Carl von Ossietzky Universität Oldenburg
Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Testing of Machine Learning Algorithms and Models

Von der Fakultät für Informatik, Wirtschafts- und Rechtswissenschaften der Carl von Ossietzky Universität Oldenburg zur Erlangung des Grades und Titels eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

angenommene Dissertation von
Arnab Sharma

Gutachter
Prof. Dr. Heike Wehrheim
Prof. Dr. Michael Leuschel

Tag der Disputation: 28.11.2023

Abstract

Machine learning (ML) based software systems are increasingly being used in several application domains affecting our daily lives substantially. In many of those domains, ML software has shown exceptional performances, however, has also shown catastrophic failures. This raises concerns about applying such software to critical application areas and therefore, it is necessary to test whether ML software conforms to specific properties before deploying them for a specific application.

In this thesis, we present testing approaches considering the entire *pipeline* of the ML software systems. To this end, we first consider testing the learning algorithms and then the *models* which are generated after the learning phase. For testing the learning algorithms part, first of all, we gave a specific property of the ML algorithms, termed as *balanced data usage*. We formally defined this and then presented a *metamorphic* testing approach to test the property. This approach is further implemented in a testing tool which we evaluated on diverse ML algorithms taken from prominent ML libraries. Surprisingly, we found a large number of ML algorithms do not conform to this property.

Depending on the domain of the application, the ML models (i.e., the ML software systems) must guarantee certain properties such as fairness, monotonicity, security, and so on. To this end, in the last few years, there has been a plethora of work being done proposing verification and validation approaches for checking specific properties on ML models. These approaches either consider a specific type of ML model, for instance, deep neural networks or check a specific property such as robustness or fairness. However, there might be cases when the ML model is completely *black-box* to us, i.e., we do not have any knowledge about it. We might furthermore require that the ML model conforms to different kinds of properties. Considering these two cases, we developed a testing approach that allows black-box testing of ML models and contains a specification framework that allows the testers to specify the properties of their choices. Thus, our approach is independent of the ML models and their properties. We termed this as *property-driven* testing approach.

We implemented the property-driven testing approach in a tool and evaluated it in testing diverse properties on different types of ML models. The results of the evaluations have shown that MLCHECK could outperform even the tools specifically designed for a specific property. Furthermore, it was able to outperform all the existing black-box testing approaches in finding out violations of the corresponding properties.

Zusammenfassung

Auf maschinellem Lernen (ML) basierende Softwaresysteme werden zunehmend in verschiedenen Anwendungsbereichen eingesetzt, die unser tägliches Leben wesentlich beeinflussen. In vielen dieser Bereiche hat ML-Software außergewöhnliche Leistungen gezeigt, aber auch katastrophale Ausfälle verursacht. Daher ist es notwendig zu testen, ob eine ML-Software den spezifischen Anforderungen entspricht, bevor sie für eine bestimmte Anwendung eingesetzt wird.

In dieser Arbeit werden Testansätze vorgestellt, welche die gesamte Pipeline von ML Software Systemen berücksichtigen. Zuerst wird das Testen von ML Algorithmen betrachtet, gefolgt vom Testen der ML Modelle, die im Wesentlichen als ML Software Systeme verwendet werden. Zu diesem Zweck wurde zunächst eine spezifische Anforderung an die ML Algorithmen formuliert, die als *balanced data usage* bezeichnet wird. Diese Eigenschaft wurde formal definiert und dann wurde ein *metamorpher* Testansatz vorgestellt, um sie zu testen. Auf der Grundlage dieses Testansatzes wurde TiLe entwickelt und mit verschiedenen ML-Algorithmen aus mehreren bekannten ML-Bibliotheken evaluiert. Überraschenderweise wurde festgestellt, dass mehrere ML-Algorithmen diese Anforderung nicht erfüllen.

Je nach Anwendungsbereich müssen die ML Modelle (d. h. die ML Software systeme) bestimmte Eigenschaften wie Fairness, Monotonie, Sicherheit usw. garantieren. Zu diesem Zweck wurde in den letzten Jahren eine Fülle von Arbeiten durchgeführt, in denen Verifikations- und Validierungsansätze zur Überprüfung bestimmter Eigenschaften von ML Modellen vorgeschlagen wurden. Diese Ansätze berücksichtigen entweder einen bestimmten Typ von ML Modellen, z. B. Deep Neural Networks, oder prüfen eine bestimmte Eigenschaft wie Robustheit oder Fairness. Es kann jedoch auch Fälle geben, in denen die ML Modelle eine vollständige *black-box* sind, d. h., dass keine Informationen über sie bekannt sind. Außerdem könnte verlangt werden, dass die ML Modelle verschiedene Arten von Anforderungen erfüllen. Unter Berücksichtigung dieser beiden Fälle wurde ein Testansatz entwickelt, der Black-Box-Tests von ML Modellen ermöglicht und einen Spezifikationsrahmen enthält, der es den Testern ermöglicht, die Eigenschaften ihrer Auswahl zu spezifizieren. Somit ist der Ansatz unabhängig von den ML Modellen und ihren Eigenschaften. Dies wird als eigenschaftsgesteuerter Testansatz bezeichnet.

Der eigenschaftsgesteuerte Testansatz wurde in einen Tool namens MLcheck implementiert und beim Testen verschiedener Eigenschaften für unterschiedliche Arten von ML Modellen evaluiert. Die Ergebnisse der Evaluierungen haben gezeigt, dass MLcheck sogar die speziell für eine bestimmte Eigenschaft entwickelten Tools übertreffen konnte. Darüber hinaus war es in der Lage, alle bestehenden Black-Box-Testansätze beim Auffinden von Verstoßen der entsprechenden Eigenschaften zu übertreffen.

Acknowledgements

When I started writing this part of my thesis, so many wonderful people came to my mind that I hope in the end I could acknowledge all of them. Without them, this thesis would probably not have been done so easily.

First and most importantly I would like to thank my supervisor Heike Wehrheim for all the guidance, support, and patience. During my Ph.D, more often I would come in with a chaotic pile of questions, problems, and issues while having meetings with her. She would always listen to me, refining the orders in my problems and discussing the solutions of my issues which would then clear my mind and the path ahead. This thesis would not have been possible without her guidance and indispensable feedback. I will always be grateful to her for allowing me to work in her research group and giving me academic freedom.

I would also like to thank my Ph.D. committee members Prof. Dr. Martin Fränzle, and Dr. Christian Schönberg, and the reviewer of my thesis Prof. Dr. Michael Leuschel for their feedback, comments, and all the help in making this thesis in a much better shape.

I am really lucky to have had some wonderful colleagues during my Ph.D. I am really thankful to Oleg Travkin with whom I initially shared my office. I would also like to express my appreciation to all my colleagues for insightful discussions, and constructive feedback, and for making my workplace so much more enjoyable. When I look back, I cannot remember how many countless times I went to them, asking questions, and they would always answer and discuss no matter how busy they were and I am wholeheartedly thankful for that. For you, I will always happily remember and cherish my time doing the Ph.D.

Thanks to all the coauthors and collaborators of all the research papers that we published together. In particular, I am really thankful to Vitalik Melnikov for all the discussions that I had with him which helped me a lot to understand the project we were working on and also to decide on my thesis topic. Since my thesis was on testing of machine learning systems, he being an ML expert would always give me constructive feedback on my research.

I want to thank Elisabeth Schlatt for her help with all the administrative stuff right from the beginning of my Ph.D. Because of her help, I could smoothly do all the paper-works needed to start in Germany. I am always thankful for all the assistance I still get from her. This work would also not have been possible without the funding from the CRC-901 (On-The-Fly Computing) project. This has also given me the opportunity to travel to different scientific conferences across the world.

I would like to thank all my friends and especially Chiara Cappello with whom I could always share my happiness, pains, and all those emotions that I had during my journey of working on this thesis. Being a Ph.D. student herself, she probably could understand it more than the others.

Finally, I am really thankful to my family and especially to my brother Apurba and my mother who have always encouraged me, no matter what. Last but not least I am really thankful to Jana who has always supported me and allowed me to work even during the weekends and holidays. This thesis would not have been possible without your constant support and compromises.

Contents

1 Introduction

In recent years, there have been increased advancements in the field of machine learning (ML). This enabled us to develop high performing ML-based software systems which are now effectively replacing humans in decision-making process. For instance, from autonomous cars [MWYY20] to loan granting software [GPKB20] and even in medical diagnosis and prognosis [SGSG19], software systems are controlling every aspect of our lives. Furthermore, in the USA the justice system uses ML software to decide who goes to jail and who is to be set free [VDO+19]. Due to such critical nature of the application areas, the need for the ML software systems to be working as expected is crucial.

This has prompted the stakeholders to mandate the assurance of the quality of ML-based software systems. Thanks to some quality research done by the researchers to this end, we can find a plethora of works [ALN+19, ZHML22, ZWS+20, BSS+19, ZVGG17] aims to ensure that such software meets the desired requirements. These works can be broadly classified into two major categories, (a) developing ML software by taking into account the desired requirements [ZVGG17, CKP09], and (b) *verifying* or *testing* the software after being developed [BSS+19, ZWS+20]. The research shows that the software which by design is ensured to guarantee a desired requirement, often does not perform as expected and the undesirable behaviour still persists [GBM17]. Therefore, verifying or testing the ML software is still required to perform to check for failures. Although a large body of works considers verification as a means to achieve a provable guarantee that the software conforms to the given set of requirements, it often does not scale to ML-based software systems because of high dimensional input space. Most importantly, verification requires access to the *internals* of the given software which might not be available in some cases. Therefore, in this dissertation, we primarily develop testing techniques to validate the ML-based software systems (without considering its internals) for finding out potential undesirable behaviours in its different *phases*.

Testing of ML-based software systems is an active research area and there exist a number of research works to this end. However, there are specific issues that are not addressed in the related works. In this Chapter, we start by briefly stating the research gaps as the motivation behind this dissertation and the problem at hand (Section 1.1). Then we describe the contributions and give the outline of this thesis in Section 1.2. Finally, we give the publications corresponding to the thesis in Section 1.3.

1.1 Motivation and Problem Statement

Motivation. Unlike traditional software, ML-based software systems are not developed by writing programs, rather the functionality of the software, i.e., what it is supposed to do, is *learned*. Therefore, the core of an ML-based software system is a *machine learning model* which is generated by training a machine learning (ML) algorithm on

a set of data *instances* (a set of vectors with fixed sizes). Essentially, a supervised[1] ML algorithm 'learns' the behaviour as a sort of generalization from a set of training data instances using statistical methods, thereby generating an ML model (or in short model). This model is then used as a decision-making software in several applications.

The emergence of the machine learning model as a decision-making software system has found its way into various critical application areas [DMBT17, McK20, YBK18, LBM+18, DHB20]. ML models are now being used in autonomous cars, replacing humans completely. For example, what started as a research project by Google back in 2009 of building *self-driving* cars, is now becoming a reality by Waymo[2]. They have already started commercializing autonomous cars in Phoenix and California in the USA. In hospitals, doctors are using ML software for early detection, prediction, and even diagnosis of several chronic diseases [YBK18]. What was seemingly impossible to understand like dementia, Alzheimer, and other brain-related diseases, a conjunction of ML models and medical research are on the verge of even curing those [NZK+20]. When someone applies for a loan in the bank, nowadays the application is granted or rejected based on the decision given by an ML-based software [3]. In some of the states in the USA, an ML-based decision support system–Correctional Offender Management Profiling for Alternative Sanctions (COMPAS)–is used in the judiciary system to determine the likelihood of a criminal doing a crime again and thereby deciding the sentencing of the offender [SCDG]. Unknowingly, we are led by ML-based software in what to buy (online shopping websites), which movies to watch, or where to go for vacations.

Although such ML software systems have shown to be performing astonishingly well in such areas, they are prone to undesirable behaviours which in some cases led to catastrophic failures. For example, in 2018 the e-commerce company Amazon.com found out that the machine learning algorithm it was using for recruiting employees, was biased as it was favoring the men [Das] in giving recruitment decisions. In the same year, within a span of a few months, an autonomous car from Uber fatally injured and killed a woman in Arizona [AAH], because the underlying ML model mistook the woman for something else. In another case, an ML model for predicting cancer treatment approaches, developed by IBM was recommending wrong treatments [Che]. Although these incidents[4] are isolated but they have one thing in common, failing of the ML software systems to function correctly.

Research gap. As the usage of the ML models in developing decision-making software is growing, the research in an effort to ensure the quality of the models is also increasing. Currently, there are two orthogonal approaches to achieve this goal: (a) designing an ML algorithm to generate an ML model which is guaranteed to satisfy a specific requirement, (b) verifying or testing the requirements of the ML model. Both of these approaches have their limitations and they do not encompass all the aspects of checking a machine learning *paradigm*. For instance, on the one hand, the first approach to designing ML algorithms is limited to a handful of requirements like fairness, and robustness. Moreover, some recent works suggest that these ML algorithms might actually not ensure generating ML models guaranteeing the corresponding requirement

[1] In this work, we only consider supervised learning, hence for the rest of the dissertation we will simply use 'ML algorithm'.
[2] https://waymo.com/
[3] https://sdk.finance/machine-learning-deep-learning-forecasting-for-banking-industry/
[4] see https://incidentdatabase.ai/summaries/incidents for more such incidents

(see [GBM17, ALN+19]). On the other hand, approach (b) is possible to use only when either a validation (verification or testing) technique exists to be applied on a specific *type* (e.g. neural network, random forest, etc.) of ML model [PLS20, TN20], or can only be used to check a *specific* property [ALN+19, XLLL23]. Moreover, the verification technique cannot be applied to a model when it is a *black-box* to us, i.e., we do not have any knowledge about the internals of the given model to check. So to summarize the problem, the existing verification and testing techniques are either restricted to a specific type of ML model or to a single property. This brings us to a question about the research gap,

> What if someone wants to check different properties without having the knowledge of the internals of an ML model?

A machine learning algorithm in its learning phase uses the training data to *generalize* from it in order to generate a *predictive function* or the ML model. There are specific requirements on the ML models that are defined in the related works and there are validation techniques for checking such requirements. Furthermore, there are techniques to develop a requirement-guaranteed learning algorithm that by design can ensure the requirement to a certain extent. However, this will not necessarily guarantee learning from the training data as 'expected'. Thus, in both of these cases, we can not draw any conclusion on the learning phase of an ML algorithm. In fact, it is essentially unclear what the *correct* outcome of the learning phase is, i.e., unclear what it means to learn what is in the training data. There are some works focusing on identifying the implementation bugs in the ML algorithms (see [PLQT19, DAS+18]. However, none of these works define the expected outcome by the ML algorithms or give validation techniques to check the outcome of the learning phase. So we ask the following question:

> How to define the requirement of the learning phase and check it?

Our solution. In this thesis, we focus on checking different phases of the machine learning paradigm. More specifically, we look at the learning phase of the ML algorithm in generating models and the *prediction* phase of the ML models as the predictive software. Since validating the requirements of the ML models only is not sufficient, we, therefore, also look at the learning phase. By validating, we mean testing the requirements of the machine learning paradigm. To this end, first of all, we define a concept called 'balancedness', describing what it means to be 'learning from the data', and subsequently give a testing mechanism to check it. Next, we give a testing technique that can be applied to check arbitrary types of *non-stochastic* requirements on the arbitrary types of machine learning models. This approach allows the testers to specify the requirement of interest and then a *targeted* test generation technique is performed to find out the violation of the given requirement.

We use testing instead of verification as a means of checking the requirements. There are two main reasons for this. First, there exist many different types of ML algorithms and the models, such as naive Bayes, random forests, logistic regression, etc. Therefore, to verify each of them, a separate verification technique is required. Second, these techniques might also differ based on the type of properties being considered for checking. The aim of our thesis is to develop validation mechanisms that could be

used for any type of ML algorithm or model. Thus, to achieve this goal we do not consider developing verification techniques, rather we develop two testing techniques which consider the algorithms and the models as *black-box*. In other words, the testing techniques we develop assume to have no knowledge of the given algorithms and the models to check.

1.2 Contributions and Structure of Thesis

After describing the necessary background on machine learning along with the relevant testing approaches and some related works in Chapter 2, the contributions of this thesis can be structured as follows:

Chapter 3 First, we give our approach of *logically encoding* two machine learning models, decision trees, and neural networks. This is required for the testing approaches that we describe in the subsequent chapters. More specifically, as part of the testing approach proposed for ML algorithms (in Chapter 4), we require the logical encoding of the decision tree model. Moreover, the testing approach proposed for ML models (in Chapter 5) also requires the encoding of decision trees and neural networks models. Apart from the encoding of the models, we also describe the approach for checking requirements on the logically encoded models. Thus, Chapter 3 gives a much required foundations of the testing approaches we propose later on.

Chapter 4 We present a testing approach to test the machine learning algorithms in Chapter 4. For this, we first define a property called balanced data usage, which formalizes a way to conceptualize the correct outcome of the learning phase. Then we propose a *metamorphic* testing approach [CKL+18] to test the property. Subsequently, we validate the property on several popular machine learning libraries such as `scikit-learn`, `WEKA`, `XGBoost`, `LightGBM`, and `CatBoost`.

Chapter 5 As mentioned earlier, we aim to develop a testing mechanism for ML models which is model *agnostic*, i.e., applicable to any type of model. To achieve this, we propose *verification-based testing* which is based on the idea of *learning-based testing* [AMM+18]. In this approach, an ML model is learned to approximate the *functional behaviour* of the model under test. Thereafter, a verification technique is applied to the learned model to analyze the given requirement on it. We describe in detail about this testing approach in Chapter 5.

Chapter 6 Next, we take the verification-based testing approach one step further. We develop a property specification mechanism on top of it to develop what we call a *property-driven* testing technique. This then allows the tester to specify any property in the *pre-* and *post-condition* format. Based on the specification, the underlying verification-based testing technique then uses the property to generate test cases in order to find a violation of the specified property. In Chapter 6 we define the syntax and semantics of the specification language and furthermore describe how we incorporate it to develop the property-driven testing approach.

Chapter 7 In Chapter 7, we give the evaluation results of our property-driven testing tool while applying it to validate diverse properties on the different types of ML models. While doing so, we also compare the performance of our tool with several

state of the art testing approaches and report the results. Moreover, we perform several experiments to argue about several design approaches that we have taken in developing this tool.

Chapter 8 At the end, we summarize the works done in this thesis and end it with some possible future works in Chapter 8.

Furthermore, the technical details of the tools we presented in this thesis are described in Appendix A. It also includes the links to all the artifacts developed as part of this thesis.

1.3 Publications

The work presented in Chapter 4 on testing of machine learning algorithms for balancedness was published in IEEE International Conference on Software Testing, Verification and Validation (ICST) in 2019 [SW19].

The concept of *verification-based* testing presented in Chapter 5 was published in ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA) in 2020 [SW20b]. Apart from presenting the testing technique, in this paper, we also used it to test the *monotonicity* property of the ML models.

In Chapter 6 we extend the idea of verification-based testing to develop the property-driven testing mechanism. We employed this testing approach for testing several types of *fairness* properties. This work was published in IFIP International Conference on Testing Software and Systems (ICTSS) in 2020 [SW20a]. Later we extended the specification language and applied our testing mechanism to test a security property and the concept relationships on specific types of ML models. This work was published in IEEE International Conference on Machine Learning and Applications (ICMLA) in 2021 and detailed in Chapter 7.

We furthermore validated diverse numerical properties on a specific type of ML model, namely the *regression* model by appropriately extending our approach. This work was published in ACM SIGSOFT International Conference on Formal Methods in Software Engineering (FormaliSE) in 2022.

2 Background

In this thesis, we focus on testing machine learning (ML) algorithms and learned models. Therefore, first of all, we start by giving a brief description of the necessary background in machine learning. More specifically, we explain the typical workflow of ML and some classification and regression algorithms that form the basis for the work of this thesis. Furthermore, we mention some application areas of ML and the properties of interest in such areas.

As mentioned before, we use testing as a means to validate several properties on the ML algorithms and models. So that the reader can comprehend the idea of software testing, we briefly describe the typical testing approach and later explain how some specific approaches are particularly useful in the context of ML testing. Finally, we end this chapter by describing some of the existing works, the research gap, and how our work address this gap in this thesis. As a quick outlook of this chapter, we focus on the following three aspects:

1. typical workflow of machine learning with necessary formalization, including a brief description of some algorithms (Section 2.1)

2. some traditional software testing approaches, primarily focus on the approaches we use in this work (Section 2.2)

3. related work of some specific ML testing approaches (Section 2.3)

2.1 Machine Learning Fundamentals

We start by introducing some basic terminologies in machine learning and then briefly describe some specific learning algorithms.

A supervised machine learning algorithm essentially has two main steps. First of all, in the learning phase, the *learning* algorithm generalises from a set of data instances called *training data* to generate a *predictive* function. This function is called the ML model or in short model. Next, this model is used in the *prediction* phase as a decision-making software to predict or classify classes for unknown data instances. Essentially, there exist two broad categories of learning algorithms depending on the nature of the class values: classification and regression. While the classification algorithm learns to classify integer values as the class values, regression learns to predict[1] real-valued numbers. Moreover, depending on the number of classes, i.e., whether a single class or a set of classes is predicted, the learning algorithm can be further categorized into single-label and multi-label learning approaches. Below we give a formalism, which caters to all these types of learning strategies.

Formally, the ML algorithm learns a predictive function or model as follows:

$$M : X_1 \times \ldots \times X_n \rightarrow Y_1 \times \ldots \times Y_m$$

[1]We use the term 'predict' for both classification and regression and 'classify' for only classification

where X_i is the value set of the *feature* i (also called attribute i) and $1 \leq i \leq n$, Y_j is the class for the jth *label* and $1 \leq j \leq m$. Depending on the *type* of feature values we can divide the features into three broad categories: categorical, numerical, and text features. Categorical features have a finite set of feature values and are more often defined as a *string*, such as features like gender which could be *male* or *female*. Numerical features as the name suggests, contain numerical values which can be integers or rational numbers. Finally, text features consist of text, such as product reviews or social media posts (containing a collection of strings) and only a specific type of classifiers (natural language processing algorithms) can be used to learn from them.

This formalism we defined so far will now enable us to define the different types of learning approaches. When $m > 1$, it is defined as *multi-label* learning and in case of $m = 1$, it is the *single-label* learning approach. Basically, a multi-label learning algorithm learns to predict a vector as output, whereas a single-label learner learns to predict a single value. Furthermore, when $|Y_i| > 2$ it is considered as a *multi-class* classification, and when $|Y_i| = 2$, it is a *binary* classification problem, and in case of regression, $Y_i \subseteq \mathbb{R}$.

We define $\vec{X}$ for $X_1 \times \ldots \times X_n$ and similarly $\vec{Y}$ for $Y_1 \times \ldots \times Y_m$. We use $\vec{x} \in \vec{X}$, and $y \in \vec{Y}$ as the corresponding class and therefore $(\vec{x}, y)$ forms an element of $\vec{X} \times \vec{Y}$. The training data consists of a set of such pairs $(\vec{x}, y) \in \vec{X} \times \vec{Y}$. The learning algorithm generalizes the dataset, using a statistical learning technique, thereby generating the function M to which when an $\vec{x}$ is given, an y as its class is predicted. We furthermore define $\vec{x}(i)$ as the ith feature of the data instance and similarly when considering multi-label learning y_j as the jth class. Based on this formalization, next we define some machine learning algorithms.

2.1.1 ML Algorithms and Models

In this section, first we give a brief description of some machine learning algorithms which are relevant in order to understand the work of this dissertation. Then we mention some of the most important properties of ML models based on the application areas.

k-nearest neighbors. The k-nearest neighbors (or in short k-NN) algorithm is a simple supervised learning algorithm that can be used for both classification and regression tasks. The learning process in this algorithm does not depend on the distribution of the training data, which is why it is called a *non-parametric* method. It assumes that similar instances stay together and therefore, it finds the k nearest data points for a given input instance and predicts the label of that instance based on the k data points found.

In its learning phase, the algorithm simply stores the training data instances. Later in its prediction phase, given an instance, it computes the distances from that instance to all the data instances of the training set. It then takes k-nearest instances and returns the class which occurs most frequently amongst these k instances. The distance metric used in this algorithm is most often the Euclidean distance measure, however, other distance measures (such as Manhattan or Minkowski) are also used. The value of k plays an important role and can be optimized depending on the problem at hand. On the one hand, a smaller value of k results in a low *bias* but high *variance* model [2]

[2] Please refer to this book [AMMIL12] for more on bias and variance of the machine learning models.

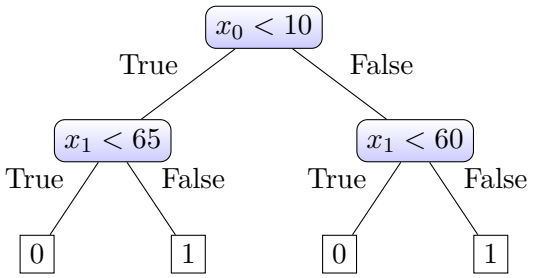


Figure 2.1: A decision tree of depth 1

and therefore, prone to overfitting, and on the other hand, a higher value of k results in a high bias but low variance model, which might lead to underfitting. Therefore, the optimal value for this depends on the specific dataset and task at hand.

Decision tree. Given a set of training instances $\vec{X} \times \vec{Y}$, a decision tree learner learns decision rules inferred from the features of the data instances. The collection of such rules forms a binary tree which is then used as a classification or a regression model. A perfectly balanced decision tree has 2^d number of nodes, where d is the depth of the tree. Each internal node corresponds to a decision function or a condition defined on a feature, is of the form $\vec{x}(i) \sim v_i$, where $\sim$ can be any conditional operators like $<, \leq, \geq$, etc. and v_i can be a value of type integer, or real[3].

Each of the decision function on a feature recursively splits the input space into two sub-spaces. The decision as to which feature should be used for splitting is done based on the *information theory*, essentially by calculating the *entropy* before and after applying the split. The feature, splitting on which would give the maximum information gain, is considered. In other words, we generate the decision function on a feature that would give us a more refined decision rule in predicting a class [Qui97]. The tree is evaluated in a top-down manner, i.e., starting with the root node of the tree and then subsequently satisfying the decision conditions in each internal node until reaching the leaf. When a leaf is reached, an output class associated with the leaf node is returned. Eventually, such a tree specifies a partition of the input space into *hyper-rectangles* and form a *piece-wise* constant approximation function. Further details of different types of decision trees and their applications can be found here [WKQ+08].

Figure 2.1 shows a simple classification tree of depth 1. Considering the dataset containing only two features x_0 and x_1, all inner nodes have conditions on the arguments x_0 and x_1 and one child for the `True` (left edge) and one for the `False` (right edge) case. However, it might happen that some features are not considered in the tree. This generally happens when either using a feature in a decision function does not result in gaining new information or simply the tree size needs to be reduced and hence, less important features are removed. For a concrete input to the model, the tree is traversed depending on the values of the features and their corresponding conditions on the nodes of the tree. The value of the leaf reached in this traversal gives the predicted output which in this example is either 0 or 1[4].

Neural network. This ML algorithm exists in different forms such as perceptron, feed-forward, convolutional, and more. Depending upon the *activation* functions used,

[3]Note that, we consider that the training dataset containing categorical attributes are pre-processed to yield a dataset containing only numerical values.

[4]Note that, for simplicity, we consider here the example of binary classification.

those networks can be further categorized. However, we do not discuss all of these different types (for interested readers we refer to this book [Agg18]), rather we focus on describing the-feed-forward neural network with ReLU (Rectified Linear Unit) as activation function, which is relevant to understand the work of this dissertation.

Figure 2.2 shows a feed-forward neural network and we consider this network to have ReLU as the activation function. A feed-forward network can be described as a sequence of layers starting with the input layer. Each layer consists of a number of neurons. In this example, for simplicity, we assume that we have only two hidden layers each consisting of three neurons. We consider an input instance of size 3, therefore, we have 3 neurons in the input layer and for simplicity, we consider the binary classification, hence, two neurons in the output layer [5].

Each neuron consumes a linear combination of the outputs of neurons from the previous layer. For example, the first neuron ($n_0^{(1)}$) of the hidden layer gets the values from the input layer and then sums them up with its bias as $x_0 * w_{00}^{(1)} + x_1 * w_{10}^{(1)} + x_2 * w_{20}^{(1)} + b_0^{(1)}$, where $w_{00}^{(1)}, w_{10}^{(1)}, w_{20}^{(1)}$ are the weights associated with the edges connecting the input layer neurons x_0, x_1, x_2, respectively, and the bias term $b_0^{(1)}$ in this case is 0.784. It then applies an activation function which in our case is ReLU (Rectified Linear Unit) [NH10], and can be described as $ReLU(x) = max(0, x)$. Thus, the output of the first neuron would be $ReLU(x_0 * w_{00}^{(1)} + x_1 * w_{10}^{(1)} + x_2 * w_{20}^{(1)} + b_0^1)$. The same process is repeated for all the other neurons in this layer. Similarly, a neuron in the second layer gets its inputs from the previous layer neurons. For example, assuming the outputs of the neurons $n_0^{(1)}$, $n_1^{(1)}$ and $n_2^{(1)}$ are $o_0^{(1)}$, $o_1^{(1)}$ and $o_2^{(1)}$ respectively. Now, the output of the first neuron $n_0^{(2)}$ would be $ReLU(o_0^{(1)} * w_{00}^{(2)} + o_1^{(1)} * w_{10}^{(2)} + o_2^{(1)} * w_{20}^{(2)} + b_0^2)$ where $w_{00}^{(2)}, w_{10}^{(2)}, w_{20}^{(2)}$ are the weights associated with the edges connecting the previous layer neurons. This is furthermore repeated for all the neurons in the second layer.

Finally, the output layer neuron does not apply ReLU and moreover, the number of neurons can vary depending on the problem at hand. For example, when the *sigmoid* function [GBC16] is applied to the output layer, the function bounds the output–which is calculated as a linear combination of the outputs of neurons of the previous layer plus the bias term of the output neuron–between 0 and 1. On the other hand, if we consider a *softmax* function [GBC16] for binary classification, then the number of neurons in the output layer is 2 and the output neuron which receives the maximum value from the previous layer is considered as the output class [6]. Essentially, the *softmax* function converts a vector of N real numbers into a probability distribution of N possible outcomes. For instance, in our example if the output neuron $n_0^{(3)}$ has got the maximum value from the previous layer then the predicted class would be 0. Note that, in case of multi-class classification, the number of neurons depends on the number of classes, and softmax is applied to choose the right class [HTF09]. Finally, in case of regression, no such function is applied to the output value and there would be a single neuron in the output layer.

The weights and biases of the neural network are initially set to some random values (although in some cases the initial values are determined from the training dataset). Afterward, the *backpropagation* algorithm [RHW86] is used to find the optimized weights

[5] Note that depending on the activation function chosen for the output layer the number of neurons in the output layer can also be one for binary classification.

[6] Note that, in this case the ascending order of classes are considered from top to bottom, i.e., the top most output neuron has the lowest class value and the bottom most has the highest.

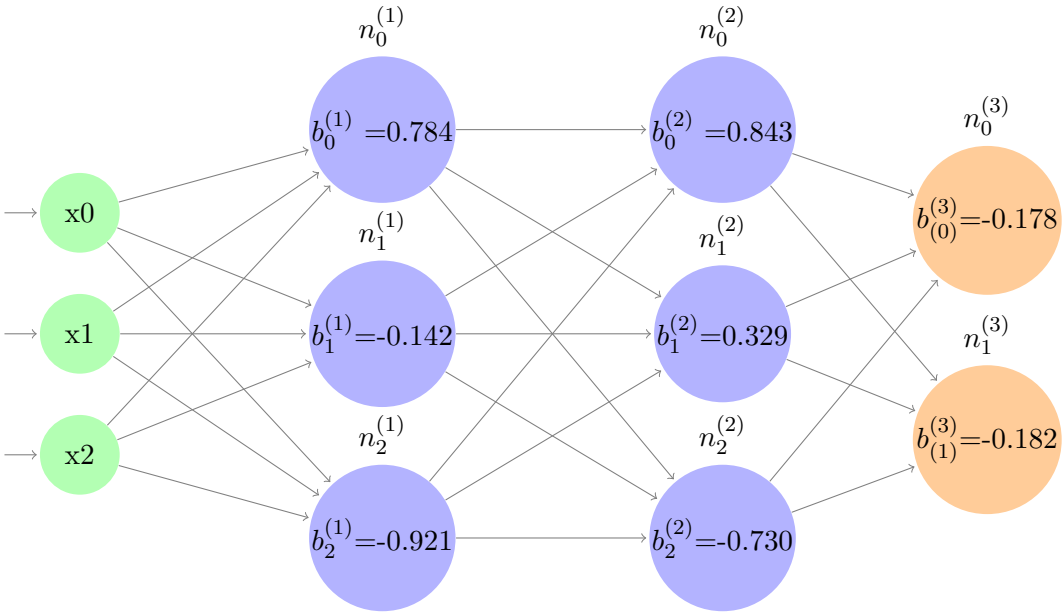


Figure 2.2: A feed-forward neural network

and biases. In its simplest form, starting with the first training instance, the algorithm gives the instance to the network and thereby gets a class value which might or might not be the actual class for the given instance. If the actual class is predicted, the algorithm proceeds with the next instance. If it is a different class, it updates the weights and biases so that the network learns the correct class for the first instance. However, this is inefficient and time consuming as often the number of training instances are huge as well as the size of the network. Therefore, modern-day algorithms use faster optimization algorithms such as gradient descent, stochastic gradient descent, etc. [Nie15] for this purpose.

Random forest. This is an *ensemble* learning algorithm, combining a collection of decision tree models where each model learns from a random subset of the training dataset [HTF09]. The idea of such an ensemble was proposed to improve the learning of a single decision tree. The decision tree learning algorithm is a simple, yet effective technique and in many cases works really well. However, when the size of the tree is too big, then it tends to *overfit* the training dataset, i.e., not giving good results on the unseen data instances. If the size of the tree is small it does not generalize well on the training dataset (*variance* problem). This is more commonly known as *bias-variance* tradeoff in learning (see [AMMIL12]). The random Forest learning algorithm is proposed for reducing the variance by getting an ensemble of a number of decision trees. Essentially, each tree generated in this ensemble is completely independent of each others as they learn from a subset of the training data instances randomly chosen with replacement. The collection of such trees forming the random forest model is said to be significantly reducing the variance of the model.

Formally, a random forest R can be defined as a collection of l decision trees, such as $R = \langle D_1 \dots D_l \rangle$, whereas the final predicted class for a given data instance is decided after evaluating it on all the trees of the ensemble. For an input instance $\vec{x} \in X_1 \times \dots \times X_n$, the output given by each tree is essentially an ordered set of probabilities, where each of the values in this set denotes the probability of predicting

the corresponding class [7]. The final class for $\vec{x}$ would be then the one with the highest mean probability estimated across all the decision trees of the ensemble. For example, suppose, p_i^j is the probability of class j given by tree i such that, $\sum_{j=1}^{k} p_i^j = 1$, where k is the number of classes and $1 \leq j \leq k$, $1 \leq i \leq l$. So for a class j, we have.

$$y^j = \frac{1}{l} \sum_{i=1}^{l} p_i^j$$

So the predicted class for a classification problem can be defined as,

$$R(\vec{x}) = \arg\max_j y^j$$

On the other hand, in case of regression each tree gives a rational number as output and the final prediction for an input instance would be simply

$$R(\vec{x}) = \frac{1}{l} \sum_{i=1}^{l} v_i$$

where v_i is simply the value given by the tree i.

Gradient boosted trees. Unlike random forest learning algorithm, which attempts to reduce the variance of the model, the boosted trees reduce the biasedness. First of all, given the training dataset, each of the instances in the dataset gets a weight value of $1/N$, where N is the number of training instances. Then a base ('weak') learner is used to train on the dataset which would minimize the *loss* which defines a quantitative measurement as how many of such instances are wrongly predicted by the learned model. The idea of learning is to minimize the loss defined by using a function. There exist several loss functions, depending upon the problem at hand [HTF09]. Note that, in the case of boosted algorithms, the loss function considers the weight given to each training instance.

Initially, the learner is trained on the training dataset, the loss function computes the loss and this is used to update the weights given to the training instances. This is repeated a number of times (based on the number of base learners) and at each iteration, these weights are modified individually when a learner is trained. For example, at any step i, the training instances– misclassified by the learner G_{i-1} generated in the previous iteration –have their weights increased, and the correctly classified training instances are given lesser weights. The idea is to rectify the mistakes made by the previous learners and thereby collectively generate a strong model. At each step of boosting, such a learner is searched by using a gradient method in order to minimize the loss incurred by the previous learner. This process iterates over the number of learners (chosen by the user). In the end, collectively the learned models are said to have learned from the data, minimizing the loss and reducing the bias. Considering regression learning, the prediction for a given instance $\vec{x}$ is as follows:

[7] Note that a decision tree in its leaf nodes essentially gives the probability values corresponding to each class. Thus, amongst those probabilities, the one with the highest value is chosen as the corresponding class value of a leaf node in case of a decision tree model. For random forest, however, the probability values are taken for further computation.

$$G(\vec{x}) = \sum_{i=1}^{k} \alpha_i G_i(\vec{x})$$

Here α_i is the weight given to a learner i based on how much loss is incurred by it, i.e., more accurate model is given higher weight in order to increase its influence in the final prediction. The idea of boosting was first proposed by Schapire [Sch90]. Later the usage of the gradient method for searching a weak learner and the use of the tree as the weak learner was proposed by Breiman [Bre96]. In case of the gradient boosted trees, the learning algorithm works a bit differently. The derivation from the basic boosting algorithm to the gradient boosted tree is out of scope for this dissertation, for interested readers we refer to [HTF09]. Next, we describe the steps of the gradient boosted trees through an example.

Let us assume, we are given a set of training instances $D = \{d_1, \ldots, d_N\}$ where $d_i = (\vec{x}^i, y^i)$, is a data instance and its corresponding class. Initially, we calculate the average of all the class values, i.e., $y_a^1 = (y^1 + \ldots + y^N)/N$. Then we subtract each y^i from the y_a^1 to get a set of *residual* values, $\{y_r^{11}, \ldots, y_r^{1N}\}$. Now we have a set of instances in the form $D_r^1 = \{(\vec{x}^1, y_r^{11}), \ldots, (\vec{x}^N, y_r^{1N})\}$. Next a regression decision tree is learned on D_r, where the predicted value for a $\vec{x}^i$ is a residual value y_r^{1i}, i.e., a value denoting how different it from the actual prediction y^i is. Then the residual is used to calculate the actual prediction as $y_a^1 + \alpha * y_r^{1i}$, where α is learning rate, which can be chosen beforehand[8]. We again compute how this value is different from the actual prediction by subtracting y^i to it. We do this for all the instances and again get a set of residual values as $\{y_r^{21}, \ldots, y_r^{2N}\}$ thereby resulting a new training set for the second tree $D_r^2 = \{(\vec{x}^1, y_r^{21}), \ldots, (\vec{x}^N, y_r^{2N})\}$. This process is repeated to a fixed number of times and at each iteration the residual i.e., the loss decreases until it becomes a small value. So the idea is at each iteration the newly learned tree rectifies the mistake of the previous tree. The number of trees (i.e the iterations) is a hyper-parameter to this algorithm and can be chosen at the start of the algorithm. Formally for a given $\vec{x}$, the prediction of a gradient boosted regression tree can be defined as,

$$G(\vec{x}) = \sum_{i=1}^{k} G_i(\vec{x})$$

Here k is the number of decision trees chosen initially (hyper-parameter value). For classification, the learning algorithm constructs a single *strong* model for each class (also by iteratively learning weak decision trees). Suppose c is the number of classes, formally, the classifier G_c is a sequence of c GBT regressors, $G_c = \langle G_{r_1} \ldots G_{r_c} \rangle$. Each of the GBT regressors is furthermore consisting of let us say k decision trees (i.e., decided based on the hyper-parameter selection). That is a GBT regressor $G_{r_j} = \langle DT_1 \ldots DT_k \rangle$, where $1 \leq j \leq c$ For a given input instance $\vec{x}$, each of the regressors gives a real value and thereby we have c number of such values. The class is chosen with the maximal value, that is

$$G_c(\vec{x}) = \arg\max_j (G_{r_j}(\vec{x}))$$

[8] This is one of the *hyper-parameters* of the gradient boosted trees. There are detailed studies available how to choose this value [PVG+11].

Support Vector Machine (SVM). This learning algorithm attempts to find the best possible *decision boundary*, that represents the largest separation, or *margin*, between two groups of classes [9]. Such a decision boundary maximally separates the data instances into different classes. In its simplest form, this can be done by finding a line separating different classes, if the data instances are linearly separable. However, when the data instances are not linearly separable, SVM generates a *hyperplane* as the maximal separator between two sets of classes by transforming the input data into a higher-dimensional feature space.

Given a set of labeled data instances, SVM first employs a technique called the *kernel* method [CV95], which allows to implicitly transform the original feature space into a higher-dimensional space. Precisely, the kernel function calculates the similarity between pairs of data instances to perform such a transformation. After that, the SVM searches for the hyperplane that maximizes the *margin* between the nearest data points of different classes. The margin is defined as the distance between the hyperplane and the closest data points, known as *support vectors*. The SVM aims to find the hyperplane with the largest margin while also minimizing the loss. This is achieved by formulating an optimization problem that involves maximizing the margin and at the same time minimizing the loss occurred by the generated hyperplane.

Essentially, a hyperplane is represented by a set of *weights* assigned to each feature, along with an *offset* (also called a bias) term. For instance, for a binary classification problem, such a hyperplane takes the form: $w_0 * \vec{x}(0) + \ldots + w_n * \vec{x}(n) + b$, where w_i is the weight associated with the feature i and b is the offset term. At the end of the training phase of SVM, optimal values for w_is and b are obtained that maximize the support vectors between the classes while minimizing the losses. Thus, classification of a new data point $\vec{x}$ is done by evaluating the equation $w_0 * \vec{x}(0) + \ldots + w_n * \vec{x}(n) + b$. If the result is positive, the data point is assigned to one class; if negative, it is assigned to the other class.

Next, we highlight some important properties of several application areas of ML models.

2.1.2 Properties

Previously we discussed a range of application areas of machine learning models such as social, economic, transportation, medicine, law, etc. Now, there exists several properties concerning these application areas, which need to be ensured before the deployment of the ML models. As the model is generated through the training phase, we also need to ensure that the learning algorithm learns the 'correct' model in this phase. Primarily, the properties of ML-based software can be classified into two different categories: functional and non-functional [ZHML22].

Note that, in the typical notion of software testing, functional and non-functional properties correspond to different kinds of requirements compared to ML testing. For instance, functional properties correspond to what the software is supposed to do and how it should respond to different inputs. In other words, functional properties can be thought of as the specific requirements that outline the desired behavior of the software. To this extent, in ML testing, functional properties do relate to the functionalities of the ML model, such as properties like *accuracy* or *model relevance*. However, in the

[9] Note that for simplicity we consider the case with the Binary classification problem. However, this concept can easily be extended to multi-class classification problems.

case of non-functional properties which concern performance, reliability, and maintainability in traditional software testing, in ML testing this means something different. More specifically, in machine learning, non-functional properties relate to *fairness*, *robustness*, *monotonicity*, *security*, *interpretability* and more such properties. Since in this thesis, we always talk about ML testing, with the functional and non-functional properties we would always point to their respective definitions in ML testing.

Functional properties. These properties mainly concern the correctness and model relevance. The correctness of the model is defined as the model *accuracy*. More formally, suppose $\vec{x}^i$ is an unseen data instance (i.e., it has been not observed by the learning algorithm in the training phase) and y^i as the true prediction for it and $M(\vec{x}^i)$ is the prediction given by the learned model. In its simplest form, correctness for a classification model can be defined as,

$$\frac{1}{T}\sum_{i=1}^{T} L(M(\vec{x}^i), y^i)$$

Here, T is the number of test instances and $L(M(\vec{x}^i), y^i)$ is a binary function gives 1, when $M(\vec{x}^i) = y^i$ otherwise 0. However, for regression, this measure cannot be used since finding out whether two real values match exactly might be intractable. Instead, error metrics such mean squared error, mean absolute error or root mean squared error are used in practice.

So accuracy essentially measures the success rate of an ML model on a number of previously unseen data instances. However, the availability of such data instances are often scarce and therefore a more practical approach determines the accuracy of the learned model by splitting the training dataset into training and validation data. To this end, the learning algorithm gets trained on the training data and the validation set later determines how well the learned model performs on the previously unseen data. However, a potential drawback of this approach is the selection of the size of either of the set. On the one hand, If we select a large size of training data and a small number of validation data instances, the accuracy measure might not give a real estimate. On the other hand, setting aside a large number of validation data might give a badly trained model [AMMIL12]. A technique called the K fold cross-validation approach is proposed to tackle this issue. For this, the training instances are divided into $\lceil \frac{N}{K} \rceil$ number of groups. Now the training process iterates over different groups and at each training phase the learning algorithm gets trained on $\lceil \frac{N}{K} \rceil - 1$ groups and one group of K instances are used for validation. At the end of the iterations, we will have K number of accuracy results which is then averaged to get a final estimate [AMMIL12]. Training a model and measuring the accuracy to further improve the model is strongly co-related to each other and are mainly done in the training phase by the model developer.

Model relevance, closely related to the accuracy, measures whether the model is too complex for learning the dataset (*overfitting*) or does not generalise the training data well enough (*underfitting*) [VLL94]. Cross-validation, the same as before, can be used to detect those. However, it is essentially unclear, what an acceptable overfitting is. There exists a large body of work to tackle this problem, for example, by resampling the data [MLL+18], by generating *adversarial* data instances [WGS19], etc. We, however, in this work do not focus on the model relevance or correctness, rather we intend to find out whether the learning phase works correctly.

As we have mentioned before, due to the statistical nature of learning, it is often not clear what to expect as the outcome of the learning phase. We can of course measure the correctness of a model by observing how well it performs on unknown data instances, however, there does not exist a property defining what is expected to be learned from the training data. To this end, we introduce a property called *balanced data usage*, which essentially says, in the learning phase the learning algorithm learns what is in the training data. As the purpose of the learning is to learn from the given training instances, therefore the learning algorithm in its learning phase should not leave out any instances and consider each one of them in its learning phase. In the next chapter (Chapter 4), we formally define this property and give a testing approach to validate the property.

Non-functional properties. As mentioned beforehand, non-functional properties for machine learning models are specific requirements that correspond to properties like fairness, robustness, monotonicity, etc. [ZHML22]. In this dissertation, we primarily look at some of these non-functional properties of the ML models. To this end, we validate four different types of such properties on the classification models such as fairness, monotonicity, concept relationship, trojan attack, and a number of mathematical properties on the regression models.

To start with, there exist multiple definitions of the fairness property and they can be broadly categorized into two categories: *similarity-based measures* and *statistical* fairness. Since our approach cannot be used to check statistical fairness properties, in this thesis we only look at the similarity-based definitions. To give an example of such let us consider *individual discrimination*. This property requires the ML model to give similar predictions for similar individuals. For example, a loan granting software is discriminating against any individual with respect to gender if the change of gender from male to female and keeping other feature values the same, changes the prediction (individual discrimination). A weaker definition (fairness through awareness) does not strictly enforce equality on the non-protected feature values, rather it says if two instances are similar–which is defined using a distance measure–then the model prediction should also be the same. The statistical fairness definitions such as predictive parity, and equal opportunity [RT16] require to have equal probabilities to predict a class by the ML model for different *groups* of people in the society.

Similar to the fairness property, monotonicity is another non-functional property where the model predictions of two input instances are compared to find the violation of the property. This too has many variants. The basic idea is: if some feature values are increasing in the input instance, then the prediction of the ML model should also be increasing. For example, consider a loan granting model that is predicting the chance of giving a loan to any person. The chance of getting a loan should be higher for a person with a higher income compared to a person with a low income. This requirement occurs frequently in domains like credit scoring, house pricing, insurance premium determination, etc.

As machine learning models are frequently applied in the security domain, ML models need to be robust against vulnerability attacks. As an ML model is designed to give a prediction for a specific instance, it can be manipulated by the attacker to achieve a desired prediction. For example, consider a spam filtering case, where an ML model is used to detect any incoming mail for possible spam. In this case, if the attackers can find out which specific features are responsible for a mail being predicted as spam by the model, they can alter the values of these features, thereby fooling the model

into detecting the mail as a non-spam mail. This type of attack is called a trojan attack [LMA+18] where such a specific input pattern if present in the input data instance the attacker expects to yield a specific prediction.

The non-functional properties, that we have discussed so far, mainly concern single-label classification models. Our ML model testing approach can, however, also be used for multi-label classification models where we consider a specific application area known as knowledge graphs [DN21]. These graphs are being learned by the models to categorize entities according to given concepts that are fixed in an ontology. Ontologies not only describe concepts (like animal, dog, or cat), but also state the relationship between them (e.g. "every dog is an animal"). In such a setting, we get a multi-label classifier where each concept is a class label and the labels are binary (for e.g. an instance is a cat and an animal, but not a dog). To this end, we consider a property called the *concept relationship* which is basically defined as a Boolean expression over the class labels checking whether a multi-label classification model holds such relationships.

Apart from these properties, we also look at a number of numerical properties of the regression models and a specific type of numeric function called *aggregation* functions. We give the formal definition of these properties in Chapter 7.

Next, we describe some specific software testing techniques and later we describe how these can be used to test ML algorithms and models.

2.2 Software Testing

Based on [MSB11], "Software testing is the process of executing a program with the intention of finding errors." The errors are essentially unexpected behaviour or cases where the software fails to comply with specific requirements. Testing involves generating the test cases, and then executing the software on the generated test cases in order to find the violations of the requirements. The quality of these test cases generally depends on the *coverage* criteria, i.e., what percentage of the source code the test cases can cover. For example, one measure of such could be how many lines of code have been covered by a set of test cases. Test case generation technique that can cover most lines of code for most of the programs is deemed to be effective. Intuitively, if most lines of the program are covered, then the parts of the program which is *faulty*, must have been covered or has more chance to be covered by the test cases. As the effectiveness of the testing primarily depends upon the quality of the test cases, over the years a plethora of works have been proposed to this end. Instead of mentioning them individually here, we refer the readers to these works [Har07, ABC+13, BCR21, CS13].

Now, different test case generation techniques can be further categorized depending on the accessibility of the software under test (SUT), i.e., whether the internal structure or the implementation of the software is available to us. For this, there exist two most prevalent strategies: black-box and white-box testing. For example, test generation techniques like random or adaptive random testing, model-based testing, and metamorphic testing do not require to have the implementation of the SUT. However, the testing techniques like symbolic execution, grey or white-box fuzzing, concolic testing need the implementation of the software for generating test cases. As we consider the given learning algorithm and the ML model as a black-box in our thesis, we restrict our discussion to mainly black-box testing techniques here.

Algorithm 1 Test Generation for ART

Input: *testSet*, POOL_SIZE, MAX_SAMPLE
Output: set of test cases

```
1:  count := 0;
2:  while | testSet | < MAX_SAMPLES do                    ▷ extend start set
3:      cand := ∅; count :=0;
4:      while count < POOL_SIZE do                        ▷ generate candidates
5:          cand := cand ∪ {genRandom()}; count++;
6:      c_fur := oneOf(cand);
7:      maxDist := 0;
8:      for c ∈ cand do                  ▷ determine "furthest away" cand.
9:          dist := minDistance(c, testSet);
10:         if dist > maxDist then
11:             c_fur := c; maxDist := dist;
12:     testSet := testSet ∪{c_fur};
13: return testSet;
```

2.2.1 Adaptive Random Testing

We start with a simple testing technique that was essentially developed to encounter the weakness of simple random testing [CLM04]. In random testing, test cases are first generated uniformly at random and then those test cases are executed on the program under test in order to find an error[10]. Although, in some cases it has been shown to be surprisingly useful, often generating test cases blindly might not give a desirable coverage of the program and therefore can be ineffective in finding errors. The main goal of adaptive random testing (ART) is to bring *diversity* to the generated test cases. The process can be best described by Algorithm 1 [Wal18].

The algorithm starts with *testSet*, which is initially a randomly generated set of test cases and user given input parameters POOL_SIZE and MAX_SAMPLE. First of all, given a set of initial test cases, we generate a set of *candidate* test cases randomly as *cand* (Lines 4-5). Afterward, we take a new test case from *cand* which is the 'furthest away' from the set of initial test cases *testSet*. It starts at line 8, where we first take a test case c from *cand*, compute the minimal distance between c, and all the test cases from *testSet* and in the end, keep it in c_{fur}. In the second iteration, if the new test case from *cand* has a greater minimal distance than the previous one, we replace the furthest away candidate with the new c. This process is repeated for all the candidate test cases from *cand* and in the end, we select single a test case from this candidate set. We repeat this process until a maximum number of test cases are generated. Note that, there are other termination criteria available such as time out. However, most of the algorithms consider a limit on the number of test cases.

The distance function plays an important role in generating new test cases in ART. The default approach is to use a Euclidean distance metric [Wal18]. However, it cannot be applied in some cases. For example, if the input parameters are of different scales or the test cases are not numeric, the Euclidean distance measure is not applicable. To this end, there have been a number of works proposing several distance measures,

[10] We define error as the failure of the program to produce desirable behaviour.

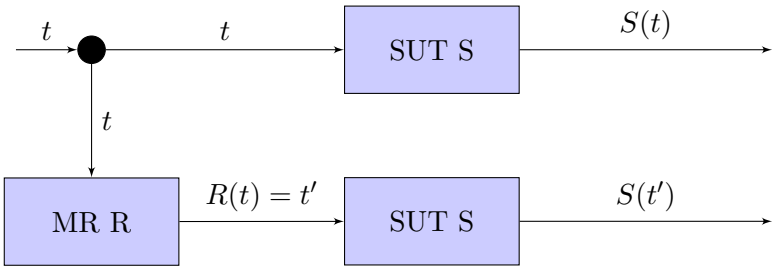


Figure 2.3: Workflow of metamorphic testing. Test input t is transformed based on a specific metamorphic relation R to t', and executing these two test inputs generate two outputs. Depending upon R, the relation between the outputs $S(t) \sim S(t')$ is determined.

depending on the problem at hand. In this work, we also propose such a distance metric [SW20a] based on the property we check. We use ART as one of the baseline approaches to compare against our testing approach and in Chapter 7, we give details of it. For a comprehensive survey of different distance metrics and the usage of ART, we refer to the works of Huang et al. [HSX+21].

