *column permutation*. It also indicates on which classifiers we could perform equivalence computation (e-comp). None of the classifiers from the library are sensitive to shuffling of feature names. For only 2 classifiers unbalancedness could not be detected.

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

To also see how often such sensitivities occur, we computed an approximation of the transformation specific balancedness indicators bi_t. Approximation here means that our values are relative to the number of tested mm-transformations. For this, we used the real-world training data sets only as to get a better idea for what happens in practice.

Table III shows the transformation specific balancedness indicator per classifier (given in percentage instead of a value in [0,1]). It shows that there is a significant difference in the indicators, ranging from below 1% to almost 50%.

These results show the classifier's sensitivities in our experiments, however, cannot explain them. Currently, we have identified five reasons for sensitivies:

- Tie-breaking: In case of ties, some algorithms use the ordering among features or in the training data to choose how to proceed. As already mentioned, this is the case for the k-NN classifier. It also applies to decision tree classifiers which compute entropies of features to decide for the labels of nodes at branches. In case of two features having the same entropy, the one appearing first in the feature list F is chosen.
- Randomness: Some ML classifiers inherently involve some sort of randomness, most notably random forests, and hence already give different results for different runs even when the training data is not touched at all.
- Imprecise numerical calculations: The numerical calculations employed in ML classifiers are inherently imprecise and hence get sensitive to reordering of

Table III
TRANSFORMATION SPECIFIC BALANCEDNESS INDICATORS (MEASURED ON REAL WORLD DATA SETS ONLY)

Classifiers	*FN shuffle*	*Row perm.*	*Col. perm.*
k-NN	0.0	1.24	0.00
Decision Tree	0.0	2.89	35.19
Naive Bayes	0.0	0.00	0.00
SVM	0.0	9.85	0.00
Neural Network	0.0	1.05	25.65
Logistic Regression	0.0	0.63	0.02
Random Forest	0.0	27.97	26.26
AdaBoost	0.0	0.03	1.66
Bagging Classifier	0.0	28.61	16.73
Extra Trees	0.0	0.00	19.05
Gradient Boosting	0.0	0.70	1.45
Gaussian	0.0	0.00	0.00
Linear Regression	0.0	0.76	0.54
Elastic Net	0.0	0.00	11.10

Table IV
SENSITIVITY TO METAMORPHIC TRANSFORMATIONS OF DIFFERENT FAIR/UNFAIR CLASSIFIER

Classifiers	*FN shuffle*	*Row perm.*	*Col. perm.*
k-NN$_{ws}$	✗	✓	✗
Fa1	✓	✗	✗
Fa2	✓	✗	✗
UnBa	✗	✗	✓

arguments (e.g., fail to be commutative).

- Batch processing: A number of classifiers process the training data in batches and the models they learn depend on the contents within batches and their ordering.
- Initialization: Neural networks initialize weights in the layers in an asymmetric way as to make the neural network learn better.

RQ2. Table IV shows the sensitivity results for the intentionally unbalanced classifiers. We see that all four classifiers can be identified as being unbalanced. As expected, k-NN$_{ws}$ is sensitive to row permutation because the use of a window size lets the classifier learn from different data after permuting rows. The two explicitly fair classifiers are sensitive to shuffling of feature names as they explicitly try to be fair wrt. some features and the change of feature names leads to models being fair wrt. different features. Finally, the hand-written unbalanced classifier ignoring the last feature is sensitive to column permutation.

RQ3. For RQ3 we constructed non-equivalent predictive models by changing a fixed percentage of class labels and training the classifiers on original and changed training set. Basically, all training data sets involve binary classification and thus have two classes. The change thus involves flipping the class labels.

