Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

# Evaluating program behavior with different Machine Learning approaches

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

eingereicht von: Martin Eberlein
geboren am: 20.08.1995
geboren in: Nürnberg

Gutachter/innen: Prof. Dr. Lars Grunske
Prof. Dr. Andreas Zeller

eingereicht am: .................... verteidigt am: ....................

# Contents

**1. Introduction** **5**

**2. Related Work** **9**

**3. Background** **11**
3.1. Terminology . . . . . 11
3.2. Machine Learning . . . . . 12
3.2.1. Decision Tree . . . . . 12
3.2.2. Random Forest . . . . . 13
3.2.3. Gradient Boosting Trees . . . . . 14
3.3. Context-Free Grammars . . . . . 16
3.4. Alhazen: Learning Circumstances of Software Behavior . . . . . 17

**4. Evaluating Software Behavior with Different Machine Learning Alternatives** **19**
4.1. Training new Classification Model . . . . . 22
4.2. Extracting new Feature Requirements . . . . . 25
4.3. Continuing the Feedback Loop . . . . . 27
4.4. Explaining the Prediction Theory . . . . . 30

**5. Experimental Setup** **33**

**6. Evaluation** **37**
6.1. AlhazenML as Predictor . . . . . 37
6.1.1. Experimental Results . . . . . 37
6.1.2. Experimental Results over Time . . . . . 43
6.1.3. Experimental Analysis . . . . . 44
6.2. AlhazenML as Producer . . . . . 46
6.2.1. Experimental Results . . . . . 46
6.2.2. Experimental Analysis . . . . . 48
6.3. Threats to Validity . . . . . 50

**7. Discussion and Limitations** **51**

**8. Conclusion and Future Work** **53**

**A. Appendix** **60**

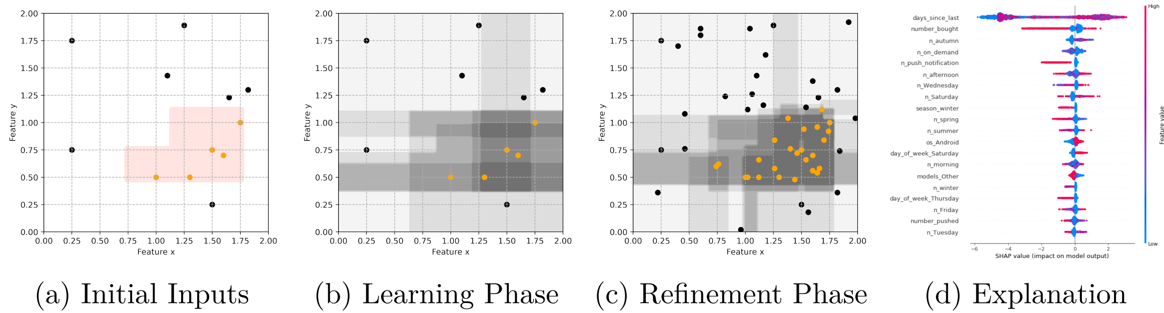
# Abstract

Why does an input fail in an application? To help developers find the root causes of the program failure, Kampmann et al. [KHSZ20] proposed an approach to automatically determine the circumstances of program behavior. Their tool *Alhazen* associates the program's failure with the features of the input data, allowing them to learn and extract the properties that result in *bug-triggering* behavior. First, *Alhazen* uses a grammar to extract the syntactical features of the inputs samples. Then, *Alhazen* trains a Decision Tree to learn the features that are responsible for the behavior in question and creates a hypothesis as to why the inputs result in a failure of the program. Finally, by using the grammar to generate more inputs iteratively, *Alhazen* can refine and strengthen its initial hypothesis. However, Decision Trees are known to overfit the training data and thus are usually outperformed by other machine learning approaches. In this thesis, we build on the work of Kampmann et al. and replace the Decision Tree learner in *Alhazen* with more advanced and more powerful machine learning models, which could capture failure circumstances more precisely. However, while other machine learning estimators may provide better predictions, they are also much harder to interpret or explain. We tackle this problem by proposing a modified learning method to extract and refine two ensemble estimators: the Random Forest and the Gradient Boosting Tree. We implemented our approach AlhazenML as an extension of *Alhazen* and evaluated its effectiveness on a set of ten real-world subjects. Our evaluation shows that our approach was able to improve the behavior classification for five subjects significantly and performs equally to *Alhazen* for the remaining subjects.

# 1. Introduction

Nowadays, software applications are often deployed in safety-critical or insecure domains, where the delivery of correct programs is undeniably crucial. In particular, modern software applications control safety-critical components, including medical diagnoses [LNV+18], traffic control [RBVK18], and self-driving cars [FNR14], where faulty or unreliable programs can have a serve impact on human lives. Therefore, unsurprisingly, developers need to apply extensive testing methods to account for all circumstances. Unfortunately, modern software becomes increasingly complicated to test and verify due to the steady increase in complexity. Nevertheless, recent advantages in testing have shown great success in finding erroneous behavior and vulnerabilities in many different components. Recent testing techniques, like fuzzing [MFS90, FMEH20], generate random input data and enhance or mutate them to trigger potential defects or software vulnerabilities. Although they have proven capable of detecting and generating erroneous input data, they often lack an explanation of why specific input data results in incorrect behavior. However, when diagnosing why a program fails, the first step is to determine the circumstances under which the program failed. Kampmann et al. [KHSZ20] presented an approach to automatically discover the circumstances of program behavior. Their approach associates the program's failure with the syntactical features of the input data, allowing them to learn and extract the properties that result in the specific behavior. Their tool *Alhazen* can generate a diagnosis and explain why, for instance, a particular bug occurs. More formally, Alhazen forms a hypothetical model based on the observed inputs. Additional test inputs are generated and executed to refine or refute the hypothesis, eventually obtaining a prediction model of the circumstances of why the behavior in question takes place. *Alhazen* use a Decision Tree classifier to learn the association between the program behavior and the input features. Although Decision Tree learners are among the most popular machine learning algorithms [TNČT20], given their comprehensibility and simplicity, they often are incredibly prone to small changes in the input data. Minor variations may already result in a complete change in the tree and consequently the final predictions [Bre01].

In this thesis, we build on the work of Kampmann et al. [KHSZ20] and replace the Decision Tree learner in *Alhazen* with more advanced and more powerful machine learning models, namely Random Forests and Gradient Boosting Trees, which could capture failure circumstances more precisely. However, while other machine learning estimators may provide better predictions, they are much harder to interpret or explain. Many so-called *strong learners* trade comprehensibility and interpretability for better accuracy and precision. Due to the lack of interpretability, the main challenges are utilizing better estimators to generate additional inputs to refine the hypothesis and providing the developers with a human-interpretable explanation of the failure circumstances. To limit the scope of this thesis, we focus on white-box machine learning models, particularly ensemble estimators that construct and operate white-box models, specifically Decision Trees. We selected the Random Forest and the Gradient Boosting Tree for our approach. On the one hand, we chose the ensemble estimators due to their




(a) Initial Inputs (b) Learning Phase (c) Refinement Phase (d) Explanation

Figure 1: Overview of our approach iteratively learning to approximate the *error region* (pink shaded area in Subfigure (a)) by generating additional *inputs* (orange and black dots) close to the decision boundaries ((b) and (c)). Finally, we interpret the machine learning models decision and derive an explanation of the features that are responsible for the program behavior (d).

tendency to correct Decision Trees' habit of overfitting to the training data and, on the other hand, due to their recent success for many different classification and regression tasks [CCW18]. Figure 1 gives an overview of the individual steps of our approach and how we adapted the initial steps of *Alhazen* to refine the hypotheses of our new ensemble classifiers. Similar to *Alhazen*, our approach starts with a set of initial inputs - *bug-triggering* and *non-bug-triggering* - and their corresponding input language format also known as grammar (a). Then, the ensemble estimator, i.e., the Random Forest, tries to associate the syntactical features with the program behavior and hypothesizes why the program fails (b). In the next step, we refine the hypothesis by analyzing the decision boundaries of the ensemble classifier and generating additional inputs (c). Finally, in the last step, we analyze why the estimator made specific decisions and explain what part of the input files leads to the observed outcome (d).

With this thesis, we aim to explore and evaluate different machine learning alternatives for *Alhazen*. In addition, we implemented our extension in the tool AlhazenML to evaluate and compare the different machine learning candidates on a set of established real-world programs. The following research questions guide the evaluation of our approach:

**RQ1** Does AlhazenML allow us to predict the behavior of a program more precisely?

**RQ2** Does AlhazenML allow us to produce more defect triggering inputs efficiently?

In summary, this thesis makes the following contributions:

- First, we propose two new tree-based machine learning models for *Alhazen*, namely

the Random Forest and the Gradient Boosting Tree, to efficiently and reliably identify the circumstances of a program's behavior.

- We implement our concepts in AlhazenML and extend the recently proposed *Alhazen* by introducing an adapted learning process.
- We report our results from an extensive evaluation and show that AlhazenML compares favorably for many subjects.

The remainder of this thesis is structured as follows: Section 2 surveys the state of the art of explaining the behavior of programs, test case generation, and explainable machine learning. Section 3 provides the theoretical background on machine learning techniques and context-free grammars, as well as an introduction of the tool *Alhazen*. Then, in Section 4, we explain our approach of incorporating different machine learning alternatives to determine the circumstances of the program's behavior. Section 5 and Section 6 present the experimental methodology and the experimental results of our extensive evaluation. Afterwards, we discuss the results in Section 7. Finally, we conclude this thesis in Section 8 and sketch some directions for future research.

# 2. Related Work

In this Section, we present the related work, literature, and current research directions.

**Program Behavior Classification** Recently, Tizpaz-Niari [TNČT20] proposed an approach to determine and explain differential performance bugs in machine learning libraries. Their tool *DPFuzz* uses an evolutionary fuzzing approach to generate interesting inputs and then clusters them according to their execution and performance time. Finally, *DPFuzz* uses a Decision Tree to determine and explain the performance differences in terms of program inputs and internals. However, in contrast to *Alhazen* and our approach, Tizpaz-Niari et al. do not generate additional inputs according to the predictions of the tree and solely rely on the mutational fuzzing operators.

Machine learning techniques are also often used to learn models that classify programs into benign and malicious software for vulnerability detection. For example, Elish et al. [EE08] use support vector machines (SVMs) to predict defect-prone software modules and show that SVMs can be used to increase the confidence in software applications. Additionally, Lo et al. [LCH+09] proposed a technique that first extracts iterative patterns from program traces of known ordinary and failing executions. Then they perform feature selection to select the most promising features for classification. These features are then used to train a classifier to detect program failures. Furthermore, Delphine Immaculate et al. [DIFBF19] evaluate the practical relevance of deploying machine learning techniques to help developers increase the reliability of their software. Similar to our approach, they consider different models and conclude that Random Forests are preferred due to their generalizability.

**Generative Adversarial Networks** A generative adversarial network (GAN) is a class of machine learning frameworks designed by Goodfellow et al. [GPAM+20]. It contains two neural networks that compete against each other. This technique generates new data with the same statistics as the training. The core idea is based on the "indirect" training through the discriminator, which is also updated dynamically. This concept is also loosely related to *Alhazen* and our approach, where the machine learning model and the input generator can be considered to be two separate players. In our case, the generator tries to fool the machine learning model by generating additional data close to the decision boundaries. By executing the new inputs and comparing the prediction to the actual outcome, we can use these new inputs to improve the learned associations between the input features and the resulting outcome. After several iterations of the feedback loop, the machine learning model can refine its hypothesis and may predict program behavior more precisely.

**Explainable Machine Learning** With the availability of large databases and recent improvements in deep learning methodology, the performance of AI systems is reaching or even exceeding the human level on an increasing number of complex tasks. However, because of their nested non-linear structure, these highly successful machine learning models are usually applied in a black-box manner, i.e., no information is provided

about what exactly makes them arrive at their predictions [SML+21]. Since this lack of transparency can be a major drawback, e.g., in medical applications [LNV+18], or finance [BGMP21], research towards explainable machine learning has recently attracted increasing attention. The literature can often be separated into deriving local or global explanations. While local explanations interpret the machine learning model for exactly one prediction, global explanations focus on the entire model behavior. Ribeiro et al. [RSG16] proposed a local interpretable surrogate model (*LIME*) to explain individual predictions of black-box models. These surrogate models are trained to approximate the decisions of the underlying machine learning model. *LIME* achieves this by probing the black-box model with slightly changed data and observing the predictions of the machine learning model. By only searching locally with slight variations, *LIME* can explain individual predictions. Lundberg and Lee [LL17] proposed a coalitional game-theoretic approach called *SHAP* to combine *LIME* with Shapley Values [Sha16] to determine the contributions of individual features to a final prediction. Later, Lundberg et al. [LEC+20] extended their approach with a faster implementation, particularly for tree-based models, which significantly reduced computational overhead.

While *SHAP* can be used for local explanations, their approach also allows global interpretations of machine learning models (e.g., feature importance, summary plots, or dependency plots). Since many global explanation approaches, including *SHAP*, reduce the black-box models' decision to a limited set of features, Ibrahim et al. [ILMP19] introduced an approach to explain the predictions across many different sub-populations. Their local attributions technique detects global patterns in a dataset by grouping similar data points into a cluster. They then use clustering algorithms to identify the K-local attributions minimizing the pairwise dissimilarity within a cluster, which allows them to derive concrete and precise explanations for similar data.

**Generative Input Generation** A key component of *Alhazen* and our approach is the additional input generation to refine the hypothesis. Randomly generating inputs is an effective means to test a program for robustness. Recent strategies, like fuzzing, have shown immense success in finding many different defects in various software applications [FMEH20]. However, to reach deep layers of the program, inputs must conform to the required input language and, thus, be syntactically valid. Havrikov and Zeller [HZ19] introduced a grammar-based method to syntactically cover elements of the grammar that allows revealing more profound defects cost-effectively. In the context of grammar-based fuzzing, the generation of new inputs can also be guided by probabilities attached to competing rules in the grammar, thus controlling the distribution of syntactical elements. Using a set of input seeds to obtain a probabilistic grammar, Sorumekun et al. [SPH+20] generate similar inputs to the seeds, or, by inverting the probabilities of the grammar, generate dissimilar inputs. Fuzzing can also be combined with learning mechanisms. For example, Cummins et al. [CPML18] use a neural network to learn from real-world code and then generate new inputs to discover various defects. Similarly, Godefroid et al. [GPS17] use a neural network in an unsupervised machine learning setting to test security-critical parser and compiler.

# 3. Background

Our extension of *Alhazen* focuses on iteratively refining machine learning techniques and learning syntactical features of inputs. These syntactical properties are extracted from a context-free grammar and are associated with the program's behavior. In this Section, we define the necessary terminology and present the necessary foundations of our approach. We give a brief overview of different machine learning alternatives, context-free grammars, and a short introduction of the tool *Alhazen*.

## 3.1. Terminology

In the following we will state some terminology that may have different connotations among the literature. These definitions are closely related to [Mol19]:

**Definition 1** (Machine Learning)**.** Machine Learning is a set of methods that allow computers to learn from data to make and improve certain predictions.

**Definition 2** (Machine Learning Algorithm and Model)**.** A Machine Learning Algorithm is used to train a Machine Learning Model that associates input data with the outcome. In the following we will also call them classifier or estimator.

**Definition 3** (Black-Box Model)**.** A black-box model is a machine learning model that does not reveal the decisions it made to map the input data to the output. For instance, Neural Networks are considered black-box models, as we often do not know on what properties of the input data they based their prediction.

**Definition 4** (White-Box Model)**.** Contrary to black-box models, white-box models are often referred to as interpretable models, and allow us to comprehend each decision the model made to derive a prediction. In this thesis, we will mainly focus on white-box models or ensemble models that operate and construct a set of white-box models.

**Definition 5** (Weak/Strong Learner)**.** Weak learners are models that perform just slightly better than random guessing, whereas strong learners can perform arbitrarily well for a given classification task.

**Definition 6** (Ensemble Estimator)**.** An ensemble estimator, in general, is a model that makes predictions by training and operating several different models. Combining different models makes an ensemble machine learning model more flexible, less biased, and less data-sensitive (less variance). We can use ensemble techniques like *bagging* (Section 3.2.2) and *boosting* (Section 3.2.3) to combine many weak learners into one strong learner.

## 3.2. Machine Learning

*Alhazen* and our approach use machine learning teqchniques to associate the program's behavior with the features of the input data. Usually, machine learning approaches can be divided into three main categories:

- supervised learning,
- unsupervised learning,
- and reinforcement learning.

Supervised learning uses training data and their desired outputs to learn a general rule that maps inputs to outputs. For an unsupervised learning setting, the corresponding algorithm is given no outputs labels, and it has to identify the structural differences between the inputs on its own. Therefore, unsupervised learning is often used to detect clusters of data or hidden patterns. Finally, for reinforcement learning, the algorithm or machine learning model can interact with the surrounding environment, where it tries to maximize a particular goal. For the context of this thesis, we will be focusing on supervised learning.