2.2.2 Metamorphic Testing

Although software testing is frequently used to ensure quality by validating requirements, however, it often suffers from the *oracle* problem. The oracle problem refers to the scenario, where we do not know what would be the expected outcome for a test case. Generally, in software testing, first, we generate a sequence of test cases $T = \langle t_1, t_2, .., t_n \rangle$ and then we execute the software S on these test cases to find out whether the outputs $\langle S(t_1), S(t_2), ..S(t_n) \rangle$ match the expected outcomes $\langle f(t_1), f(t_2), ..f(t_n) \rangle$. Here, the function f (also can be called a target function) works as the oracle, deciding whether the output generated by the software S is as expected. If any of the test cases generate an unexpected behaviour such as $S(t_i) \neq f(t_i)$, we say that the software does not conform to the requirement. However, in many applications, there is no oracle f available, i.e., it cannot be defined what a correct outcome is supposed to be.

The metamorphic testing (MT) technique was introduced to alleviate this oracle problem in software testing. Although it was first intended as simply a test case generation technique (i.e., generating new test cases from a set of some existing randomly generated test cases), but soon it became obvious that this new testing technique could be used to tackle the oracle problem. When implementing metamorphic testing, first of all, a sequence of input instances as T are randomly generated and then through a specific *metamorphic-relation* R a follow-up input sequence $T' = \langle t'_1, t'_2, .., t'_n \rangle$ are generated where, $R : t \rightarrow t'$. Once we have the two sets of test cases T and T', both of these are then executed on the software under test to generate two different outputs which are then compared with reference to the metamorphic relation (MR) R (see Fig. 2.3).

For a given software, a metamorphic relation is a property of the target function. For example, to check the program implementing *sin* function, one MR could be to apply negation on the input, i,e, to check for an input x and $-x$, if $sin(-x) = -sin(x)$. If this relation does not hold, x is said to be a failed test case. We do not intend to summarize all about metamorphic testing and different metamorphic relations and all

of its application in various domains here (For interested readers, we refer to the works of Chen et al. [CKL+18]).

Metamorphic properties. Several non-functional properties of ML models which we have considered in this thesis are metamorphic in nature. For example, the individual discrimination property states: if two instances have the same values for all the features except for the protected attribute(s), then the output prediction should also be the same; is a metamorphic property as the original and the follow-up test cases have a specific MR and thereby the output predictions are expected to hold the equality relation. We consider more such metamorphic properties in our thesis. In Chapter 7 we give a detailed overview of the ones we validate. Note that, in software engineering literature such properties are also called *hyper-properties* [BF22] where two outputs corresponding to two inputs are needed to check a property. In an existing literature, these are called *k-specific properties* [CEH+22]. In this thesis, we, however, call them metamorphic and hyper-properties interchangeably.

2.2.3 Model-Based Testing

This is a specific type of testing technique where a model is generated (or learned) from the software under test (SUT) and then it is used to guide the test generation process. The model is essentially an abstract and partial representation of the SUT and can be of different types such as information, workflow, behavioural, etc. [Sch12]. Once the model is there, test cases are generated from the model in order to find a violation of the given specification. The test cases are of the same abstraction level as the model. Moreover, as these are generated without considering the structure of the SUT, this testing technique can be essentially used as black-box testing. There exist several ways to generate test cases from the model and that depends on two main factors: a) type of testing, for example, requirements testing, conformance testing, system testing, etc., and b) type of models, such as finite state machines or ML models. As in this work, we focus on testing whether a given requirement specification is violated by the SUT, we discuss here model-based testing (MBT) in the context of requirements testing.

Models are at the heart of this testing technique and there are several ways to learn them. However, the basic idea of learning remains somewhat the same. Model learning requires the existence of a *learner*, which interacts with both the learned model and the SUT in order to find out whether the model that is being learned, is accurate enough to capture the intended behaviour of the SUT [AMM+18]. First of all, a set of inputs are generated, which are given to the SUT, and for which we get a set of outputs. Thus, a set of input-output pairs are generated which is then used to generate an initial model. Further test inputs are generated, feeding them to the model and the SUT in order to find out whether their outputs match. If they agree on the inputs, we achieve equivalency (with some theoretical bounds), otherwise, failed cases are used to improve the learned model. This process is repeated until we do not find any more failed cases, or we reach a bound of generating the number of input cases. This learning framework is first proposed by Angluin in [Ang87] and is defined as *Minimally Adequate Teacher* (MAT).

Most of the existing works of learning a model evolve around learning finite state machines. Peled et al. [PVY99] first proposed the idea of combining Angulin's learning algorithm [Ang87] with the *model checking* [BK08] technique in order to perform black-box checking. Much like ours, they assume the SUT to be black-box, and only the

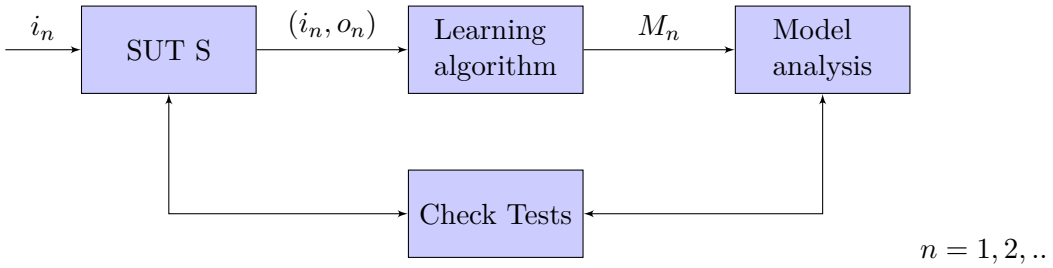


Figure 2.4: Workflow of model-based testing. At any time instance, given a set of input-output pairs (i_n, o_n), the learning algorithm learns to generate a model M_n. The model is then further analyzed to find violations, which are either returned in case they are valid, or used to improve the model.

input-output behaviour is known. To this end, first of all, they build a Büchi automaton from the *negation* of the given specification which is called a specification automaton $\bar{P}$. Let us consider, the automaton $\bar{P}$ accepts a set of words over a finite alphabet Σ. We denote this set as $L(\bar{P})$ and call it the *language* of $\bar{P}$. Note that, $L(\bar{P}) \subseteq \Sigma^*$. Then a sequence of automata $M_1, M_2, ..$ are generated based on the learning algorithm. Essentially, they start with a very trivial automaton and check the property on the automaton. For instance, suppose at the i-th iteration we have the generated automaton as M_i, and $L(M_i)$ is the language of the automaton. It is checked whether $L(M_i) \cap L(\bar{P}) \neq \emptyset$. If the intersection is not empty then a counter-example is provided and checked with the SUT. If it matches, then a violation has been detected, otherwise, this is used as a failed case of equivalence and furthermore used to improve M_i to get another automaton. There have been an extensive number of works done in this area in the past few years considering different types of FSMs such as Mealy machines [SG09], register automaton [CHJS16], labeled transition systems [HNS03], etc. However, in our context, learning an FSM is not suitable, as the ML models (SUT) we consider often perform complicated numerical operations. Moreover, the properties we consider cannot be easily described by any automaton. Hence, we use the idea of learning an ML model from a given ML model as the SUT and then analyze it to find a violation of the given property.

The idea of learning a machine learning model from a given software and then using the model to generate test cases is termed *learning-based testing* (LBT). This renders the idea of model-based testing and was first proposed by Meinke et al. [MN10]. Using the same basic idea of model learning described above, they first learn a piece-wise polynomial function from the SUT. Essentially, this function learns the functional behaviour of the SUT through input-output pairs (i_n, o_n), where i_n is the input given to the SUT and o_n is the corresponding output. Using a set of such pairs, the polynomial function is being learned, approximating the SUT. The function is then converted to a logical formula and conjoined to the first order logic expression of the negation of the property specification. Then the Hoon-Collins cylindric algebraic decomposition (CAD) [CJ12] algorithm is used for checking the satisfiability of this formula. If a counter-example (CEX) is found, then it is checked with the SUT. In case of a false positive, the model is retrained with all the previous input-output pairs along with this new counter-example. This process goes on until a CEX is found, or a timeout is reached.

Next, we describe the *property-based* testing technique, the idea of which in com-

bination with learning-based testing forms the basis of our property-driven testing concept.

2.2.4 Property-Based Testing

This testing technique was first introduced by Claessen et al. [CH00] in order to generate test cases for Haskell programs. The idea herein lies in the automation of the test case generation process. Previously, the test case generation process was dependent on the tester. Given a set of test cases, whether it fails or passes should have been checked each time by the tester. Also, the test case generation process was limited to a specific requirement, i.e., the test case generator has to be changed manually each time a new requirement needed to be checked. This was a cumbersome process and hindered the testing process, especially when a large number of requirements needed to be checked. The property-based testing approach can solve these two problems by proposing a domain-specific language that the testers can use to specify the requirement they want to check. First of all, the tester can specify the constraint on the test inputs to be generated as *pre-condition* as well as the expected output as *post-condition*. Afterward, a number of test cases would be generated following the pre-condition given, executed on the software under test, and then automatically checked whether the output of the software under test conforms to the requirement based on the post-condition. For generating test cases, Claessen et al. [CH00] proposed to use random generation as the default approach. In this thesis, we give a testing mechanism that allows the tester to specify requirements much like using the specification language of the property-based testing.

2.2.5 Property-Driven Testing

Here, we combine the idea of learning-based testing with property-based testing to develop a property-driven testing mechanism. More specifically, first of all, we generate (i.e., learn) a known ML model (for e.g., a decision tree) D from the given ML model under test. Here, the learning works in the same way as described before, i.e., generating a number of input-output pairs by executing the SUT and thereby generating a training dataset which is then trained on a learning algorithm for generating a model. The learned model D is then converted to a conjunctive normal formula (CNF) ϕ_D using a specific translation mechanism (described in Chapter 3). Consequently, the property specification (specified using a *pre-post* condition-based format) is negated and converted to a CNF formula $\phi_{\neg p}$. Then the formula $\phi_D \wedge \phi_{\neg p}$ is given to the satisfiability modulo theory (SMT) solver Z3 [MB08]. If the SMT solver finds a counter-example (i.e., violation) to the property, we first check it with the SUT. If the counter-example (CEX) is an invalid one, we use a sort of *incremental* SMT solving approach to generate a number of CEXs from the initial one and then retrain the learned model by appending the invalid CEXs to further improve the learned model. This process repeats until a user-defined timeout occurs or we find a counter-example violating the property on the SUT. We detail several parts of this technique in the subsequent chapters.

However, if a counter-example is not found then we could either stop the testing process which would be considered as the failure of the process, or we could start the entire process by generating a new model approximating the model under test and try again. If a number of trials still lead to no counter-example generation, then we stop.

2.3 Machine Learning Testing

Finally, in this section, we discuss some existing works of machine learning testing which use the testing strategies we have discussed so far.

Metamorphic testing. The idea of using metamorphic testing to test machine learning software was first introduced by Murphy et al. [MKA07]. They first used metamorphic testing to validate the implementation of two ML algorithms: support vector machine (SVM) and MartiRank. While the implementation of MartiRank did not reveal any error, SVM indeed produced some unexpected results. In order to find out whether the implementation of the SVM learning algorithm worked as expected, they considered *permutative* MR. They basically trained the learning algorithm twice–once with the original dataset and then with the permuted version of the dataset–thereby resulting in two ML models. However, these two models turned out to be different which was not expected, and the reason was the batch processing of the data by the optimization algorithm used in SVM. In later works [MKHW08, MSK09], they considered several other metamorphic relations (MRs), such as additive, multiplicative, and inclusive to check the implementation of several other classification algorithms such as Naive Bayes, K-nearest neighbors, and decision trees in WEKA [WFH11] library. In a more recent work, Dwarakanat et al. [DAS+18] introduced a specific MR for SVM classification algorithm with *non-linear kernel* and some MRs for a deep learning based image classifier ResNet [HZRS16]. In this thesis, we, however, define a property of the learning phase called balancedness, based on some specific metamorphic relations. The MRs we consider, are inspired by the work of Murphy et al. [MSK09]. However, in our case, they are used to define the balancedness property of the learning phase. We also give a metamorphic testing approach to check this property. In the next chapter (Chapter 4), we formally define this property and the MRs we consider and detail our testing mechanism.

There exists a number of works [DKH17, XLY+22, ZWG+21, ZS18, FWJ+22, JFL+22] which use metamorphic testing to test the ML models and these works mainly consider testing deep neural networks (DNN). For example, Zhang et al. [ZWG+21] gave a metamorphic testing framework for DNN based image classifiers where they defined a number of metamorphic relations based on the background-related changes such as blurring the background of the images. In [XLY+22], Xiao et al. proposed metamorphic testing for deep learning based compilers by using several semantic preserving MRs on the DNN model. In a recent work, Ji et al. [JFL+22] used MT to test a deep learning-based speech recognition system and for that, they considered several transformations on the speech input as metamorphic relations to find a violation. In this dissertation, we do not use metamorphic testing to test ML models, rather we check some metamorphic properties using the property-driven testing technique.

Model-based testing. There have been a number of works extracting automaton from ML models for the purpose of understanding or explaining the learned rules of the ML models [Jac05, OG96, AEG18], or to simply infer the automaton in order to facilitate the future analysis process [WGY18, OWSH20], or to check properties on the automaton [MVY20, KNR+21]. Here, we limit our discussion to the latter two types of works. The pioneering work of Giles et al. [OG96], first proposed the idea of extracting deterministic finite automaton (DFA) as a way to understand the rules of a recurrent neural network (RNN). They applied a clustering algorithm to extract several DFAs from the network. Then a heuristic method was applied to find the optimized

DFA model approximated from the RNN. The clustering approach of extracting DFA essentially works by finite partitioning of the state space of the RNN and dividing them into some k intervals. However, this technique suffers from state space explosion problems and is not suitable for modern day RNN models.

More recent work by Weiss et al. [WGY18] tackled this issue by using the Angluin learning paradigm by employing L* learning algorithm. Given an RNN R accepting a regular language L over some alphabet Σ, they, first of all, started with a hypothesis automaton H and through membership queries they checked whether H and the RNN R differs on any input, if so, then it was used to improve the automaton. Later Okudono et al. [OWSH20] extended this approach to learning a weighted finite automaton (WFA), as they argued, for probabilistic classification, learning *quantitative finite state machine* such as WFA, was more amenable compared to DFA.

In [MVY20], Yovine et al. proposed an *on-the-fly* property checking technique, where they learned the intersection of the given RNN and the negation of the property to be checked on it. If the intersection showed to be non empty, it was a correct counter-example with probability 1, otherwise they provided a probabilistic guarantee (i.e., provably approximate correct (PAC)), that the RNN satisfied the property. In a more recent work [KNR+21] Khmelnitsky et al. presented an approach by adapting the idea of Angluin's algorithm. They first of all, learned a hypothesis automaton H from the given RNN R and at the same time, they also generated specification automaton A. Given $L \subseteq \Sigma^*$ and $L(H)$ denoting the regular language of the automaton, they checked whether the automaton $L(H) \subseteq L(A)$, which amounts to checking whether the learned automaton model satisfies the property in hand. If they could find a counter-example, then they checked whether it was a valid one, if yes, return it as a violation otherwise use it to improve the automaton H. However, when there is no counter-example, they check whether $L(R) \subseteq L(H)$ where $L(R)$ regular language of the given RNN R.

As discussed above, using the idea of model-based testing to check the property on the given ML model mostly (if not all) (a) checks a specific type of neural network RNN, (b) considers the cases where the inputs can be deduced as regular language and (c) learn automaton as the model approximating the given ML model. Our approach, on the other hand, can be used for any ML model (not only RNN) and the 'language' of the model does not need to be regular in nature. Therefore, we resort to the idea of using an ML model to learn the given model under test. In the subsequent chapters, we give more details about our technique and we strongly believe at the end of this dissertation, the readers would be convinced about the need and the advantages of using our technique over the others.

3 Logical Encoding

In this chapter, we describe techniques to encode two machine learning models, namely decision trees and neural networks, into logical formulas. We require these logical encodings for two reasons. Firstly, while testing decision tree algorithms with respect to *balanced data usage* property (defined in Chapter 4), we need to perform *equivalence checking* to find out whether two decision tree models are *equivalent*. This requires encoding two decision tree models into logical formulas and then performing satisfiability checking using an appropriate solver. Secondly, we need to have the logical encodings of decision tree or neural network models (as *inferred* models) in our *property-driven* testing approach in order to generate test cases on a given model under test (MUT).

We start with the basic formulations of logics and theories and then give the encodings of two models and furthermore exemplify them. The content of this chapter is structured as follows.

1. logic and linear arithmetic theories (Section 3.1),
2. logical encoding of the decision tree and neural network along with the examples(Section 3.2),
3. computing property on the logically encoded model (Section 3.3).

3.1 Logic and Theories

Propositional logic. We begin with the propositional logic which provides the basis of our encoding approach. Typically, a propositional logical formula φ can be inductively defined over a set of Boolean variables using the grammar as follows.

$$\varphi ::= \text{true} \mid \text{false} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \Rightarrow \varphi \mid \varphi \Leftrightarrow \varphi \mid \neg\varphi$$

Essentially, the formula in its simplest form can either be true or false. It can furthermore also be composed of conjunction ($\wedge$), disjunction ($\vee$), or a mix of both of these operations. The negation is essentially a *unary* operator with the highest operator precedence, followed by conjunction and disjunction. The rest of the binary operators, implication ($\Rightarrow$) and bi-implication ($\Leftrightarrow$) can be furthermore expressed by using the aforementioned operations.

Example 1. As a simple example, consider the formula

$$\varphi \equiv (x \wedge y) \vee z$$

where we have a set of Boolean variables $v(\varphi) = \{x, y, z\}$ in the formula φ and it is composed of two operations, conjunction and disjunction. Note that we use the symbol $\equiv$ here to denote that we are defining the formula φ to be the formula on the right of $\equiv$.

Interpretation. If we consider $v(\varphi)$ to be the set of Boolean variables in the formula φ, then an interpretation I of the formula φ is defined as the mapping from the variables in the set $v(\varphi)$ to true or false [Cur63]. Therefore, essentially, we can write $I : v(\varphi) \rightarrow \{\text{true, false}\}$, and moreover for syntactic simplicity we write $I(\varphi)$ to denote the *evaluation* of the formula where we replace the variables in the formula φ with the interpretation I.

Example 2. For our example formula

$$\varphi \equiv (x \wedge y) \vee z$$

we have an interpretation I as,

$$I = \{x \mapsto \text{true}, y \mapsto \text{true}, z \mapsto \text{true}\}$$

When we apply I to the formula φ, we get

$$I(\varphi) = (\text{true} \wedge \text{true}) \vee \text{true}$$

which evaluates to true.

Note that, we do not give here the evaluation rules for propositional logical formulas, and assume the readers have the basic knowledge of such rules, however, if required, the readers can look here [Cur63].

Satisfiability checking. Satisfiability checking of a propositional logical formula φ involves finding an assignment of its Boolean variables $v(\varphi)$ such that the formula evaluates to true. To this end, if we essentially find an interpretation such that the formula φ is evaluated to be true, then we say I is a *logical model* of the formula, denoted as $I \models \varphi$.

Example 3. Consider our example formula $\varphi \equiv (x \wedge y) \vee z$. This formula is satisfiable and (one of) the model is, $I = \{x \mapsto \text{true}, y \mapsto \text{true}, z \mapsto \text{false}\}$.

Additionally, when no such interpretations exist such that the formula could be evaluated to be true, then the formula is said to be *unsatisfiable*.

Example 4. Consider the following formula

$$\varphi \equiv (x \vee y) \wedge \neg x \wedge \neg y$$

This formula is unsatisfiable since there does not exist an interpretation of the variables in the formula which would evaluate it to be true.

Conjunctive normal form. To find out the satisfiability and the corresponding logical model of a propositional logical formula, it needs to be given to a *satisfiability solver*. However, before the solver is applied to the logical formula, it needs to be converted into *conjunctive normal form* (CNF). This is required since the core algorithm behind the satisfiability solving technique (namely DPLL algorithm [DP60]) requires the formula to be in this form.

A formula φ is said to be in CNF, if it has the following form: $\varphi \equiv \varphi_1 \wedge \ldots \wedge \varphi_n$, where each of the subformula φ_i is called a *clause* and each clause further constitutes of a set of *literals*. Each literal forms the atomic part of the formula, for instance, a Boolean variable or its negation. Note that any propositional logical formula can be converted into an equivalent CNF, by using De Moragn law [Ros02]. However, using

this law to convert a formula into CNF might sometimes be highly inefficient, and therefore, a more common approach is to use Tseitin's transformation [Tse83]. This transformation produces a CNF formula of size linear in the size of the given non-CNF formula. In this chapter, we propose an encoding approach which generates the logical formula of the ML models in the CNF form, and therefore, exempt the need for using any further CNF conversion techniques.

Satisifiability modulo theories. The propositional logical formula, given in CNF, in its purest form, however, does not suffice in encoding the decision tree or neural network models. Since these models involve logical comparison and arithmetic operations between real-valued variables, we need to go beyond using propositional logical formulas which only constitute Boolean variables. To this end, we consider an extended version of the propositional logic defined using *theories*, and therefore, the satisfiability checking problem is now termed as satisfiability modulo theories (SMT) [1] solving problem.

There exist many theories to this end, for instance, *bit-vectors* (for modeling machine arithmetic), *arrays* (for encoding arrays), *strings* (for string values) and so on. In this thesis, we use *linear real arithmetic* (LRA) theory, which extends the propositional logic by considering all the rational number constants along with the set of arithmetic operations $\{+, -, *\}$ and logical comparators $\{>, <, \leq, \geq, \neq\}$. A formula φ in this case is said to be satisfiable if it evaluates to true for an interpretation I over the set of variables $v(\varphi)$ of the formula, such that $I : v(\varphi) \rightarrow \mathbb{R}$.

Example 5. Consider the following formula $\varphi \equiv (x - y \geq 0) \wedge (y > 5)$. This formula given in LRA is satisfiable and one of the model satisfying this formula would be: $I = \{x \mapsto 5, y \mapsto 5\}$. So, if we apply I to the formula we get: $(5 - 5 \geq 0) \wedge (5 > 5) \Leftrightarrow$ true $\wedge$ true $\Leftrightarrow$ true.

Note that, while evaluating the above LRA formula, the standard rules for the evaluation of arithmetic inequalities are applied, for instance, $0 \geq 0 \equiv$ true, or, $5 > 5 \equiv$ false.

Here, we do not discuss the SMT-solving algorithms since those are not relevant for describing the work done in this thesis (for interested readers we recommend to look here [BSST09]). Once, our encoding approach generates the logical formula describing the ML model (in SMTLIB format [2]), we can directly apply the SMT solver Z3 [MB08] to check the satisfiability of that formula.

Next, we give some formalization of machine learning models which are required to describe the approach of encoding the decision tree and neural network models into logical formulas.

3.2 Encoding

First, we revisit [3] the basic formalizations that are needed to describe the concepts of this Chapter.

[1] Note that, some recent works such as the works of Silva et al. [IIM22] or Ghosh et al. [GBM21] propose propositional logical encoding of decision tree which could be an interesting extension of our current approach.

[2] http://smtlib.cs.uiowa.edu/

[3] Note that, we only describe the formalization here which is needed to explain our testing concepts and further details can be found in Chapter 2.

We define a machine learning model to be a predictive function which takes the input from the set $X_1 \times \ldots \times X_n$ and predicts output(s) depending on the type of the *learning* considered. X_i is the value set of the feature i and we write $\vec{X}$ to denote $X_1 \times \ldots \times X_n$. Next we describe the three types of learning approaches that we consider in this chapter.

Single-label classification. In this type of learning the predictive model M takes the following form:

$$M : \vec{X} \to Y$$

In this case, given an instance $\vec{x} \in \vec{X}$, M returns a single class value $y \in Y$. The set of possible class values for this is the set of positive integers and thus, $Y \subseteq \mathbb{Z}^+$.

Multi-label classification. The predictive model in this case returns an output vector instead of a single value and can be defined as follows:

$$M : \vec{X} \to Y_1 \times \ldots \times Y_m$$

We define $\vec{Y}$ to denote $Y_1 \times \ldots \times Y_m$. If a single instance $\vec{x} \in \vec{X}$ is given to the model M, an output vector $\vec{y} \in \vec{Y}$ is predicted as the (set of) class values in this case. We use the label names to denote the variables corresponding to each of the class j, where $1 \leq j \leq m$, and let $L = \{L_1, \ldots, L_m\}$ be the set of such label names. In multi-label learning, we have the predicted class as a binary vector, and hence, $Y_i \subseteq \{0, 1\}$.

Regression. Finally, in case of regression learning, the model can be defined as follows:

$$M : \vec{X} \to \mathbb{R}$$

Thus, the predicted value in this case is simply a real-valued number.

In this work we require the logical encodings for decision trees and neural network models for two reasons, (a) computing the balanced data usage property of decision tree algorithms and, (b) analyzing either of the models which is learned as a white-box model in our property-driven testing approach. Now, for the latter case, based on the MUT, we require three sorts of encodings of the ML models (for both decision tree and neural network). If the given MUT is a single-label classification model, predicting discrete values as class labels, we learn a single-label classification model and in the case of a multi-label or regressor MUT, we learn a multi-label or a regression model. Therefore, we require the encodings for single- and multi-label classification and regression models of decision trees and neural networks.

The generic encodings for the classification and the regression cases, however, do not differ much except for the final evaluation of the class labels. For instance, in the case of the decision tree, the encodings of the branches remain the same except for the leaf nodes for these three cases. Similarly, in the case of the neural network, only the final layer of the network would require a different encoding. Hence, we do not describe the encodings of classification and regression in isolation for these three cases, rather, we give a unified encoding only differentiate when required.

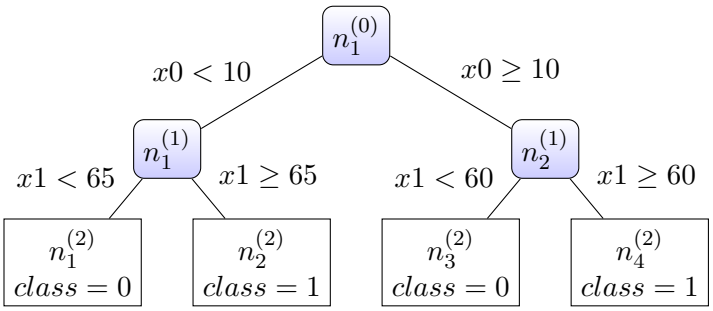


Figure 3.1: A decision tree of depth 1

3.2.1 Decision Tree

A decision tree model as described in Chapter 2 is a binary tree, in which every inner node (including the root node, but excluding the leaf nodes) represents Boolean conditions over the input feature values and every leaf node is labeled with a prediction giving the class values. An edge connecting a parent node to its child node is labeled with the result of the condition defined on the parent node and thus for each internal node, we have two edges connecting to its child nodes, when the condition evaluates to true and to false. For a given input $\vec{x} \in \vec{X}$ to the tree, a specific path of the tree is visited by satisfying the conditions on the edges before reaching to a leaf node. The class value corresponding to that leaf node is then returned as the prediction for $\vec{x}$.

For the ease of explanation, we consider the single-label decision tree classifier in Figure 3.1, which is adapted from the tree in Figure 2.1 of Chapter 2 . Here, instead of labelling a node with the Boolean condition we label each edge from the node with the corresponding condition (and with the negation of the condition). Thus, for each parent node connecting to its two child nodes, we have two edges, one labelled with condition and other with its negated version. For instance, the root node $n_1^{(0)}$ is connected with its two child nodes $n_1^{(1)}$ and $n_2^{(1)}$ via two edges which are labelled as $x0 < 10$ and $\neg(x0 < 10) \equiv (x0 \geq 10)$ respectively.

Next we give the encoding of the decision tree in two steps, first we describe the encoding of the internal nodes (considering the root node as a special case of an internal node) and then we describe the encoding for the leaf nodes. For the latter, we differentiate the encoding of the classification and regression trees.

We encode the decision tree as a logical formula by encoding every paths of the tree. We essentially do this by giving the encoding for each of the nodes in the tree. To this end, we use two Boolean variables for defining a node and the condition connecting the node with its parent node. The idea herein is to encode that a node variable becomes true only when its parent node is true and the condition on the edge holds. We repeat this for all the nodes of the tree and finally give the constraints corresponding to the predictions for each of the leaf nodes of the tree. The prediction corresponding to any leaf node is returned, only if the node variable corresponding to that node is true. This way we ensure that for a specific input to the tree only the nodes in a single path in the tree are visited, and the others are not taken. Note that, the encoding of the decision tree could be done by using some other methods (such as in [EGSS08]), however, we choose this approach since it reduces the size of the formula compared to the previous work and is linear in the size of the tree.

To this end, we first define for each level i in the tree $n_j^{(i)}$ be the j-th node (while

considering $n_0^{(i)}$ be the leftmost node at level i), and $n_{pre(j)}^{(i-1)}$ be its parent or predecessor node at the level $i-1$. The Boolean condition defined over an element of input $\vec{x}$ on the edge between $n_{pre(j)}^{(i-1)}$ and $n_j^{(i)}$ is written as $cond_{pre(j)}^{(i)}$ and the negation of the condition as $\neg cond_{pre(j)}^{(i)}$. We consider $n_j^{(i)}$ to be a Boolean variable denoting the node, which becomes true when the node is visited, otherwise, it is false. Using the node and the condition variables next we describe the logical constraints for the internal nodes.

Root node constraint. We begin with the constraint for the root node of the tree, which can be written as follows:

$$C_{root} \equiv n_1^{(0)}$$

This constraint is always true, since the traversal of the root node does not depend on any condition beforehand.

Example. Consider the example decision tree model depicted in Figure 3.1, the root node of the tree is $n_1^{(0)}$, the constraint for which can be written as:

$$C_{root} \equiv \text{true}$$

This constraint is always true and thus the Boolean variable $n_1^{(0)} = \text{true}$.

Internal node constraint. Next, for every internal node $n_j^{(i)}$ of the tree, we get one constraint as follows:

$$C_j^{(i)} \equiv (n_{pre(j)}^{(i-1)} \wedge cond_{pre(j)}^{(i)} \wedge n_j^{(i)}) \vee ((\neg n_{pre(j)}^{(i-1)} \vee \neg cond_{pre(j)}^{(i)}) \wedge \neg n_j^{(i)})$$

This constraint ensures that the node variable $n_j^{(i)}$ becomes true only when the predecessor node variable $n_{pre(j)}^{(i-1)}$ and the condition on the edge between these two nodes $cond_{pre(j)}^{(i)}$ are true. However, if either the $\neg n_{pre(j)}^{(i-1)}$ or the $\neg cond_{pre(j)}^{(i)}$ is true then $\neg n_j^{(i)}$ becomes true. In other words, if either the predecessor node [4] $n_{pre(j)}^{(i-1)}$ is false (while it is not visited) or the condition $cond_{pre(j)}^{(i-1)}$ is false, then the node $n_j^{(i)}$ becomes false and thus implies in either of these two cases the node $n_j^{(i)}$ will not be visited.

Example. We continue with our example tree in Figure 3.1 where for the inner node $n_1^{(1)}$ we get the constraint as:

$$C_1^{(1)} \equiv (\text{true} \wedge (x0 < 10) \wedge n_1^{(1)}) \vee ((\text{false} \vee \neg(x0 < 10)) \wedge \neg n_1^{(1)}).$$

Since the predecessor node of the $n_1^{(1)}$ is the root node, which is always true, and if the condition on the edge $x0 < 10$ holds, we can derive the node variable $n_1^{(1)}$ to be true. Otherwise, if the condition does not hold, then the $n_1^{(1)}$ would be false, implying that the node is not visited. Similarly, we get the constraint for the leaf node $n_1^{(2)}$ as follows:

$$C_1^{(2)} \equiv (n_1^{(1)} \wedge (x1 < 65) \wedge n_1^{(2)}) \vee ((n_1^{(1)} \vee \neg(x1 < 65)) \wedge \neg n_1^{(2)})$$

[4] While describing the encoding, we use the terms node and node variable interchangeably.

Thus, the leaf node variable $n_1^{(2)}$ can be further derived as true in case the node variable $n_1^{(1)}$ is true and the condition $x1 < 65$ holds.

Note that, these encodings for the node constraints do not depend on any learning problem and therefore remain the same for (single and multi-label) classification and regression learning.

Prediction constraint. Once we get the constraints for all the non-root nodes, next we derive the constraints describing the prediction of the classes by the leaf nodes of the tree. Since the predictions are different based on the different learning scenarios, we therefore describe these constraints for the three learning problems separately.

Single-label classification. The decision tree predicts a single class value in case of single-label classification which is essentially the value associated with the leaf node. We define a variable *class* to denote the class value corresponding to a leaf node. For every leaf node $n_j^{(i)}$ with the prediction $c \in Y$, along with the constraint for the leaf node, we furthermore get a constraint for the prediction of the class value as,

$$R_j^{(i)} \equiv n_j^{(i)} \Rightarrow (class = c).$$

Essentially, when the leaf node variable $n_j^{(i)}$ becomes true, then the corresponding class value of the leaf node is predicted.

Multi-label classification. For the multi-label classification, we require one Boolean variable $class_l$ for every label $l \in L$. Then, for every leaf node $n_j^{(i)}$, with the prediction $\wedge_{\ell \in L}(l = c)$ (where $c \in Y_l$), we get the following constraint for the class values:

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

Hence, whenever the $n_j^{(i)}$ becomes true, the final prediction, in this case, is a set of class values, described as a conjunction of L class values.

Regression. For the regression tree, we introduce a variable called *value* which denotes the prediction associated with the leaf node, a real-valued number. Now, for a leaf node $n_j^{(i)}$, we define the prediction associated to it as, $r_j^{(i)} \in \mathbb{R}$. The prediction constraint in this case can be defined as,

$$R_j^{(i)} \equiv n_j^{(i)} \Rightarrow (value = r_j^{(i)})$$

When the $n_j^{(i)}$ is true, the final prediction is the value associated to the leaf node [5], which is $r_j^{(i)} \in \mathbb{R}$.

Example. To exemplify the encoding given for the prediction of the decision tree, let us again consider the tree in Figure 3.1.

- As the tree depicted in Figure 3.1 is a single-label classification tree, a leaf node here is associated with a single class value only. For instance, for the leaf node

[5] Note that, for all these three cases we assume the decision tree always makes deterministic prediction, i.e., at a time only a single leaf is chosen.

$n_1^{(2)}$, we have the corresponding class value as $class = 0$. Thus, we get the constraint corresponding to the prediction of the node $n_1^{(2)}$ as follows:

$$R_1^{(2)} \equiv n_1^{(2)} \Rightarrow (class = 0)$$

If the node variable of the previous node $n_1^{(1)}$ becomes true and the condition on the edge $x1 < 65$ between the previous node $n_1^{(1)}$ and the current leaf node $(n_1^{(2)})$ becomes true, then the class 0 is predicted. For example, if the input feature vector to the tree is $(5, 12)$ as $(x0, x1)$, and if we feed this to the tree in Figure 3.1, then the condition $x0 < 10$ becomes true, and thereby, $n_1^{(1)}$ is derived to be true. Furthermore, $x1 < 65$ is evaluated to be also true and thus $n_1^{(2)}$ becomes true and the class value associated with the input vector $(5, 12)$ becomes 0.

- For the multi-label classification tree model, each leaf node is associated with a set of class labels, unlike the tree we consider in Figure 3.1. To give an example of such, let us assume the leaf nodes of the tree we consider have the following structure:

$n_1^{(2)}$	$n_2^{(2)}$	$n_3^{(2)}$	$n_4^{(2)}$
$class_{\ell 1} = 0$	$class_{\ell 1} = 0$	$class_{\ell 1} = 0$	$class_{\ell 1} = 1$
$class_{\ell 2} = 1$	$class_{\ell 2} = 1$	$class_{\ell 2} = 0$	$class_{\ell 2} = 1$

Here, we consider each leaf node is associated with two class labels $class_{\ell 1}$ and $class_{\ell 2}$. Now, since we consider the rest of the tree remains the same, the constraints for the inner nodes, along with the constraints of the leaf nodes (i.e., the node constraints) do not change. We, however, get different constraints only corresponding to the prediction of the class labels. For instance, for the leaf node $n_1^{(2)}$, we get the constraint for the class labels as follows:

$$R_1^{(2)} \equiv n_1^{(2)} \Rightarrow ((class_{\ell 1} = 0) \wedge (class_{\ell 2} = 1))$$

Hence, when the leaf node variable $n_1^{(2)}$ becomes true, then the predicted class labels are described as the conjunction of labels 1 and 2 which translates to the output vector $(0, 1)$. Similarly, we can derive the prediction constraints for the leaf nodes $n_2^{(2)}$, $n_3^{(2)}$ and $n_4^{(2)}$ as follows:

$$R_2^{(2)} \equiv n_2^{(2)} \Rightarrow ((class_{\ell 1} = 0) \wedge (class_{\ell 2} = 1))$$
$$R_3^{(2)} \equiv n_3^{(2)} \Rightarrow ((class_{\ell 1} = 0) \wedge (class_{\ell 2} = 0))$$
$$R_4^{(2)} \equiv n_4^{(2)} \Rightarrow ((class_{\ell 1} = 1) \wedge (class_{\ell 2} = 1))$$

- Finally, for the regression tree, the encoding of the leaf node is similar to the single-label classification, except the class value in this case is a real valued number. Assuming, the class value associated with the leaf node $n_1^{(2)}$ as 3.28, the constraint $R_1^{(2)}$ corresponding to the class value of the node can be described as:

$$R_1^{(2)} \equiv n_1^{(2)} \Rightarrow (class = 3.28).$$

After deriving the constraints for all the nodes of all the paths of the tree and the constraints for the predictions for all the leaf nodes we conjoin them to get the final logical formula describing the decision tree. Assuming we have a tree with depth d and at each depth we have n_a number of nodes, where $1 \leq n_a \leq 2^{(d-1)}$ we get the logical formula describing the tree as:

$$C_{tree} \equiv \bigwedge_{i=1}^{d} \bigwedge_{j=1}^{n_a} C_j^{(i)} \wedge R_j^{(i)}$$

For instance, in case of our example decision tree from Figure 3.1 we get the final formula of the tree [6] as:

$$C_{tree} \equiv C_1^{(1)} \wedge C_2^{(1)} \wedge C_1^{(2)} \wedge C_2^{(2)} \wedge C_3^{(2)} \wedge C_4^{(2)} \wedge R_2^{(2)} \wedge R_3^{(2)} \wedge R_4^{(2)}$$

For an input $\vec{x}$, if for instance, a single-label decision tree model predicts a class of c, then it can be proven that, following our encoding, the corresponding leaf node (denoted as n_c) variable would be true and thus we would get the prediction condition $n_c \Rightarrow c$ to be evaluated as true.

Next we describe the encoding of the feed-forward neural network model with ReLU as an activation function.

3.2.2 Neural Network

The second model we use in our property-driven testing approach as the approximated white-box model for the given MUT, is the neural network model. In the literature, there exist a long line of works that proposed SMT encodings for translating neural networks into logical formulas [GZW+19, FJ18, INM19]. The encoding approach we follow here is in a spirit similar to the approach proposed by Bastani et al. [BIL+16].

To start with, we assume to get a feed-forward neural network with ReLU (Rectified Linear Unit) activation functions modeling a predictive model M. Such a network can be described as a sequence of layers starting with an input layer, ending with an output layer, and might include a number of hidden layers in between. Each layer furthermore consists of a number of neurons that form the atomic parts of the neural networks. Each of these neurons consume a linear combination of the outputs of the neurons from the previous layer (see Chapter 2 for more explanation). The number of neurons in the input layer depends on the size of the input vector $\vec{x} \in \vec{X}$, and thus is $|\vec{x}|$. The number of output layer neurons varies depending on the type of the learning approach and moreover, the number of classes.

For instance, in the case of a multi-label learning, with the predictive function $M : \vec{X} \rightarrow \vec{Y}$, the number of neurons in the output layer would be $|\vec{Y}|$. For a single-label learning problem, as we have only a single class label, the number of output nodes is the cardinality of the set of the class values, i.e., $|Y|$. Since the prediction for the regression approach is simply a real-valued number, therefore, we only have a single neuron in the output layer. The number of hidden layer neurons is provided as part of the hyper-parameter setting during the learning process.

To exemplify the encoding of the neural networks, we consider the network model depicted in Figure 3.2 which we previously described in the Background (Chapter 2).

[6] For the entire encoding of this decision tree see the Figure 3.3 (see Lines 2-7)

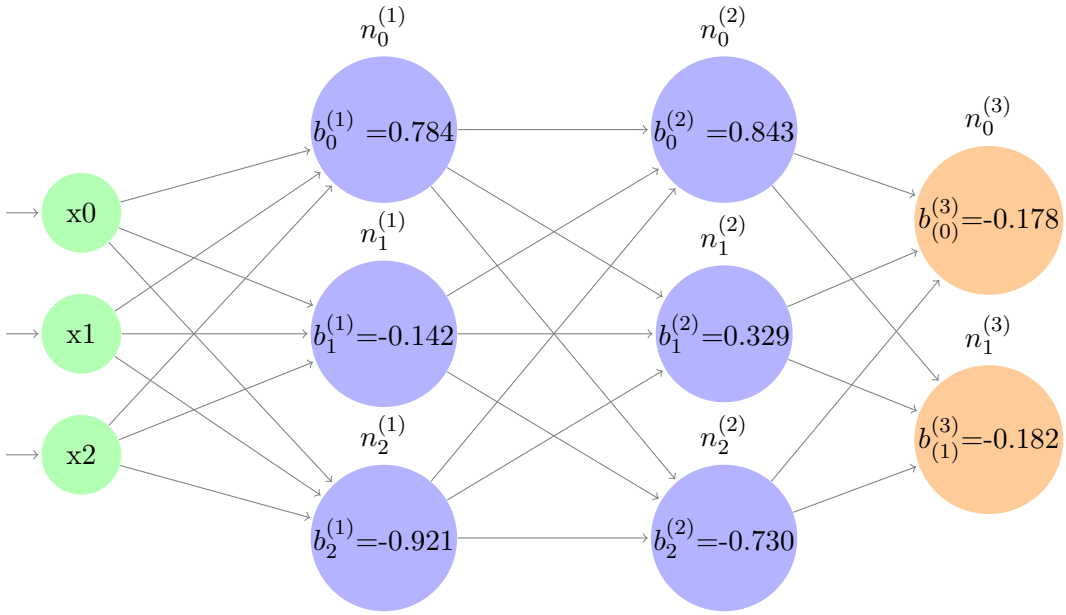


Figure 3.2: A feed-forward neural network

This network accepts inputs of size of 3 and thus it has 3 neurons at the input layer. The output layer contains 2 nodes, and we consider it to be a single label classification model with two classes, while explaining the encoding of the output layer for single-label network model. We consider the model to predict two output labels when defining the encoding of the output layer of the multi-label network model. Note that, the architecture of the neural network remains the same for a single-label classifier with c class values and a multi-label classifier with c different class labels. The only thing which is different for these two cases is the way the output layer is encoded.

Hidden layer encoding. We start with the encoding of the hidden layers of a network model. Suppose, we have $n = |\vec{X}|$ input nodes, and k number of hidden layers and in each layer we have n_i neurons, $1 \leq i \leq k$. Moreover, we set the number of input neurons n_0 to be n and n_{k+1} as the number of output neurons (which would vary depending on the learning problem). Since we consider a fully connected feed-forward neural network model, each neuron l in layer $i+1$ is connected via an edge to each of the neuron n_i of layer i. There is a *weight* associated with this edge connecting neuron j from layer i to the neuron l in layer $i+1$, and we denote it as $w_{jl}^{(i)}$. Each neuron j in layer i has also an associated *bias* term denoted as $b_j^{(i)}$.

For the logical encoding of this part, we furthermore introduce two real-valued variables $in_l^{(i)}$ and $out_l^{(i)}$ which indicate the input and the output of the l-th neuron on the i-th layer respectively. For every neuron l in a hidden layer i, we define two constraints, one for the input and the other for the output. The conjunction of all the input constraints of all the neurons of layer i forms the constraint $C_{in}^{(i)}$, describing the conditions on the inputs to the layer i, and similarly, $C_{out}^{(i)}$ describes the condition on the outputs of all the neurons of the layer i. These two constraints can be described as follows:

$$C_{in}^{(i)} \equiv \bigwedge_{l=1}^{n_i} (in_l^{(i)} = \Sigma_{j=1}^{n_{i-1}} w_{jl}^{(i-1)} out_j^{(i-1)} + b_l^{(i)})$$
$$C_{out}^{(i)} \equiv \bigwedge_{l=1}^{n_i} (in_l^{(i)} < 0 \wedge out_l^{(i)} = 0) \vee (in_l^{(i)} \geq 0 \wedge out_l^{(i)} = in_l^{(i)})$$

Essentially, the input constraint $C_{in}^{(i)}$ fixes the input of layer i as the weighted sum over all the outputs from the previous $(i-1)$ layer neurons, plus the bias terms. On the other hand, $C_{out}^{(i)}$ encodes the ReLU activation function as the output of the i-th layer, which essentially set the output as 0, if the input is a negative value, otherwise it sets the output as the value equal to the input. Note that, the input layer of the network model does not have ReLU, and therefore, for the output of the l-th neuron on the input layer, we add the constraints $out_l^0 = \vec{x}(l)$ where $\vec{x} = \vec{x}(1), \ldots, \vec{x}(n)$. So, the input to a neuron l on the first hidden layer $in_l^{(1)}$ is the weighted sum over all the x_is plus the bias term of the neuron l. Therefore, the constraint defining the input to the first hidden layer can be described as follows.

$$C_{in}^{(1)} \equiv \bigwedge_{l=1}^{n_1} (in_l^{(1)} = \Sigma_{j=1}^{n} w_{jl}^{(0)} x_j + b_l^{(1)})$$

Thus, we generate the logical constraints for k hidden layers and conjunct them all together to obtain the final constraint describing the hidden layers of the neural network model as follows.

$$C_{hidden} \equiv \bigwedge_{i=1}^{k} C_{in}^{(i)} \wedge C_{out}^{(i)}$$

Example. We use the example network model depicted in Figure 2.2 to exemplify the hidden layer encoding we described above. We begin with the first hidden layer of the network model and to this end, the constraints defining the inputs to the neurons $n_0^{(1)}$, $n_1^{(1)}$ and $n_2^{(1)}$ can be described as follows:

$$in_0^{(1)} = w_{00}^{(0)} * x0 + w_{10}^{(0)} * x1 + w_{20}^{(0)} * x2 + 0.784$$
$$in_1^{(1)} = w_{01}^{(0)} * x0 + w_{11}^{(0)} * x1 + w_{21}^{(0)} * x2 + 0.142$$
$$in_2^{(1)} = w_{02}^{(0)} * x0 + w_{12}^{(0)} * x1 + w_{22}^{(0)} * x2 + 0.921$$

The conjunction of all the input constraints $in_0^{(1)} \wedge in_1^{(1)} \wedge in_2^{(1)}$ form the input constraint $C_{in}^{(1)}$ of the first hidden layer, i.e.,

$$C_{in}^{(1)} \equiv in_0^{(1)} \wedge in_1^{(1)} \wedge in_2^{(1)}.$$

Next, the encoding of the activation function ReLU is performed by considering two cases for each neuron. For instance, for the neuron $n_0^{(1)}$, we get $(in_0^{(1)} < 0 \wedge out_0^{(1)} = 0)$ when the input to the neuron–as the linear combination of the previous layer neurons (in this case the input neurons) and the weights, plus the bias terms–is computed to

be a negative value. Similarly, we get the constraint $(in_0^{(1)} \geq 0 \wedge out_0^{(1)} = in_0^{(1)})$, when the input value is positive. Next, we perform the disjunction of these two constraints to get a single constraint defining the ReLU for neuron $n_0^{(1)}$. Similarly, we derive the ReLU constraints for the other neurons and thus, in the end we get the constraint defining the output of the first hidden layer as follows:

$$\begin{aligned} C_{out}^{(1)} \equiv ((in_0^{(1)} < 0 \wedge out_0^{(1)} = 0) \vee (in_0^{(1)} \geq 0 \wedge out_0^{(1)} = in_0^{(1)})) \\ \wedge((in_1^{(1)} < 0 \wedge out_1^{(1)} = 0) \vee (in_1^{(1)} \geq 0 \wedge out_1^{(1)} = in_1^{(1)})) \\ \wedge((in_2^{(1)} < 0 \wedge out_2^{(1)} = 0) \vee (in_2^{(1)} \geq 0 \wedge out_2^{(1)} = in_2^{(1)})) \end{aligned}$$

In a similar manner, we derive the input constraint for the second hidden layer as $C_{in}^{(2)}$ and the output constraint as $C_{out}^{(2)}$, which can be described as,

$$\begin{aligned} C_{in}^{(2)} \equiv (in_0^{(2)} = w_{00}^{(1)} * out_0^{(1)} + w_{10}^{(1)} * out_1^{(1)} + w_{20}^{(1)} * out_2^{(1)} + 0.843) \\ \wedge(in_1^{(2)} = w_{01}^{(1)} * out_0^{(1)} + w_{11}^{(1)} * out_1^{(1)} + w_{21}^{(1)} * out_2^{(1)} + 0.329) \\ \wedge(in_2^{(2)} = w_{02}^{(1)} * out_0^{(1)} + w_{12}^{(1)} * out_1^{(1)} + w_{22}^{(1)} * out_2^{(1)} - 0.730) \end{aligned}$$

$$\begin{aligned} C_{out}^{(2)} \equiv ((in_0^{(2)} < 0 \wedge out_0^{(2)} = 0) \quad \vee (in_0^{(2)} \geq 0 \wedge out_0^{(2)} = in_0^{(2)})) \\ \wedge((in_1^{(2)} < 0 \wedge out_1^{(2)} = 0) \vee (in_1^{(2)} \geq 0 \wedge out_1^{(2)} = in_1^{(2)})) \\ \wedge((in_2^{(2)} < 0 \wedge out_2^{(2)} = 0) \vee (in_2^{(2)} \geq 0 \wedge out_2^{(2)} = in_2^{(2)})) \end{aligned}$$

We furthermore conjunct all the above constraints to derive the hidden layer constraints for the network as,

$$C_{hidden} \equiv C_{in}^{(1)} \wedge C_{out}^{(1)} \wedge C_{in}^{(2)} \wedge C_{out}^{(2)}.$$

Output layer encoding. Same as the decision tree model, here we also describe the encoding of the output layer for three individual cases: single-label and multi-label classification and then regression learning.

Single-label classification. First of all, for the single-label classification, we define a variable *class* denoting the final prediction of the model. Here, the number of output neurons is equal to the number of class values, i.e., $|Y|$, and the output prediction is determined by considering the input values received by each of the output neurons. For instance, if the l-th output neuron receives the maximal input, the final prediction by the neural network would be l [7]. Now, to encode the output layer $k+1$, we first need the constraint $in_c^{(k)}$ for every $c \in |Y|$, or in other words, for every output neuron. This can be derived as $C_{in}^{(k+1)}$ using the encoding technique described beforehand and written as follows:

[7] Note that, typically the classes associated with the output neurons are determined in a top-down manner. Hence, the l-th class is associated with the l-th neuron from the top.

$$C_{in}^{(k+1)} \equiv \bigwedge_{l=1}^{n_{k+1}} (in_l^{(k+1)} = \Sigma_{j=1}^{n_k} w_{jl}^{(k)} out_j^{(k)} + b_l^{(k+1)})$$

Next, the output constraint for the output layer is defined by encoding *arg-max* function over the inputs of the output layer neurons which essentially encodes the technique for predicting class values by the single-label neural network. For each class c, the constraint can be defined as follows [8]:

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

This constraint encodes that if the input value received by the output neuron, corresponding to the class c ($in_c^{(k+1)}$) has a larger value than all the other input values received by all the other output neurons, then class c will be selected.

Note that, a common approach in the case of single-label classification of the neural network model, is to perform a sort of *normalization* on the output layer, for instance applying the *softmax* function. This function essentially converts a vector of N numbers to a probability distribution of N possible outcomes [GBC16]. However, such transformation does not alter the final prediction we compute by considering the output node with the maximum value.

Multi-label classification Now for the multi-label classifier, we define a Boolean variable $class_\ell$ for every label $\ell \in L$. Unlike the single-label classification approach, here the $class_\ell$ for a label ℓ, can be either 0 or 1, decided based on a threshold th (which is learned). More specifically, if the input $in_\ell^{(k+1)}$ for the l-th output neuron is greater than th, the prediction will be given as 1, otherwise 0. The constraint for the output layer thus can be described as follows:

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

Regression Finally, for regression learning the output layer consists of a single neuron and the output prediction out^{k+1} is simply the input to the neuron in^{k+1} and can be described as follows:

$$C_{out}^{(k+1)} \equiv out^{k+1} = (in^{(k)} = \Sigma_{j=1}^{n_k} w_{jl}^{(k)} out_j^k + b_l^{(k+1)})$$

Thus, the output constraint in this case simply fixes the output of the neural network model to be the input to the neuron of the output layer [9].

In this way, we get two constraints for the output layer and we define the output layer constraint for any learning approach as follows:

[8] For simplicity, the encoding described here assumes there are no ties.

[9] Note that, since the input constraint $C_{in}^{(k+1)}$ can be similarly obtained for the multi-label classification and regression learning as described for the single-label classification approach, we do not repeat it for the former two approaches.

$$C_{output} \equiv C_{in}^{(k+1)} \wedge C_{out}^{(k+1)}$$

Finally, the conjunction of the hidden layer constraint and the output layer constraint gives us the encoding describing the entire neural network model (C_{nn}) as follows:

$$C_{nn} \equiv C_{hidden} \wedge C_{output}$$

Example. We exemplify the encoding of the output layer considering the three learning cases separately using our example network model in Figure 2.2.

