# Automatic Unit Test Generation for Machine Learning Libraries: How Far Are We?

Song Wang
York University, Toronto, Canada
wangsong@yorku.ca

Nishtha Shrestha
York University, Toronto, Canada
shrestha.nishtha@gmail.com

Abarna Kucheri Subburaman
York University, Toronto, Canada
abarnaks@my.yorku.ca

Junjie Wang
Chinese Academy of Sciences, Beijing, China
junjie@iscas.ac.cn

Moshi Wei
York University, Toronto, Canada
moshiwei@yorku.ca

Nachiappan Nagappan*
Microsoft Research, Redmond, USA
nnagappan@acm.org

***Abstract*—Automatic unit test generation that explores the input space and produces effective test cases for given programs have been studied for decades. Many unit test generation tools that can help generate unit test cases with high structural coverage over a program have been examined. However, the fact that existing test generation tools are mainly evaluated on general software programs calls into question about its practical effectiveness and usefulness for machine learning libraries, which are statistically-orientated and have fundamentally different nature and construction from general software projects.**

**In this paper, we set out to investigate the effectiveness of existing unit test generation techniques on machine learning libraries. To investigate this issue, we conducted an empirical study on five widely-used machine learning libraries with two popular unit test case generation tools, i.e., `EVOSUITE` and `Randoop`. We find that (1) most of the machine learning libraries do not maintain a high-quality unit test suite regarding commonly applied quality metrics such as code coverage (on average is 34.1%) and mutation score (on average is 21.3%), (2) unit test case generation tools, i.e., `EVOSUITE` and `Randoop`, lead to clear improvements in code coverage and mutation score, however, the improvement is limited, and (3) there exist common patterns in the uncovered code across the five machine learning libraries that can be used to improve unit test case generation tasks.**

***Index Terms*—Empirical software engineering, test case generation, testing machine learning libraries**

## I. Introduction

Software unit testing is widely recognized as a crucial part of software development process to foster and assure software quality. However, writing effective unit test cases is an extremely costly and time-consuming practice [1]. To reduce such a burden for developers, many automatic techniques and tools based on different mechanisms have been proposed, e.g., `EVOSUITE` [2] generates unit test cases based on genetic algorithms, `Randoop` [3, 4] generates unit test cases by using feedback-directed approaches. To evaluate the effectiveness and usefulness of these tools, many studies have been conducted to compare the quality of automatically generated and manually written unit test suites [5, 6, 7, 8]. Results confirm that the automated test generation tools, e.g., `EVOSUITE` and `Randoop`, are effective at producing unit test suites with high code coverage and these tools can also be an useful aid to help write unit test cases. However, the fact that most of these studies used randomly selected general projects leaves unanswered the more directly relevant question: *Do the automatic unit test generation techniques also work for machine learning libraries?*

We are witnessing a wide adoption of Machine Learning (ML) models in many software systems lately. Software applications powered by ML are being used in critical sectors of our daily lives; from finance and energy, to health and transportation [9, 10, 11]. Thus, building reliable and secure ML systems has become an increasingly critical challenge for software developers. However, ML libraries are often statistically-orientated, and have fundamentally different nature and construction compared to general software projects [10, 12], which makes the usefulness of existing automatic test generation tools on them unknown.

In this paper, we set out to investigate the effectiveness of the widely-used automatic unit test generation techniques on ML libraries. Specifically, we select five widely-used ML libraries, i.e., Weka [13], Stanford CoreNLP [14], Mallet [15], OpenNLP [16], and Mahout [17]. Additionally, to better understand ML libraries, inspired by existing studies [10, 12], we decompose a ML library into three different types of components, i.e., `data process`, `core model`, and `util` (Details are in Section II-B). We use two typical automatic unit test generation tools, i.e., `EVOSUITE` and `Randoop`, as the experiment objectives following prior studies [5, 6].

For our study, we first perform an empirical study on the five ML libraries to unveil the effectiveness of their current unit test suites regarding commonly applied quality metrics such as code coverage and mutation score [18]. We then apply `EVOSUITE` and `Randoop` on these ML libraries to generate unit tests and check whether `EVOSUITE` and `Randoop` could improve test effectiveness on these libraries, regarding code coverage and mutation score, by comparing the automatically generated tests against the existing manually created ones. Note that, to better understand the effectiveness and scope of the two unit test generation tools, we apply them on the three

*This work was done when Dr. Nagappan was with Microsoft Research.

different components, i.e., `data process`, `core model`, and `util` from each experimental ML library.

We find that most ML libraries do not maintain a high-quality unit test suite regarding code coverage (on average, 34.1%) and mutation score (on average, 21.3%). Our manual analysis show that most test effort of ML libraries are spent on testing a subset of valid functionalities while leaving a large portion of code uncovered. In addition, the examined unit test generation tools, i.e., `EVOSUITE` and `Randoop`, can lead to clear improvements in code coverage and mutation score, however, the improvement is limited. Our manual investigation on randomly selected 150 classes from the five libraries show that there exist common patterns in the uncovered code across the five ML libraries which can be used to improve the unit test case generation tasks for future research.

This paper makes the following contributions:

- We conduct a comprehensive investigation of current unit test practices on five widely-used machine learning libraries.
- We examine the effectiveness and usefulness of two widely-used automatic unit test generation tools on five machine learning libraries.
- We identify gaps between existing automatic unit test generation techniques and unit testing practices on machine learning libraries.
- We discuss general lessons learned and future directions from the application of the automatic unit test generation to machine learning libraries.

The rest of this paper is organized as follows. Section II describes the background on automatic unit test generation and machine learning libraries. Section III shows the setup of our empirical studies. Section IV presents the results of our study. Section V discusses open questions and the threats to the validity of this work. Section VI surveys the related work. Finally, we summarize this paper in Section VII.

## II. Background

### A. Automatic Unit Test Generation

Software unit testing is widely adopted to test individual units/components of software projects. A unit test is an executable piece of code that validates a functionality of a class or a method under test performing as designed. While there are many techniques to automatically generate unit tests, we focus on two types of typical and scalable approaches based on random generation of call sequences, i.e., `Randoop`, and search-based optimization of call sequences, i.e., `EVOSUITE`, in this work.

*1) Random Testing:* Random testing is a basic approach for test generation [19], which consists of invocation of functions with random inputs. Guided random testing is a refined approach that starts with random input data, then uses extra knowledge to guide input data generation. One typical example of random testing is feedback-directed random testing [4], which enhances random test generation by incorporating feedback collected from executing test inputs that is used to avoid generating duplicate and illegal input data. Feedback-directed based approaches build test inputs incrementally, and then the newly created test inputs extend previous ones. These test inputs are executed as soon as they are created. The results collected from these executions will then be used to guide the generation of new test inputs. We use the representative feedback-directed unit test generation tool, i.e., `Randoop`, in our experiments.

*2) Search-based Testing:* Search-based software testing is an approach that transforms the unit test generation tasks into optimization problems [20], where the objective of the test generation is implemented by a fitness function that guides the search. Genetic algorithm is widely used as the search techniques for test generation [21]. In the genetic algorithm, randomly selected candidate solutions are evolved by applying evolutionary operators, such as mutation and crossover, resulting in new offspring individuals, with better fitness values. The widely used objective function for unit test generation is the code coverage of the generated tests [22]. In this work, we examine the typical search-based unit test generation tool, i.e., `EVOSUITE`.