Supervised learning algorithms build a mathematical model of training data containing the inputs and the desired outputs, also called labels. In the mathematical model, the training data is often represented as a vector of properties or features. These vectors are also known as *feature vectors*. One of the most common applications of supervised learning is in the context of classification. Through iterative optimization of an objective function, supervised machine learning algorithms learn a function that can predict the output of new inputs. An optimal function will allow the algorithm to correctly determine the output for inputs that were not part of the training data. An algorithm that improves the accuracy of its outputs or predictions over time has learned to perform that task. In the following, we will introduce three commonly used supervised learning algorithms:

### 3.2.1. Decision Tree

The Decision Tree classifier [SH77] is a white-box classification model that predicts a target variable (e.g., if an input is *bug-triggering* or *non-bug-triggering*) by inferring specific decision rules. As the name already suggests, a Decision Tree is a tree-like structure where an internal node represents a predicate $f \geq v$, with $f$ being the feature, the branch represents a decision rule, and each leaf node represents the prediction outcome. The topmost node in a Decision Tree is known as the root node. For instance, Figure 2 shows a learned Decision Tree on the passenger data of the Titanic along with the classification labels that state whether a passenger survived. The tree shows that female passengers and male children under the age of 10 had the highest chances

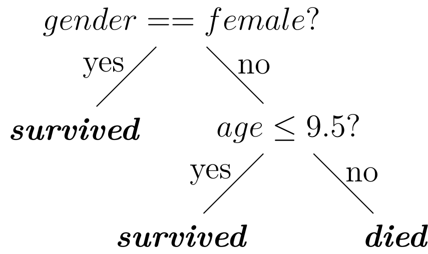



Figure 2: A Decision Tree (of height $h = 2$) showing the passengers' chances of survival on the Titanic. The highest chances of survival had passengers that were either (*i*) female or (*ii*) male with age less than 9.5.

of survival[1]. The tree learned to associate the passengers' survival according to the features *gender* and *age*. When a Decision Tree model classifies a sample $s$, it traverses its internal structure in the following way: Starting at the root node, the predicate in the node is checked against the (features of) the sample. If it is fulfilled, the yes branch is examined next; otherwise, the traversal continues at the no branch. As soon as the traversal reaches a leaf, the label of this leaf is the prediction.

In order to use such a Decision Tree for classification tasks, the trees need to learn how to partition a set of training data based on feature values. One of the most common techniques to grow a Decision Tree is to use the CART algorithm [BFOS84]. The main idea of the algorithm is to select input features and split points on those variables until a suitable tree is constructed. The selection of which input features to use and the specific split or cut-point is chosen using a greedy algorithm that minimizes a cost function. The tree construction ends with a predefined stopping criterion, for instance, a minimum number of training instances assigned to each leaf node.

The recursive partitioning of the CART algorithm and the tree-like structure make it easy for us to examine which features and predicates were used for the classification. Furthermore, the visualization is like a flowchart diagram that easily mimics human-level thinking. This is one of the main reasons why Decision Trees are often straightforward to understand and interpret. Resultingly, we consider the Decision Tree to be a white-box type of machine learning model. This is because it shares internal decision-making logic (i.e., the decision path traversal), which is often not available in black-box models such as Neural Networks. In addition, the Decision Tree is a distribution-free or non-parametric method, which does not depend upon probability distribution assumptions. Therefore, decision trees can handle high-dimensional data with reasonable accuracy.

### 3.2.2. Random Forest

Nonetheless, Decision Trees have a significant disadvantage: if they are grown too deep, they often cause overfitting to the training data, resulting in a high variation of

[1] Note that this is a simplified example, the actual data and the precise classification model also include the number of spouses or siblings aboard.

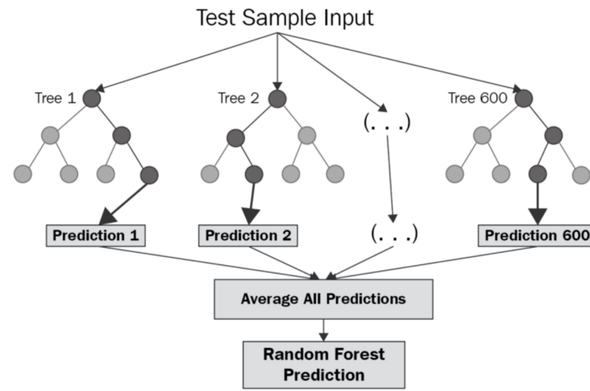



Figure 3: Overview of the Random Forest classifier [UKHM19] and how it uses a combination of many Decision Trees to derive a prediction for a sample input.

the classification outcome for a slight change in the input data. Moreover, they are susceptible to their training data, making them error-prone to most real-world datasets.

To increase the robustness of the machine learning model, we can use a technique called *bagging*, where we train many machine learning models (weak learners) in parallel. As a result, each model learns from a random subset of the data. One of the most notable applications for bagging is found in the Random Forest estimator [Bre01]. The algorithm is based on random subspaces and uses the CART Decision Trees as the base algorithm. Therefore, a Random Forest trains and operates a combination of different Decision Trees in parallel.

Figure 3 shows an illustration of the Random Forest algorithm. To classify a new sample $s$, the sample must pass down each Decision Tree of the Random Forest. Then, each tree considers a different part of the sample feature vector and gives a prediction outcome. Finally, the forest chooses the classification of having the most 'votes' (for discrete classification outcome) or the average of all trees in the forest (for numeric classification outcome). This procedure is often also referred to as majority voting. Since the Random Forest algorithm considers the outcomes of many different Decision Trees, it can reduce the variance resulting from considering a single tree for the same dataset. This means that while the predictions of a single tree are highly sensitive to noise in its training set, the average of many trees is not, as long as the trees are not correlated. Simply training many trees on a single training set would give strongly correlated trees (or even the same tree many times, if the training algorithm is deterministic); bagging is a way of de-correlating the trees by showing them different training sets.

### 3.2.3. Gradient Boosting Trees

Contrary to training many models in parallel, we can also reduce the bias of an individual Decision Tree by training many models *sequentially*. This technique is known as *boosting*, where we combine many weak learners into a single strong learner, and each model learns from the mistakes of the previous model. One of the most notable

applications for boosting is found in the Gradient Boosting Tree estimator [Fri01]. Gradient Boosting Trees represent a different kind of training algorithm. An ensemble of trees is incrementally built by training each new tree based on the data samples misclassified by the previous trees. For Random Forests, the final classification output is typically obtained through a voting mechanism (e.g., majority voting). Contrary, Gradient Boosting Trees use a binary reduction scheme, such as one-vs-all or one-vs-one, for multi-class classification. All the trees are connected in series, and each tree tries to minimize the error of the previous tree. Due to this sequential connection, boosting algorithms are usually slow to learn; however, they can also be highly accurate.

We train the individual Decision Trees such that each new learner fits into the residuals of the previous tree. The final model aggregates the result of each tree, and thus, a highly accurate and precise machine learning model is obtained. To detect the residuals, we use a loss function. For instance, the mean squared error (MSE) can be used for a regression task, and the logarithmic loss (log loss) can be used for classification tasks. Finally, it is worth noting that existing trees in the model do not change when a new tree is added because the added Decision Tree learns from the mistakes of the previous model.

One problem that we may encounter in Gradient Boosting Trees but not Random Forests is overfitting because of too many trees. In Random Forests, the addition of too many trees will most likely not cause overfitting. Consequently, the model's accuracy does not improve after a certain point; however, we mitigate the threat of overspecializing. On the other hand, for Gradient Boosting Trees, we have to be careful about the number of trees we select. By adding too many sequential Decision Trees, we run the risk of overfitting due to the overspecialization of the training data.

$\langle Expr \rangle \rightarrow \langle Term \rangle$ | $\langle Expr \rangle$ "+" $\langle Term \rangle$ | $\langle Expr \rangle$ "-" $\langle Term \rangle$ ;
$\langle Term \rangle \rightarrow \langle Factor \rangle$ | $\langle Term \rangle$ "/" $\langle Factor \rangle$ | $\langle Term \rangle$ "*" $\langle Factor \rangle$ ;
$\langle Factor \rangle \rightarrow$ "+" $\langle Factor \rangle$ | "-" $\langle Factor \rangle$ | "(" $\langle Expr \rangle$ ")" $\langle Factor \rangle$ | $\langle Int \rangle$;
$\langle Int \rangle \rightarrow \langle Digit \rangle$ | $\langle Digit \rangle \langle Int \rangle$;
$\langle Digit \rangle \rightarrow$ "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";

Figure 4: Context-Free Grammar $G$, which allows us to produce arithmetic expressions.

## 3.3. Context-Free Grammars

Context-free grammars are a well studied field of theoretical computer science, compiler design, and linguistics [HMU01]. Context-free grammars are used to describe programming languages and can be used to automatically generate parser-programs in compilers.

**Definition 7** (Context-Free Grammar)**.** A *context-free grammar* is a *4-tuple (N,T,P,s)*, where $N$ is the set of *non-terminals*, $T$ the set of *terminals*, $P$ the set of *productions rules* with $P : N \rightarrow (N \cup T)$, and $\langle s \rangle \in N$ the initial starting symbol. Production rules are used to expand a *non-terminal* $\langle S \rangle \in N$ to one of its $n$ alternatives $A_i$:

$$\langle S \rangle \rightarrow A_1 \mid A_2 \mid A_3 \mid ... \mid A_n \tag{1}$$

A tuple $(u, v) \in P$ can also be described using the binary relation $u \rightarrow v$, which is called a derivation. The most important short hand for derivations is $u \ \rightarrow^* v$ which signifies $u$ derives to $v$ using any amount of derivations. An expression $w$ is called a word of the Grammar $G$ if it is derivable via $s \ \rightarrow^* w$. A collection of words is called language.

To demonstrate the usage of a *context-free grammar*, lets assume a simple example grammar, that allows the production of arithmetic expressions. Figure 4 shows the production rules $P$ of grammar $G$, with the *non-terminals* $N = \{\langle Expr \rangle, \langle Term \rangle, \langle Factor \rangle, \langle Int \rangle, \langle Digit \rangle\}$, the *terminals* $T = \{$"0", "1", "2", .., "9", "+", "-", "*", "/", "(", ")"$\}$, and $S_0 = \{\langle Expr \rangle\}$ as the starting symbol. By following the specified rules, this grammar can now be used to generate arithmetic expressions or can be used to verify if an input is part of the language that the grammar portrays. Let us assume the word *3+6*. One possible derivation sequence of the grammar $G$ is: (1) $\langle Expr \rangle$ (2) concatenation of $\langle Expr \rangle$ "+" $\langle Term \rangle$ (3) $\langle Expr \rangle$ (4) $\langle Term \rangle$ (5) $\langle Factor \rangle$ (6) $\langle Int \rangle$ (7) $\langle Digit \rangle$ (8) "3" (9) "+" (10) $\langle Term \rangle$ (11) $\langle Factor \rangle$ (12) $\langle Int \rangle$ (13) $\langle Digit \rangle$ (14) "6". Therefore the word *3+6* is in the language of $G$.

The property of a word (or input) that can be described using only a context-free grammar is called a *syntactical feature*. For instance, for our grammar $G$, we can define the syntactical feature that states whether the *non-terminal* $\langle Factor \rangle$ was used to derive a word of the language. Therefore, the word *3+6* has that property because in step (11) of the derivation sequence, we used $\langle Factor \rangle$ to create the word.

## 3.4. Alhazen: Learning Circumstances of Software Behavior

Considering the difficulties of determining the circumstances of a program's behavior, Kampmann et al. [KHSZ20] presented an approach to automatically learn the associations between the failure of a program and the input data. Their proposed idea affiliates specific syntactical features of the input, like input length or presence of specific derivation sequences, with the behavior in question. This allows their tool *Alhazen* to form a hypothesis on why failure-inducing input files result in a defect.

Given a set of initial input files - at least one failure-inducing input file is required - and their execution outcomes, which determine whether the behavior in question is present or not, *Alhazen* parses the input files into its elements using a given input grammar. To extract and learn the properties and features of the input data, *Alhazen* uses a Decision Tree learner. The learner is used to capture the circumstances that distinguish the program's behavior. Hence, the Decision Tree learns associations between the program's behavior and the features of the individual inputs. In general, Decision Trees - in particular classification trees - can function as a predictive model to classify observable properties of input data to predict the targeted outcome. Kampmann et al. [KHSZ20] use a set of artificial features to capture whether a particular property in the input is present or not. These features are derived from the context-free grammar of the input language. The authors carefully used feature engineering - the process of using specific domain knowledge to extract features (characteristics, properties, attributes) from the raw input data - to extract the following purely *syntactical features* from the parse tree of the individual input files:

- *The presence or absence of non-terminal symbols (Existence)*
- *The length of individual nodes in the parse tree (Length)*
- *The code point of characters of a parse tree node (Maximal Code Point)*
- *The numeric representation of a parse tree node (Numeric Interpretation)*

According to the extracted features, the Decision Tree constructs a tree that explains and predicts whether the behavior in questions occurs. In particular, *Alhazen* forms a hypothesis based on the syntactical features and tries to generate a Decision Tree to distinguish between all observations so far. Since the initial hypothesis may still be optimizable, further inputs are necessary to refine the model. Hence, *Alhazen* uses the provided grammar as a means to produce more input samples. By extracting the predicates responsible, e.g., for the failure of a program, from the Decision Tree, *Alhazen* can use these new inputs specifications to generate similar inputs to verify - or falsify - the learned model.

The input generator starts by reducing and slicing the input grammar, excluding all production rules and alternatives not required by an input specification. In the next step, Kampmann et al. eliminate all specifications that are infeasible within the grammar. For instance, if the input specification contains existence features that contradict each other, the generator will not be able to produce such input. Consequently, the specification is deemed infeasible and is not considered. Due to the grammar ambiguity introduced with the grammar transformation, all possible parse trees - that derive the same word - need to be considered and checked if an input specification is eligible for removal. Finally, the input generator produces new candidates by constructing grammar derivation sequences that fulfill the input specification. If their tool cannot produce an input within two minutes, the specification is again considered infeasible and removed.

After the newly generated input files have been executed and labeled, the feedback loop start over with parsing and extracting the syntactical features of the new inputs. During each iteration of the feedback loop, *Alhazen* trains a new Decision Tree on all known samples and uses the learned predicates to generate additional input samples, to eventually produce a theory of the circumstances that resulted in the program's behavior.

As previously mentioned in Section 3.2, Decision Tree classifier are extremely prone to minor changes in the data set, resulting in completely different prediction outcomes. Due to the randomized nature of generating new inputs to refine the hypothesis, *Alhazen* could profit from more consistent and more precise machine learning techniques.

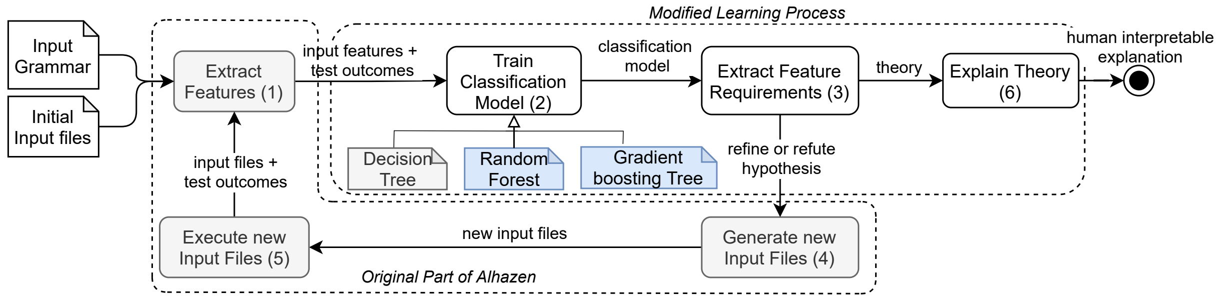



Figure 5: Overview of our approach.

# 4. Evaluating Software Behavior with Different Machine Learning Alternatives

In this Section, we will present our approach, a modified extension of the tool *Alhazen*, that allows us to predict and classify the circumstances of the program's behavior more precisely. The key idea behind our approach is to combine the already existing strategy with more advanced machine learning techniques, aiming to improve the overall prediction capabilities of *Alhazen*. In practice, Decision Trees are often considered *weak learners*, and other machine learning models might perform more favorably regarding accuracy and precision. Furthermore, we propose a comprehensive framework that allows developers to incorporate new classifiers and introduce an extended and modified learning process to cope with the challenges of deploying other machine learning methods. Thus, we claim that our approach may help developers and software engineers derive a better explanation for the root causes of the behavior in question, for instance, why an input file resulted in a program's crash.

Figure 5 gives an overview of the internal process structure and individual activities of our proposed extension. The activities that we did not modify (Activities 1, 4, and 5) and the Decision Tree Classifier from the initial approach are colored grey. Our contribution is the modified learning process with Activities 2, 3, and 6.

