Property-based Testing for Machine Learning Models

Vinicius H. S. Durelli
Universidade Federal de São João del-Rei
Minas Gerais, Brazil
durelli@ufsj.edu.br

Ricardo Monteiro
Universidade Federal de São João del-Rei
Minas Gerais, Brazil
ricardosm@ufsj.edu.br

Rafael S. Durelli
Universidade Federal de Lavras
Minas Gerais, Brazil
rafael.durelli@ufla.br

Andre T. Endo
Universidade Federal de São Carlos
São Paulo, Brazil
andreendo@ufscar.br

Fabiano C. Ferrari
Universidade Federal de São Carlos
São Paulo, Brazil
fcferrari@ufscar.br

Simone R. S. Souza
Universidade de São Paulo
São Paulo, Brazil
srocio@icmc.usp.br

ABSTRACT

There has been a growing interest in machine learning due to its potential to address a myriad of problems that would otherwise be difficult to solve. Consequently, the adoption of machine learning based programs has become mainstream. Owing to this widespread adoption, it is imperative to develop automated approaches to assess the quality of machine learning-based solutions. Although significant research has been devoted to creating automated test input generation methods for machine learning programs, some promising approaches to test data generation have received limited attention. This paper introduces a property-driven approach to test data generation that leverages the training of an interpretable model, specifically a decision tree, to predict the behavior of the model under test. The tree-like structure of the resulting interpretable model provides valuable insights into the model's behavior under test. These insights are then transformed into executable properties, enabling the generation of test data. A primary advantage of property-based testing is its capacity to generate a vast number of inputs from a single property, thereby offering a more rigorous evaluation of machine learning models. The results of our experiment suggest that our property-driven approach has the potential to generate test data that more thoroughly examine models compared to more widely used methods for evaluating the performance and generalizability of machine learning models.

KEYWORDS

Software testing; property-based testing; machine learning

1 INTRODUCTION

In recent years, machine learning algorithms have garnered significant attention as a promising solution for addressing various Software Engineering challenges [18]. With this increased adoption, it is crucial to assess the quality of machine learning-based solutions, just as with any other part of a software system, preferably in an automated fashion. Although researchers have devoted considerable efforts to automated test input generation for machine learning programs [21], to the best of our knowledge, exploring how property-based testing can be used to enable the generation of test data remains a largely under-explored problem. To address this gap, we set out to investigate how property-based testing can be used to automate test data generation for machine learning models.

Property-based testing is a methodology in which testers write executable specifications of software elements, and an automated harness checks these specifications (i.e., properties) against numerous automatically generated inputs [7]. Since its inception, which can be traced back to the QuickCheck library [3], property-based testing has evolved significantly and has been increasingly adopted in industrial practice by companies such as Amazon [2]. Nevertheless, relatively little is known about the strengths and weaknesses of property-based testing and the areas where further technical advances are needed [4, 7]. Most studies have focused on using other fault-finding tools in various settings. We argue that property-based testing offers unique tools and merits thorough investigation, particularly for addressing the complexities of testing machine learning models.

We set out to develop a property-driven approach for testing machine learning models. When faced with the task of testing a machine-learning model, the first step in our property-driven approach is to build a decision tree using the available training data. This decision tree model serves as the basis for targeted data generation: our approach leverages the internal structure of decision tree models to guide the selection of test inputs. More specifically, test generation involves encoding details related to the internal structure of the decision tree and its criteria [15] as logical formulae. These formulae dictate how the tree should be traversed and are subsequently turned into properties. The test case generation process relies on a systematic, property-dependent construction of test suites, eliminating the need for tester-supplied information. Our approach builds on the idea that properties are created by understanding the internal structure of a white-box model (i.e., a decision tree model). By knowing this internal structure, we encode the verification of the model in terms of rules (i.e., criteria) on how the tree should be traversed. Therefore, contrary to property-based testing, where testers must devise properties based on rules that should always hold true for the program under test [8], our approach allows for the generation of a property-driven test suite without the tester needing to examine the model and devise properties.

The remainder of this paper is organized as follows. Section 2 presents related work. Section 3 provides background on property-based testing, Section 4 outlines our property-driven approach to test data generation for machine learning models. Section 5 describes the experimental design we used to investigate our research question and Section 6 presents the statistical analysis of the experiment results. Section 7 discusses threats to the validity and Section 8 presents concluding remarks.

2 RELATED WORK

A closely related study presented by Sharma et al. [16] is comparable to our research and shares several key similarities with the methodology described in this paper. Specifically, Sharma et al. propose a property-driven approach to testing machine learning models, termed MLCheck. This approach encompasses *(i)* a domain-specific language for specifying non-stochastic properties, and *(ii)* a systematic test data generation technique. The specification language employs syntax elements akin to those utilized by property-based testing frameworks. Unlike our approach, the approach proposed by Sharma et al. requires the tester to specify the properties of interest, upon which targeted generation of test data is performed. Similar to our approach, MLCheck treats the model under test as a black-box. Additionally, the test data generation step also mandates a verifiable white-box model. To address this, [16] train a white-box model (either a decision tree or a neural network) and compute the test data using this model. However, in contrast to our approach, the white-box model in MLCheck is used to verify the properties, with the resulting counterexamples selected as test inputs for the black-box model under test, rather than serving as the primary source of information for property creation as is the case in our approach.

Another related study conducted by Aggarwal et al. [1] reports on a symbolic approach to test case generation. Similarly to our method, their approach involves generating a decision tree from the model under test. The resulting decision tree is then subjected to dynamic symbolic execution to generate test cases.

3 PROPERTY-BASED TESTING

Property-based testing is a random test case generation technique, where the behavior of the system under test is described by a high-level specification of valid inputs along with the properties expected to hold when the system is subjected to these inputs [10]. In property-based testing, these high-level specifications comprise general rules or assertions about the system's behavior that should hold true for a wide range of inputs. Simply put, these specifications, termed *properties*, can be seen as high-level recipes for data generation. In effect, properties are similar to partial specifications in the sense that they are more compact and easier to write and understand than full system specifications.

Property-based testing relies heavily on tool assistance [8], as the rules about the expected behavior of the system have to be turned into executable code so that they can be run by a specific framework. To automate the process of transforming properties into inputs of increasing complexity, property-based testing frameworks provide built-in data generators for common data types (e.g., integers, strings, lists). Thus, one of the main benefits of property-based testing is the ability to generate many input-output relations with a single specification and generator.

Traditional tests are typically example-based: testers have to come up with a list of various inputs for the program under test and specify the corresponding expected outputs. The effectiveness of these tests depends on including examples (i.e., sets of inputs and expected outputs) that cover all possible states of the program under test. In contrast, when applying property-based testing, testers only need to specify the generic structure of valid inputs and the properties expected to hold for all valid inputs. With this information, property-based testing tools can automatically generate complex random inputs and an automated test harness checks these inputs against the system under test while monitoring its execution.