### B. Machine Learning Libraries

Machine learning is a type of artificial intelligence technique that requires huge volumes of data to be able to converge for making meaningful inferences, decisions, or predictions [23]. Most machine learning techniques share a common basic procedure. The first step when constructing a ML model is to collect data from a domain where specific concepts can be learned using some algorithms. Once data is collected, often a pre-processing step will be conducted before it can be used for learning. The most common pre-processing operations include data format converting, noise data filtering, and feature engineering. The result of data pre-possessing often can significantly affect the quality of trained models. Once the data is cleaned and features are extracted or selected properly, a learning algorithm is used to infer relations capturing hidden patterns in the data. During this learning process, the parameters of the algorithm are tuned to fit the input data through an iterative process. After training steps, the model is evaluated, and can be deployed in a specific application environment and interact with other components of the application to finish expected tasks.

A ML library is a collection of machine learning, data mining, or natural language processing algorithms, which provides usable and easily extensible API for both software developers and research scientists. To facilitate these algorithms, most ML libraries often contain data processing and utility components [10, 12].

To understand how existing automatic unit test generation tools can help test the different steps of ML models, i.e., data pre-processing and model building, we broke a ML library into the following three components at class level:

- `data process`: Classes that contain APIs to pre-process data before building machine learning models, e.g., data format transformation, data sampling,

data normalization, noise data filtering, and feature engineering related tasks.
- `core model`: Classes that implement the core ML algorithms. For example, Classification (e.g., image classification, software bug prediction), Regression (e.g., temperature prediction, software fault density prediction), Clustering (e.g., pattern recognition, image segmentation), Translation, Named Entity Recognition, Sentiment Analysis, Topic Segmentation, etc.
- `util`: Classes that provide miscellaneous services for evaluating models (e.g., cross-validation), measuring performance ( e.g., confusion matrix), internationalization, loading and outputting data, etc.

Note that, machine learning can be classified into conventional machine learning and deep learning. Machine learning tasks like Classification, Regression, Clustering, Translation, Named Entity Recognition, Sentiment Analysis, and Topic Segmentation, belong to conventional machine learning. Deep learning [24] adopts multiple layers of nonlinear processing units for feature extraction and transformation. Typical deep learning algorithms often follow some widely used neural network structures like Convolutional Neural Networks (CNNs) [25] and Recurrent Neural Networks (RNNs) [26]. This paper only focuses on conventional machine learning libraries as most deep learning libraries such as TensorFlow [27], Theano [28], PyTorch [29], Scikit-learn [30], are developed in Python, while existing widely-used automatic unit test generation tools are designed for object oriented languages such as Java, C++, and C#, which fit most conventional machine learning libraries.

## III. Empirical Study Setup

This section describes our experiment methodology. The main objective of this study is to evaluate the effectiveness of existing widely-used unit test generation tools in ML libraries, and to understand the barriers for machine learning developers when adopting these tools.

### A. Research Questions

To achieve the mentioned goal, we have designed experiments to answer the following research questions:

**RQ1:** *What is the quality of the current unit testing in ML libraries?*

The studied five ML libraries have been developed for years. They also maintain a stable unit test suite to test the provided functionalities. This RQ first reveals the quality of existing unit test in the machine learning libraries regarding their testing coverage and mutation scores.

A large number of widely-adopted ML libraries originated from academic research studies and are maintained by researchers (i.e., academic-led). Meanwhile, with the rapid adoption of AI technologies, many open-source ML libraries have been developed by the open-source community (i.e., community-led). Little is known about the difference in unit testing practices between academic-led and community-led ML libraries. In this RQ, we further explore the unit testing quality of ML libraries from these two different categories to better understand the difference in unit testing practices between academic-led and community-led ML libraries.

**RQ2:** *How effective are automatic unit test generation tools on ML libraries?*

Recent studies [5, 7, 31, 32] confirmed that the automatic test generation tools, i.e., `EVOSUITE` and `Randoop`, are effective at generating unit test suites with high code coverage and mutation scores for general software projects. This RQ intends to assess the capability of the two automatic unit test generation tools in ML libraries. Specifically, we use `Randoop` and `EVOSUITE` to generate test cases for each class in the five ML libraries and further evaluate the effectiveness of the generated unit test cases by examining the increased code coverage and mutation score in these classes.

**RQ3:** *What is the covered and uncovered code with original unit test suites in ML libraries?*

To understand the testing focus of these ML libraries, i.e., which parts of a ML library have been tested and which parts have not been tested with the original unit test suites, we conduct a manual analysis to check the covered and uncovered code after executing the original test cases from each ML library.

**RQ4:** *To what extent can automatic unit test generation tools help test ML libraries?*

Even with the test cases generated by automatic unit test generation tools, not all code can be covered [7, 31]. Following the results of RQ3, this RQ checks to what extent the test cases generated, by `Randoop` and `EVOSUITE`, for these ML libraries can help cover code missed by the original unit test suites of ML libraries. Based on the analysis, we further explore the potential strategies to improve unit test generation tasks for ML libraries in Section V.

### B. Studied Unit Test Generation Tools

In this work, two widely-used mature unit test generation tools that can generate JUnit supported test cases are selected, i.e., `Randoop` and `EVOSUITE` .

`Randoop` implements feedback-directed random test generation which first builds test inputs incrementally, and then the newly created test inputs extend the previous ones. It can generate tests containing assertions that capture the current state. To generate unit tests for each class, we used the default configurations and set the time limit to 3 minutes as suggested by [32]. `EVOSUITE` generates test suites with the aim of maximising code coverage (e.g., branch coverage), minimising the number of unit tests and optimising their readability. The generated tests include assertions that capture the current behaviour of the implementation. In our experiments, we use the default configurations of `EVOSUITE`.

In this work, we run both `Randoop` and `EVOSUITE` on a 4.0GHz i7-3930K desktop with 16GB of memory.

### C. Subjects of Study

In this work, we set out to investigate five Java-based widely-used ML libraries, i.e., Weka [13], Stanford CoreNLP [14], OpenNLP [16], MALLET [15], and Mahout [17].

Weka (Waikato Environment for Knowledge Analysis) is a ML library developed at the University of Waikato, and has been downloaded over 3.6 million times [6]. It provides multiple algorithms for data pre-processing, attribute selection, classification and regression, clustering, association rules mining, etc. Stanford CoreNLP is one of the dominating ML based toolkits for the processing of natural language text developed by the Natural Language Processing Group at Stanford University. It supports services for NLP tasks, e.g., language detection, tokenization, sentence segmentation, part-of-speech tagging, named entity extraction, chunking, etc. MALLET, developed by University of Massachusetts Amherst in 2002, is a Java-based ML library for document classification, topic modeling, information extraction, and other machine learning applications to text.

The above three ML libraries were originated from academic research studies. We also collect two open source community-led ML libraries, i.e., Apache OpenNLP and Apache Mahout. Apache OpenNLP library is a machine learning based toolkit for the processing of natural language text, which provides similar NLP services as Stanford CoreNLP. Apache Mahout is a powerful, scalable machine-learning library that runs on top of Hadoop MapReduce. It provides machine learning algorithms for clustering, classification and batch based collaborative filtering, etc. Table I shows the statistic of the five ML libraries.

### D. Classifying Classes in Machine Learning Libraries

We conduct a manual study to group each class in a machine learning library into three categories, i.e., `data process`, `core model`, and `util`. For each class in a machine learning library, three of the authors independently use the following steps to conduct the manual classification tasks:

- Step 1: We read the Javadoc documentation of the class to understand its functionality. A category will be assigned to the class based on its major intent. Specifically, if the class provides the services related to data process, we categorize it as `data process`, if the class implements a machine learning algorithm, we categorize it as `core model`. Note that, if the class involves both data process and machine learning algorithm ($< 3\%$ of all classes analyzed), we label it as `core model` only.
- Step 2: If a decision cannot make based on the description of the class's API as the description might not contain enough information, we further check the source code following the instructions in Step 1 to label the class.
- Step 3: A class that does not belong to `data process` and `core model` categories will be grouped as `util`.