- First of all, considering the network performing single-label classification, we encode the output layer in such a way that the node with the maximum input value is chosen for the corresponding class value. To this end, we first encode the input constraint to the output layer as follows:

$$\begin{aligned} C_{in}^{(3)} \equiv (in_0^{(3)} = w_{00}^{(2)} * out_0^{(2)} + w_{10}^{(2)} * out_1^{(2)} + w_{20}^{(2)} * out_2^{(2)} - 0.178) \\ \wedge (in_1^{(3)} = w_{01}^{(2)} * out_0^{(2)} + w_{11}^{(2)} * out_1^{(2)} + w_{21}^{(2)} * out_2^{(2)} - 0.182) \end{aligned}$$

  For the two output neurons at the output layer, we need to derive two constraints for each of those, giving the prediction for the two classes, 0 and 1. Thus, the output constraints can be described as follows:

$$\begin{aligned} C_{out}^{(3)}(0) \equiv ((in_0^{(3)} \geq in_1^{(3)}) \wedge class = 0) \\ C_{out}^{(3)}(1) \equiv ((in_1^{(3)} \geq in_0^{(3)}) \wedge class = 1) \end{aligned}$$

  The conjunction of these two constraints form the output layer constraint as

$$C_{out}^{(3)} \equiv C_{out}^{(3)}(0) \wedge C_{out}^{(3)}(1).$$

  For example, assuming $in_1^{(3)} = 0.584$ and $in_0^{(3)} = 0.0$, we have the constraint $C_{out}^{(3)}(0) \equiv (\text{false} \wedge class = 0)$, and $C_{out}^{(3)}(1) \equiv (\text{true} \wedge class = 1)$, leading to the prediction of class 1.

- Consider our example network to be a multi-label classifier with two class labels. The input constraint of the output layer in this case remains the same as before, i.e., $C_{in}^{(3)}$. To encode the output constraints in this case, we require two Boolean variables $class_{\ell 1}$ and $class_{\ell 2}$, denoting two class labels corresponding to the output neurons $n_0^{(3)}$ and $n_1^{(3)}$, respectively. Essentially, when either of the output neurons receiving input from the previous layer has a larger or equal value than a threshold th, the prediction corresponding to that neuron would be considered as 1. Hence, in this case, we get two constraints for each of the output neurons and the conjunction of these two forms the final output constraint, which can be defined as follows:

$$\begin{aligned} C_{out}^{(3)} \equiv ((in_0^{(2)} \geq th \wedge class_{\ell 1} = 1) \vee (in_0^{(2)} < th \wedge class_{\ell 1} = 0)) \\ \wedge ((in_1^{(2)} \geq th \wedge class_{\ell 2} = 1) \vee (in_1^{(2)} < th \wedge class_{\ell 2} = 0)) \end{aligned}$$

For example, assuming $th = 0.5$ and $in_0^{(2)} = 0.76$ and $in_1^{(2)} = 0.32$, we have $((\text{true} \wedge class_{\ell 1} = 1) \vee (\ \text{false} \wedge class_{\ell 1} = 0)) \wedge ((\texttt{false} \wedge class_{\ell 2} = 1) \vee (\ \texttt{true} \wedge class_{\ell 2} = 0))$, which in turn evaluates to $(class_{\ell 1} = 1) \wedge (class_{\ell 2} = 0)$.

- Finally, assuming the example network to be a regression model, we have only a single neuron $n_0^{(3)}$ in the output layer. Thus, we get a single input constraint for the output neuron as,

$$C_{in}^{(3)} \equiv (in_0^{(3)} = w_{00}^{(2)} * out_0^{(2)} + w_{10}^{(2)} * out_1^{(2)} + w_{20}^{(2)} * out_2^{(2)} - 0.178).$$

The output constraint in this case simply fixes the prediction to be equal to the input to the neuron as $in_0^{(3)}$,

$$C_{out}^{(3)} \quad \equiv \quad out^3 = (in^{(3)} = w_{00}^{(2)} out_0^2 + w_{10}^{(2)} out_1^2 + w_{20}^{(2)} out_2^2 + b_0^{(3)})$$

Here out^3 essentially gives the output prediction as a real-valued number by the neural network model.

The conjunction of the input and the output constraints for the output layer, considering any learning approach gives us the output layer constraint for this network as follows:

$$C_{output} \equiv C_{in}^{(3)} \wedge C_{out}^{(3)}$$

Finally, the conjunction of the hidden layer constraint C_{hidden} and the output constraint C_{output} gives us the logical formula describing the entire network model [10].

Next, we describe how we use the logically encoded formula of an ML model to compute a specific property by using satisfiability solving technique.

3.3 Computation of Property

Once we have the logically encoded model, next we use it to compute a specific property on it. As mentioned before, we require the encoding of the two ML models for two different contributions of this thesis, for computing the *equivalency* between two decision tree models and computing a specific property on the approximated model (either decision tree or neural network), thereby testing the property on the given model under test. In both of these cases, we compute a specific property on the encoded model. While the property we check in the former work (i.e., testing balanced data usage) is the *equivalence* property, for the latter case, the property we check depends on the specification as specified by the tester. However, the technique to compute the property for both of these two cases remains the same and is performed by using SMT solving technique.

The *types* of properties we consider in this thesis are typically written in *pre-* and *post-condition* format. The pre-condition specifies constraints on the inputs of the model under test, and the post-condition specifies the constraints on the output(s). Let us assume, φ_{pre} and φ_{post} denote the logical constraints for pre- and post-conditions

[10] Note that the code implementing the models to logical formulas can be found in this link: https://github.com/arnabsharma91/model2logic

1 ; Encoding of the tree of Figure 3.1
2 $(\text{true} \wedge (x0 < 10) \wedge n_1^{(1)}) \vee ((\text{false} \vee \neg(x0 < 10)) \wedge \neg n_1^{(1)})$
3 $(n_1^{(1)} \wedge (x1 < 65) \wedge n_1^{(2)}) \vee ((\neg n_1^{(1)} \vee \neg(x1 < 65)) \wedge \neg n_1^{(2)}) \wedge (n_1^{(2)} \Rightarrow class_1 = 0)$
4 $(n_1^{(1)} \wedge (x1 \geq 65) \wedge n_2^{(2)}) \vee ((\neg n_1^{(1)} \vee \neg(x1 \geq 65)) \wedge \neg n_2^{(2)}) \wedge (n_2^{(2)} \Rightarrow class_1 = 1)$
5 $(\text{true} \wedge (x0 \geq 10) \wedge n_2^{(1)}) \vee ((\text{false} \vee \neg(x0 \geq 10)) \wedge \neg n_2^{(1)})$
6 $(n_2^{(1)} \wedge (x1 < 60) \wedge n_3^{(2)}) \vee ((\neg n_2^{(1)} \vee \neg(x1 < 60)) \wedge \neg n_3^{(2)}) \wedge (n_3^{(2)} \Rightarrow class_1 = 0)$
7 $(n_2^{(1)} \wedge (x1 \geq 60) \wedge n_4^{(2)}) \vee ((\neg n_2^{(1)} \vee \neg(x1 \geq 60)) \wedge \neg n_4^{(2)}) \wedge (n_4^{(2)} \Rightarrow class_1 = 1)$
8
9 ;Constraint describing the negation of the property
10 $((x0 = 0) \wedge (x1 = 0))$
11 $\neg(class = 0)$

Figure 3.3: Logical encoding of the property and the decision tree model

respectively, and φ_{model} denotes the logical formula describing the model under test. The property specified in this format (φ_{prop}) takes the following form:

$$\varphi_{prop} \equiv \varphi_{pre} \Rightarrow \varphi_{post}$$

Thus, the specification mandates if the pre-condition is satisfied then the post-condition must also be satisfied. Now, in computing a property using satisfiability solving approach, we essentially aim to find out whether we can find a case where the property is *violated*. For this, we take the negation of the property specification (denoted as $\varphi_{\neg prop}$) which can be derived as follows:

$$\varphi_{\neg prop} \equiv \neg(\varphi_{pre} \Rightarrow \varphi_{post}) \equiv \neg(\neg\varphi_{pre} \vee \varphi_{post}) \equiv \varphi_{pre} \wedge \neg\varphi_{post}$$

In summary, we aim to find out given the pre-conditions are evaluated to be true, if there exists a case where the post-condition is false or the negation of the post-condition is true. Hence, to compute this on the ML model, we generate a formula of the following form:

$$\varphi \equiv \varphi_{model} \wedge \varphi_{pre} \wedge \neg\varphi_{post}$$

Essentially, given this formula to the solver, it will attempt to find a satisfiable example where φ is evaluated to be true. For this, each part of the formula needs to be true, since they are logically conjoined. Given this formula to an SMT solver, it would then return a logical model as the satisfiable example of this formula (if exists), where the $\neg\varphi_{pre}$, φ_{post}, and φ_{model}, are true. This would then imply that an input to the model is found satisfying the pre-condition, the output corresponding to which fails to satisfy the post-condition, thus violating the property. On the other hand, if the solver does not find any satisfiable example to this formula, it would then imply that there does not exist any interpretation of the formula for which the $\neg\varphi_{post}$ is true [11] and thus the model satisfies the property.

Next, we give a simple example to illustrate how the computation of a property on the logical formula of an ML model is performed using the satisfiability solving technique.

For this, we take a simple property that specifies if the elements of the input instance

[11] Note that, the model formula φ_{model} is always true.

are all zeros, then the prediction should also be zero [12]. More formally:

$$\forall \vec{x} \in \vec{X} : \forall i \in \{1, \ldots, n\} : \vec{x}(i) = 0 \Rightarrow M(\vec{x}) = 0$$

To compute this property, let us take the example decision tree depicted in Figure 3.1 (page 41). Based on the inputs $x0$ and $x1$ of the tree, we can write the logical constraint describing the property for this tree as,

$$\varphi_{prop} \equiv ((x0 = 0) \wedge (x1 = 0)) \Rightarrow (class = 0)$$

The property essentially contains two parts, the constraint $((x0 = 0) \wedge (x1 = 0))$ specifies the pre-condition, and the constraint $(class = 0)$ specifies the post-condition. Thus, the negation of the property can be written as,

$$\varphi_{\neg prop} \equiv ((x0 = 0) \wedge (x1 = 0)) \wedge \neg(class = 0)$$

Figure 3.3 shows the encoding of the entire formula (tree model and the property with the negated post-condition) we use to compute the property on the model. Note that, for simplicity, we do not write the conjunction between the lines and implicitly assume that all the constraints in different lines are conjoined. When this formula is given to the satisfiability modulo theory (SMT) solver, it would then attempt to find a satisfiable example which in this case cannot be found. This would then imply, since we cannot find any satisfiable example to the formula with the negation of the property [13], the model satisfies the property. In other words, in this case, we cannot find an interpretation of the variables in this formula, such that the formula can be evaluated to be true and thus we prove the satisfaction of the property on the model.

Now, supposedly consider, we want to check a property as, *if* $x0 > 20$ *and* $x1 < 50$ *then the prediction is* $class = 1$, which can be formulated as,

$$((x0 > 20) \wedge (x1 < 50)) \Rightarrow (class = 1)$$

Note that, to compute this property on the model in order to find a violation of the property, we take the negation of the post-condition, and thus, the negation of the property is written as,

$$((x0 > 20) \wedge (x1 < 50)) \wedge \neg(class = 1)$$

After connecting the above constraint to the logical formula of the tree of Figure 3.1, and then applying an SMT solver on the entire formula, we could find a logical model as the satisfiable example, for instance as, $\{x0 \mapsto 21, x1 \mapsto 49, class \mapsto 0\}$. Hence, this property is said to be violated on the decision tree model and the logical model returned by the solver would be then considered as a *counter-example* to the property.

Note that, this property computation method renders the idea of *constraint-based verification* technique, where this is used to formally prove that a software program meets its specified requirements. It involves the use of logic and reasoning to verify the correctness of a program [HH19]. This technique is further used to verify *robustness*

[12] Note that, this property is termed as infimum property and in our thesis we later show the evaluation of this property on the learned regression models.

[13] For simplicity, we write the 'negation of the property' to denote the property with the negated post-condition.

properties of the deep neural network models, for instance, in the works of Ehlers et al. [Ehl17], Katz et al. [KBD+17, KHI+19]. However, in contrast to these approaches we do not use this technique to verify robustness property on the given model. Rather, we use this to verify a (specified) property on the inferred model, thereby testing the property on the MUT. Moreover, we use this technique in checking equivalence property of two decision tree models while testing balancedness property of the algorithm, which none of the previous works have considered.

To conclude, in this chapter, we gave the foundation of logic and theories and described how we encode two machine learning models into logical formulas through examples. Furthermore, we have shown the property computation technique on the learned model by using the satisifiability checking approach. In the subsequent chapters, we use these techniques in the testing of the balanced data usage property of the decision tree algorithm (Chapter 4) and in describing the test data generation technique for our property-driven testing approach.

4 Balancedness Testing of ML Algorithms

The principle idea of this thesis is to develop testing mechanisms for both the learning and prediction phases of the machine learning pipeline. To achieve this goal, in this chapter, we start with the testing of learning algorithms (or, in short learners). A machine learning algorithm in its learning phase generalizes from a given training dataset to generate a predictive function or model to which if an input instance is given, an output is predicted. However, it is essentially unclear, whether the generated model is the *expected one*, since there does not exist an *oracle* to define a so called *correct* outcome of the learning phase. In other words, a ground truth in terms of a *target* function to compare the generated predictive function or the model against is mostly missing.

The missing oracle problem is not new in software testing and it can be found in many other types of programs, for instance in compilers, search engines, and programs implementing mathematical functions, where characterizing specific requirements on the output is not possible. These types of programs are referred to as *non-testable* or *untestable* programs, terms coined by Weyuker in her work on discussing this issue in software testing [Wey82]. To tackle the oracle problem, researchers first came up with the idea of implementing several versions of a program and comparing their outputs to detect discrepancies, which is termed N-way testing [MK01]. However, implementing several versions of a large program is a complicated task. Therefore, Chen et al. [CLM04] proposed the metamorphic testing technique to alleviate the missing oracle problem. The basic idea of the metamorphic testing approach is to apply a specific *transformation* (also called a *metamorphic relation*) on an initial set of (randomly generated) inputs and then check whether the outputs generated before and after applying the transformations, by executing both the inputs on the programs under test, obey the desired relationship. For example, a program that outputs the shortest distance between a source and a distance point should be producing the same result even when we change the source to destination and destination to source. The metamorphic relations to consider in this testing approach, therefore, depend on the domain and the problem at hand.

Now, a program implementing a machine learning algorithm can also be categorized as a non-testable program since it cannot be determined whether the output (which in this case is a statistical model) is the correct one. Thus, we need a definition for the correct outcome of the learning phase. To solve this problem, we first define a property called the *balanced data usage* or in short *balancedness* of the learning algorithms. We say, as the sole purpose of the learning algorithm is to learn from the training instances in its learning phase, the algorithm should be learning from the entire dataset. In other words, the learning algorithm should not keep out any of the instances or features and *treat all the features and instances of the training dataset in the same way.* We aim to determine whether the learner is learning what is in the training data or by design is not considering some specific instances or features in the dataset. This would then essentially imply that the learning algorithm is not learning from the dataset the way

it is expected.

Precisely, we use the idea of metamorphic testing to define the balancedness property of learning algorithms. This is done by formally defining balancedness by considering some specific metamorphic transformations (MT) on the training data. More specifically, we define the property with the use of three *domain independent* MTs on the training data, and constrain for each of the transformations applied on the training data, the learning algorithm should generate the same model, before and after applying the transformations. To test this property, we furthermore provide a testing approach, implemented in a tool named TiLe.

Now, one significant challenge in this testing approach is to check the equality of the two models generated before and after applying a specific metamorphic transformation, or, in other words, finding out the *equivalency* between two models. We employ two techniques for it: (a) *computing* and (b) testing the equivalency between two models. There are pros and cons to both of these techniques. However, in combination, they constitute an effective equivalency checking mechanism.

We start by formally defining the balanced data usage property in Section 4.1. Then we present our testing framework using the idea of metamorphic testing in Section 4.2. We have implemented our testing approach in a tool called TiLe (details of which are given in Appendix A.1). Then we evaluated TiLe on a number of machine learning algorithms which we describe in Sections 4.3 and 4.3.2. Finally, we discuss the related works in Section 4.5 to end the chapter.

4.1 Balanced Data Usage

We start by introducing some basic terminologies in machine learning and then formally define the balanced data usage property.

A supervised learning algorithm gets a training dataset in its learning phase and *learns* by generalizing the dataset to generate a predictive model (or in short model). This model can be defined as follows:

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

Here X_i denotes the value set of feature i (also called attribute or characteristic i), and $1 \leq i \leq n$. Y denotes the set of classes[1] We furthermore use $\vec{X}$ to write $X_1 \times \ldots \times X_n$. Note that, depending on the *type* of the feature values X_i, the feature i can be categorical (string value), or numerical (integer or real values) [2].

After the learning, in the prediction phase, the learned model gets a data instance $(x_1, \ldots, x_n) \in \vec{X}$, to which the model M assigns a class $y \in Y$ to it. We assume $\mathcal{M}$ to be the set of all such models which are generated on arbitrary value sets and classes. Let M_1 and M_2 are two models where $M_1, M_2 \in \mathcal{M}$ and we can define their equivalency as follows:

Definition 4.1 *Any two models M_1 and M_2, are said to be* equivalent*, denoted as $M_1 \equiv M_2$, if for any data instance $x = (x_1, \ldots, x_n) \in \vec{X}$, we have $M_1(x) = M_2(x)$.*

[1] In this chapter, we only consider checking the single-label ML algorithms and therefore, we only give here the formalization corresponding to that.

[2] There are other types of features such as text or array which constitute of these two basic feature types.

Table 4.1: Example banking data set

No.	*income*	*age*	*gender*	*class*
1	1000.0	35	male	1
2	800.0	40	male	0
3	1200.0	53	female	1
4	900.0	30	female	0
5	800.0	38	female	0

To learn a predictive model, the learning algorithm in its learning phase gets a training dataset which contains a set of *training instances* T, where $T \in \vec{X} \times Y$. Suppose, the set of training instances consists of m number of instances and each of these instances are vector of size n, hence, $T = ((x_1^1, \ldots x_n^1), y^1), \ldots ((x_1^m, \ldots x_n^m), y^m)$. Along with the set of training instances, the training dataset also contains a list of feature names, $F = (f_1, \ldots, f_n)$. In this thesis, we only consider a *tabular* representation of the dataset which is essentially a two dimensional matrix, containing rows as data instances and columns as feature values. The first row of the dataset defines the feature names, and the last column gives the class values for each of the instances. Assuming $\mathcal{F}$ to be the set of all feature name lists and $\mathcal{T}$ be the set of all training data, we define the learner (i.e., the learning algorithm) as a function of the following form:

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

The number of features or the size of the feature names list fits the number of columns in the training dataset.

The learner in its learning phase gets a training dataset $(F, T) \in \mathcal{F} \times \mathcal{T}$, and then generates a predictive model as M. Table 4.1 shows the example of a small training dataset (inspired by one of our previous work [SW20b]). Here, we have three features $F = (income, age, gender)$, amongst which *income* and *age* are numerical and *gender* is categorical (i.e., containing string values). Now, for each of these features, we have five feature values, giving five training instances and for each of these instances, we get class values given by the last column (named as *class*).

Balanced data usage or balancedness property requires the learning algorithm to use the training data entirely and not keep out any of them. For instance, in case of our example dataset when presented as training data to a learner, it should use all these five instances and three feature values in the same way to learn a model. In other words, the position of the instances and the features or their names and values should not affect the learning process in any way. More formally, we want the learning algorithm to be *invariant* in terms of generating a model when specific transformations are applied to the training dataset. This matches with the typical idea of metamorphic testing, where an input and its transformed version are given to a system under test, and the two outputs are then checked for a specific relation. The transformations that we consider here are thus metamorphic transformations and we furthermore expect that the two models learned from these two datasets (i.e., original and transformed) are equivalent (based on the definition of equivalency in Definition 4.1). We use the term *transformation* instead of *relation* in the rest of the chapter, as we use them to transform the original training data into a new one.

To this end, we consider four metamorphic transformations: permuting the rows

(instances), columns (feature values), and feature names, and *renaming* the feature values. Since our transformations are permutative in nature we first of all define permutation through a bijective function $\pi : \mathbb{N}_l \to \mathbb{N}_l$, where $\mathbb{N}_l = \{1, \ldots, l\}$. The set of all such permutations with l elements can be defined as S_l, thus, $|S_l| = l!$. Based on this, next we define the metamorphic transformations.

Permutation of instances. Given a set of training instances, this metamorphic transformation changes the ordering of the instances, i.e., it gives a new ordering of the instances or rows of the training set. More formally, assuming we have a training dataset as $T = ((x_1^1, \ldots x_n^1), y^1), \ldots ((x_1^m, \ldots x_n^m), y^m)$ containing m instances. For a permutation function $\pi \in S_m$ applied on the training set T we get the transformed dataset as, $\pi(T) = ((x_1^{\pi(1)}, \ldots x_n^{\pi(1)}), y^{\pi(1)}), \ldots ((x_1^{\pi(m)}, \ldots x_n^{\pi(m)}), y^{\pi(m)})$.

Example. We now give an example of applying the instance permutation operation on the dataset presented in Table 4.1. For this dataset T=((1000.0, 35, male), yes), ((800.0, 40, male), no), ((1200.0, 53, female), yes), ((900.0, 30, female), no), ((800.0, 38, female), no) containing 5 instances, if we apply a permutation function $\pi \in S_5$, we get a permutation as, $\pi(T)$ =((800.0, 40, male), no), ((1200.0, 53, female), yes), ((1000.0, 35, male), yes), ((800.0, 38, female), no), ((900.0, 30, female), no). Here the first row is moved to the third position. The second and the third rows become the first and the second rows respectively.

Permutation of features. This transformation when applied on the training dataset changes the order of the columns. However, this is only applied to the set of columns containing feature values and exempts the column containing the class values. For a given dataset T, containing n features and $(n + 1)$th column denoting the class values, a permutation function $\pi \in S_n$ when applied on T, gives $((x_{\pi(1)}^1, \ldots x_{\pi(n)}^1), y^1), \ldots ((x_{\pi(1)}^m, \ldots x_{\pi(n)}^m), y^m)$.

Example. For our example dataset, when the feature permutation is applied as $\pi \in S_3$, we get the transformed dataset as $\pi(T)$ =((40, 800.0, male), no), ((53, 1200.0, female), yes), ((35, 1000.0, male), yes), ((38, 800.0, female), no), ((30, 900.0, female), no). The first and the second columns here have moved their positions to second and first columns respectively.

Shuffling the feature names. The shuffling of the feature names simply permutes the feature names of the dataset. We take a permutation function $\pi \in S_n$ and apply it to the list of feature names $F = (f_1, \ldots, f_n)$, and get, $\pi(F) = (f_{\pi(1)}, \ldots f_{\pi(n)})$. We do not consider the class as a feature name and this transformation is only applied to the feature names and not to the data instances.

Example. In this case, the rows and the columns of the dataset remain the same with only exception being the column names or feature names. Hence, for our running example dataset, we get a new list of feature names as for example, $\pi(F) = (gender, age, income)$.

Renaming the feature values. In this transformation, we convert the categorical features to numerical ones. More specifically, if a feature contains categorical values, we replace them with numerical values by using fixed mapping. We assume, the domain of such a categorical feature X_i to be finite, $|X_i| = d_i$, where $1 \leq i \leq n$.

The renaming function can be defined as a bijective function, $num_i : X_i \rightarrow \mathbb{N}_{d_i}$, and $\mathbb{N}_{d_i} = \{1, \ldots, d_i\}$. The resulted training set after applying this transformation is, $((num_1(x_1^1), \ldots num_k(x_n^1)), y^1), \ldots (num_1(x_1^m), \ldots num_k(x_k^m)), y^m)$.

Example. In our example banking dataset we have two numerical features, *income* and *age*, and one categorical feature *gender*. Hence, we rename the feature values of our example dataset by considering *male* mapping to 0 and *female* to 1.

We define a set P containing the three permutation transformations–permutation of features, permutation of instances, and shuffling of feature names, and we use p to denote an element of it, i.e., $p \in P$. We further define $Types = \{p_r, p_c, s_f, r_f\}$ to be the set of transformation *types* we consider. For this, we denote row permutation as p_r, column permutation as p_c, feature name shuffling as s_f and renaming of feature values as r_f. We furthermore let $type(p) \in Types$. We assume $(F, T) \in \mathcal{F} \times \mathcal{T}$ and a metamorphic transformation $p \in P$ when applied to (F, T) gives a transformed training data as $p(F, T)$. Finally, we are ready to formally define the balancedness definition which can be defined as follows:

Definition 4.2 *An ML algorithm* $learn : \mathcal{F} \times \mathcal{T} \rightarrow M$ *is said to be* balanced*, if for any feature vector and training data* $(F, T) \in \mathcal{F} \times \mathcal{T}$*, and for all the metamorphic transformations* $p \in P$*,* $learn(F, T) \equiv learn(p(F, T))$*.*

The metamorphic transformations we consider in balancedness or balanced data usage are *domain-independent*, since they are not considered based on the specifics of any learning algorithms. For instance, permuting the rows or columns, or shuffling the feature names should not have any effect on the learning, as any type of learning algorithm does not consider the position of the instances or columns and the feature names as part of its learning process. Therefore, we expect, all the ML algorithms to be invariant under these transformations. If a learning algorithm generates two different models before and after applying any of these transformations on the training dataset, we say that the algorithm is *sensitive* to that transformation and hence, is not balanced [3].

The idea is, although, for some machine learning experts, this is known that applying the metamorphic transformations on the training dataset we consider here, might change the generated models for some algorithms, due to the specifics of the implementations of those algorithms, an ordinary software developer with no knowledge of machine learning would not expect that. For instance, Pham et al. [PQW+20] found that the changes in a generated model due to the specific implementations of an ML algorithm were even unknown to machine learning practitioners both in industries and academia. Thus, we believe the study of testing the balancedness of ML algorithms would help to understand the extent to which ML algorithms generate different models when changes are applied to the training data.

Next, we describe the testing approach for checking the balanced data usage property.

[3] We do not consider renaming of feature values transformation as the part of the balanced data usage property, see Section 4.3 for more explanation.

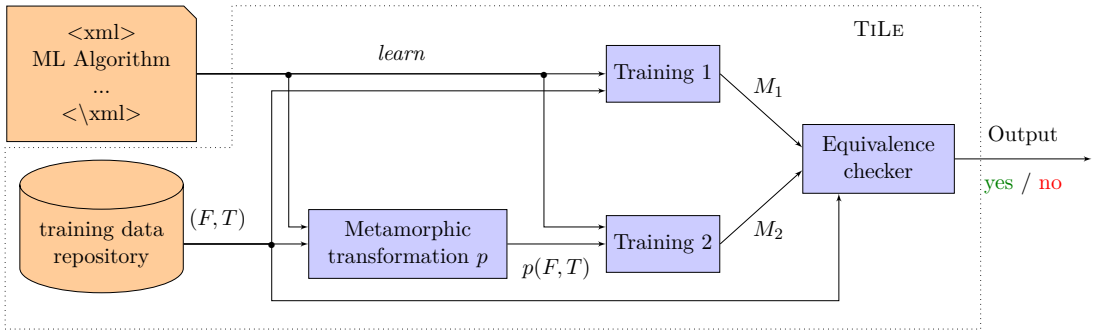


Figure 4.1: Workflow of the testing approach

4.2 Testing Approach

The central idea of the balanced data usage is to perform three metamorphic transformations on the training data, and then finding out whether, in each of these cases, equivalent models are being generated. As described beforehand (in Section 2.2.2 of Chapter 2), the metamorphic testing approach works by first taking two inputs (the original and the transformed ones) and then executing both of them separately on the system under test to get two different outputs. These are then compared based on the transformation that is being applied. Hence, using metamorphic testing to test the balanced data usage property fits quite naturally. To test the balanced data usage property we propose a metamorphic testing approach that consists of two main steps: (a) applying the metamorphic transformations on the training dataset, (b) checking the equivalence of the ML models generated before and after applying a transformation. We give a brief overview of our approach next and then describe these two steps in detail.

4.2.1 Overview

The workflow of our testing approach is depicted in Figure 4.1. We have several parts in this testing approach. First of all, as inputs, we give an ML classification algorithm implemented in any of the machine learning libraries we consider[4] and a metamorphic transformation to apply to the training data, both of which can be specified by the tester using an `XML` file [5].

As part of our framework, we furthermore have a set of training datasets, forming a training data repository. Along with that, we have a unit applying the desired metamorphic transformation p on the training dataset. Finally, there is an equivalence checking mechanism which is used to check the equivalency between two models. Given an input ML algorithm and a specific metamorphic transformation, first of all, we take a training dataset (F, T) from the data repository and apply the user-given transformation p on it. Next, we train the algorithm on the original (F, T) and the transformed dataset $p(F, T)$, thereby resulting in two different models M_1 and M_2. These models are then supplied to the equivalence checker, which then checks whether the two models are equivalent. This process is then repeated for every (F, T) in the

[4]The libraries we consider in this work are detailed in Section 4.3.
[5]An example of such an XML file can be found in Figure A.1 in page 146.

data repository, and each time the models are checked by the equivalence checker. If they are equivalent then the checker returns 'yes', otherwise 'no'.

We check the balancedness property by considering each of the metamorphic transformations $p \in P$ (defined in Section 4.1), and for each of these transformations, we check whether the generated models are equivalent. When any of the transformations p resulted in non-equivalent models reported by the equivalence checker, for any of the datasets in the training data repository, unbalancedness is detected and reported. More formally, for our training data repository TR, if

$$\forall (F,T) \in TR: \ \forall p \in P: \ learn(F,T) \not\equiv learn(p(F,T))$$

then we say the transformation p applied on the training set (F,T) has caused unbalancedness.

However, apart from getting a simple yes/no answer in testing balancedness, we are also interested in computing *relative* unbalancedness. This would then essentially give us an idea of how often an ML algorithm shows sensitivity to a specific transformation. Given a metamorphic transformation $p \in P$ and for a training set (F,T), we suppose,

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

Essentially, the $diff(F,T,p,learn)$ function returns 1, if the transformation p when applied on the training data (F,T) resulted in a non-equivalent model, otherwise it returns 0. We use this to first compute how each metamorphic transformation individually contributes to unbalancedness. In other words, we aim to find out the relative unbalancedness for each transformation or the *transformation specific* balancedness indicator. For this, we furthermore define $\mathcal{P}_t(F,T) = \{p \in P \mid type(p) = t \wedge p \text{ is applicable to } (F,T)\}$. For a dataset (F,T), we can apply the permutation π function on it only if the permutation function π belongs to group S_m for the corresponding transformation p. For instance, let us consider the p to be the row permutation and $type(p)$ as rp. Then for a (F,T) with m instances, we can apply the transformation function only if $\pi \in S$. Hence, here 'applicable' refers to the fact that the considered permutative metamorphic transformation for the balanced data usage fits the number of rows of the training set. Thus, in the end, $\mathcal{P}_t(F,T)$ gives the set of different permutations corresponding to a metamorphic transformation t.

The relative unbalancedness or the transformation specific balancedness indicator with respect to a transformation t, defined as $bi_t(learn)$, can be deduced as follows:

$$\begin{aligned} bi_t(learn,F,T) &= \frac{\Sigma_{p \in \mathcal{P}_t(F,T)} diff(F,T,p,learn)}{|\mathcal{P}_t(F,T)|} \\ bi_t(learn) &= \frac{\Sigma_{(F,T) \in TR} bi_t(learn,F,T)}{|TR|} \end{aligned}$$

With the $bi_t(learn,F,T)$, we first compute the relative unbalancedeness with respect to the metamorphic transformation t for the dataset (F,T). Note that the denominator in this case $\mathcal{P}_t(F,T)$ gives the total number of permutations for the specific transformation t applied on (F,T). The numerator sums up the equivalence results (either 0 or 1) for each of the permutations applied on the training data. $bi_t(learn)$ then computes the average of relative unbalancedness over all the training datasets in

the data repository.

However, the transformation specific balancedness indicator, in this case, cannot be exactly computed in practice, since the number of possible permutations while considering a typical training dataset, is often too large. For example, a training dataset, Census-income, considered as part of our training data repository, contains 48,842 instances. Hence, for row permutation operation, generating 48,842! permutation of instances and checking each of them is practically impossible. Therefore, in our work we do not compute the exact values of different balancedness indicators or, we do not check the equivalency considering each of the permutations. Rather we consider *approximated* versions of them by considering some *specific* permutations. Our empirical evaluation (in Section 4.3) showed, without considering all the permutations, we still could find algorithms sensitive to different such transformations. Next, we describe these specific permutation strategies we adopt to check the balanced data usage property.

4.2.2 Permutation Strategies

Balanced data usage property essentially requires three types of metamorphic transformations to be applied to the training data–permutation of training instances, permutation of feature values and the shuffling of feature names. However, due to a large number of instances or features in most of the datasets, we cannot check for all the possible permutations. Therefore, to this end, we employ some specific strategies to select some of them. Next, we briefly describe each of the permutation strategies.

Random. In this case, we sample a fixed percentage of the total number of permutations uniformly at random. This percentage value can be configured by the tester during the testing process. For example, let us consider that the tester specified a value of 50% in performing column permutation (like in the example XML file depicted in Figure A.1 on page 146). This means that 50% of all possible permutations will be selected randomly and then will be checked when column permutation is performed. For example, if a dataset contains 6 features, then $\lceil(6! \times 50)/100\rceil$ or 320 different permutations will be checked. In other words, we would then generate 320 different permutations of the columns resulting in 320 different transformed datasets. For each of these 320 cases, we would check the equivalency between the models generated on the original and transformed dataset.

Ordering. This is done by first of all giving the set of class values Y an ordering and then sorting the training instances of the dataset based on that. To this end, we perform the ordering of the instances either in ascending or in descending order by keeping the ordering of the training data within the classes. This strategy can only be applied for the permutation of the training instances and is not possible to be used for permuting the feature values (i.e., columns).

Alternating. Here we again use the ordering on the set of class values and then reorder the training instances so that the classes corresponding to each of the instances alternate.

Reversing. This transformation simply inverses the order of the original training dataset, either row-wise or column-wise (based on the specified transformation). For ex-

ample, let us suppose, the training dataset we consider contains m instances. This is defined as the set of ordered instances, $\langle i_1, i_2, \ldots, i_{m-1}, i_m \rangle$, on which the reversing the row operation would result in an order as $\langle i_m, i_{m-1}, \ldots, i_2, i_1 \rangle$. We can similarly perform the reversing of the columns.

Batch-flipping. For this, first of all, we divide the training set into b number of batches based on the user-given parameter `shuffleBatchSize`. For example, let us assume that the size of each batch is 10 and therefore, if we have N number of training instances we will have $b = \lceil N/10 \rceil$ number of batches. Once we have the batches, next we randomly move the batches to any of the b positions. Initially, we have b choices as we have b number of such blocks to fill in. This can be thought of as if we have b number of training instances and we are performing a random permutation on them. Except, in this case, each of them is not a single instance, but rather a set of training instances. The total number of possible permutations with b batches can again be very large if b is large. This can also be controlled by the tester by providing a percentage value. This would then be used to select the percentage of all possible permutations uniformly at random. Furthermore, the size of the batch should not be really small, for instance, if it is 1, we will end up performing a random permutation of instances.

Note that, except for random and reversing permutation strategies, the rest of them are only applicable to the permutation of rows and not to feature values or columns. In case of feature name shuffling transformation, we simply consider a single random shuffle and not all possible permutations. Furthermore, as several of the ML algorithms can only consider numerical attributes, we do not take the transformation –renaming of the feature values– as a part of the balanced data usage property. Rather, we apply this transformation to the training dataset beforehand as a part of the data processing step, and therefore, none of the ML algorithms are considered to be sensitive to this.

After applying a metamorphic transformation on the training dataset we train the given ML algorithm on it to generate a new model. Next, we check whether this newly generated model is equivalent to the one we generated using the original training dataset. This is done by performing equivalence checking methods which we describe next.

4.2.3 Checking Equivalence

In this step, we check the equivalence relationship between the models generated before and after applying a transformation. To this end, we have several challenges to overcome. First of all, as we discussed in the last chapter (Chapter 2), machine learning models can be of various types, depending on the learning algorithms being used. On top of that, the learned models are essentially complex entities and therefore, we need specific mechanisms to check the equivalency between them. For that, we employ two mechanisms, namely *computing* and testing equivalence. First, we describe the equivalence computation method and then we reason why this approach can only be used for some algorithms. Thereafter, we give our equivalence testing approach and in the end, we present an algorithm to effectively combine both of these mechanisms.

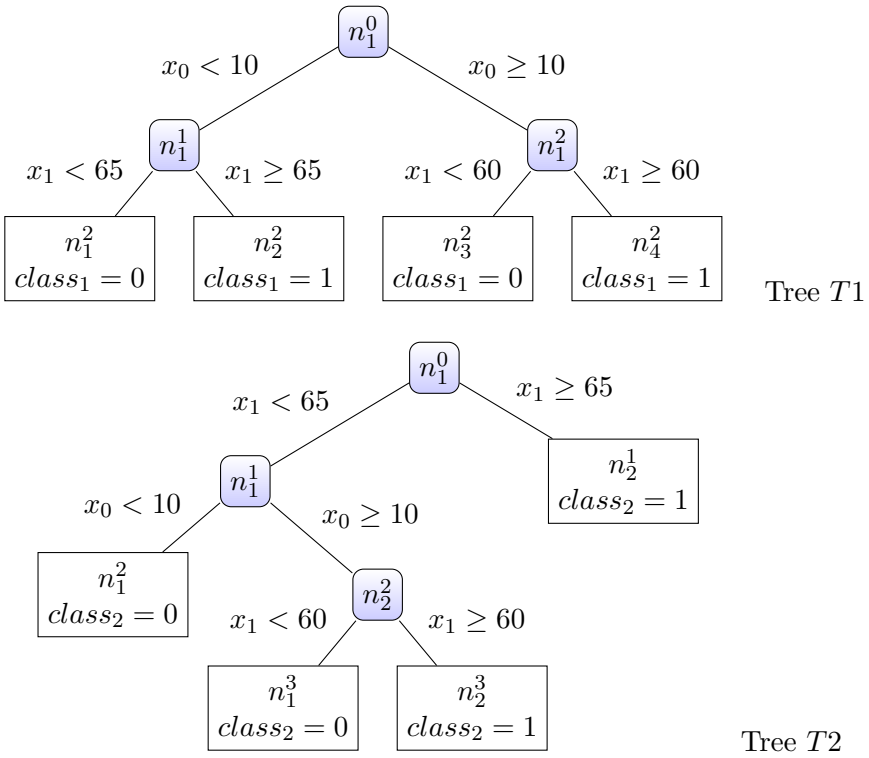


Figure 4.2: Two syntactically different but equivalent trees

Computing Equivalence

This approach cannot be applied to all the classification algorithms and moreover, the techniques would be different for different types of ML algorithms. We start by describing the equivalence computation method for the decision tree algorithm.

Decision tree. Given a training dataset, a decision tree algorithm learns a (binary) tree of decisions where the branches are conjoined with the conditions on the feature values and the leaves denote classes (see Section 2.1.1 of Chapter 2 for more details on decision trees). Let us consider Figure 4.2, where we have two trees, $T1$ and $T2$. Let us assume that trees $T1$ and $T2$ are generated before and after applying a specific metamorphic transformation respectively. These two trees look different as they have different representations, however, they are defining the same model. Therefore, if we use a simple technique like comparing tree branches and leaf nodes, it will fail in this case and is not sufficient to determine whether the two trees are equivalent. Hence, we give an equivalence checking mechanism by translating both of the decision trees into logical formulas and then checking their *logical equivalence* by using the state-of-the-art satisfiability modulo theory (SMT) solver Z3 [MB08].

To perform this checking, first of all, we need the logical formulas for the two trees. For this, we use the encoding approach described in Section 3.2.1 of Chapter 3. Let us assume, the logical encoding of the tree $T1$ is denoted as φ_{T1} and the tree $T2$ as φ_{T2}. We furthermore use the variables $class_1$ and $class_2$ to represent the class predictions for $T1$ and $T2$ trees respectively.

Once we get the encodings of the trees, next we define the logical constraint specifying the equivalency. Two machine learning models are said to be equivalent if for any input, the predictions given by the models are the same, i.e., $(class_1 = class_2)$ (see

1 ; Encoding of the tree of Figure 3.2
2 $(\text{true} \wedge (x0 < 10) \wedge n_{11}^{(1)}) \vee ((\text{false} \vee \neg(x0 < 10)) \wedge \neg n_{11}^{(1)})$
3 $(n_{11}^{(1)} \wedge (x1 < 65) \wedge n_{11}^{(2)}) \vee ((\neg n_{11}^{(1)} \vee \neg(x1 < 65)) \wedge \neg n_{11}^{(2)}) \wedge (n_{11}^{(2)} \implies class_1 = 0)$
4 $(n_{11}^{(1)} \wedge (x1 \geq 65) \wedge n_{12}^{(2)}) \vee ((\neg n_{11}^{(1)} \vee \neg(x1 \geq 65)) \wedge \neg n_{12}^{(2)}) \wedge (n_{12}^{(2)} \implies class_1 = 1)$
5 $(\text{true} \wedge (x0 \geq 10) \wedge n_{12}^{(1)}) \vee ((\text{false} \vee \neg(x0 \geq 10)) \wedge \neg n_{12}^{(1)})$
6 $(n_{12}^{(1)} \wedge (x1 < 60) \wedge n_{13}^{(2)}) \vee ((\neg n_{12}^{(1)} \vee \neg(x1 < 60)) \wedge \neg n_{13}^{(2)}) \wedge (n_{13}^{(2)} \implies class_1 = 0)$
7 $(n_{12}^{(1)} \wedge (x1 \geq 60) \wedge n_{14}^{(2)}) \vee ((\neg n_{12}^{(1)} \vee \neg(x1 \geq 60)) \wedge \neg n_{14}^{(2)}) \wedge (n_{14}^{(2)} \implies class_1 = 1)$
8
9 ; Encoding of the tree of Figure 3.3
10 $(\text{true} \wedge (x1 < 65) \wedge n_{21}^{(1)}) \vee ((\text{false} \vee \neg(x1 < 65)) \wedge \neg n_{21}^{(1)})$
11 $(n_{21}^{(1)} \wedge (x0 < 10) \wedge n_{21}^{(2)}) \vee ((\neg n_{21}^{(1)} \vee \neg(x0 < 10)) \wedge \neg n_{21}^{(2)}) \wedge (n_{21}^{(2)} \implies class_2 = 0)$
12 $(n_{21}^{(1)} \wedge (x0 \geq 10) \wedge n_{22}^{(2)}) \vee ((\neg n_{21}^{(1)} \vee \neg(x0 \geq 10)) \wedge \neg n_{22}^{(2)})$
13 $(n_{22}^{(2)} \wedge (x1 < 60) \wedge n_{21}^{(3)}) \vee ((\neg n_{22}^{(2)} \vee \neg(x1 < 60)) \wedge \neg n_{21}^{(3)}) \wedge (n_{21}^{(3)} \implies class_2 = 0)$
14 $(n_{22}^{(2)} \wedge (x1 \geq 60) \wedge n_{22}^{(3)}) \vee ((\neg n_{22}^{(2)} \vee \neg(x1 \geq 60)) \wedge \neg n_{22}^{(3)}) \wedge (n_{22}^{(3)} \implies class_2 = 1)$
15 $(\text{true} \wedge (x1 \geq 65) \wedge n_{21}^{(2)}) \vee ((\text{false} \vee \neg(x1 \geq 65)) \wedge \neg n_{21}^{(2)}) \wedge (n_{21}^{(2)} \implies class_2 = 1)$
16
17 ; Equivalency constraint
18 $\neg(class_1 = class_2)$

Figure 4.3: Logical encoding in finding out equivalence of tree models $T1$ and $T2$

Definition 4.1). Essentially, we aim to find out a satisfiable example for the formula $\varphi_{T1} \wedge \varphi_{T2} \wedge \neg(class_1 = class_2)$ by giving it to the SMT solver. If the solver finds a satisfiable example to this formula, it would then give us a *logical model* as the assignments to the features which in this case are x_0 and x_1 (and also the node variables), for which the values of $class_1$ and $class_2$ are different. In other words, from the logical model we could extract a feature vector, which given to the models $T1$ and $T2$, would produce two different class values, and thus $\neg(class_1 = class_2)$ constraint evaluates to true and hence, the models are not equivalent. However, if the solver cannot find a satisfiable example to the formula, then the two models are said to be equivalent since the constraint $\neg(class_1 = class_2)$ cannot be evaluated to be true for any assignment of values to the features x_0 and x_1.

The encodings of the two trees $T1$ and $T2$ are given in Figure 4.3, based on the approach discussed in Chapter 3, from Lines 2-7 and 10-15 respectively. Note that, each line here represents the encoding of a node and all the lines are essentially conjoined [6]. Finally, in line 18, we give the logical constraint describing the negation of the equivalency constraint. Given, this logical formula to a satisfiability checker, we do not find a satisfiable example, and thus, it proves that these two trees are equivalent. However, if we change any of the class values for either of the trees and keep the others unchanged, we will have two non-equivalent trees. For instance, if we change the class values of the leaf node $n_1^{(3)}$ from 0 to 1 of the tree $T2$, then the trees are not equivalent and we will get a counter-example as $I = \{x0 \mapsto 10, x1 \mapsto 0, class_1 \mapsto 0, class_2 \mapsto 1\}$. It tells us, for the input vector (10, 0), the tree $T1$ gives a class value of 0, and the tree $T2$ gives the class value of 1 and therefore, they are not equivalent.

Neural network. It is possible to use a similar sort of technique, as described for the decision tree, in checking the equivalency of neural network (NN) models, i.e., by translating the NN model into logical formula (using the technique described

[6] Note that we do not put a conjunction operation at the end of each line for the brevity.

in Chapter 3) and computing for equivalency. However, there are two significant challenges in performing such computations on the neural networks: (a) only specific types of neural networks can be translated to logical formulas, and (b) even if we are able to translate the NN model to logical formula, the satisfiability checking would be time consuming due to the presence of a large number of arithmetic operators in the encoding of the model. Therefore, in this case, we check the equivalency between two neural networks models by comparing the *weights* and *biases*, i.e., by comparing the two sets of *parameters* defining the networks. The two networks considered for equivalence checking has the same architectures, i.e., an equal number of *layers* and *neurons*, since they are generated by training a single given neural network algorithm (i.e., the algorithm under test). However, note that, having the same architecture in two neural networks does not automatically imply that they are equivalent models since the parameters learned for the two networks could still be significantly different. Our technique to compare the learned parameters of two networks can only be used for ReLU network, is fast, and furthermore not bounded by the size of the network compared to the satisfiability checking approach. For checking the equivalency of any other types of networks, testing equivalency technique can be used [7].

Support vector machine (SVM). For SVM, we check the equivalency by comparing the *hyperplanes* between two SVM models. As described in Section 2.1.1 of Chapter 2, the objective of this learning algorithm is to generate a *hyperpalane* in the n dimensional space (where n is the size of the feature vector) which should distinctly classify the data points. Hence, after the learning phase, this algorithm generates a hyperplane as the learned model [8]. For a binary classification problem, the hyperplane takes the following form:

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

Here, w_j is the weight associated with the feature j and b is defined as the offset value. The weights associated with the features and the offset are learned during the learning phase of the SVM. Thus, a learned SVM model has a unique set of weights and offset. To check the equivalency between two SVMs, we compare these parameters between two models. If the difference between these two sets of parameters for two SVM models are significant, we conclude that they are not equivalent.

Apart from these three algorithms, we also have an equivalence computation mechanism for the logistic regression model. We do not describe it separately as this method works similarly to the method described for SVM, i.e., by comparing the weights corresponding to the features.

However, we do not have computation techniques for checking the equivalence of other types of machine learning models. There are two reasons for this: (a) some ML algorithms do not generate any models after the learning phase which could be computed for equivalence, and (b) even if we have models, computing equivalence can be really *expensive* in some cases. ML algorithms like k nearest neighbors (k-NN) and naive Bayes do not generate any model after the learning phase. For instance, k-NN algorithm after the learning phase simply stores the training instances. During

[7] In this work, we check the balancedness of ReLU network, and checking other types of networks such as convolutional or binarized networks are potential future work.

[8] Note that, if the training dataset contains instances that are not linearly separable, a non-linear transformation is first performed on the plane by performing higher order polynomial transformations [AMMIL12].

the prediction phase, for a given input instance, k nearest data points are computed. The classes of those data points are then used to predict the class value of the input instance (see Chapter 2 for more details). Thus, k-NN does not generate any model to be used for the prediction phase. Naive Bayes algorithm works by simply computing the *posterior* probability of predicting a class by using the Bayes theorem, or the Gaussian process. Thus, naive Bayes also does not generate any model which could be used for equivalence computation.

For the tree-based ensemble algorithms like random forest and boosting, it is possible to apply a similar sort of mechanism as is used for decision trees. However, solving the SMT formula describing the ensemble containing a large number of decision trees would incur a huge runtime and therefore, SMT solving to find out equivalency in this case is costly [9]. As a result, currently, we use testing as a means to check the equivalency of such models. To summarize, in this thesis, we developed equivalence computation methods for decision trees, SVM, neural networks and logistic regression algorithms, and for the rest, we perform equivalency testing which we describe next.

Testing equivalence

This testing approach follows the traditional random testing approach of generating test inputs by using the uniform distribution and then executing them on the software under test in order to find a failure. In our case, we randomly generate a test instance x and then check if $M_1(x) = M_2(x)$, and we do it for a number of such instances. Since it is not possible to perform exhaustive testing, i.e., testing for all possible test inputs using random testing, to this end, we employ some strategies to generate test cases in a systematic way.

- First of all, we randomly generate some data instances from the domain of the feature values $\vec{X}$ of the dataset. The number of such test instances generated can be controlled by a tester-defined parameter `INPUT-RATIO` (see Algorithm 3) as a percentage of all possible inputs of the feature values $\vec{X}$.
- We also use training data for testing the models. We randomly get a number of training instances from the training dataset based on the input given by the tester in `TRAIN-RATIO`. This value indicates the percentage of the training data to be used for testing.
- Finally, in order to cover the corner cases we generate two instances with the minimum and maximum of all the feature values (as obtained from the training dataset). Along with that we also generate instances with the arithmetic mean and median of all the feature values.

Algorithm 2 describes the overall equivalence checking approach which combines the equivalence computation and the testing methods. This algorithm requires the inputs as the two models to be checked; M_1 and M_2, and the original training dataset. In Line 2 first we check, whether there exists an equivalence computation method for the given type of model. If it exists, we compute, and if we find the two models to be equivalent, we return true (in Line 5). Otherwise, when our computation method

[9] Note that there are some recent works such as [IISM22] which gives SAT encodings for these models. As a potential future work, this can be used to compute equivalency for the tree based models.