According to Hebert [8], with properties and a framework that automates the test data generation and execution, it is possible to explore the problem space in a more in-depth fashion. Thus, property-based testing is much more effective at exploring a broader range of behaviors and identifying potential problems in the system under test. Properties act as a checkable partial specification of the system's behavior [10] that is more general than any set of unit tests. Since properties are much more concise than a series of unit tests, property-based testing can subject the system under test to a wider variety of inputs than a tester would typically be willing or able to write. Consequently, compared to testing software systems with manually-written test cases (i.e., in an example-based fashion), property-based testing is both a faster and less monotonous process [10]. Additionally, by introducing randomness to generate inputs property-based test data generation has the potential to trigger edge cases.

The origins of property-based testing can be traced back to the functional programming community [3, 4]; however, it has recently begun to gain traction within a broader audience. Currently, property-based testing frameworks are available in most programming languages [3, 8, 11, 20]. According to a JetBrains survey conducted in 2021, Python's `Hypothesis` framework [11][1] had an estimated user base of 500,000 [7]. However, these 500,000 users account for merely 4% of the total Python user base, while the framework's maintainers estimate the potential user base to be at least 25%. In comparison, the most widely used testing framework, pytest, holds a 50% market share [7].

As mentioned, property-based testing frameworks provide built-in data generators only for common data types. Thus, when tasked with testing machine learning models using property-based testing, testers must either write generator functions from scratch for each model or use the dataset's information to develop properties. We argue that this approach often misses certain types of faults. To cope with this, we propose a property-driven approach to testing machine learning models. In our approach, we build a decision tree from the available training data and leverage the structure of this model as the basis for test data generation.

4 PROPERTY-BASED TESTING FOR MACHINE LEARNING MODELS

Our property-driven approach is specifically designed for supervised machine learning models. The models generated by supervised learning algorithms can be represented by the function illustrated in Equation 1.

$$M : F_1 \times \ldots \times F_n \rightarrow Z_1 \times \ldots \times Z_m \qquad (1)$$

In the function shown in Equation 1, F_i denotes the value of feature i (i.e., predictor), $1 \leq i \leq n$, and every Z_j, $1 \leq j \leq m$, represents the classes (i.e., response) for the jth label. When $m > 1$,

[1] https://hypothesis.readthedocs.io/en/latest/

the learning problem is a multilabel classification problem. Specifically, when $|Z_j| = 2$ for all j, the learning problem is a binary classification problem, when $|Z_j| > 2$ for some j, it is a multiclass classification problem. For our study, we assume $m = 1$ (i.e., a single-label problem where each instance must be categorized into exactly one of two or more classes) and refer to the response in Equation 1 as Z. In most settings, the resulting function is treated as a *black box*, in the sense that no one is concerned with the form of the function as long as it yields accurate predictions for Z [9].

We are interested in understanding the association between Z and $F_1, \ldots, F_n$: we cannot treat the function in Equation 1 as a black box. Therefore, we want to estimate the underlying function, but our goal is not necessarily to make predictions for Z; instead, we utilize an estimate of how the function maps the features to predictions to guide automatic random test data generation through properties. In this setting, we seek to address a problem related to machine learning interpretability [13] in order to better inform the random generation of test data. The next section outlines our approach for addressing this issue.

4.1 Decision Tree Surrogate Model

A method for extracting insights from machine learning models, thereby explaining the behavior of a black-box model (i.e., relationships either inherent in the data or those learned by the model), is the application of a *global surrogate* model [12]. In essence, a global surrogate is an interpretable model that is trained on the same dataset to approximate the predictions of the black-box model. Consequently, by interpreting the global surrogate model, it becomes feasible to derive conclusions regarding the black-box model under test. The main goal is to approximate the black-box prediction function as closely as possible using the surrogate model prediction function, with the constraint that the surrogate model prediction function must be interpretable. As a result, any interpretable model can serve as surrogate.

Training a surrogate model is a model-agnostic method, as it does not require any information about the inner workings of the black-box model. As stated by Molnar [12], the global surrogate method can be applied even if the machine learning model under examination is replaced with a different black-box model. This is possible because the choice of the black-box model is independent of the type of surrogate model used. In our approach, we chose to employ a decision tree as the surrogate model.

Decision tree is a learning algorithm that results in tree-like graph models that are amenable to human understanding [9]. The main idea behind our property-driven approach is to train an interpretable model (i.e., a decision tree) to predict the behavior of the black-box model. Since the resulting decision tree is a white-box model, we can then leverage the tree-like internal structure of this surrogate model to guide the selection and generation of test data. Furthermore, the internal structure of the decision tree allows us to apply decision tree coverage criteria [15] when rendering the surrogate model into properties. This enables us to gain greater control over the test data generation.

A decision tree algorithm works by recursively splitting the feature space into non-overlapping regions. These splits are guided by rules that create a tree-like structure. Internal nodes in the tree represent the decision points, while leaf nodes represent the final outcome after a series of decisions [9]. In simpler terms, internal nodes map the relationships among the features and leaf nodes denote the potential outcomes. The rationale behind our encoding decision tree coverage criteria into properties is to further leverage the resulting tree structure. This allows us to generate test data that effectively traverses the surrogate model (i.e., reaches all leaf nodes). This approach is analogous to evaluating test data adequacy for traditional programs, where a common criterion is to ensure coverage of all program branches. Consequently, the resulting properties generate test data that randomly samples inputs to “cover” all leaves in the surrogate model.

4.2 Decision Tree Coverage Criteria

Traditional approaches to selecting candidate test data for evaluating machine learning models often rely on randomization.We argue that such random selection methods are insufficient and ineffective for thorough model evaluation. The results from our previous investigation [15] suggest that increasing leaf/decision coverage leads to a more effective test suite. Moreover, this prior research also established criteria grounded in the premise that the internal structure of decision tree models can be exploited to guide test data sampling. These criteria are straightforward and can be defined as follows:

> **Decision Tree Coverage (DTC) Definition:** given a decision tree model M, a test set T is deemed DTC-adequate if it contains tests that traverse the tree from its root to every leaf node at least once.

The other criterion is founded on the widely accepted principle in software testing that test cases exploring boundary conditions are essential for effective testing [14]. This criterion employs the black-box test design technique known as boundary value analysis [14], which is utilized to derive test cases that examine the limits of the input domain. However, this technique has been adapted for application in a white-box context. When applied to decision trees, the range of values represented in the internal decision nodes is analyzed and used to derive test cases. Formally, this criterion can be described as follows:

> **Boundary Value Analysis (BVA) Definition:** test cases are strategically designed to target valid boundary values of decision nodes. Specifically, when designing test cases, this criterion emphasizes the selection of input values that explore both the lower and upper limits of each decision.