During this manual analysis, all disagreements were discussed until a consensus was reached. The results of this phase have a Cohens kappa of 0.77, which is a substantial level of agreement [33]. Figure 1 shows the distribution of the three different types of classes in the five ML libraries. Surprisingly, `util` category is dominating across all the projects, which takes up to 81.2% of all the classes (in Mahout) and is 54.6% on average. Only 30% classes are `core model` classes across these five ML libraries and the percentage of classes belonging to `data process` varies dramatically across the five projects and ranges from 0.3% (Mahout) to 26.4% (OpenNLP).

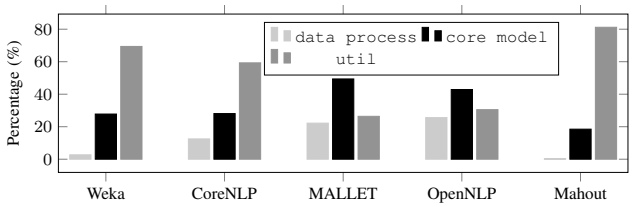


Fig. 1: Percentage of the three types of components in each experimental machine learning library.

Overall, Weka and Mahout have dramatically unbalanced distributions among the three categories. Our further investigation on the two projects finds possible reasons for this. Unlike CoreNLP, OpenNLP, or MALLET, which support unstructured data, Weka requires specific data format, i.e., `arff` [13], to run its machine learning algorithms. Although data with some other structured formats (e.g., csv, json, etc.) can also be loaded but they have to be saved to arff format for later use. In addition, Weka does not support unstructured data. Most of Weka's `data process` classes focus on data format transformation. Mahout has fewer `data process` classes since it is mainly used to provide machine learning analysis in Hadoop processing pipeline and it maintains a large number of collection data structures that belong to `util`.

### E. Metrics for Evaluating Testing Quality

To measure the quality of a unit test suite given a project, following existing work [34], we use both code coverage [35] and mutation score [36] as the metrics.

Our coverage analysis was performed using JaCoCo[1], which can measure instruction and branch coverage. The instruction coverage refers to Java bytecode instructions and thus is similar to statement coverage on source code. As JaCoCo's definition of branch coverage counts only branches of conditional statements, not edges in the control flow graph, we only use instruction coverage for measuring code coverage.

Mutation testing generates program variants (i.e., mutants) for the original program under test using mechanical transformation rules (i.e., mutation operators). Each mutant is the same with the original program except for the mutated statement. A mutant is `killed` by a test suite if any test from the suite failed with the mutant. Mutation score is defined as the percentage of killed mutants with the total number of mutants. A higher mutation score means a better quality of a test suite. Existing studies [37, 38, 39] have shown that mutation faults

[1]https://www.eclemma.org/jacoco/

TABLE I: Details of the experimental machine learning libraries.

| Project | Type | Description | Version | #KLOC | #Class | #Test Case |
|---|---|---|---|---|---|---|
| Weka | Academic-Led | A general machine learning library. | 3.9.4 | 340 | 1,616 | 4,071 |
| Stanford CoreNLP | Academic-Led | A set of machine learning based NLP tools. | 3.9.2 | 573 | 1,720 | 1,262 |
| MALLET | Academic-Led | A machine learning for language toolkit. | 2.1 | 77 | 615 | 184 |
| Apache OpenNLP | Community-Led | A machine learning based toolkit for NLP. | 1.9.2 | 71 | 672 | 785 |
| Apache Mahout | Community-Led | A distributed machine learning framework. | 2.12 | 25 | 142 | 2,538 |

TABLE II: The number of generated mutations on each category of classes in the five machine learning libraries.

| | Weka | CoreNLP | MALLET | OpenNLP | Mahout |
|---|---|---|---|---|---|
| `data process` | 3,788 | 24,521 | 2,868 | 3,149 | 18 |
| `core model` | 21,168 | 34,306 | 16,600 | 7,215 | 6,087 |
| `util` | 42,180 | 38,055 | 8312 | 3,260 | 18,687 |

are close to real faults. Along this line, mutation testing has been used to evaluate the quality of existing test suites [40, 41].

In this work, we use the widely-used mutation test framework PITest [42] to generate mutants and measure the mutation score of a unit test suite. Note that PITest contains many different mutation operators for generating different mutants, in this work we use the default 11 operators list in its website[2].

## IV. Empirical Analysis

### A. RQ1: Current Unit Test Quality

To answer this question, we first execute the original unit test suite provided by these machine learning libraries, which are JUnit test cases. As we described in Section III-E, we use JaCoCo to collect the coverage information and use PITest to generate mutation faults and further get mutation scores on each project. These five libraries are maintained by Maven. To enable JaCoCo and PITest, we manually add the JaCoCo and PITest maven plugins into each project's Maven configuration file. JaCoCo calculates the coverage for each class via checking whether a specific instruction is covered by at least one test case. PITest is configured to mutate specified classes in a project and check the generated mutants against specified unit test cases.

Table II shows the numbers of mutants generated for different types of classes in each library. We further compute the Spearman correlation between the number of mutants and the number of classes in each category from these libraries. The high correlation value (0.74) indicates that the number of mutants generated for a category has a positive correlation with the number of classes in the category, e.g., Weka has more mutants on `core model` and `util` than `data process` since the number of classes from Weka's `core model` and `util` components are larger than that of `data process` component. This is reasonable since the mutation operators provided by PITest do not have potential bias for specific programs [42]. In this question, we run PITest with all classes and the original unit tests from each library.

Table III shows the average coverage and mutation score for classes in the three different categories in each machine learning library. Surprisingly, the code coverage on these libraries is low and the overall coverage ranges from 16.1% (on CoreNLP) to 58.9% (on Mahout), which is significantly lower that the code coverage on 100 randomly chosen general open-source projects, i.e., around 70% [43]. The overall mutation score on these libraries ranges from 10.9% (on CoreNLP) to 46.0% (on Mahout) and on average is 21.3%, which is also lower than the average mutation score of a set of 154 top open source general software project collected from Github, i.e., around 40% [44].

In addition, as we categorize the ML libraries into two different categories, i.e., academic-led and community-led , and decompose a ML library into three different components, i.e., `data process`, `core model`, and `util`, we further explore the quality of unit tests for different categories and different components, as shown in Table III. Overall, the two open source community-led ML libraries have both higher coverage and mutation scores than that of academic-led ML libraries. Specifically, the average coverage on the three academic-led ML libraries is 20.0%, which is 36.7% lower than that of community-led ML libraries. The average mutation score of academic-led ML libraries is 11.5%, which is 24.4% lower than that of community-led ML libraries. We further conduct the Wilcoxon signed-rank test to compare the unit testing quality regarding coverage and mutation scores between academic-led and community-led ML libraries. Results suggest that community-led machine learning libraries significantly ($p < 0.05$) outperform academic-led ML libraries regarding either code coverage or mutation scores.