Algorithm 2 *equi* (checking equivalence of models)

Input: M_1, M_2 ▷ *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(M1)) then                    ▷ compute equivalence
3:     equal := computeEqui_type(M1)(M1, M2);
4: if (equal) then
5:     return true;
6: else                                                ▷ test equivalence
7:     equal:= equiTest(F, T,M1,M2);
8: return equal;
```

finds the models to be not equivalent or in case of non-applicability of the equivalence computation method, we call the *equiTest* algorithm (in Lines 6-7).

The *equiTest* algorithm 3 essentially performs testing to check the equivalency between two models. We earlier mentioned our two main sources of getting the test inputs: random input generation from the domain of feature values and randomly taking the test inputs from the training dataset. In this algorithm, we effectively combine these two approaches. First, we randomly generate `INPUT-RATIO` percentage of instances from the domain of feature values $\vec{X}$. The function random($\vec{X}$) in Line 4 generates an input instance uniformly at random. If this input is not generated beforehand, it is included in test set ts (Lines 5-6). With this newly generated instance, we check if it shows the violation of the equivalency, if yes, we return. Otherwise, we go on until we generate a fixed number of test inputs (given by the value of `no-random`).

After testing with the randomly generated instances, we furthermore randomly take the instances from the training dataset as test inputs and test equivalency (Lines 10-15). Finally, the minimum, maximum, arithmetic mean, and median are computed from the set of feature values $\vec{X}$. For each of these functions f in Line 16, we generate the corresponding instance by applying the function on the training set T in Lines 17-18. If these instances are not already generated by the random generation process beforehand they are added to the test set ts and then used to check for equivalency (Lines 19-22).

The equivalency checking between two models corresponding to a test instance is performed by the Algorithm 4. To this end, for any generated test inputs $t \in \vec{X}$, assuming the prediction corresponding to M_1 and M_2 are y_1 and y_2 respectively, then we check whether $y_1 = y_2$.

The entire framework implementing the workflow from Figure 4.1 and the Algorithms 2, 3 and 4 are implemented in a testing tool called TiLe [10]. We give a brief description of the tool in Section A.1 of Appendix A.

4.3 Evaluation

In this section, we describe the experimental evaluation of our testing tool TiLe (Test-Ing of LEarners) on learning algorithms. In our evaluation, we aim at testing the ML classification algorithms taken from 5 different ML libraries with respect to the bal-

[10] https://github.com/arnabsharma91/TiLe

Algorithm 3 *equiTest* (testing equivalence of models)

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

```
1: ts:= ∅; count :=0;
2: no-random := (INPUT-RATIO×|X⃗|)/100;
3: while count < no-random do
4:     t := random(X⃗); count := count + 1;
5:     if (t ∉ ts) then
6:         ts := ts ∪ {t};
7:         if ¬equiv(M1, M2,t) then
8:             return false;
9: no-train := (TRAIN-RATIO×|T|)/100; count :=0;
10: while count < no-train do
11:     t := random(T); count := count + 1;
12:     if (t ∉ ts) then
13:         ts := ts ∪ {t};
14:         if ¬equiv(M1, M2,t) then
15:             return false;
16: for f ∈ {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 ∉ ts) then
20:         ts := ts ∪ {t};
21:         if ¬equiv(M1, M2,t) then
22:             return false;
23: return true;
```

anced data usage property, and furthermore, find out the effectiveness of our tool. Essentially, for the latter, we perform an *internal evaluation* of our tool in finding out the effectiveness of the transformations we consider and our equivalence checking approach. We first describe the research questions which drive our evaluations and the corresponding setup for it in Section 4.3.1. Then we report the results of evaluating TiLe in Section 4.3.2.

4.3.1 Experimental Setup

The evaluation of our testing approach focuses on two main aspects. We first perform the *external evaluation* of TiLe which looks at whether the classification algorithms implemented in the state-of-the-art ML libraries use the training data in the learning phase in a balanced way. Furthermore, we also check whether we could use TiLe to detect unbalancedness in known unbalanced algorithms. The following two research questions are designed considering the external evaluations of TiLe.

RQ1 Are ML algorithms used in the popular machine learning libraries balanced?

RQ2 Can TiLe be used to detect unbalancedness in *by-design-unbalanced* ML algorithms?

Next, we aim to find out the effectiveness of our tool by performing some internal evaluations. To this end, we check the effectiveness of different permutation strategies (described in Section 4.2.2) and also our equivalence testing mechanism. The following research questions are considered for the internal evaluations of our tool.

Algorithm 4 *equiv* (testing equality of models M_1 and M_2 on given input data instance t)

Input: M_1, M_2 ▷ *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** $(y_1 = y_2)$;

RQ3 Which permutation strategies are effective in finding unbalancedness?
RQ4 How effective is our equivalence testing mechanism?

Next we describe the experimental setup that we design to evaluate the above-mentioned research questions.

RQ1. For evaluating this research question, we consider twenty nine machine learning algorithms taken from the 5 prominent ML libraries, namely `scikit-learn` [PVG+11], `WEKA` [WFH11], and for ensemble algorithms, `XGBoost` [CG16], `CatBoost` [PGV+18], and `LightGBM` [lig19]. Note that for the evaluations we set the *hyper-parameters* of the algorithms we check, to the default values (as given in the corresponding libraries) [11]. For example, in the case of neural networks, the hyper-parameter includes the number of layers and the number of neurons in each layer. We set these numbers to their default values for the neural networks (as defined in the `scikit-learn` library). Moreover, we take a support vector machine with the *linear kernel*, a naive Bayes algorithm with the Gaussian process, and ada boost algorithm with the decision trees as the *base* algorithm.

RQ2. In this evaluation, we consider the ML algorithms which are by design not balanced. For instance, there are *fairness-aware* algorithms, that learn from the training data in a specific way, as to ensure that the generated model is *fair* with respect to a specific fairness definition (see Section 7.2.1 of Chapter 7 for details on fairness in ML). These algorithms are required to treat the training data in an unbalanced way during the learning phase to mitigate the *unfairness* present in the dataset. Therefore, the fundamental idea of our balanced data usage property–learning what is in the training data–is implicitly violated by any fairness-aware algorithm at the cost of learning a fair model. We check the balancedness of two such fair-aware algorithms, proposed by Zafar et al. [ZVGG17] and Calders et al. [CKP09]. As the implementation of the latter was not publicly available, we implemented their approach using the techniques described in their paper.

Moreover, we consider a similar type of unbalanced algorithm, which learns to guarantee the *monotonicity* property. These are known as *monotonicity-aware* algorithms. Alike the algorithms discussed above, monotonicity-aware algorithms are required to generate *monotone* models, even when the training data contains

[11] For an ML algorithm, a set of hyper-parameters of the learning algorithm is required to be set to some specific values before the learning process. They control the behavior of the learning algorithm and affect how the model is trained.

non-monotonicity. For this, we consider the monotonicity-aware algorithms from two ML libraries, `XGBoost`[Che19] and `LightGBM`[lig19].

A variant of the k-NN algorithm considers a specific number of training instances in its learning phase. Since k-NN requires storing the training instances in its learning phase, in some cases, when the number of instances in the training dataset is high, the algorithm stores a subset of such instances. The number of instances considered in this set is called the *window size*. Since this process involves ignoring some training instances deliberately, this k-NN algorithm is inherently unbalanced. We take the k-NN algorithm from the `scikit-learn` library and fix a window size of 500, and denote the algorithm as k-NN$_{ws}$.

RQ3. In this evaluation, we consider the experiments conducted for RQ1 and record how often each of the permutations reveals unbalancedness. For this, we compute the relative unbalancedness as described in Section 4.2 to find out the effectiveness of different permutation strategies described in Section 4.2.2 by only considering the row permutations. We, however, do not include the random permutation strategy in this experiment, since it could randomly generate the permutations of one of the other strategies.

RQ4. With this research question, we aim to find out the effectiveness of our equivalence testing approach. For this, we consider generating non-equivalent models by applying a specific metamorphic transformation. Precisely, we take the *flipping* operation which is guaranteed to generate non-equivalent predictive models when applying it on the training data.

To this end, this transformation, when applied on a training dataset, a specific percentage of the total number of instances are selected randomly, and their corresponding class values are changed to different values. For example, suppose we have a total of N number of instances in the dataset, and we apply the flipping transformation with P% change. Then $\lceil N * P/100 \rceil$ number of data instances selected randomly will have their class values changed. Basically, we simply flip the class values for these selected instances from 1 to 0, or 0 to 1, and any class value of ≥ 1, is changed to 0.

4.3.2 Results and Discussions

We now report the results for all the research questions described above.

RQ1. In Table 4.2, we show the results of TiLe applied to 11 classification algorithms. The first column of the table specifies the name of the different algorithms and the next three columns give the results of the three metamorphic transformations corresponding to the balanced data usage: feature name shuffling (*FN shuffle*), row permutation (*Row perm.*) and column permutation (*Col. perm.*). Here we give the results for two of the most popular libraries[12], `sklearn` [PVG+11] and `Weka` [WFH11]. The ✓ indicates that the algorithm is sensitive to that specific transformation, i.e., when retrained on the transformed training data, a different model is generated compared

[12]Note that, since the other libraries we consider contain only ensemble algorithms, for uniformity, we report the results of these libraries along with the results of ensemble algorithms for sklearn together.

Table 4.2: Sensitivity to metamorphic transformations

ML algorithms	*FN shuffle* sklearn/Weka	*Row perm.* sklearn/Weka	*Col. perm.* sklearn/Weka
k-NN	✗/✗	✓/✓	✓/ ✗
Decision Tree	✗/✗	✗/✗	✓/✓
Naive Bayes	✗/✗	✗/✗	✗/✗
SVM	✗/✗	✓/✓	✗/✗
Neural Network	✗/✗	✓/✓	✓/✓
Logistic Regression	✗/✗	✓/✗	✓/✓
AdaBoost	✗/✗	✗/✗	✓/✓
Bagging Classifier	✗/-	✓/-	✓/-
Extra Trees	✗/✗	✗/✗	✓/✓
Gaussian	✗/-	✗/-	✗/-
Elastic Net	✗/-	✗/-	✓/-

Table 4.3: Sensitivity to metamorphic transformations of random forest and gradient boosting algorithms

Classifiers	*FN shuffle* skl/XGB/LGBM/CB/Wk	*Row perm.* skl/XGB/LGBM/CB/Wk	*Col. perm.* skl/XGB/LGBM/CB/Wk
Random Forest	✗/✗/✗/✗/✗	✓/✓/✓/-/✓	✓/✓/✓/-/✓
Gradient Boosting	✗/✗/✗/✗/✗	✓/✗/✗/✓/✓	✓/✓/✓/✓/✓

to the model trained on the original training data. On the other hand, ✗ denotes that the algorithm is not sensitive to that transformation. If the implementation of a specific ML algorithm is not available in the corresponding library, we mark it as '-'. For instance, the bagging and the elastic net classification algorithms are not implemented in the `WEKA` library.

Since all the algorithms do not consider feature names during their learning phase, none of them are sensitive to feature name shuffling. However, it is quite evident from the results of Table 4.2, that except for naive Bayes and Gaussian process, the rest of the ML algorithms are unbalanced because of their sensitivity to either the row or column permutations. The results are consistent across `scikit-learn` and `WEKA` libraries except for two algorithms, k-NN, and logistic regression. The reasons for such differences might be due to the different implementation approaches used for these algorithms.

In addition to the two libraries we discussed above, we furthermore consider several boosting libraries in testing the implementations of the random forest and gradient boosting algorithms, results for which are depicted in Table 4.3. Note that, here we consider three boosting libraries: `XGBoost` (`XGB`) [Che19], `LightGBM` (`LGBM`) [lig19], `CatBoost` (`CB`) [cat21], along with `scikit-learn` (`skl`) [PVG+11] and `WEKA` (`Wk`) [WFH11]. The results further show that most of these ensemble algorithms across different libraries are sensitive to row and column permutations. For example, the random forest algorithm is sensitive to both row and column permutations across all the libraries. The implementations of boosting algorithms in some libraries show row sensitivities, for instance in `CatBoost` or `scikit-learn` library, however, not present in the `XGBoost` library.

Table 4.4: Transformation specific balancedness indicators

Classifiers	*Row perm.* `sklearn/Weka`	*Col. perm.* `sklearn/Weka`
k-NN	2.24/0.88	1.98/0.00
Decision Tree	0.00/0.00	21.19/25.19
SVM	5.85/2.98	0.00/0.00
Neural Network	1.05/4.98	25.65/19.09
Logistic Regression	0.63/0.00	0.02/1.19
AdaBoost	0.00/0.00	1.66/5.09
Bagging Classifier	28.61/-	16.73/-
Extra Trees	0.00/0.00	19.05/28.10
Elastic Net	0.00/-	11.10/-

Table 4.5: Transformation specific balancedness indicators for random forest and gradient boosting algorithms

Classifiers	*Row perm.* `skl/XGB/LGBM/CB/Wk`	*Col. perm.* `skl/XGB/LGBM/CB/Wk`
Random Forest	11.20/9.01/2.35/-/18.98	21.09/12.87/18.88/-/19.19
Gradient Boosting	2.12/0.00/0.00/8.81/1.21	17.12/16.53/3.98/9.09/11.17

We further investigate the algorithms which showed unbalancedness in any form, to find out how often such sensitivities occur. To this end, we compute transformation-specific balancedness indicators bi_t (see Section 4.2.1) for row and column permutations, which give us relative measures of sensitivities for each transformations. Tables 4.4 and 4.5 give the results in percentages of total test instances, causing non-equivalency for the corresponding transformation. Here, we omit the *FN shuffle* transformation and the naive Bayes, Gaussian process algorithms since neither the feature name shuffling transformation nor those algorithms revealed any sort of unbalancedness in Table 4.2.

Now, these results show the unbalancedness in the form of sensitivities to the metamorphic transformations we consider, however, do not explain the reasons for such. Below we describe some of the reasons that we found that causing unbalancedness in the ML algorithms.

Tie breaking. This is one of the reasons, causing the k-NN algorithm to be sensitive to the row permutations and the tree-based algorithms to column permutations. k-NN learning algorithm is sensitive to the ordering of the rows, especially when there are ties involved (see Section 2.1.1 of Chapter 2 for details on k-NN algorithm). More specifically, if a new instance to be predicted has the same shortest distance to more than two instances in the dataset, which is a tie situation, the instance to be included in the k nearest neighbors would depend on the order of the instances. Therefore, the changes in the ordering of the rows in the training data can change the model being generated. This similar reasoning can also be applied to tree-based algorithms. In the case of the decision tree, while deciding which feature to be used for splitting at a certain depth of the tree (see Section 2.1.1 of Chapter 2), we take the one which causes maximum

entropy changes. Now, for some datasets, this might not be a unique feature and there might exist several of them. If we change the ordering of the columns, the choice of such a feature to split would be affected, thereby causing a change to the model being generated. This same reasoning can also be used for any tree-based ensemble algorithms like random forest or boosting are sensitive to column permutations.

Randomness. This is prevalent in the implementations of many ML algorithms and in some cases even impossible to avoid. For example, for the decision tree, the hyper-parameter `random_state` in `scikit-learn` library controls which feature to be selected in case of a tie while finding the best split. Algorithms like support vector machine and logistic regression randomly shuffle the data while learning in batches. Moreover, the random forest algorithm learns by randomly selecting a subset of instances from the training dataset. In our experiments, we attempted to control such randomness by fixing the parameter `random_state` to a fixed value. However, even after that randomness still persists in some ML algorithms and is one of the reasons for the ML algorithms being unbalanced.

Imprecise numerical calculations. The numerical calculations performed in ML algorithms are inherently imprecise and some algorithms are required to perform a lot of such computations. For example, the algorithms like SVM and neural networks perform a large number of arithmetic operations involving high-precision decimal values. Many of these operations are non-commutative and therefore, changing the ordering of the rows can definitely lead to the generation of non-equivalent models.

Initialization. The implementations of some ML algorithms use specific optimization algorithms and/or initialize some of the *learned parameters* to specific values based on the initial observations of the training data. This in some cases is done based on the structure of the training data, and therefore, if the structure changes in any way (after performing row or column permutations), the optimization algorithms would probably give different results or we would get a different set of initial values for the parameters. In either of the cases, we might then get different models. For example, the implementation of k-NN in `scikit-learn` library contains different types of learning algorithms for the learning process, such as k-d tree, ball tree, and so on. The selection of such an algorithm is performed based on the *nature* of the training dataset. Hence, if we perform column permutation, it would potentially change the structure of the dataset and eventually the choice of the learning algorithm as a result would also change. Moreover, these different algorithms have different ways of breaking the tie if it occurs. For instance, k-d tree does it randomly and the ball tree algorithm uses the ordering in the training dataset. As a result, even a column permutation can lead to a change in the generated model for k-NN (which we also saw in Table 4.2). The neural network algorithm in `scikit-learn` assigns some initial values to the learned parameters: weights and biases, based on the structure of the data. Hence, any kind of changes, row or column-wise might lead to a different set of initial values for the weights and biases, ultimately leading to the generation of a different model.

These reasons indicate that essentially the implementation choices that are taken

Table 4.6: Sensitivity to metamorphic transformations in fair-aware and monotonicity-aware algorithms

Classifiers	*FN shuffle*	*Row perm.*	*Col. perm.*
Fair1	✓	✗	✗
Fair2	✓	✗	✗
Montonic (`XGBoost`)	✓	✗	✗
Montonic (`LightGBM`)	✓	✗	✗
k-NN$_{ws}$	✗	✓	✗

while developing the machine learning algorithms are to blame for the unbalanced data usage. The list of reasons that are mentioned here is not exhaustive, and there might be more. Finding out more such reasons could be an interesting future work. Next, we report the results of applying TiLe on the by-design unbalanced algorithms.

RQ2. Table 4.6 shows the results of applying TiLe to the classification algorithms which use the training data in unbalanced ways in their learning phases. As we presume, all of them are sensitive to some specific transformations. For instance, the fair aware (Fair1 [ZVGG17] and Fair2 [CKP09]), or the monotonicity aware algorithms (taken from `XGBoost` and `LightGBM` libraries) are expected to be sensitive with respect to the feature name shuffling. In the learning phase, such an algorithm takes the column values corresponding to a user-given feature name and aim to generate a fair or monotone model, by either changing the dataset or learning in a specific way in order to generate a fair or a monotone model. Therefore, these algorithms inherently treat the training data in an unbalanced way. Hence, changing the name of the features would definitely cause the learner to take a different column and thereby generate a different model. Finally, the k-NN$_{ws}$ which considers storing a subset of instances of the training dataset in its learning phase, as expected, shows to be sensitive to the row permutation.

Thus, by using TiLe we were successfully able to detect unbalancedness present in the algorithms which by design use the data in the learning phase in an unbalanced way. With this, we can also see the potential use of TiLe to detect unbalancedness present in any algorithms. Next, we look at the effectiveness of different row permutation strategies.

RQ3. The aim of this research question is to find out which of the row permutation strategies are effective in detecting the sensitivities occurring due to row permutations. As mentioned earlier, due to a high number of permutations possibilities, we do not consider all of these, rather we take some specific strategies in selecting those. Figure 4.4 compares the result of average deviations over all the algorithms[13] (across all the libraries), caused by different row permutation strategies. The x-axis gives the percentage of average deviations and the y-axis gives different row permutation strategies. Essentially, we record on average how many percentages of test instances show sensitivity for a specific row permutation strategy. The results suggest, almost all of them are equally successful in showing unbalancedness, except 'Reversing' strategy surprisingly being a bit more effective than the others. Note that, the values depicted in Figure 4.4 cannot be directly compared to the results of Table 4.4. However, the

[13]Note that here we do not consider the algorithms which did not show any sensitivity to the row permutation, for e.g., decision tree, Ada boost, etc.

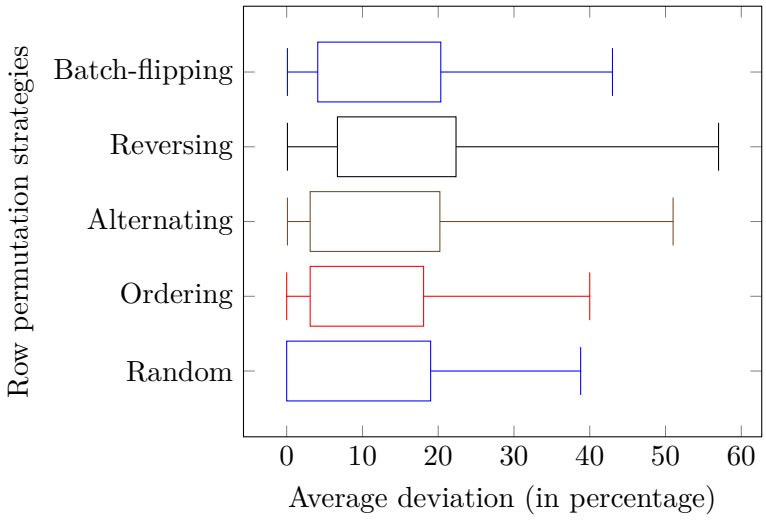


Figure 4.4: Deviation detected by different row permutation strategies

Table 4.7: Percentage of test cases classified differently by the original and transformed classifier after reversing the rows of the dataset

Classifiers	*Test cases*
k-NN	0.41
SVM	19.99
Neural Network	2.35
Logistic Regression	0.48
Random Forest	11.35
Bagging Classifier	11.59
Gradient Boosting	0.86

maximal values corresponding to this figure are coming from the bagging classifier, which matches the highest relative sensitivity for the row permutations as shown by the algorithm.

We further investigate the relative unbalancedness of different classification algorithms when we apply the 'Reversing' rows permutation on the training dataset, the results of which are depicted in Table 4.7. The results show the percentage of test instances that are classified differently by the original and the transformed models after applying the permutation 'Reversing'. To this end, we find the SVM algorithm to be more sensitive to this transformation compared to any other ML algorithms.

RQ4. In evaluating this research question, we construct non-equivalent models by applying the metamorphic transformation *flipping*. For this, we randomly select some instances and flip their class labels.

Selecting just a single instance and then flipping its class label might not change the model, since such a tiny change can often be regarded as an outlier and therefore, might be ignored in the learning process, specifically when the number of training instances is really high. Hence, to this end, we consider a 2% flip applied on the dataset (as this would do the slightest yet detectable changes). For this experiment, we consider five classification algorithms: k-NN, decision tree, SVM, naive Bayes, and neural network. The reason we choose these algorithms is because they would give a good variation

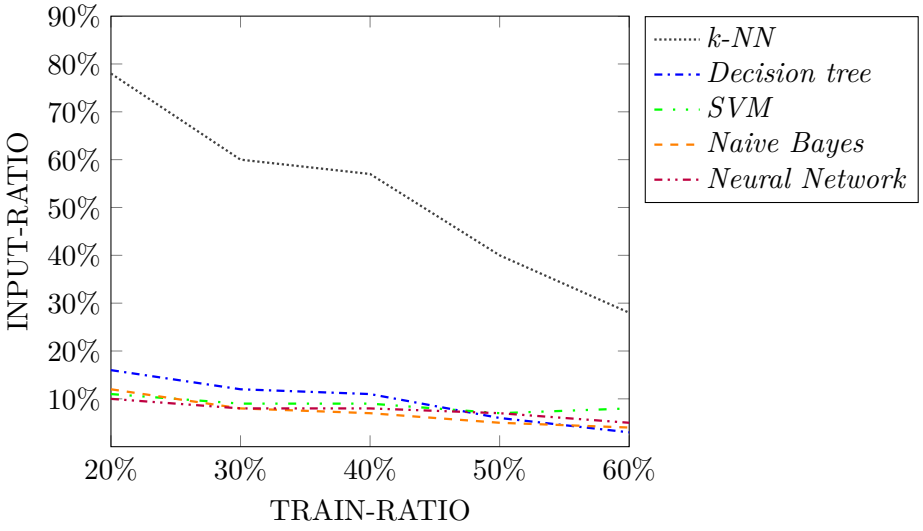


Figure 4.5: Relation between TRAIN-RATIO and INPUT-RATIO

over different learning techniques.

Our testing approach is driven by the generation of test cases which in turn is dependent on two important parameters `TRAIN-RATIO` and `INPUT-RATIO`. These two parameters dictate how many test instances should be generated for testing (see algorithm 4). Thus, to evaluate the effectiveness of our equivalence testing approach we look at how the values of these two parameters affect non-equivalency detection. Figure 4.5 shows the results of our evaluation. On the x-axis different `TRAIN-RATIO` values are given and the y-axis gives the corresponding minimal `INPUT-RATIO` needed to detect non-equivalence. For example, if we take the decision tree, then for a `TRAIN-RATIO` of 20%, we require an approximately 16% of the randomly generated test data to find out the non-equivalence caused due to the 2% class labels flipping. Here we see that non-equivalence detection requires larger randomly generated test inputs initially, however, this number reduces as we take more training instances for testing. The k-NN algorithm learns by storing the training dataset and therefore, is highly dependent on the training dataset. Hence, the drastic drop in requiring the randomly generated test data in the case of k-NN is in line with its dependency on the training dataset. For all the other classification algorithms `TRAIN-RATIO` of 30% and `INPUT-RATIO` of 20% are good choices for detecting non-equivalence. We furthermore used a 1% flip to construct non-equivalent predictive models and then compared these two testing parameters. The results suggest a similar sort of trend as presented in Figure 4.5. Thus, in our evaluations, we see that our equivalence testing approach is effective enough to detect the non-equivalency. However, this result cannot easily be generalized since we only considered a single transformation to generate non-equivalent models. In the future, this can be extensively evaluated by generating non-equivalent models, considering different types of such transformations.

4.4 Threats to Validity

There exist several threats to the validity of the results presented here. First of all, there are several randomnesses involved in our experiments, since we perform random row or column permutations on the training data. Moreover, the ML algorithms considered involve random computations. As a result, the runs of TiLe cannot simply be repeated. However, this does not have an effect on the results of Table 4.2, since we report the sensitivity to any of the transformations whenever we found a run (amongst 20 runs) showing non-equivalence between the two models. The results reported in Tables 4.7,or 4.4 can vary, as these numbers indicate the relative unbalancedness computed based on the randomly generated test instances. We tried to minimize variances in these numbers by performing each experiment 20 times and taking an average of them. The results reported in these tables give an average over all these 20 different values of relative sensitivities.

Next, the choice of training data could also be a potential threat to validity as the unbalancedness we find can occur only on these particular sets of training data. However, to mitigate this effect we consider a number of diverse real-world datasets with varying numbers of rows and columns.

Finally, the correctness of our implementation is another threat to validity. We have performed an extensive evaluation of our testing approach in testing 29 different types of ML algorithms belonging to 5 different machine learning libraries. However, the results of our experiments show some inconsistent behaviours which in some cases cannot be explained and therefore, might be difficult to detect in a large code base. For instance, the unbalancedness might be caused because of simply using an off-the-shelf library which is part of a different package written in a different programming language. In fact, to foster fast computations in the learning phase, `scikit-learn` uses several packages written in the C programming language. Testing of inherently unbalanced algorithms such as fair-aware or monotonicity-aware and using flipping to get expected non-equivalent models in some ways improve on this situation, however, do not represent all the cases.

4.5 Related Work

The work described in this chapter is in line with the existing works described in Section 2.3 of Chapter 2 about using metamorphic testing to test the machine learning based software systems. There we discussed the related works in using metamorphic testing technique in general. In this chapter, however, we keep our focus only on discussing the works of testing the implementation of ML algorithms.

A series of works are done by Murphy et al.[MKA07, MKHW08, MSK09] considering metamorphic testing technique to test the implementations of several machine learning algorithms. In their works, they considered testing SVM, naive Bayes, decision tree, and k-NN algorithms implemented in `WEKA` [WFH11] library, with respect to several types of metamorphic transformations. To perform these transformations they considered some artificially generated training datasets. On the contrary, in this chapter, we defined a property of the learning phase namely balanced data usage which requires any ML algorithm to be invariant under the row, column permutations, and feature name shuffling. To apply the transformations we constructed a training data repository containing a number of datasets- artificial as well as real-world. We then

tested the balancedness property on the implementations of a number of classification algorithms (29 in total), taken from 5 different machine learning libraries, while the works by Murphy et al. [MSK09] considered only four classification algorithms taken from `WEKA`.

Nakajima et al. [NB16] proposed a systematic way to look into an ML algorithm and derive the desired metamorphic transformations. To this end, they only considered the SVM classification algorithm and focus on finding out appropriate transformations. Moreover, they proposed a testing technique that derives test data based on the transformations, which is also limited to SVM. Dwarakanath et el. [DAS+18] proposed some specific metamorphic transformations for SVM and deep learning networks. These are however also specific to the two classification algorithms they evaluated. The metamorphic transformations we consider, as part of the balanced data usage property, are domain independent and not specific to any algorithm. Moreover, our testing mechanism can also be used for testing any classification algorithms.

Pham et al. [PQW+20] in their works reported the generation of two different models in two runs with the same training data for a single deep learning (DL) algorithm. They identified several types of non-determinism in DL tasks such as random seed generation, weights and biases initialization, and the random shuffling of data during learning. Since modern deep learning libraries attempt to give faster results even with a complicated architecture, parallel processing and the Graphics processing unit (GPU) are employed during the learning process. This also results in generating two models in two runs. They reported that DL libraries like `PyTorch` and `TensorFlow` deliberately use non-determinism in order to generate more accurate models. However, some of them, for instance, caused by arithmetic operations in the GPU programming are not deliberate and cannot be controlled, unless tasks are performed in a fully serialized manner incurring a huge runtime to train a model. Such changes in models with the same training data with different runs are surprising to many researchers and practitioners, as is found as a result of their survey.

In recent years, there have been some more works on testing only the deep learning libraries [WYC+20, GXL+20, WLQ+22]. These works focus on finding the implementation bugs such as crash bugs, NaN value bugs, and inconsistency bugs. In these works, they used sophisticated testing techniques, specific to DL algorithms in order to find such buggy behaviours. For instance, in [WYC+20], the authors used model mutation testing specifically designed for DL algorithms or in [GXL+20], the authors used a variation of the causal testing approach to find the implementation bugs in DL libraries.

There exist some approaches using metamorphic relations like the permutation of features and class labels, in order to study the accuracy of the classifier and even improve them later on. For example, in [OG10], Ojala et al. evaluated the effect of such permutation strategies on the training data, and on the performance of the classification. Ding et al. [ZDMR17] used metamorphic relations like inclusion, exclusion, or duplicating training instances on the training dataset to inspect the effect of such transformations on the accuracy of the generated model.

Finally, a balancedness like property is studied by Urban et al. [UM18] where they used static analysis approach to find out whether some specific data are unused in a program. This sort of behaviour, for instance, ignoring certain feature values during the learning phase is another sort of unbalancedness of the ML algorithms. To this end, Urban et al. only gave a theoretical framework based on the abstract interpretation

approach, however, did not perform any experimental evaluation.

To conclude, in this chapter of the thesis, we defined the balanced data usage as a property of the learning phase which requires any ML algorithm to use the entire training data in its learning phase. To this end, we developed a metamorphic testing tool TiLe which could be used to test this property. We furthermore tested 29 different classification algorithms taken from five machine learning libraries with respect to the balancedness property. The results of our evaluation indicate a number of machine learning algorithms are not balanced.

5 Verification-based Testing

In the *white-box* testing approach, we assume that the internals of the machine learning model is known. Thus, a testing technique can be developed considering the structure of the model. However, such a technique would be specific to that model. More often the existing works focus on using this testing approach to test a specific type of model, namely a neural network model, such as in the works of [PCYJ17, SWR+18, HYL+22]. Now, in some cases, it might happen that we need to test different types of ML models, the types of which are not known beforehand. Such a *black-box* nature of the model poses much more challenges in generating test cases that can potentially reveal errors in the model. Apart from a few works (such as [Agg18, ASH+21, XW20]) which consider checking only a specific type of property, a black-box testing technique for testing any type of ML model is largely missing.

To solve this problem, we propose the *verification-based* testing technique in this chapter which enables us to perform testing of machine learning models without considering the types of them. Our approach systematically explores the input space of the given model under test (MUT) by generating effective test inputs with respect to a specific property. This is done by (1) inferring a known white-box model through the machine learning process, which approximates the MUT, and then (2) *computing* the property on the generated white-box model using the verification technique we described in Chapter 3. If the verification results in 'failure', implying the violation of the property, we would get a *counter-example* to the property. The counter-example in this case, is the set of values corresponding to an input instance that resulted in the violation of the output. We can further generate multiple numbers of such counter-examples by employing a technique called *pruning*. Next, these are confirmed with the MUT, since the counter-examples are essentially generated on the inferred white-box model and not on the MUT we are testing. If confirmed, they are returned as the violated test cases with respect to the specified property, otherwise, they are used to improve the quality of the inferred white-box model. This process goes on until we find the violation of the property or a user-defined timeout occurs. However, if no counter-example is found, then the testing process could either stop or proceed further by generating a new white-box model and then repeating the entire process.

We start with the formalization required to describe our testing technique in Section 5.1. Then we describe different steps involved in the verification-based testing in Section 5.2.

5.1 Formalization

We begin by revisiting some of the formalizations we introduced in Chapter 2. Formally, we represent the machine learning model as a predictive function of the following form:

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

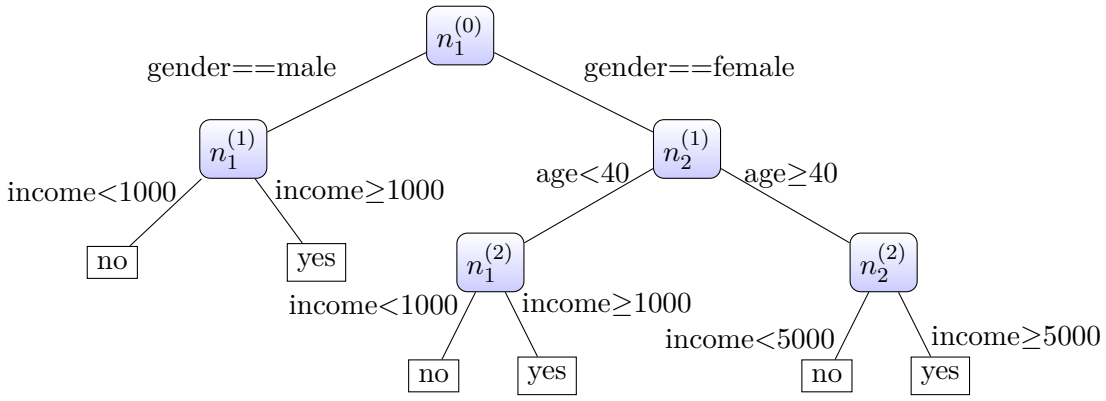


Figure 5.1: A decision tree for predicting loan

where X_i is the value set of feature i, and Y is the set of *classes* [1]. We define $\vec{X}$ to write $X_1 \times \ldots \times X_n$. We furthermore write $\vec{x} \in \vec{X}$, and $\vec{x} = \vec{x}(1), \ldots, \vec{x}(n)$ denotes a feature vector with n features, and $y \in Y$ denotes a class.

In the learning phase, the learning algorithm gets the training data of the form $\vec{X} \times Y$, i.e., a set of data instances with known associated classes (a set of $(\vec{x}, y)$ pairs), which is then used by the algorithm to generalize the *relationship* between the instances and the corresponding classes, and thus, a model is generated which is said to have learned the relationship. The model is then further used to predict classes for unknown data instances.

Next, for the sake of explaining our testing technique, we consider a specific property to test in this chapter. To this end, we take a type of *fairness* property called *individual discrimination* which can be formally defined as follows:

Definition 5.1 *A machine learning model M is* fair *with respect to a* sensitive feature *i, if for any two data instances, $\vec{x}_1, \vec{x}_2 \in \vec{X}$ we have $(\forall_j : j \neq i . \vec{x}_1(j) = \vec{x}_2(j))$ imply $M(\vec{x}_1) = M(\vec{x}_2)$.*

This definition of fairness is introduced by Galhotra et al. [GBM17] and requires the ML model to be *invariant* to the changes of the sensitive feature i. In other words, the ML model should give the same prediction even if the feature values of the sensitive feature i are different in $\vec{x}_1$ and $\vec{x}_2$ instances. For example, consider the decision tree depicted in Figure 5.1 which works as a predictive model to decide whether a person should be given a loan based on the income, age, and gender [2]. Clearly, this tree is not *fair* with respect to the feature gender. For instance, consider the following pair of instances:

$$\vec{x}_1 = \text{income=4500, age=50, gender=female}$$
$$\vec{x}_2 = \text{income=4500, age=50, gender=male}$$

For the first instance $\vec{x}_1$, we get a prediction of 'no' and for $\vec{x}_2$ we get the prediction as 'yes', and thus, the tree gives different predictions to the male and female applicants.

We further use this property and the tree model to illustrate the working of our verification-based testing approach.

[1] Note that, for the simplicity of describing our testing approach, we consider here the single-label classification, extending to other types of learning problems is merely a technical work and not related to the concepts of the approach.

[2] This tree is taken from one of our previous work [SW20a] on fairness testing.

5.2 Testing Methodology

Our testing approach is a form of *learning-based* testing technique where a *model* is being 'learned' and then it is used to generate test cases. The model in this context can be a finite state automaton [PVY99], or a Markov decision process [TAB+19], or a piece-wise linear function [MN10]. The idea of using an inferred model for the sake of analysing a given black-box *system* was first introduced by Angluin in her pioneering work in [Ang87]. Later, Peled et al. [PVY99] extended this idea to apply the *model-checking* technique to analyze specific properties on the learned model. We already gave a detailed discussion of such related works in Section 2.3 of Chapter 2.

Our work is in a spirit similar to this idea of learning-based testing where we use machine learning techniques to learn an ML model and then use a verification technique to analyze the model for a specific property. Thus, we term our testing technique verification-based testing. Since in our work, we primarily focus on testing ML models, therefore, we approximate the model under test by using another ML model, rather than using a finite state automaton model. To this end, we can learn two types of ML models, either a decision tree or a neural network model to approximate the MUT, and then apply a known analysis technique to generate test cases on them. Our testing approach is essentially composed of four main steps: (a) white-box model learning, (b) computing property, (c) pruning, and (d) cross-checking and retraining. We discuss in detail each of these steps below.

5.2.1 White-box Model Learning

Our first step in this testing approach is to learn a known ML model which approximates the model under test (MUT). Since the internals of this model are available to us, we term this as a white-box model [3]. To this end, the learning process follows the traditional machine learning techniques for which we, first of all, need a training dataset.

The dataset in this case is essentially generated by querying the MUT by using a number of input instances. More specifically, we randomly generate a number of instances as the input vectors to the MUT and then we execute all these randomly generated instances on the MUT to get a set of output predictions. Thus, we get a set of $(\vec{x}, y)$ pairs which represents the *functional behaviour* of the MUT. Furthermore, this set of instance-class value pairs takes the form of a typical training dataset $\vec{X} \times Y$. We term this dataset as the *oracle data* and use this to train our desired learning algorithm.

To this end, we can use either a decision tree or a neural network algorithm to be trained on the oracle data to generate the inferred model. The choice of using either of those ML models is driven by the intention of being able to apply a verification technique on the inferred model. Let us denote the MUT as M and the inferred model as M'. Once we learn the model M' approximating the MUT M, in the next step we compute the property on the inferred model.

There are a number of works in the area of *explainable* machine learning, where such an approach is used to learn a so called *interpretable* model from a black-box machine learning model [KW17, TSHL17, TSHW20]. This kind of explanation approach is model *agnostic* since this technique can be applied to any type of ML model. Essen-

[3] Note that, this terminology is consistent with the jargon used in testing domain.

tially, this step in our verification-based testing technique is in a spirit similar to these works. However, in our case, we utilize this concept to learn a model on which we can apply an analysis technique to compute a specific property, the method for which we describe next.

5.2.2 Property Computation

Once we learn an ML model, next we compute the specified property on that model by using the technique described in Chapter 3. To this end, we translate the model and the property into logical formulas and then apply satisfiability solving technique to it. More specifically, first of all, by using the encoding mechanism introduced in Section 3.2 of Chapter 3, we encode the inferred model M' into the logical formula $\varphi_{M'}$. Then we also get the logical formula describing the property as φ_{prop}. Since in this step, we aim to find out whether a violation of the property in the form of a *counter-example* exists, we essentially conjoin the logical formula $\varphi_{M'}$ with the negation of the property formula $\varphi_{\neg prop}$ to get the final formula as $\varphi_{M'} \wedge \varphi_{\neg prop}$ which is then given to an SMT solver. Essentially, we aim to find out, given the conjoined formula whether there exists a satisfiable example to that formula. In other words, we see whether we can find a counter-example, showing the violation of the property on the model M'.

For example, let us consider after the white-box model learning step, we get the decision tree model as in Figure 5.1 for M', and we aim to compute the fairness property (defined in Definition 5.1) on this model. Firstly, we encode this model into the logical formula, and then we also get the logical formula describing the property [4]. Figure 5.2 shows the entire logical formula describing the decision tree model with the conjunction of the negation of the property [5]. Since the property we consider here require two instances $\vec{x}_1$ and $\vec{x}_2$ to define, we thus need to encode the tree model twice and differentiate between them by using a simple variable renaming approach.

Herein, the variables $gender1$, $income1$, and $age1$ correspond to the input feature vector $\vec{x}_1$ and $gender2$, $income2$, and $age2$ correspond to $\vec{x}_2$. Similarly, $class1$ and $class2$ are defined as the outputs for $\vec{x}_1$ and $\vec{x}_2$ respectively. After the model encoding step (described in Figure 5.2 in Lines 3-12 and 15-24), next, we add the logical formula describing the negation of the property. Now, with respect to the input and the output of the model we consider in our example tree, the fairness property can be written as follows:

$$((income1 = income2) \wedge (age1 = age2) \wedge \neg(gender1 = gender2)) \Rightarrow (class1 = class2)$$

The first part of the formula described on the left side of the operator $\Rightarrow$ gives the *pre-condition* on the inputs and $(class1 = class2)$ defines the *post-condition* on the output. However, since in the property computation step our aim is to find out whether there exists a counter-example to the property, we essentially take the negation of the formula describing the property which can be deduced as follows:

$$((income1 = income2) \wedge (age1 = age2) \wedge \neg(gender1 = gender2)) \wedge \neg(class1 = class2)$$

[4] Note that, the property translation mechanism is in detailed described in Chapter 6, and for this Chapter the reader does not need to understand this translation technique.

[5] In this formula each of the lines are conjoined and for brevity we omit the conjunction symbol between the lines.

1 ; male=1, female=0, no=0, yes=1
2 ; Encoding of the tree for instance $\vec{x}_1$
3 $(\text{true} \wedge (gender1 = 1) \wedge n_{11}^{(1)}) \vee ((\text{false} \vee \neg(gender1 = 1)) \wedge \neg n_{11}^{(1)})$
4 $(n_{11}^{(1)} \wedge (income1 < 1000) \wedge n_{11}^{(2)}) \vee ((\neg n_{11}^{(1)} \vee \neg(income1 < 1000)) \wedge \neg n_{11}^{(2)}) \wedge (n_{11}^{(2)} \Rightarrow class1 = 0)$
5 $(n_{11}^{(1)} \wedge (income1 \geq 1000) \wedge n_{21}^{(2)}) \vee ((\neg n_{11}^{(1)} \vee \neg(income1 \geq 1000)) \wedge \neg n_{21}^{(2)}) \wedge (n_{21}^{(2)} \Rightarrow class1 = 1)$
6 $(\text{true} \wedge (gender1 = 0) \wedge n_{21}^{(1)}) \vee ((\text{false} \vee \neg(gender1 = 0)) \wedge \neg n_{21}^{(1)})$
7 $(n_{21}^{(1)} \wedge (age1 < 40) \wedge n_{31}^{(2)}) \vee ((\neg n_{21}^{(1)} \vee \neg(age1 < 40)) \wedge \neg n_{31}^{(2)})$
8 $(n_{31}^{(2)} \wedge (income1 < 1000) \wedge n_{11}^{(3)}) \vee ((\neg n_{31}^{(2)} \vee \neg(income1 < 1000)) \wedge \neg n_{11}^{(3)}) \wedge (n_{11}^{32)} \Rightarrow class1 = 0)$
9 $(n_{31}^{(2)} \wedge (income1 \geq 1000) \wedge n_{21}^{(3)}) \vee ((\neg n_{31}^{(2)} \vee \neg(income1 < 1000)) \wedge \neg n_{21}^{(3)}) \wedge (n_{11}^{32)} \Rightarrow class1 = 1)$
10 $(n_{21}^{(1)} \wedge (age1 \geq 40) \wedge n_{41}^{(2)}) \vee ((\neg n_{21}^{(1)} \vee \neg(age1 \geq 40)) \wedge \neg n_{41}^{(2)})$
11 $(n_{41}^{(2)} \wedge (income1 < 5000) \wedge n_{31}^{(3)}) \vee ((\neg n_{41}^{(2)} \vee \neg(income1 < 5000)) \wedge \neg n_{31}^{(3)}) \wedge (n_{31}^{32)} \Rightarrow class1 = 0)$
12 $(n_{41}^{(2)} \wedge (income1 \geq 5000) \wedge n_{41}^{(3)}) \vee ((\neg n_{41}^{(2)} \vee \neg(income1 \geq 5000)) \wedge \neg n_{41}^{(3)}) \wedge (n_{41}^{32)} \Rightarrow class1 = 1)$
13
14 ; Encoding of the tree for $\vec{x}_2$
15 $(\text{true} \wedge (gender2 = 1) \wedge n_{12}^{(1)}) \vee ((\text{false} \vee \neg(gender2 = 1)) \wedge \neg n_{21}^{(1)})$
16 $(n_{12}^{(1)} \wedge (income2 < 1000) \wedge n_{12}^{(2)}) \vee ((\neg n_{12}^{(1)} \vee \neg(income2 < 1000)) \wedge \neg n_{12}^{(2)}) \wedge (n_{12}^{(2)} \Rightarrow class2 = 0)$
17 $(n_{12}^{(1)} \wedge (income2 \geq 1000) \wedge n_{22}^{(2)}) \vee ((\neg n_{12}^{(1)} \vee \neg(income2 \geq 1000)) \wedge \neg n_{22}^{(2)}) \wedge (n_{22}^{(2)} \Rightarrow class2 = 1)$
18 $(\text{true} \wedge (gender2 = 0) \wedge n_{22}^{(1)}) \vee ((\text{false} \vee \neg(gender2 = 0)) \wedge \neg n_{22}^{(1)})$
19 $(n_{22}^{(1)} \wedge (age2 < 40) \wedge n_{32}^{(2)}) \vee ((\neg n_{22}^{(1)} \vee \neg(age2 < 40)) \wedge \neg n_{32}^{(2)})$
20 $(n_{32}^{(2)} \wedge (income2 < 1000) \wedge n_{12}^{(3)}) \vee ((\neg n_{32}^{(2)} \vee \neg(income2 < 1000)) \wedge \neg n_{12}^{(3)}) \wedge (n_{12}^{32)} \Rightarrow class2 = 0)$
21 $(n_{32}^{(2)} \wedge (income2 \geq 1000) \wedge n_{22}^{(3)}) \vee ((\neg n_{32}^{(2)} \vee \neg(income2 < 1000)) \wedge \neg n_{22}^{(3)}) \wedge (n_{12}^{32)} \Rightarrow class2 = 1)$
22 $(n_{22}^{(1)} \wedge (age2 \geq 40) \wedge n_{42}^{(2)}) \vee ((\neg n_{22}^{(1)} \vee \neg(age2 \geq 40)) \wedge \neg n_{42}^{(2)})$
23 $(n_{42}^{(2)} \wedge (income2 < 5000) \wedge n_{32}^{(3)}) \vee ((\neg n_{42}^{(2)} \vee \neg(income2 < 5000)) \wedge \neg n_{32}^{(3)}) \wedge (n_{32}^{32)} \Rightarrow class2 = 0)$
24 $(n_{42}^{(2)} \wedge (income2 \geq 5000) \wedge n_{42}^{(3)}) \vee ((\neg n_{42}^{(2)} \vee \neg(income2 \geq 5000)) \wedge \neg n_{42}^{(3)}) \wedge (n_{42}^{32)} \Rightarrow class2 = 1)$
25
26 ;Encoding of the property
27 $(income1 = income2) \wedge (age1 = age2) \wedge \neg(gender1 = gender2)$
28 $\neg(class1 = class2)$

Figure 5.2: Logical encoding of the fairness property and the decision tree model

The idea is to find an input to the model which satisfies the pre-condition, however, the output corresponding to which fails to satisfy the post-condition, thus violating the property. We conjoin this constraint to the model formula in Lines 27-28 of Figure 5.2. For solving the entire formula we use the SMT solver by giving the formula to the solver in the appropriate form (i.e., in SMTLib format [6]). Given this formula to the solver, we indeed could find a counter-example which is returned as a *logical model* of the formula giving the values of the features corresponding to the instances ($\vec{x}_1$ and $\vec{x}_2$) for which the decision tree model violates the fairness property. Figure 5.3 shows the logical model of the formula as returned by the SMT solver Z3 [MB08] which essentially matches with the counter-example we showed for Definition 5.1.

As we can see, the counter-example in this case is a pair of the form $((\vec{x}_1, y_1), (\vec{x}_2, y_2))$ where y_1 and y_2 are the classes as predicted by the decision tree for the instances $\vec{x}_1$ and $\vec{x}_2$ respectively. We can basically consider y_1 as $M'(\vec{x}_1)$ and y_2 as $M'(\vec{x}_2)$. Since the white-box model learning step (described in Section 5.2.1) is imprecise, the counter-example thus generated on the inferred model might not be valid for the MUT. Hence, we next check the validity of the counter-example on the MUT and find out whether $M(\vec{x}_1) = y_1$ and $M(\vec{x}_2) = y_2$, i.e., if we get the same prediction for the $\vec{x}_1$ and $\vec{x}_2$

[6] http://smtlib.cs.uiowa.edu

```
sat (model
  (define-fun age1 () Int 50)
  (define-fun income1 () Real 4500.0)
  (define-fun gender1 () Int 0)
  (define-fun class1 () Int 0)
  (define-fun age2 () Int 50)
  (define-fun income2 () Real 4500.0)
  (define-fun gender2 () Int 1)
  (define-fun class1 () Int 1)
```

Figure 5.3: Logical model corresponding to the formula of Fig. 5.2

from the MUT as we got from the inferred decision tree model. If yes, then we return this as a valid counter-example to the property, otherwise, we add $(\vec{x}_1, M(\vec{x}_1))$ and $(\vec{x}_2, M(\vec{x}_2))$ to the oracle data in order to improve the white-box model by retraining. However, just by adding such a single counter-example to the oracle data and retraining to generate a new white-box model again is an ineffective process and thus, we next propose a technique called *pruning* in order to generate a number of counter-examples.

If the SMT solver does not return any counter-example, we conclude that the generated white-box model satisfies the corresponding property, however not necessarily the MUT and thus, our verification-based testing technique is unable to find any test cases with respect to the property we are checking.

5.2.3 Pruning

In this step, we essentially aim to generate a number of counter-examples given an initial counter-example and retrain the white-box model by including them in the oracle data. The generation of a number of such instances as counter-examples requires to use the SMT solver in a specific way. More specifically, we need several logical models to be returned by the solver for the same satisfiable query. For this, we propose two pruning techniques and the idea herein is to find out more counter-examples by (a) repeatedly *negating* the *interpretation* (i.e., the values) of the instance variables for the current one and (b) searching for counter-examples in different *regions* of the input space. Next, we describe these two techniques in more detail.

Pruning data instance. We start by using the feature values as returned by the SMT solver as a counter-example. The idea herein is to simply negate the feature values which are currently returned by the SMT solver as a logical model and conjunct it to the original logical formula. For instance, in our running example, by using the logical model depicted in Figure 5.3, we can negate the feature value for $age1$, by simply adding the constraint $(\neg(age1 = 50))$ to the logical formula in Figure 5.2 and giving it to the SMT solver. As a result, the solver would then return a different satisfiable example to the formula with the $age1$ variable assigned to 51. In this way, we can generate a large number of counter-examples by simply disallowing a feature value in the previously generated counter-example. Furthermore, we can do this for all the other features as well (except for the *gender* feature since this is binary). A similar sort of approach is used by Udeshi et al. [UAC18] where they used a searching algorithm to generate more counter-examples as violated test cases from an initial set of test cases. In our case, the search is performed by the SMT solver on the logical formula.

In Algorithm 5, we describe this pruning approach which sort of prunes the search

Algorithm 5 *prunInst* (Pruning data instances)

Input: $\vec{x}$ ▷ *candidate instance*
φ ▷ *Logical formula*
Output: set of candidate instances
1: cand-set:= $\emptyset$;
2: **for** $i := 1$ to n **do** ▷ *n: number of features*
3: $\psi := \varphi \wedge \neg(\text{name}_i = \vec{x}(i))$;
4: **if** SAT(ψ) **then**
5: cand-set := cand-set $\cup$ get-model(ψ);
6: **return** cand-set;

Algorithm 6 *prunBranch* (Pruning branches)

Input: $\vec{x}$ ▷ *candidate instance*
tree ▷ *Decision tree*
φ ▷ *Logical formula*
Output: set of candidate instances
1: cand-set:= $\emptyset$;
2: $(c_1, \ldots, c_m)$:= getPath(tree,$\vec{x}$); ▷ *Path of $\vec{x}$ in tree*
3: **for** $i := 1$ to m **do** ▷ *toggle path conditions*
4: $\psi := \varphi \wedge \neg c_i$;
5: **if** SAT(ψ) **then**
6: cand-set := cand-set $\cup$ get-model(ψ);
7: **return** cand-set;

space. As inputs, the algorithm gets the candidate instance $\vec{x}$ which is the initial counter-example, and the logical formula φ describing the conjunction of the model and the (negated) property. Next, we negate the feature values as returned by the SMT solver in $\vec{x}$, by performing $\neg(\text{name}_i{=}\vec{x}(i))$ and conjunct it to the formula φ in line 3, where, name_i denotes the name of the i-th feature, for e.g., $age1$. If we find a satisfiable example for this formula, we add it to the candidate set of test instances (Line 5) [7]. This process is then repeated for all the feature values of instance $\vec{x}$ and finally the algorithm returns a set of candidate instances. Note that, in our running example, instead of a single instance we get a pair of instances as counter-example, and thus, we extend the loop from lines 2-5 to repeat the process for the instance $\vec{x}_2$ as well.

The disadvantage of this pruning approach is that the SMT solver would give counter-examples only considering a specific region of the input space. Therefore, we might not be able to find counter-examples that reside in different regions of the input space of the MUT. To solve this problem, we consider a different pruning approach which we describe next.

Pruning branches. This strategy can only be used for the tree-based algorithms and therefore, we can apply this pruning mechanism only when the chosen white-box model approximating the MUT is the decision tree. Unlike the previous pruning approach which attempts to find the counter-examples *locally*, here we search for more

[7] Note that, we call these set of instances as *candidate* set, since we do not know at this point whether they are valid or invalid counter-examples on the MUT.

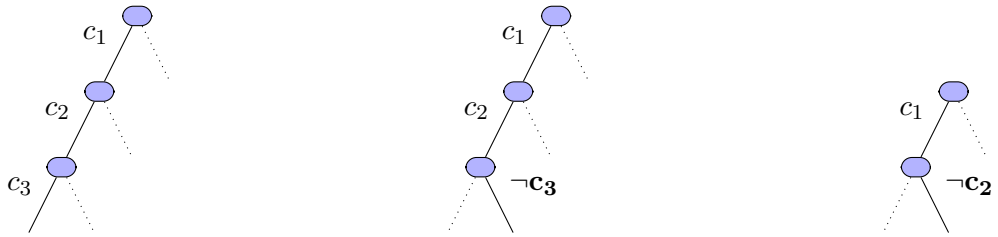


Figure 5.4: Illustration of condition toggling

counter-examples *globally*, by traversing as many paths of the decision tree as possible. Algorithm 6 describes this pruning approach. To this end, the algorithm is first called with the initial counter-example as the instance $\vec{x}$, the tree model (tree) and the logical formula φ describing the tree model and the conjunction of (the negation of) the property. Next, we identify the path in the tree which is taken by the instance $\vec{x}$ (Line 2). Such a path in the tree is essentially a collection of conditions on the feature values denoted as $(c_1, \ldots, c_m)$, where each c_i is of the form $\vec{x}(i) \sim d_i$, and $\sim$ is a conditional operator from the set $\{<, \leq, \geq, >, =\}$, and $d_i \in X_i$. Next, we *toggle* each of these conditions, one after the other. Essentially, we take the negation of a condition and conjunct it to the formula φ, and thus, at iteration i, we have the formula as, $\varphi \wedge \neg c_i$ which is then given to the SMT solver (Lines 3-5). If we find a satisfiable example to the formula, we add it to the set of candidate instances as potential counter-examples. Furthermore, in our running example property, we essentially get two branches for $\vec{x}_1$ and $\vec{x}_2$, and thus, we execute the loop from lines 3-6 in Algorithm 6 twice.

Let us consider Figure 5.4 (taken from a previous work of ours [SW20b]) where we illustrate the workings of the condition toggling approach. Suppose, we get the decision path corresponding to the instance $\vec{x}_1$ as depicted in the leftmost branch of the Figure 5.4. Here we have three conditions c_1, c_2, and c_3 (in a top-down manner) which denote the conditions $gender1$ =female, $age1 \geq 40$, and $income1 < 5000$ respectively, corresponding to the tree in Figure 5.1. We get this branch corresponding to the counter-example $\vec{x}_1$ as `gender1=0,age1=50, income1=4500.0` [8]. We start with the bottommost constraint c_3, which we negate and conjunct to the formula φ as, $\varphi \wedge \neg c_3$. This formula is then given to an SMT solver which then attempts to find a counter-example considering a different path of the tree (depicted as the second path in Figure 5.4), and if a counter-example is found, we add it to the set of candidate-instances.

In this way, we further generate the formula $\varphi \wedge \neg c_2$, and perform satisfiability solving in order to find a new counter-example. We then repeat the same procedure (loop from Lines 3-6 of Algorithm 6) for the instance $\vec{x}_2$ as well. It is possible to use several other strategies to perform toggling and moreover, such condition toggling approach is already used in a well-established testing technique, called concolic testing [SMA05]. The idea therein is to use the condition toggling to traverse a new path in the program. The branch pruning approach we describe here, however, is in a spirit similar to the approach proposed by Aggarwal et al. [Agg18] where they used such toggling technique on a decision path (learned through machine learning techniques) to generate test cases.

In summary, while the instance pruning approach works locally by generating counter-examples negating the feature values of an instance, the branch pruning technique works globally by negating the branch conditions of the tree. Hence, these local and

[8] As mentioned beforehand, here the gender value 0 means female.

Algorithm 7 *veriTest* (Verification-based testing)

Input: M ▷ *Model under test*
φ_{prop} ▷ *Property in logical formula*
Output: counter example to the property or unknown