**Overview** Similar to the initial approach, our extension starts with extracting the initial features based on the user-provided grammar and the set of initial input samples. The grammar defines the input language of the subject under test (SUT) and allows us to associate syntactical grammar features with the program behavior. Then, with the help of the grammar, the initial input files are parsed according to a pre-defined set of features (**Activity 1**). Based on the extracted syntactical features and the test outcomes of the input files, we can deploy supervised learning methods to determine a mathematical model that maps input features to the test outcome (**Activity 2**). Since we are interested in explaining the circumstances of the observed behavior, we assign inputs to either (i) *behavior is present (bug-triggering)*, or (ii) *behavior is absent (benign)* class. The ultimate goal is to learn a general rule that maps the syntactical input features to the test outcomes. An optimal function will allow the machine learning

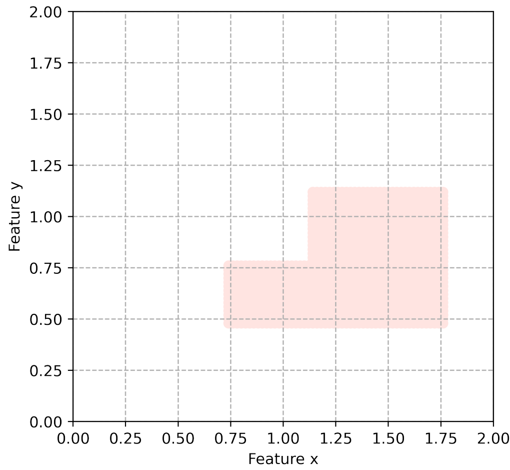
model to correctly determine the program behavior for an input file, which was not part of the training data. For our approach we propose the Random Forest classifier and the Gradient Boosting Tree, to learn the properties of the inputs that are responsible for the program behavior. However, to learn such an optimal function, a classifier needs to be trained on a sufficient amount of input files. Unfortunately, developers often have only a handful of input files that trigger the specific behavior. To counteract this problem, *Alhazen* uses the initially learned classification model to generate additional input files based on the learned input features. The goal is to refine or refute the initial hypothesis to explain the circumstances of the program's behavior. Thus, the features the classifier associates with the program behavior need to be extracted (**Activity 3**) and additional input files generated.

The input file instructions (specifications) for the new inputs are parsed to the input generator, which generates additional input files based on a given set of feature requirements (**Activity 4**). Then, the new input files are executed, and the corresponding test outcomes are obtained to verify if the new input files result in the program behavior in question (**Activity 5**). Finally, we can use the feedback and test outcome from the new inputs to start the learning and classification process again, iteratively refining and finetuning the hypothesis of the classifier, eventually deriving a mathematical model to predict the output associated with new inputs correctly.

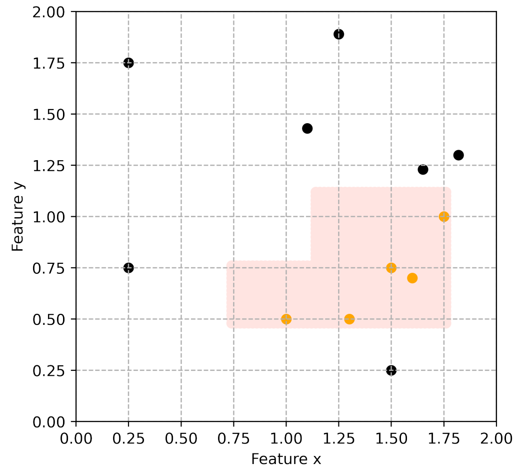
After a pre-defined stopping criterion is fulfilled, e.g., the number of iterations or time budget is exceeded, the last step of our approach is to derive a final explanation or theory on the circumstances of the behavior in question (**Activity 6**). In the initial approach, this step was indirectly included with the natural and easy interpretability of the Decision Trees and their good explainability to humans. However, more advanced machine learning approaches are not as easy to interpret and thus require a separate interpretation phase before the final theory on the circumstances of the behavior can be obtained.

**Feature Space** Before we explain the individual steps of our approach, we quickly highlight the construction of the feature space. After *Alhazen* parsed the input files and extracted the syntactical features, each input can be represented as a feature vector where each dimension represents one of the syntactical features in correspondence to the input grammar (Section 3.4). Thus, each syntactical valid input file corresponds to a point in an $n$-dimensional feature space. However, note that not all points in the feature space correspond to valid inputs. For instance, let us consider the custom calculator grammar from Kampmann et al. [KHSZ20]. An input file cannot have the two existence features <exists(⟨*function*⟩ == **`"sqrt"`**)> and <exists(⟨*function*⟩ == **`"tan"`**)> simultaneously equal to 1, since this would contradict the grammar. A point that lives in this feature plane does not belong to a valid input of the grammar.

To illustrate how our proposed machine learning models use the feature space for the refinement of their hypothesis, let us consider the following simplified example:




(a) Feature Space and Error Region




(b) Inputs in the Feature Space

Figure 6: 2-Dimensional Feature Space with Error Region. Dots within the error region are displayed in orange and are evaluated to *bug-triggering*; black dots outside the error region correspond the *benign* inputs.

Figure 6a shows a two-dimensional feature space. Additionally, it shows an error region inside the feature space. Therefore, every valid input (represented as a two-dimensional feature vector) that corresponds to a point in the error region will be evaluated to be *bug-triggering*. Figure 6b shows the feature space and a set of inputs projected into the plane. Inputs displayed in orange are evaluated to be *bug-triggering*, and the black dots correspond to *non-bug-triggering* (*benign*) inputs. The goal is to learn the associations between the features and the outcome and precisely predict whether an input will fail the program. The following sections will continuously extend this example and illustrate how the different models derive their predictions and how our approach refines its hypothesis.

In the following, we will describe each step and activity of our modified learning process in more detail.

## 4.1. Training new Classification Model

After the inputs and the corresponding feature vectors are obtained, the next step is to learn the features and predicates responsible for the program behavior. Given the set of feature vectors and their test outcome labels, the problem of discriminating learning becomes a standard classification problem. Subsequently, *Alhazen* and our approach use this information to localize regions in the feature space responsible for the behavior in question.

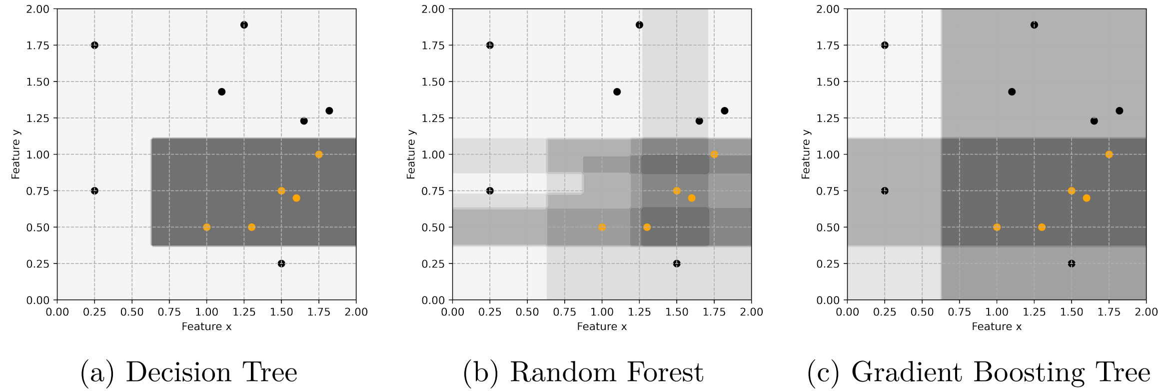
This step is illustrated in Figure 5 as *Activity 2* and marks the beginning of our modified learning process. Initially, Kampmann et al. [KHSZ20] proposed a Decision Tree Model to learn the features associated with the program's behavior. Unfortunately, although Decision Trees have an advantage regarding interpretability, they often can not compete with other so-called *strong learners* regarding accuracy and precision. Moreover, Decision Trees tend to overfit, resulting in poorer performance on the "real-world data" than on the training data. To improve the classification, we replace the Decision Tree learner and extend *Alhazen* with two more advanced machine learning techniques to mitigate overfitting and increase accuracy and precision: (*i*) *Random Forests* and (*ii*) *Gradient Boosting Trees*. Both classifiers are an ensemble learning method operating by constructing a multitude trees at training time.

**Decision Tree** Before we talk about the Random Forest and Gradient Boosting Trees, let us think about Decision Trees and what they are doing to the syntactical *feature space*. Each feature is a dimension in the feature space, and each input is a feature vector. A Decision Tree recursively splits up the inputs (points in feature space) based on one feature at a time. So a Decision Tree essentially draws dividing lines in the dimensions of feature space and recursively subdivides along other dimensions, allowing it to separate the *bug-triggering* from the *benign* inputs. The Decision Tree eventually constructs instructions to explain why a set of inputs (associated with the syntactical features) results in the program's behavior.

**Random Forest** We selected Random Forests due to their tendency to correct Decision Trees' habit of overfitting to their training data by constructing many individual Decision Trees at training time. In addition, because we want to distinguish between two classes (benign and bug-triggering), the Random Forests output is the class predicted by most trees, also known as majority voting.

Random Forests are among the best performing Machine Learning algorithms and have seen wide adoption to many domains and applications [EKG+14]. Although they are harder to interpret than a single Decision Tree, they bring many advantages, such as improved performance (wisdom of the crowds) and better generalization.

Similar to the single Decision Tree, a Random Forest also recursively subdivides the feature space dimensions; however, by operating on multiple trees and the procedure of how a final decision is obtained, each decision boundary is not as sharp. Consequently, to predict an input to be, for instance, *bug-triggering*, at least half of the Decision Trees




(a) Decision Tree (b) Random Forest (c) Gradient Boosting Tree

Figure 7: Decision boundaries of three different machine learning methods. Dots displayed in orange are evaluated to *bug-triggering*; black dots correspond the *benign* inputs. The darker the region of the feature space, the more likely a point in the feature space will be predicted as *bug-triggering*.

need to come to the same conclusion. This is the case if the predicted regions of most trees in the feature space overlap.

**Gradient Boosting Trees** Additionally, we selected Gradient Boosting Trees to determine the features responsible for the shown behavior. They have recently shown great success for many different classification and regression tasks and produce state-of-the-art results for many commercial applications [BGMP21].

As a reminder, boosting is an ensemble technique, which means that an ensemble of simpler estimators (*weak learners*) makes a prediction. While this theoretical framework makes it possible to create an ensemble of various estimators, in practice, we almost always use Gradient Boosting Trees over Decision Trees. This is also the combination that we selected for our approach. Thus, similar to the Random Forests, Gradient Boosting Trees are an ensemble method consisting of multiple Decision Trees. Gradient Boosting Trees aim to train an ensemble of trees, given that we know how to train a single Decision Tree. This technique is called boosting because we expect an ensemble to work much better than a single estimator (The same is true for the Random Forest).

**Example** To illustrate the differences of the individual approaches, let us again continue the example introduced at the beginning Section 4. Figure 7 shows the decision boundaries of three different machine learning models, namely for the Decision Tree, Random Forest and the Gradient Boosting Tree, for our two-dimensional feature space. Again, inputs that trigger the behavior in question are colored orange; otherwise, they are displayed in black. Each method tries to learn the set of predicates $P$ responsible for the observed outcome, for instance, the occurrence of a bug. To highlight the differences of the classifiers, we display the decision boundaries and color the predicted regions of the feature space according to the likelihood of an input being evaluated to either *benign* or *bug-triggering*. Hence, the darker a region in the feature space, the more likely

a point is predicted to be *bug-triggering*.

We observe that the Decision Tree (Figure 7a) strictly divides the feature space into two classes (noticeable because there are only two different shaded areas). This reflects the nature of Decision Trees and their hyperrectangular cuts along the dimensions. Simultaneously, this allows Decision Trees to be easily interpreted, with clear instructions (See Section 3) on how to determine the regions of interest. Unfortunately, this is also the reason why Decision Trees tend to overfit based on the training data.

Contrary to the Decision Tree, we see that the Random Forest (Figure 7b) divides the space into many differently shaded areas. This is because each decision boundary corresponds to one of the Decision Trees inside the Random Forest. Thus, the more Decision Trees overlap and predict the same area to be bug-triggering, the darker the region is displayed. This perfectly shows how Random Forests derive their predictions - majority voting of the individual estimators (*weak learners*).

We can make similar observations for the Gradient Boosting Tree (Figure 7c), which divides the feature space into different prediction areas, with some being more likely to contain bug-triggering inputs than others. However, as described in Section 3, how a prediction is derived is fundamentally different compared to Random Forests, even though they both are ensemble estimators of multiple Decision Trees. Moreover, since it is difficult to interpret the underlying loss function, understanding what predicates resulted in the Gradient Boosting Tree's prediction is often not possible. While boosting can increase the accuracy compared to single Decision Trees, it sacrifices comprehensibility and interpretability.

## 4.2. Extracting new Feature Requirements

As seen in the example (Figure 7), the initial performance regarding the approximation of the error region of all classifiers is far from being satisfactory. However, to derive a precise explanation of the circumstances of the program's failure, we need to know what predicates are responsible for the observed behavior. Unfortunately, the main limitation of many machine learning models is an insufficient training data set; thus, the classifiers can only derive a limited hypothesis about the error regions they try to explain. More precisely, an explanation and hypothesis of why a specific bug occurs will often not be sufficient based on the initial inputs. Consequently, we need to generate additional input data to improve our initial hypothesis. However, before we can produce new inputs, we need to extract the predicates of the hypothesis to guide the input generation process as efficiently as possible. Since this procedure is directly tied to the previously learned machine learning model, this activity is part of our modified learning process, displayed as *Activity 3* in Figure 5.

Generating additional inputs can be done in two fashions: ($i$) randomly generating input or ($ii$) incorporating the classifier's previous predictions to refine the decision boundaries efficiently. Generating new inputs by randomly adding points (represented as feature vectors) to the feature space is a cost-effective method to improve the classifier's performance. However, as mentioned at the beginning of Section 4, not every point in the feature space represents a syntactically valid input defined by the input grammar. Therefore, the risk of creating feature vectors that do not belong to the grammar's search space is very high. Furthermore, due to the high dimensionality of the feature space and the fact that the error region usually only occupies a small area of the entire feature space, the chances of randomly generating a bug-triggering input are extremely small.

Thus, to effectively guide the generation of new inputs and improve the classifier's predictions, Kampmann et al. [KHSZ20] generate additional data close to the decision boundaries of the learned Decision Tree. They achieve this by following the individual paths of the Decision Tree and systematically negating individual features. Then, the resulting predicates are used as instructions to generate new inputs. In the following, we call these instructions requirements $r$, and a set of predicate requirements forms a specification $s$ of a new input file. Since we treat both Random Forests and Gradient Boosting Trees as white-box models, we adapted the algorithm of Kampmann et al. [KHSZ20] to extract these requirements from the classifiers and generate additional inputs that the learner deems to result in *benign* or *bug-triggering* behavior.

Algorithm 1 shows the pseudocode of our requirement extraction algorithm and displays how we utilize the structure of the individual trees to extract new instructions for each complete tree path. The algorithm requires the structural representation of the ensemble tree-based estimators as input to extract new requirements. The output is a set of input specifications $s$ to generate additional samples. Since Random Forests and Gradient Boosting Trees are composed of individual Decision Trees, we start our

**Algorithm 1:** Extract Requirements

**Input:** A ensemble tree-based estimator $T$

**Output:** A set of input specifications $S$ to generate new inputs

```
1 requirementSpecifications ← ∅
2 foreach decisionTree ∈ T do
3     decisionTreePaths ← getAllPathsForDecisionTree(decisionTree)
4     decisionTreeRequirementSpecification ←
        getAllCombinations(decisionTreePaths)
5     requirementSpecifications ←
        requirementSpecifications ∪ decisionTreeRequirementSpecification
6 end
  // Returns the final list all requirements specifications
7 return requirementSpecifications
```

algorithm by iterating over all individual Decision Trees (Algorithm 1 Line 2-6). For each Decision Tree, we obtain all complete paths (root to leaf) with the function *getAllPathsForDecisionTree*(*decisionTree*) (Line 3). The function uses a depth-first approach, and once a leaf node is reached, it returns the complete path of predicates. Since in our domain (binary classification), Decision Trees are binary trees, the total number of paths from the root to leaf of a complete tree $t$ with height $h$ is the number of leaves that is: $2^h$. After extracting all Decision Tree paths, we use the same idea as Kampmann et al. and calculate all subsets of features and negate them. Since this reduces to calculating the powerset of predicates on a path $p$ of length $h$, the number of possible new input specifications per path is exponential to the length of each path (*getAllCombinations*(), Line 4). More precisely, for each path of length $h$, we extract $2^h$ (complexity of the powerset) new input specifications.

Resultingly, the maximum number of new input specifications $s$ of an estimator $e$, given $n$ trees of height $h$, is limited by the upper bound:

$$\begin{aligned} \textit{numberOfSpecifications(e)} &\leq n \times 2^h \times 2^h \\ &\leq 2^{2h} n \end{aligned} \tag{2}$$

Thus, we can expect that for a fixed height $h$, the ensemble estimators will generate considerably more inputs than the Decision Tree, dependent on the number of constructed trees $n$. Note that we use all paths in the trees to generate additional inputs, not only those expected to be *bug-triggering*. This follows the idea that *benign* inputs act as a counterpart to the *bug-triggering* samples. Without them, the estimators would determine the failure area too broad, resulting in an underfitting of the error region.

Although we use the same algorithm to extract and eventually generate new inputs for both Random Forest and Gradient Boosting Tree, the conceptual changes regarding the improvement of the classifiers are fundamentally different. This is strongly connected to the differences in the construction and operation of both estimators.

**Random Forest:** For the Random Forest, this extraction algorithm follows the idea that we want to improve and sharpen the decision boundaries of each Decision Tree. Therefore, each Decision Tree obtains more training data and can adjust its hypothesis. With the majority voting procedure, the combined individual improvements of the Decision Trees will result in an overall improved estimator.