The next section details how the internal structure of a decision tree is translated into properties for systematic test data generation.

4.3 A Motivating Example

As previously mentioned, property-based testing fundamentally depends on advanced tool support. To address this requirement, we developed a proof-of-concept tool that streamlines our property-based approach to test data generation.[2] Our tool is built on top of `Hypothesis` [11], extending this property-based test generation library with a decision tree property generator and integration with

[2]Our tool is available at the following repository: https://github.com/ricardo-s-m/property_driven_ml_models.

`pytest`.By leveraging `Hypothesis`, our tool inherits its robust test generation functionalities, enhancing both its practical utility and overall performance.

This section outlines our approach, detailing the main functionalities of the tool and its integration into the testing workflow. To illustrate our property-driven approach, we apply it to the well-known Iris flower dataset[3] [5]. Figure 1 shows the resulting decision tree model obtained by applying the decision tree algorithm to the Iris dataset. The resulting decision tree has 15 nodes, with eight leaf nodes and six internal nodes.[4]

To simplify the example and enhance clarity, we restrict our focus to a single path within the decision tree, rather than considering all possible paths. Specifically, we examine the path from the root node (i.e., node #0) to node #5: [0, 2, 3, 4, 5]. By examining this specific path, we demonstrate how the tool's property generator leverages information encoded within internal nodes to apply the decision tree coverage criteria and define intervals for random test data generation.

As mentioned, internal nodes represent the relationships among the features, while leaf nodes denote the potential outcomes. Thus, when traversing the specified path, the property generator produces a property designed to randomly generate test data that should be classified as class 1, i.e., *Iris versicolor.* To generate this test data, the property generator derives an interval of possible values based on the decisions in the internal nodes along the path. For instance, considering the path from node #0 to node #5, the only features present within the internal nodes pertain to petal length (x[2]) and petal width (x[3]). Additionally, the generated code includes an assertion to verify that the model under test predicts class 1 (i.e., *Iris versicolor*) when provided with test data that follows the path from node #0 to node #5 in the decision tree.

As shown in Table 1, features are analyzed to derive tuples representing intervals. These intervals are denoted as pairs comprising the minimum and maximum values: (minimum, maximum). Notably, most paths do not encompass information regarding all features. Features that do not appear in a given path are not specified and are represented by the tuple (?, ?). In this context, ? denotes the inability to extract information for either the minimum or maximum value of a given feature while traversing a path. Additionally, it is possible for a feature to be described in a given path solely in terms of its minimum or maximum value, e.g., (?, maximum).

Table 1: Information extracted from the internal nodes when following the path from node #0 to node #5.

Sepal Length (x[0])	**Sepal Width** (x[1])	**Petal Length** (x[2])	**Petal Width** (x[3])	**Assertion**
(?, ?)	(?, ?)	(?, 4.95)	(0.80, 1.65)	1

In the subsequent step, the property generator examines incomplete intervals represented as tuples that contain either a minimum or a maximum value, but not both, i.e., (minimum, ?) or (?, maximum). To complete these intervals, the property generator extracts data from the training dataset. Specifically, our approach identifies the unique values that characterize each feature of every class type. For instance, when analyzing a tuple with a missing maximum value, (minimum, ?), the property generator searches the dataset for the largest value corresponding to the given feature, considering only entries labeled as class 1. Similarly, for each tuple with a missing minimum value, our property generator searches the dataset for the smallest value that can describe that feature. After this step, the property generator identifies that the smallest possible value in the dataset for the petal length of an Iris versicolor is 1, as shown in Table 2.

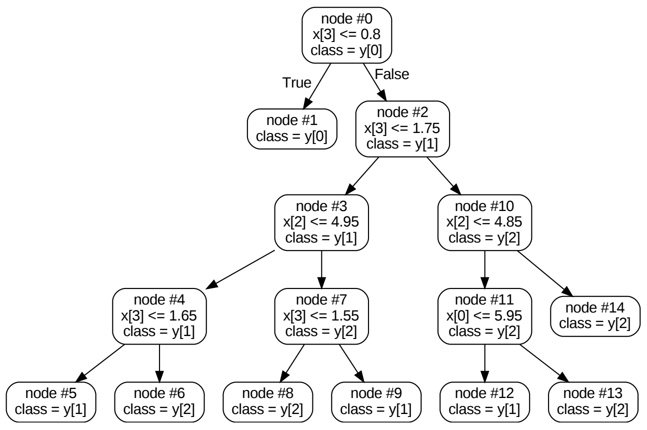

Figure 1: Decision tree model generated from the Iris dataset. The features depicted in the figure correspond to: x[0] = sepal length, x[1] = sepal width, x[2] = petal length, x[3] = petal width. The Iris dataset contains information regarding three classes of iris plants, each representing a different species: Iris setosa (class 0), Iris versicolor (class 1), and Iris virginica (class 2).

Table 2: Updated intervals extracted from the internal nodes when following the path from node #0 to node #5, with missing values in the intervals supplemented by data from the dataset.

Sepal Length (x[0])	**Sepal Width** (x[1])	**Petal Length** (x[2])	**Petal Width** (x[3])	**Assertion**
(?, ?)	(?, ?)	(1.00, 4.95)	(0.80, 1.65)	1

In the third step, the property generator addresses features that are not present within the decision nodes for a specific path. Considering the path from node #0 to node #5, the features x[0] (i.e., sepal length) and x[1] (i.e., sepal width) do not appear in any of the decision nodes. During model training and generation, this omission indicates that the learning algorithm inferred these features as irrelevant to the classification outcome. In the proposed approach, the property generator randomly extracts a subset of values (i.e., three values) from the training dataset for each feature omitted

[3]This dataset is also known as Fisher's Iris data set.
[4]The decision tree model was generated using the `scikit-learn` library. For simplicity's sake, we set `max_depth=4` to limit the maximum depth of the decision tree in this example.

from the analyzed path. These values are derived from data samples belonging to the same class as the test assertion (i.e., the leaf node). The final set of values and intervals extracted from the example path are shown in Table 3.

Table 3: Final set of values and intervals extracted from the training dataset and internal nodes along the path from node #0 to node #5.

Sepal Length (x[0])	**Sepal Width** (x[1])	**Petal Length** (x[2])	**Petal Width** (x[3])	**Assertion**
[5.1, 5.4, 6.3]	[2.5, 2.7, 3.4]	(1.00, 4.95)	(0.80, 1.65)	1

As mentioned in Section 4.2, we encode two decision tree criteria into properties: DTC and BVA. Our tool generates a test file for each criterion. For the generation of the DTC test suite, all the necessary information is in Table 3. Utilizing this information, our tool generates a Python test code file that leverages the `Hypothesis` framework for property-based test data generation and the `pytest` framework for its assertion utilities.