We also observe an extremely lower code coverage and mutation score in at least one category from each of the academic-led ML libraries. For example, the `util` category in project Weka has a significantly lower code coverage (i.e., 3.3%) and mutation score (i.e., 1.6%) compared to the other two categories, i.e., code coverage is larger than 40% and mutation score is larger than 24%. The same phenomenon can be observed on the `data process` category in CoreNLP, and the `core model` category in MALLET. We show the detailed distributions of code coverage of classes from each category in Figure 2, and further conduct the Wilcoxon signed-rank test ($p < 0.05$) to compare the unit testing quality regarding code coverage among the three different categories in each project. Results show that the `data process` and `core model` in Weka have a significantly higher code coverage than `util`. In CoreNLP, `util` has a significantly higher code coverage than `data process` and `core model`. In MALLET, `util` and `data process` have a significantly higher code coverage than `core model`. We do not find any significant differences regarding the code coverage among the three categories

[2]https://pitest.org/quickstart/mutators/

TABLE III: The average coverage (i.e., Coverage) and mutation score (i.e., MScore) of the unit test suite for each machine learning library. Overall indicates the results on all the classes in a project.

| Class Type | Weka | | CoreNLP | | MALLET | | OpenNLP | | Mahout | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Coverage | MScore | Coverage | MScore | Coverage | MScore | Coverage | MScore | Coverage | MScore |
| `data process` | 43.5 | 24.8 | 12.2 | 5.1 | 31.5 | 20.3 | 61.7 | 38.1 | 91.7 | 38.9 |
| `core model` | 44.1 | 28.6 | 17.8 | 11.1 | 11.0 | 7.9 | 50.6 | 16.6 | 63.8 | 50.3 |
| `util` | 3.3 | 1.6 | 18.1 | 14.3 | 23.9 | 17.6 | 54.6 | 34.1 | 57.5 | 44.6 |
| **Overall** | 23.9 | 11.5 | 16.1 | 10.9 | 16.7 | 12.1 | 54.5 | 25.8 | 58.9 | 46.0 |

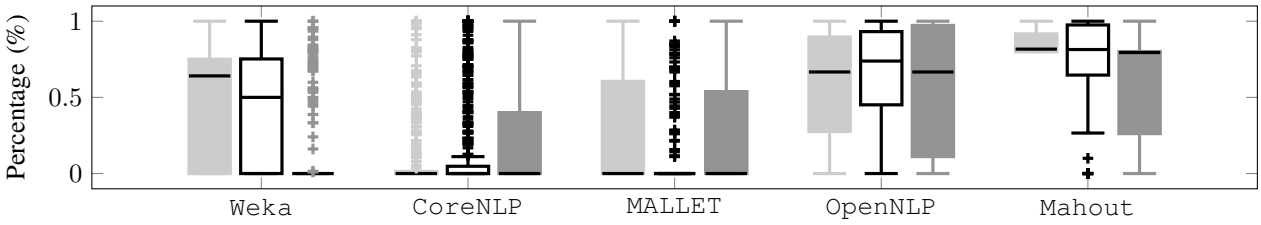



Fig. 2: The distribution of code coverage for `data process` (i.e., ●), `core model` (i.e., ○), and `util` (i.e., ●) in each project.

from the two community-led ML libraries. Note that we also observe a similar trend in mutation scores, but we do not show the detailed distribution of mutation scores due to space limitations. This indicates that the testing effort of academic-led ML libraries is unbalanced among different components.

> Current unit test suite in ML libraries has lower quality regarding code coverage (on average, 34.1%) and mutation score (on average, 21.3%). In addition, the testing effort of academic-led ML libraries is unbalanced distributed and their unit test quality is significantly worse than that of community-led ML libraries.

### *B. RQ2: Effectiveness of Test Generation Tools*

To answer this question, we run `Randoop` and `EVOSUITE` to generate test cases for each of the five ML libraries as described in Section III-B. With the generated test cases, we further run JaCoCo to collect the coverage information for classes from the three different categories, i.e., `data process`, `core model`, and `util`, on each library. Note that, both `Randoop` and `EVOSUITE` generated flaky or uncompilable test classes. In order to collect coverage information and mutation scores, we have removed these flaky or uncompilable test cases with the approaches suggested in [32]. Specifically, first, we removed all non-compiling test classes. Second, we executed each compilable test suite five times, and removed all new flaky tests from the executions. This process was repeated until all remaining tests passed five times in an iteration.

Table IV presents the number of generated test cases for classes of each category in each project, the absolute coverage of the generated test cases, and the additional coverage compared to the original unit test suite. Note that we did not run PITest for each category of classes collecting mutation scores as we cannot finish running PITest within a reasonable time frame (i.e., 100 hours) for most projects with large numbers of generated test cases (i.e., ranging from 5K to 140K). To explore the mutation scores of the generated test cases from `Randoop` and `EVOSUITE`, we randomly selected 20 classes from each category of each ML library and ran PITest with them to generate mutation faults and further obtain mutation scores. We show the additional mutation scores compared to mutation scores of the original unit test suite in Table V.

From Table IV, we can see that the number of generated test cases significantly vary in different projects, e.g., overall the two tools generate more test cases in Weka and CoreNLP than that in OpenNLP, MALLET, and Mahout. One of the possible reasons for this is that Weka and CoreNLP have a lot more classes where `Randoop` and `EVOSUITE` can collect more method sequences to generate test cases. We also observe that `Randoop` generates more test cases than `EVOSUITE` in four of the five projects, which is because `Randoop` does not target a specific class under test [32].

Column '$\Delta$ Coverage' shows the increment in code coverage of test cases generated by `Randoop` and `EVOSUITE`, from which we can see that `EVOSUITE` has a much higher increased code coverage (an average of 17.6 percentage points) than `Randoop` (an average of 5.2 percentage points) and we further conduct the Wilcoxon signed-rank test ($p < 0.05$) to compare the increased code coverage between `Randoop` and `EVOSUITE` in these ML libraries. Results confirm that `EVOSUITE` produces a significantly higher increased code coverage than `Randoop`. However, we also find that even with the test cases generated by `Randoop` and `EVOSUITE`, the percentage of uncovered code in these projects are still high, i.e., ranges from 20.1% (Mahout with `EVOSUITE`) to 81.5% (CoreNLP with `Randoop`) and is 45.4% on average, which indicates the limited improvements of these tools.

Table V shows the increment in mutation score of test cases generated by `Randoop` and `EVOSUITE` on sample classes. As we can see from the table, both `Randoop` and `EVOSUITE` can improve the mutation scores on these ML libraries. In addition, similar with the trend in increased code coverage,

TABLE IV: The number of generated test cases (i.e., #generated test), the absolute coverage (i.e., Coverage), and the increased code coverage (i.e., Δ Coverage) of Randoop and EVOSUITE on the five experimental machine learning libraries.

| Project | Category | Randoop | | | Evosuite | | |
|---|---|---|---|---|---|---|---|
| | | #generated test | Coverage (%) | Δ Coverage (%) | #generated test | Coverage (%) | Δ Coverage (%) |
| Weka | data process | 6,574 | 62.8 | 19.3 | 1,935 | 75.9 | 32.4 |
| | core model | 6,923 | 45.1 | 1.0 | 5,179 | 44.8 | 0.7 |
| | util | 25,696 | 18.5 | 15.2 | 6,061 | 25.4 | 22.1 |
| | **Overall** | 39,193 | 32.4 | 8.5 | 13,175 | 41.8 | 17.9 |
| CoreNLP | data process | 66,022 | 18.1 | 5.9 | 4,119 | 40.6 | 28.4 |
| | core model | 42,925 | 20.1 | 2.3 | 7,380 | 44.2 | 26.4 |
| | util | 34,493 | 19.2 | 1.1 | 3,920 | 39.8 | 21.7 |
| | **Overall** | 143,440 | 18.5 | 2.4 | 15,419 | 40.6 | 24.5 |
| MALLET | data process | 5,022 | 41.4 | 9.9 | 1,364 | 36.0 | 4.5 |
| | core model | 8,581 | 22.8 | 11.8 | 3,012 | 21.9 | 10.9 |
| | util | 524 | 29.3 | 5.4 | 339 | 47.8 | 23.9 |
| | **Overall** | 14,310 | 27.0 | 10.3 | 4,715 | 31.8 | 15.1 |
| OpenNLP | data process | 20,264 | 64.6 | 2.9 | 1,177 | 73.8 | 12.1 |
| | core model | 27,422 | 57.1 | 6.5 | 1,989 | 51.8 | 1.2 |
| | util | 37,970 | 56.5 | 1.9 | 1,884 | 77.1 | 22.5 |
| | **Overall** | 85,656 | 58.8 | 4.3 | 5,050 | 64.0 | 9.5 |
| Mahout | data process | 792 | 91.8 | 0.1 | 13,156 | 100 | 8.3 |
| | core model | 3,930 | 65.0 | 1.2 | 10,575 | 66.0 | 2.2 |
| | util | 4,866 | 57.7 | 0.2 | 14,582 | 84.3 | 26.8 |
| | **Overall** | 9,588 | 59.3 | 0.4 | 38,313 | 79.9 | 21.0 |