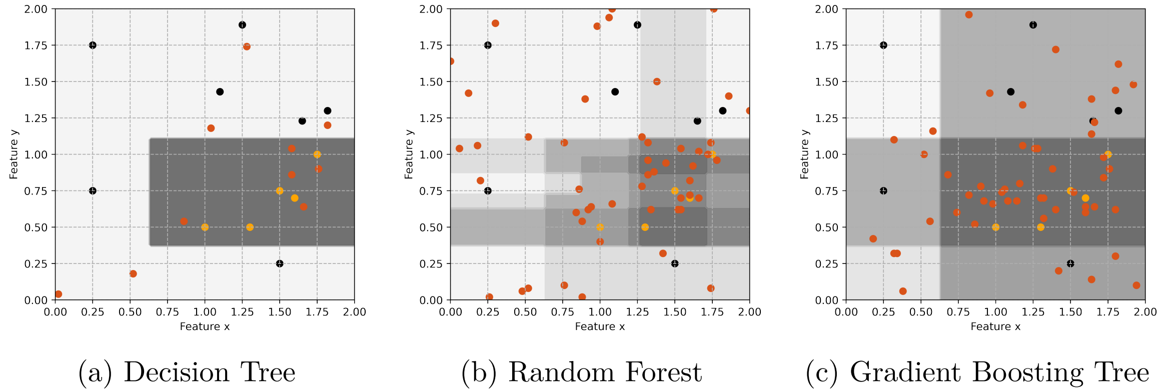
**Gradient Boosting Tree:** In contrast, we cannot apply the same concept of improvement to the Gradient Boosting Tree. Because the Gradient Boosting Tree is built in a stage-wise fashion, the Decision Trees are not independent. More precisely, as in other boosting variants, each Decision Tree attempts to correct the errors of its predecessor. Thus, by generating more inputs for each tree, we are not optimizing each weak learner. However, the algorithm allows us to extract and generate new inputs effectively; and with the availability of additional inputs, we expect the Gradient Boosting Tree to refine and improve its hypothesis by minimizing a given loss function.

## 4.3. Continuing the Feedback Loop

**Activity 4: Generating new Inputs** After extracting the requirements for the additional inputs, we parse the input specifications to the input generator (Figure 5, *Activity 4*). We rely on the input generator proposed by Kampmann et al. [KHSZ20] to produce the new candidates for this activity. The additional input specifications for the Random Forest and the Gradient Boosting Tree increase the computational overhead of the input generator. Theoretically, the time needed to produce new inputs increases linearly with the number of individual Decision Trees of a fixed height[2]. However, this is only an upper bound, as the generator utilizes several strategies to reduce the computational overhead. For instance, if a produced input already covers an input specification, the generator considers this specification as satisfied and will not produce an additional input. As many trees learn and identify similar predicates, the actual computational overhead is lower.

**Example** Continuing with the learned classifiers from our example and using our proposed algorithm to extract the requirements for the new input specifications, we can use the input generator of *Alhazen* to produce additional inputs. Figure 8 (Page 28) shows the previously learned models (along with their previously learned decision boundaries) and the newly produced inputs (displayed in red). We observe that we generate, as expected, much more inputs for the Random Forest and Gradient Boosting Tree than for the Decision Tree. This is due to the increased number of individual trees

[2]If the height of the trees is not fixed, the number of input specifications grows exponentially



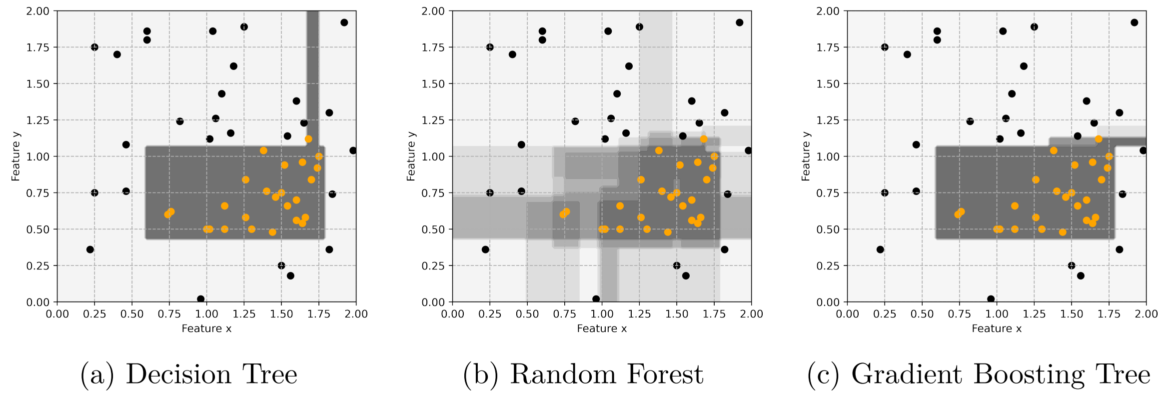

(a) Decision Tree (b) Random Forest (c) Gradient Boosting Tree

Figure 8: Decision boundaries of the previously learned models and newly generated input data (displayed in red) after the feature extraction.

in the ensemble estimators and the resulting increased number of input specifications. Although more training data is necessary to refine the models, producing new inputs and learning from them is expansive. In particular, if the produced inputs are similar to the already existing data, or the inputs cannot be generated (Section 3.4), which means we are essentially wasting resources, more inputs may not necessarily account for an improved model.

**Activity 5: Execute new Input Files** Before we can use the newly generated input files to refine the hypothesis, we need to execute the input files and obtain labels that classify whether the input files trigger the behavior (Figure 5, *Activity 5*). In particular, we are interested in the test outcomes and whether we were able to generate new input files that result in the behavior in question. Resultingly, these input files allow us to mark and narrow down the error boundaries in the inducted feature space. When executing the input files, we treat the program under test as a black box and only observe the output, for instance, if the input files crashed the program, thus making this process extremely efficient. However, this also indicates the necessity of an oracle that accurately tells us the outcome of the executed input file. For many cases, this oracle is relatively simple since it reduces to observing if the program crashed or not. Nonetheless, there are also scenarios where we cannot introduce a reliable oracle that states whether the behavior in question occurred. For instance, if we want to determine why an input requires an unusual amount of execution time, the oracle needs to account for external influences and fluctuating execution times for the same input. Consequently, this problem is directly tied to the oracle problem [BHM+15], which is related to the controllability and observability of a behavior. We further discuss the importance of a reliable oracle in Section 7.

After the input files have been executed and labeled, the process starts over with the feature extraction of the newly generated input files (Figure 5, *Activity 1*). The feedback loop continues until a pre-defined stopping criterion is fulfilled. For instance,




(a) Decision Tree (b) Random Forest (c) Gradient Boosting Tree

Figure 9: Refined decision boundaries of the three machine learning models after the generation, execution, and training of the additional inputs files.

this could be the exhaustion of the time budget, the number of iterations or no more estimator changes compared to the previous iteration.

**Example** Finally, to complete our running example, Figure 9 shows the refined decision boundaries and the performance of the classifiers. After we executed the input files and obtained the corresponding outcomes and labels, we used the additional inputs to retrain the machine learning models. As a result, we observe that all models could, as expected, derive a better approximation of the error region compared to the initial setting. We now repeat this feedback loop and iteratively refine the decision boundaries. With this adapted process, we can use the ensemble estimators to derive a better approximation of the error region, resulting in a better understanding of the predicates responsible for the program behavior.

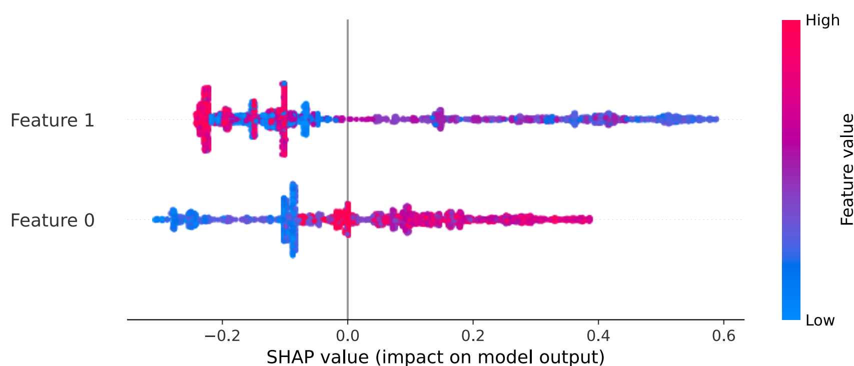


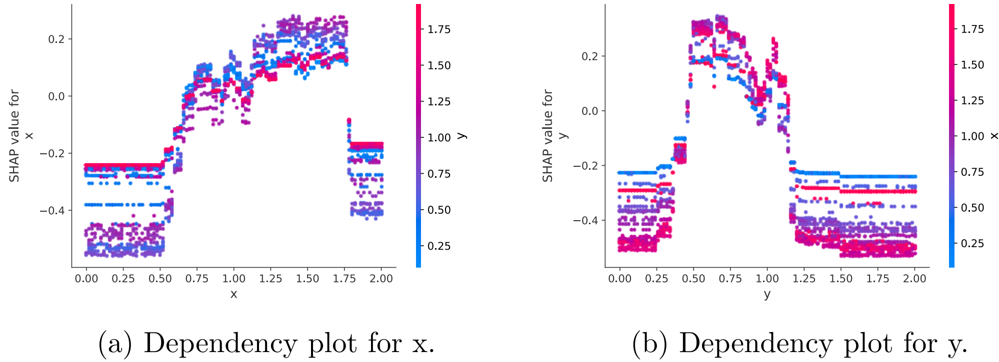

Figure 10: TreeSHAP [LEC+20] summary plot for the prediction of the previously learned and refined Random Forest classifier. The plot shows the importance of the features and their contribution to the final prediction. The color represents the value of the feature from low to high. Dots with a higher SHAP value have a higher contribution to the final prediction than dots with a low SHAP value.

## 4.4. Explaining the Prediction Theory

In the previous Sections, we have seen how we can use more advanced machine learning estimators to classify and predict the program's behavior. However, one final step is still missing: the explanation why the programs fail. Contrary to the initial approach, we use machine learning models that are not as easy to interpret as the Decision Tree - a straightforward interpretation is one of the main advantages of the Decision Tree in the first place. To help developers improve their software's reliability, we need to know what decisions of the machine learning algorithm lead to the predicted outcome. As previously mentioned and seen in our minimal example, it is often unclear what exact predicates caused the prediction of the Random Forest or the Gradient Boosting Tree. Therefore, the final step of our approach is devoted to explaining why the input triggers the program's behavior (Figure 5 Activity 6). With this additional step, we want to highlight that developers need to invest additional efforts before they can derive a reasonable answer for the program's behavior.

Explaining why a model made specific decisions is also extremely important for many other areas, particularly for medicine [RDK19], finance [JPC19], or autonomous driving [HUAM+19], where it is crucial to understand and interpret the choices of the machine learning model. Consequently, explainable machine learning is the subject of recent research [SMV+19, GSC+19].

One such method to explain the decisions of machine learning approaches is the tool called SHAP by Lundberg and Lee [LL17]. SHAP is a game-theoretic approach to explain the output of a machine learning model. To explain the decisions of a model, it computes Shapley values [Sha16] from coalitional game theory. The goal of SHAP is to explain the decision by computing the contribution of each feature to the final prediction. A SHAP value for a feature of a specific prediction represents how much the model prediction changes when we observe that feature. Recently, Lundberg et al.




(a) Dependency plot for x. (b) Dependency plot for y.

Figure 11: Dependency plots for the prediction of the previously learned and refined Random Forest classifier, visualizing the feature interaction.

[LEC+20] also proposed TreeSHAP, a faster variant of SHAP for tree-based machine learning models such as Decision Trees, Random Forests, and Gradient Boosting Trees. Thus, TreeShap can analyze our learned models and may help developers interpret the circumstances of the programs' behavior.

To complete our running example and to exemplarily show how TreeSHAP can be used, we try to explain the predictions made by the Random Forest from our example classification problem. Note that our example is minimal, and SHAP may work differently on real-world data. Particularly for high dimensional data.

The summary plot shown in Figure 10 (Page 30) combines feature importance with feature effects. Each dot on the summary plot is a Shapley value for a feature and an instance (input file). The position on the y-axis is determined by the feature and on the x-axis by the Shapley value. Note that when dots do not fit together on the line, they pile up vertically to show density. The color represents the value of the feature from low to high, in our case from 0 to 2. The displayed features 0 and 1 correspond to the "x-feature" and "y-feature", respectively. The plot shows the relationship between the value of a feature and its impact on the prediction. We see that the Random Forest predicts inputs to be *bug-triggering* if they have a relatively high x-value and a relatively low y-value. Contrary, a low x-value or a high y-value did not have a high contribution to the *bug-triggering* prediction. If we compare the assumption to the error region shown in Figure 6a (Page 21), we can confirm this initial observation.

However, we have to look at SHAP dependence plots to further confirm this first indication of the relationship between the values. Figure 11 shows the feature dependence plots with the feature interaction visualization. In these plots, each dot represents a single prediction for an input; the x-axis is the value of the feature, and the y-axis is the SHAP value for that feature. The color corresponds to a second feature that may have an interaction effect. We generated two plots, one for the x-value with the interaction of y-value (Figure 11a) and one for the y-value with the interaction of the

x-value (Figure 11b). As expected, we observe a high correlation of the features. We see that the Random Forest can derive a good explanation and approximation of the error region. Inputs that have an x-value between 0.75 and 1.75 and a relatively low y-value of about 0.5 to 1.25 are predicted to be *bug-triggering*

This extremely limited example already shows that developers need to invest serious efforts to derive an explanation for their program behavior. For accurate behavior classification, developers need to evaluate thousands of syntactical features. However, this is also true for the initial approach. In their evaluation, Kampmann et al. mitigate this threat by learning Decision Trees of height five. Thus *Alhazen* is limited to a maximum of five features (per tree path) to distinguish between *bug-triggering* and *non-bug-triggering* inputs. This is similar to only considering the top five features that have the highest impact on the prediction outcome.

**Summary** In this Section, we proposed an extended version of the tool *Alhazen*. We used the existing strategy of refining a hypothesis and applied this technique to more advanced machine learning models to improve the prediction of program behavior. We showed how we use the individual weak learners of the Random Forests and Gradient Boosting Trees to extract meaningful and effective input specifications. These specifications are then used to produce new inputs that may help the machine learning model refine or refute its theory on why a particular behavior is observed. However, better machine learning alternatives do not come without a price. What they might gain in accuracy and precision, they lose in comprehensibility and interpretability. On a small example, we showed how the tool TreeSHAP could be used to explain the decisions of the ensemble estimators. In the following Sections, we will evaluate our approach on ten-real world bugs and examine if our proposed machine learning methods can improve the bug prediction effectively.

# 5. Experimental Setup

In this Section, we evaluate the effectiveness of our approach by performing experiments on ten real-world bugs and applications. We have implemented our approach in the tool AlhazenML[3], and we compare our comprehensive framework to the initial approach of Kampmann et al. [KHSZ20], which will serve as the baseline. As proposed in the introduction (Section 1), we ask the following research questions:

**RQ1** Does AlhazenML allow us to **predict** the behavior of a program more precisely?

**RQ2** Does AlhazenML allow us to **produce** more defect triggering inputs efficiently?

To answer these research questions, we compare the baseline to the two configurations $\text{AlhazenML}_{RF}$ and $\text{AlhazenML}_{GBT}$, which use the Random Forest and the Gradient Boosting Tree to learn the association between the grammar features and the program behavior, respectively:

| Approach | Machine Learning Model |
|---|---|
| *Alhazen* | Decision Tree |
| $\text{AlhazenML}_{RF}$ | Random Forest |
| $\text{AlhazenML}_{GBT}$ | Gradient Boosting Tree |

By assessing the quality of the different approaches both as predictors (**RQ1**) and producers (**RQ2**), we ensure that they neither overspecialize nor overgeneralize.

**Technical Setup** To give all approaches similar starting conditions, we use *Alhazen* as initially proposed, implemented with a single Decision Tree, a maximum height of 5, and CART [BFOS84] as the construction algorithm. For the tree-based ensemble estimators instantiation of AlhazenML, we used a maximum number of 10 trees, again with a maximum height of 5. The reason for the relatively low number of trees is based on the exponential amount of input specifications, presented in Section 4.2. More trees were not feasible in our experimentations because the input generator could not finish within a reasonable time (More than one hour for one iteration).

**Subjects** In order to examine the effectiveness of AlhazenML, we evaluate our tool on the same test subjects and bugs (program behavior) that Kampmann et al. have originally covered with their proposed approach. These test subjects require two,

[3]We make our implementations and raw experiment data available for replication at `https://gitlab.informatik.hu-berlin.de/eberlema/master-thesis`

| Bug ID | Predicate of Interest | Bug ID | Predicate of Interest |
|---|---|---|---|
| calculator.1 | error message | closure.3178 | exception |
| closure.1978 | exception | closure.3379 | exception |
| closure.2808 | exception | rhino.385 | exception |
| closure.2842 | exception | rhino.386 | exception |
| closure.2937 | exception | genson.120 | exception |

Table 1: Subjects and Predicates of Interest

in complexity varying input formats, namely JSON and JavaScript. Rhino [Rhi18] and Closure [Clo19] serve as the JavaScript subjects, whereas Genson [Gen17] serves as the JSON subject. Table 1 lists the subjects and bugs for our evaluation. All test subjects are widely used in browsers and web applications. A further description of all subjects, along with their grammars, can be found in the initial work of Kampmann et al. However, in their paper, they use two more subjects, namely *grep* and *find*, which we were not able to evaluate. These subjects are part of the dbgbench benchmark [BSC+17]. They require unique execution environments, which we were able to put up; however, we did not have the computational resources to execute the evaluation (with the anticipated number of repetitions per approach) in a reasonable time, which is why we had to neglect them.