With a **2%** flip, our testing approach could detect non-equivalence of predictive models for all classifiers except

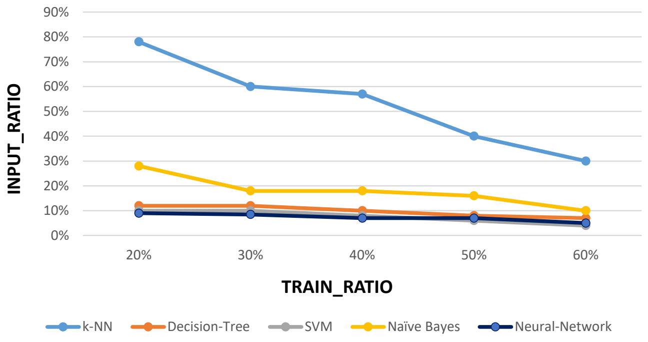

Figure 3. Relation between TRAIN_RATIO and INPUT_RATIO

k-NN using TRAIN_RATIO = 20% and INPUT_RATIO = 20%. The classifier k-NN already needs 20%/45% (TRAIN_RATIO, INPUT_RATIO) for detection. To evaluate different values of TRAIN_RATIO and INPUT_RATIO we furthermore used a 1% flip to construct predictive models which only slightly differ. Figure 3 shows the results of our evaluation (for classifiers k-NN, Decision Tree, SVM, Naive Bayes and Neural Network). On the x-axis different TRAIN_RATIO values are shown. The y-axis shows the minimal INPUT_RATIO needed to detect non-equivalence. Here, we again see that non-equivalence detection for k-NN needs a larger amount of test inputs. For the other classifiers values of 30% for TRAIN_RATIO together with 20% for INPUT_RATIO seem to be a good choice for detecting non-equivalence. These values are, however, specific to the sort of change we have applied to the training data (class flipping) and are thus difficult to generalize to other cases.

RQ4. We have measured the effectiveness of different *row permutation* strategies for those classifiers which are sensitive to such mm-transformations. Figure 4 shows the average deviation (averaged over all classifiers) caused by different row permutation strategies, i.e. the percentage of test runs in which the permutations generated by a strategy were successful in showing non-balancedness. The results show that all strategies are approximately equal with respect to their effectiveness of detecting unbalancedness, with somewhat surprisingly "Reversing" taking the lead (surprising because "Reversing" generates a single permutation only). The plot also shows the minimal and maximal deviation of a strategy achieved for one classifier. All minimal values belong to the Logistic Regression, all maximal values to Bagging classifier. This also matches their high and low, respectively, balancedness indicators.

Table V furthermore shows the percentage of our test cases that are classified differently by original and transformed classifier after applying permutation "Reversing". For this, we ran Algorithm *equiTest* with all selected test cases, not stopping when a test case showing non-equivalence of models has been found. The table indicates that reversing the training dataset in particular has a signif-

Table V
PERCENTAGE OF TEST CASES CLASSIFIED DIFFERENTLY BY THE ORIGINAL AND TRANSFORMED CLASSIFIER AFTER "REVERSING"

Classifiers	*Test cases*
k-NN	0.41
Decision Tree	0.66
Naive Bayes	0.00
SVM	19.99
Neural Network	2.35
Logistic Regression	0.48
Random Forest	11.35
AdaBoost	1.00
Bagging Classifier	11.59
Extra Trees	0.00
Gradient Boosting	0.86
Gaussian	0.00
Linear Regression	0.53
Elastic Net	0.00

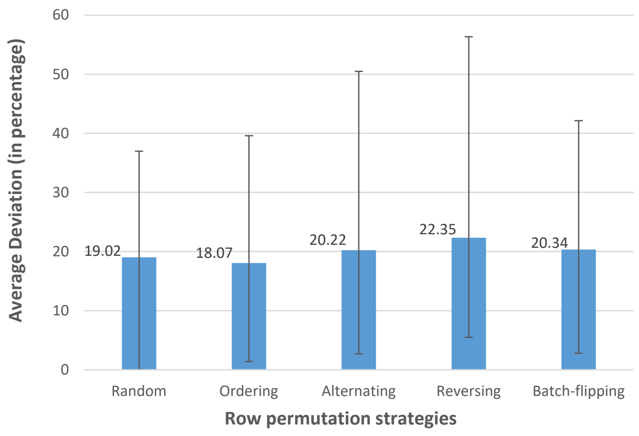

Figure 4. Deviation detected by different row permutation strategies

icant impact on the training of the SVM classifier. A lot of classifiers only show very little difference (below 1% of the test cases).