TABLE V: Increased mutation score (i.e., Δ MScore) of Randoop and EVOSUITE on sample classes.

| Project | Randoop | Evosuite |
|---|---|---|
| | Δ MScore (%) | Δ MScore (%) |
| Weka | 17.5 | 31.8 |
| CoreNLP | 15.1 | 24.2 |
| MALLET | 8.4 | 17.9 |
| OpenNLP | 10.3 | 14.5 |
| Mahout | 2.3 | 19.0 |

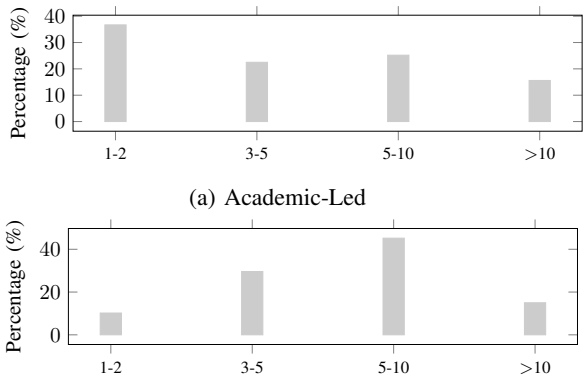
EVOSUITE has much higher increased mutation scores (on average is 21.5%) than Randoop (on average is 10.7%), which is confirmed by our Wilcoxon signed-rank test ($p < 0.05$).

EVOSUITE and Randoop lead to clear improvements in code coverage and mutation score compared to the original unit test suites of ML libraries. However, on average, 45.4% code is still uncovered with the generated test cases.

### C. RQ3: Covered & Uncovered Code

Results of RQ1 show that the studied five machine learning libraries have low test quality regarding either code coverage or mutation score. In this RQ, to understand the testing focus of these ML libraries, we conduct an manual analysis and explore which parts of a machine learning library have been tested and which parts have not been tested with the original unit test suites provided by these ML libraries. For our analysis, we randomly selected 10 classes from each category (i.e., data process, core model, and util) of each ML library, i.e., in total 150 classes are collected. We fetch the the test cases of each class, and collect the covered code and uncovered code for each class by using JaCoCo.

We first manually analyze the original test cases to check the covered functionalities of ML libraries. Surprisingly, almost all libraries only test a part of supported valid functionalities. Taking the class weka.classifiers.trees.J48 from project Weka as an example, which is a decision tree based classifier. For testing this class, developers created one test case in which an object of J48 with default parameters was created and then the classifier was trained and tested with its testing data, while other valid parameters are not covered. We further calculate the number of functionalities tested. Figure 3 shows the distributions of the number of functionalities tested in the examined classes from academic-led (i.e., Figure 3a) and community-led (i.e., Figure 3b) libraries respectively. For academic-led libraries, we can see that around 35% classes have only tested one or two functionalities and only about 45% classes test more than 5 functionalities, while for community-led libraries, more than 65% classes have tested 5 or more different functionalities. Nevertheless, among the sampled 150 classes, each class has an average of 15 valid functionalities.




(a) Academic-Led

(b) Community-Led

Fig. 3: The distribution of tested functionalities in the examined classes from academic-led (a) and community-led libraries (b).

Based on the 150 classes, we further check the uncovered code from each class to understand what has been missed from current unit test suite of machine learning libraries. Three authors work together to manually categorize the uncovered code into five different types. We use code from Weka library as examples to illustrate each category as follows.

**Lack of testing valid behaviors (VB).** A class often has multiple valid behaviors, e.g., a parameter can be assigned with different values, while only a set of valid behaviors have been tested. Below is a `VB` example from class `J48` (i.e., an algorithm used to generate a decision tree). Method `setOptions()` is used to set up the essential options before building the `J48` classifier. The option 'N' showed in the code (line 3) is used for pruning the built decision tree, which could affect the performance of the built decision tree and the default value is assigned in line 7. The corresponding unit case of `J48` did not specify a value for 'N' when calling method `setOptions()`, which leaves line 5 uncovered.

```
1 public void setOptions (String[] options)
      throws Exception {
2 ...
3 String numFoldsString = Utils.getOption('N',
      options);
4 if (numFoldsString.length() != 0) {
5  m_numFolds=Integer.parseInt(numFoldsString);
6 } else {
7     m_numFolds = 3;}
8 ...
9 }
```

**Lack of testing invalid behaviors (IVB).** A class also has possible invalid behaviors, e.g., a parameter often has a valid scope, values outside the scope will be invalid, while not all invalid behaviors have been tested. Below is a `IVB` example from class `RandomTree` (i.e., an algorithm used to generate a random tree based classifier). Method `buildClassifier()` is used to build `RandomTree` classifier. The method first checks the validity of parameters, in which 'm_KValue' is number of attributes used to build the classifier. Developers can set 'm_KValue' before calling `buildClassifier()`, a valid 'm_KValue' ranges from 1 to the size of the used attribute set. The corresponding unit case of `RandomTree` did not test any invalid value for 'm_KValue', which leaves line 5 uncovered.

```
1 public void buildClassifier(Instances data)
      throws Exception {
2 ...
3 // Make sure K value is in range
4 if (m_KValue > data.numAttributes() - 1) {
5     m_KValue = data.numAttributes() - 1;
6     }
7 ...
8 }
```

**Lack of testing exception (EX).** In Java, developers can throw an exception in a method by using the throw keyword, which will cause an exception to be raised and will require the calling method to catch the exception or throw the exception to the next level in the call stack. JUnit enables developers to test exception by using the `ExpectedException` rule. We find that most exceptions in these libraries have not been tested. Below is a `EX` example from class `CSVLoader` (i.e., a class used to load CSV format data for building classifiers). Method `setOptions()` is used to set up the parameters of `CSVLoader`. The method first parses data from the input arguments and then checks the validity of parameters. 'B' is the size of the in-memory buffer used to load data, a valid 'B' should be larger than 1 (line 7), if the value is smaller than 1, the method will throw an Exception. The corresponding unit case of `CSVLoader` did not test any value for 'B' that can triage the exception throw statement, which leaves line 8 uncovered.

```
1 public void setOptions(String[] options)
      throws Exception {
2 ...
3 //'B': The size of the in memory buffer
4 tmpStr = Utils.getOption('B', options);
5 if (tmpStr.length() > 0) {
6     int buff = Integer.parseInt(tmpStr);
7     if (buff < 1) {
8     throw new Exception("Buffer size must be
          >= 1");
9     }
10         setBufferSize(buff);}
11 ...
12 }
```