```
1: orcl_data := ∅; ts := ∅; cs := ∅;
2: while |orcl_data| < MAX_ORCL do
3:     x⃗:= random(X⃗);
4:     if (x⃗, M(x⃗)) ∉ orcl_data then
5:         orcl_data := orcl_data ∪ {(x⃗, M(x⃗))};
6: while not TIMEOUT do
7:     model := trainModel(orcl_data);
8:     φ := model2Logic(model);
9:     φ := φ ∧ ¬φ_prop;
10:    if UNSAT(φ) then
11:        return Unknown;
12:    (x⃗, y) := getModel(φ);
13:    if (x⃗, M(x⃗)) ⊭ prop then                ▷ A valid CEX is found
14:        return (x⃗, M(x⃗));
15:    cs := {(x⃗, M(x⃗)};
16:    if type(model) = tree then
17:        cs := cs ∪ prunBranch(x⃗, φ);
18:    cs := cs ∪ prunInst(x⃗, φ);
19:    for (x⃗, M(x⃗)) ∈ cs do
20:        if (x⃗, M(x⃗)) ⊭ prop then
21:            return (x⃗, M(x⃗));
22:        orcl_data := orcl_data ∪ {(x⃗, M(x⃗))};
23: return Unknown;
```

global approaches of counter-example generation techniques in combination complement each other and thereby give an effective approach to generating a number of counter-examples from a single SMT formula.

5.2.4 Cross Checking and Retraining

After generating a set of counter-examples as test inputs by using two pruning strategies, we need to check the *validity* of those on the MUT. Since we get the counter-examples on the inferred white-box model, it might happen that the MUT does not conform to these test cases. Therefore, we perform cross-checking of the test inputs and to this end, we can have two different scenarios.

Property violated. If any of the generated test instances on the white-box model also conforms to be a counter-example to the property under test on the MUT, we are done with the testing and return the violated instance as a proof of the property violation.

Property not violated. Since the white-box model is a mere approximation of the MUT, it might happen that none of the generated counter-examples shows the

violation on the MUT even though the MUT might not guarantee the property. Therefore, we need to improve the approximation quality by retraining the white-box model with the invalid set of counter-examples.

5.2.5 Overall Algorithm

Algorithm 7 summarizes all the steps of our testing mechanism we have discussed so far. First of all, in Lines 2-5 we randomly generate a number of data instances based on a *user-controllable* parameter MAX_ORCL to construct our oracle data. Then we train an ML algorithm (either a decision tree or a neural network) on the oracle data (Line 7) to generate the model inferring the MUT. After that, the model is converted to the logical formula φ, and the negation of the property under test $\neg\varphi_{prop}$ is conjoined to that formula (Lines 8-9). We then check for the satisfiability of the entire formula and if we find it to be unsatisfiable we return UNKNOWN, since we cannot say for sure whether the MUT satisfies the property. However, if we find a satisfiable example of the formula as $(\vec{x}, y)$, we return it as a candidate counter-example to the property on the MUT. Next, we check it with the MUT and if it is valid, we return it as the counter-example to the property, and our job is done (Lines 13-14). However, in case it is not a valid counter-example on the MUT, we store it, and proceed to generate more counter-examples by using the two pruning approaches.

Since the branch pruning strategy can only be applied to the tree-based models, next we check the *type* of the inferred white-box model, and if it is a decision tree, then we apply the branch pruning to generate a number of counter-examples from the initial one (Lines 16-18). However, regardless of the model type, we can always apply the instance pruning strategy, and thus, we next generate more test inputs as counter-examples by applying this technique. After collecting a set of such instances in the candidate set *cs*, we check if any of the instances in this set is also a counter-example to the MUT and if so, we return it, otherwise, we add that instance to the oracle data (Lines 19-22). This process repeats until we find a violation of the property on the MUT or a user-defined TIMEOUT occurs.

To conclude, in this chapter, we have described the verification-based testing technique which can be used as the test case generation technique for the ML models. To this end, we have described the steps involved in this testing process with the necessary algorithms. Next, we extend this approach by including a property specification mechanism that would allow the tester to specify any property to develop the property-driven testing approach. The property would then be used by the verification-based testing technique to generate test cases automatically.

6 Property-driven Testing

There is a growing importance to ensuring the quality of machine learning-based software or more specifically the ML models. To this end, a number of properties exist in various application domains which need to be satisfied by these models. Researchers are trying to mitigate this issue either by designing a specific property-guaranteed algorithm, such as the *fairness-aware* algorithms [ZVGG17, JSW22], or by developing verification or testing mechanisms to validate properties on the models already trained on the training datasets [OBK22, BSS+19, ALN+19]. The latter techniques, however, either focus on a specific type of property or on a specific type of model. Therefore, in a scenario when we do not have any knowledge about the model under test and furthermore, we need to test several types of properties on the given model, we need a testing mechanism irrespective of the model and properties.

To this end, there already is a well-known testing approach called *property-based* testing [CH00] (PBT) which can be used for testing programs with respect to user-specified properties. It was initially designed to perform unit testing of programs written in the Haskell programming language and later adopted for other programming languages (such as Java and Python) as well. A property in this testing approach is specified typically over the inputs and the outputs of the program under test. Given a specific input constraint, in this case, the program must satisfy the output constraint. Based on this specification, PBT then automatically generates test cases satisfying the input constraint, which are then executed on the program under test. For each execution, the output is recorded and checked to find out whether a violation of the output constraint has occurred, if yes, the violated test case is returned as the counter-example to the specified property.

The generation of the test cases based on the given input constraint is however performed randomly in this approach and the property as a whole is not taken into consideration. To alleviate the problem of random data generation in property-based testing, there are some works that propose alternative approaches to generating test inputs, such as coverage guided fuzzing [LHP19] or search-based approaches [LS18]. However, they either require white-box access to the program under test, for instance, to get information about specific paths in order to evaluate path coverage or, do not consider the property while generating the test cases.

We propose the *property-driven testing* (PDT) approach which tackles this issue by using the property to generate test cases in a targeted manner, thereby allowing for a more systematic test generation process compared to random test input generation. Here, given the property specified in a particular format, we adapt our verification-based testing approach (described in Chapter 5) to use the property to generate test cases in order to find a violation. Essentially, we employ a form of *learning-based* testing to learn an ML model approximating the model under test and then applying the satisfiability solving technique by taking into account the property to generate test cases. Since our approach does not depend on the input model, we can apply our approach to test any machine learning model as well as *programmed* functions

(see Section 7.2.3 of Chapter 7 for more details). Furthermore, the employment of a property specification mechanism allows us to test the models (or even functions) with respect to user-specified properties, provided in a novel, domain-specific specification language.

In this chapter, we start with some basic formalizations and then describe our property specification language (Section 6.1). We furthermore give details about how we use the property to generate test cases (Section 6.2) and then finally end the chapter by discussing some related works (Section 6.3).

6.1 Property Specification Language

First, we revisit [1] the basic formalizations that are needed to describe the concepts of this Chapter. To this end, we first define the machine learning model to be a predictive function taking the following form:

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

The X_i is defined as the value set of the feature i and $1 \leq i \leq n$, and Y is defined the set of classes. We denote $\vec{X}$ for $X_1 \times \ldots \times X_n$ and $\vec{x}$ as an input feature vector to M and $\vec{x} \in \vec{X}$, $y \in Y$. Assuming the input feature vector $\vec{x}$ consisting of n number of elements, then each of these is denoted as $\vec{x}(1), \ldots, \vec{x}(n)$. An element $\vec{x}(i)$ is also called the i-th *argument* (or i-th feature) of the input instance $\vec{x}$.

In property-based testing, the property to check on a program is specified using a specification language that has the `assume-assert` form. The `assume` part specifies the conditions on the inputs of the program and the `assert` part specifies a condition to be satisfied by the outputs of the program for the inputs generated by satisfying the condition in the `assume` statement. We follow a similar style of specifying properties in our property-driven testing approach. However, unlike the property-based testing approach, we do not use the specification as *executable* predicate [2], rather, the constraints corresponding to the specification of the property are used to generate a logical formula describing the property. This logical formula is then conjoined with the logical formula of the *white-box* model (approximating the model under test). Then a solver is applied to the entire formula to find a violation of the property.

To this end, we require the property specification to be composed of two parts: (a) the condition specifying the constraint on the input and the output of the model, and (b) the numerical values (if required) corresponding to the conditions in order of their syntactical occurrence. In a more technical sense, we use two *functions* `Assume` and `Assert` for specifying a property that takes the following form:

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

The `Assume` part specifies the conditions on the input of the program and the `Assert` specifies the condition on the outputs. The first parameter `<condition>` corresponds to a *string* describing the logical condition specified on the input (in `Assume`) or the output

[1] Note that, we only describe the formalization here which is needed to explain our testing concepts and further details can be found in Section 2.1 of Chapter 2.

[2] An executable predicate is a predicate that is used to determine the validity of some condition in a program and is able to be executed directly by the program.

(in `Assert`) of the model, whereas the rest of the parameters are values corresponding to the variables associated in the logical condition, in order of their first occurrences. Thus, we connect the condition specified as a string with the values to derive a logical formula specifying the property to check. Next, we define the syntax and the semantics of the specification language which is used to specify the logical conditions on the inputs and the outputs of the model under test (MUT).

6.1.1 Syntax and Semantics

The syntax of our specification language defines the rules for constructing valid statements and expressions. We specify these rules via a grammar G, termed as Backus-Naur Form (BNF). Along with the syntax, we also informally describe the semantics of this language which describe the meaning of different statements and the expressions of the language. More specifically, semantics give the rules for interpreting and assigning meaning to different elements of our specification language such as statements, variables, and conditions.

We start with the initial template of our specification language. As mentioned before, our specification language allows specifying conditions on the input of the MUT (using `Assume()`) and on the output (using `Assert()`). Let us take $\langle assume \rangle$ and $\langle assert \rangle$ to denote the conditions specified in our `Assume` and `Assert` functions respectively. The property specification condition $\langle Prop \rangle$ (starting symbol of our grammar G) in our approach can be described using the following basic form:

$$\langle Prop \rangle ::= [\forall \vec{x}_1, \ldots, \vec{x}_m]\ [\forall c \in R]\ \langle assume \rangle \Rightarrow \langle assert \rangle$$

In the grammar [..] refers to optional parts, | stands for alternatives, and $\langle .. \rangle$ are used for non-terminal symbols. Note that, the properties we consider might need a single input instance and the corresponding output value to compute the property or in the case of *hyper-properties* [CS10], we might require multiple input instances and their corresponding output values to compute the property. A typical example for the latter is the monotonicity property where we require a pair of input instances $\vec{x}_1, \vec{x}_2$ and their corresponding output values $M(\vec{x}_1), M(\vec{x}_2)$ to define the property.

To take care of such kind of hyper-properties, we allow writing the universal quantifiers $\forall \vec{x}_1, \ldots, \vec{x}_m$ specifying the number of inputs required for a property to check, in front of the property specification. Moreover, we might want a property to be valid for all the values c (integer or real) in a specific range R and this is denoted as $\forall c \in R$. The final part of the specification contains conditions on the inputs of the MUT as $\langle assume \rangle$ and the output condition as $\langle assert \rangle$. The implication operator ($\Rightarrow$) in between these two defines if the $\langle assume \rangle$ condition is evaluated to be true, then the condition in $\langle assert \rangle$ must hold too. Next, we describe the production rules of G to further derive these two conditions.

$$\langle assume \rangle ::= \langle assume \rangle\ \langle bool\text{-}op \rangle\ \langle assume \rangle \mid [\langle arg\text{-}quant \rangle]\ \langle cond \rangle$$

$$\langle bool\text{-}op \rangle ::= \wedge \mid \vee \mid \Rightarrow \mid \Leftrightarrow$$

$$\langle assert \rangle ::= \langle cond \rangle$$

The specification in $\langle assume \rangle$ can be derived to a single condition prefixed with the optional part defining the number of input arguments ($[\langle arg\text{-}quant \rangle]$) or, it can be

further connected to more $\langle assume \rangle$ conditions through a Boolean operator (denoted as $\langle bool\text{-}op \rangle$). For this, we allow the use of standard Boolean operators $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$ to connect two assume conditions.

Note that, certain properties might require specifying conditions only on specific elements of the input instance $\vec{x}$ and thus, we use $[\langle arg\text{-}quant \rangle]$ to define the set of such elements. We write it as:

$$\langle arg\text{-}quant \rangle ::= \forall i \in I \mid \exists i \in I$$

where I defines a fixed set of input elements. For instance, we formally write the monotonicity property we currently take as a running example as:

$$\forall \vec{x}_1, \vec{x}_2 \in \vec{X} : \forall i \in \{1, \ldots, n\} : \vec{x}_1(i) \leq \vec{x}_2(i) \Rightarrow M(\vec{x}_1) \leq M(\vec{x}_2)$$

To define this property we need a conjunction of a number of $\langle assume \rangle$ conditions defined on each of the input arguments as $\vec{x}_1(i) \leq \vec{x}_2(i)$ and the set of input elements I, which in this case are all the elements $\{1, \ldots, n\}$ of the input. The quantification over the input elements, along with the conjunction of several $\langle assume \rangle$ conditions thus allow us to define such kind of properties.

We define the atomic part ($\langle cond \rangle$) of the $\langle assume \rangle$ and the $\langle assert \rangle$ conditions as follows:

$$\langle cond \rangle ::= \langle cond \rangle \ \langle bool\text{-}op \rangle \ \langle cond \rangle \mid \neg \ (\langle cond \rangle) \mid \langle pred \rangle$$

$$\langle pred \rangle ::= \text{true} \mid \text{false} \mid \langle term \rangle \ \langle comp\text{-}op \rangle \ \langle term \rangle$$

$$\langle comp\text{-}op \rangle ::= < \mid > \mid = \mid \leq \mid \geq \mid \neq$$

Essentially, the conditions in $\langle assume \rangle$ and $\langle assert \rangle$ are predicates defined on the inputs and the outputs of the MUT respectively, hence, a single such condition can be connected to another by using different Boolean operators ($\langle bool\text{-}op \rangle$) which we described beforehand. Along with that, we allow a single condition to be negated $\neg(\langle cond \rangle)$ as well.

The predicates in the specification condition ($\langle pred \rangle$) can be of various types depending on the condition required to be specified. For instance, it can be simply 'true' or 'false', or a logical constraint defined over two *terms* ($\langle term \rangle$) connected via a comparison operator ($\langle comp\text{-op} \rangle$). The comparison operators we use here have their usual meanings, for instance, a predicate involving two terms connected via the *less than or equal to* operator ($\leq$) should be evaluated to be true if the left side of this operator is essentially having a lesser or equal value compared to the term on the right side.

The terms in the predicate can be of different types which are described as follows:

$$\begin{aligned} \langle term \rangle ::= \ & z \mid r \mid \vec{x}_j(i) \mid M(\vec{x}_j) \mid \langle term \rangle \ \langle bin\text{-}op \rangle \ \langle term \rangle \\ \mid \ & \langle una\text{-}op \rangle \ \langle term \rangle \mid \langle vec\text{-}op \rangle \ \vec{x}_j \end{aligned}$$

Accordingly, each term in the predicate can be an integer (z) or, a real (r), valued number, an input argument of a specific (j-th) input instance $\vec{x}_j(i)$ or denoting the output corresponding to that instance, $M(\vec{x}_j)$ [3]. The term $M(\vec{x}_j)$ can however only

[3] Note that, if we have more than one output, then we can use $M(\vec{x}_j)(k)$ to denote the k-th output variable. For brevity, here we discuss the syntax assuming that the model has a single output.

occur in ⟨*assert*⟩ condition. Moreover, each term can further be combined with other terms through arithmetic operators ⟨*bin-op*⟩, described as follows:

⟨*bin-op*⟩ ::= + | - | * | \

Thereby, we allow deriving a complex expression as a specific term in the predicate involving these arithmetic operators. Apart from these operations, our specification also takes care of specific *unary* operations on the elements of the input instance. For instance, we allow writing *absolute* value or the modulus operation. This is defined by using the unary operator symbol ⟨*una-op*⟩.

As an example, let us consider the Lipschitz property [CMN+19] which requires that the (Manhattan) distance between any two input vectors to the model must be greater than or equal to the difference of the outputs for two corresponding vectors. Formally, this property has the following form:

$$\forall \vec{x_1}, \vec{x_2} : true \Rightarrow c * \sum_{i=1}^{n} abs(\vec{x}_1(i) - \vec{x}_2(i)) \geq abs(M(\vec{x_1}) - M(\vec{x_2}))$$

The constant c is specific with respect to the MUT. The *abs* operator denotes the absolute value operation. On the left side of the $\geq$ operator, we specify the distance measure between two input instances $\vec{x}_1$ and $\vec{x}_2$. The possibility of using a combination of several arithmetic operators and the unary operator *abs* in our specification mechanism therefore, allows specifying such a complex property.

Finally, some specific properties require to specify conditions on the minimum or maximum input argument value of the input instance. For this, we use ⟨*vec-op*⟩ to be applied on the input instance which can be derived as follows:

⟨*vec-op*⟩ ::= `min` | `max`

For instance, when $min(\vec{x})$ is specified on the input instance $\vec{x}$, the minimum value of all the arguments of $\vec{x}$ is considered. The maximum value operation *max* refers to the maximum of all the input arguments in the input vector. To exemplify such a case, some specific regression models (see details about such models in Section 7.2.3 of Chapter 7) require to satisfy the *disjunctive* property which requires the model output to be greater or equal to the maximum value of the input arguments. It can formally be described as:

$$\forall \vec{x} : true \Rightarrow \max(\vec{x}) \leq M(\vec{x})$$

Thus, the ⟨*vec-op*⟩ operations on the input vector allow to specifying such kind of properties by using our specification language.

6.2 Test Input Generation

Once we have the property specified in the appropriate format, the next step in our property-driven testing approach is to employ our verification-based testing technique to generate test cases on the given MUT. Figure 6.1 gives an overview of our test data generation approach. In step 1, first of all, a known ML model (either a decision tree or a neural network) is learned from the given MUT. Since the internals of the model are known to us, we call it a *white-box* model. This in the next step (step 2)

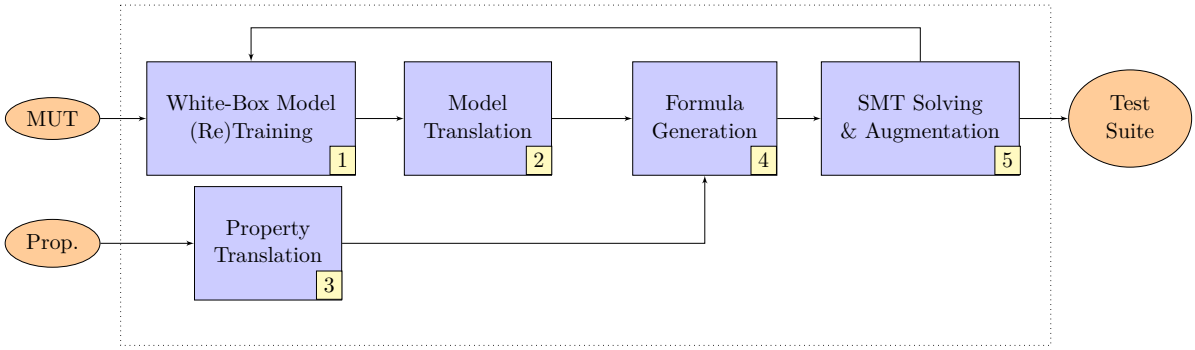


Figure 6.1: Workflow of Test Data Generation Approach in Property-driven Testing

is translated to a logical formula. To compute the property on the learned white-box model, we however also need the property in the form of a logical constraint.

In the previous section, we described how our property specification language can be used to specify properties. However, to check the property on the learned white-box model and thereby on the MUT, we still need to perform two steps: (step 3) translating the property specification into a logical formula, and (step 4) connecting this formula to the logical formula describing the model and thus generating a conjoined formula. Once, we are done with these two steps, we apply the satisfiability modulo theory (SMT) solving technique to generate test cases on this conjoined formula (step 5).

Next, we describe the method for translating the specified property into the logical formula and furthermore the method to connect it to the logical formula of the model.

6.2.1 Connecting to Model Encoding

Before describing the mechanism for translating the property in logical constraints, we describe how we connect the translated property to the logical formula describing the white-box model φ_{model}. More specifically, we describe how to adapt the encoding of the white-box model based on the property specification, in order to analyze the property on the model. To this end, if a property requires a single call to the MUT, thereby requiring a single input instance and the corresponding output value to check the property, a singly encoded model suffices. However, for hyper-properties, we require more than one input instance and output value pairs to compute the property.

In our specification language, we allow for the specification of hyper-properties by defining universal quantification over several input instances (i.e., $\forall \vec{x}_1, \ldots, \vec{x}_m$). To connect such property to the logical formula of the model φ_{model}, we create one instance of this formula for each input instance defined in the given property specification (indicated by $\forall \vec{x}_i$). To do so, we create φ_{model} as $\varphi^1_{model}, \ldots, \varphi^m_{model}$ formulas with respect to each input $\vec{x}$ in $\forall \vec{x}_1, \ldots, \vec{x}_m$. As described beforehand (see Chapter 3), the encoded formula of the model φ_{model} contains constraints over the input arguments $\vec{x}(i)$ of the input instance as well as over the variables describing the outputs of the model. Hence, to create m copies of the model encoding, we simply name the input arguments and the output variables based on the copy j. For instance, we define the i-th input argument of the j-th instance $\vec{x}_j$ as x^j_i and analogously for output variables. This process is furthermore repeated for all the other variables used in the logical formula of the model. Thus, we create m copies of the model encodings. Finally, the conjunction of all these model encodings gives us the final formula describing the

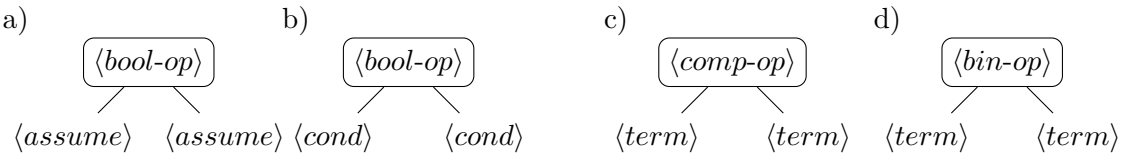


Figure 6.2: Parse trees for parsing expressions describing specification conditions

model as follows:

$$\varphi_{model} \equiv \varphi^1_{model} \wedge \ldots \wedge \varphi^m_{model}$$

Once the model encoding is done, we encode the property into logical formula. To this end, our property encoding consists of two parts, the logical formula describing the *assume* condition φ_{assume}, and the logical formula describing the *assert* condition φ_{assert} [4]. Thus we can write the property specification using the following logical formula:

$$\varphi_{prop} \equiv \varphi_{assume} \Rightarrow \varphi_{assert}$$

However, since our aim is to find a counter-example falsifying the property, we take the negation of the specification. Thus, we derive the negation of the logical formula of the property $\varphi_{\neg prop}$ as follows:

$$\varphi_{\neg prop} \equiv \neg(\varphi_{assume} \Rightarrow \varphi_{assert}) \equiv \neg(\neg\varphi_{assume} \vee \varphi_{assert}) \equiv \varphi_{assume} \wedge \neg\varphi_{assert}$$

Thus, we conjoin the logical formula describing the *assume* and the negation of the *assert* conditions. Using this constraint allows us to find a counter-example to the property on the white-box model. We next describe the translation mechanism of these two conditions in logical formulas.

6.2.2 Property Translation

Given the property specification in the *assume* and *assert* form following the grammar described beforehand (Section 6.1.1), we next translate the property into logical formulas (or more specifically into SMTLIB [5] format). The translation of the specification conditions is done by breaking down the predicates of the conditions and then rebuilding it using the valid operators in SMTLIB. In our specification language, we have predicates involving standard arithmetic, Boolean, and logical comparison operators which are directly available in the SMTLIB and, specific operators such as *minimum*, *maximum*, or *absolute* value functions for which we employ customized functions.

We start with the method of converting simple specification conditions written in *assume* or *assert*. Since we require the expression to be in prefix notation (also termed as Polish notation) for the SMTLIB format, we use regular expressions to parse the expression of the specification condition and convert it to prefix form. This allows breaking expressions into its atomic components. Hence, we can use it to convert our property specification into the desired format. Precisely, the parsing of the regular expression generates an output as a tree with the stack of operations. The depth-first

[4] For brevity, instead of ⟨*assume*⟩ and ⟨*assert*⟩, we now call them *assume* and *assert* respectively.
[5] http://smtlib.cs.uiowa.edu

search (DFS) of such a tree would then give us the prefix expression we require, while preserving the order of the operations.

The Figure 6.2 a–d, shows different parse trees for parsing different types of specification conditions.

a) On the highest level, we have different *assume* statements which get connected via Boolean operators.

b) Each *assume* condition can be a single predicate ($\langle cond \rangle$) or a combination of predicates, which can be further connected through different Boolean operators ($\langle bool\text{-}op \rangle$).

c,d) The basic components of each condition are then terms that can be either connected by using comparison operators ($\langle comp\text{-}op \rangle$) or be combined using arithmetic operators such as $+, -, *$.

Note that, a specification condition can be really complex, for instance, a term can further be a large arithmetic expression of the form $\langle term \rangle \langle bin\text{-}op \rangle \langle term \rangle \langle bin\text{-}op \rangle \ldots \langle term \rangle$. We can first take each atomic part of this expression of the form $\langle term \rangle$ $\langle bin\text{-}op \rangle$ $\langle term \rangle$ and create a parse tree similar to the one in Figure 6.2 and then proceed on to generate a large tree with a number of nested basic parse trees. For brevity, we just give here the sketch of the process for translating the specification condition into our desired format by using the atomic cases.

While parsing any of the trees, we first visit the root node and then the left sub-tree and the right sub-tree, thereby generating an expression in prefix notation (which we require for applying the SMT solver). For instance, in the case of the rightmost parse tree in Figure 6.2, by following such a traversal we generate an expression of the form:

$$\langle bin\text{-}op \rangle \langle term \rangle \langle term \rangle$$

Specific Cases. The terms ($\langle term \rangle$) corresponding to the the pre-condition *assume* can either be an integer or a real number or a specific input argument i for the j-th input instance [6] $\vec{x}_j$, defined as $\vec{x}_j(i)$. A specification condition might further depend on one or more of these input arguments ($\forall i \in I$, $\exists i \in I$). Hence, we need to translate such quantification defined over the set I (containing a number of input elements), to give a valid formula for the specified condition. As explained above (in Section 6.1), for each input argument, the second *parameter* of the `Assume` function provides the values of it, and therefore the translation of the universal quantification $\forall i \in I$ for some finite set I describing the argument positions, can be translated as follows:

$$\bigwedge_{n \in I} \varphi_{assume} \wedge \neg \varphi_{assert}[i \mapsto n]$$

Therefore, i is replaced in the translated formula with the fixed values of n from I. Similarly, in case of the existential quantification over some input elements, the formula can be derived as,

$$\bigvee_{n \in I} \varphi_{assume} \wedge \neg \varphi_{assert}[i \mapsto n]$$

[6] Note that, we can have more than one input instances based on the type of property we consider.

Apart from the input arguments, our property specification mechanism further allows to define a free variable with a specific range (if required). For instance, such a specification can be written in form $\forall c \in R : \ assume \Rightarrow assert$ where $R = [a, b]$. The $\forall c \in R$ part with the range $[a, b]$ can be then translated to logical formula as $(a \leq c) \wedge (b \geq c)$.

A single term in our specification condition might also have unary operators applied to it, such as *absolute* value operation, or vector operations *min* or *max* applied on the input as a whole. Since there do not exist any operators to be used directly in SMTLIB, we implement customized functions for them, as described below:

- The absolute value operator is applied to a specific input element and for this we define a custom function in SMTLIB, which can be called when needed and defined as follows:

  ```
  (declare-fun abs ((x Real)) Real
      (ite  (>= x 0) x (- x)))
  ```

  The `ite` operator here works as the traditional `if-then-else` approach which for the above function states, if the value of the variable `x` is greater than or equal to 0, then the returned value of the function would be the value of `x` itself, otherwise, the returned value would be the one with the minus (-) operator applied, which then is simply the positive value of `x`.

- The specification condition might contain *min* or *max* operations to describe the condition on the minimum or maximum values of the input instance. These two operations are also not available in SMTLIB. Therefore, we write individual functions implementing them. For instance, the *min* function in SMTLIB is implemented as follows:

  ```
  (declare-fun min ((x Real) (y Real)) Real
      (ite  (<= x y) x (y)))
  ```

  If the input value x is lesser or equal to the input value of y, then x is returned, otherwise, y. The function implementing the *max* operation works similarly using $\geq$ instead of $\leq$ operation. For brevity, we only show the function for these operations involving two inputs. However, in reality, we have often a high number of arguments in the input instance $\vec{x}$. Thus, we write an extension of the function described above implementing the complete *min* operation (and *max* operation).

Next we exemplify our property translation mechanism.

6.2.3 Property Translation Example

Again we use the monotonicity property as an example. First, we translate the *assume* part of this property to generate the formula φ_{assume}. The *assume* condition in this property is specified as,

$$\forall \vec{x}_1, \vec{x}_2 \in \vec{X} : \ \forall i \in \{1, \ldots, n\} : \ \vec{x}_1(i) \leq \vec{x}_2(i)$$

Here, we have two instances $\vec{x}_1$ and $\vec{x}_2$ of size n which can be denoted in the logical formula by using a simple numbering scheme to rename the input instances as x_1 and

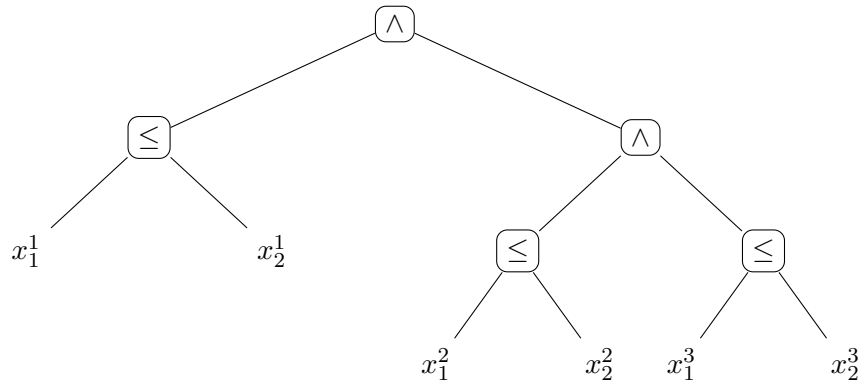


Figure 6.3: Parse tree for parsing the *assume* condition in monotonicity property

x_2 and the i-th elements of the instances as x_1^i and x_2^i respectively. Next, since the specified condition is over all the input elements, we generate n number of corresponding logical constraints for $\vec{x}_1(i) \leq \vec{x}_2(i)$, which are connected via the Boolean logical conjunction operator ($\wedge$). Assuming, $n = 3$, we have three conditions in the *assume*, which are logically conjoined as:

$$(x_1^1 \leq x_2^1) \wedge (x_1^2 \leq x_2^2) \wedge (x_1^3 \leq x_2^3)$$

This forms the logical constraint φ_{assume}. A parse tree for this is depicted in Figure 6.3. By visiting this tree in dfs manner, we generate the expression in SMTLIB format (in a prefix notation) thereby replacing the operators with the appropriate symbols (which are mostly simple keywords) as follows:

```
(and (<= x11 x12) (<= x21 x22) (<= x31 x32))
```

The `and` corresponds to the logical conjunction $\wedge$, and `xji` is the i-th element of the j-th instance ($\vec{x}_j(i)$).

After generating the logical constraint for *assume*, we translate the *assert* part of the property which can be written as: $out_1 = out_2$, where out_1 is the variable denoting the output of the MUT for the instance $\vec{x}_1$ (denoted as $M(\vec{x}_1)$), and similarly out_2 denotes the output corresponding to the instance $\vec{x}_2$ (denoted as $M(\vec{x}_2)$) [7]. However, since our aim in testing is to find a counter-example to the property, we take the negation of the *assert* condition, and we write the formula as $\neg(out_1 = out_2)$ which in SMTLIB is,