The resulting test file in Listing 1 comprises two main parts: a test function (i.e., a function with the `test_` prefix in lines 6–11) and a **`@given`** decorator that specifies how the function's arguments should be generated (lines 1–4). The `sampled_from` and `floats` functions return what, in the lexicon of `Hypothesis`, is termed a *search strategy*. In essence, a search strategy is an object with methods that describe how to generate specific types of values. These strategies offer various arguments that allow for customization of the test data generation process. In the example shown in Listing 1, we employ two distinct search strategies: randomly selecting a value from a list (`sampled_from` function, lines 1 and 2) and generating floating-point numbers within a specified range by defining minimum and maximum values (`floats` function, lines 3 and 4). The **`@given`** decorator then takes the test function and transforms it into a parameterized one that runs 100 times (line 5) against a range of randomly generated data matching the search strategies. In each test run, features values received as arguments are mapped to an input list (line 7), and used to call the black-box model under test in line 9. The value predicted by the model is then compared with the expected value in line 11.

Listing 1: DTC test suite code.

```
1  @given(st.sampled_from([5.1, 5.4, 6.3]),
2            st.sampled_from([2.5, 2.7, 3.4]),
3            st.floats(min_value=1.00, max_value=4.95),
4            st.floats(min_value=0.80, max_value=1.65))
5  @settings(phases=[Phase.generate], max_examples=100)
6  def test_pbt_dtc_1(self, feat_1, feat_2, feat_3, feat_4):
7       x_test = [feat_1, feat_2, feat_3, feat_4]
8       y_expected = [1]
9       y_predicted = self.model.predict([x_test]).tolist()
10
11      assert y_expected == y_predicted
```

To generate the BVA test suite, our property generator begins with the values shown in Table 1. To randomly generate values near the boundaries of each interval, the property generator narrows down each interval. For instance, given an interval with minimal and maximal values, our tool randomly selects either the minimal or the maximal value and generates a new corresponding value for the tuple. When the minimal value is chosen, our implementation assigns a new maximum value that narrows the interval. Specifically, the new maximum value is set closer to the minimal value, thereby generating data inputs that are near the boundary or at the edges rather than the center of the input domain. This approach increases the likelihood of uncovering potential issues. Features with only one boundary value, either minimum or maximum, are handled by assigning a new value close to the existing boundary, thus narrowing the interval.

As shown in Table 4, the same values previously extracted from the dataset are utilized for features that do not appear in any decision node along the path under analysis. As for petal length, since only the maximum value for the interval (i.e., 4.95) appears in the decision nodes, the property generator refines the interval by assigning a minimum value close to this maximum boundary: (4.16, 4.95). This skews the test data generation process towards the upper limit. For petal width, given that both the minimum and maximum values are available, the property generator randomly selects which boundary to modify. In this example, the upper limit is adjusted to a value closer to the lower limit, thereby narrowing the test data generation process to values nearer to the lower boundary. The code of the resulting test suite is shown in Listing 2.

Table 4: Set of values and intervals extracted from the training dataset and internal nodes upon applying the BVA criterion for test data generation. For the sake of brevity, the column Assertion was omitted.

Sepal Length (x[0])	**Sepal Width** (x[1])	**Petal Length** (x[2])	**Petal Width** (x[3])
[5.1, 5.4, 6.3]	[2.5, 2.7, 3.4]	(4.16, 4.95) **(focus on upper boundary)**	(0.80, 0.97) **(focus on lower boundary)**

Listing 2: BVA test suite code.

```
@given(st.sampled_from([5.1, 5.4, 6.3]),
          st.sampled_from([2.5, 2.7, 3.4]),
          st.floats(min_value=4.16, max_value=4.95),
          st.floats(min_value=0.80, max_value=0.97))
@settings(phases=[Phase.generate], max_examples=100)
def test_pbt_bva_1(self, feat_1, feat_2, feat_3, feat_4):
     x_test = [feat_1, feat_2, feat_3, feat_4]
     y_expected = [1]
     y_predicted = self.model.predict([x_test]).tolist()

     assert y_expected == y_predicted
```

5 EXPERIMENT SETUP

We set out to compare our property-driven approach to testing machine learning models with cross-validation, which is a resampling technique commonly used to gauge the performance of machine learning models. We conjecture that our property-driven approach is more suitable for generating test suites for machine learning-based programs. This hypothesis is based on the fact that our approach leverages the internal structure and decision-making information of decision tree models, utilizing decision tree coverage criteria to identify test requirements. Furthermore, by employing property-based test data generation, our approach can explore a vast range of input data, often more than traditional example-based tests, leading to more comprehensive coverage. In our approach, all test data generation is based on properties extracted from a global surrogate model. We assume that decision nodes play a key role in the model's behavior and that it is possible to design more effective test cases by capitalizing on the structure and underlying decision information of the model.

We designed an experiment to answer the following research question (RQ):

RQ: How effective are the test cases derived from our property-based driven approach in comparison with random test cases from a 10-fold cross validation?

5.1 Scoping

The scope of our experiment is determined by defining its objectives, which were outlined using the Goal/Question/Metric (GQM) template [19] as follows:

Analyze our property-driven approach
for the purpose of comparison
with respect to their effectiveness
from the point of view of the researcher
in the context of testing machine learning based programs.

In Section 5.2 we present the hypotheses we used to investigate the RQ and provide the *operational definition* [17] of effectiveness.

5.2 Hypotheses Formulation

We framed our prediction as follows: our property-driven approach to machine learning testing is more effective than random testing. In this section, we turn our RQ and prediction into hypotheses, allowing us to conduct statistical tests. Consequently, our RQ was translated into the following hypotheses:

Null hypothesis, H_0: there is no difference in effectiveness between our property-driven approach to testing machine learning models and random testing.

Alternative hypothesis, H_1: there is a significant difference in effectiveness between our property-driven approach to testing machine learning models and random testing.

Our main objective is to determine whether our property-driven approach leads to more effective test cases. To evaluate this assumption, our property-driven approach begins by training an interpretable model (i.e., a decision tree) to predict the behavior of the model under test, which is typically a black-box model. The resulting interpretable model is then rendered into properties that generate test data based on two decision coverage criteria (as described in Section 4.3).

Subsequently, we use this randomly generated test data to evaluate the performance of the model under test, which, in our experiment, is a k-Nearest Neighbors (k-NN) model. We train the black-box models by applying k-NN to the original dataset. We employ 10-fold cross-validation for both training and testing the k-NN models. Finally, the results obtained from this 10-fold cross-validation are compared with the results achieved by subjecting the black-box models to the property-based test suites.