**Lack of testing auxiliary methods (AUX).** Most classes provide auxiliary methods to interact with attributes or perform other tasks. We find that most auxiliary methods in these libraries have not been tested. Below is a `AUX` example from class `GaussianProcesses` (i.e., a class that implements Gaussian processes for regression analysis). Method `getStandardDeviation()` is used to get standard deviation of the prediction at the given instance. The method was used inside `GaussianProcesses` as an auxiliary method. However, the corresponding unit case of `GaussianProcesses` did not test this method, which leaves it uncovered.

```
1 public double getStandardDeviation(Instance
      inst) throws Exception {
2 ...
3 }
```

**Lack of testing message handling behaviors (MEB).** A class often provides a series of methods for handling behaviors to help print messages for developers. Our analysis reveals that a large number of message handling behaviors in these libraries have not been tested. Below is a `MEB` example from class `DecisionTable` (i.e., a class that build a decision table based classifier). Method `printFeatures()` is used to print string description of the features selected. The method was used inside `DecisionTable` as a message handling method. However, the corresponding unit case of `DecisionTable` did not test this method, which leaves this method uncovered.

```
1 public String printFeatures() {
2     int i;  String s = "";
3     ...
4     return s;
5 }
```

TABLE VI: Distribution of uncovered code in each ML library (in percentage).

| | **VB** (%) | **IVB** (%) | **EX** (%) | **AUX** (%) | **MEB** (%) |
|---|---|---|---|---|---|
| Weka | 23.5 | 6.0 | 21.4 | 24.3 | 24.8 |
| CoreNLP | 13.7 | 8.2 | 5.6 | 53.3 | 19.2 |
| MALLET | 15.8 | 3.6 | 17.1 | 58.5 | 4.8 |
| OpenNLP | 30.6 | 5.0 | 22.3 | 29.7 | 12.4 |
| Mahout | 19.8 | 11.4 | 16.7 | 40.6 | 11.5 |
| **Overall** | 16.5 | 7.6 | 17.7 | 42.4 | 15.8 |

With the above five categories of uncovered code, we manually categorized each of the uncovered code block from the 150 studied classes into one specific category, and then we obtain the ratios of the five categories on each ML library. Table VI shows the distribution of uncovered code from each machine learning library regarding the above five categories. For example, category **VB** takes up 23.5% among all the uncovered code blocks in Weka.

As we can see from the table, overall the ratios of the five different categories vary among different projects. However, **AUX** is dominating across the five ML libraries (ranges from 24.3% to 58.5%), which takes up 42.4% of all the uncovered code blocks in these ML libraries. The Wilcoxon signed-rank test ($p < 0.05$) also confirms that the ratio of **AUX** is significant higher than other categories. We also find that categories **VB**, **EX**, and **MEB** also have ratios higher than 15% of all the uncovered code blocks. Comparing to the other four categories, the ratio of category **IVB** is minor (ranges from 3.6% to 11.4%), one of the possible reasons is that the size of code blocks about invalid behaviors is smaller.

> Overall, the unit test suites in ML libraries mainly focus on a subset of valid functionalities. In addition, there exists common patterns among the uncovered code of the studied ML libraries.

### *D. RQ4: Improvement of Unit Test Generation Tools*

In RQ3 (Section IV-C), we show the five different categories about the uncovered code in these ML libraries. We then conduct a further analysis to explore which part of source code can be covered by using test cases generated by `Randoop` and `EVOSUITE` on the 150 classes collected in RQ3. Specifically, for each class, we collected its uncovered code blocks after executing the original unit test suite and the covered code blocks after executing test cases generated by `Randoop` and `EVOSUITE`. Then, for each piece of the uncovered code blocks (after executing the original unit test suite), we further check whether it is covered by test cases generated by `Randoop` and `EVOSUITE`. For each category of uncovered code on a ML library, we calculate the percentage of removed uncovered code blocks after executing test cases generated by `Randoop` and `EVOSUITE`, e.g., `Randoop` can help remove 9.1% uncovered code of the original unit test suite of Weka on category **VB**. The details are shown in Table VII.

As we can see from the table, both `Randoop` and `EVOSUITE` can help reduce a larger portion of uncovered code for the five different uncovered code categories. In addition, we can also see that `EVOSUITE` removes much more uncovered code (ranges from 36.4% to 93.5%) than that of `Randoop` (ranges from 11.8% to 63.8%), and we further conduct the Wilcoxon signed-rank test ($p < 0.05$) to compare the percentages of removed uncovered code between `EVOSUITE` and `Randoop`. Results suggest that `EVOSUITE` performs significantly better than `Randoop` on each category of uncovered code.

TABLE VII: Removed uncovered code blocks after executing test cases generated by `Randoop` and `EVOSUITE` (in percentage).

| | | **VB** (%) | **IVB** (%) | **EX** (%) | **AUX** (%) | **MEB** (%) |
|---|---|---|---|---|---|---|
| Weka | `Randoop` | 9.1 | 14.3 | 26.0 | 54.0 | 60.3 |
| | `EVOSUITE` | 41.8 | 42.9 | 44.0 | 86.0 | 96.6 |
| CoreNLP | `Randoop` | 16.0 | 6.7 | 20.0 | 25.8 | 93.3 |
| | `EVOSUITE` | 36.0 | 40.0 | 20.0 | 66.7 | 100 |
| MALLET | `Randoop` | 46.2 | 0 | 21.4 | 45.8 | 50.0 |
| | `EVOSUITE` | 46.2 | 33.3 | 21.4 | 47.9 | 50.0 |
| OpenNLP | `Randoop` | 13.5 | 0 | 14.8 | 36.1 | 86.7 |
| | `EVOSUITE` | 40.5 | 16.7 | 18.5 | 41.7 | 93.3 |
| Mahout | `Randoop` | 10.5 | 18.1 | 6.2 | 23.1 | 27.3 |
| | `EVOSUITE` | 42.1 | 54.5 | 75.0 | 66.7 | 81.8 |
| **Overall** | `Randoop` | 15.2 | 11.8 | 19.0 | 38.8 | 63.8 |
| | `EVOSUITE` | 40.0 | 43.1 | 36.4 | 64.1 | 93.5 |

We can also observe that both `Randoop` and `EVOSUITE` perform better on **AUX** and **MEB** categories, i.e., overall at least 38% of them can be removed by `Randoop` and at least 64% of them can be removed by `EVOSUITE`, while overall at most 19% uncovered code from other three categories (**VB**, **IVB**, and **EX**) can be removed by `Randoop` and at most 43% of them can be removed by `EVOSUITE`. Our Wilcoxon signed-rank test ($p < 0.05$) shows that the performances of both `Randoop` and `EVOSUITE` on **AUX** and **MEB** are significantly better than that of other three categories. The main reason is that most **AUX** and **MEB** related methods do not require input arguments or only need simple input arguments, which makes it easy for both `Randoop` and `EVOSUITE` to generate compilable and valid sequences of method calls to cover them. However, to cover **VB**, **IVB**, and **EX**, one needs test cases with valid parameters, invalid parameters, different parameter values, different input data, etc., which often cannot generate by existing unit test generation tools, e.g., `EVOSUITE` and `Randoop`.

> Both `EVOSUITE` and `Randoop` can significantly help cover **AUX** and **MEB**, while the performance on other three categories, i.e., **VB**, **IVB**, and **EX**, is limited.

## V. Discussion

### *A. Implications*

Our study reveals several interesting findings that can serve as the practical guidelines for improving unit test generation tasks for ML libraries.