**Data Sets** Similar to the evaluation of Kampmann et al., we use the same method introduced by Soremekun et al. [SPH+20] to generate our evaluation data sets. Their grammar-based generator uses a probabilistic grammar, in which probabilities are assigned to choices of production rules. The distribution of these probabilities is learned from a sample of inputs. Consequently, test inputs produced by a learned probabilistic grammar are similar to the sampled inputs. Soremekun et al. call this idea "more of the same" [SPH+20] because the produced inputs share the same features as the sampled inputs. We used their approach to generate 1000 unique bug-triggering samples. We stopped with a smaller number of samples if 20 re-runs could not generate enough bug-triggering samples or a timeout of 1 hour was exhausted. Table 2 reports the sample data used to evaluate the approaches. We see that the number of *non-bug-triggering* and *bug-triggering* inputs differ noticeably. In particular, for the subject *closure2842*, we were only able to generate 177 bug-triggering inputs. Contrary to the evaluation of the initial work, we evaluate the performance of the approaches with an initial input corpus of just two inputs. This design decision follows the idea that we want to know if our modified learning process can generate meaningful additional data and thus can improve its accuracy and precision on its own. The two initial inputs, one *benign* and one *bug-triggering*, are provided by the authors of *Alhazen*.

| Subjects | *Benign* | *Bug-Triggering* |
|---|---|---|
| calculator | 3809 | 1245 |
| genson120 | 1864 | 1242 |
| rhino385 | 6088 | 1103 |
| rhino386 | 4003 | 1112 |
| closure1978 | 3140 | 1128 |
| closure2808 | 4043 | 1157 |
| closure2842 | 6222 | 177 |
| closure2937 | 4058 | 1147 |
| closure3178 | 3078 | 1048 |
| closure3379 | 3944 | 1218 |

Table 2: Number of *bug-triggering* and *non-bug-triggering* (benign) inputs.

**Measures** To quantify the performance of the approaches, we use the statistical measures *accuracy*, *precision*, and the *F1 score*. Informally, accuracy is the fraction of predictions the classifier successfully predicted. Formally, accuracy has the following definition:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3}$$

where TP = True positive; FP = False positive; TN = True negative; FN = False negative. However, accuracy alone does not tell the whole story. Precision is the fraction of correct predictions among all the retrieved predictions and tries to answer what proportion of positive identifications was actually correct. Formally, precision is defined as follows:

$$Precision = \frac{TP}{TP + FP} \tag{4}$$

Since we are working with a class-imbalanced data set (see Table 2), there is a significant disparity between positive and negative labels. To account for the subtleties of class imbalances, we measure the F1 score, which is the harmonic mean of the precision and recall. Thus, the relative contribution of precision and recall to the F1 score are equal. The F1 score reaches its best value at 1 and worst score at 0. The formula for the F1 score is:

$$F1\ score = \frac{2 \times TP}{2 \times TP + FP + FN} \tag{5}$$

**Research Protocol** Because of the partially non-deterministic learning process and the random nature of the input generation, we cannot rely on a single evaluation run. Consequently, we repeated the experiments ten times per subject and approach. To answer **RQ1**, we proceeded as follows: $(i)$ First, we generated the evaluation data

sets. (*ii*) Then, we started each approach for each subject with two initial inputs and performed at most 40 iterations of the learning and refinement process. We stopped if we did not generate new inputs in an iteration or the approach could not finish the 40 iterations within 3 hours. (*iii*) Finally, we measure the performance of the final iteration of each approach.

For **RQ2**, we proceeded as follows: (*i*) First, we obtained the final machine learning models (Decision Tree, Random Forest, and Gradient Boosting Tree) from *RQ1* and then (*ii*) used them to produce new *bug-triggering* and *non-bug-triggering* inputs. (*iii*) Finally, we measure whether the machine learning models' prediction matches the actual program behavior.

**Computational Setup** The experiments were run concurrently on a *Dell R920* compute server with four *Intel Xeon E7-4880* v2 processors (60 cores), 2.5GHz, 1024GB of system memory, and a Suse Leap 15 OS.

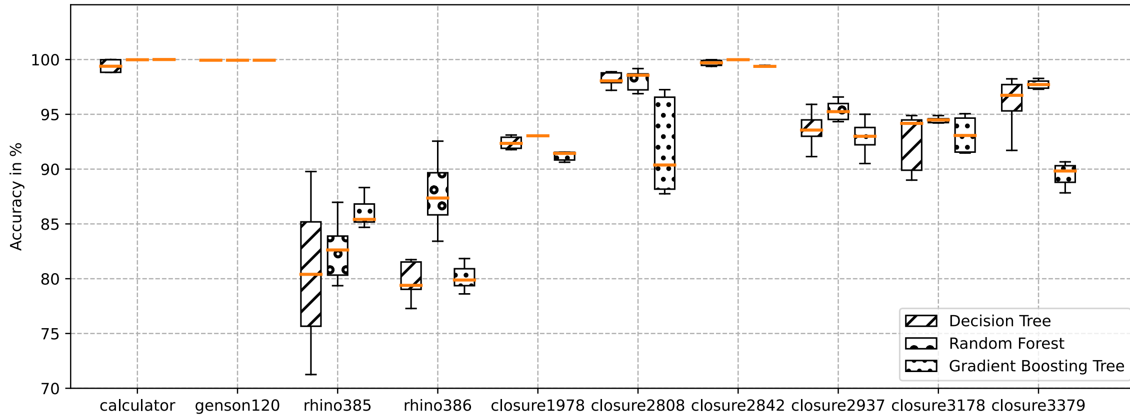



Figure 12: Achieved **accuracy** of *Alhazen*, AlhazenML$_{RF}$, and AlhazenML$_{GBT}$ for each subject over ten runs.

# 6. Evaluation

In this Section, we present the results of our experiments and evaluate the performance of each approach. To answer **RQ1** and **RQ2**, we examine whether the different approaches and configurations can precisely **predict** the behavior of a program and can be used to **produce** new failure-inducing inputs efficiently. Finally, we discuss the threats to validity.

## 6.1. AlhazenML as Predictor

This Section evaluates whether AlhazenML can accurately and reliably predict the behavior of a program (**RQ1**). With this assessment, we ensure that the different approaches do not overspecialize.

### 6.1.1. Experimental Results

Figure 12 through 14 show the obtained statistical measures for the ten subjects and ten repetitions. The boxplot is a standardized way of displaying the distribution of data and gives us a good indication of how the values in the data are spread out. Each boxplot shows the achieved performance of the three approaches - the baseline, AlhazenML$_{RF}$, and AlhazenML$_{GBT}$ - regarding their *accuracy*, *precision*, and *F1 scores*. The vertical axis represents each approach's achieved performance (i.e., statistical measure) in percent, and the horizontal axis displays the corresponding subject. The median measurements are displayed by the horizontal lines inside the boxplots and are highlighted in orange. To answer our **RQ1**, we compare the performance achieved by the three approaches. In particular, we investigate whether AlhazenML reaches at least the same percentage regarding the accuracy, precision, and F1 score as the baseline.

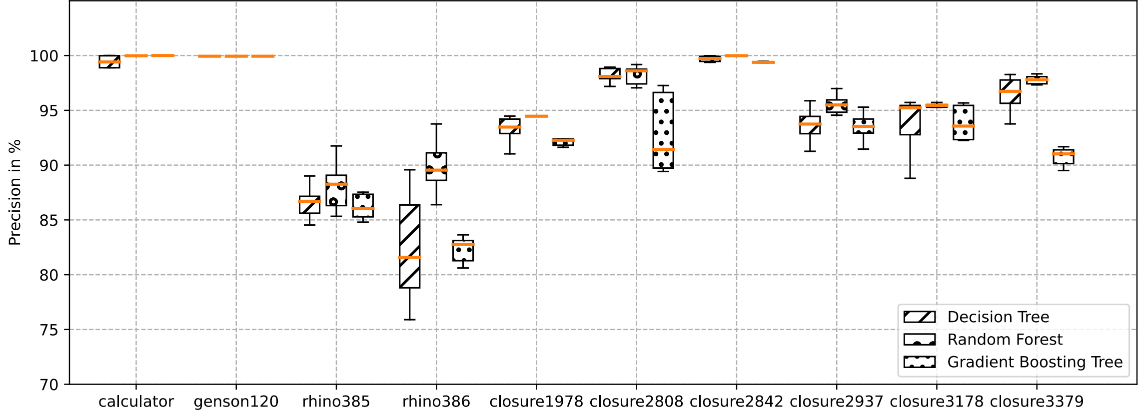



Figure 13: **Precision** of *Alhazen*, AlhazenML$_{RF}$, and AlhazenML$_{GBT}$ for each subject over ten runs.

**Accuracy:** Figure 12 displays the achieved accuracy of the approaches for each subject. We observe that AlhazenML$_{RF}$ reaches a higher median accuracy than the baseline for all subjects. For instance, our configuration with the Random Forest gained almost 100% accuracy for the custom *calculator* subject, whereas the baseline cannot reliably reach the full classification potential. Most notably, the Random Forest outperforms the baseline for the Rhino subjects *rhino385* and *rhino386*, with an increase in accuracy of 3% and 6%, respectively. Furthermore, the same trend can be observed for the *Closure*-related subjects, although the median increase is not as prominent. However, in return, the interquartile range (50% of all data) of AlhazenML$_{RF}$ is often notably smaller than for the baseline. This indicates that our approach achieved good results more reliably than the baseline over the ten runs.

Regarding accuracy, we observe that our approach AlhazenML$_{GBT}$, configured with the Gradient Boosting Tree, cannot compete with the other two approaches. On the contrary, the Gradient Boosting Tree often performs even worse than the baseline. For instance, for the subjects *closure2808* and *closure3379*, the approach performs remarkably worse than the other two configurations and does not accurately predict the program behavior. Only for the Rhino-related subject *rhino385* can it achieve the highest accuracy among all approaches.

**Precision:** Figure 13 shows the achieved precision of all three approaches. Similar to the reached accuracy, we observe that AlhazenML$_{RF}$ outperforms the baseline and the Gradient Boosting Tree regarding the achieved median precision. In addition, we can see a clear improvement for the subjets *rhino386* and *closure2937* compared to the other two configurations. Furthermore, AlhazenML$_{RF}$ achieves a noticeable smaller standard deviation for the remaining subjects and bugs (e.g., *closure1978* and *closure3379*). While the configuration with the Random Forest compares favorably to the baseline, the Gradient Boosting Tree often cannot achieve similar results. Only for the subject *rhino386* can AlhazenML$_{GBT}$ provide a slightly better median

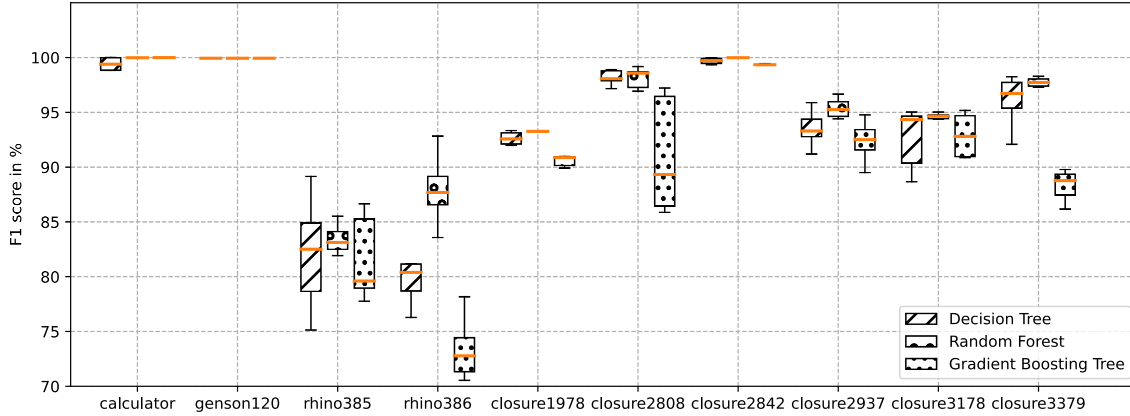



Figure 14: Achieved **F1 score** of *Alhazen*, ALHAZENML$_{RF}$, and ALHAZENML$_{GBT}$ for each subject over ten runs.

precision over the ten runs. However, for *rhino386*, the other techniques were able to generate much higher maximum values, indicated by the whiskers of the boxplots.

**F1 Score:** Finally, Figure 14 shows the achieved F1 score over the ten experimentations per subject and approach. As already mentioned in Section 5, one of the main advantages of the accuracy is that it is very easily interpretable. Still, the disadvantage is that it is not robust when the data is unevenly distributed. To mitigate this threat, we use the F1 score to compare the models. The visual data shows that ALHAZENML$_{RF}$ compares well against the other two configurations and archives a higher median F1 score for all subjects. This indicates that the Random Forest strikes a better balance between precision and recall than the other approaches. Similar to the previous metrics, the configuration with the Random Forest achieves the best results for the subjects *rhino386* and *closure2937*, increasing the median F1 score up to 7% and 3% compared to the baseline. Comparing all three approaches, we observe that the performance of ALHAZENML$_{GBT}$ is the weakest among the different methods. For almost all subjects, the achieved F1 score is noticeably worse than that of the baseline or ALHAZENML$_{RF}$.

Table 3 (Page 40) shows the accumulated results of our experiments over the ten repetitions. In addition, for each model, we display the *minimum*, *median*, and *maximum* values, as well as the *standard deviation* (SD) of the achieved accuracy, precision, and F1 scores. The detailed investigation supports our prior findings. We see that, apart from the subject *genson120*, ALHAZENML$_{RF}$ achieves the highest median accuracy and median F1 scores for all subjects. We also observe that our configuration with the Random Forest comes out on top regarding the maximum of all three metrics, with *rhino385* being the only outlier. Furthermore, comparing the standard deviation of all approaches reveals that ALHAZENML$_{RF}$ often achieves an order of magnitude lower SD values than the baseline. This indicates that the Random Forest reached the