There are many different methods for evaluating and comparing machine learning models. For this study, we leverage a set of widely used performance metrics, including precision, recall, accuracy, and the F_1 score, to assess the effectiveness of the models employed. The dependent variable of interest for addressing our RQ is *effectiveness*, which is measured using these performance metrics within the context of this experiment.

In the context of our study, if a model M performs worse on test inputs derived from a random test data selection approach A_1 compared to test inputs generated by applying approach A_2, then A_1 is considered *more effective* than A_2. Given a metric f, that measures the quality of a model M, and a test suite T, we use $f(M(T))$ to denote the score obtained by running T against M according to f. Consequently,

$$f(M(T_1)) < f(M(T_2))$$

indicates that T_1 is more effective than T_2 according to f because T_1 reveals more instances in which M *fails* compared to T_2.

5.3 Instrumentation and Execution

We developed two types of experiment objects: *(i)* Python scripts for loading and processing datasets (including test case generation using our tool) and *(ii)* scripts for analyzing the results. We utilized Google Colaboratory (Colab),[5] a cloud-based environment for running Python scripts in the browser. Google Colab enables users to create notebooks[6] that integrate executable Python code with rich text in a single document. For data wrangling and analysis, we employed `pandas`.[7] All machine learning algorithms in our experiment objects were implemented using `scikit-learn`,[8] which is the most prominent Python library for machine learning [6, 9].

To evaluate the effectiveness of our property-driven approach, we used the resulting randomly generated test suites to assess k-NN models. To this end, we first employ 10-fold cross-validation for both training and testing the k-NN models. An overview of this training and testing step is shown in Figure 2. We selected 5-NN for our evaluation due to the simplicity of k-NN, where model generation involves merely storing the training data.

Following a 10-fold cross-validation, each of the resulting 5-NN models are subjected to the property-based test suites, as shown in Figure 3. The test suites shown in Figure 3 use properties to automatically generate complex, random, valid inputs. These inputs

[5]https://colab.research.google.com/
[6]Google Colab notebooks are Jupyter notebooks.
[7]https://pandas.pydata.org
[8]https://scikit-learn.org/stable/

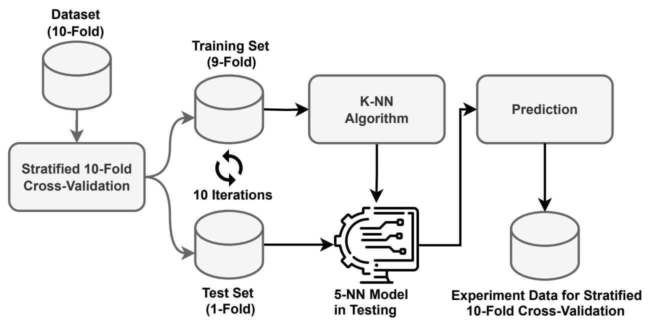

Figure 2: The stratified 10-fold cross-validation process used in our experiment. The dataset is divided into 10 folds, with 9 folds used as the training set and 1 fold as the test set, iterated 10 times. The 5-NN model is then applied to the testing phase, and the results are recorded for each iteration.

are then fed into the test harness implemented by our tool, which assesses them against the black-box model under evaluation (a 5-NN model in the context of our experiment). As mentioned, to evaluate model performance, we focus on four key metrics: precision, recall, accuracy, and F_1. After training the models on the datasets, we store their prediction results for further analysis.

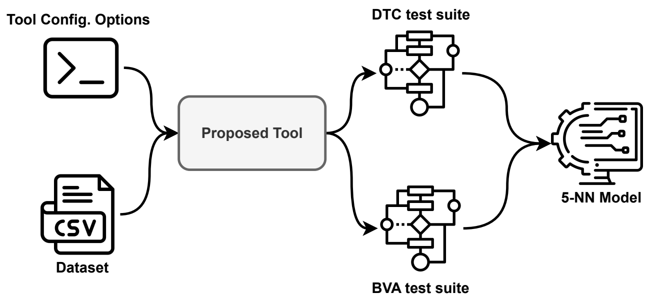

Figure 3: Configuration and application of the proposed tool for generating test suites for the 5-NN model. Upon receiving a dataset, the tool employs a decision tree algorithm to build a global surrogate model. Subsequently, the property generator component of the tool turns the information gleaned from the surrogate model into a set of properties based on two decision tree testing criteria, producing the DTC and BVA test suites.

5.4 Datasets

We selected 21 commonly used for training machine learning models. These datasets tend to be less complex and cleaner than real-world data, leading to the creation of smaller and more interpretable decision trees. An overview of the datasets used in our experiment is shown in Table 5.

As shown in Table 5, the datasets used in our experiment exhibit significant heterogeneity in terms of sample size, number of attributes, class distribution, and the number of classes. The sample sizes range from as few as 150 (Iris) to over 45,000 (Bank Marketing), indicating a wide variation in dataset scale. The number of attributes also varies significantly, from as few as 4 (Haberman's Survival) to as many as 61 (Sonar; Mines vs. Rocks). Additionally, the class distribution within the datasets is often imbalanced, as seen in the Oil Spill dataset with 896 samples in one class and only 41 in the other. Furthermore, the number of classes ranges from binary classification tasks (e.g., Banknote Authentication) to more complex multiclass problems such as Glass Identification with six classes.

Table 5: Overview of the datasets used in our experiment.

Dataset	Samples	Attributes	Classes	Samples per Class
Bank Marketing	45,211	17	2	[39,922; 5,289]
Banknote Authentication	1,372	5	2	[762; 610]
Blood Transfusion	748	5	2	[570; 178]
Breast Cancer Wisconsin	569	31	2	[212; 357]
Cleveland Heart Disease	297	14	2	[160; 137]
Glass Identification	214	10	6	[70; 76; 17; 13; 9; 29]
Haberman's Survival	306	4	2	[225; 81]
Horse Racing Tipster Bets	38,248	10	2	[30,565; 7,683]
Indian Liver Patient	579	11	2	[414; 165]
Ionosphere	351	35	2	[126; 225]
Iris	150	5	3	[50; 50; 50]
Oil Spill	937	50	2	[896; 41]
Page Blocks Classification	5,473	11	5	[4,913; 329; 28; 88; 115]
Phoneme	5,404	6	2	[3,818; 1,586]
Pima Indians Diabetes	768	9	2	[500; 268]
Seeds	210	8	3	[70; 70; 70]
Sonar; Mines vs. Rocks	208	61	2	[111; 97]
Students Knowledge Levels	403	6	4	[50; 129; 122; 102]
Vertebral Column	310	7	3	[100; 60; 150]
Wine Recognition	178	14	3	[59; 71; 48]
Woods Mammography	11,183	7	2	[10,923; 260]

6 EXPERIMENTAL RESULTS

In this section, we present the experimental results of comparing our property-driven test data generation approach with the widely used 10-fold cross-validation resampling technique. Before delving into this comparison, we outline the outcomes of applying our tool to the 21 selected datasets. This discussion primarily focuses on the number of properties extracted from the models, as well as the number of test inputs generated from DTC- and BVA-based properties.