We have also conducted experiments to find out the best `PERM_RATIO` value for TILE. It has turned out that increasing its value above 5% does not help to detect significantly more deviations. Thus, in our experiments with TILE we have used a `PERM_RATIO` value of 5%.

D. Threats to Validity

There are several threats to the validity of our results. The first is the randomness involved in our experiments. Because we randomly choose permutations during metamorphic transformations and test inputs during equivalence checking, the runs of TILE are not easily repeatable. Similarly, some classifiers involve random computations and their runs are not repeatable. This does not influence the results in Table II since this table only shows that there is unbalancedness in some run of the classifier on some transformation of training data. The numbers of Table III do, however, depend on the random choices taken and thus can only be seen as approximating the real balancedness indicators.

The next threat to validity could be our choice of training data. It could be possible that unbalancedness only occurs on this training data. We tried to lower this threat by taking real-world, publically available training data.

The third threat to validity is the correctness of our implementation. We extensively tested TILE. However, as the outcome of the experiments is essentially unclear, the presence of conceptual errors in the code is difficult to detect. The construction of artifical training sets, where we knew the outcome for some classifiers, and similarly, the development of intentionally unbalanced classifiers improved on this situation, but these examples are not representative for the majority of cases.

V. RELATED WORK

Our work is in line with a number of recent works on validating properties of machine learning algorithms. We separately discuss those targetting the *prediction* and the *learning* phase.

With respect to prediction, there are a number of recent works looking at *verification* of (deep) neural networks via SMT-solving [14], [15], [16] or abstract interpretation [17]. These works aim at proving specific, user specified properties of ML models like the holding of postconditions on outputs of a trained neural network when fed with inputs satisfying a given precondition. Testing of ML applications, in particular self-driving cars, is studied by Pei et al. [18]. A fault of an ML application ML1 is therein defined as deviation in the behaviour from that of similar applications ML2 and ML3. This way, they circumvent the problem of not knowing the "correct" predictive model. Testing of deep neural networks (DNNs) is also studied in [19], [20]. They in particular focus on different coverage requirements for DNNs (like neuron coverage). For our equivalence testing, we could not work with such coverage criteria as the classifier is basically a black box for us and we wanted to develop an equivalence testing technique applicable to any classifier.

A well-defined and well-studied property on predictive models of ML algorithms is *fairness*. Verma and Rubin [4] give a survey of different such fairness definitions. They all target the discrimination of feature groups in the prediction. A testing approach for fairness is given by Galhotra et al. [5]. Albarghouthi et al. [21] propose FairSquare, a tool employing probabilisitic verification to determine fairness of (machine learning) decision makers.

Investigations in properties of the *learning* phase almost exclusively employ metamorphic testing (see [22], [23] for surveys). The work closest to us is that of Xie et al. [24] (and previous approaches like [25]). Like us, they use various metamorphic relations to test the "correctness" of machine learning classifiers. Unlike us, they do not require every classifier to pass the same set of metamorphic transformations, but divide them into necessary and expected properties per classifier. Since we concentrate on a specific property (balancedness), only one of their metamorphic relations agrees with one of ours: MR-1.2 Permutation of Attributes. In the experiments, Xie et al. [24] study two classifiers (k-NN and Naives Bayes taken from the WEKA repository) and use artificially constructed training data sets only. For their testing approach, they do not detail which permutations they use for testing MR-1.2 nor do they detail how they choose from the other set of metamorphic transformation (e.g. affine transformations allow for basically an infinite number of options).