**Combining unit test generation with parameter analysis:** Our manual analysis in Section IV-D shows that most of the uncovered code from **VB** category is caused by missing testing valid parameters. Different from general software projects,

most ML libraries contain a large number of parameters. However, neither EVOSUITE nor Randoop can help generate test cases with valid parameters as most parameters are maintained in property documents, thus one of the future directions to improve unit test generation for ML libraries is combining existing test generation with parameter analysis.

**Combining unit test generation with data generation:** Most of existing unit test generation tools (e.g., EVOSUITE and Randoop) only focus on generating test cases by selecting method call sequences and finding arguments from previously-constructed inputs. However, ML libraries are data-driven, to cover most **IVB** and some **EX**, test cases with special training or input data are needed. Thus, another future direction to improve unit test generation for ML libraries is combining existing test generation with test data generation together to generate method call sequences with newly generated input data.

**Transferring unit test across ML libraries:** Different from general software projects, most ML libraries share the same knowledge domain and often provide the same data processing steps, machine learning algorithms, and utility support services, which makes it possible to transfer unit test cases of a specific machine learning component between ML libraries, e.g., the studied Weka and MALLET share around 50 classification algorithms with the same or similar parameters. Thus, transferring unit tests across ML libraries that share similar algorithms could be an applicable attempt to generate unit test for ML libraries.

### B. Threats to Validity

**Internal Validity**: Our study uses EVOSUITE and Randoop for automatic test generation. It is possible using different automatic test generation tools may yield different results. Nevertheless, these two tools are modern test generation tools, and their generated tests (both in format and coverage) is similar to these produced by other modern test generation tools, such as TestFul [45], JavaPathFinder [46], Pex [47], JCrasher [48], and others. In this work, following existing studies [5, 6, 7, 8], we use code coverage to measure the quality of unit test suites, given the fact that there exists debates on the correlation between code coverage and test effectiveness [49], we plan to examine the effectiveness of unit tests with more criteria.

**External Validity**: In this work, all the experiment subjects are open source projects and written in Java. Although they are popular projects and widely used in both academic research studies and real-world applications, our findings may not be generalizable to commercial projects or projects in other ecosystems. Thus, future research should revisit our study in machine learning models with other languages. To mitigate this threat, we plan to explore the effectiveness of on Python projects in the future. However, note that our findings does not have to rely on language specific features and therefore we believe the findings in this work are still valuable for guiding the unit test practice with other program languages.

## VI. Related Work

### A. Automatic Test Generation

Over the past years, researchers have made many advancements to solve the unit test generation problem. As random testing can be easily scaled to large and complex systems, random test-data generation has been widely explored [50, 48, 51, 52, 53, 3, 4]. JTest [52] is a commercial tool that leveraged random testing to generate test data that meets structural coverage. JCrasher [48] created sequences of method calls for Java programs and reported sequences that throw certain types of exceptions. Eclat [51] and Randoop [3, 4] used random search to create tests that are likely to expose fault. All these tools use random search without enough guidance which could lead to achieving low code coverage.

To improve random testing, search-based algorithms have been leveraged to generate test cases [2, 45, 54]. eToc [55] used genetic algorithms to generate test data to test primitive types and strings. TestFul [45], AutoTest [54], and EvoSuite [2] are typical tools that automated test case generation by using genetic algorithms. Their objective is to reach the maximum coverage for a given criterion (e.g., cover all branches in the system) by using evolutionary algorithms. Symbolic execution is an alternative approach to improve random testing [46, 56, 46], such as JPF-symbc [57] and Pex [47]. Such approaches are not scalable because they cannot deal with complex statements, native function call, or external libraries [58].

Almost all of the existing unit test case generation tools target at producing test cases for general software projects, and studies have already showed that they can help produce unit test suites with high code coverage for general software projects [5, 6, 7, 18]. This work conducts the first study to evaluate the effectiveness of automatic unit test generation tools on machine learning libraries.

### B. Machine Learning Testing

To test machine learning algorithms, metamorphic testing based approaches have been proposed to infer possible "test oracle" of an algorithm to indicate what the correct output should be for different inputs [59, 60, 61, 62, 63, 64, 65]. Murphy et al. [63] discussed the properties of machine learning algorithms that may be adopted as metamorphic relations to detect implementation bugs. Along this line, Xie et al. [59] conducted the first study about leveraging metamorphic testing to test the implementations of two machine learning classification algorithms, i.e., KNN, and Naive Bayes.

Recently, many studies have been conducted to survey new techniques to test machine learning [66, 10, 12, 67, 68, 69, 70, 68]. Hang et al. [66] analyzed the main challenges of testing two machine learning algorithms (i.e., Naive Bayesian classifier and DNN classifier) that perform a classification task. Masuda et al. [67] examined the challenges of software quality assurance for ML-as-a-services (MLaas), which are machine learning services available through APIs on the cloud. Ishikawa et al. [69] discussed the foundational concepts

that may be used in any and all machine learning testing approaches. Ma et al. [71] and Huang et al. [72] surveyed the security of deep learning models. Guo et al. [70] characterized deep learning development and deployment across different frameworks and platforms. Braiek et al. [10] and Zhang et al. [12] reviewed current existing testing practices for machine learning and deep learning frameworks. Results of the above surveys showed that most recent techniques to test machine learning mainly focused on testing deep learning models, e.g., Pei et al. [9] presented the first white-box testing of deep learning models, Tian et al. [73] proposed to utilize fuzz testing to generate more test data for autonomous driving cars, Ma et al. [74] proposed new mutation operators to evaluate the effectiveness of test cases on deep learning models, and Nejadgholi et al. [68] studied the test oracle practice in deep learning libraries.

Prior studies on testing machine learning mainly focus on specific machine learning algorithms or the correctness of generated machine learning models. In this work, we present the first important step to understand how developers perform unit testing in machine learning libraries, which is not studied by prior work.

## VII. Conclusion

This paper conducts the first study to investigate the effectiveness of existing unit test generation techniques on machine learning libraries. To investigate this issue, we have conducted an empirical study on five widely-used machine learning libraries with two popular unit test case generation tools, i.e., `EVOSUITE` and `Randoop`. Our analysis finds that (1) most of the machine learning libraries do not maintain a high-quality unit test suite regarding commonly applied quality metrics such as code coverage (on average is 34.1%) and mutation score (on average is 21.3%), (2) unit test case generation tools, i.e., `EVOSUITE` and `Randoop`, lead to clear improvements in code coverage and mutation score, however, the improvement is limited, and (3) there exist common patterns in the uncovered code across the five machine learning libraries that can be used to improve unit test case generation tasks.

## Acknowledgement

The authors thank the anonymous reviewers for their feedback which helped improve this paper. This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the National Natural Science Foundation of China under grant No.62072442.

## References

[1] P. Runeson, "A survey of unit testing practices," *IEEE software*, vol. 23, no. 4, pp. 22–29, 2006.

[2] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *FSE'11*, 2011, pp. 416–419.

[3] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 815–816.

[4] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 75–84.

[5] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated unit test generation really help software testers? a controlled empirical study," *TOSEM'15*, vol. 24, no. 4, pp. 1–49, 2015.

[6] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *TOSEM'14*, vol. 24, no. 2, pp. 1–42, 2014.

[7] D. Serra, G. Grano, F. Palomba, F. Ferrucci, H. C. Gall, and A. Bacchelli, "On the effectiveness of manual and automatic unit test generation: ten years later," in *MSR'19*, 2019, pp. 121–125.

[8] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated white-box test generation really help software testers?" in *ISSTA'13*, 2013, pp. 291–301.

[9] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *SOSP'17*, pp. 1–18.