Table 6 presents an overview of the properties extracted from the global surrogate models and the number of unique test inputs generated by the proposed tool. The number of properties extracted varies significantly across datasets, ranging from as few as 9 for the Iris dataset to as many as 7,389 for the Horse Racing Tipster Bets dataset.

The average number of properties extracted from the global surrogate models across all datasets is 609.76, resulting in an average of 60,976.19 test inputs generated per dataset. The total number of randomly generated test inputs follows directly from the number of properties: this is the case because, as shown in line 5 of Listings 1 and 2, the tool is set to generate 100 test inputs for each property. Thus, the dataset that generated the largest number of test inputs was the Horse Racing Tipster Bets dataset, with 738,900 inputs, while the Iris dataset produced the fewest test inputs, totaling 900.

Table 6: Overview of the number of properties and test inputs generated from the datasets in our experiment.

Dataset	Properties	Test Inputs per Property	Generated Test Inputs	Unique Test Inputs	
	DTC / BVA	DTC / BVA	DTC / BVA	DTC	BVA
Bank Marketing	3,679	100	367,900	364,747	364,422
Banknote Authentication	27	100	2,700	2,568	2,535
Blood Transfusion	198	100	19,800	19,717	19,695
Breast Cancer Wisconsin	22	100	2,200	2,142	2,143
Cleveland Heart Disease	57	100	5,700	5,641	5,652
Glass Identification	50	100	5,000	4,889	4,891
Haberman's Survival	103	100	10,300	10,277	10,279
Horse Racing Tipster Bets	7,389	100	738,900	734,365	734,202
Indian Liver Patient	111	100	11,100	10,968	10,976
Ionosphere	23	100	2,300	2,278	2,279
Iris	9	100	900	877	874
Oil Spill	35	100	3,500	3,458	3,456
Page Blocks Classification	165	100	16,500	16,137	16,107
Phoneme	521	100	52,100	51,667	51,647
Pima Indians Diabetes	129	100	12,900	12,623	12,622
Seeds	16	100	1,600	1,567	1,564
Sonar, Mines vs. Rocks	22	100	2,200	2,171	2,174
Students Knowledge Levels	31	100	3,100	3,040	3,023
Vertebral Column	45	100	4,500	4,396	4,390
Wine Recognition	12	100	1,200	1,162	1,164
Woods Mammography	161	100	16,100	15,912	15,914
Descriptive Statistics					
Max	7,389	–	738,900	734,365	734,202
Min	9	–	900	877	874
Mean	609.76	–	60,976.19	60,504.86	60,476.62
Median	50	–	5,000	4,889	4,891

Upon further analysis, we found that our tool produces a high number of unique (i.e., non-duplicate) test inputs, averaging 60,504.86 for DTC-based properties and 60,476.62 for BVA-based properties. This demonstrates the tool's efficiency in generating diverse test data while minimizing duplication.

The results presented in Table 6 indicate that properties based on the DTC criterion lead to the largest number of unique test cases. This is evident from the consistently higher unique test input counts for DTC compared to BVA. For instance, considering the Bank Marketing dataset, DTC-based properties generated 364,747 unique test inputs compared to BVA's 364,422. Similarly, for the Horse Racing Tipster Bets dataset, DTC-based properties produced 734,365 unique test inputs, while BVA-based properties yielded 734,202. Across the board, properties based on DTC consistently outperform BVA-based properties in generating a higher number of unique test cases, which might indicate that DTC-based properties are more effective at exploring a more diverse set of scenarios within the test data.

6.1 Hypothesis Testing

To investigate whether our property-driven approach leads to the creation of test data that is more effective at evaluating the performance of machine learning models, we conducted two-tailed paired t-tests to assess differences in the mean values of the metrics used in our experiment (i.e., precision, recall, accuracy, and F_1). Prior to performing these tests, we verified the normality of the data distributions using the Shapiro–Wilk test. The results of the Shapiro-Wilk test confirmed that all distributions of the results, derived from applying test data generated from DTC- and BVA-based properties to a k-NN model, as well as those from a 10-fold cross-validation, are normal (as shown in Table 7).

As shown in Table 7, 10-fold cross-validation consistently yields the highest scores across all metrics, indicating that it may not push the models to their limits as effectively as the property-driven approach. DTC- and BVA-based properties test suites generally produce lower scores, suggesting they are more effective at testing the models' boundaries and exercising edge cases.

Based on the results of the paired t-tests shown in Table 8, there are statistically significant ($p \leq 0.001$) differences between DTC-based properties and 10-fold cross-validation, as well as BVA-based properties and 10-fold cross-validation. Specifically, the test data from property-driven approaches resulted in lower scores across all metrics, which suggests that these approaches are better suited at more thoroughly evaluating the models. Although not reported in Table 8, it is worth noting that the differences between DTC and BVA properties were not statistically significant, indicating that these two approaches perform similarly in terms of the evaluated metrics.

The violin plots shown in Figure 4 provide a comparative analysis of the three approaches. As mentioned, the plots corroborate that 10-fold cross-validation yields higher scores across all metrics, indicating a lesser degree of model stress testing compared to the property-driven approaches. Conversely, both DTC- and BVA-based test suites show lower median scores and greater variability. While test data generated from DTC- and BVA-based properties perform similarly, the latter often exhibits slightly better performance, with lower scores in most cases, suggesting it might be more effective in generating challenging test cases.

6.2 Discussion

Our results suggest that while 10-fold cross-validation remains a popular and reliable method for model evaluation, our property-driven approach provides a more rigorous evaluation by generating data that thoroughly explores the models' limitations. This is crucial for identifying potential weaknesses and ensuring robustness, particularly in real-world applications where edge cases and uncommon scenarios can significantly impact performance. The lower scores obtained through our property-driven approach highlight the need for more nuanced evaluation criteria that prioritize robustness and boundary testing, ensuring that models perform well in typical scenarios and under more challenging conditions. Therefore, incorporating property-driven test suites into the evaluation process can enhance the comprehensiveness of model testing, leading to more reliable and resilient machine learning models.

7 THREATS TO VALIDITY

One potential threat to external validity is that our selected datasets may not represent the broader target population. These datasets can be considered "toy" datasets, as they are relatively smaller and cleaner than those encountered in real-world scenarios. Therefore, we cannot rule out the possibility that different results might have been obtained if larger and more complex datasets had been used in our experiment.

A threat to construct validity arises from the possibility that the measures employed in the experiment may not adequately evaluate the effects we intended to investigate. Specifically, while precision,

Table 7: Performance comparison of the two property-driven approaches and 10-fold cross-validation across four metrics.