```
(not (= out1 out2))
```

Finally, we conjunct the formula φ_{assume} and $\varphi_{\neg assert}$ to create the final formula specifying the negation of the property.

6.2.4 SMT Solving

After translating the property specification into the logical formula, next we conjoin this formula to the logical encoding of the model as $\varphi_{model} \wedge \varphi_{assume} \wedge \neg\varphi_{assert}$ to generate the final SMT formula φ (in Step 4 of Figure 6.1). This is then checked for satisfiability by using the SMT solver Z3 [MB08]. If a satisfiable example to the formula φ is found, we call it the counter-example to the property, and we store this

[7] Note that, we do not give the parse tree for the *assert* condition, since this is quite trivial, containing a node with the $\neq$ operator and two leaf nodes pertaining to $M(\vec{x}_1)$ and $M(\vec{x}_2)$.

and generate more such counter-examples by using the pruning technique (described in Section 5.2.3 of Chapter 5). Next, we check the validity of these counter-examples by executing them on the actual model under test (MUT) and if any one of them is found to be valid, we have found a counter-example to the property (in Step 5 of Figure 6.1) and therefore, we stop by returning the counter-example. Otherwise, we further retrain a new model using the invalid counter-examples to generate a model which better approximates the MUT, thus going back to Step 1) and repeating the subsequent steps. The entire process goes on until a user defined timeout occurs or the SMT solver does not find any counter-example to the property.

6.3 Related Work

In this chapter of our thesis, we gave an approach to test a black-box model with respect to any property, specifiable in a particular format. To this end, we combine verification-based testing (described in Chapter 5) and a property specification mechanism to develop a property-driven testing approach. The idea of verification-based testing approach is closely related to the learning-based testing approach [Ang87, PVY99, MN10], whereas generating test cases based on a given specification stems from the idea of property-based testing [CH00]. Therefore, we discuss the related works published in these two research areas.

Learning-based testing. This is a specific type of testing technique where a *model* is learned from the given *function* under test and then the learned model is used for the test generation process. The model that is being learned can be an automaton [KNR+21, CHJS16], a piece-wise function [MN10], or a machine learning model [PW15]. Note that, the automaton models are more frequently used in the literature and there exists a large body of works to this end. We do not mention them here since we already discussed them in detail in Chapter 2. Here, we mainly focus on the works where a machine learning model is learned in the context of testing.

To this end, our work is probably closest to the work of Meinke et al. [MN10] where given a black-box numeric function, they first learn a piece-wise polynomial function using a set of input-output pairs, obtained by executing a randomly generated set of inputs on the function under test. Once the polynomial function is learned, this is then converted to a logical formula and moreover, the property described in first order logic is conjoined to the former formula. Next, the Hoon-Collins cylindric algebraic decomposition (CAD) algorithm [CJ12] is applied to check the satisfiability of the entire formula. If a satisfiable example as a counter-example to the property is found, it is validated on the original function under test. If valid, the counter-example is returned as a proof of violation to the property, otherwise, it is added to the set of input-output pairs (which was generated initially to learn the polynomial model) and used to modify the inferred model.

The basic methodology of our approach is the same as in the approach of Meinke et al., however, we follow a more systematic method compared to their approach. First, we learn a sophisticated (machine learning) model (from one of the two models) approximating the given model (or a function) under test. To this end, we have the possibility to learn either of the two ML models (decision tree or neural network) which are taken from the state-of-the-art machine learning library `scikit-learn` [PVG+11]. Second, the learned model is checked for satisfiability by using the SMT solver Z3 [MB08] which has been shown to be much more effective and efficient than the CAD algo-

rithm. We moreover use the pruning technique (see Chapter 5) to generate a number of counter-examples to the property under test which allows us to retrain the learned model with not just a single instance, but with a number of instances. Finally and most importantly, we have a property specification approach that allows us to check the given model with respect to any property (as long as it could be specified using our domain-specific language, see Section 6.1).

The work of Briand et al. [BLBS09] proposed the learning of a decision tree from a given function under test. The tree is learned to evaluate the existing test cases (which are already available to test the program) and update them if necessary. They convert the concrete test cases into abstract cases by using the behavioural specification available for the function and then use these abstract test cases to train a decision tree algorithm. The generated tree model is then manually analyzed to find out whether the existing test cases suffice or whether more test cases are needed. In short, they manually analyze the learned decision tree model to extract the coverage information of the existing test cases and add more test cases if required. In contrast, we do not intend to evaluate a set of test cases, rather we learn a decision tree model to analyze the property on the model automatically by using an SMT solver, in order to find a violation.

The work of Papadopoulos et al. [PW15] proposed an approach to learn a decision tree in order to generate test cases. They first learn a decision tree using a set of random input-output pairs (generated using an approach similar to ours) and then apply the Z3 solver to generate test cases covering each branch of the tree. Thus, after this step a list of k inputs (which corresponds to the number of branches in the tree) and their corresponding outputs (based on the decision tree) are generated. Then the outputs are checked with the function under test, if all of them match or a bug is found they stop the process. Note that, the idea herein is to generate test cases by using the inferred model, and each time such test inputs are generated, they execute the test cases on the actual function under test in order to find undesired behaviour. Their experimental evaluation suggests that their test generation process has more coverage than the randomly generated test cases. Our approach, on the other hand, uses the inferred model to analyze the property by using the SMT solver Z3. If a counter-example to the property is found we use a novel technique (pruning) to generate more test cases and all of which are then validated on the original model under test.

Property-based testing. In this testing approach, the tester specifies a property in pre- and post-condition format. Then a test generation process is used to generate test cases (satisfying the pre-conditions), which are then executed on the function under test in order to find a violation of the post-condition specified on the output. The rationale behind this approach is to use the specification to guide the test generation process and automate the testing. To this end, in the literature, we found a number of works that we present next.

Gourley [Gou83] proposed the idea of using a formal specification to guide the test generation process. He discussed the need for using formal language with well defined semantics to define the specification with respect to which the function would be tested. Later, Richardson et al. proposed to use a specification for several tasks in testing [ROT89], such as using the specification to partition the input space and then finding out faults, generating inputs based on a specification, and then checking for faults manually, attempting to force a violation of the oracle as embodied in the specification. Stocks et al. [SC93] proposed to use the Z specification [Spi89] to define test

templates to be used for specification. They discussed the usage of such specifications beyond testing, e.g. for maintenance, refinement of software, and so on.

Khurshid et al. [KM04] proposed TestEra, an automated testing technique for Java programs. The idea therein is that the tester can provide the specification using the specification language framework Alloy [JV00] and then the Alloy-alpha analyzer [JV00] is used to generate a number of test cases (with a bound given by the tester) matching the pre-condition of the specification. Each of these test cases is then executed on the given function to find out the violation specified as post-condition. However, the property specification capability of Alloy is limited and the tester needs to, first of all, understand the specification mechanism (which is a non-trivial task as mentioned by the authors) to specify properties. In a similar sort of approach, Tan et al. [TSL04] proposed to use linear temporal logic (LTL) formula to specify properties and then the usage of the model checking tool to generate test cases. Along with the *type* of properties that can be specified, in this case, the approach is also limited to testing specific programs. In a recent work, Schumi et al. [SS21] proposed the K framework [RS10] to specify properties to be tested for compilers. Essentially, they combined the K framework and a fuzzer [NPS+20] to generate test cases. However, this framework is much more suitable for testing programs pertaining to interpreters, and compilers.

The property specification framework of our approach allows the tester to specify a large variety of properties including hyper-properties which none of the above works are capable of. Our specification language is relatively simple and intuitive since it adheres to the syntax and semantics of any standard programming language. The idea of using such a specification language was first given by Claessen et al. [CH00]. They proposed `QuickCheck` [8], a tool to test programs written in Haskell. In their work, they proposed a simple domain-specific language that allows the tester to specify properties on the input and output variables directly, considering standard Boolean, conditional, and arithmetic operators. Since then, a number of property-based testing tools have been developed for several other programming languages such as in Python [hyp23], or in Java [jqu23]. The main idea behind property-based testing is to use the specification to generate test cases, by executing the predicates defined in the property specification, and thereby finding a violation of the property. Usually, the test case generation method is done either randomly or based on the test generator (defined as a probability distribution of the input). Hence, this method generates test cases without taking the property into consideration. In contrast, in our property-driven testing approach, we use the property to generate test cases in a targeted manner. In the next Chapter, we show the superiority of our testing mechanism over property-based testing, by validating several types of models and functions with respect to a large variety of properties.

To summarize, in this chapter, we described the specification language which we use to specify properties in our testing mechanism. We gave the syntax and the semantics of the language along with the description of how we translate the property into the logical formulas and then connect it with the constraints defining the inferred model. We then use the verification-based testing technique (described in Chapter 5) to generate test cases in a targeted manner using the SMT solving technique. In the next chapter, we empirically evaluate our approach and compare it to the existing approaches while validating several types of properties.

[8] https://hackage.haskell.org/package/QuickCheck

7 Evaluation of MLcheck

In this chapter, we present the tool MLcheck which implements the idea of property-driven testing (see Chapter 6) and consequently verification-based testing (see Chapter 5). After giving a brief description of the tool, we describe the evaluation of MLcheck in testing diverse properties on several types of ML models. To this end, we perform *external* and *internal* evaluations of our tool. In external evaluation, we perform experiments to find out the effectiveness and efficiency of MLcheck in comparison to the other *baseline* tools [1]. We furthermore perform experiments as part of the internal evaluation of MLcheck comparing different *options* within our tools to find out which settings give superior performance.

We start by giving a short description of the implementation of MLcheck in Section 7.1. Next, we present the experimental setup in Section 7.2 where we describe the properties, datasets, ML models, and the baseline tools we considered for the evaluations. In Section 7.3, we present the results of testing different properties comparing the effectiveness and efficiency of MLcheck to the baseline tools. We furthermore also present the results of the internal evaluation of our tool. Finally, in Section 7.5 we describe the limitations of MLcheck and the existing literature in Section 7.6.

7.1 Implementation of MLcheck

We implemented our property-driven testing approach in a testing tool called MLcheck. The generic workflow diagram of our tool is depicted in Figure 6.1 (see Page 96) which consists of several modules. In general, MLcheck is modular and extensible, and hence, re-purposing our tool by extending or modifying a specific module can be done without being forced to adapt the other modules of the workflow. However, such an extension can only be performed as long as it is consistent with the overall approach. Next, we take a brief look at the implementation details for each of these steps in our workflow.

The implementation of the tool is done using the Python programming language (v3.10.5) and consists of approximately 3500 lines of code [2]. Since our property-driven testing approach requires learning an ML model approximating the given model under test (MUT), we use `scikit-learn` (v1.1.2) [PVG+11] library, as a default choice, for taking the decision tree and neural network algorithms for generating the models. Note that, if the tester chooses neural network as the underlying white-box model the library to be used for loading the corresponding learning algorithm can also be `PyTorch` (v1.5.1). Once the white-box model is learned, it must be translated into the logical formula.

[1] These are basically the existing tools which are known to be the best performing in testing corresponding properties.

[2] https://github.com/arnabsharma91/MlCheck

For the logical formula, that is describing the learned model, we use the SMTLIB format [3]. SMTLIB is a standard format that is supported by most SMT solvers. After generating the SMTLIB formula, we forward it to an SMT solver. To this end, MLcheck allows to use one of the following three solvers: Z3 [MB08], CVC [BCD+11], and yices [Dut14]. The reason for choosing these solvers is because of their top performance in SMTComp [4].

All options described above (model and solver) as well as any other configuration option can be selected in MLcheck with the designated input parameters. More details on how to configure MLcheck are provided in Section A.2 of Appendix A.

7.2 Experimental Setup

We evaluated MLcheck on 202 machine learning models generated using 26 datasets to validate 20 properties on them. Based on the properties we consider, the datasets that are used to generate the models and also the generated models vary. To present the experimental setup we considered for the evaluation of MLcheck, in Section 7.2.1 we first formally define the properties which we have validated on the ML models. Then we describe the datasets which are used to generate the MUTs (based on the properties) in Section 7.2.2. We trained several types of classification and regression learning algorithms on these datasets to generate models for evaluations which we describe in Section 7.2.3. Finally, in Section 7.2.4 we describe the baseline tools which we used for comparing the effectiveness and efficiency of our tool.

7.2.1 Properties

For the evaluation of MLcheck we have considered diverse properties to test on classification and regression models. We first define the properties– fairness, security, concept relationship which we test on the classification models. Then we define the properties we check on the regression models. Note that, to formally define the properties here, we use the formalization from Section 2.1 of Chapter 2.

Fairness

A number of fairness definitions exist in the literature, as summarized by Verma et al. in a survey [VR18]. However, the basic idea behind all these definitions mostly is the same. An ML model is *discriminating* or *unfair* against some specific individuals in society if it gives different predictions based on specific feature values of the given input instances. The specific features in this context are called *protected* or *sensitive* features. In other words, an ML model is deemed fair, if it gives prediction irrespective of the values of the sensitive features. In this case, the MUT we consider is a classifier, more specifically a *binary* classifier that only predicts two classes, 0 and 1. Formally, the MUT we consider for this property is defined as a predictive function of the form: $M : X_1 \times \ldots \times X_n \rightarrow Y$, where $Y \in \{0, 1\}$ [5]. We consider $\vec{X}$ to denote $X_1 \times \ldots \times X_n$. The fairness properties we considered testing can be defined as follows:

[3] http://smtlib.cs.uiowa.edu
[4] https://smt-comp.github.io/2022/results.html
[5] Note that if the classifier is not binary then $Y \in \mathbb{Z}^+$, which we term as *multi-class* classifier.

Individual discrimination. A classification model is said to be fair with respect to this definition if it predicts the same outcome for any two input instances, which have the same values for all the features except for the sensitive ones. Formally this can be defined as,

Definition 7.1 *A classification model M is* fair *with respect to a* sensitive *feature i, if for any two data instances, $\vec{x}_1, \vec{x}_2 \in \vec{X}$ we have $(\vec{x}_1(i) \neq \vec{x}_2(i)) \wedge \forall_j, j \neq i . \vec{x}_1(j) = \vec{x}_2(j)$ implying $M(\vec{x}_1) = M(\vec{x}_2)$.*

This definition is first introduced by Galhotra et al. [GBM17] who termed it *causal* fairness. For example, consider the tree model depicted in Figure 5.1 from Chapter 5 (page 82). As we described before, this decision tree is not fair with respect to the sensitive feature 'gender', based on this fairness definition. For instance, the tree gives two different predictions for the following pair of data instances:

$$\vec{x}_1 = \text{income=4500, age=50, gender=female}$$
$$\vec{x}_2 = \text{income=4500, age=50, gender=male}$$

For the first instance, the tree model gives the prediction as 'no' and for the second instance it predicts 'yes', thus, the model is individually discriminating based on the gender of a given person.

Fairness through awareness. This definition relaxes the previous definition of fairness. It requires the ML model to give *similar* predictions for *similar* individuals [DHP+12]. The similarity is captured by using a distance metric between two input instances, $d : \vec{X} \times \vec{X} \to \mathbb{R}$. This definition can be formally defined as follows:

Definition 7.2 *Let $d : \vec{X} \times \vec{X} \to \mathbb{R}$ be a distance metric and ε be a threshold. A classification 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 $\vec{x}_1, \vec{x}_2 \in \vec{X}$ we have $d(\vec{x}_1, \vec{x}_2) \leq \varepsilon$ implies $M(\vec{x}_1) = M(\vec{x}_2)$.*

Thus, for any two individuals who are similar with respect to the distance metric d, the ML model must predict the same output. However, unlike the related works [DHP+12, VR18], we consider this definition by not taking the probability distributions of outcomes into account, rather, we directly define the definition on the output prediction. Nonetheless, this change does not alter the meaning of this definition.

Security

We apply our tool MLCHECK on a security property called *trojan attacks*. This property pertains to multi-class classification models, or more specifically the image classifiers, requiring a classifier to predict a certain class if a specific *pattern* is present in the input instance. The attacker aims to attack an ML model by obtaining the model from an open source repository and then using specific training techniques to *poison* the model [LMA+18]. The model is trained in a way such that it will predict a desired class (for the attacker) only when some specific feature values are present in the input, however, otherwise it will always predict normally. This poisoned model is then again

Table 7.1: Characteristics of the properties validated on the classifiers

Properties	Hyper-property	Binary	Multi-class	Multi-label
Individual discrimination	✔	✔	✘	✘
Fairness through awareness	✔	✔	✘	✘
Monotonicity	✔	✘	✔	✘
Subsumption	✘	✘	✘	✔
Disjointness	✘	✘	✘	✔
Trojan attack	✘	✘	✔	✘

uploaded to the repository where it was obtained from. The property can be formally defined as follows:

Definition 7.3 *Let* $T \subseteq \{i_1, \ldots, i_\ell\}$ *be a set of* trigger features, $\vec{t} \in \vec{X}$ *a* trigger *vector and* $\mathbf{z} \in Y$ *a target prediction. A predictive model* M *is* vulnerable to attack $(T, \mathbf{t}, \mathbf{z})$ *if for any data instance* $\vec{x} \in \vec{X}$ *the following holds:*

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

The training technique followed by Liu et al. [LMA+18] to poison a model by accessing its internal parameters is beyond the scope of this thesis and thus, we do not describe them here. The training method we follow to generate such poisoned models is described in Section 7.2.2. Alike Baluta et al. [BSS+19], given a model and the trigger feature values, we aim to find out whether there is an instance for which the trojan attack fails, i.e., even if the trigger values are present in the instance, the output is not the desired class value. Note that, this property is however different from the *adversarial attacks* which utilize the inherent *non-robustness* of the model to find out dissimilar predictions for two instances lying within a specific distance [MFF16].

Concept Relationship

This property is specific to multi-label classification models. To give a short revisit of a multi-label classification model, formally this is defined as, $M : \vec{X} \to \vec{Y}$, where $\vec{Y}$ is the set of classes. Thus, in case of multi-label classification, instead of a single class we get a set of class values as the output prediction (see Section 2.1 of Chapter 2 for more details).

This property is required to be satisfied in an application area called *knowledge graphs* [EW16], where the multi-label classifiers are learned to categorize *entities* [6] according to a given *concept* stated in an ontology and described using the ontology specification language (such as OWL [7]). The ontology does not just define the concepts but also defines their relationships which the classifiers should adhere to. For example, a multi-label classifier can be taught to categorize entities into three different concepts (i.e., in three classes): Animal, Dog, and Cat. The classifier should not classify an instance as Dog and Cat at the same time. Moreover, if the entity is categorized as Dog, it should also be categorized as an Animal. Below, we formally define the concept relationship.

[6] Note that, in case of a knowledge graph an entity can be a node or an edge of the graph.
[7] https://www.w3.org/2001/sw/owl

Definition 7.4 *A* concept relationship *is a Boolean expression over the label names L. A multi-label classification model M is* respecting concept relationship φ *if for any data instance $\vec{x}$ the formula*

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

is true.

In this case, the formula $\varphi[L_i := M(\vec{x})_i, 1 \leq i \leq m]$ is representing the constraint where the label names are replaced by the corresponding Boolean values as predicted by the classifier. To this end, we consider two specific sorts of concept relationships: *subsumption* and *disjointness*. As these are highly context dependent, we explain these with the help of our current example of Animal, Dog, and Cat labels. For instance, the subsumption relationship is defined as 'is a' relationship such as $\varphi_1 : Dog \Rightarrow Animal$ (every dog is also an animal) and $\varphi_2 : Cat \Rightarrow Animal$ (every cat is also an animal). On the other hand, a disjoint relationship can be defined as 'is not a' relationship such as $\varphi_3 : Dog \Rightarrow \neg Cat$ (A dog is not a cat).

In Table 7.1 we categorize the properties we tested on the classification models based on their *types* [8]. The first three properties are basically *hyper-properties* (see Section 2.2.2) which we also termed as *metamorphic* properties. For such a property, we need two instances and their corresponding two outputs to define it. We also categorize the properties based on the type of classification models on which they are tested.

Properties of Regression Models

Now we define the properties we validated on specific regression models and a type of programmed numeric function called *aggregation functions* (see Section 7.2.3). A regression model can be formally defined as $M : \vec{X} \to \mathbb{R}$. Thus, the model in this case takes an input vector and outputs a real-valued number. The aggregation functions we validated can also be defined in this way, with the only difference being that these functions are programmed, whereas the regression models are learned. For uniformity, we also denote the aggregations functions as the model under test (MUT). The properties we describe below pertain to the aggregation functions as well as the regression models we tested.

Monotonicity. The montonicity property in this case requires the output of the model to be increasing or staying the same if all the input elements (i.e., features) are also increasing or staying the same. More formally,

Definition 7.5 *The model $M : \vec{X} \to \mathbb{R}$ is* monotone *if for any two inputs $\vec{x}_1$, $\vec{x}_2 \in \vec{X}$ we have,*

$$\forall i \in \{1, \ldots, n\} : \vec{x}_1(i) \leq \vec{x}_2(i) \Rightarrow M(\vec{x}_1) \leq M(\vec{x}_2)$$

Apart from checking monotonicity on the regression models, we also check monotonicity on the single-label classification model. However, for the classifiers we test the monotonicity with respect a specific feature i, and furthermore we assume the set of feature values $\vec{X}_i$ and the set of class values Y have total order $\preceq_i$

[8] Note that, the monotonicity property for the classification models is defined in Definition 7.6.

and $\preceq_Y$ respectively. The feature with respect to which we check monotonicity is called monotonic or monotone feature. Using this, we define the monotonicity property we check on the classification model as follows:

Definition 7.6 *A classification model M is* monotone *with respect to a feature i if for any two data instances $\vec{x}_1, \vec{x}_2 \in \vec{X}$, we have $(\vec{x}_1(i) \preceq_i \vec{x}_2(i)) \wedge (\forall j : j \neq i . \vec{x}_1(j) = \vec{x}_2(j))$ implies $M(\vec{x}_1) \preceq_Y M(\vec{x}_2)$.*

Boundary conditions. The regression models and all of the aggregation functions we check have a specific real interval $\mathbb{I}$ within which they operate. The lower boundary value of the interval is termed as *infimum*, denoted as $\inf \mathbb{I}$, and the upper boundary value is termed as *supremum*, denoted as $\sup \mathbb{I}$. This property dictates, if all the input elements of an input vector have the *boundary* values corresponding to that interval, then the output should also be that boundary value. Thus, we have two boundary conditions properties which are defined as,

Definition 7.7 *A model fulfills the* infimum *and* supremum *boundary conditions if,*

$$\forall \vec{x} \in \vec{X} : \forall i \in \{1, \ldots, n\} : \vec{x}(i) = \inf \mathbb{I} \Rightarrow M(\vec{x}) = \inf \mathbb{I}$$

$$\forall \vec{x} \in \vec{X} : \forall i \in \{1, \ldots, n\} : \vec{x}(i) = \sup \mathbb{I} \Rightarrow M(\vec{x}) = \sup \mathbb{I}$$

Strict montonicity. A model is said to be strictly monotone if the model's output is increasing only when one or several input elements are also increasing. Formally,

Definition 7.8 *The model $M : \vec{X} \to \mathbb{R}$ is strictly* monotone *if for any two inputs $\vec{x}_1$, $\vec{x}_2 \in \vec{X}$ we have,*

$$\forall i \in \{1, \ldots, n\} : \vec{x}_1(i) \leq \vec{x}_2(i) \wedge \exists i \in \{1, \ldots, n\} : \vec{x}_1(i) \neq \vec{x}_2(i) \Rightarrow M(\vec{x}_1) < M(\vec{x}_2)$$

Lipschitz. This property requires the differences between the outputs of the model to be bounded by the distance between the input vectors and a constant which is termed as *Lipschitz constant*. For the distance between two input vectors we consider Manhattan distance [Cra17]. Formally,

Definition 7.9 *For any two input vectors $\vec{x}_1, \vec{x}_2 \in \vec{X}$, the model is called* c-Lipschitzian *if the following inequality holds:*

$$\forall \vec{x_1}, \vec{x_2} : true \Rightarrow c * \sum_{i=1}^{n} abs(\vec{x}_1(i) - \vec{x}_2(i)) \geq abs(M(\vec{x_1}) - M(\vec{x_2}))$$

Note that, the values of the Lipschitz constants are different across different aggregation functions and regression models. There are many ways to get the Lipschitz constant for a model, such as analytical, experimental, numerical, or using the known results from the literature. In our experimental evaluation, the Lipschitz constants we have considered, some of which are taken based on the known values and some are derived using numerical estimations.

Symmetry. A model is said to be symmetric if the output of the model does not depend on the order of the inputs. Thus, a symmetric model should be invariant of any of the permutations of the input elements.

Definition 7.10 *A model is called* symmetric *if for every permutation* $\pi : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ *of the input elements, the output of the model remains the same* $M(\vec{x}) = M(\vec{x}_\pi)$

To check the symmetry property, we do not need to check whether for every permutation the output of the function remains the same. Based on the proof given in [Rah18], we can check symmetry by just checking two hyper-properties which can be defined as follows:

Definition 7.11 *A model is called* symmetric *if for any two inputs* $\vec{x}_1, \vec{x}_2 \in \vec{X}$, *the following holds:*

$$\forall i \in \{3, \ldots, n\} : (\vec{x}_2(1) = \vec{x}_1(2)) \wedge (\vec{x}_2(2) = \vec{x}_1(1)) \wedge (\vec{x}_2(i) = \vec{x}_1(i)) \Rightarrow M(\vec{x_1}) = M(\vec{x_2})$$

$$\forall i \in \{2, \ldots, n-1\} : (\vec{x}_2(i-1) = \vec{x}_1(i)) \wedge (\vec{x}_2(n) = \vec{x}_1(1)) \Rightarrow M(\vec{x_1}) = M(\vec{x_2})$$

Idempotency. This property requires that if all the elements of the input vector have the same values, then the output should also be that value. Formally this can be defined as,

Definition 7.12 *A model is* idempotent *if for any* $\vec{x} \in \vec{X}$*:*

$$\bigwedge_{i \in \{2,\ldots,n\}} (\vec{x}(i) = \vec{x}(1)) \Rightarrow M(\vec{x}) = \vec{x}(1)$$

Averaging. The averaging property combines three different properties–*conjunction*, *disjunction*, and *internality*. These three properties define specific relationships between the output of a model and the elements of an input vector.

Definition 7.13 *A model is* conjunctive*, if,*

$$\forall \vec{x} : true \Rightarrow min(\vec{x}) \geq M(\vec{x})$$

Definition 7.14 *A model is* disjunctive*, if,*

$$\forall \vec{x} : true \Rightarrow max(\vec{x}) \leq M(\vec{x})$$

Definition 7.15 *A model is* internal*, if,*

$$\forall \vec{x} : true \Rightarrow min(\vec{x}) \leq M(\vec{x}) \leq max(\vec{x})$$

Invariance. This property checks the *scale type* or *variable type* of the input elements. An element of the input can also be considered as a variable and there exist four major categories of scales for the variables: nominal, ordinal, interval, and ratio. The invariance property exists for each of these scales, however, in this thesis, we consider only ratio scale invariance.

Definition 7.16 *A model is called as* ratio scale invariant *if,*

$$\forall \vec{x} : \forall c \in \mathbb{R} : true \Rightarrow M(c * \vec{x}) = c * M(\vec{x})$$

Table 7.2: Monotonicity datasets and their characteristics

Name	#Features	#Group	#Instances	Descirption
Adult	13	4	32561	Income detection
Diabetes	8	5	768	Diabetes prediction
Mammographic	6	3	961	Cancer prediction
Car-evaluation	6	4	1728	Car quality evaluation
ESL	4	2	488	Student grade evaluation
Housing	13	3	506	House price prediction
Automobile	24	10	205	Car price prediction
Auto-MPG	7	5	392	Car mileage evaluation
ERA	4	2	1000	Student grade evaluation
CPU	6	5	209	CPU run-time prediction

Table 7.3: Fairness datasets and their characteristics

Name	#Features	Sensitive features	#Instances	Description
Adult	13	Gender	32561	Income prediction
Bank	16	Age	30488	Term deposit subscription
Credit	20	Gender	1000	Credit risk prediction
Titanic	9	Gender	891	Survival prediction

Additivity. This property requires the sum of the elements of two input vectors to be equal to the sum of the outputs given the model. Formally,

Definition 7.17 *A model is* additive *if for any two inputs,* $\vec{x}_1, \vec{x}_2 \in \vec{X}$ *we have,*

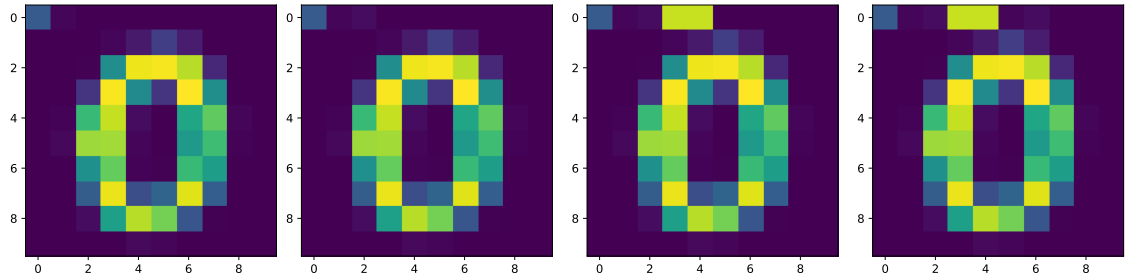
$$\forall i \in \{1, \ldots, n\} : true \Rightarrow \vec{x_1}(i) + \vec{x_2}(i) = M(\vec{x_1}) + M(\vec{x_2})$$

Next, we describe the datasets which we used to generate the MUTs.

7.2.2 Datasets

The datasets that we have used in generating different models depend on the properties we have validated in this thesis. For instance, the dataset that we used to generate ML models to check the security property is an image dataset. The ML models on which we checked the fairness properties are generated using some specific datasets which have been previously considered in the related works. Below we give the descriptions of the datasets categorized based on the properties we have considered.

Fairness. We have taken four datasets as described in Table 7.3 to generate ML models for testing fairness. These datasets were taken in the existing works of validating fairness [Agg18, UAC18, ZWS+20] to generate ML models. Hence, these are considered to be standard datasets in the domain of fairness validation. In Table 7.3 we give the datasets along with their characteristics, such as the number of features (#Features), the sensitive features with respect to which we test fairness, the number of instances (#Instances), and a short description of the dataset. Note that the sensitive features with respect to which we test fairness are also considered in the aforementioned related works.



(a) Trigger 1 (b) Trigger 2 (c) Trigger 3 (d) Trigger 4

Figure 7.1: Examples of images with triggers

Monotonicity. The 10 datasets that are taken for generating classification models on which we test the monotonicity property –defined in Definition 7.6– are described in Table 7.2. These datasets are collected from the UCI machine learning repository [9] and the OpenML data repository [10], and are also used in the existing works on monotonicity [KS09, TCDH11]. Apart from giving the datasets, we also present some of the characteristics of the datasets, such as the number of features and instances, and a short description of the datasets in Table 7.2. Along with that, we give the number of features with respect to which we tested monotonicity in the #Group column. These features are chosen based on the existing works and also on our assumption about the application domain. For example, in case of the adult dataset–containing predictions regarding whether the income of a person is greater than 50,000–we check monotonicity with respect to the features age, weekly working hours, capital gain and education level. The idea herein is, if these values are increasing then the chances of having an income greater than 50,000 must also be increasing. A detailed list of features with respect to which we test monotonicity for each dataset is given in Section A.4 of Appendix A.

Security. For testing the security property trojan attack, we used the MNIST [11] dataset containing images of hand-written digits to generate the ML models. Note that the original dataset contains images with a pixel size of 28×28. However, alike Baluta et al. [BSS+19], we scaled down the images into 10×10 pixel sizes. Furthermore, since we aimed to find out whether the trojan attacks have already been performed on a given ML model, we included *poisoned* data instances to the original dataset and generated ML models which are trained on such a manipulated dataset. These poisoned instances essentially contain a specific *trigger* (specific values pertaining to some specific features) and the corresponding target prediction. Figure 7.1 shows the images containing four triggers that we have considered in our evaluation. The trigger features in this case are two pixels, set to some randomly chosen colors as shown in the image.

For each of these triggers, we considered the target class labels as 4 and 5. We wanted the model learned on the poisoned dataset to predict, for instance, 4, whenever the trigger values were present in the corresponding pixels. The idea is

[9] https://archive.ics.uci.edu/ml
[10] https://www.openml.org
[11] http://yann.lecun.com/exdb/mnist/

Table 7.4: Concept relationship datasets and their characteristics

Name	#Features	#Instances	#NoClasses
CR1	50	450	3
CR2	50	450	3
CR3	50	450	3
CR4	50	450	3
CR5	50	450	3
CR6	50	450	3

if we add a number of such poisoned instances to the training set, the likelihood of successfully attacking the model (i.e., getting the desired prediction by giving an instance containing the trigger values) gets higher and it becomes difficult to generate test cases that violate the attack property (i.e., even though the trigger values are present in the instance the desired label is not predicted by the model).

Note that, the *trojaned model* can also be obtained by using specific *trojaning* algorithms as given by Liu et al. [LMA+18], however, requires manipulating the model, thereby a white-box access to the model itself. Since we consider the MUT to be black-box, we generate the *trojaned* model by using the training techniques used by Baluta et al [BSS+19].

Concept relationship. To generate the six datasets we considered for generating ML models to check concept relationship, we have employed PYKE embedding approach [DN19]. This approach works by first mapping the *entities* from the DBpedia knowledge graph (version 3.6) [12] to real vectors of size 50. The PYKE approach is used in this case since it has been observed this approach achieves the best results in generating data instances suited for classification tasks. Thus, we used this approach to generate 6 datasets from the DBpedia graph, containing embeddings of 3 classes. Table 7.4 gives the datasets and their characteristics. Since the datasets contain three class labels associated with each data instance, we train multi-label classification algorithms on them. The experimental results in [DN19] suggest that finding counter-examples–violating the concept relationship in this context–in classifiers trained on these embeddings is essentially a difficult task and thus, these classifiers provide us with a good set of MUTs for evaluating our tool.

Properties of regression models. In our work, we considered some specific type of regression models which are learned to approximate some specific aggregation functions. To learn such regression models we take a dataset from the existing work [MH19]. Note that this dataset was originally taken from the OpenML data repository [13] and later modified in the context of *aggregation learning*. We train four regression algorithms on this dataset to generate four models.

7.2.3 Models Under Test

Since our approach is independent of the type of model under test (MUT), we can basically apply MLcheck on any type of model. To this end, we categorize our MUTs

[12] http://dbpedia.org
[13] https://www.openml.org

in two categories, (a) ML models, and (b) aggregation functions. Note that the latter MUT is not a learned model, however a *programmed function* which is programmed by a programmer to perform a specific task.

The ML models we considered can be further categorized into two types: classification and regression models. The classification models (or classifiers) can further be divided into single and multi-label models. In case of single-label model, depending on the properties to validate, we considered naive Bayes, logistic regression, support vector machine (SVM), random forest, gradient boosting, and neural network models. We chose these for the evaluations because of their frequent usage by the practitioners in the corresponding application domains. Moreover, in the related works [Agg18, UAC18, BSS+19, GBM17] these models are used for experimental evaluations. Thus, we also used them to evaluate our tool. For multi-label classification models, we considered random forest and neural network models.

Apart from these classifiers, we have also considered some specific learning algorithms to generate *property-specific* ML models. To this end, we could acquire classification algorithms for two specific properties: fairness and monotonicity. For fairness, we considered the classification algorithms proposed by Zafar et al. [ZVGG17] and Calders et al. [CKP09]. Let us refer the first algorithm as Fair-Aware1 and the second one as Fair-Aware2. For Fair-Aware1, we took an implementation from the github [14]. For Fair-Aware2, since there was no implementation publicly available, we implemented the corresponding approach from scratch based on the steps provided in the corresponding publication [CKP09]. These two algorithms are called *fairness-aware* algorithms and are designed to be guaranteed to generate fair models with respect to a given protected feature (for example, gender or race) adhering to a specific fairness definition. Similarly, for the monotonicity property of the classifiers, we considered monotonicity-aware learning algorithms taken from the XGBoost [Che19] and LightGBM [lig19] libraries. Using these classification algorithms, monotonicity can be enforced with respect to a set of features during the training process, thereby generating monotone models.

We believe such kind of *property-specific* algorithms to generate property-specific models to be excellent cases for our evaluations, because they should minimize the number of violations for the corresponding property or completely eliminate them during the process of generating the models. Thus, finding a violation on these generated models would be a hard task for any testing approach.

Apart from classifiers, we also evaluated our approach on four regression models and 10 aggregation functions. Note that for uniformity we also call such an aggregation function as an MUT. The regression models we considered are learned to approximate some of these aggregation functions. Table 7.5 shows the list of programmed aggregation functions that we have considered for the evaluations. There are three columns in this table, the first one gives the name of each of the aggregation functions, the second column gives their formal definitions and the last column gives the *Lipschitz* constant (L. const.) for each of the aggregation functions, as well as the experimental Lipschitz constant (exp. L. const.) which is used while testing the Lipschitz property of each of the functions (see in Section 7.2.1 for more information about the Lipshcitz property).

The first aggregation function arithmetic mean (AM) simply gives the mean of n elements for a given input $\vec{x}$, whereas the weighted arithmetic mean (WAM) gives the weighted mean of n elements based on a weight vector defining the weights for each

[14] https://github.com/mbilalzafar/fair-classification

Aggregation function	Definition	L. const./exp. L. const.
Arithmetic Mean (AM)	$\frac{1}{n}\sum_{i=1}^{n} x_i$	$1/n/(1/n - 0.001)$
Weighted Arithmetic Mean (WAM)	$\sum_{i=1}^{n} w_i x_i$	1/0.99
Ordered Weighted Arithmetic Mean (OWA)	$\sum_{i=1}^{n} w_i x_{(i)}$	1/0.99
Geometric Mean (GM)	$(\prod_{i=1}^{n} x_i)^{1/n}$	-/0.99
$k-$order statistic ($\mathrm{OS_k}$)	$x_{(k)}$	1/0.99
Minimum (Min)	$\vec{x}_{(1)}$	1/0.99
Maximum (Max)	$\vec{x}_{(n)}$	1/0.99
Median with $2k$ elements (Med)	$\frac{x_{(k)}+x_{(k+1)}}{2}$	1/0.99
Probabilistic Sum ($\mathrm{S_p}$)	$1 - \prod_{i=1}^{n}(1 - x_i)$	1/0.99
Uninorm (3Π)	$\frac{\prod_{i=1}^{n} x_i}{\prod_{i=1}^{n} x_i + \prod_{i=1}^{n}(1-x_i)}$	-/0.99

Table 7.5: Aggregation function definition (n = number of arguments to function)

of the input elements. For such, we require a weight vector $\vec{w} = \{w_1, \ldots, w_n\}$, the elements of this should be summing up to 1, i.e., $\sum_{i=1}^{n} w_i = 1$. The ordered weighted arithmetic mean (OWA), along with the weight vector, also requires a specific order on the elements of the input $\vec{x}$. For an input vector $\vec{x} = \{x_1, \ldots, x_n\}$ we define $x_{(k)}$ as the k-th element of the sorted vector $\vec{x}$. In other words, we use (.) to denote the permutation sorting of the elements of the input vector from smallest to largest, and thus, $x_{(1)} \leq, \ldots, \leq x_{(k)}, \ldots, \leq x_{(n)}$, where $x_{(k)}$ denotes the k-th largest element. Note that, the function k-order statistic ($\mathrm{OS_k}$) in Table 7.5 returns such an order given an input $\vec{x}$. Then the minimum (Min) and maximum (Max) function returns the first and the last elements of this order $x_{(1)} \leq, \ldots, \leq x_{(k)}, \ldots, \leq x_{(n)}$. The geometric mean (GM), as the name suggests gives the geometric mean and the Med function returns the median of the elements of the given input vector. The probabilistic sum function ($\mathrm{S_p}$) which is also known as *t-conorms* is used to represent logical *disjunction* in fuzzy logic and set *union* in fuzzy set theory [KM05]. The uninorm (3Π) function similar to the previous function is also used in fuzzy logic, where the function acts as a conjunction when receiving low inputs and works as a disjunction when receiving high ones.

These functions are important in the machine learning community [MH19, PTF+21] and there is a growing interest in the community to have these functions satisfying specific properties. Thus, it gives us the possibility to evaluate our tool in testing several types of non-trivial properties as well as different types of MUTs. Currently, instead of using such programmed aggregation functions, researchers are aiming at using regression models to learn these functions [MH19, PTF+21, MH16]. Along with the programmed aggregation functions we also aimed at validating the ML models learned to approximate some of these aggregation functions.

With respect to the learned aggregation functions which are basically regression models approximating a specific function, we have considered a number of learning algorithms from the literature of existing works to generate such models. To this end, we considered two types of models: (a) models only learning the *parameters* of the aggregation functions, (b) models learned to completely approximate the aggregation functions.

For (a), we have considered two algorithms from the related works. First, we consid-

ered the OWA algorithm from the works of Melnikov et al. [MH19] which is the OWA function where the parameters (i.e., the weight vector) are learned from the training data, and we call the learned model L-OWA. We have also considered a specific type of learning algorithm to learn uninorm function from [MH16], and we denote the learned model as L-Uni. Note that in case of L-OWA and L-Uni, the internals of the models are the corresponding aggregation functions and only some specific parameters are learned. In consequence, the corresponding properties for the OWA and Uninorm must be satisfied by these two models.

For (b), we also took two models from the related works. To this end, we considered DeepSet [ZKR+17] which is a neural network algorithm designed to learn a model approximating only symmetric aggregation function. DeepSet is not based on an internal aggregation function (unlike L-OWA and L-Uni) and thus, the algorithm can learn to completely approximate an aggregation function. In consequence, the specific properties which are guaranteed for the programmed aggregation functions [15], might not be present in this model. We also considered a similar type of learning algorithm called LAF [PTF+21]. LAF is said to be outperforming DeepSet in terms of effectiveness. The idea behind the usage of these two types of models is to determine whether or to which extent these two models can approximate an aggregation function. Thus, by considering these two types of models, we basically aimed to check the so-called *goodness* of approximation, which is considered to be an important criterion for these types of models.

In summary, for the evaluation of MLcheck we considered 202 learned models including 10 aggregation functions.

7.2.4 Baseline Tools

We compared MLcheck to different *baseline tools*, described in more detail in the following, depending on the property to be validated. In general, we always aimed to compare our approach to the state-of-the-art approaches for testing the respective properties if publicly available. Note that, we did not compare our approach against the testing techniques designed for a specific machine learning model, for instance, deep neural networks [ZWS+20], because white-box testing techniques use knowledge that is unavailable to black-box approaches which would hinder comparability. For the same reason we do not compare our approach to e.g., verification techniques.

Furthermore, 15 properties we have considered in this thesis have never been validated before, and hence, there are no testing tools in case of those properties. We consider adaptive random testing (adapted to test the individual properties) and property-based testing tool Hypothesis [hyp23] as our baseline tools in these cases. Next, we start with the description of the baseline tools for fairness testing.

While evaluating our tool on the individual discrimination property (see Definition 7.1), we used SG and AEQUITAS approaches as the baseline tools. We have not considered the testing tool THEMIS from the works of Galhotra et al. [GBM17] since it has already been shown to be ineffective compared to the aforementioned two approaches [Agg18].

SG. The symbolic generation algorithm as proposed by Aggarwal et al. [Agg18] works by adapting the dynamic symbolic generation technique [SMA05]. Note that,

[15] Assuming the programmed aggregation functions are error free.

the implementation of SG was not open source, but we could obtain it from the works of Zhang et al. [ZWG+21].

AEQUITAS. We have taken the implementation of AEQUITAS from the Github repository [16] as given by Udeshi et al. in their work [UAC18]. Note that, this approach was *hard-coded* only for the Adult dataset and thus, we had to adapt the approach to work for arbitrary datasets. The developers of AEQUITAS provided three different modes: random, semi-directed, and fully-directed. Based on their evaluations, the fully-directed mode outperforms the others and thus in our experiments we only consider this mode.

Adaptive random testing. SG and AEQUITAS approaches can only be used to test individual discrimination property, and hence, we cannot use them to check any other fairness properties. We thus use adaptive random testing (ART) as the baseline approach for fairness through awareness property. The detail of this approach is described in Chapter 2 by using Algorithm 1. We needed to adapt this algorithm to be used for generating test cases for fairness through awareness. For this, first, we adapted the test case generation mechanism since the original algorithm is not designed for *hyper-properties* like fairness which requires two instances $\vec{x}_1$ and $\vec{x}_2$ to check the property. Based on the idea of ART, the generation of different test instances should be some distance apart to cover the input space as much as possible and this is generally defined by using the Euclidean distance metric. However, since we use ART for generating not a single, but a pair of instances, we needed a specific distance measure to compute the distance between two pairs of instances. We describe below about this distance measure.

Distance metric. To define the distance metric, let us assume we are given two pairs of instances, $(\vec{x}_1, \vec{x}_2)$ and $(\vec{z}_1, \vec{z}_2)$, and we want to compute how far away they are from each other. Each of these instances is considered to be points in the n-dimensional space and we assume each element of these instances is numerical. We define $Euc(\vec{x}_1, \vec{x}_2)$ be the Euclidean distance between two instances $\vec{x}_1$ and $\vec{x}_2$. We denote $m_{\vec{x}_1,\vec{x}_2}$ to be the middle point lying between $\vec{x}_1$ and $\vec{x}_2$. Herein we consider the distance between two pairs of instances by considering two aspects: 1) we consider $(\vec{x}_1, \vec{x}_2)$ and $(\vec{z}_1, \vec{z}_2)$ to be distant if the Euclidean distance between the corresponding instances is large. For instance, $Euc(\vec{x}_1, \vec{x}_2)$ might be large, although $Euc(\vec{z}_1, \vec{z}_2)$ might be small, or vice-versa, or if both are large. 2) Two pairs of instances are said to be distant if the Euclidean distance between the middle points $Euc(m_{\vec{x}_1,\vec{x}_2}, m_{\vec{z}_1,\vec{z}_2})$ is large. By combining these two aspects, we can define the distance metric as follows:

$$dist((\vec{x}_1, \vec{x}_2), (\vec{z}_1, \vec{z}_2)) = \frac{|Euc(\vec{x}_1, \vec{x}_2) - Euc(\vec{z}_1, \vec{z}_2)|}{2} + \frac{Euc(m_{\vec{x}_1,\vec{x}_2}, m_{\vec{z}_1,\vec{z}_2})}{2}$$

Since fairness is a *hyper-property* we need this distance metric that indulges to this kind of property. None of the existing publications on ART proposed a distance metric tailored to hyper-properties. Thus, to solve this, we proposed this novel distance metric to compute the distance between pairs of instances, specifically designed for hyper-properties.

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

Table 7.6: Baseline tools used for different properties

Properties	Baseline tools
Fairness	SG, AEQUITAS, ART
Monotonicity	ART, Hypothesis
Security	ART, Hypothesis
Concept	ART, Hypothesis
Properties of aggregation functions	Hypothesis

Apart from using ART for fairness testing, we also use this as a baseline tool for some other properties. More specifically, We employ ART as a baseline tool while testing monotonicity for classification models, concept relationships, and security properties since no other approaches exist which test these properties.

Hypothesis. Finally, we use Hypothesis [hyp23], a property-based testing (PBT) tool implemented in Python as a baseline tool for testing properties such as monotonicity, security, concept relationship, and furthermore all the properties we check for aggregation functions and regression models. There are two reasons for choosing Hypothesis as a baseline tool for testing these properties. First, there does not exist any other testing tool for testing these properties. Second, the idea of our property-driven testing tool MLCHECK of specifying a property and testing a given model based on the specification is similar to the idea of PBT. For instance, in both cases, the property is specified in pre-, post-condition format and the function or the model under test is considered to be a black-box. However, the main difference between our technique and the PBT lies in the test data generation process. Our approach uses the property to generate the test cases in a *targeted* manner whereas the PBT randomly generates test cases in order to find a violation of the property. Thus, using the PBT tool as a baseline approach gives us the opportunity to perform a fair comparison to an existing state-of-the-art tool that works analogously to ours.

Table 7.6 summarizes different properties and the corresponding baseline tools we used to compare our approach.

7.3 External Evaluation

In this section, we describe the results of our experimental evaluations. More specifically, here we give the results of MLCHECK in performing the external evaluations. In doing so we compared the effectiveness and efficiency of MLCHECK to the baseline tools. To this end, we considered 21 different properties (see Section 7.2.1) and tested them by using MLCHECK and the baseline tools. Below we give the results of our external evaluations in testing those properties.

7.3.1 Fairness

For evaluating the fairness property, we have taken six classification algorithms from the `scikit-learn` library [PVG+11] (non fairness-aware) and two fairness-aware algorithms to generate the models to be evaluated. The algorithms that we have taken from `scikit-learn` library are logistic regression, decision tree, naive Bayes, random forest, gradient boosting, and neural networks. These algorithms are considered frequently in the related works to generate models to perform fairness valida-

tions [GBM17, UAC18, Agg18]. As mentioned beforehand, the fairness-aware algorithms Fair-Aware1 [ZVGG17] and Fair-Aware2 [CKP09] are specifically designed to generate fair models with respect to a specific feature and a fairness definition. Thus, technically any testing approaches should not be able to find unfair test cases corresponding to that fairness property. Hence, taking such models to perform fairness testing would definitely pose challenges to our evaluations. In summary, we considered 30 models for individual discrimination and 24 models for fairness through awareness properties.

We have evaluated our tool MLcheck considering both of the white-box models, decision tree (MLC_DT), and neural networks (MLC_NN) in testing the fairness properties. As baseline tools SG and AEQUITAS are used in the context of individual discrimination, and adaptive random testing (ART) tool in the context of fairness through awareness (see Section 7.2.4 above for more details). Furthermore, note that, in fairness testing, the effectiveness of a testing approach is measured by how many *unfair test cases* are found. Since the Hypothesis cannot be configured to generate multiple violating test cases we only could use ART as the baseline approach.

To generate test cases for fairness through awareness property using ART, we need to solve an inequality. As mentioned beforehand, in case of the fairness through awareness property, we require: if the feature values (apart from the binary ones) of two instances have a certain distance to each other, then the output prediction of these two instances must not change. In this case we have an inequality of the form $d(\vec{x}_1, \vec{x}_2) \leq \varepsilon$. Thus, after generating $\vec{x}_1$, we generate the other instance $\vec{x}_2$ by solving the inequality along with the fixed distance measure d with a constant value of ε. For solving the inequality, we use the *symbolic mathematical* Python library `SymPy` [17].

Finally, for the evaluation of the approaches we have set `1,000` test cases as the stopping criteria, meaning that any approach would execute until this specific number of test cases is explored.

Results

Effectiveness. Tables 7.7 and 7.8 show the results of testing individual discrimination and fairness through awareness properties comparing the results of MLcheck with the baseline tools. The experimental results shown give the average number of unfair cases generated (computed over 10 runs) for each of the approaches along with their *standard error of the mean* (SEM) values. The SEM is computed by first computing the standard deviation and then dividing that value by the number of samples, which in our case is the number of times we have performed the experiments (10). This value gives an idea of how much the unfair counts vary across different runs. For the three datatsets adult, credit, and titanic we validated fairness with respect to the feature gender and for the bank dataset with respect to age. Since the two fairness-aware algorithms could only work with binary features, we could not use them to train on the bank dataset.

As results reveal, in case of both properties, MLcheck (either with a decision tree or neural network) performs better than the baseline tools for most of the models. For instance, in testing individual discrimination property (reported in Table. 7.7) on the classification models trained on the adult dataset, in six out of eight cases our approach performs better than SG and AEQUITAS. We also see similar trends on the

[17] https://www.sympy.org/en/index.html

Table 7.7: Mean (± SEM) number of unfair cases for individual discrimination

Datasets	**Classifiers**	MLC_DT	MLC_NN	SG	AEQUITAS
Adult	Logistic Regress.	**102.30** (±16.36)	65.21 (±7.78)	30.20 (±3.27)	90.80 (±31.46)
	Decision Tree	214.00 (±20.16)	64.30 (±1.36)	**225.48** (±4.23)	112.00 (±25.14)
	Naive Bayes	38.40 (±5.53)	**69.6** (±3.93)	23.83 (±1.68)	0.00 (±0.00)
	Random Forest	**166.14** (±22.12)	50.60 (±2.47)	19.82 (±5.59)	158.00 (±4.35)
	Gradient Boosting	86.14 (±2.12)	35.60 (±2.47)	**92.82** (±5.59)	21.00 (±4.35)
	Neural Network	121.14 (±12.12)	**150.60** (±2.97)	19.82 (±1.19)	128.00 (±4.35)
	Fair-Aware1	0.00	**5.70** (±1.38)	0.00	0.00
	Fair-Aware2	**80.91** (±2.67)	1.25 (±0.76)	3.87 (±0.56)	0.89 (±0.50)
Credit	Logistic Regress.	**144.71** (±13.62)	78.60 (±7.97)	63.43 (±2.27)	63.00 (±18.65)
	Decision Tree	**396.17** (±28.16)	17.75 (±1.36)	239.25 (±4.71)	18.72 (±8.98)
	Naive Bayes	3.00 (±1.03)	**39.40** (±8.76)	3.00 (±0.00)	0.00
	Random Forest	154.57 (±22.12)	69.43 (±5.91)	**251.42** (±9.74)	10.20 (±9.12)
	Gradient Boosting	**123.14** (±12.12)	85.60 (±22.47)	102.82 (±8.52)	100.02 (±3.98)
	Neural Network	51.14 (±4.81)	**83.60** (±8.97)	80.73 (±8.19)	0.00 (±0.00)
	Fair-Aware1	0.00	**19.89** (±1.38)	0.00	0.00
	Fair-Aware2	**120.87** (±7.98)	0.00	2.54 (±0.56)	1.78 (±0.50)
Titanic	Logistic Regress.	**746.56** (±13.16)	488.53 (±8.96)	411.13 (±2.16)	378.64 (±18.16)
	Decision Tree	**682.52** (±11.23)	328.55 (±3.16)	273.10 (±3.25)	626.37 (±23.85)
	Naive Bayes	10.75 (±1.23)	**64.28** (±4.60)	0.00 (±0.00)	0.00 (±0.00)
	Random Forest	**730.52** (±15.73)	380.53 (±5.69)	682.28 (±6.68)	701.54 (±13.16)
	Gradient Boosting	551.29 (±14.98)	423.77 (±5.85)	458.93 (±2.40)	**623.71** (±12.76)
	Neural Network	**679.83** (±16.86)	613.54 (±21.69)	601.82 (±16.32)	621.11 (±22.16)
	Fair-Aware1	0.00 (±0.00)	0.00 (±0.00)	0.00 (±0.00)	0.00 (±0.00)
	Fair-Aware2	**7.87** (±2.00)	0.00 (±0.00)	0.00 (±0.00)	0.00 (±0.00)
Bank	Logistic Regress.	**132.41** (±22.12)	58.73 (±24.12)	23.00 (±5.06)	12.21 (±1.02)
	Decision Tree	**239.26** (±18.44)	209.81 (±14.14)	200.42 (±5.29)	17.00 (±1.11)
	Naive Bayes	61.64 (±7.34)	**174.41** (±8.33)	85.63 (±2.18)	43.12 (±8.78)
	Random Forest	**328.71** (±22.02)	256.00 (±8.96)	98.47 (±4.76)	30.67 (±1.71)
	Gradient Boosting	**254.48** (±4.54)	162.22 (±14.44)	112.27 (±5.02)	0.00 (±0.00)
	Neural Network	**257.20** (±21.92)	129.57 (±3.5)	122.86 (±5.28)	239.12 (±36.81)

models trained on the credit, titanic, and bank datasets, where we see 7, 7, and 8 cases our approach generates more unfair cases. Further, we can make one important observation. We could find unfair cases in both of the two fair-aware models. In conclusion, these two fair-aware algorithms might in some cases generate unfair models.

In evaluating fairness through awareness property, we see in all the cases our approach outperforms ART in terms of finding unfair cases. Since there are no other approaches available for testing this fairness property, we could only compare our results with ART. Compared to MLCHECK, we have given a larger timeout for ART, however, even with that, ART could not find more unfair cases. We can conclude that our approach, with respect to our experimental setup, is the best to validate the fairness-through awareness property.

Finally, we also see that in some cases our approach with the neural network performs better than with a decision tree. For instance, in case of the model generated using the Fair-Aware1 algorithm, MLCHECK with decision tree could not find unfair cases, whereas with the neural network it could find in average 5.70 and 19.89 number of unfair cases for adult and credit datasets respectively. We could also see for most of the datasets trained on the naive Bayes model, MLCHECK with neural network generates more unfair cases than the others. This shows that the use of a neural network as the white-box model in our approach in fact pays off.

Efficiency. Figures. 7.2 and 7.3 show the runtime comparisons while checking

Table 7.8: Mean (± SEM) number of unfair cases for fairness through awareness

Datasets	**Classifiers**	MLC_DT	MLC_NN	ART
Adult	Logistic Regress.	**155.63** (±14.03)	65.21 (±11.74)	30.20 (±1.21)
	Decision Tree	**146.00** (±8.28)	42.80 (±4.87)	37.99 (±1.31)
	Naive Bayes	**105.40** (±8.93)	67.4 (±3.93)	55.83 (±2.19)
	Random Forest	**96.41** (±12.83)	60.00 (±5.88)	20.73 (±2.94)
	Gradient Boosting	**113.23** (±7.24)	88.25 (±7.88)	39.19 (±2.01)
	Neural Network	**352.8** (±17.75)	167.40 (±11.27)	23.5 (±1.2)
Credit	Logistic Regress.	124.23 (±27.93)	**218.95** (±23.25)	56.10 (±10.11)
	Decision Tree	**655.47** (±24.66)	571.72 (±21.85)	79.98 (±18.23)
	Naive Bayes	51.23 (±8.11)	**234.50** (±27.77)	20.87 (±3.33)
	Random Forest	224.97 (±12.31)	**400.00** (±15.12)	55.78 (±6.76)
	Gradient Boosting	**307.87** (±21.00)	287.99 (±18.83)	78.62 (±9.19)
	Neural Network	**410.91** (±22.54)	441.52 (±17.73)	88.89 (±16.61)
Bank	Logistic Regress.	**461.26** (±13.20)	221.62 (±11.48)	51.87 (±4.76)
	Decision Tree	**243.62** (±11.54)	129.75 (±14.84)	89.27 (±4.18)
	Naive Bayes	**313.76** (±16.83)	203.39 (±17.54)	28.95 (±7.37)
	Random Forest	**301.48** (±5.55)	146.55 (±12.97)	40.72 (±3.83)
	Gradient Boosting	**162.49** (±7.88)	141.28 (±11.37)	38.88 (±6.40)
	Neural Network	**203.45** (±19.45)	141.87 (±16.87)	44.12 (±3.98)
Titanic	Logistic Regress.	**364.66** (±4.87)	117.52 (±5.63)	88.81 (±3.45)
	Decision Tree	**550.63** (±8.53)	481.93 (±9.12)	87.36 (±3.64)
	Naive Bayes	**309.73** (±13.84)	228.65 (±11.72)	40.42 (±8.88)
	Random Forest	**380.77** (±3.83)	127.54 (±11.95)	82.91 (±2.22)
	Gradient Boosting	**322.65** (±12.73)	100.85 (±14.86)	74.87 (±9.53)
	Neural Network	**213.85** (±12.56)	131.53 (±13.88)	77.20 (±5.76)

individual discrimination and fairness through awareness properties respectively. The x-axis here gives the number of *solved tasks*, ordered by the runtime for each tool from the fastest to slowest. Thus, we only considered the models on which the corresponding testing tool found a violation. Since SG and AEQUITAS could not be configured to apply on one of the fair-aware models (Fair-Aware1), their curves end at 26 tasks.

We can see in evaluating individual discrimination property, for a number of tasks, our approach has higher runtimes compared to SG and AEQUITAS. However, the differences are not significantly large and we have already seen that we could generate more unfair cases at the cost of a slightly higher runtime. Moreover, AEQUITAS although initially for some test cases shows lower runtimes, for the rest, has the highest runtimes compared to all other approaches. This contributes to the usage of initial random search for finding unfair test cases by AEQUITAS. SG on the other hand, learns a decision tree path (but not an entire decision tree), and thus shows a bit lower runtime than our approach. On the other hand, in evaluating fairness-through awareness property, we see the runtimes of ART are much higher compared to our approach. This is again due to the random generation of test cases by solving inequalities.

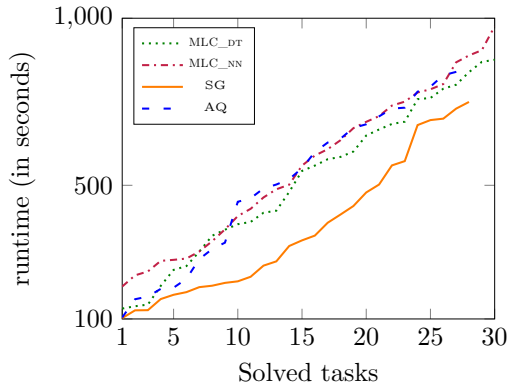


Figure 7.2: Runtime for checking fairness (Def. 7.1)

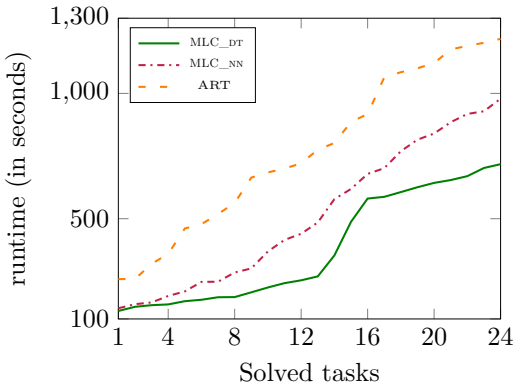


Figure 7.3: Runtime for checking fairness (Def. 7.2)

7.3.2 Monotonicity

For testing the monotonicity property (Definition 7.6) on the classification models, we took 8 algorithms from the `scikit-learn` library and 2 monotonicity-aware algorithms from LightGBM[lig19] and XGBoost[CG16] libraries. The algorithms we have taken from `scikit-learn` are k-NN, logistic regression (Log), naive Bayes (NB), support vector machine (SVM), neural network (NN), random forest (RF), Ada boost (AB), and gradient boosting (GB). These algorithms belong to the most basic family of classification algorithms and the others are derived from these. The monotonicity-aware algorithms taken from LightGBM and XGBoost libraries are specifically designed to learn monotone models with respect to a given set of features from the training dataset. Thus, they are excellent benchmarks for evaluating any monotonicity testing approach. This is due to the fact that, if such a classifier performs as intended there would be no non-monotonicity cases and, even if there are such cases, the number of them would be very few and thus, poses a challenge for the testing approach to detect such cases. In summary, we considered 100 classification models to test the monotonicity property.

As the baseline tools for testing this property, we considered the adaptive random testing (ART) approach and the property-based testing (PBT) tool Hypothesis. Since there are no testing approaches available for testing the monotonicity property of the black-box machine learning models, we used these two as the baseline tools. Moreover, since monotonicity is a hyper-property, as part of ART we implemented the distance metric we proposed in Section 7.2.4.

Results

Effectiveness. Table A.4 shows the results of the monotonicity violations over 10 classification models, for 10 datasets, comparing the three testing tools, MLCHECK with decision tree (MLC_DT) and neural network (MLC_NN), adaptive random testing (ART) [CLM04], and the property-based testing (PBT) tool Hypothesis [hyp23]. In this table, the ✓ indicates that a non-monotonicity case is found for that corresponding dataset and the classifier, and ✗ indicates that such a case is not found. For instance, in the first row of Table A.4 the ✓ corresponding to the classifier k-NN and the adult dataset represents that a non-monotonicity case has been found by MLCHECK (MLC) on the k-NN model trained on the adult dataset. We further summarize the overall

Table 7.9: Overall non-monotonicity detection

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

number of non-monotonicity detections in Table 7.9. Note that the ground truth defining which models are non-monotone is unknown. However, the results reported here with the non-monotonicity cases found in all the models are all true positives.

The results in Table 7.9 show that MLcheck could find violations in overall 68 cases, whereas baseline tools ART and PBT could find 58 and 42 such cases. Thus, we see that ART with our proposed distance metric achieves better results than the PBT tool Hypothesis. One interesting observation we have from Table A.4 is that there are some cases where monotonicity violations are not detected by MLcheck (neither with decision tree nor with neural network), however, are detected by ART and PBT. To take a closer look at these cases and to find out how many of such cases are there, we plot a Venn diagram in Figure 7.4 to show the distribution of all the violated cases found by all three tools.

In this figure, we see that there are 26 models for which monotonicity violations are detected by all three tools. Then there are 24 models for which non-monotonicity cases are detected by both MLC and ART and for 8 models MLC and PBT both could find out non-monotonicity. More importantly, the diagram further shows that there are 9 models for which the monotonicity violations cannot be detected by our approach MLC, and can only be detected by either ART or PBT or, both. These are the models trained on ERA (6 models) and ESL datasets. The models generated on these two datasets have extremely low accuracy rates. For instance, the models trained on the ERA dataset have accuracy rates of around 0.5 and for the ESL dataset, the models have accuracies around 0.6. Thus, learning from these two datasets, in general, are difficult, and when our white-box model (either decision tree or neural network) approximates a model generated on either of these two datasets, the learned model barely approximates the MUT and thus we get poor performance.

Finally, the models generated by the monotonicity-aware algorithm LightGBM were found to be violating the property for two models. This was only found by MLcheck which none of the other tools could find. This furthermore shows the effectiveness of our tool.

Efficiency. We compare the efficiency of our approach with ART and PBT tools by comparing the runtimes in Figure 7.5. Here, the x-axis enumerates the solved tasks which in this case are 100 MUTs that we considered for checking monotonicity and these are sorted in ascending order of their runtimes. The y-axis gives the runtimes for

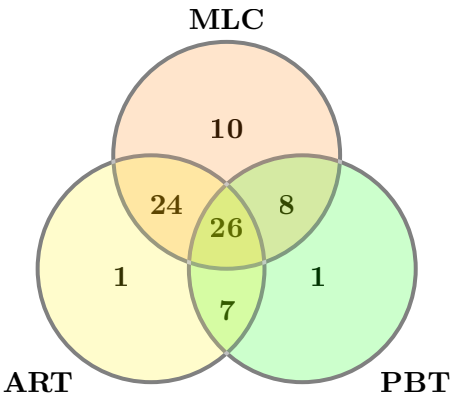


Figure 7.4: Venn diagram

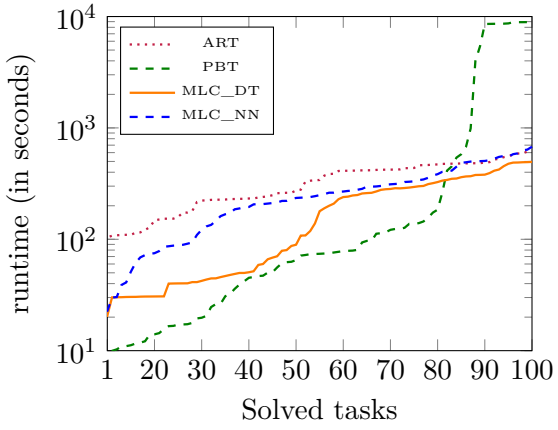


Figure 7.5: Runtime in checking monotonicity

testing the corresponding task. The figure shows that for most of the tasks the PBT tool Hypothesis has the lowest runtime, followed by MLC_DT, MLC_NN, and ART. However, for some MUTs, PBT shows a high runtime and thus, we see a sudden rise of the curve. These are the cases for which the monotonicity violation is not found by the tool even after executing for a long time. In case of ART, it requires the most amount of time in finding non-monotonicity cases because of the cost of generating test inputs which are the furthest away from each other. With respect to MLCHECK, we get a lower runtime with the decision tree compared to the neural network model.

7.3.3 Security

For testing the security property trojan attacks, we generated models on the poisoned dataset which is generated by adding the poisoned data instances to the original MNIST hand-written recognition image data. To this end, we considered two neural network models considering two different architectures, NN1 and NN2. The NN1 neural network contains 3 layers with each layer containing 50 neurons and thus having $50 \times 50 \times 50$ neurons. The NN2 network contains 4 layers with each layer containing 50 neurons and thus having $50 \times 50 \times 50 \times 50$ neurons in total. Both of these networks were then trained on two MNIST datasets elevated by 1,000 and 10,000 poisoned training instances containing specific triggers with the corresponding target labels.

For instance, first, we added 1,000 poisoned instances to the original dataset considering a specific trigger T1 and its corresponding target label as 4. We then trained two different neural network algorithms to generate two models NN1 and NN2 on this poisoned training dataset. Then we added 1,000 instances to the original datasets with the trigger T1, however, this time with the target label as 5. We then similarly generated two models NN1 and NN2 on this dataset. This process is then repeated for all the other triggers T2, T3, and T4 (as shown in Figure. 7.1 on page 113) considering the target predictions for each of the triggers as 4 and 5. Thus, in total, in case of 1,000 poisoned instances, we got 16 different poisoned models. We furthermore repeated this entire process by considering 10,000 poisoned instances to be added to the original dataset. Thus, in total, we test the trojan attack property on 32 different poisoned models.

Table 7.10: Detected violations of trojan attacks (✓ = violation, ✗ = no violation) (data set with 1,000 poisoned instances)

Trigger	**Classifiers**	MLC_DT	MLC_NN	PBT	ART
T1-4	NN1	✗	✓	err	✓
	NN2	✗	✗	err	✓
T1-5	NN1	✓	✓	err	✓
	NN2	✗	✗	err	✓
T2-4	NN1	✓	✓	err	✗
	NN2	✗	✓	err	✓
T2-5	NN1	✓	✓	err	✗
	NN2	✗	✓	err	✓
T3-4	NN1	✗	✓	err	✓
	NN2	✗	✓	err	✓
T3-5	NN1	✓	✓	err	✓
	NN2	✗	✗	err	✓
T4-4	NN1	✗	✓	err	✓
	NN2	✗	✓	err	✓
T4-5	NN1	✗	✓	err	✓
	NN2	✗	✓	err	✓
Overall		4	13	-	14

The idea is to find out when the trigger values are present in a given instance whether the output given by the poisoned model corresponding to that instance is not the desired label. In other words, we aimed to find out cases where the trojan attack on the poisoned model failed. Furthermore, since there does not exist a testing tool for validating this property, we use adaptive random testing (ART) and property-based testing tools (PBT) as our baseline tools.

Results

Effectiveness. Tables 7.10 and 7.11 show the results of our experiments for trojan attacks with 1,000 and 10,000 poisoned instances respectively. We considered four triggers T1, T2, T3, and T4, and for each of them, we considered the target labels as 4 and 5. For example, trigger 1 with target labels 4 and 5 are denoted as T1-4 and T1-5 respectively. The PBT tool Hypothesis was not able to generate any test cases, and after running for a long time, it ended up with the error message "hypothesis.errors.Unsatisfiable: Unable to satisfy assumptions of hypothesis". We suspect, since we have 100 features as inputs and thereby 100 pre-conditions on the feature values, this caused the tool to crash. The developer of the PBT tool possibly could not envisage the use of 100 inputs for a MUT.

Our baseline approach ART turned out to be performing quite well in comparison to our approach MLC_DT and MLC_NN. We see that ART performs better than MLC_DT, however, shows comparable performance to MLC_NN on the models generated on the training dataset containing 1,000 poisoned instances. In case of models generated on 10,000 poisoned data instances (where finding the violations to the Trojan attack property is difficult), our approach with the neural network model could find out overall 6 violations whereas ART could find 5.

Efficiency. Figure 7.6 shows the runtimes comparing MLcheck with decision tree and neural network to ART approaches in finding out a single violation of the trojan

Table 7.11: Detected violations of trojan attacks (✓ = violation, ✗ = no violation) (data set with 10,000 poisoned instances)

Trigger	**Classifiers**	MLC_DT	MLC_NN	PBT	ART
T1-4	NN1	✗	✓	err	✓
	NN2	✗	✓	err	✓
T1-5	NN1	✗	✓	err	✓
	NN2	✗	✗	err	✗
T2-4	NN1	✗	✓	err	✓
	NN2	✗	✗	err	✗
T2-5	NN1	✗	✗	err	✓
	NN2	✗	✗	err	✗
T3-4	NN1	✗	✗	err	✗
	NN2	✗	✓	err	✗
T3-5	NN1	✗	✗	err	✗
	NN2	✗	✗	err	✗
T4-4	NN1	✗	✗	err	✗
	NN2	✗	✓	err	✗
T4-5	NN1	✗	✗	err	✗
	NN2	✗	✗	err	✗
Overall		0	6	-	5

attack. X-axis shows the number of models tested for violation (i.e., solved tasks) in ascending order of the runtimes needed to test the corresponding task. Y-axis gives the corresponding runtimes. In this case, we show the runtimes considering the 1,000 poisoned instances (based on the results from Table 7.10). It shows that the performance of ART comes with the cost of high runtimes. Furthermore, as we have also seen previously, MLCHECK with the decision tree gives lower runtimes compared to neural network model. This is again attributed to the high runtimes in SMT solving of the neural network. However, in this case, MLCHECK with neural network gives superior performance at the cost of a higher runtime.

7.3.4 Concept Relationship

For validating this property, we first collected 6 datasets (CR1-CR6) by using the PYKE embedding approach [DN19] from the DBpedia knowledge graph. Then we used these datasets to train multi-label random forest and neural network algorithms. For random forest, we used the default hyper-parameter settings from the `scikit-learn` library and for the neural network we considered the architecture containing 3 layers, with each layer containing 50 neurons and thus in total $50 \times 50 \times 50$ neurons. Thus, with 6 datasets and 2 multi-label classification algorithms, we generated in total 12 models. Based on the datasets generated using the DBpedia knowledge graph, we considered 3 different concept relationships, one of which is a subsumption relationship (S1) and the other two are disjoint relationships (D1, D2) which can be defined as:

- Every actor is a person (S1)
- A planet is not a person (D1)
- A celestial body is not a person (D2)

Table 7.12: Detected violations of concept relationship (✓ = violation, ✗ = no violation)

Dataset	Classifiers	MLC_DT S1/D1/D2	MLC_NN S1/D1/D2	PBT S1/D1/D2
CR1	Neural network	✓/✗/✓	✓/✗/✓	✗/✗/✗
	Random forest	✗/✗/✗	✗/✗/✗	✗/✗/✗
CR2	Neural network	✓/✓/✓	✓/✓/✓	✓/✓/✓
	Random forest	✗/✗/✗	✗/✗/✗	✗/✗/✗
CR3	Neural network	✓/✓/✓	✓/✓/✓	✓/✓/✓
	Random forest	✗/✗/✗	✗/✗/✗	✗/✗/✗
CR4	Neural network	✓/✗/ ✓	✓/✗/✓	✗/✗/✗
	Random forest	✗/✗/✗	✗/✗/✗	✗/✗/✗
CR5	Neural network	✓/✓/✓	✓/✓/✓	✓/✓/✓
	Random forest	✗/✗/✗	✗/✗/✗	✗/✗/✗
CR6	Neural network	✓/✓/✓	✓/✓/✓	✓/✓/✓
	Random forest	✗/✗/✗	✗/✗/✓	✗/✗/✗
Overall		6/4/6	6/4/7	4/4/4

In this case, the datasets (CR1-CR3) contain three classes (i.e., three concepts): actor, planet person and the datasets (CR4-CR6) contain three classes: actor, person, celestial body. Since this property is not tested before, we used the PBT tool Hypothesis as the baseline tool for this.

Results

Effectiveness. Table 7.12 shows the results of evaluating one subsumption and two disjoint relationship properties. For each dataset, we give the results corresponding to the neural network and the random forest model trained on the corresponding dataset. The results show that our approach with both neural networks and decision tree models perform mostly the same for S1 and the D1 properties in finding out 6 and 4 violations respectively, whereas the PBT tool could find 4 violations for S1 and 4 for D1. In case of the D2 property, MLcheck with decision tree could find 6 violations and with neural network 7 violations, whereas PBT tool could find 4 such cases. Furthermore, there are no unique cases of violations found by the PBT which could not be found by our tool.

Efficiency. Figure 7.7 shows the runtimes for testing the concept relationship. Here on the x-axis, we have 36 tasks, which come from 2 classifiers trained on 6 datasets and testing 3 different relationship properties and thus $2 \times 6 \times 3$. These tasks are ordered based on the increasing runtimes of the three tools used for testing. As can be seen, MLcheck with decision tree requires the least amount of time in testing while PBT and MLcheck with neural network have comparable runtimes.

7.3.5 Properties of Regression Models and Aggregation Functions

We have described the regression models and the aggregation functions in Section 7.2.3 and their properties in Section 7.2.1. To generate the regression models, we used the ecoli dataset taken from [MH19] and trained four regression algorithms (described in Section 7.2.3) to generate four regression models. Apart from the regression models,

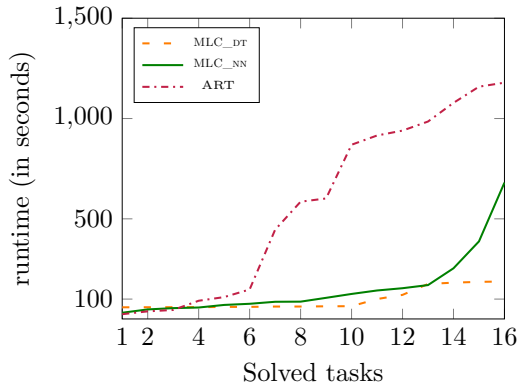


Figure 7.6: Runtime for trojan attacks

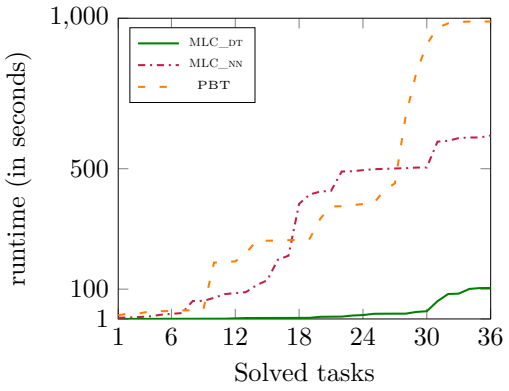


Figure 7.7: Runtime for concept

we also evaluated MLCHECK in testing 10 aggregation functions. In summary, we tested 12 different properties on 4 regression models and 10 aggregation functions. Note that the results reported here are obtained using MLCHECK with decision tree. Since MLCHECK with neural network does not give notably different results, we omit this case.

Results

Effectiveness. Tables A.5 (in page 153) and 7.13 show the results of our evaluations in testing 12 different properties on 10 aggregation functions and four regression models respectively. For all the aggregation functions we know the ground truth, meaning we could say whether a property is present in the function. We indicate this by marking a cell as blue. Thus, a blue-shaded cell indicates that the property is present in that aggregation function and a cell without color indicates the absence of the property. For example, the aggregation function AM satisfies the strong monotonicity (str. mon.), symmetry (symm.), idempotency (idemp.), internality (intern.), invariance (invar.) and additivity (addit.) properties and does not satisfy Lipschitz (Lipsch.), conjunctivity (conjun.), and disjunctivity (disjun.) properties. Furthermore, the two regression models L-OWA [MH19] and L-Uni [MH16] are designed with the corresponding aggregation functions as their core and thus, we know the ground truth for them also. However, we do not know the ground truth for the learned aggregation functions LAF [PTF+21] and DeepSet [ZKR+17] models since they are purely learned to approximate the aggregation function OWA.

The results show the effectiveness of MLCHECK (MLC) compared to the PBT tool Hypothesis for both the aggregation functions and the regression models. For the aggregation functions, MLCHECK could find a test input violating the corresponding property in 51 cases, whereas the PBT tool could find 41 such cases. Note that, all these violations generated by both of these approaches are true positives, i.e., in all of these cases the property is not satisfied by the corresponding aggregation functions. Furthermore, MLCHECK could find all those cases where the property is not present in the aggregation functions.

We report the results for the learned aggregation functions or the regression models in Table 7.13 where MLCHECK could find 32 violations and the PBT tool could find 24 such cases. As mentioned beforehand, for 2 of the regression models, LAF and DeepSet,

Table 7.13: Detected violations for learned (aggregation) functions (✓ = violation, ✗ = no violation)

	L-OWA MLC/PBT	**L-Uni** MLC/PBT	**LAF** MLC/PBT	**DeepSet** MLC/PBT	**Total** MLC/PBT
mon.	✗/✗	✗/✗	✓/✓	✓/✓	**2/2**
infi.	✗/✗	✗/✗	✓/✓	✓/✓	2/2
supr.	✗/✗	✗/✗	✓/✓	✓/✓	2/2
str. mon.	✓/✓	✓/✗	✓/✓	✓/✓	**4/3**
Lipsch.	✓/✓	✓/✓	✓/✓	✓/✗	**4/3**
symm.	✗/✗	✗/✗	✗/✗	✗/✗	0/0
idemp.	✗/✗	✓/✗	✓/✓	✓/✓	**3/2**
conjunc.	✓/✓	✗/✗	✓/✓	✓/✓	3/3
disjunc.	✓/✓	✓/✓	✓/✓	✓/✗	**4/3**
intern.	✗/✗	✓/✗	✓/✓	✓/✓	**3/2**
invar.	✗/✗	✓/✗	✓/✓	✓/✓	**3/2**
addit.	✓/✓	✗/✗	✓/✗	✓/✗	**3/1**
total	5/5	**6/2**	**11/10**	**10/7**	**32/24**

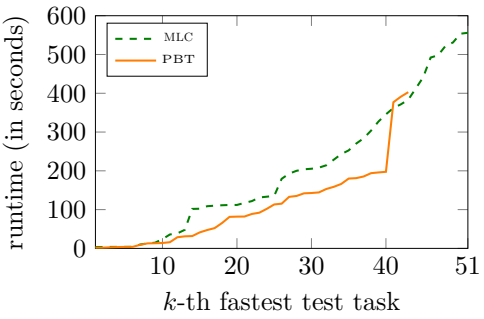


Figure 7.8: Run-time in sorted order

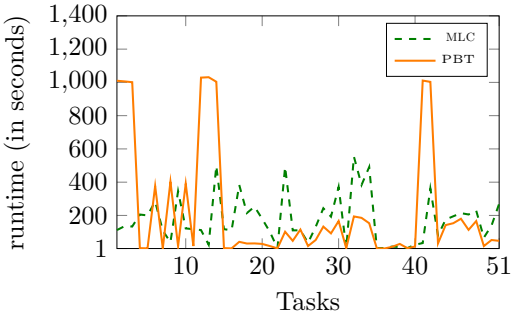


Figure 7.9: Comparison for each task

the ground truths are not known, and hence, we cannot say for sure whether our tool was able to find all the cases where the property was not present. Most importantly, as can be seen in the Table 7.13, we could find LAF and DeepSet not satisfying the mandatory properties, i.e., monotonicity (mon.), infimum (inf.), and supremum (sup.), required to be satisfied by any aggregation function. This gives an important result that these two learned models cannot be considered as aggregation functions. This is a valuable outcome and shows the applicability of MLcheck in finding out whether a learned model can be used as an aggregation function. This information can be further used by the ML community to better develop such models.

Efficiency. We report the efficiency of MLcheck (MLC) in comparison to PBT in Figures 7.8 and 7.9 in finding out the violations of only the aggregation functions. We have only considered the cases where a function does not satisfy a property. Therefore, in x-axis we have 51 tasks in total and the curve for MLC is extended till 51 and for PBT it ends in 41 (in Figure 7.8). This figure suggests that PBT has lower average runtimes

for its k fastest tasks and there are 10 tasks for which PBT times out. Thus, we made another comparison which shows the task by task comparisons between MLC and PBT, depicted in Figure 7.9. It shows the cases with high spikes for PBT where it times out. For the rest of the cases we see comparable runtimes between two approaches.

7.3.6 Discussions

In summary, we have applied our approach to 202 different models (including 10 aggregation functions) while validating 20 different properties. We have found that in most of those cases our approach performs better than the baseline tools we have considered in this thesis. Having said that, since our approach is *generic*, it might not perform as well as a specific technique designed to test a specific type of model or a specific property. For instance, the fairness testing tool specifically designed for deep neural networks by Zhang et al. [ZWS+20] would definitely generate more unfair test cases on a given deep neural network model, compared to our approach. Similarly, the quantitative verification approach proposed to check trojan attacks on binarized neural networks by Baluta et al. [BSS+19] would give better results in finding attack violations than ours. However, since our aim was to build a testing tool that could be used for testing any ML models, and for any specifiable properties, we compromise the specialization–focusing on a specific ML model or a property–with the generalization in our testing tool. Although despite being a generic testing framework, our tool was able to outperform state-of-the-art fairness testing tools such as AEQUITAS and SG. Moreover, MLCHECK has been shown to be performing better than adaptive random testing and the existing property-based testing tool Hypothesis.

Soundness. MLCHECK can be considered sound since it only generates *true positives*, meaning that the counter-examples generated by our tool are all valid ones. More specifically, since our approach involves learning a white-box model from the MUT and then applying SMT solving technique to it, we first generate the counter-examples on the learned white-box model and not on the MUT. However, the counter-examples that are generated are then validated on the MUT, and only those which are found to be valid with respect to the property are returned. Furthermore, using the `bound_cex` parameter, we can include the range of permissible values for the input features (as described in the XML configuration file, see Appendix A in page 149) while generating counter-examples. This is added as constraints to the logical formula describing the model and thus, the SMT solver generates counter-examples satisfying the constraints defined for the range of values for individual features. For example, the value of gender can only be 0 or 1, and just by activating `bound-cex`, we could add this as a constraint to the SMT formula and include it during the process of counter-example generation.

Scalability. As mentioned beforehand, our approach involves an SMT solving approach to generate counter-examples as violations to the property on the MUT. Since the satisfiability solving problem is considered to be NP-complete problem, our approach requires a large runtime if the input vector size is large or if the input values consist of rational numbers. For instance, our approach can probably not be applied to the image classifiers which take input images containing thousands of pixels. We have seen in the experiments involving trojan attacks that even with 10×10 pixels as input, our approach became quite slow (see Figure 7.6). However, we did not intend to build MLCHECK for image classifiers, for which a number of specialized tools are available (such as [PCYJ17, TZO+20]), rather we aimed to develop a testing tool that

could be used for testing any classifiers or regression models. As mentioned before, we compromise specializations for generality.

7.4 Internal Evaluation

Apart from evaluating MLcheck in validating a number of properties on the MUTs, we have also performed some internal evaluations. To this end, we give here the experimental evaluations comparing two aspects of MLcheck: (a) comparing the performance of the white-box models in validating different properties on the MUTs, (b) comparison of two pruning approaches to find out their test case generation abilities. Below we describe them in more detail.

7.4.1 Model Comparison

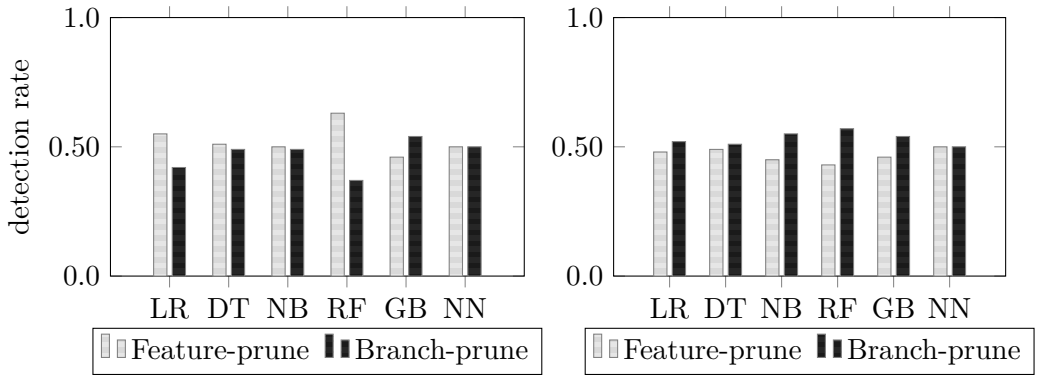
Since our approach allows to use of two types of white-box models to learn from the MUT, we are interested to find out which one performs better than the other in finding out the violations of the properties. In our external evaluation, we have already shown this comparison between decision trees and neural network models. We now discuss these results here by taking another look at those tables and figures.

In case of fairness properties where we intend to find out a number of unfair cases, the decision tree performs better than the neural network (NN) in most of the cases. For instance, MLcheck with decision tree was able to find out the most number of unfair instances in 38 MUTs whereas, MLcheck with NN was able to find out the most unfair instances for 11 MUTs, as can be seen in Tables 7.7 and 7.8 for two fairness properties. The difference in these numbers is due to the fact that the SMT solving of the neural network model requires a larger runtime compared to decision tree model which is attributed to the large number of arithmetic operations required in NN to generate the output prediction for a given input. Moreover, these operations involve floating point numbers. Thus, a combination of both of these two slows down MLcheck while using neural networks. We can also see this when we compare the runtimes in finding violations for monotonicity in Figure 7.5, concept relationship in Figure 7.7, and trojan attack in Figure 7.6.

After seeing these high runtimes in comparison to the decision tree, one might ask whether it is required to use neural network at all as an alternative option for the white-box model. This can be answered by considering two cases. First, although we got a low number of unfair instances when we used NN, for the Fair-Aware1 model, for which the decision tree was unable to generate any unfair instances, MLcheck with NN was able to generate some, as shown in Table 7.7. Secondly, in case of trojan attack property, we can see that MLcheck with NN performs better than the decision tree in case of both 1,000 and 10,000 poisoned instances as can be seen in Tables 7.10 and 7.11. The performance gain over decision tree by neural network does not come with the cost of a really high runtime. Thus, we can say that usage of NN as the secondary white-box model is an added advantage and two models in combination makes our approach much more effective rather than using a single one.

7.4.2 Pruning Comparison

To generate a number of counter-examples instead of a single one from a single SMT query, we have employed two techniques in MLcheck, instance and branch pruning.



(a) Individual discrimination (b) Fairness through awareness

Figure 7.10: Performance comparison of branch and feature pruning

This basically serves us with two purposes: (a) retrain the white-box model more effectively with a number of counter-examples, (b) if multiple violating test cases (such as in fairness) are required to generate for a property, we could use pruning. In this evaluation, we aimed to find out which pruning strategy performs better compared to the other in terms of generating violated test cases. For the evaluation, we have only considered the decision tree as the white-box model since on the neural network model the branch pruning technique cannot be applied. Furthermore, we have considered here 4 properties: 2 fairness properties, individual discrimination and fairness through awareness (Definition 7.1, 7.2), and 2 monotonicity properties. In Definition 7.6, we defined a monotonicity property for the classification models. We term this as *weak* monotonicity and furthermore consider a stronger definition of this property taken from [SW20b] for the evaluation of pruning comparison. We term this as *strong* monotonicity which is defined as follows:

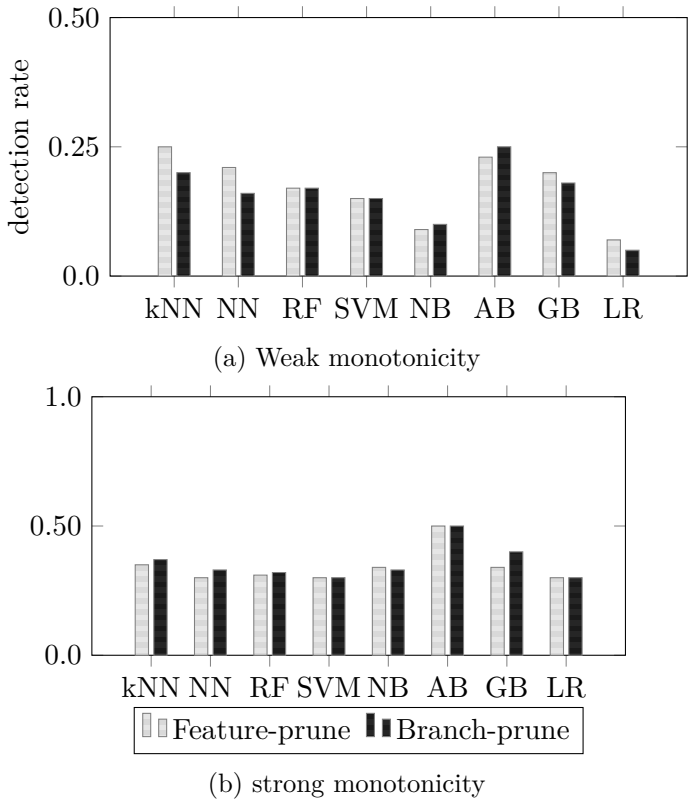
Definition 7.18 *A classification model M is* strongly monotone[18] *with respect to a feature i if for any two data instances $\vec{x}_1$, $\vec{x}_2 \in \vec{X}$ we have $\vec{x}_1(i) \preceq_i \vec{x}_2(i)$ implies $M(\vec{x}_1) \preceq_Y M(\vec{x}_2)$.*

In comparison to the Definition 7.6, this definition does not require the features except i to have the same or increasing values.

To compare the two pruning approaches, we have used *detection rate*, which is defined as the number of violations detected, divided by the total number of test cases generated, i.e., $\frac{\#\text{violations}}{\#\text{test cases}}$. Finally, as it is clear from the properties we mentioned, we performed this evaluation only on the classification models. For this, we considered 14 classification models.

Figures 7.10 and 7.11 show the detection rates of two pruning strategies for fairness and monotonicity properties respectively. We see in Figure 7.10 that for individual discrimination property, on a majority of the classifiers, instance pruning performs better than the branch pruning. On the other hand, in case of fairness through awareness, we see an opposite trend. In case of monotonicity, we found that for strong monotonicity, branch pruning performs better than instance pruning, and instance pruning performs better in case of weak monotonicity. Thus, we can see a sort of pattern be-

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



(a) Weak monotonicity

(b) strong monotonicity

Figure 7.11: Performance comparison of branch and feature pruning

tween these two results of two different properties. Since individual discrimination, or strong monotonicity requires most of the feature values in the instance pair to be the same, branch pruning cannot really explore different parts of the decision tree easily. Thus, the full potential of branch pruning cannot be utilized in these cases and hence, instance pruning outperforms the branch pruning. When the features values are not needed to be equal, such as in fairness through awareness and strong monotonicity properties, branch pruning shows better performance. Thus, we can conclude that we require both of these pruning approaches to be performing in combination in order to achieve the best results out of our approach.

7.5 Limitations and Threats to Validity

Our approach requires learning a white-box model from a given model under test (MUT) and for this process, we require training data. Since our aim is to learn the behavior of the MUT, to this end, we query the MUT using instances generated randomly and for each of the queries, we store the result. Thereby, we get a data set containing input-output pairs which form the required training data. The set of such instances and their corresponding outputs are later used to train either a decision tree or a neural network algorithm to generate the corresponding model which is said to be approximating the MUT. Thus, we can capture the behavior or approximate the MUT only if the randomly generated data instances cover most of the input region of the MUT. However, since this process is random, we cannot guarantee to generate

instances that are spread out well in the input space of the MUT and give us a good approximation of the space. To mitigate this, we have set the number of randomly generated instances high enough, such that the input space should be covered assuming that it can be covered uniformly—this number can of course be (re-)configured by the the user. For instance, if the user does not find any counter-example for a specified property, a counter-example might be found in another try for which this number has been increased. This offers potential for future improvements of the technique, for example, by raising the question: how to minimize the number of instances to generate while approximating a model accurate enough to argue about certain properties?

Furthermore, the solver cannot take into consideration the input distribution of the MUT and generates test inputs as counter-examples by considering a uniform distribution of the input. However, in reality, this might not be the case, since any ML model which is learned on the specific dataset might have a specific input distribution. Thus, while operating in the use case scenario the model might expect the data coming only from that specific data distribution. Hence, it would be more meaningful to check ML models considering the input data only coming from a specific distribution and discard *out-of-distribution* data. However, this is a difficult requirement to specify in the SMT formula while generating counter-examples as test cases. With respect to MLCHECK, we allow to set the parameter `bound_cex`, which would then be used to generate test inputs from a specific range of values. A more sophisticated approach would be to incorporate *auto-encoder* [XKN22] to detect out-of-distribution data, which could be considered as a potential future work.

In our approach, one core part is embodied in the usage of an SMT solver. Since SMT solvers are limited to deal with integers and real numbers, our approach shares this limitation. However, this can be mitigated by replacing categorical values with numerical ones, e.g., a category such as gender can be converted by using an enumeration: 0. male, 1. female. In the ML domain this is frequently done by using methods like *label*, *binary*, *one hot* or *count* encoding techniques which are generally performed as data *pre-processing* steps, before the training of the model. In our evaluation, we have applied the label encoding approach to the datasets to convert categorical features into numerical ones before generating the MUTs.

The specification language we have proposed to specify the properties is a simple domain-specific language that could only be used to specify *non-stochastic* properties. Even if we could modify our specification language to allow to specify stochastic properties, since our approach uses an SMT solving technique to generate test cases, we cannot check such properties. For instance, the statistical fairness definitions such as *statistical parity* [RT16] require to have equal probabilities to have the same predictions for two specific feature values of a protected attribute (for example male and female). This cannot be checked with our approach.

An internal threat to validity could be the high degree of randomness involved in our approach. For instance, first, we generate a number of data instances randomly to generate the training dataset which is then used to get the white-box model. Even if this training dataset remains the same in 2 runs, the learned white-box model can still be different. Thus, properties, for which we need to generate multiple violated test cases (for example in case of fairness properties), the two runs of the tool might give different numbers of violations, or in cases where we return after finding a single violation, we might get two completely different outcomes (property violated and not violated). To mitigate these threats, we have performed each experiment 10 times and

the results reported here give the mean over these 10 runs.

The threats to validity of our experimental evaluation could be the choice of the training datasets and the ML algorithms we trained on them. To this end, the datasets we considered were taken from the standard cases which were also considered in the related works. Furthermore, we have taken a number of ML algorithms in our experimental evaluations to generate the models to test. These include simple algorithms like naive Bayes, k-NN, to tree-based algorithms random forest, decision tree, to neural networks. Since we have taken a large number of standard datasets and a variety of ML models in our evaluations we can say that these are diverse enough to reflect real-world cases.

7.6 Related Work

We have already discussed a number of related works in the previous Chapter (Chapter 6), related to the property-driven testing technique (the underlying approach of the MLcheck) we introduced in this thesis. We have stated their limitations and described how our approach can help to bridge the gap. Since in this chapter, we focus on validating different properties on different types of ML models, we first briefly discuss some approaches related to the validation of the properties we have considered in our evaluation. There does not exist a work validating the properties of the aggregation functions we considered. However, these functions are essentially numeric functions and thus, we then describe some related works concerning the testing of such functions.

Fairness. There are a number of works in the literature considering the testing of the fairness property of ML models. Galhotra et al. [GBM17] first proposed a black-box random test generation technique called THEMIS for testing individual discrimination and statistical parity properties. Later Udeshi et al. [UAC18] combined random test generation and exploit the inherent *non-robustness* property of the ML models to develop AEQUITAS. This could generate a number of failed test cases outperforming THEMIS. Our approach is closest to the work of Aggarwal et al. [ALN+19] and a very recent work by Xiao et al. [XLLL23]. In the former work, first, they employ LIME [RSG16] tool to generate a *partial* decision tree, basically a path of the tree. Then the dynamic symbolic execution technique is applied to generate multiple test cases on the tree. In our work instead of learning a path of the tree, we learn an *entire* decision tree approximating the MUT. Then we use the specified property by the tester to generate test cases falsifying the property. Xiao et al. [XLLL23] use generative adversarial networks (GAN) [CWD+18] to generate a dataset approximating the data distribution of the original one (i.e., the dataset which was used to train the MUT). After that, they train a support vector machine (SVM) model with linear kernel on the generated dataset. The SVM model is then said to be approximating the model under test (MUT). The data points that lie near the *decision boundary* of the SVM model are extracted and used to generate test cases violating individual discrimination property. Using GAN to generate a dataset although effective since it reflects the actual input distribution of the MUT, however really costly, and thus, we abstain from using this step. However, as a future work, this could be used as a preliminary step in MLcheck.

There exist other fairness testing approaches focusing on specific types of ML models, ADF [ZWG+21], FairNeuron [ZCD+22], ASTRAEA [SUC22], MT-NLP [MWL20]. ADF and FairNeuron can only be used to test individual discrimination property for deep neural network models. ASTRAEA and MT-NLP on the other hand are

used to test fairness in natural language processing (NLP) models. In a more recent work, Perera et al. [PAT+22] proposed *fairness degree* as the fairness definition for the regression models and gave a search-based technique for testing the property on such models. All these works either focus on a specific category of fairness property or a specific type of ML model and hence, cannot be used for any other types of models and in testing any specified fairness property. Apart from the fairness testing works we discussed here, there exist more of such works. For a detailed survey of fairness testing, we refer the interested readers to look here [CZH+22].

Montonicity. The existing works on monotonicity focus on specific ML models with the aim of making them monotone. For instance, Archer et al. [AW93] first proposed to build monotone neural network models by controlling the number of instances in the training dataset. Dugas et al. [DBB+09] later improved this technique for neural network models by constraining the weights of the model to be non-negative while employing a monotonic activation function. You et al. [YDC+17] proposed to build deep neural networks with multiple layers of lattice in order to build a model which is guaranteed to satisfy monotonicity with respect to a set of features. Later, Liu et al. [LHZL20] proposed to use a *regularizer* in the learning process of neural networks to enforce monotonicity. Apart from neural networks, some existing works also focus on building monotone models considering some other types of models. For instance, Lauer et al. [LB08] proposed to build a monotone support vector machine with linear kernels by constraining the *gradients* of the corresponding feature to be positive within a specific range. We found the work of detecting whether a model is monotone only for the Gaussian model where Siivola et al. [SPV16] proposed to use the *virtual derivative observations* to detect monotonicity. However, this approach can only be used if the ML model to be tested for monotonicity is linear.

Trojan attacks. There have been several works on performing trojan attacks on neural network models by using specific learning strategies (see [LDS+22] for a detailed survey) after Liu et al. [LMA+18] proposed the idea of performing trojan attacks on deep neural network models. Since then, only the work of Baluta et al. [BSS+19] proposed a technique to check trojan attacks on a given model. To this end, they gave a *quantitative verification* technique to verify trojan attacks on the binarized neural networks (BNN). They first encode the BNN into logical formula and then instead of using SMT solving technique, they apply ApproxMC3 [YM21] a specific SAT solver to perform *model counting*. However, their approach is limited to the BNN model and the property needs to be provided manually to check.

The concept relationships in the domain of knowledge graph embeddings [DN19, DN21] occur frequently however, previously not been considered to validate. Since there does not exist any related literature to this end, we do not discuss them.

Black-box ML testing. There is a black-box testing framework which is developed by Aggarwal et al. [ASH+21] for testing ML models with respect to three different properties: fairness, robustness, and model accuracy. Much like ours, they also consider a similar setting, where they assume to have an ML model to test, the internals of which is unknown. In their work, they developed a testing technique considering ML models generated on different types of datasets such as tabular, text, audio and time series data. For testing models generated on the tabular dataset, they use the approach as proposed in [ALN+19] where they learn a path of a decision tree and then apply dynamic symbolic generation method to generate test cases. For other kinds of datasets, they use a metamorphic testing approach to check the model. Thus, depending on

the property to check, they incorporate a specific test generation technique and this process is somehow *hard-coded* with the associated property.

In contrast, our approach is more generic and we allow the tester to specify properties for which the test case generation technique is not hard-coded into our approach. Although our approach can be used to check ML models generated only on the tabular dataset, it can be still used to check a number of properties. Moreover, as we have shown before, the approach used for tabular datasets by Aggawal et al. [Agg18] (called as SG in Table 7.7) is less effective in generating unfair test cases compared to our approach.

Properties of numeric functions. Cox et al. [CHJ+04] proposed a black-box testing technique to validate arithmetic mean, standard deviation, and polynomial regression functions implemented in well-known libraries and software such as Microsoft Excel, Matlab, and some other Java libraries. A web-based facility in this case is used to first generate the *oracle dataset* defining the test inputs and their actual outputs. The test inputs are then executed on the software under test and the produced result is then compared with the result of the oracle. Further works by kempf et al. [KK17] provided a regression testing technique to test such functions of the numerical library DUNE [BHM10] where the test inputs are generated by involving a human expert in that domain.

The work closest to us to this end is the work by Meinke et al. [MN10], where they first learn a number of piece-wise polynomial models from the set of test-input and program output pairs, thus approximating the program under test. Then they apply the CAD algorithm [CJ12] to check the satisfiability of the learned polynomial model. However, the properties they checked on the numerical functions are needed to be provided manually and furthermore, satisfiability checking of such a polynomial function is highly inefficient. Thus, in our work, we take advantage of the state-of-the-art machine learning techniques to learn an ML model approximating the MUT and then use an SMT solving technique to generate test cases.

Regression models. There are works on testing regression models which focus on prioritising generated test inputs [FSG+20], minimising the test inputs [WKRL17], and selecting the test inputs [GKZ+14] for deep learning regressors. For instance, the work by Feng et al. [FSG+20] proposed to prioritize test cases generated beforehand in order to minimise the effort of humans to label whether a test instance is failing or not. The test instances are first generated in this case to check the robustness property of a deep learning regression model. Wolschke et al. [WKRL17] proposed to compare two sets of test inputs to remove the redundant cases for the regression model and thus minimising the test inputs.

The work on verifying gradient boosted regression trees by Einziger et al. [EGSS08] is probably the closest to our work. They proposed a verification technique for the boosted trees model by using SMT solving technique. To this end, they first convert the ensemble of decision trees into SMT formula. To encode each decision tree into the logical formula, they use an encoding technique similar to ours. Then they conjoin the negation of the robustness property to the SMT formula of the boosted trees. Finally, the entire formula is given to the SMT solver to find counter-example to the property. A similar verification technique for neural network regression model is given by Venzke et al. [VC21] where they used the mixed integer linear programming (MILP) approach to encode the model into logical formula and then apply SMT solving technique to check the robustness property on it.

8 Conclusion & Future Work

In this thesis, we presented approaches for testing learning algorithms and the learned models. We subsequently implemented our corresponding approaches; for testing the ML algorithms we developed TiLe and then for testing ML models we developed MLcheck. Essentially in this thesis, we aimed to develop testing tools that could be used to test the ML algorithms before they enter into the learning phase and then test the learned models with respect to specified properties, before deploying the models into the real world.

In this Chapter, we summarize the works done in this thesis. We first give a brief summary of the approaches we presented in this thesis in Section 8.1. Then in Section 8.2, we discuss the results obtained by evaluating our approaches (implemented in the corresponding tools) on ML algorithms and models. Finally, we end this chapter by giving some possible future extensions of the tools we developed: TiLe and MLcheck in Section 8.3.

8.1 Summary

Testing of ML algorithms. With respect to the validation of the ML algorithms, there are no specific requirements. The main reason behind this is that, the expected outcome by a learning algorithm which in this case is a learned model cannot be specified by any requirements, since it is not clear beforehand what the correct learned model is. Thus, in testing terminology, we say the ground truth as a kind of oracle defining the correct output of the learning algorithm is missing. To mitigate this gap, we first defined a property that we termed as balanced data usage or in short balancedness as an essential property of the learning phase. The property requires if we apply row, and column permutations, and feature name shuffling on the training dataset, the generated models before and after applying these transformations should be the same.

We consider the balancedness to be a reasonable requirement that any learning algorithm should guarantee, specifically if we consider how any learning algorithm learns from the data. For instance, in the learning phase, the learning algorithm considers each of the training instances and its corresponding class labels and in the end, it attempts to generate a model which has a low *error* in predicting the class labels of the corresponding training instances. Now, depending on the type of ML algorithms, the process of minimizing the error would be different. For example, in case of the decision tree, this is done by splitting the input space, or in the case of the neural network through using the gradient descent and adjusting the weights and the biases of the network (see Chapter 2 for more details). However, the order of the data instances in any way should not influence this learning process.

We then developed a testing tool TiLe based on the idea of the metamorphic testing approach to test the ML algorithms with respect to the balancedness property. Our approach has two important steps: applying the transformations and checking that

the models are *equivalent* (i.e., the same). For the latter, we proposed a constraint-based technique to compute the equivalency for decision tree models. In case of some other algorithms, we checked the equivalency by comparing the learned parameters (before and after applying the transformations). Then, for the rest, we gave a testing approach to test the equivalency between models. As discussed in Chapter 4, although we expected the ML algorithms to be generating equivalent models before and after applying the transformations we considered, we found a number of algorithms to be actually sensitive to such changes in the training data.

Testing of ML models. Apart from testing the learning algorithms, there is a need to validate the ML models with respect to several types of properties. The existing testing approaches either focus on a specific type of model or on validating a specific type of property. Thus, we aimed to develop a testing technique that allows *model agnostic* testing with respect to the specified property. Our contributions to this end are two folds: developing a testing approach irrespective of the model under test (MUT), and giving a specification language for specifying the property to be checked on the model.

In our thesis, we allow the given MUT to be black-box in nature and we believe this is a realistic assumption. For instance, it might happen that someone wants to use an already trained model for a specific task, however does not want to know the internals (or might not have any access to it), and is only interested in the inputs and outputs. From the testing perspective, this is also a viable scenario where the tester has been asked to validate the given model without being given much knowledge about the model to be tested. This is stated in an industry scenario by the works of Aggarwal et al. [ASH+21] which we mentioned in the related work. In IBM, for instance, they have developed a black-box testing approach to test ML models, the internals of which are unknown beforehand and only the information of the training dataset is available. Moreover, in some cases, the internals of the model if known, can allow the attacker to *attack* the model or the learned parameters of the model might disclose sensitive information. Thus, in those cases, the information regarding the model might not be available to us and our approach can be a perfect tool to be used in such cases for validating the model.

We first developed the verification-based testing approach which is model agnostic and renders the idea of learning-based testing. In this testing approach, generally, an automaton model is learned and then used to generate test cases. However, unlike the existing approaches, the model in our case is a machine learning model and not an automaton. We believe the use of an ML model to learn the MUT (which is also an ML model), instead of an automaton, is a better choice. This is partly because the input space of the MUT might consist of higher dimensional real-valued vectors as input. Moreover, an ML model might better capture another ML model rather than an automaton. Once we learn the model, we translate the model into logical formulas and use the conjunction of the model formula and the negation of the property to generate test cases on the MUT.

Apart from the model agnostic testing approach, we also aimed to develop an approach that could be used to test any specified properties on the MUT. Thus, we developed a domain-specific language tailored to specify properties that frequently occur in the domain of machine learning and hence, are necessary to validate. Since an ML model might need to be validated with respect to several properties before being deployed in a specific domain, the use of a testing approach that allows specifying

properties by the tester can be of real importance. Although our specification language is expressive enough to be used to specify several types of properties, this certainly cannot be used for stochastic properties. Of course, we could have extended the language to allow for specifying such properties, however, the underlying technology for test case generation by using SMT solver would still be a hindrance to generating test cases with respect to such properties. In future work part, we describe it a bit detail how this still could be achieved by appropriately extending our work.

We have combined the idea of property specification language with the verification-based testing to develop the property-driven testing approach for testing ML models. This testing approach is essentially implemented in a tool called MLcheck. To show the applicability of our approach, we evaluated it on different types of ML models, in testing a number of different properties.

8.2 Discussion

We briefly summarize the findings of our experimental evaluations in evaluating our approaches of testing ML algorithms and models.

Evaluation of TiLe. We have evaluated our approach of ML algorithm testing tool TiLe on 23 classification algorithms taken from `scikit-learn`, `WEKA`, `XGBoost`, `LightGBM`, `CatBoost` libraries in testing balancedness property. Surprisingly, we have found many of them to be sensitive to simple row or column permutations and thus violate the balancedness property. In Chapter 4, we have given a detailed discussion of our findings and reported the causes that we have identified for the ML algorithms being unbalanced.

As we mentioned earlier, looking at the learning algorithm itself, we assume that it cannot be that the algorithm outputs differently when applying some specific metamorphic transformations to the training data. However, as we found out in our evaluation, the main reason for unbalancedness lies not in the learning algorithm, but rather in how they are implemented. For instance, since the learning algorithm needs to be really fast even for a high dimensional input dataset, in ML libraries, often specific optimization algorithms are being used. The use of such optimization techniques often makes the learning algorithms to be sensitive to row or column permutations on the training data. There can be other factors too, like tie-breaking, large floating point number calculations, and more. A later work [PLQT19] has shown that the generation of different models by the learning algorithms because of the above-mentioned reasons is unknown to many non-ML persons and industry practitioners. Thus, our work in this aspect can be helpful, and we believe such unbalancedness would then be considered before using the ML algorithms.

Evaluation of MLcheck. We have evaluated MLcheck in testing 20 different properties on 202 single and multi-label classification and regression models as well as some specific numeric functions. While doing so, we have also compared our tool to the existing state-of-the-art tools. For instance, we have tested the fairness property individual discrimination and compared MLcheck with respect to generating unfair test cases with the state-of-the-art black-box fairness testing tools. For many of the properties we considered, such as monotonicity, concept relationship, security, and the properties for programmed and learned aggregation functions, there are no testing tools available. Hence, we considered adaptive random testing and the property-based testing as the baseline tools to compare with MLcheck. The overall results of the

evaluation have shown promising results in using MLcheck for testing such properties. More specifically, in evaluating fairness properties, we found our tool to be performing better than the existing fairness testing tools, in finding more failed test cases for most of the ML models. In case of monotonicity and security properties, MLcheck performs better than the adaptive random testing (for which we also contributed with a distance measure) and the property-based testing tools. We have found a similar sort of a trend while testing multi-label classifiers with respect to concept relationship and testing the properties of the regression models and the numeric functions.

However, since our property-driven testing approach is generic, it probably will not be comparable to the validation or verification technique designed specifically for an ML model or for a specific property. For example, there exist testing techniques designed for validating deep neural networks with respect to individual discrimination property. However, we did not consider comparing our technique with this approach since our testing tool MLcheck would not be able to outperform such a specific approach. Moreover, the comparison with model- or property-specific approaches would not have been a fair evaluation. In this thesis, with respect to the ML model testing, we did not intend to build a testing approach that is specific to a model or a property, rather, we aimed to build an approach that can be used as a general-purpose tool to test any ML models and with respect to the properties that the tester intends to test. In Chapter 7, we saw that, without changing anything, and just by activating some parameters in MLcheck, we could check 20 different properties for testing 202 different models. Moreover, in most of those cases, our approach has shown superior performance.

8.3 Future Work

For both of our testing approaches, we have a number of improvements and future research directions we can think of. We again categorize our future work in these two parts below.

Future direction of TiLe. We have so far implemented TiLe to test for balanced data usage property. That means with our approach we could only check whether an ML algorithm generates non-equivalent models when three specific transformations are applied to the training data. The transformations we have considered are domain-independent, meaning, we assumed that any ML algorithms should not be sensitive to such transformations. However, depending on the ML algorithm at hand, there can be specific metamorphic transformations for the training data for which the generated models might exhibit some specific relationships. Since there does not exist an oracle with respect to which the implementations of the ML algorithms could be tested, we have to rely on performing such transformations on the training dataset to find out whether we could detect any bug in the implementations.

For example, in case of the random forest algorithm, if we *scale up* the training dataset by adding a constant value to all the instances along with their labels, the generated model should also be scaled up. Now, this is already considered for some algorithms like naive Bayes, k-NN, and SVM with linear kernel. However, the implementations of tree-based or boosted algorithms in several libraries are not considered for testing with respect to such transformations. Hence, to this end, we could think of extending TiLe to incorporate several of such metamorphic transformations to be applied to the training data. In fact, we could even find out new types of metamor-

phic transformations for such algorithms. There are now techniques available (such as [ATA+21]) that could find out metamorphic transformations based on the given program. We could even employ such strategies to discover new transformations to be used for testing ML algorithms.

Equivalency checking. So far we can only check the equivalency between the two models, however, this is not enough if we intend to check some specific relationships between the generated models. Thus, to this end, our equivalence checking technique could be extended to incorporate mechanisms to check specific relationships between two models. For instance, if we allow to apply scaling transformation on the training data, we need to extend our equivalency checking mechanism to check a specific relationship between the generated models. Furthermore, to perform equivalency checking, we mostly perform simple random testing (except for some specific algorithms) and this definitely could be improved by employing more advanced techniques such as search-based technique or we could even think of using our own property-driven testing technique MLcheck for this purpose.

Other types of ML algorithms. Finally, while testing ML algorithms, we only considered the algorithms trained on the tabular dataset. Our approach could further be extended to be used for testing algorithms considering different types of datasets as well, such as image, audio, text, or time series datasets. There have been some works regarding the testing of image-based classification algorithms, however, to the best of our knowledge not many works exist on testing ML algorithms trained on other kinds of datasets. We could check such algorithms by adapting and applying the transformations we had for our balancedness property, and at the same time we could find out other appropriate metamorphic transformations for these kinds of algorithms and datasets and further incorporate those in TiLe.

Future direction of MLcheck. We have proposed a new form of testing approach– property-driven testing and then developed MLcheck based on this approach. To this end, we could again think of numerous extensions of our approach and further future works. We do not describe them all, however, just briefly mention some of them.

Property-specification language. In our approach, we proposed a property specification language that could be used to specify properties. However, currently, this could only be used to specify non-stochastic properties and there are some stochastic properties that are often of importance and need to be checked on the ML models. For example, there exist stochastic properties in the fairness domain such as statistical parity or equal opportunity rate [VR18] which are important to validate in some specific application areas. Thus, an extension of our property specification language allowing to specify such properties could be a useful extension. However, it does not suffice to have specification language which would allow for such properties to specify, we also need a way to check them on the given model, which we describe next.

Verification-based testing. In this testing approach, so far, we could learn a white-box ML model, either a decision tree or a neural network approximating the model under test, and to this end, we have two choices. The choice of these two types of models come from the fact that these can be translated into logical formulas and therefore, we can apply an SMT solving technique to generate test cases on them. The choice of the white-box model could further be extended by considering other types of models such as random-forest, or boosted tree models which are basically ensembles of decision trees, and could be also realised using the SMT formulas. However, since the size of the formula to this end could be quite big (as the number of ensemble trees is typically

of large number), we would then need a way to efficiently solve the formula by using some sort of reduction techniques. To this end, we could probably split up the input values in specific ranges and then keep the ensemble that is required for that range and discard the rest. In this way, we could reduce the size of the ensembles and then apply the SMT solving technique parallelly to different ensembles corresponding to different input ranges and obtain the counter-examples, and finally collect them to be used as test cases on the MUT.

There have been already some works [SKT22, ZTK22] on the extension of our proposed verification-based testing technique to make it more efficient in generating test cases. These works consider the encoding of a decision tree as a white-box model into the SAT formula instead of SMT. This basically opens up a whole lot of new possibilities for our testing approach. Usage of SAT solving process instead of SMT solving could make our approach much faster in generating test cases, which is already shown by Zhao et al. [SKT22]. This process could further enable us to consider ensemble tree models as the white-box model since there exist already some works that give the SAT encoding of such models. Apart from that, we could also use *model counting* techniques (for example ApproxMC4 [SGM20]). The model counting technique would then be used to generate multiple counter-examples for a single SAT query and replace our pruning technique with more advanced techniques. Furthermore, a stochastic SAT solver [GBM21] could also be used which could then allow us to perform the validation of the stochastic properties we mentioned before.

Evaluation. The basic idea of our test generation technique renders the idea from the learning-based testing approach where traditionally an automaton is learned to approximate a given function and then test cases are generated on the automaton for the function under test. There exists numerous literature to this end and one of the most prominent approaches is the AALPy library [MAP+22]. An interesting evaluation could be comparing our MLcheck approach to the AALPy and seeing how our approach performs. To this end, we could think of evaluating our framework on the benchmark cases on which AALPy is typically applied.

Appendix A

Technical Details & Extra Results

In this Appendix, we give the implementation details of the tools we developed corresponding to our approaches. To this end, first, we describe the technical details of the tool TiLe in Section A.1 which was developed to test the balancedness property of the ML algorithms. Apart from giving the implementation details we also describe how to configure this tool. Next, we present the details of the tool MLcheck in Section A.2 which we developed with respect to the testing of ML models and corresponding to the property-driven testing approach (described in Chapter 6). Since to use the tool, the tester needs to configure some parameters, here we give details on how to configure the tool to be used for testing ML models. Next, we provide the link of the aforementioned tools in Section A.3. Finally, in Section A.4 we give the features–corresponding to the datasets–with respect to which we tested monotonicity. We also present the large results table corresponding to the monotonicity testing and aggregation function testing.

A.1 Tool Implementation of TiLe

In this section, we describe the technical details of the tool TiLe. This is implemented as the testing tool for balancedness testing as described in Section 4.2 of Chapter 4. The implementation of the tool follows the workflow diagram depicted in Figure 4.1 [1]. We have different parts in our testing tool which we describe next.

Input configuration. First of all, we start with the input required in our testing tool. Figure A.1 gives an example of a typical `XML` specification file required by TiLe. As part of the *parameters* describing the input learning algorithm, we first require the library name from where the algorithm is taken. This is specified by the `<library>` tag. Along with that, we need the package to be *loaded* in order to get the classification algorithm under test, which can be set using the `<package>` tag. Finally, in our running example, we test a gradient boosting classifier with the hyper-parameter `random_state` set to 1, which corresponds to the `<classifier>` tag. Hence, the element corresponding to this tag describes the classification algorithm along with the hyper-parameter setting for the classifier.

Next, the desired metamorphic transformation to be applied to the training data is specified using `<metaTrans>` tag, which in this example is–permuting the columns or the feature values randomly. We use simple natural language 'permute-column' to specify the column permutation transformation since it is quite intuitive and simple to specify. Similarly, for example, the metamorphic transformations, permuting rows

[1] While checking the ML algorithms of the WEKA library, we extend some parts of the tool in Java. However, this is a minor implementation detail and therefore we omit them.