| | | *Alhazen* (DecisionTree) | | | | AlhazenML$_{RF}$ (RandomForest) | | | | AlhazenML$_{GBT}$ (Gradient Boosting Tree) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | min | median | max | SD | min | median | max | SD | min | median | max | SD |
| calculator | Accuracy | 0.988326 | 0.993866 | 1.000000 | 6.050571e-03 | 0.998219 | 0.999802 | 1.000000 | 5.374370e-04 | 0.999604 | 1.000000 | 1.000000 | 0.000134 |
| | Precision | 0.988854 | 0.994131 | 1.000000 | 5.772373e-03 | 0.998232 | 0.999802 | 1.000000 | 5.335599e-04 | 0.999605 | 1.000000 | 1.000000 | 0.000133 |
| | F1 score | 0.988416 | 0.993911 | 1.000000 | 6.003376e-03 | 0.998221 | 0.999802 | 1.000000 | 5.367836e-04 | 0.999604 | 1.000000 | 1.000000 | 0.000134 |
| genson120 | Accuracy | 0.999356 | 0.999356 | 0.999356 | 1.170278e-16 | 0.999356 | 0.999356 | 0.999356 | 1.170278e-16 | 0.999034 | 0.999356 | 0.999356 | 0.000136 |
| | Precision | 0.999357 | 0.999357 | 0.999357 | 0.000000e+00 | 0.999357 | 0.999357 | 0.999357 | 0.000000e+00 | 0.999034 | 0.999357 | 0.999357 | 0.000136 |
| | F1 score | 0.999356 | 0.999356 | 0.999356 | 0.000000e+00 | 0.999356 | 0.999356 | 0.999356 | 0.000000e+00 | 0.999034 | 0.999356 | 0.999356 | 0.000136 |
| rhino385 | Accuracy | 0.712557 | 0.803991 | 0.897789 | 6.296875e-02 | 0.793631 | 0.826222 | 0.869712 | 2.429858e-02 | 0.846892 | 0.854123 | 0.883187 | 0.012686 |
| | Precision | 0.845326 | 0.867015 | 0.890114 | 1.441814e-02 | 0.853307 | 0.882625 | 0.917521 | 1.999343e-02 | 0.798592 | 0.860582 | 0.875306 | 0.022819 |
| | F1 score | 0.751361 | 0.825191 | 0.891482 | 4.620270e-02 | 0.797203 | 0.831364 | 0.855189 | 1.586676e-02 | 0.777634 | 0.796105 | 0.866524 | 0.035678 |
| rhino386 | Accuracy | 0.772825 | 0.793939 | 0.878983 | 3.516092e-02 | 0.834213 | 0.873607 | 0.925513 | 2.766833e-02 | 0.786119 | 0.798827 | 0.818377 | 0.010010 |
| | Precision | 0.758984 | 0.815819 | 0.895753 | 4.870663e-02 | 0.837585 | 0.895441 | 0.937584 | 2.787103e-02 | 0.781802 | 0.827803 | 0.836398 | 0.016737 |
| | F1 score | 0.762831 | 0.803929 | 0.883859 | 3.791486e-02 | 0.835765 | 0.877059 | 0.928303 | 2.526362e-02 | 0.705442 | 0.727817 | 0.781674 | 0.023382 |
| closure1978 | Accuracy | 0.885895 | 0.923500 | 0.931115 | 1.643844e-02 | 0.930178 | 0.930412 | 0.939082 | 3.047325e-03 | 0.906279 | 0.914363 | 0.932521 | 0.007537 |
| | Precision | 0.896585 | 0.934716 | 0.944765 | 1.577048e-02 | 0.944469 | 0.944616 | 0.949294 | 1.677888e-03 | 0.916272 | 0.922683 | 0.937155 | 0.005977 |
| | F1 score | 0.888814 | 0.925681 | 0.933347 | 1.593554e-02 | 0.932488 | 0.932709 | 0.940812 | 2.850734e-03 | 0.899130 | 0.908585 | 0.929296 | 0.008678 |
| closure2808 | Accuracy | 0.947500 | 0.980577 | 0.988846 | 1.236314e-02 | 0.968846 | 0.985673 | 0.991731 | 8.529472e-03 | 0.877500 | 0.903750 | 0.972500 | 0.041610 |
| | Precision | 0.946808 | 0.980880 | 0.989379 | 1.272111e-02 | 0.970540 | 0.986004 | 0.991771 | 7.830234e-03 | 0.894174 | 0.914353 | 0.972625 | 0.034435 |
| | F1 score | 0.946915 | 0.980674 | 0.988942 | 1.258168e-02 | 0.969271 | 0.985736 | 0.991745 | 8.358407e-03 | 0.858707 | 0.893273 | 0.972175 | 0.049408 |
| closure2842 | Accuracy | 0.993749 | 0.997031 | 0.999844 | 2.451068e-03 | 0.999844 | 0.999844 | 0.999844 | 1.170278e-16 | 0.993280 | 0.993749 | 0.994687 | 0.000459 |
| | Precision | 0.993789 | 0.997319 | 0.999845 | 2.457956e-03 | 0.999845 | 0.999845 | 0.999845 | 0.000000e+00 | 0.993326 | 0.993789 | 0.994716 | 0.000462 |
| | F1 score | 0.993361 | 0.997104 | 0.999844 | 2.571726e-03 | 0.999844 | 0.999844 | 0.999844 | 1.170278e-16 | 0.992827 | 0.993361 | 0.994412 | 0.000513 |
| closure2937 | Accuracy | 0.911431 | 0.935639 | 0.959078 | 1.326513e-02 | 0.922382 | 0.952450 | 0.965802 | 1.260323e-02 | 0.905091 | 0.929971 | 0.950048 | 0.014328 |
| | Precision | 0.912617 | 0.937462 | 0.958731 | 1.318339e-02 | 0.930390 | 0.954912 | 0.969855 | 1.078739e-02 | 0.914595 | 0.935310 | 0.952879 | 0.012064 |
| | F1 score | 0.911958 | 0.932915 | 0.958835 | 1.330028e-02 | 0.924588 | 0.952594 | 0.966590 | 1.194204e-02 | 0.895016 | 0.924974 | 0.947674 | 0.016792 |
| closure3178 | Accuracy | 0.799564 | 0.941711 | 0.948861 | 4.703836e-02 | 0.942075 | 0.944862 | 0.948861 | 2.377571e-03 | 0.914687 | 0.930683 | 0.950557 | 0.016702 |
| | Precision | 0.887242 | 0.952245 | 0.957181 | 2.771232e-02 | 0.952693 | 0.954566 | 0.957181 | 1.599513e-03 | 0.922510 | 0.935614 | 0.956752 | 0.016127 |
| | F1 score | 0.812366 | 0.943496 | 0.950269 | 4.412072e-02 | 0.943857 | 0.946492 | 0.950269 | 2.248605e-03 | 0.908615 | 0.928143 | 0.951722 | 0.020326 |
| closure3379 | Accuracy | 0.917086 | 0.967358 | 0.982371 | 2.293411e-02 | 0.951182 | 0.977141 | 0.982759 | 1.058903e-02 | 0.878342 | 0.898295 | 0.957381 | 0.022347 |
| | Precision | 0.937617 | 0.967343 | 0.982664 | 1.620906e-02 | 0.956236 | 0.977917 | 0.983285 | 8.956741e-03 | 0.895052 | 0.910243 | 0.958203 | 0.018033 |
| | F1 score | 0.920812 | 0.967293 | 0.982457 | 2.166807e-02 | 0.952291 | 0.977337 | 0.982884 | 1.023732e-02 | 0.861699 | 0.887498 | 0.956217 | 0.026654 |

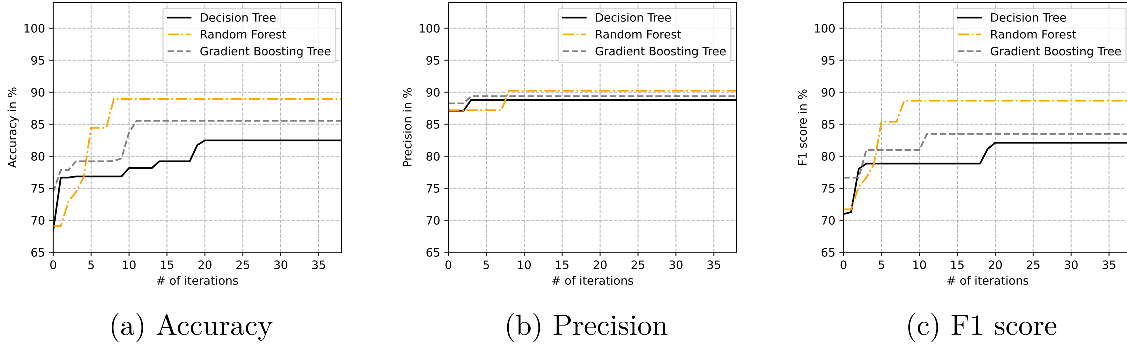
Table 3: Accumulated results for each approach and subject over ten runs. The table shows the *minimum*, *median*, and *maximum* values, as well as the *standard deviation* (SD) of the achieved accuracy, precision, and F1 scores.

given results more consistently. However, the table also shows that AlhazenML$_{GBT}$ with the Gradient Boosting Tree configuration did not yield any major improvement compared to the baseline; it often performs alike or worse than the baseline.

To further increase the confidence in our conclusions, we conducted a statistical analysis. As we consider independent samples and cannot make any assumption about the distribution of the results, we perform a non-parametric Mann-Whitney U test [MW47] to check whether the achieved results of the approaches differ significantly for each subject. For the analysis, we compare the achieved performance of AlhazenML$_{RF}$ and AlhazenML$_{GBT}$ to the baseline and report the results in Table 4 (Page 42). The table shows the median measurements for each subject and approach. In addition, it displays the *p-values* of the comparisons: baseline vs. AlhazenML$_{RF}$ and baseline vs. AlhazenML$_{GBT}$. The statistical analysis confirms that AlhazenML$_{RF}$ significantly improved accuracy, precision, and the F1 score for the *calculator*, *rhino386*, *closure1978*, *closure2842*, and *closure2937* subjects (*p-value* below 0.05). Although AlhazenML$_{RF}$ also achieves higher median values for the remaining subjects, we cannot conclude that the Random Forest significantly increased the performance compared to the baseline. For the configuration with the Gradient Boosting Tree, we can, as expected, only confirm a significant improvement for the *calculator* subject and the accuracy of *rhino385*.

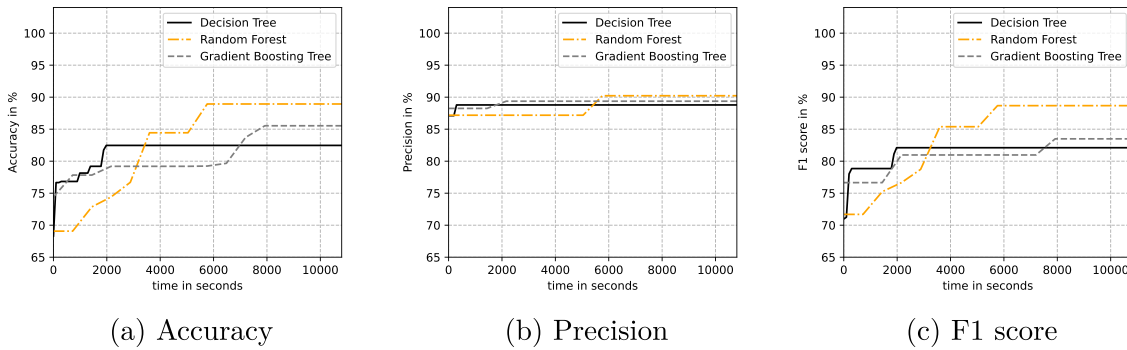
| | | *Alhazen* median | AlhazenML$_{RF}$ median | AlhazenML$_{RF}$ p-value | AlhazenML$_{GBT}$ median | AlhazenML$_{GBT}$ p-value |
|---|---|---|---|---|---|---|
| calculator | Accuracy | 0.993866 | **0.999802** | **0.037628** | 1.000000 | **0.003421** |
| | Precision | 0.994131 | **0.999802** | **0.037628** | 1.000000 | **0.003421** |
| | F1 score | 0.993911 | **0.999802** | **0.037628** | 1.000000 | **0.003421** |
| genson120 | Accuracy | 0.999356 | 0.999356 | 0.514744 | 0.999356 | 0.782064 |
| | Precision | 0.999357 | 0.999357 | 0.514744 | 0.999357 | 0.782064 |
| | F1 score | 0.999356 | 0.999356 | 0.514744 | 0.999356 | 0.782064 |
| rhino385 | Accuracy | 0.803991 | 0.826222 | 0.264424 | **0.854123** | **0.026213** |
| | Precision | 0.867015 | 0.882625 | 0.082747 | 0.860582 | 0.710629 |
| | F1 score | 0.825191 | 0.831364 | 0.369682 | 0.796105 | 0.573286 |
| rhino386 | Accuracy | 0.793939 | **0.873607** | **0.000752** | 0.798827 | 0.369682 |
| | Precision | 0.815819 | **0.895441** | **0.001943** | 0.827803 | 0.455899 |
| | F1 score | 0.803929 | **0.877059** | **0.000752** | 0.727817 | 0.999995 |
| closure1978 | Accuracy | 0.923500 | **0.930412** | **0.002598** | 0.914363 | 0.955395 |
| | Precision | 0.934716 | **0.944616** | **0.002598** | 0.922683 | 0.982269 |
| | F1 score | 0.925681 | **0.932709** | **0.002598** | 0.908585 | 0.978371 |
| closure2808 | Accuracy | 0.980577 | 0.985673 | 0.544101 | 0.903750 | 0.999897 |
| | Precision | 0.980880 | 0.986004 | 0.485256 | 0.914353 | 0.999838 |
| | F1 score | 0.980674 | 0.985736 | 0.514744 | 0.893273 | 0.999935 |
| closure2842 | Accuracy | 0.997031 | **0.999844** | **0.003421** | 0.993749 | 0.998057 |
| | Precision | 0.997319 | **0.999845** | **0.003421** | 0.993789 | 0.997402 |
| | F1 score | 0.997104 | **0.999844** | **0.003421** | 0.993361 | 0.998057 |
| closure2937 | Accuracy | 0.935639 | **0.952450** | **0.014403** | 0.929971 | 0.876275 |
| | Precision | 0.937462 | **0.954912** | **0.001943** | 0.935310 | 0.602032 |
| | F1 score | 0.932915 | **0.952594** | **0.004465** | 0.924974 | 0.917253 |
| closure3178 | Accuracy | 0.941711 | 0.944862 | 0.071570 | 0.930683 | 0.342105 |
| | Precision | 0.952245 | 0.954566 | 0.071570 | 0.935614 | 0.630318 |
| | F1 score | 0.943496 | 0.946492 | 0.071570 | 0.928143 | 0.397968 |
| closure3379 | Accuracy | 0.967358 | 0.977141 | 0.095158 | 0.898295 | 0.999978 |
| | Precision | 0.967343 | 0.977917 | 0.061503 | 0.910243 | 0.999978 |
| | F1 score | 0.967293 | 0.977337 | 0.082747 | 0.887498 | 0.999978 |

Table 4: This table shows the median accuracy, precision and F1 scores achieved by all approaches and their corresponding p-values. Values below 0.05 indicate statistical significance, and are highlighted in blue. Bold values indicate significantly higher values according to the Mann–Whitney U test.




(a) Accuracy (b) Precision (c) F1 score

Figure 15: Performance of all three approaches for the subject *rhino386* with the increasing number of iterations. Displayed is the median run of each approach.




(a) Accuracy (b) Precision (c) F1 score

Figure 16: Performance of all three approaches for the subject *rhino386* over time (seconds). Displayed is the median run of each approach.

### 6.1.2. Experimental Results over Time

The results of the previous Section show that ALHAZENML$_{RF}$ significantly improved the performance of five subjects. Finally, to further confirm the practical relevance of ALHAZENML as a reliable **predictor** of program behavior (**RQ1**), we compare the different approaches over time. We exemplify this evaluation for the subject *rhino386* and include the remaining subjects in the Appendix (Appendix A).

As mentioned in Section 4.2, we expect our approach with the two different machine learning configurations to take considerably more time. This is due to the introduced computational complexity regarding the number of trees used to learn the models and the resulting amount of new input specifications. To demonstrate that the additional computational overhead is defensible, we exemplify how the statistical measures change with increasing iterations and time for the subject *rhino386*.

The Figures 15 and 16 show how the accuracy, precision, and the F1 score for *rhino386* get improved with the increasing number of iterations and how the statistical

measures change over time. For each figure, the vertical axis represents the achieved performance of each approach in percent, and the horizontal axis represents the number of iterations (40) and the time in seconds (up to 10800 seconds = 3 hours), respectively.

Figure 15 shows that all approaches can improve their classification results by generating additional inputs with an increasing number of iterations. However, the detailed investigation shows that for the subject *rhino386*, all models eventually reach a plateau concerning the respective statistical measure. This happens after approximately eight iterations for ALHAZENML$_{RF}$, whereas the baseline with the Decision Tree and ALHAZENML$_{GBT}$ reach a plateau after 20 and 12 iterations, respectively. After these points, none of the approaches can increase the respective measure further.

Figure 16 shows the improvement of the statistical measures over time. We observe that *Alhazen* with the Decision Tree requires, as expected, notably less time for each iteration. Consequently, the baseline reaches the plateau after only about 2000 seconds, whereas ALHAZENML$_{RF}$ needs around 5800 seconds to reach maximum performance. ALHAZENML$_{GBT}$ is the last to reach a plateau after almost 8000 seconds. However, even though the baseline needs much less time for a single iteration, it cannot achieve the same accuracy, precision, and F1 score as ALHAZENML configured with the Random Forest estimator. Thus, we conclude that the significant improvement of the classification justifies the additional time overhead for our approach ALHAZENML$_{RF}$.

#### 6.1.3. Experimental Analysis

The experimental results show that ALHAZENML$_{RF}$ improved the performance compared to *Alhazen* significantly for five of the ten subjects. In our opinion, the main reasons for this observation stem from two constituents: (*i*) less overfitting to the training data and (*ii*) the generation of more additional inputs. On the on hand, as already examined, Random Forests are ensemble estimators that adjust far better to the sample set by not overfitting the training data. This is due to the training and operating of multiple Decision Trees and the prediction procedure that determines the final prediction (i.e., majority voting). This combination allows them to outperform Decision Trees and achieve higher accuracy. On the other hand, the additional Decision Trees also enable us to extract and obtain many more additional input files per iteration. The Random Forest uses these additional input files to refine all contained Decision Trees concurrently. Consequently, ALHAZENML$_{RF}$ reaches the maximum performance with far fewer iterations than the baseline. However, because all these additional inputs need to be produced, the time required to produce new inputs for an iteration is much higher.

Nonetheless, only generating more inputs does not automatically indicate a better performance of the machine learning estimator. For example, the results show that ALHAZENML$_{GBT}$ was not sufficient to produce better predictions. Moreover, even with more additional inputs generated per iteration (compared to the baseline), the configuration of ALHAZENML performed worse for most subjects. One reason for this might be how the ensemble of Decision Trees derive their predictions. As previously stated (Section 3), the trees are built in a stage-wise fashion, each correcting the errors

of its predecessor. Since the loss function and the error-correcting coefficients are not encoded in each tree, the generation of new inputs may not have a considerable impact on the improvement of the estimator. Consequently, the interpretability of the machine learning model may have a notable impact on the performance of the approach. In particular, if we do not know what predicates the estimator based its prediction on, the effective input generation might be limited. Another significant factor is that *Alhazen* already achieves impressive results, particularly for the *Closure* subjects. Moreover, its accuracy and precision are for almost all subjects in the high nineties, making it extremely hard for the other approach to improve them further.

**Answer to RQ1** In accordance with the results and the subsequent analyses reported above, we conclude the following:

> **RQ1** Based on our evaluation, we conclude that AlhazenML with the Random Forest, performs at least equally compared to the baseline and **significantly improved** the behavior classification for **five** of the ten subjects. Additionally, AlhazenML with the Gradient Boosting Tree significantly improved one additional subject, which the Random Forest could not advance.

| Subject | Accuracy | Precision | Failing Inputs |
|---|---|---|---|
| calculator.py | 1.0000 | 1.0000 | 5 |
| genson120.py | 1.0000 | 1.0000 | 2 |
| rhino385.py | 0.9512 | 0.9768 | 2 |
| rhino386.py | 0.9443 | 0.9607 | 3 |
| closure1978.py | 0.9888 | 0.9892 | 8 |
| closure2808.py | 0.9720 | 0.9822 | 6 |
| closure2842.py | 0.9258 | 0.9825 | 2 |
| closure2937.py | 0.9081 | 0.9467 | 11 |
| closure3178.py | 0.9616 | 0.9646 | 4 |
| closure3379.py | 0.9471 | 0.9711 | 13 |

Table 5: Accuracy, precision, and number of produced failing inputs when using the baseline with the **Decision Tree** as a producer. The displayed results are the median values over 10 runs.