Dataset	DTC Test Suite				BVA Test Suite				10-fold Cross-Validation			
	Precision	Recall	Accuracy	F_1	Precision	Recall	Accuracy	F_1	Precision	Recall	Accuracy	F_1
Bank Marketing	0.51	0.43	0.51	0.48	0.51	0.45	0.51	0.48	0.88	0.87	0.88	0.86
Banknote Authentication	0.78	0.78	0.78	0.78	0.69	0.68	0.69	0.69	1.00	1.00	1.00	1.00
Blood Transfusion	0.52	0.53	0.52	0.53	0.54	0.54	0.54	0.53	0.76	0.73	0.76	0.74
Breast Cancer Wisconsin	0.75	0.75	0.75	0.75	0.73	0.73	0.73	0.74	0.94	0.93	0.94	0.94
Cleveland Heart Disease	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.64	0.63	0.64	0.64
Glass Identification	0.37	0.34	0.37	0.37	0.36	0.32	0.36	0.35	0.66	0.63	0.66	0.64
Haberman's Survival	0.62	0.58	0.62	0.60	0.61	0.57	0.61	0.59	0.72	0.69	0.72	0.69
Horse Racing Tipster Bets	0.49	0.38	0.49	0.43	0.49	0.39	0.49	0.43	0.77	0.72	0.77	0.70
Indian Liver Patient	0.51	0.41	0.51	0.52	0.50	0.42	0.50	0.49	0.67	0.66	0.67	0.66
Ionosphere	0.60	0.58	0.60	0.74	0.62	0.61	0.62	0.77	0.84	0.83	0.84	0.86
Iris	0.58	0.59	0.58	0.61	0.57	0.57	0.57	0.63	0.95	0.95	0.95	0.96
Oil Spill	0.50	0.37	0.50	0.64	0.50	0.35	0.50	0.68	0.96	0.94	0.96	0.92
Page Blocks Classification	0.40	0.39	0.40	0.40	0.41	0.40	0.41	0.40	0.96	0.95	0.96	0.95
Phoneme	0.49	0.49	0.49	0.49	0.51	0.51	0.51	0.51	0.89	0.89	0.89	0.89
Pima Indians Diabetes	0.49	0.49	0.49	0.49	0.50	0.48	0.50	0.50	0.71	0.70	0.71	0.70
Seeds	0.65	0.65	0.65	0.66	0.74	0.73	0.74	0.74	0.87	0.86	0.87	0.88
Sonar; Mines vs. Rocks	0.66	0.66	0.66	0.67	0.67	0.67	0.67	0.69	0.80	0.80	0.80	0.81
Students Knowledge Levels	0.54	0.53	0.54	0.55	0.50	0.46	0.50	0.54	0.88	0.88	0.88	0.89
Vertebral Column	0.60	0.60	0.60	0.61	0.57	0.57	0.57	0.58	0.82	0.82	0.82	0.84
Wine Recognition	0.41	0.41	0.41	0.44	0.35	0.36	0.35	0.40	0.69	0.67	0.69	0.71
Woods Mammography	0.60	0.52	0.60	0.64	0.61	0.54	0.61	0.63	0.99	0.98	0.99	0.98
Descriptive Statistics												
Max	0.78	0.78	0.78	0.78	0.74	0.73	0.74	0.77	1.00	1.00	1.00	1.00
Min	0.37	0.34	0.37	0.37	0.35	0.32	0.35	0.35	0.64	0.63	0.64	0.64
Mean	0.55	0.52	0.55	0.57	0.55	0.52	0.55	0.57	0.83	0.82	0.83	0.82
Median	0.52	0.52	0.52	0.55	0.51	0.51	0.51	0.54	0.84	0.83	0.84	0.86
Std Deviation	0.11	0.12	0.11	0.12	0.11	0.12	0.11	0.12	0.12	0.12	0.12	0.12
Normality Test												
Shapiro-Wilk (W)	W=0.96, p=0.45	W=0.96, p=0.46	W=0.96, p=0.45	W=0.97, p=0.72	W=0.95, p=0.33	W=0.96, p=0.58	W=0.95, p=0.33	W=0.96, p=0.54	W=0.94, p=0.21	W=0.93, p=0.12	W=0.94, p=0.21	W=0.92, p=0.07

recall, accuracy, and F_1 score are commonly used to evaluate machine learning models, they may not fully capture the adequacy of test data for machine learning-based programs. However, the widespread use of these performance metrics in evaluating machine learning models helps mitigate this threat. We used these metrics because the smaller the value, the better the test data is at pushing the model under test to its limits, effectively testing edge cases, and exploring scenarios not encountered during training. We acknowledge that the random generation of synthetic test data can introduce biases not present in real-world data. However, our test data is designed based on a global surrogate model that maps the model's behavior under test. Therefore, we anticipate that an adequately trained model should perform well on this type of synthetic data, as it closely approximates real data. Despite this, we recognize the inherent difficulty in validating synthetic data to ensure that it accurately represents the diversity and distribution of real-world data. Another potential threat to construct validity stems from possible faults in our tool.

Table 8: Results from the two-tailed paired t-tests.

	Paired Samples Test (t)	
Metric	DTC x Cross-validation	BVA x Cross-validation
Precision	t = -10.97, p-value $\leq$ 0.001	t = -10.88, p-value $\leq$ 0.001
Recall	t = -10.10, p-value $\leq$ 0.001	t = -10.08, p-value $\leq$ 0.001
Accuracy	t = -10.97, p-value $\leq$ 0.001	t = -10.88, p-value $\leq$ 0.001
F_1	t = -10.57, p-value $\leq$ 0.001	t = -10.30, p-value $\leq$ 0.001

8 CONCLUDING REMARKS

The main benefit of property-based testing stems from its ability to generate many inputs from a single specification (i.e., property). This often leads to the discovery of edge and corner cases that example-based tests might overlook. In the context of testing machine learning models, our approach offers a middle ground between example-based test cases and computationally expensive formal methods. Property-based testing retains the precision of traditional formal specifications and their capacity to characterize a model's behavior, while also providing a relatively fast, best-effort validation [7]. Additionally, our approach eliminates the need for testers to manually create properties by by automatically extracting these properties from a global surrogate model. These properties are high-level, abstract descriptions of the expected behavior of the