```
<Configuration>
  <algoInfo>
    <library>scikit-learn</library>
    <package>sklearn.ensemble</package>
    <classifier>GradientBoostingClassifier(random_state=1)</classifier>
  </algoInfo>
  <metaTrans>permute-column</metaTrans>
  <testingParameter>
    <inputRatio>10</inputRatio>
    <trainRatio>50</trainRatio>
  </testingParameter>
  <outputRelation>model-before = model-after</outputRelation>
</Configuration>
```

Figure A.1: An XML specification example of TiLe

randomly and shuffling feature names, are specified using 'permute-row' and 'shuffle-feature' respectively[2].

After specifying the algorithm and the metamorphic transformation to check, we also allow the tester to specify some of the testing parameters corresponding to the equivalence testing part of our framework. Note that, this part of the `XML` specification file is optional and if the tester does not specify any values to these testing parameters, some default values will be automatically set. For instance, the value corresponding to the `<inputRatio>` tag (corresponds to the `INPUT-RATIO` parameter of Algorithm 3) dictates how much percentage of the total possible input instances (derived from $\vec{X}$) will be randomly generated as test inputs. Similarly, the value of the `<trainRatio>` tag (corresponds to the `TRAIN-RATIO` of Algorithm 3) indicates how much percentages of the total training instances would be selected as test inputs randomly.

Finally, the `<outputRelation>` tag allows the tester to specify whether the models generated before (`model-before`) and after (`model-after`) applying the transformation are expected to be equivalent or not equivalent. This tag essentially allows us to check for a more complex relationship between the output models, however, so far we do not have methods for checking relationships apart from model equivalency and can be considered as a possible future extension of our work.

Training data repository. We consider several training datasets as part of our training data repository. To this end, we take the real-world datasets as well as *synthetic* datasets. As the real-world datasets we consider 9 different training sets which are shown in Table A.1. Apart from the datasets, the table also gives the size of the datasets in terms of the number of features and the instances. All of them are frequently used in the ML domain and taken from the well known online data repository of UCI [3]. These datasets are chosen based on the variation in a number of instances and features. For example, we consider Census-Income dataset because it contains a high number of instances (48,842), whereas, the voice-recognition dataset contains a high number of columns (309). We could see in our evaluation (Section 4.3), indeed

[2]Note that we have a list of such keywords for each of the metamorphic transformations we consider. However, we do not describe all of them here, rather a user interested to use our tool can find it on the tool's GitHub page:https://github.com/arnabsharma91/TiLe.

[3]https://archive.ics.uci.edu/

Table A.1: Real-world datasets taken for training data repository

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

such a varying number of rows and columns was helpful to find out whether an ML algorithm is sensitive to a metamorphic transformation.

Metamorphic transformation. For each of the metamorphic transformations, we write a *function* implementing that transformation. Hence, when a metamorphic transformation is specified in the XML configuration file, the corresponding function would then be executed to apply the transformation to the training data.

Training. Essentially, there are two training phases, before and after applying the transformations on the dataset. However, both of them follow a similar approach where the last column of the dataset is considered to be the class values for the training instances and the rest of the columns are feature values. The training process is then performed following the typical approach taken in machine learning. Since, so far we do not consider any Deep learning models, we train all the instances of the dataset on the given learning algorithm once [4].

Equivalence checker. The equivalence checking approach in our framework consists of two parts, computing, and testing equivalence (see Section 4.2.3). For the equivalence computation, we implement approaches based on the types of the models. For instance, in case of the decision tree, we implement a translation mechanism that translates a decision tree model to Z3 [MB08] code. Using this mechanism, we translate two tree models (which are required to be computed for equivalence) into their corresponding logical forms, and then we add the translated equivalency constraint to it. To compute the equivalency of two neural network models, we essentially compare the learned parameters, or more specifically the weights and biases of the two models (with the same architecture). To this end, we compute the Euclidean distances between the two sets of parameters to find out whether the two network models are not equivalent. We follow a similar approach while computing the equivalency of the two SVM models and logistic regression models.

For the testing part, we simply implement the Algorithm 3, and then combine both the computation and testing approach using the implementation of Algorithm 2.

A.2 Configuration of MLcheck

To run MLcheck, the tester can set several parameters, however, most are optional and thus, default values are assumed if the respective parameters are not set. To this

[4]Note that, it is often the case in case of Deep learning to train the dataset on the learner multiple times before generating the final model.

Table A.2: Parameters of MLcheck

Parameter (req.)	Type	Purpose
`model`	$M : \vec{X} \to \vec{Y}$	model under test
`XML_file`	XML	data format
`instance_list`	$\vec{X}^*$	sequence of instance variables
Parameter (opt.)	**Type**	**Purpose**
`train_avail`	boolean	availability of training data of the MUT
`train_path`	string	path to the training data
`train_ratio`	float	percentage of training data to be selected
`init_oracl`	integer	initial training dataset size
`wbm`	$\{dt, nn\}$	white-box model to be used
`nn_library`	$\{torch, scikit\}$	ML library to be used when nn is chosen
`dt_hyp_param`	XML	hyper-parameter values of decision tree
`nn_hyp_param`	XML	hyper-parameter values of neural network
`solver`	$\{cvc, z3, yices\}$	SMT solver to be used
`multi`	boolean	single or multiple CEX
`bound_cex`	boolean	constrain the values of CEX
`max_samples`	integer	size of test suite
`deadline`	float	time limit of running MLcheck

end, there are two types of parameters in MLcheck, *required* and *optional* parameters. Table A.2 summarizes all these parameters along with their types and a short description of their purposes. Below we describe these parameters amongst which the first three are the required parameters that must be set by the tester in order to start the testing process.

MUT. First, the MUT must be provided by the tester in the appropriate form. It could be provided as an executable format (loaded directly by calling a *function*), or in a serialized compressed format, for which we use the `pickle` [5] format. For the latter, the tester must provide the specific file location to load the model from, for example, `model='../Documents/mut.joblib'` where `mut.joblib` is the compressed MUT file.

Input-output schema. Next, we require the input-output format of the MUT given as an `XML` file. Figure A.2 shows an example of such an `XML` file. In this example, the input vector to the model contains 3 features namely `age`, `income`, and `gender` and the model predicts the `loan` as the output. For each of the features, the tester can furthermore specify the *type* and the minimum and maximum values. We only allow numerical value types (integer and rational) for the features.

Depending on the `type` tag of the `output`, the MUT can be considered to be either a classifier or a regressor. In the example schema, the type of the output is Binary, thus, the MUT is considered to be a Binary classifier. Alternatively, if the value corresponding to the `type` is `float` for the output, the MUT is considered to be a regression model. This information is used to decide which

[5] https://docs.python.org/2/library/pickle.html.