## 6.2. AlhazenML as Producer

This Section evaluates whether AlhazenML can produce *bug-triggering* inputs efficiently (**RQ2**). With this assessment, we ensure that the different approaches do not overgeneralize.

We used the machine learning models (the baseline, AlhazenML$_{RF}$ with the Random Forest, and AlhazenML$_{GBT}$ with the Gradient Boosting Tree) generated in Section 6.1 and obtained the predicate sets and input specifications from the final models. We then evaluated whether the model's prediction matches the actual program behavior. The idea is to use the machine learning models to produce more failure-inducing inputs to help developers generate additional inputs for future test cases.

### 6.2.1. Experimental Results

Tables 5 through 7 show the results of our experiments. The "Accuracy" column shows the percentage of produced inputs (bug-triggering and non-bug-triggering) and whether they are actually bug-triggering and non-bug-triggering. We observe that all approaches achieve high accuracy, with the baseline producing the best median results for all subjects. The "Precision" column shows the percentage of inputs that the model predicted to be failure-inducing and whether the inputs actually caused the program behavior. Intuitively, precision can be interpreted as a metric on how sure we are that the input we are about to produce actually results in the predicted failure. Consequently, the higher the precision, the more efficient is the approach to produce bug-triggering inputs. Contrary, a low precision indicates that the failure-inducing inputs we want to produce are not indeed failing inputs, meaning we are wasting resources. The results again show that the baseline achieves the highest median precision. This indicates that

| Subject | Accuracy | Precision | Failing Inputs |
|---|---|---|---|
| calculator.py | 1.0000 | 1.0000 | 10 |
| genson120.py | 1.0000 | 1.0000 | 9 |
| rhino385.py | 0.9043 | 0.9653 | 29 |
| rhino386.py | 0.9206 | 0.9612 | 25 |
| closure1978.py | 0.9490 | 0.9672 | 33 |
| closure2808.py | 0.9427 | 0.9610 | 48 |
| closure2842.py | 0.9176 | 0.9530 | 24 |
| closure2937.py | 0.9119 | 0.9365 | 50 |
| closure3178.py | 0.9186 | 0.9646 | 15 |
| closure3379.py | 0.9261 | 0.9639 | 63 |

Table 6: Accuracy, precision, and number of produced failing inputs when using ALHAZENML with the **Random Forest** as a producer. The displayed results are the median values over 10 runs.

| Subject | Accuracy | Precision | Failing Inputs |
|---|---|---|---|
| calculator.py | 1.0000 | 1.0000 | 81 |
| genson120.py | 0.9826 | 0.9831 | 280 |
| rhino385.py | 0.9073 | 0.9042 | 118 |
| rhino386.py | 0.9120 | 0.9199 | 126 |
| closure1978.py | 0.9527 | 0.9540 | 129 |
| closure2808.py | 0.9424 | 0.9422 | 220 |
| closure2842.py | 0.9262 | 0.9304 | 81 |
| closure2937.py | 0.9110 | 0.9068 | 166 |
| closure3178.py | 0.9495 | 0.9503 | 56 |
| closure3379.py | 0.8948 | 0.8924 | 271 |

Table 7: Accuracy, precision, and number of produced failing inputs when using ALHAZENML with the **Gradient Boosting Tree** as a producer. The displayed results are the median values over 10 runs.

most inputs produced by the baseline, which were predicted to be bug-triggering, are, in fact, also bug-triggering. However, we recognize a noticeable difference in the total number of produced failing inputs ("Failing Inputs" column) between the approaches. The number of failing inputs ranges from 2 to 13 for the Decision Tree, 9 to 63 for the Random Forest, and 56 to 280 for the Gradient Boosting Tree. We expect this difference because we extract more input specifications for the ensemble estimators than for the single Decision Tree. Although we produce more failing inputs for the Gradient Boosting Tree, we are not as precise. Because the time needed to generate a single input from an input specification is the same for all approaches, one would prefer an approach that accurately and precisely generates inputs to use the available

resources as efficiently as possible.

To support our observation, we again perform a non-parametric Mann-Whitney U test [MW47] to check whether the achieved results of the approaches differ significantly. Table 8 shows the obtained results and displays the median accuracy and median precision achieved by all approaches for each subject. We again display the corresponding *p-values* of the comparisons: baseline vs. AlhazenML$_{RF}$ and baseline vs. AlhazenML$_{GBT}$. According to the Mann–Whitney U test, bold values indicate significantly higher values. We observe that the baseline achieves equal or better results than AlhazenML. Particularly, the baseline outperforms the Gradient Boosting Tree and achieves a significant better precision for the subjects *genson120*, *rhino385*, *closure1978*, *closure2842*, and *closure3379*. The baseline is also more accurate than the ensemble classifier for the subjects *genson120* and *rhino385* (AlhazenML$_{GBT}$). Additionally, compared to AlhazenML$_{RF}$ the baseline is more accurate for the subjects *rhino385* and *closure1978*. We cannot report a significant difference in precision for the baseline and the Random Forest.

### 6.2.2. Experimental Analysis

One reason the baseline is a more efficient producer may be related to extracting and producing new inputs for AlhazenML. Because the baseline only uses a single Decision Tree, each path of the tree corresponds to precisely one prediction outcome, namely either *bug-triggering* or *non-bug-triggering*. However, this is not the case for AlhazenML. As previously stated, the Random Forest and the Gradient Boosting Tree are ensemble classifiers. These models also learn and operate Decision Trees, but the prediction procedure is different. Because of the effects of, for instance, majority voting, a single tree can predict an input to be *bug-triggering*, whereas the remaining trees could predict the opposite (the final prediction would then be *non-bug-triggering*). Consequently, each tree only has a small impact on the final decision, and the prediction of a single tree does not necessarily reflect the decision of the ensemble. However, for our approach, we consider each Decision Tree independently from the collection to produce new inputs. Nevertheless, the real advantage of a Random Forrest is the combined predictive power of the weak learners. Hence, considering each tree on its own may be counterproductive for producing new inputs efficiently. Unfortunately, the same is true for the Gradient Boosting Trees. Even worse, because the Decision Trees within the Gradient Boosting Tree are often just slightly better than random classifiers, we see that the Gradient Boosting Tree performs significantly worse than the baseline.

Therefore, the production of new inputs is directly tied to the interpretability of the model. We can derive clear instructions to produce more inputs only when we precisely know why the model made the prediction. Consequently, because the ensemble estimators are much harder to interpret, we observe a slight loss in precision, particularly for the Gradient Boosting Tree.

| | | DecisionTree | RandomForest | | GradientBoosting | |
|---|---|---|---|---|---|---|
| | | median | median | p-value | median | p-value |
| calculator | Accuracy | 1.000000 | 1.000000 | 0.864334 | 1.000000 | 0.864334 |
| | Precision | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| genson120 | Accuracy | **1.000000** | 1.000000 | 1.000000 | 0.982617 | **0.001106** |
| | Precision | **1.000000** | 1.000000 | 1.000000 | 0.983056 | **0.002213** |
| rhino385 | Accuracy | **0.951190** | 0.904315 | **0.015573** | 0.907268 | **0.015573** |
| | Precision | **0.976763** | 0.965311 | 0.075222 | 0.904187 | **0.000323** |
| rhino386 | Accuracy | 0.944272 | 0.920648 | 0.366817 | 0.912026 | 0.285303 |
| | Precision | 0.960655 | 0.961215 | 0.909688 | 0.919864 | 0.103980 |
| closure1978 | Accuracy | **0.988768** | 0.949021 | **0.043886** | 0.952663 | 0.051404 |
| | Precision | **0.989199** | 0.967193 | 0.074562 | 0.954030 | **0.030576** |
| closure2808 | Accuracy | 0.972027 | 0.942686 | 0.213330 | 0.942400 | 0.120307 |
| | Precision | 0.982221 | 0.960951 | 0.519895 | 0.942201 | 0.161341 |
| closure2842 | Accuracy | 0.925824 | 0.917645 | 0.297587 | 0.926250 | 0.235502 |
| | Precision | **0.982517** | 0.952998 | 0.073463 | 0.930370 | **0.008594** |
| closure2937 | Accuracy | 0.908069 | 0.911867 | 0.352623 | 0.911009 | 0.454844 |
| | Precision | 0.946746 | 0.936534 | 0.909688 | 0.906787 | 0.307308 |
| closure3178 | Accuracy | 0.961594 | 0.918639 | 0.153380 | 0.949490 | 0.454793 |
| | Precision | 0.964600 | 0.964629 | 0.909586 | 0.950320 | 0.383957 |
| closure3379 | Accuracy | 0.947109 | 0.926050 | 0.366865 | 0.894800 | 0.092938 |
| | Precision | **0.971087** | 0.963860 | 0.909688 | 0.892397 | **0.017216** |

Table 8: Values show the median accuracy, precision, and corresponding p-values when comparing to the Decision Tree. Bold values indicate significantly higher values according to the Mann–Whitney U test; if the baseline with the Decision Tree achieved significantly higher results, the corresponding p-values are highlighted in red.

**Answer to RQ2** In accordance with the results and the subsequent analyses reported above, we conclude the following:

> **RQ2** Due to the lack of interpretability, AlhazenML is **equally or less efficient** as a producer than the baseline.

## 6.3. Threats to Validity

In this thesis, we rely on a search-based approach to precisely predict the behavior of a program; as such, we see the following potential threats to the validity of our work.

**Internal** The main threats to internal validity are caused by the random nature of machine learning algorithms. Therefore, it requires a careful statistical assessment to make sure that observed behaviors are not randomly occurring. Consequently, we repeated all experiments ten times and reported the descriptive statistics of our results.

To match the evaluation of Kampmann et al. [KHSZ20], we used the same set of subjects, bugs, and seed inputs. Furthermore, we automated the data collection and statistical evaluation. Finally, we did not tune the parameters of the baseline and AlhazenML to reduce the threat of overfitting to the given grammars and bugs. Only for the number of trees used for the Random Forest and Gradient Boosting Tree, we determined appropriate numbers to make the approaches computational feasible.

Another threat to internal validity is the selection of the measurement metrics. To compare our approach to the original work of Kampmann et al., we measure the accuracy and precision of the individual models. However, the generated data sets show a noticeable difference in the class data distribution; hence, solely relying on the accuracy may not be sufficient. We included the F1 score to mitigate this threat, allowing us to account for the subtleties of class imbalances and strengthen our results' confidence.

**External** The main threat to external validity is the generalizability of the experimental results that are based on a limited number of subjects and bugs. However, similar to Kampmann et al., practically relevant subjects with different complexities (small-sized grammars like JSON, and rather complex grammars like JavaScript) and widely used subjects (e.g., Rhino and Closure) have been selected. As a result, we are confident that our approach will also work on other grammars, subjects, and bugs.

# 7. Discussion and Limitations

In this Section, we further discuss the results of this thesis and present some limitations of our proposed approach.

**Different Use Cases** Our evaluation shows that AlhazenML with the Random Forest significantly improved the predictive power of *Alhazen* for many subjects. However, the improvement of the performance does not come without a cost. Due to our predicate extraction mechanism and the associated computational overhead (Section 4.2), AlhazenML needs considerably more time to complete an iteration of the feedback loop compared to the baseline. Nonetheless, our evaluation also shows that *Alhazen* eventually reaches a plateau for each subject, after which we do not observe any further improvement of the statistical measurements. The same is true for our approach, but AlhazenML achieves significantly higher accuracy, precision, and F1 scores than the baseline, justifying the additional time needed. Consequently, one could use the different approaches with their specific performance trade-offs in different scenarios. For example, on the one hand, if the goal is to explain the circumstances of a bug as efficiently as possible, a developer could use *Alhazen* and obtain good results in a short period of time. However, on the other hand, if the goal is to obtain a reliable, accurate, and precise model that classifies inputs correctly, developers would want to invest the additional time overhead. This allows them to train a better predictive model, which the baseline cannot reach.

**AlhazenML as a Debugging Aid** In their last evaluation step, Kampmann et al. [KHSZ20] examine how much the Decision Tree produced by their tool *Alhazen* allows developers to focus on the relevant aspects of the bug. They judge its capability to aid in debugging by evaluating the number of *non-terminal* nodes from the grammar that occur within the Decision Tree. The idea behind this is that the developers need to focus only on specific aspects of the grammar, essentially reducing the search space and making it easier to locate the failure. However, Random Forests and Gradient Boosting Trees are not as easy to interpret as the Decision Tree. Furthermore, we currently have no measurement to compare and quantify the usefulness of their explanations. Although our approach has been shown to significantly improve the predictive power of *Alhazen*, the quality of their explanation as to why the bug occurred depends on other methods (e.g., *SHAP* [LL17]). One possible solution for future work could be to use the concept of *surrogate models* and train, for instance, a Decision Tree on the generated inputs of AlhazenML and examine if it compares differently to the initial approach.

**Surrogate Models** As already briefly mentioned, learning a different machine learning model, such as a Decision Tree, is one option to interpret another classifier. In the literature, these models are also known as surrogate models [RSG16]. For example, a global surrogate model is an interpretable model trained to approximate the predictions of a black-box model [Mol19]. The idea is to draw conclusions about the black-box model by interpreting the surrogate model, essentially solving machine learning interpretability

by using more machine learning. Although these models were out of the scope of this thesis, future work could evaluate how surrogate models could be used to support the interpretability of *Alhazen* practically. For example, as hinted at before, one could use AlhazenML to derive a precise classification and then use, for instance, the interpretability of a Decision Tree to explain the circumstances of the program behavior. However, the drawn conclusions should be handled with caution because the surrogate model only observes the black-box model and does not evaluate the data.

**Support Vector Machines** Initially, we also wanted to evaluate Support Vector Machines (SVM) as a different machine learning alternative for the Decision Tree because of their ability classify non-linearly separable data. A linear SVM is a machine learning algorithm that labels unknown data points by relating them to a hyperplane of similar data points from a training dataset. For instance, let us consider a real-valued training dataset $X \in R^{m \times n}$ and a categorical label vector $Y \in \{-1, 1\}^n$ so that every data point $X^{(i)}$ is related to $Y^{(i)}$. A linear support vector machine tries to find the maximum-margin hyperplane that divides the binary training dataset. It, therefore, determines a support vector for each label so that the related data points are on and above or rather on and below the vectors. Both support vectors are parallel to each other, and their distance is maximized. The maximum-margin hyperplane lies halfway between the two support vectors and can then be used to determine the label of an unknown data point. Although they can achieve good results, particularly on non-linearly separable data, interpreting them is extremely challenging for many kernels. Even further, directly interpreting the higher dimensional support vectors to generate additional inputs was not expedient. Without the help of other methods, such as explainable machine learning techniques like SHAP, we could not extract and produce meaningful inputs. To limit the scope of this thesis, we set out to concentrate on white-box machine learning models, or ensemble models composed of white-box classifiers, to which most configurations of SVMs not belong. Thus we did not further advance them and neglected them for this thesis.

**Observable Behavior** Kampmann et al. [KHSZ20] developed *Alhazen* to explain the circumstances of the program behavior in association with the syntactical properties of the grammar. One of the main advantages of their tool is the feedback loop and the resulting refinement of the hypothesis. One reason why this feedback loop works is because of the observable outcome of the executed inputs. *Alhazen* can generate various inputs and directly obtain the corresponding outcomes to train a supervised model. Consequently, *Alhazen* only works if the oracle can reliably and accurately detect the program behavior in question. As mentioned in Section 4.3, this problem is directly connected to the oracle problem [BHM+15]. Often test oracles are built on assumptions of previous program versions or just the experience of the developer. However, this may undermine the test oracles: as they rely on implied conclusions and assumptions. If the oracle cannot label the program behavior correctly, *Alhazen* and, in particular, the different machine learning alternatives may draw incorrect conclusions.

# 8. Conclusion and Future Work

In this thesis, we proposed a modified extension of the tool *Alhazen* to predict and classify the circumstances of program behavior more precisely. For our approach, we replaced the proposed Decision Tree learner with more advanced machine learning models. We selected the Random Forest and the Gradient Boosting Tree as suitable candidates due to their tendency to correct the Decision Trees' habit of overfitting and their recent success for many different classification tasks. These ensemble estimators usually outperform Decision Trees, but unfortunately, they also sacrifice interpretability and explainability - a crucial property for *Alhazen*. In order to improve the initial hypothesis of the failure circumstances, *Alhazen* uses a feedback loop to iteratively refine a theory of why the input resulted in the observed program behavior. However, the feedback loop requires the generation and execution of additional inputs to strengthen the hypothesis. Therefore, we extend *Alhazen* with a modified learning process to train the different machine learning models and to extract the predicates needed for input production. Furthermore, to guide the input generation efficiently, we treat the individual learners of the ensemble estimator as white-box machine learning models. The individual Decision Trees allow us to extract the predicates the ensemble estimators deem responsible for the program behavior and generate different inputs accordingly.

We evaluated the effectiveness of our approach by performing experiments on ten real-world bugs and comparing our comprehensive framework AlhazenML to the initial approach of Kampmann et al. [KHSZ20]. Furthermore, we examined the different approaches in two settings: (*i*) as Predictors (how precisely can they predict the failure of a program) and (*ii*) as Producers (how efficiently can they produce more failure-inducing inputs). For the latter, the results suggest that our approach is generally not as efficient and cannot produce failure-inducing inputs as precise as the baseline. However, the results also show that AlhazenML equipped with a Random Forest was able to improve the behavior classification for five subjects significantly and performs equally to *Alhazen* for the remaining subjects.