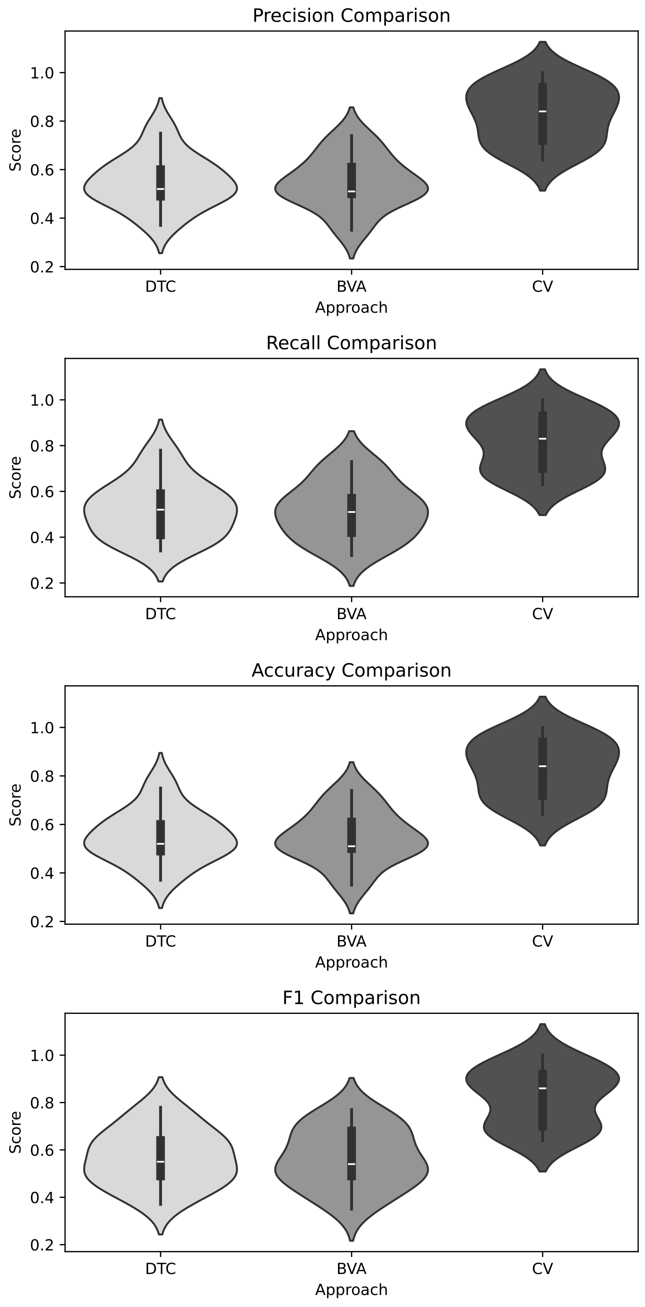

Figure 4: Violin plots comparing the performance of the two property-driven approaches and 10-fold cross-validation across four metrics. These plots show the distribution and variability of the scores, where lower scores indicate better performance.

model under test and are used to generate many test cases automatically. As indicated by our results, property-based test data covers a broad spectrum of possible values, which we surmise results in a thorough examination of the model's behavior. In future work, we plan to investigate the implications of using different white-box models as global surrogates on the resulting properties and the generated test data.

Acknowledgments. The authors would like to acknowledge the financial support of the São Paulo Research Foundation (FAPESP) through grant #2023/00577-8 to Andre T. Endo, the Minas Gerais Research Foundation (FAPEMIG) through grant #APQ-03513-18 to Rafael S. Durelli, and the National Council for Scientific and Technological Development (CNPq) through grants #312086/2021-0 and #308445/2021-0 to Fabiano C. Ferrari and Simone R. S Souza, respectively.

REFERENCES

[1] A. Aggarwal, P. Lohia, S. Nagar, K. Dey, and D. Saha. 2019. Black box fairness testing of machine learning models. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 625–635.

[2] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, J. Van Geffen, and A. Warfield. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. ACM, 836–850.

[3] K. Claessen and J. Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, 268–279.

[4] A. L. Corgozinho, M. T. Valente, and H. Rocha. 2023. How Developers Implement Property-Based Tests. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 380–384.

[5] R. A. Fisher. 1936. The Use of Multiple Measurements in Taxonomic Problems. *Annals of Eugenics* 7, 2 (1936), 179–188.

[6] A. Geron. 2019. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow* (2nd ed.). O'Reilly. 600 pages.

[7] H. Goldstein, J. W. Cutler, D. Dickstein, B. C. Pierce, and A. Head. 2024. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. ACM.

[8] F. Hebert. 2019. *Property-Based Testing with Proper, Erlang, and Elixir: Find Bugs Before Your Users Do*. Pragmatic Bookshelf. 376 pages.

[9] G. James, D. Witten, T. Hastie, R. Tibshirani, and J. Taylor. 2023. *An Introduction to Statistical Learning: With Applications in Python*. Springer. 607 pages.

[10] A. Löscher and K. Sagonas. 2017. Targeted property-based testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 46–56.

[11] D. R. MacIver, Z. Hatfield-Dodds, and Many Other Contributors. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019).

[12] C. Molnar. 2020. *Interpretable Machine Learning: A Guide for Making Black Box Models Explainable*. Lulu.com. 318 pages.

[13] C. Molnar, G. Casalicchio, and B. Bischl. 2020. Interpretable Machine Learning – A Brief History, State-of-the-Art and Challenges. In *ECML PKDD Workshops*. Springer, 417–431.

[14] Glenford J. Myers, Corey Sandler, and Tom Badgett. 2011. *The Art of Software Testing* (3 ed.). Wiley Publishing.

[15] S. Santos, B. Silveira, V. Durelli, R. Durelli, S. Souza, and M. Delamaro. 2021. On Using Decision Tree Coverage Criteria forTesting Machine Learning Models. In *Proceedings of the 6th Brazilian Symposium on Systematic and Automated Software Testing (SAST '21)*. ACM, 1–9.

[16] A. Sharma, C. Demir, A. Ngonga Ngomo, and H. Wehrheim. 2021. MLCHECK – Property-Driven Testing of Machine Learning Classifiers. In *20th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 738–745.

[17] Scott W. VanderStoep and Deidre D. Johnson. 2008. *Research Methods for Everyday Life: Blending Qualitative and Quantitative Approaches*. Jossey-Bass. 352 pages.

[18] S. Wang, L. Huang, A. Gao, J. Ge, T. Zhang, H. Feng, I. Satyarth, M. Li, H. Zhang, and V. Ng. 2023. Machine/Deep Learning for Software Engineering: A Systematic Literature Review. *IEEE Transactions on Software Engineering* 49, 3 (2023).

[19] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. 2012. *Experimentation in Software Engineering*. Springer. 236 pages.

[20] K. Yatoh, K. Sakamoto, F. Ishikawa, and S. Honiden. 2014. ArbitCheck: A Highly Automated Property-Based Testing Tool for Java. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 405–412.

[21] J. M. Zhang, M. Harman, L. Ma, and Y. Liu. 2022. Machine Learning Testing: Survey, Landscapes and Horizons. *IEEE Transactions on Software Engineering* 48, 1 (2022), 1–36.