In [8] Nakajima et al. propose a systematic way to derive metamorphic properties for ML classifiers. Basically, their idea is to manually look at the algorithm and decide which metamorphic transformations are appropriate. They apply their technique on an SVM implementation. They also propose a method for finding out appropriate test data to check for the metamorphic relations, but for SVMs only.

Some approaches use metamorphic relations to improve the accuracy of the classifier. Accuracy here means that the predictive model produces correct results on *known* data instances (i.e., where the class labels are given). To this end, a typically 10-fold cross validation is carried out on the training data. Ding et al. [26] use metamorphic transformations for evaluating the training data with respect to the accuracy of the classifier obtained when using this training data. Their transformations include removing, adding or duplicating instances in the training data. A recent work proposes the use of application specific metamorphic relations to enhance the performance of the classifiers: In [27] Xu et al. adopt metamorphic relations to change a certain sort of images (artcode) which are the inputs to the classifier based on knowledge about the classification task. Conceptually, this is some form of data preprocessing, detailed towards the intended classification.

Permutation of features and class labels in the training data is also used to study the accuracy of the classifiers. In [28] Ojala et al. inspect the effect of these permutations on the training data on the classifiers's predictions.

Finally, there are two approaches providing validation techniques for balancedness-like properties which are not based on metamorphic testing. Chen et al. [29] study the problem of "reducer commutativity", i.e., the question whether reducers in a Map-Reduce framework are insensitive to commutation of inputs. Their approach is based on model checking, but requires the input data to be of a fixed length. A static analysis approach for determining *unused data* in programs has recently been proposed by Urban et al. [30]. Not using data (e.g., not using certain feature values) is another sort of unbalancedness of classifiers and we intend to incorporate this into our testing approach in the future.

VI. Conclusion

In this paper, we proposed the new notion of *balancedness* for machine learning classifiers. Balancedness is a property of the learning phase of a classifier, not its prediction, and thereby complements existing fairness definitions for predictive models.

We developed TiLe, a metamorphic testing framework for balancedness. TiLe encompasses both a systematic way of generating metamorphic transformations as well as a technique for validating equivalence of predictive models.

We evaluated the effectiveness of our testing framework and employed it to check the classifiers of the standard ML library `scikit-learn` for balancedness. Our major finding is that a large percentage of these ML classifiers are not balanced. As future work, we plan to analyze the classifiers in the WEKA repository [31].

References

[1] A. Liptak, "Sent to prison by a software program's secret algorithms," https://nyti.ms/2qoe8FC, accessed: 2017-05-01.

[2] N. J. Goodall, "Can you program ethics into a self-driving car?" *IEEE Spectrum*, vol. 53, no. 6, pp. 28–58, 2016.

[3] P. Olson, "The algorithm that beats your bank manager," https://www.forbes.com/the-algorithm-that-beats-your-bank, accessed: 2011-03-15.

[4] S. Verma and J. Rubin, "Fairness definitions explained," in *International Workshop on Software Fairness, FairWare@ICSE*, Y. Brun, B. Johnson, and A. Meliou, Eds. ACM, 2018, pp. 1–7. [Online]. Available: http://doi.acm.org/10.1145/3194770.3194776

[5] S. Galhotra, Y. Brun, and A. Meliou, "Fairness testing: testing software for discrimination," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 498–510.

[6] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, Tech. Rep., 1998.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[8] S. Nakajima and H. N. Bui, "Dataset coverage for testing machine learning computer programs," in *Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific*. IEEE, 2016, pp. 297–304.

[9] V. Vapnik, *Statistical learning theory*. Wiley, 1998.

[10] L. Breiman, *Classification and regression trees*. Routledge, 2017.

[11] Z. John Lu, "The elements of statistical learning: data mining, inference, and prediction," *Journal of the Royal Statistical Society: Series A (Statistics in Society)*, vol. 173, no. 3, pp. 693–694, 2010.