In conclusion, there is no single best-performing machine learning model to determine the program's behavior for any possible bug. Indeed, our evaluation shows that different models perform better or worse for different grammars and subjects. However, our approach shows that we can use other machine learning models in combination with *Alhazen* and achieve better predictions for specific subjects.

**Future Work** There are several intriguing directions for future research. The *no-free-lunch theorem* argues that no single machine learning model is better than any other model [WM97]. Nonetheless, future work should further examine different machine learning alternatives, particularly black-box models, such as neural networks. The main challenge remains the generation of additional inputs. However, in contrast to our approach, which only explains the final theory, treating the learner inside *Alhazen* as a black-box may require more advanced explainable machine learning techniques to extract the predicates and refine the hypothesis.

Another exciting research direction is to combine *Alhazen* with grammar-based

fuzzing. The explanations of *Alhazen* in association with the syntactic features of the grammar could efficiently guide the input generation of fuzzing approaches. Furthermore, since most fuzzers randomly generate new data, using the refinement process to reduce the search space of the grammar and limiting the scope to a handful of non-terminals could be highly beneficial.

Finally, future work should also evaluate if *Alhazen* can be used to explain the circumstances of *non-functional* behavior. As previously mentioned, *Alhazen* requires a good test oracle that separates the classification labels and reliably determines whether, for instance, a non-functional requirement was violated. However, non-functional behavior, like execution time or memory consumption, may be subject to external influence. For example, running multiple processes simultaneously may have a high impact on the response time, and repeated measurements may fluctuate. Resultingly, training labels will be noisy, limited, or imprecise. In this scenario, Random Forests or Gradient Boosting Trees might have an advantage over Decision Trees because they may less overspecialize to certain outliers.

# References


[BFOS84] L Breiman, JH Friedman, R Olshen, and CJ Stone. Classification and regression trees. 1984.

[BGMP21] Niklas Bussmann, Paolo Giudici, Dimitri Marinelli, and Jochen Papenbrock. Explainable machine learning in credit risk management. *Computational Economics*, 57(1):203–216, 2021.

[BHM+15] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[Bre01] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[BSC+17] Marcel Böhme, Ezekiel Olamide Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 11th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE 2017, pages 1–11, 2017.

[CCW18] Yung-Chia Chang, Kuei-Hu Chang, and Guan-Jhih Wu. Application of extreme gradient boosting trees in the construction of credit risk assessment models for financial institutions. *Applied Soft Computing*, 73:914–920, 2018.

[Clo19] Google closure. `https://github.com/google/closure-compiler`, 2019. v20180101.

[CPML18] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 95–105, 2018.

[DIFBF19] S. Delphine Immaculate, M. Farida Begam, and M. Floramary. Software bug prediction using supervised machine learning algorithms. In *2019 International Conference on Data Science and Communication (IconDSC)*, pages 1–7, 2019.

[EE08] Karim O Elish and Mahmoud O Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649–660, 2008.

[EKG+14] Katherine Ellis, Jacqueline Kerr, Suneeta Godbole, Gert Lanckriet, David Wing, and Simon Marshall. A random forest classifier for the prediction of energy expenditure and type of physical activity from wrist and hip accelerometers. *Physiological measurement*, 35(11):2191, 2014.

[FMEH20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*, 2020.

[FNR14] P. Filipovikj, M. Nyberg, and G. Rodriguez-Navas. Reassessing the pattern-based approach for formalizing requirements in the automotive domain. In *2014 IEEE 22nd International Requirements Engineering Conference (RE)*, pages 444–450, 2014.

[Fri01] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[Gen17] Genson. https://github.com/owlike/genson, 2017. Version 1.4.

[GPAM+20] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

[GPS17] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 50–59. IEEE, 2017.

[GSC+19] D. Gunning, M. Stefik, J. Choi, T. Miller, S. Stumpf, and G-Z. Yang. Xai-explainable artificial intelligence. *Science Robotics*, 4(37), December 2019.

[HMU01] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.

[HUAM+19] Markus Hofmarcher, Thomas Unterthiner, José Arjona-Medina, Günter Klambauer, Sepp Hochreiter, and Bernhard Nessler. Visual scene understanding for autonomous driving using semantic segmentation. In *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, pages 285–296. Springer, 2019.

[HZ19] Nikolas Havrikov and Andreas Zeller. Systematically covering input structure. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 189–199. IEEE, 2019.

[ILMP19] Mark Ibrahim, Melissa Louie, Ceena Modarres, and John Paisley. Global explanations of neural networks: Mapping the landscape of predictions. In *Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society*, pages 279–287, 2019.

[JPC19] Kristin Johnson, Frank Pasquale, and Jennifer Chapman. Artificial intelligence, machine learning, and bias in finance: toward responsible innovation. *Fordham L. Rev.*, 88:499, 2019.

[KHSZ20] Alexander Kampmann, Nikolas Havrikov, Ezekiel Soremekun, and Andreas Zeller. When does my program do this? learning circumstances of software behavior. In *ESEC/FSE 2020*, June 2020.

[LCH+09] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. Classification of software behaviors for failure detection: a discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 557–566, 2009.

[LEC+20] Scott M. Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M. Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. From local explanations to global understanding with explainable ai for trees. *Nature Machine Intelligence*, 2(1):2522–5839, 2020.

[LL17] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017.

[LNV+18] Scott M Lundberg, Bala Nair, Monica S Vavilala, Mayumi Horibe, Michael J Eisses, Trevor Adams, David E Liston, Daniel King-Wai Low, Shu-Fang Newman, Jerry Kim, et al. Explainable machine-learning predictions for the prevention of hypoxaemia during surgery. *Nature biomedical engineering*, 2(10):749–760, 2018.

[LZP+17] Peng Liu, Xiangyu Zhang, Marco Pistoia, Yunhui Zheng, Manoel Marques, and Lingfei Zeng. Automatic text input generation for mobile testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 643–653. IEEE, 2017.

[MFS90] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, 1990.

[Mol19] Christoph Molnar. *Interpretable Machine Learning*. 2019.

[MW47] H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[RBVK18] Robert Regtuit, Clark Borst, and Erik-Jan Van Kampen. Building strategic conformal automation for air traffic control using machine learning. In *2018 AIAA Information Systems-AIAA Infotech@ Aerospace*, page 0074. 2018.

[RDK19] Alvin Rajkomar, Jeffrey Dean, and Isaac Kohane. Machine learning in medicine. *New England Journal of Medicine*, 380(14):1347–1358, 2019.

[Rhi18] Mozilla Rhino. https://github.com/mozilla/rhino, 2018. Version 1.7.8.

[RSG16] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.

[SH77] Philip H. Swain and Hans Hauska. The decision tree classifier: Design and potential. *IEEE Transactions on Geoscience Electronics*, 15(3):142–147, 1977.

[Sha16] L. S. Shapley. *17. A Value for n-Person Games*, pages 307–318. Princeton University Press, 2016.

[SML+21] Wojciech Samek, Grégoire Montavon, Sebastian Lapuschkin, Christopher J Anders, and Klaus-Robert Müller. Explaining deep neural networks and beyond: A review of methods and applications. *Proceedings of the IEEE*, 109(3):247–278, 2021.

[SMV+19] Wojciech Samek, Gregoire Montavon, Andrea Vedaldi, Lars Kai Hansen, and Klaus-Robert Muller. Explainable ai: Interpreting, explaining and visualizing deep learning, 2019.

[SPH+20] Ezekiel Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. Inputs from hell learning input distributions for grammar-based test generation. *IEEE Transactions on Software Engineering*, 2020.

[TNČT20] Saeid Tizpaz-Niari, Pavol Černỳ, and Ashutosh Trivedi. Detecting and understanding real-world differential performance bugs in machine learning libraries. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 189–199, 2020.

[UKHM19] Shahadat Uddin, Arif Khan, Md Ekramul Hossain, and Mohammad Ali Moni. Comparing different supervised machine learning algorithms for disease prediction. *BMC medical informatics and decision making*, 19(1):1–16, 2019.

[WM97] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

# A. Appendix

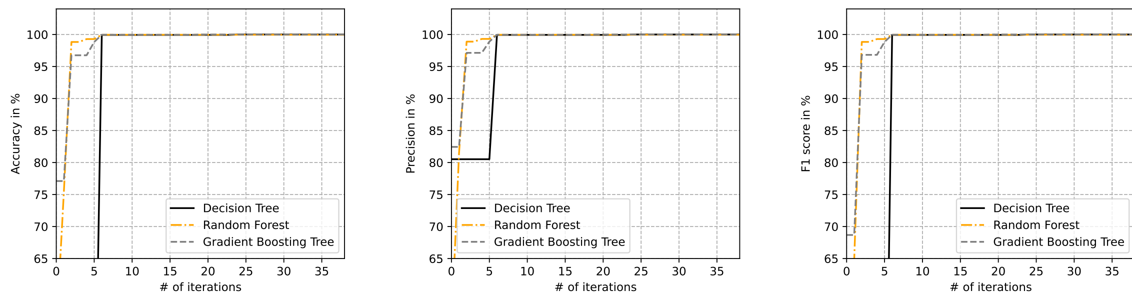
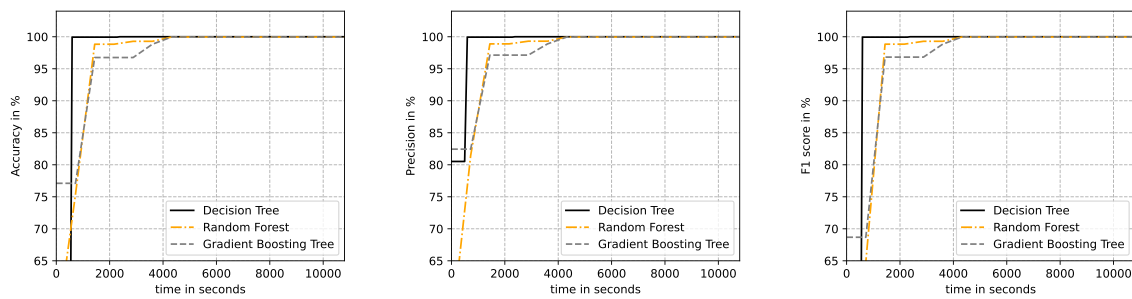



Figure 17: Performance for the subject *calculator* over increasing number of iterations.




(a) Accuracy

(b) Precision

(c) F1 score

Figure 18: Performance for the subject *calculator* over time..

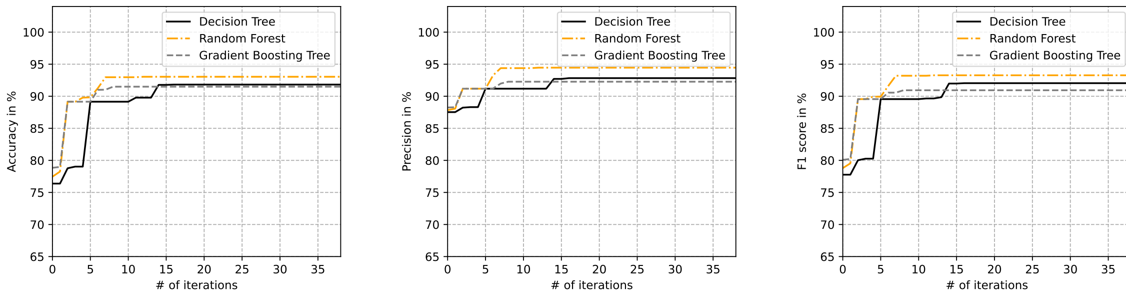
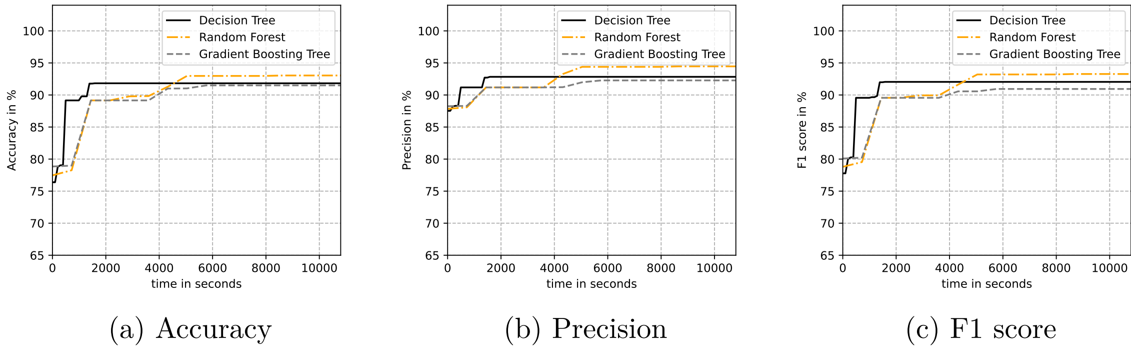



Figure 19: Performance for the subject *closure1978* over increasing number of iterations.




(a) Accuracy (b) Precision (c) F1 score

Figure 20: Performance for the subject *closure1978* over time..

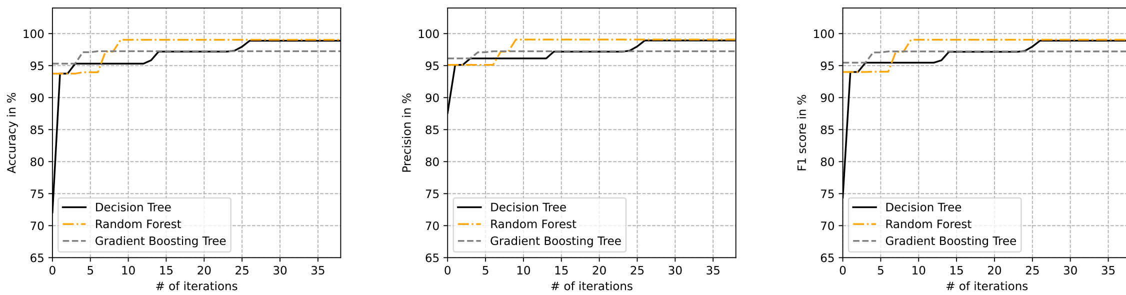
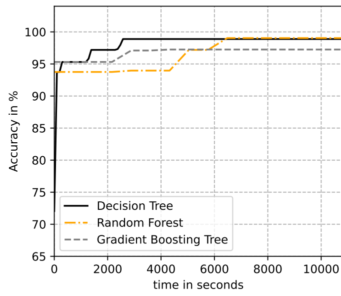
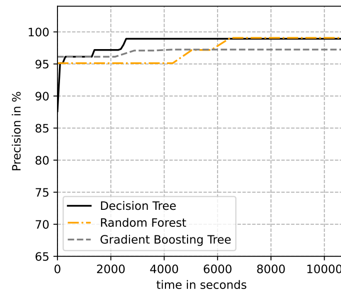


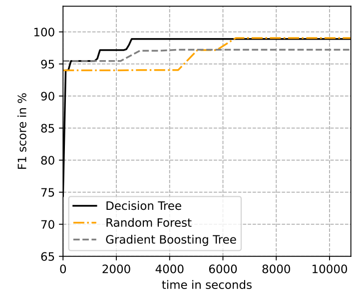

Figure 21: Performance for the subject *closure2808* over increasing number of iterations.




(a) Accuracy

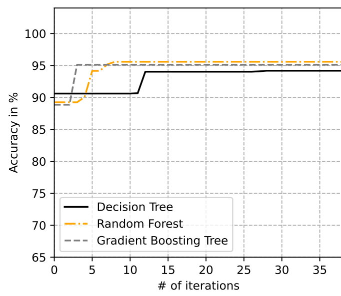


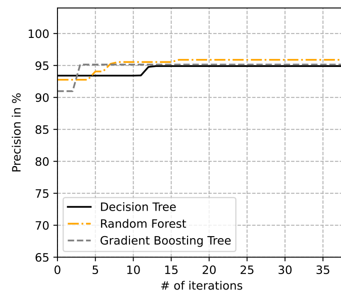

(b) Precision




(c) F1 score

Figure 22: Performance for the subject *closure2808* over time.



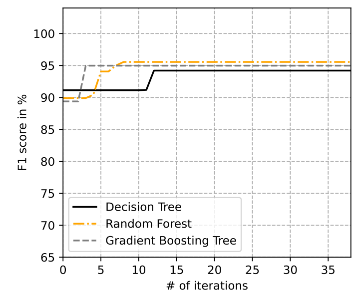
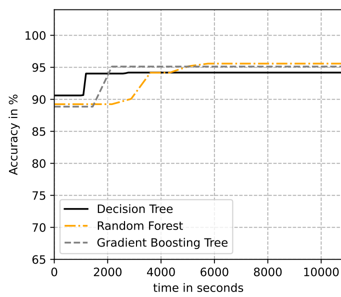
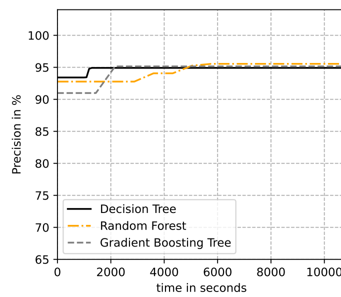



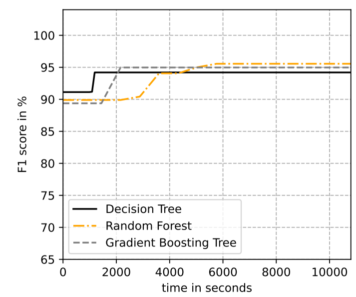

Figure 23: Performance for the subject *closure2937* over increasing number of iterations.




(a) Accuracy




(b) Precision




(c) F1 score

Figure 24: Performance for the subject *closure2937* over time.