[10] H. B. Braiek and F. Khomh, "On testing machine learning programs," *JSS'20*, vol. 164, p. 110542, 2020.

[11] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *ICCV'15*, pp. 2722–2730.

[12] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *TSE'20*, 2020.

[13] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[14] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," in *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, 2014, pp. 55–60.

[15] A. K. McCallum, "Mallet: A machine learning for language toolkit," *http://mallet. cs. umass. edu*, 2002.

[16] J. Baldridge, "The opennlp project," *URL: http://opennlp. apache. org/index. html,(accessed 2 February 2012)*, p. 1, 2005.

[17] D. Lyubimov and A. Palumbo, *Apache Mahout: Beyond MapReduce*. CreateSpace Independent Publishing Platform, 2016.

[18] A. Bacchelli, P. Ciancarini, and D. Rossi, "On the effectiveness of manual and automatic unit test generation," in *ICSEA'08*, 2008, pp. 252–257.

[19] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *TSE'11*, vol. 38, no. 2, pp. 258–277, 2011.

[20] P. McMinn, "Search-based software test data generation: a survey," *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[21] M. Harman and B. F. Jones, "Search-based software engineering," *IST'01*, vol. 43, no. 14, pp. 833–839, 2001.

[22] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012.

[23] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of machine learning*, 2018.

[24] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[25] Y. Kim, "Convolutional neural networks for sentence classification," *arXiv preprint arXiv:1408.5882*, 2014.

[26] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*, 2013, pp. 6645–6649.

[27] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[28] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio, "Theano: new features and speed improvements," *arXiv preprint arXiv:1211.5590*, 2012.

[29] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, 2019, pp. 8024–8035.

[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[31] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *ICSE-SEIP'17*, 2017, pp. 263–272.

[32] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *ASE'15*, 2015, pp. 201–211.

[33] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica: Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.

[34] J. M. Rojas, G. Fraser, and A. Arcuri, "Automated unit test generation during software development: A controlled experiment and think-aloud observations," in *ISSTA'15*, pp. 338–349.

[35] M. M. Tikir and J. K. Hollingsworth, "Efficient instrumentation for code coverage testing," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4, pp. 86–96, 2002.

[36] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *TSE'10*, vol. 37, no. 5, pp. 649–678, 2010.

[37] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *FSE'14*, pp. 654–665.

[38] H. Do and G. Rothermel, "A controlled experiment assessing test case prioritization techniques via mutation faults," in *ICSM'05*, pp. 411–420.

[39] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *ICSE'05*, pp. 402–411.

[40] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *TSE'12*, vol. 38, no. 2, pp. 278–292.

[41] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *FSE'11*, pp. 212–222.

[42] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *ISSTA'16*, 2016, pp. 449–452.

[43] M. Hilton, J. Bell, and D. Marinov, "A large-scale study of test coverage evolution," in *ICSE'18*, 2018, pp. 53–63.

[44] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive mutation testing," *TSE'08*, vol. 45, no. 9, pp. 898–918, 2018.

[45] L. Baresi, P. L. Lanzi, and M. Miraz, "Testful: an evolutionary test approach for java," in *ICST'10*, 2010, pp. 185–194.

[46] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with java pathfinder," pp. 97–107, 2004.

[47] N. Tillmann and J. De Halleux, "Pex–white box test generation for. net," in *International conference on tests and proofs*, 2008, pp. 134–153.

[48] C. Csallner and Y. Smaragdakis, "Jcrasher: an automatic robustness tester for java," *Software: Practice and Experience*, vol. 34, no. 11, pp. 1025–1050, 2004.

[49] G. Gay, M. Staats, M. Whalen, and M. P. Heimdahl, "The risks of coverage-directed test case generation," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 803–819, 2015.

[50] J. H. Andrews, S. Haldar, Y. Lei, and F. C. H. Li, "Tool support for randomized unit testing," in *Proceedings of the 1st international workshop on Random testing*, 2006, pp. 36–45.

[51] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *ECOOP'05*, 2005, pp. 504–527.

[52] "jtest," https://www.parasoft.com/jtest, 2013.

[53] J. H. Andrews, T. Menzies, and F. C. Li, "Genetic algorithms for randomized unit testing," *TSE'11*, vol. 37, no. 1, pp. 80–94, 2011.

[54] A. Leitner, I. Ciupa, M. Oriol, B. Meyer, and A. Fiva, "Contract driven development= test driven development-writing test cases," in *FSE'07*, 2007, pp. 425–434.

[55] P. Tonella, "Evolutionary testing of classes," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 119–128, 2004.

[56] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[57] C. S. Păsăreanu and N. Rungta, "Symbolic pathfinder: symbolic execution of java bytecode," in *ASE'10*, 2010, pp. 179–180.

[58] C. S. Păsăreanu, N. Rungta, and W. Visser, "Symbolic execution with mixed concrete-symbolic solving," in *ISSTA'11*, 2011, pp. 34–44.

[59] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *JSS'11*, vol. 84, no. 4, pp. 544–558, 2011.

[60] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," *arXiv preprint arXiv:2002.12543*, 2002.

[61] T. Y. Chen, J. W. Ho, H. Liu, and X. Xie, "An innovative approach for testing bioinformatics programs using metamorphic testing," *BMC bioinformatics*, vol. 10, no. 1, p. 24, 2009.

[62] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *TSE'16*, vol. 42, no. 9, pp. 805–824, 2016.

[63] C. Murphy, G. E. Kaiser, and L. Hu, "Properties of machine learning applications for use in metamorphic testing," 2008.

[64] C. Murphy, K. Shen, and G. Kaiser, "Using jml runtime assertion checking to automate metamorphic testing in applications without test oracles," in *ICST'09*, 2009, pp. 436–445.

[65] C. Murphy, G. Kaiser, and M. Arias, "Parameterizing random test data according to equivalence classes," in *ASE'07*, 2007, pp. 38–41.

[66] S. Huang, E.-H. Liu, Z.-W. Hui, S.-Q. Tang, and S.-J. Zhang, "Challenges of testing machine learning applications." *International Journal of Performability Engineering*, vol. 14, no. 6, 2018.

[67] S. Masuda, K. Ono, T. Yasue, and N. Hosokawa, "A survey of software quality for machine learning applications," in *ICSTW'18*, 2018, pp. 279–284.

[68] M. Nejadgholi and J. Yang, "A study of oracle approximations in testing deep learning libraries," in *ASE)*. IEEE, 2019, pp. 785–796.

[69] F. Ishikawa, "Concepts in quality assessment for machine learning-from test data to arguments," in *ICCM'18*, 2018, pp. 536–544.

[70] Q. Guo, S. Chen, X. Xie, L. Ma, Q. Hu, H. Liu, Y. Liu, J. Zhao, and X. Li, "An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms," in *ASE'19*, 2019, pp. 810–822.

[71] L. Ma, F. Juefei-Xu, M. Xue, Q. Hu, S. Chen, B. Li, Y. Liu, J. Zhao, J. Yin, and S. See, "Secure deep learning engineering: A software quality assurance perspective," *arXiv preprint arXiv:1810.04538*, 2018.

[72] X. Huang, D. Kroening, M. Kwiatkowska, W. Ruan, Y. Sun, E. Thamo, M. Wu, and X. Yi, "Safety and trustworthiness of deep neural networks: A survey," *arXiv preprint arXiv:1812.08342*, 2018.

[73] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *ICSE'18*, 2018, pp. 303–314.

[74] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao *et al.*, "Deepmutation: Mutation testing of deep learning systems," in *ISSRE'18*, 2018, pp. 100–111.