[12] M. B. Zafar, I. Valera, M. Gomez-Rodriguez, and K. P. Gummadi, "Fairness constraints: Mechanisms for fair classification," in *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA*, ser. Proceedings of Machine Learning Research, A. Singh and X. J. Zhu, Eds., vol. 54. PMLR, 2017, pp. 962–970. [Online]. Available: http://proceedings.mlr.press/v54/zafar17a.html

[13] T. Calders, F. Kamiran, and M. Pechenizkiy, "Building classifiers with independency constraints," in *ICDM Workshops 2009, IEEE International Conference on Data Mining Workshops, Miami, Florida, USA, 6 December 2009*, Y. Saygin, J. X. Yu, H. Kargupta, W. Wang, S. Ranka, P. S. Yu, and X. Wu, Eds. IEEE Computer Society, 2009, pp. 13–18. [Online]. Available: https://doi.org/10.1109/ICDMW.2009.83

[14] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety verification of deep neural networks," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 3–29.

[15] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient smt solver for verifying deep neural networks," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 97–117.

[16] R. Ehlers, "Formal verification of piece-wise linear feed-forward neural networks," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2017, pp. 269–286.

[17] 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 *2018 IEEE Symposium on Security and Privacy, SP*. IEEE, 2018, pp. 3–18. [Online]. Available: https://doi.org/10.1109/SP.2018.00058

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

[19] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE*, M. Huchard, C. Kästner, and G. Fraser, Eds. ACM, 2018, pp. 109–119. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238172

[20] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th International Conference on Software Engineering, ICSE*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 303–314. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180220

[21] A. Albarghouthi, L. D'Antoni, S. Drews, and A. V. Nori, "Fairsquare: probabilistic verification of program fairness," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 80, 2017.

[22] S. Segura, G. Fraser, A. B. Sánchez, and A. R. Cortés, "A survey on metamorphic testing," *IEEE Trans. Software Eng.*, vol. 42, no. 9, pp. 805–824, 2016. [Online]. Available: https://doi.org/10.1109/TSE.2016.2532875

[23] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 4:1–4:27, 2018. [Online]. Available: http://doi.acm.org/10.1145/3143561

[24] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software*, vol. 84, no. 4, pp. 544–558, 2011.

[25] C. Murphy, G. E. Kaiser, L. Hu, and L. Wu, "Properties of machine learning applications for use in metamorphic testing." in *SEKE*, vol. 8, 2008, pp. 867–872.

[26] J. Ding, X. Kang, and X.-H. Hu, "Validating a deep learning framework by metamorphic testing," in *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*. IEEE, 2017, pp. 28–34.

[27] L. Xu, D. Towey, A. P. French, S. Benford, Z. Q. Zhou, and T. Y. Chen, "Enhancing supervised classifications with metamorphic relations," in *Proceedings of the 3rd International Workshop on Metamorphic Testing*. ACM, 2018, pp. 46–53.

[28] M. Ojala and G. C. Garriga, "Permutation tests for studying classifier performance," *Journal of Machine Learning Research*, vol. 11, pp. 1833–1863, 2010.

[29] Y. Chen, C. Hong, N. Sinha, and B. Wang, "Commutativity of reducers," in *TACAS*, ser. Lecture Notes in Computer Science, C. Baier and C. Tinelli, Eds., vol. 9035. Springer, 2015, pp. 131–146. [Online]. Available: https://doi.org/10.1007/978-3-662-46681-0_9

[30] C. Urban and P. Müller, "An abstract interpretation framework for input data usage," in *ESOP*, ser. Lecture Notes in Computer Science, A. Ahmed, Ed., vol. 10801. Springer, 2018, pp. 683–710. [Online]. Available: https://doi.org/10.1007/978-3-319-89884-1_24

[31] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1656274.1656278