```
<Schema>
    <input>
     <feature name="age">
         <type>integer</type>
         <min-value>18</min-value>
         <max-value>70</max-value>
     </feature>
     <feature name="income">
         <type>float</type>
         <min-value>500.00</min-value>
         <max-value>50,000.00</max-value>
     </feature>
     <feature name="gender">
         <type>Boolean</type>
         <min-value>0</min-value>
         <max-value>1</max-value>
       </feature>
    </input>
    <output>
     <feature name="loan">
         <type>Binary</type>
         <min-value>0</min-value>
         <max-value>1</max-value>
     </feature>
    </output>
</Schema>
```

Figure A.2: Example input-output schema in XML

type of white-box model (i.e., classification or regression) has to be learned to approximate the MUT. Moreover, when the MUT is a multi-label classifier, then the number of elements corresponding to the `<output>` tag would be more than `1`, and thus, in this case, a multi-label classifier would be learned based on the number of outputs.

List of instances. Apart from the input-output format and the MUT, we further need to have the instance variables that would be used in the specification of the property. For example, if the tester wants to specify a property like monotonicity, two instance variables (e.g., `x1`, `x2`) are required which could be provided as `instance_list='x1, x2'`.

Optional parameters. Along with the three required parameters described above, we have a number of optional parameters that if not set by the tester, would be set to default values. For instance, if the training data which was used for training the MUT is available, then this dataset along with the randomly generated data could be used for the training of the underlying white-box model which approximates the MUT. The tester could set the option `train_avail` to true and give the location of the training data by using the parameter `train_path` to include the original training dataset of MUT. Furthermore, the size of this training dataset,

Table A.3: Features to test monotonicity

Name	Montonicity Features
Adult	{age, weekly-working-hours, capital-gain, education}
Diabetes	{PlasmaGlucose, BP, weight, age, Pregnancies}
Mammographic	{Shape, Density, severity}
Car-evaluation	{No.ofDoors, No.ofPersons, LugBootSize, YearofProduction}
ESL	{in1,in2,in3,in4}
Housing	{bedrooms, bathrooms, stories }
Automobile	{FuelType, Aspiration, Doors, power, Cylinders, size, wheels, length, width, height}
Auto-MPG	{Cylinders, Horsepower, Weight, Acceleration, Displacement}
ERA	{in1,in2,in3,in4}
CPU	{CycleTime, MinMemory, MaxMemory, CacheMemory, MaxChannel}

i.e., the percentage of instances to be taken from the original training data, and the number of randomly generated instances can be specified by using the parameters `train_ratio` and `init_oracl` respectively.

The parameter for choosing the underlying white-box model (*wbm*) to be learned, by default is set to the decision tree (*dt*), which could also be set to the neural network (*nn*). If the chosen white-box model is set to *nn*, then the library to fetch it from could be either set as `PyTorch` (*torch*) or `scikit-learn` (*scikit*), as default it is set to the latter. The hyper-parameters of the two white-box models can also be set by the tester using `XML` files, an example for decision tree is partially shown in Figure A.3 [6]. The tester in this case could specify all the possible hyper-parameters of the decision tree model available in the `scikit-learn` library. We have `<possible-values>` tag denoting all the possible values for a hyper-parameter as allowed in the library, and `<default-value>` tag denoting the default values set initially. Testers could give their choice corresponding to the `<your-value>` tag.

The parameter `solver` can be used to choose one of the following three solvers: Z3 [MB08] (*z3*), CVC [BCD+11] (*cvc*), and Yieces [Dut14] (*yieces*), by default z3 is used. If we find a violation of the specified property in the form of a counter-example on the learned model and also on the MUT, then we can either stop and return the counter-example or, we can generate a number of such counter-examples. For the latter, the parameter `multi` needs to be set `True`. Moreover, based on the minimum and maximum values given in the input schema file (as shown in Figure A.2) for each of the features, we can furthermore bound the feature values for a counter-example. This is often required since the counter-example generated by the solver might be outside their actual range, thus, using the parameter `bound_cex` the values of the generated test cases can be bounded within the specified range [7].

[6] Note that, for brevity we do not give here all the hyper-parameters that could be set by the tester for the decision tree.

[7] Note, if the minimum and maximum values are not provided for any feature, then the counter-example bounding parameter will have no affect.

Finally, our tool MLcheck can run until a certain timeout, defined using the `deadline` parameter (default is 1000 seconds) or, until generating a specific number of test samples, defined by `max_samples`. In the latter case, MLcheck stops after exploring the given number of instances in order to find a property violation.

A.3 Artifact

In this thesis, we proposed two approaches, (a) an approach for testing machine learning algorithms, and (b) an approach for testing machine learning models. These two approaches are implemented in TiLe and MLcheck respectively. The links to all the code and the necessary datasets used in this thesis are as follows:

- TiLe: https://github.com/arnabsharma91/TiLe
- MLcheck: https://github.com/arnabsharma91/MlCheck

Note that, in the repository of MLcheck, all the necessary datasets, models, and code that are needed to replicate the results mentioned in the paper are given. Furthermore, we have also given a separate link to the repository which would lead to the final version of the tool.

A.4 Monotonicity Features & Extra Results

In Table A.3, we give the list of features for each of the datasets with respect to which we tested monotonicity on the ML models generated on these datasets. The features considered here are based on the existing works [KS09, TCDH11] where they have also considered these features to validate the monotonicity. Note that, the ESL and ERA datasets are based on the grades given to the students based on four specific subjects (in1, in2, in3, and in4). However, in the original datasets, they have not specified the name of these.

In Table A.4 we give the results of the monotonicity testing on 10 different models generated on 10 different datasets, thus, generating 100 different models.

Table A.5 gives the results of the validation of 9 properties on 10 different aggregation functions.

Table A.4: Non-monotonicity detections for each classifier per dataset

Model	**Tool**	Adult	Diab.	Mam.	Car	ESL	Ho.	Auto	MPG	ERA	CPU
k-NN	MLC_DT	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
	MLC_NN	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
	ART	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
	PBT	✓	✓	✓	✓	✗	✓	✗	✓	✗	✓
Log	MLC_DT	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓
	MLC_NN	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓
	ART	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓
	PBT	✓	✓	✓	✗	✗	✓	✗	✓	✗	✓
NB	MLC_DT	✓	✓	✗	✓	✗	✓	✓	✓	✗	✓
	MLC_NN	✓	✓	✗	✓	✗	✓	✓	✓	✗	✓
	ART	✓	✗	✗	✓	✓	✓	✗	✗	✗	✓
	PBT	✓	✓	✗	✓	✓	✓	✗	✗	✗	✓
SVM	MLC_DT	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
	MLC_NN	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
	ART	✓	✗	✓	✓	✓	✓	✓	✓	✗	✓
	PBT	✓	✗	✗	✓	✓	✓	✗	✗	✗	✓
NN	MLC_DT	✓	✗	✓	✓	✓	✓	✓	✗	✓	✓
	MLC_NN	✓	✗	✓	✓	✓	✓	✓	✗	✓	✓
	ART	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓
	PBT	✗	✗	✓	✓	✗	✓	✗	✓	✗	✗
RF	MLC_DT	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
	MLC_NN	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
	ART	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
	PBT	✓	✗	✓	✗	✗	✗	✗	✓	✓	✓
AB	MLC_DT	✓	✗	✓	✓	✓	✓	✓	✓	✗	✓
	MLC_NN	✓	✗	✓	✓	✓	✓	✓	✓	✗	✓
	ART	✓	✗	✓	✓	✓	✗	✓	✓	✗	✓
	PBT	✓	✗	✓	✓	✗	✗	✗	✓	✗	✓
GB	MLC_DT	✓	✗	✓	✓	✓	✓	✓	✓	✗	✓
	MLC_NN	✓	✗	✓	✓	✓	✓	✓	✓	✗	✓
	ART	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓
	PBT	✓	✗	✓	✗	✗	✗	✗	✓	✓	✓
LGB	MLC_DT	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗
	MLC_NN	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗
	ART	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
	PBT	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
XGB	MLC_DT	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	MLC_NN	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	ART	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	PBT	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

Table A.5: Results of detected violations for aggregation functions (✓ = violation detected, ✗ = no violation detected)

	AM MLC/PT	**WAM** MLC/PT	**OWA** MLC/PT	**GM** MLC/PT	**OS**$_k$ MLC/PT	**Min** MLC/PT	**Max** MLC/PT	**Med** MLC/PT	**S$_p$** MLC/PT	**3Π** MLC/PT	**Total** MLC/PT
str. mon.	✗/✗	✓/✗	✓/✗	✓/✓	✓/✓	✓/✓	✓/✗	✓/✓	✓/✗	✓/✓	**9/6**
Lipsch.	✓/✓	✓/✓	✓/✓	✓/✗	✓/✓	✓/✓	✓/✓	✓/✗	✓/✓	✓/✓	**10/8**
symm.	✗/✗	✓/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	**1/0**
idemp.	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✓/✗	✓/✗	**2/0**
conjun.	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✗/✗	✓/✓	✓/✓	✓/✓	✓/✓	9/9
disjun.	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✗/✗	✓/✓	✗/✗	✓/✓	8/8
intern.	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✓/✓	✓/✓	2/2
invar.	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✓/✗	✓/✗	**2/0**
addit.	✗/✗	✗/✗	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	8/8
total	3/3	**5/3**	**5/4**	**5/4**	5/5	**4/3**	4/4	**5/4**	**7/5**	**8/6**	**51/41**

```
<Hyper-parameters>
  <parameter name="criterion">
    <possible-values>gini,entropy,log_loss</possible-values>
    <default-value>gini</default-value>
    <your-value> </your-value>
    </parameter>
  <parameter name="splitter">
    <possible-values>best,random</possible-values>
    <default-value>best</default-value>
    <your-value> </your-value>
  </parameter>
  <parameter name="max_depth">
    <possible-values>int, None</possible-values>
    <default-value>None</default-value>
    <your-value> </your-value>
  </parameter>
  <parameter name="min_samples_split">
    <possible-values>int, float</possible-values>
    <default-value>2</default-value>
    <your-value> </your-value>
  </parameter>
  <parameter name="min_samples_leaf">
    <possible-values>int, float</possible-values>
    <default-value>1</default-value>
    <your-value> </your-value>
  </parameter>
        ...
        ...
</Hyper-parameters>
```

Figure A.3: Hyper-parameter values to be set for decision tree

Index

Bibliography

[AAH] Dara Kerr Abrar Al-Heeti. Uber's fatal self-driving crash reportedly caused by software. https://www.cnet.com/roadshow/news/uber-reportedly-finds-false-positive-self-driving-car-accident/. Accessed: May 7, 2018.

[ABC+13] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, 86(8):1978–2001, 2013. doi:10.1016/j.jss.2013.02.061.

[AEG18] Stéphane Ayache, Rémi Eyraud, and Noé Goudian. Explaining black boxes on sequential data using weighted automata. In Olgierd Unold, Witold Dyrka, and Wojciech Wieczorek, editors, *Proceedings of the 14th International Conference on Grammatical Inference, ICGI 2018, Wrocław, Poland, September 5-7, 2018*, volume 93 of *Proceedings of Machine Learning Research*, pages 81–103. PMLR, 2018. URL: http://proceedings.mlr.press/v93/ayache19a.html.

[Agg18] Charu C. Aggarwal. *Neural Networks and Deep Learning - A Textbook*. Springer, 2018. doi:10.1007/978-3-319-94463-0.

[ALN+19] Aniya Aggarwal, Pranay Lohia, Seema Nagar, Kuntal Dey, and Diptikalyan Saha. Black box fairness testing of machine learning models. In Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 625–635. ACM, 2019. doi:10.1145/3338906.3338937.

[AMM+18] Bernhard K. Aichernig, Wojciech Mostowski, Mohammad Reza Mousavi, Martin Tappler, and Masoumeh Taromirad. Model learning and model-based testing. In Amel Bennaceur, Reiner Hähnle, and Karl Meinke, editors, *Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*, volume 11026 of *Lecture Notes in Computer Science*, pages 74–100. Springer, 2018. doi:10.1007/978-3-319-96562-8_3.

[AMMIL12] Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning From Data*. AMLBook, 2012.

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987. doi:10.1016/0890-5401(87)90052-6.

[ASH+21] Aniya Aggarwal, Samiulla Shaikh, Sandeep Hans, Swastik Haldar, Rema Ananthanarayanan, and Diptikalyan Saha. Testing framework for black-box AI models. In *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021*, pages 81–84. IEEE, 2021. doi:10.1109/ICSE-Companion52605.2021.00041.

[ATA+21] Jon Ayerdi, Valerio Terragni, Aitor Arrieta, Paolo Tonella, Goiuria Sagardui, and Maite Arratibel. Generating metamorphic relations for cyber-physical systems with genetic programming: an industrial case study. In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 1264–1274. ACM, 2021. doi:10.1145/3468264.3473920.

[AW93] Norman P Archer and Shouhong Wang. Application of the back propagation neural network algorithm with monotonicity constraints for two-group classification problems. *Decision Sciences*, 24(1):60–75, 1993.

[BCD+11] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. doi:10.1007/978-3-642-22110-1_14.

[BCR21] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Softw.*, 38(3):79–86, 2021. doi:10.1109/MS.2020.3016773.

[BF22] Raven Beutner and Bernd Finkbeiner. Software verification of hyperproperties beyond k-safety. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, volume 13371 of *Lecture Notes in Computer Science*, pages 341–362. Springer, 2022. doi:10.1007/978-3-031-13185-1_17.

[BHM10] Peter Bastian, Felix Heimann, and Sven Marnach. Generic implementation of finite element methods in the distributed and unified numerics environment (DUNE). *Kybernetika*, 46(2):294–315, 2010. URL: http://www.kybernetika.cz/content/2010/2/294.

[BIL+16] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya V. Nori, and Antonio Criminisi. Measuring neural net robustness with constraints. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2613–2621,

2016. URL: https://proceedings.neurips.cc/paper/2016/hash/980ecd059122ce2e50136bda65c25e07-Abstract.html.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[BLBS09] Lionel C. Briand, Yvan Labiche, Zaheer Bawar, and Nadia Traldi Spido. Using machine learning to refine category-partition test specifications and test suites. *Inf. Softw. Technol.*, 51(11):1551–1564, 2009. doi:10.1016/j.infsof.2009.06.006.

[Bre96] Leo Breiman. Bagging predictors. *Mach. Learn.*, 24(2):123–140, 1996. doi:10.1007/BF00058655.

[BSS+19] Teodora Baluta, Shiqi Shen, Shweta Shinde, Kuldeep S. Meel, and Prateek Saxena. Quantitative verification of neural networks and its security applications. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1249–1264. ACM, 2019. doi:10.1145/3319535.3354245.

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009. doi:10.3233/978-1-58603-929-5-825.

[cat21] catboost. https://github.com/catboost, 2021.

[CEH+22] Maria Christakis, Hasan Ferit Eniser, Jörg Hoffmann, Adish Singla, and Valentin Wüstholz. Specifying and testing k-safety properties for machine-learning models. *CoRR*, abs/2206.06054, 2022. arXiv:2206.06054, doi:10.48550/arXiv.2206.06054.

[CG16] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2939672.2939785, doi:10.1145/2939672.2939785.

[CH00] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM, 2000. doi:10.1145/351240.351266.

[Che] Angela Chen. Ibm's watson gave unsafe recommendations for treating cancer. Accessed: July 26, 2018.

[Che19] Tianqi Chen. xgboost. https://github.com/tqchen/xgboost, 2019.

[CHJ+04] MG Cox, PM Harris, EG Johnson, PD Kenward, and GI Parkin. Testing the numerical correctness of software. *Technical Report CMSC*, 34/04, 2004.

[CHJS16] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Active learning for extended finite state machines. *Formal Aspects Comput.*, 28(2):233–263, 2016. doi:10.1007/s00165-016-0355-5.

[CJ12] Bob F Caviness and Jeremy R Johnson. *Quantifier elimination and cylindrical algebraic decomposition*. Springer Science & Business Media, 2012.

[CKL+18] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Comput. Surv.*, 51(1):4:1–4:27, 2018. doi:10.1145/3143561.

[CKP09] Toon Calders, Faisal Kamiran, and Mykola Pechenizkiy. Building classifiers with independency constraints. In Yücel Saygin, Jeffrey Xu Yu, Hillol Kargupta, Wei Wang, Sanjay Ranka, Philip S. Yu, and Xindong Wu, editors, *ICDM Workshops 2009, IEEE International Conference on Data Mining Workshops, Miami, Florida, USA, 6 December 2009*, pages 13–18. IEEE Computer Society, 2009. doi:10.1109/ICDMW.2009.83.

[CLM04] Tsong Yueh Chen, Hing Leung, and I. K. Mak. Adaptive random testing. In Michael J. Maher, editor, *Advances in Computer Science - ASIAN 2004, Higher-Level Decision Making, 9th Asian Computing Science Conference, Dedicated to Jean-Louis Lassez on the Occasion of His 5th Cycle Birthday, Chiang Mai, Thailand, December 8-10, 2004, Proceedings*, volume 3321 of *Lecture Notes in Computer Science*, pages 320–329. Springer, 2004. doi:10.1007/978-3-540-30502-6_23.

[CMN+19] Ştefan Cobzaş, Radu Miculescu, Adriana Nicolae, et al. *Lipschitz functions*. Springer, 2019.

[Cra17] Susan Craw. *Manhattan Distance*, pages 790–791. Springer US, Boston, MA, 2017. doi:10.1007/978-1-4899-7687-1_511.

[CS10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, 2010. doi:10.3233/JCS-2009-0393.

[CS13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013. doi:10.1145/2408776.2408795.

[Cur63] Haskell Brooks Curry. *Foundations of Mathematical Logic*. New York, NY, USA: Dover Publications, 1963.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, 1995. doi:10.1007/BF00994018.

[CWD+18] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A. Bharath. Generative adversarial networks: An overview. *IEEE Signal Process. Mag.*, 35(1):53–65, 2018. doi:10.1109/MSP.2017.2765202.

[CZH+22] Zhenpeng Chen, Jie M. Zhang, Max Hort, Federica Sarro, and Mark Harman. Fairness testing: A comprehensive survey and analysis of trends. *CoRR*, abs/2207.10223, 2022. arXiv:2207.10223, doi:10.48550/arXiv.2207.10223.

[Das] Jeffrey Dastin. Amazon scraps secret ai recruiting tool that showed bias against women. Accessed: October 11, 2018.

[DAS+18] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M. Rao, R. P. Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In Frank Tip and Eric Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 118–128. ACM, 2018. doi:10.1145/3213846.3213858.

[DBB+09] Charles Dugas, Yoshua Bengio, François Bélisle, Claude Nadeau, and René Garcia. Incorporating functional knowledge in neural networks. *J. Mach. Learn. Res.*, 10:1239–1262, 2009. URL: https://dl.acm.org/citation.cfm?id=1577111.

[DHB20] Matthew F Dixon, Igor Halperin, and Paul Bilokon. *Machine learning in Finance*, volume 1406. Springer, 2020.

[DHP+12] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard S. Zemel. Fairness through awareness. In Shafi Goldwasser, editor, *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 214–226. ACM, 2012. doi:10.1145/2090236.2090255.

[DKH17] Junhua Ding, Xiaojun Kang, and Xin-Hua Hu. Validating a deep learning framework by metamorphic testing. In *2nd IEEE/ACM International Workshop on Metamorphic Testing, MET@ICSE 2017, Buenos Aires, Argentina, May 22, 2017*, pages 28–34. IEEE Computer Society, 2017. doi:10.1109/MET.2017.2.

[DMBT17] Mike Daily, Swarup Medasani, Reinhold Behringer, and Mohan M. Trivedi. Self-driving cars. *Computer*, 50(12):18–23, 2017. doi:10.1109/MC.2017.4451204.

[DN19] Caglar Demir and Axel-Cyrille Ngonga Ngomo. A physical embedding model for knowledge graphs. In Xin Wang, Francesca Alessandra Lisi, Guohui Xiao, and Elena Botoeva, editors, *Semantic Technology - 9th Joint International Conference, JIST 2019, Hangzhou, China, November 25-27, 2019, Proceedings*, volume 12032 of *Lecture Notes in Computer Science*, pages 192–209. Springer, 2019. doi:10.1007/978-3-030-41407-8_13.

[DN21] Caglar Demir and Axel-Cyrille Ngonga Ngomo. Convolutional complex knowledge graph embeddings. In Ruben Verborgh, Katja Hose, Heiko Paulheim, Pierre-Antoine Champin, Maria Maleshkova, Óscar Corcho,

Petar Ristoski, and Mehwish Alam, editors, *The Semantic Web - 18th International Conference, ESWC 2021, Virtual Event, June 6-10, 2021, Proceedings*, volume 12731 of *Lecture Notes in Computer Science*, pages 409–424. Springer, 2021. doi:10.1007/978-3-030-77385-4_24.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960. URL: http://doi.acm.org/10.1145/321033.321034, doi:10.1145/321033.321034.

[Dut14] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[EGSS08] Gil Einziger, Maayan Goldstein, Yaniv Sa'ar, and Itai Segall. Verifying robustness of gradient boosted models. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI, EAAI 2019, Hawaii, USA, 2019*, pages 2446–2453. AAAI Press, 2008. doi:10.1609/aaai.v33i01.33012446.

[Ehl17] Rüdiger Ehlers. Formal verification of piece-wise linear feed-forward neural networks. In Deepak D'Souza and K. Narayan Kumar, editors, *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*, volume 10482 of *Lecture Notes in Computer Science*, pages 269–286. Springer, 2017. doi:10.1007/978-3-319-68167-2_19.

[EW16] Lisa Ehrlinger and Wolfram Wöß. Towards a definition of knowledge graphs. In Michael Martin, Martí Cuquet, and Erwin Folmer, editors, *Joint Proceedings of the Posters and Demos Track of the 12th International Conference on Semantic Systems - SEMANTiCS2016 and the 1st International Workshop on Semantic Change & Evolving Semantics (SuCCESS'16) co-located with the 12th International Conference on Semantic Systems (SEMANTiCS 2016), Leipzig, Germany, September 12-15, 2016*, volume 1695 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016. URL: https://ceur-ws.org/Vol-1695/paper4.pdf.

[FJ18] Matteo Fischetti and Jason Jo. Deep neural networks and mixed integer linear optimization. *Constraints An Int. J.*, 23(3):296–309, 2018. doi:10.1007/s10601-018-9285-6.

[FSG+20] Yang Feng, Qingkai Shi, Xinyu Gao, Jun Wan, Chunrong Fang, and Zhenyu Chen. Deepgini: prioritizing massive tests to enhance the robustness of deep neural networks. In Sarfraz Khurshid and Corina S. Pasareanu, editors, *ISSTA: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, USA, 2020*, pages 177–188. ACM, 2020. doi:10.1145/3395363.3397357.

[FWJ+22] Ming Fan, JiaLi Wei, Wuxia Jin, Zhou Xu, Wenying Wei, and Ting Liu. One step further: evaluating interpreters using metamorphic testing. In Sukyoung Ryu and Yannis Smaragdakis, editors, *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis,*

Virtual Event, South Korea, July 18 - 22, 2022, pages 327–339. ACM, 2022. doi:10.1145/3533767.3534225.

[GBC16] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016. URL: http://www.deeplearningbook.org/.

[GBM17] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. Fairness testing: testing software for discrimination. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 498–510. ACM, 2017. doi:10.1145/3106237.3106277.

[GBM21] Bishwamittra Ghosh, Debabrota Basu, and Kuldeep S. Meel. Justicia: A stochastic SAT approach to formally verify fairness. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 7554–7563. AAAI Press, 2021. URL: https://ojs.aaai.org/index.php/AAAI/article/view/16925.

[GKZ+14] Alex Groce, Todd Kulesza, Chaoqiang Zhang, Shalini Shamasunder, Margaret M. Burnett, Weng-Keen Wong, Simone Stumpf, Shubhomoy Das, Amber Shinsel, Forrest Bice, and Kevin McIntosh. You are the only possible oracle: Effective test selection for end users of interactive machine learning systems. *IEEE Trans. Software Eng.*, 40(3):307–323, 2014. doi:10.1109/TSE.2013.59.

[Gou83] John S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Trans. Software Eng.*, 9(6):686–709, 1983. doi:10.1109/TSE.1983.235433.

[GPKB20] Anshika Gupta, Vinay Pant, Sudhanshu Kumar, and Pravesh Kumar Bansal. Bank loan prediction system using machine learning. In *2020 9th International Conference System Modeling and Advancement in Research Trends (SMART)*, pages 423–426. IEEE, 2020.

[GXL+20] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. Audee: Automated testing for deep learning frameworks. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 486–498. IEEE, 2020. doi:10.1145/3324884.3416571.

[GZW+19] Divya Gopinath, Mengshi Zhang, Kaiyuan Wang, Ismet Burak Kadron, Corina S. Pasareanu, and Sarfraz Khurshid. Symbolic execution for importance analysis and adversarial generation in neural networks. In Katinka Wolter, Ina Schieferdecker, Barbara Gallina, Michel Cukier,

Roberto Natella, Naghmeh Ramezani Ivaki, and Nuno Laranjeiro, editors, *30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019, Berlin, Germany, October 28-31, 2019*, pages 313–322. IEEE, 2019. doi:10.1109/ISSRE.2019.00039.

[Har07] Mark Harman. The current state and future of search based software engineering. In Lionel C. Briand and Alexander L. Wolf, editors, *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, pages 342–357. IEEE Computer Society, 2007. doi:10.1109/FOSE.2007.29.

[HH19] Reiner Hähnle and Marieke Huisman. Deductive software verification: From pen-and-paper proofs to industrial tools. In Bernhard Steffen and Gerhard J. Woeginger, editors, *Computing and Software Science - State of the Art and Perspectives*, volume 10000 of *Lecture Notes in Computer Science*, pages 345–373. Springer, 2019. doi:10.1007/978-3-319-91908-9_18.

[HNS03] Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-specific optimization in automata learning. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2003. doi:10.1007/978-3-540-45069-6_31.

[HSX+21] Rubing Huang, Weifeng Sun, Yinyin Xu, Haibo Chen, Dave Towey, and Xin Xia. A survey on adaptive random testing. *IEEE Trans. Software Eng.*, 47(10):2052–2083, 2021. doi:10.1109/TSE.2019.2942921.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition*. Springer Series in Statistics. Springer, 2009. doi:10.1007/978-0-387-84858-7.

[HYL+22] Pei Huang, Yuting Yang, Minghao Liu, Fuqi Jia, Feifei Ma, and Jian Zhang. ε-weakened robustness of deep neural networks. In Sukyoung Ryu and Yannis Smaragdakis, editors, *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, pages 126–138. ACM, 2022. doi:10.1145/3533767.3534373.

[hyp23] Hypothesis. https://github.com/HypothesisWorks/hypothesis, 2023.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016. doi:10.1109/CVPR.2016.90.

[IIM22] Yacine Izza, Alexey Ignatiev, and João Marques-Silva. On tackling explanation redundancy in decision trees. *J. Artif. Intell. Res.*, 75:261–321, 2022. doi:10.1613/jair.1.13575.

[IISM22] Alexey Ignatiev, Yacine Izza, Peter J. Stuckey, and João Marques-Silva. Using maxsat for efficient explanations of tree ensembles. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*, pages 3776–3785. AAAI Press, 2022. URL: https://ojs.aaai.org/index.php/AAAI/article/view/20292.

[INM19] Alexey Ignatiev, Nina Narodytska, and João Marques-Silva. Abduction-based explanations for machine learning models. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 1511–1519. AAAI Press, 2019. doi:10.1609/aaai.v33i01.33011511.

[Jac05] Henrik Jacobsson. Rule extraction from recurrent neural networks: A taxonomy and review. *Neural Comput.*, 17(6):1223–1263, 2005. doi:10.1162/0899766053630350.

[JFL+22] Pin Ji, Yang Feng, Jia Liu, Zhihong Zhao, and Zhenyu Chen. Asrtest: automated testing for deep-neural-network-driven speech recognition systems. In Sukyoung Ryu and Yannis Smaragdakis, editors, *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, pages 189–201. ACM, 2022. doi:10.1145/3533767.3534391.

[jqu23] jqwik. https://jqwik.net/, 2023.

[JSW22] Taeuk Jang, Pengyi Shi, and Xiaoqian Wang. Group-aware threshold adaptation for fair classification. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*, pages 6988–6995. AAAI Press, 2022. URL: https://ojs.aaai.org/index.php/AAAI/article/view/20657.

[JV00] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In Debra J. Richardson and Mary Jean Harold, editors, *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2000, Portland, OR, USA, August 21-24, 2000*, pages 14–25. ACM, 2000. doi:10.1145/347324.383378.

[KBD+17] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2017. doi:10.1007/978-3-319-63387-9_5.

[KHI+19] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljic, David L. Dill, Mykel J. Kochenderfer, and Clark W. Barrett. The marabou framework for verification and analysis of deep neural networks. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 443–452. Springer, 2019. doi:10.1007/978-3-030-25540-4_26.

[KK17] Dominic Kempf and Timo Koch. System testing in scientific numerical software frameworks using the example of dune. *Archive of Numerical Software*, 5(1):151–168, 2017.

[KM04] Sarfraz Khurshid and Darko Marinov. Testera: Specification-based testing of java programs using SAT. *Autom. Softw. Eng.*, 11(4):403–434, 2004. doi:10.1023/B:AUSE.0000038938.10589.b9.

[KM05] Erich Peter Klement and Radko Mesiar. *Logical, algebraic, analytic and probabilistic aspects of triangular norms*. Elsevier, 2005.

[KNR+21] Igor Khmelnitsky, Daniel Neider, Rajarshi Roy, Xuan Xie, Benoît Barbot, Benedikt Bollig, Alain Finkel, Serge Haddad, Martin Leucker, and Lina Ye. Property-directed verification and robustness certification of recurrent neural networks. In Zhe Hou and Vijay Ganesh, editors, *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18-22, 2021, Proceedings*, volume 12971 of *Lecture Notes in Computer Science*, pages 364–380. Springer, 2021. doi:10.1007/978-3-030-88885-5_24.

[KS09] Wojciech Kotlowski and Roman Slowinski. Rule learning with monotonicity constraints. In Andrea Pohoreckyj Danyluk, Léon Bottou, and Michael L. Littman, editors, *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, volume 382 of *ACM International Conference Proceeding Series*, pages 537–544. ACM, 2009. doi:10.1145/1553374.1553444.

[KW17] Sanjay Krishnan and Eugene Wu. PALM: machine learning explanations for iterative debugging. In Carsten Binnig, Joseph M. Hellerstein, and Aditya G. Parameswaran, editors, *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2017,*

Chicago, IL, USA, May 14, 2017, pages 4:1–4:6. ACM, 2017. doi:10.1145/3077257.3077271.

[LB08] Fabien Lauer and Gérard Bloch. Incorporating prior knowledge in support vector regression. *Machine Learning*, 70(1):89–118, 2008. doi:10.1007/s10994-007-5035-5.

[LBM+18] Konstantinos G Liakos, Patrizia Busato, Dimitrios Moshou, Simon Pearson, and Dionysis Bochtis. Machine learning in agriculture: A review. *Sensors*, 18(8):2674, 2018.

[LDS+22] Bo Liu, Ming Ding, Sina Shaham, Wenny Rahayu, Farhad Farokhi, and Zihuai Lin. When machine learning meets privacy: A survey and outlook. *ACM Comput. Surv.*, 54(2):31:1–31:36, 2022. doi:10.1145/3436755.

[LHP19] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. Coverage guided, property based testing. *Proc. ACM Program. Lang.*, 3(OOPSLA):181:1–181:29, 2019. doi:10.1145/3360607.

[LHZL20] Xingchao Liu, Xing Han, Na Zhang, and Qiang Liu. Certified monotonic neural networks. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/b139aeda1c2914e3b579aafd3ceeb1bd-Abstract.html.

[lig19] Lightgbm. https://github.com/Microsoft/LightGBM, 2019.

[LMA+18] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018. URL: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_03A-5_Liu_paper.pdf.

[LS18] Andreas Löscher and Konstantinos Sagonas. Automating targeted property-based testing. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, pages 70–80. IEEE Computer Society, 2018. doi:10.1109/ICST.2018.00017.

[MAP+22] Edi Muskardin, Bernhard K. Aichernig, Ingo Pill, Andrea Pferscher, and Martin Tappler. Aalpy: an active automata learning library. *Innov. Syst. Softw. Eng.*, 18(3):417–426, 2022. doi:10.1007/s11334-022-00449-3.

[MB08] Leonardo Mendonça De Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008,*

Budapest, Hungary, March 29-April 6, 2008. Proceedings, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

[McK20] Carolyn McKay. Predicting risk in criminal procedure: actuarial tools, algorithms, ai and judicial decision-making. *Current Issues in Criminal Justice*, 32(1):22–39, 2020.

[MFF16] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: A simple and accurate method to fool deep neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 2574–2582. IEEE Computer Society, 2016. doi:10.1109/CVPR.2016.282.

[MH16] Vitalik Melnikov and Eyke Hüllermeier. Learning to aggregate using uninorms. In Paolo Frasconi, Niels Landwehr, Giuseppe Manco, and Jilles Vreeken, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 756–771, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-46227-1_47.

[MH19] Vitalik Melnikov and Eyke Hüllermeier. Learning to aggregate: Tackling the aggregation/disaggregation problem for owa. In Wee Sun Lee and Taiji Suzuki, editors, *Proceedings of The Eleventh Asian Conference on Machine Learning*, volume 101 of *Proceedings of Machine Learning Research*, pages 1110–1125, Nagoya, Japan, 17–19 Nov 2019. PMLR. URL: http://proceedings.mlr.press/v101/melnikov19a.html.

[MK01] L. I. Manolache and Derrick G. Kourie. Software testing using model programs. *Softw. Pract. Exp.*, 31(13):1211–1236, 2001. doi:10.1002/spe.409.

[MKA07] Chris Murphy, Gail E. Kaiser, and Marta Arias. An approach to software testing of machine learning applications. In *Proceedings of the Nineteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2007), Boston, Massachusetts, USA, July 9-11, 2007*, page 167. Knowledge Systems Institute Graduate School, 2007.

[MKHW08] Christian Murphy, Gail E. Kaiser, Lifeng Hu, and Leon Wu. Properties of machine learning applications for use in metamorphic testing. In *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE'2008), San Francisco, CA, USA, July 1-3, 2008*, pages 867–872. Knowledge Systems Institute Graduate School, 2008.

[MLL+18] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. MODE: automated neural network model debugging via state differential analysis and input selection. In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 175–186. ACM, 2018. doi:10.1145/3236024.3236082.

[MN10] Karl Meinke and Fei Niu. A learning-based approach to unit testing of numerical software. In Alexandre Petrenko, Adenilso da Silva Simão, and José Carlos Maldonado, editors, *Testing Software and Systems - 22nd IFIP WG 6.1 International Conference, ICTSS 2010, Natal, Brazil, November 8-10, 2010. Proceedings*, volume 6435 of *Lecture Notes in Computer Science*, pages 221–235. Springer, 2010. doi:10.1007/978-3-642-16573-3_16.

[MSB11] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011.

[MSK09] Christian Murphy, Kuang Shen, and Gail E. Kaiser. Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles. In *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009*, pages 436–445. IEEE Computer Society, 2009. doi:10.1109/ICST.2009.19.

[MVY20] Franz Mayr, Ramiro Visca, and Sergio Yovine. On-the-fly black-box probably approximately correct checking of recurrent neural networks. In Andreas Holzinger, Peter Kieseberg, A Min Tjoa, and Edgar R. Weippl, editors, *Machine Learning and Knowledge Extraction - 4th IFIP TC 5, TC 12, WG 8.4, WG 8.9, WG 12.9 International Cross-Domain Conference, CD-MAKE 2020, Dublin, Ireland, August 25-28, 2020, Proceedings*, volume 12279 of *Lecture Notes in Computer Science*, pages 343–363. Springer, 2020. doi:10.1007/978-3-030-57321-8_19.

[MWL20] Pingchuan Ma, Shuai Wang, and Jin Liu. Metamorphic testing and certified mitigation of fairness violations in NLP models. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 458–465. ijcai.org, 2020. doi:10.24963/ijcai.2020/64.

[MWYY20] Yifang Ma, Zhenyu Wang, Hong Yang, and Lin Yang. Artificial intelligence applications in the development of autonomous vehicles: a survey. *IEEE CAA J. Autom. Sinica*, 7(2):315–329, 2020. doi:10.1109/jas.2020.1003021.

[NB16] Shin Nakajima and Hai Ngoc Bui. Dataset coverage for testing machine learning computer programs. In Alex Potanin, Gail C. Murphy, Steve Reeves, and Jens Dietrich, editors, *23rd Asia-Pacific Software Engineering Conference, APSEC 2016, Hamilton, New Zealand, December 6-9, 2016*, pages 297–304. IEEE Computer Society, 2016. doi:10.1109/APSEC.2016.049.

[NH10] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 807–814. Omnipress, 2010. URL: https://icml.cc/Conferences/2010/papers/432.pdf.

[Nie15] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA, 2015.

[NPS+20] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sfuzz: an efficient adaptive fuzzer for solidity smart contracts. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 778–788. ACM, 2020. doi:10.1145/3377811.3380334.

[NZK+20] Manan Binth Taj Noor, Nusrat Zerin Zenia, M. Shamim Kaiser, Shamim Al Mamun, and Mufti Mahmud. Application of deep learning in detecting neurological disorders from magnetic resonance images: a survey on the detection of alzheimer's disease, parkinson's disease and schizophrenia. *Brain Informatics*, 7(1):11, 2020. doi:10.1186/s40708-020-00112-2.

[OBK22] Matan Ostrovsky, Clark W. Barrett, and Guy Katz. An abstraction-refinement approach to verifying convolutional neural networks. In Ahmed Bouajjani, Lukás Holík, and Zhilin Wu, editors, *Automated Technology for Verification and Analysis - 20th International Symposium, ATVA 2022, Virtual Event, October 25-28, 2022, Proceedings*, volume 13505 of *Lecture Notes in Computer Science*, pages 391–396. Springer, 2022. doi:10.1007/978-3-031-19992-9_25.

[OG96] Christian W. Omlin and C. Lee Giles. Extraction of rules from discrete-time recurrent neural networks. *Neural Networks*, 9(1):41–52, 1996. doi:10.1016/0893-6080(95)00086-0.

[OG10] Markus Ojala and Gemma C. Garriga. Permutation tests for studying classifier performance. *J. Mach. Learn. Res.*, 11:1833–1863, 2010. URL: https://dl.acm.org/doi/10.5555/1756006.1859913, doi:10.5555/1756006.1859913.

[OWSH20] Takamasa Okudono, Masaki Waga, Taro Sekiyama, and Ichiro Hasuo. Weighted automata extraction from recurrent neural networks via regression on state spaces. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 5306–5314. AAAI Press, 2020. URL: https://ojs.aaai.org/index.php/AAAI/article/view/5977.

[PAT+22] Anjana Perera, Aldeida Aleti, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Burak Turhan, Lisa Kuhn, and Katie Walker. Search-based fairness testing for regression-based machine learning systems. *Empir. Softw. Eng.*, 27(3):79, 2022. doi:10.1007/s10664-022-10116-7.

[PCYJ17] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China,*

October 28-31, 2017, pages 1–18. ACM, 2017. doi:10.1145/3132747.3132785.

[PGV+18] Liudmila Ostroumova Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. Catboost: unbiased boosting with categorical features. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 6639–6649, 2018. URL: https://proceedings.neurips.cc/paper/2018/hash/14491b756b3a51daac41c24863285549-Abstract.html.

[PLQT19] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 1027–1038. IEEE / ACM, 2019. doi:10.1109/ICSE.2019.00107.

[PLS20] Long H. Pham, Jiaying Li, and Jun Sun. SOCRATES: towards a unified platform for neural network verification. *CoRR*, abs/2007.11206, 2020. URL: https://arxiv.org/abs/2007.11206, arXiv:2007.11206.

[PQW+20] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. Problems and opportunities in training deep learning software systems: An analysis of variance. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 771–783. IEEE, 2020. doi:10.1145/3324884.3416545.

[PTF+21] Giovanni Pellegrini, Alessandro Tibo, Paolo Frasconi, Andrea Passerini, and Manfred Jaeger. Learning aggregation functions. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI, Virtual Event / Montreal, Canada, 19-27 August 2021*, pages 2892–2898. ijcai.org, 2021. doi:10.24963/ijcai.2021/398.

[PVG+11] 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*, 12:2825–2830, 2011.

[PVY99] Doron A. Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. In Jianping Wu, Samuel T. Chanson, and Qiang Gao, editors, *Formal Methods for Protocol Engineering and Distributed Systems, FORTE XII / PSTV XIX'99, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and*

Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX), October 5-8, 1999, Beijing, China, volume 156 of *IFIP Conference Proceedings*, pages 225–240. Kluwer, 1999.

[PW15] Petros Papadopoulos and Neil Walkinshaw. Black-box test generation from inferred models. In Rachel Harrison, Ayse Basar Bener, and Burak Turhan, editors, *4th IEEE/ACM International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, RAISE 2015, Florence, Italy, May 17, 2015*, pages 19–24. IEEE Computer Society, 2015. doi:10.1109/RAISE.2015.11.

[Qui97] J. Ross Quinlan. Decision trees and instance-based classifiers. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 521–535. CRC Press, 1997.

[Rah18] N. A. Rahman. Symmetric Function and Allied Tables. *Royal Statistical Society. Journal. Series A: General*, 130(2):256–257, 12 2018. arXiv:https://academic.oup.com/jrsssa/article-pdf/130/2/256/49745013/jrsssa_130_2_256.pdf, doi:10.2307/2343415.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[Ros02] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Higher Education, 5th edition, 2002.

[ROT89] Debra J. Richardson, T. Owen O'Malley, and C. Tittle. Approaches to specification-based testing. In Richard A. Kemmerer, editor, *Proceedings of the ACM SIGSOFT '89 Third Symposium on Testing, Analysis, and Verification, TAV 1989, Key West, Florida, USA, December 13-15, 1989*, pages 86–96. ACM, 1989. doi:10.1145/75308.75319.

[RS10] Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.*, 79(6):397–434, 2010. doi:10.1016/j.jlap.2010.03.012.

[RSG16] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should I trust you?": Explaining the predictions of any classifier. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 1135–1144. ACM, 2016. doi:10.1145/2939672.2939778.

[RT16] Julia Rubin and Thomas Thüm, editors. *Proceedings 7th International Workshop on Formal Methods and Analysis in Software Product Line Engineering, FMSPLE@ETAPS 2016, Eindhoven, The Netherlands, April 3, 2016*, volume 206 of *EPTCS*, 2016. doi:10.4204/EPTCS.206.

[SC93] Phil Stocks and David A. Carrington. Test templates: A specification-based testing framework. In Victor R. Basili, Richard A. DeMillo, and

Takuya Katayama, editors, *Proceedings of the 15th International Conference on Software Engineering, Baltimore, Maryland, USA, May 17-21, 1993*, pages 405–414. IEEE Computer Society / ACM Press, 1993. URL: http://portal.acm.org/citation.cfm?id=257572.257664.

[SCDG] Avi Feller Sam Corbett-Davies, Emma Pierson and Sharad Goel. A computer program used for bail and sentencing decisions was labeled biased against blacks. it's actually not that clear. https://www.washingtonpost.com/news/monkey-cage/wp/2016/10/17/can-an-algorithm-be-racist-our-analysis\-is-more-cautious-than-propublicas/. Accessed: October 17, 2016.

[Sch90] Robert E. Schapire. The strength of weak learnability. *Mach. Learn.*, 5:197–227, 1990. doi:10.1007/BF00116037.

[Sch12] Ina Schieferdecker. Model-based testing. *IEEE Softw.*, 29(1):14–18, 2012. doi:10.1109/MS.2012.13.

[SG09] Muzammil Shahbaz and Roland Groz. Inferring mealy machines. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2009. doi:10.1007/978-3-642-05089-3_14.

[SGM20] Mate Soos, Stephan Gocht, and Kuldeep S. Meel. Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 463–484. Springer, 2020. doi:10.1007/978-3-030-53288-8_22.

[SGSG19] Jenni AM Sidey-Gibbons and Chris J Sidey-Gibbons. Machine learning in medicine: a practical introduction. *BMC medical research methodology*, 19:1–18, 2019.

[SKT22] Shinya Sano, Takashi Kitamura, and Shingo Takada. An efficient discrimination discovery method for fairness testing. In Rong Peng, Carlos Eduardo Pantoja, and Pankaj Kamthan, editors, *The 34th International Conference on Software Engineering and Knowledge Engineering, SEKE 2022, KSIR Virtual Conference Center, USA, July 1 - July 10, 2022*, pages 200–205. KSI Research Inc., 2022. doi:10.18293/SEKE2022-064.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In Michel Wermelinger and Harald C. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272. ACM, 2005. doi:10.1145/1081706.1081750.

[Spi89] J. Michael Spivey. An introduction to Z and formal specifications. *Softw. Eng. J.*, 4(1):40–50, 1989. doi:10.1049/sej.1989.0006.

[SPV16] Eero Siivola, Juho Piironen, and Aki Vehtari. Automatic monotonicity detection for gaussian processes. *arXiv preprint arXiv:1610.05440*, 2016.

[SS21] Richard Schumi and Jun Sun. Spectest: Specification-based compiler testing. In Esther Guerra and Mariëlle Stoelinga, editors, *Fundamental Approaches to Software Engineering - 24th International Conference, FASE 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12649 of *Lecture Notes in Computer Science*, pages 269–291. Springer, 2021. doi:10.1007/978-3-030-71500-7_14.

[SUC22] Ezekiel O. Soremekun, Sakshi Udeshi, and Sudipta Chattopadhyay. Astraea: Grammar-based fairness testing. *IEEE Trans. Software Eng.*, 48(12):5188–5211, 2022. doi:10.1109/TSE.2022.3141758.

[SW19] Arnab Sharma and Heike Wehrheim. Testing machine learning algorithms for balanced data usage. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pages 125–135. IEEE, 2019. doi:10.1109/ICST.2019.00022.

[SW20a] Arnab Sharma and Heike Wehrheim. Automatic fairness testing of machine learning models. In Valentina Casola, Alessandra De Benedictis, and Massimiliano Rak, editors, *Testing Software and Systems - 32nd IFIP WG 6.1 International Conference, ICTSS 2020, Naples, Italy, December 9-11, 2020, Proceedings*, volume 12543 of *Lecture Notes in Computer Science*, pages 255–271. Springer, 2020. doi:10.1007/978-3-030-64881-7_16.

[SW20b] Arnab Sharma and Heike Wehrheim. Higher income, larger loan? monotonicity testing of machine learning models. In Sarfraz Khurshid and Corina S. Pasareanu, editors, *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, pages 200–210. ACM, 2020. doi:10.1145/3395363.3397352.

[SWR+18] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. Concolic testing for deep neural networks. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 109–119. ACM, 2018. doi:10.1145/3238147.3238172.

[TAB+19] Martin Tappler, Bernhard K. Aichernig, Giovanni Bacci, Maria Eichlseder, and Kim G. Larsen. L^*-based learning of markov decision processes. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*, volume 11800 of *Lecture Notes in Computer Science*, pages 651–669. Springer, 2019. doi:10.1007/978-3-030-30942-8_38.

[TCDH11] Ali Fallah Tehrani, Weiwei Cheng, Krzysztof Dembczynski, and Eyke Hüllermeier. Learning monotone nonlinear models using the choquet integral. In Dimitrios Gunopulos, Thomas Hofmann, Donato Malerba, and Michalis Vazirgiannis, editors, *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2011, Athens, Greece, September 5-9, 2011, Proceedings, Part III*, volume 6913 of *Lecture Notes in Computer Science*, pages 414–429. Springer, 2011. doi:10.1007/978-3-642-23808-6_27.

[TN20] John Törnblom and Simin Nadjm-Tehrani. Formal verification of input-output mappings of tree ensembles. *Sci. Comput. Program.*, 194:102450, 2020. doi:10.1016/j.scico.2020.102450.

[Tse83] Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.

[TSHL17] Gabriele Tolomei, Fabrizio Silvestri, Andrew Haines, and Mounia Lalmas. Interpretable predictions of tree-based ensembles via actionable feature tweaking. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*, pages 465–474. ACM, 2017. doi:10.1145/3097983.3098039.

[TSHW20] Sarah Tan, Matvey Soloviev, Giles Hooker, and Martin T. Wells. Tree space prototypes: Another look at making tree ensembles interpretable. In Jeannette M. Wing and David Madigan, editors, *FODS '20: ACM-IMS Foundations of Data Science Conference, Virtual Event, USA, October 19-20, 2020*, pages 23–34. ACM, 2020. doi:10.1145/3412815.3416893.

[TSL04] Li Tan, Oleg Sokolsky, and Insup Lee. Specification-based testing with linear temporal logic. In Du Zhang, Éric Grégoire, and Doug DeGroot, editors, *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, IRI - 2004, November 8-10, 2004, Las Vegas Hilton, Las Vegas, NV, USA*, pages 493–498. IEEE Systems, Man, and Cybernetics Society, 2004. doi:10.1109/IRI.2004.1431509.

[TZO+20] Yuchi Tian, Ziyuan Zhong, Vicente Ordonez, Gail E. Kaiser, and Baishakhi Ray. Testing DNN image classifiers for confusion & bias errors. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 1122–1134. ACM, 2020. doi:10.1145/3377811.3380400.

[UAC18] Sakshi Udeshi, Pryanshu Arora, and Sudipta Chattopadhyay. Automated directed fairness testing. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 98–108. ACM, 2018. doi:10.1145/3238147.3238165.

[UM18] Caterina Urban and Peter Müller. An abstract interpretation framework for input data usage. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 683–710. Springer, 2018. doi:10.1007/978-3-319-89884-1_24.

[VC21] Andreas Venzke and Spyros Chatzivasileiadis. Verification of neural network behaviour: Formal guarantees for power system applications. *IEEE Trans. Smart Grid*, 12(1):383–397, 2021. doi:10.1109/TSG.2020.3009401.

[VDO+19] Michael Veale, S Delacroix, S Olhede, C Blacklaws, and S Adams Bhatti. Algorithms in the criminal justice system. 2019.

[VLL94] Vladimir Vapnik, Esther Levin, and Yann LeCun. Measuring the vc-dimension of a learning machine. *Neural Comput.*, 6(5):851–876, 1994. doi:10.1162/neco.1994.6.5.851.

[VR18] Sahil Verma and Julia Rubin. Fairness definitions explained. In Yuriy Brun, Brittany Johnson, and Alexandra Meliou, editors, *Proceedings of the International Workshop on Software Fairness, FairWare@ICSE 2018, Gothenburg, Sweden, May 29, 2018*, pages 1–7. ACM, 2018. doi:10.1145/3194770.3194776.

[Wal18] Neil Walkinshaw. Testing functional black-box programs without a specification. In Amel Bennaceur, Reiner Hähnle, and Karl Meinke, editors, *Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*, volume 11026 of *Lecture Notes in Computer Science*, pages 101–120. Springer, 2018. doi:10.1007/978-3-319-96562-8_4.

[Wey82] Elaine J. Weyuker. On testing non-testable programs. *Comput. J.*, 25(4):465–470, 1982. doi:10.1093/comjnl/25.4.465.

[WFH11] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data mining: practical machine learning tools and techniques, 3rd Edition*. Morgan Kaufmann, Elsevier, 2011. URL: https://www.worldcat.org/oclc/262433473.

[WGS19] Roman Werpachowski, András György, and Csaba Szepesvári. Detecting overfitting via adversarial examples. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 7856–7866, 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/28f7241796510e838db4a1384ae1279d-Abstract.html.

[WGY18] Gail Weiss, Yoav Goldberg, and Eran Yahav. Extracting automata from recurrent neural networks using queries and counterexamples. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 5244–5253. PMLR, 2018. URL: `http://proceedings.mlr.press/v80/weiss18a.html`.

[WKQ+08] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus F. M. Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael S. Steinbach, David J. Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowl. Inf. Syst.*, 14(1):1–37, 2008. `doi:10.1007/s10115-007-0114-2`.

[WKRL17] Christian Wolschke, Thomas Kuhn, H. Dieter Rombach, and Peter Liggesmeyer. Observation based creation of minimal test suites for autonomous vehicles. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE, France, 2017*, pages 294–301. IEEE, 2017. `doi:10.1109/ISSREW.2017.46`.

[WLQ+22] Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan. EAGLE: creating equivalent graphs to test deep learning libraries. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 798–810. ACM, 2022. `doi:10.1145/3510003.3510165`.

[WYC+20] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. Deep learning library testing via effective model generation. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 788–799. ACM, 2020. `doi:10.1145/3368089.3409761`.

[XKN22] Xuan Xie, Kristian Kersting, and Daniel Neider. Neuro-symbolic verification of deep neural networks. In Luc De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 3622–3628. ijcai.org, 2022. `doi:10.24963/ijcai.2022/503`.

[XLLL23] Yisong Xiao, Aishan Liu, Tianlin Li, and Xianglong Liu. Latent imitator: Generating natural individual discriminatory instances for black-box fairness testing. *CoRR*, abs/2305.11602, 2023. `arXiv:2305.11602`, `doi:10.48550/arXiv.2305.11602`.

[XLY+22] Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. Metamorphic testing of deep learning compilers. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(1):15:1–15:28, 2022. `doi:10.1145/3508035`.

[XW20] Wentao Xie and Peng Wu. Fairness testing of machine learning models using deep reinforcement learning. In *2020 IEEE 19th International Con-*

ference on Trust, Security and Privacy in Computing and Communications (TrustCom), pages 121–128, 2020. doi:10.1109/TrustCom50675.2020.00029.

[YBK18] Kun-Hsing Yu, Andrew L Beam, and Isaac S Kohane. Artificial intelligence in healthcare. *Nature biomedical engineering*, 2(10):719–731, 2018.

[YDC+17] Seungil You, David Ding, Kevin Robert Canini, Jan Pfeifer, and Maya R. Gupta. Deep lattice networks and partial monotonic functions. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 2981–2989, 2017. URL: https://proceedings.neurips.cc/paper/2017/hash/464d828b85b0bed98e80ade0a5c43b0f-Abstract.html.

[YM21] Jiong Yang and Kuldeep S. Meel. Engineering an efficient PB-XOR solver. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 58:1–58:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CP.2021.58.

[ZCD+22] Haibin Zheng, Zhiqing Chen, Tianyu Du, Xuhong Zhang, Yao Cheng, Shouling Ji, Jingyi Wang, Yue Yu, and Jinyin Chen. Neuronfair: Interpretable white-box fairness testing through biased neuron identification. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1519–1531. ACM, 2022. doi:10.1145/3510003.3510123.

[ZDMR17] Hong Zhu, Junhua Ding, Patrícia D. L. Machado, and Marc Roper. AST 2017 workshop summary. In *12th IEEE/ACM International Workshop on Automation of Software Testing, AST@ICSE 2017, Buenos Aires, Argentina, May 20-21, 2017*, page 1. IEEE Computer Society, 2017. doi:10.1109/AST.2017.19.

[ZHML22] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Trans. Software Eng.*, 48(2):1–36, 2022. doi:10.1109/TSE.2019.2962027.

[ZKR+17] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabás Póczos, Ruslan Salakhutdinov, and Alexander J. Smola. Deep sets. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 3391–3401, 2017. URL: https://proceedings.neurips.cc/paper/2017/hash/f22e4747da1aa27e363d86d40ff442fe-Abstract.html.

[ZS18] Zhi Quan Zhou and Liqun Sun. Metamorphic testing for machine translations: MT4MT. In *25th Australasian Software Engineering Conference, ASWEC 2018, Adelaide, Australia, November 26-30, 2018*, pages 96–100. IEEE Computer Society, 2018. doi:10.1109/ASWEC.2018.00021.

[ZTK22] Zhenjiang Zhao, Takahisa Toda, and Takashi Kitamura. Efficient fairness testing through hash-based sampling. In Mike Papadakis and Silvia Regina Vergilio, editors, *Search-Based Software Engineering - 14th International Symposium, SSBSE 2022, Singapore, November 17-18, 2022, Proceedings*, volume 13711 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2022. doi:10.1007/978-3-031-21251-2_3.

[ZVGG17] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez-Rodriguez, and Krishna P. Gummadi. Fairness constraints: Mechanisms for fair classification. In Aarti Singh and Xiaojin (Jerry) Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA*, volume 54 of *Proceedings of Machine Learning Research*, pages 962–970. PMLR, 2017. URL: http://proceedings.mlr.press/v54/zafar17a.html.

[ZWG+21] Zhiyi Zhang, Pu Wang, Hongjing Guo, Ziyuan Wang, Yuqian Zhou, and Zhiqiu Huang. Deepbackground: Metamorphic testing for deep-learning-driven image recognition systems accompanied by background-relevance. *Inf. Softw. Technol.*, 140:106701, 2021. doi:10.1016/j.infsof.2021.106701.

[ZWS+20] Peixin Zhang, Jingyi Wang, Jun Sun, Guoliang Dong, Xinyu Wang, Xingen Wang, Jin Song Dong, and Ting Dai. White-box fairness testing through adversarial sampling. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 949–960. ACM, 2020. doi:10.1145/3377811.3380331.

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg befolgt habe.

Ort: Oldenburg Datum: 12.12.2023

Unterschrift: Arnab Sharma