Multiple-Implementation Testing of Supervised Learning Software

Oreoluwa Alebiosu, Siwakorn Srisakaokul, Angello Astorga, Tao Xie
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{alebios2,srisaka2,aastorg2,taoxie}@illinois.edu

Abstract

Machine learning (ML) software, used to implement an ML algorithm, is widely used in many application domains such as financial, business, and engineering domains. Faults in ML software can cause substantial losses in these application domains. Thus, it is very critical to conduct effective testing of ML software to detect and eliminate its faults. However, testing ML software is difficult, especially on producing test oracles used for checking behavior correctness (such as using expected properties or expected test outputs).

To tackle the test-oracle issue, in this paper, we present a novel black-box approach of multiple-implementation testing for supervised learning software. The insight underlying our approach is that there can be multiple implementations (independently written) for a supervised learning algorithm, and majority of them may produce the expected output for a test input (even if none of these implementations are fault-free). In particular, our approach derives a pseudo-oracle for a test input by running the test input on n implementations of the supervised learning algorithm, and then using the common test output produced by a *majority* (determined by a percentage threshold) of these n implementations. Our approach includes techniques to address challenges in multiple-implementation testing (or generally testing) of supervised learning software: definition of a test case in testing supervised learning software, along with resolution of inconsistent algorithm configurations across implementations. The evaluations on our approach show that our multiple-implementation testing is effective in detecting real faults in real-world ML software (even popularly used ones), including 5 faults from 10 NaiveBayes implementations and 4 faults from 20 k-nearest neighbor implementations.

1 Introduction

The importance of machine learning (ML) has soared in many fields besides computer science. ML software plays a crucial role in marketing, stock trading, heart-failure identification, fraud identification, etc. Given this growing application, the consequences of software failures could have significant implications and unpleasant effects.

For example, in early 2016, Microsoft released Tay, an intelligent chat bot developed to conduct research on conversational understanding through interactions with millions of users in Twitter. However, within hours, Twitter users trained Tay to post offensive comments causing a flurry of uproar forcing it to be pulled from Twitter [1]. In August 2012, the Knight-Capital group lost $440 million within only 4 hours because of a software fault in their trading system [2]. Additionally, faults in ML software commonly exist. An empirical study [3] of faults in ML software shows that a

non-trivial percentage (22.6%) of faults are due to implementations that do not follow the expected behavior.

Software testing remains the most widely used mechanism for software quality assurance; it is very critical to conduct effective testing of ML software to detect faults. However, ML software is known to suffer from the "no oracle problem" [4]. A category of machine learning, supervised learning, learns a classification model from training data (i.e., labeled data) and then applies the classification model to predict the label for a future entry of application data[1]. In the context of supervised learning, a test oracle is not easily obtainable. Future entires of application data can be labeled (manually or automatically); however, using such labels as the test oracle is not feasible. The reason is that there exists some inaccuracy (i.e., predicting a wrong label) in the learned classification model. This inaccuracy is inherent and sometimes desirable to avoid the overfitting problem[2] (i.e., the classification model performs perfectly on the training data but undesirably on future application data).

To tackle the test-oracle issue, we present a novel black-box approach of multiple-implementation testing for supervised learning software. The insight underlying our approach is that there are multiple implementations available for a supervised learning algorithm, and majority of them produce the expected output for a test input (even if none of these implementations are fault-free). In particular, our approach derives a pseudo-oracle for a test input by running the test input on n implementations of the supervised learning algorithm, and then using the common test output produced by a *majority* (determined by a percentage threshold) of these n implementations. To conduct multiple-implementation testing, our approach provides a definition of a test case for testing supervised learning software, along with resolution of inconsistent algorithm configurations across implementations.

This paper makes the following main contributions:

- The first approach of multiple-implementation testing for supervised learning software.
- The first definition of a test case in testing supervised learning software for unifying fundamental concepts from the ML and software testing communities.
- Evaluations for showing that our approach detects real faults in real-world ML software (even popularly used ones), including 5 faults from 10 NaiveBayes implementations and 4 faults from 20 k-nearest neighbor implementations.

The rest of this paper is organized as follows. Section 2 provides illustrating examples. Sections 3 and 4 present our approach and its implementation, respectively. Section 5 presents our evaluations. Section 6 discusses some issues. Section 7 presents related work, and Section 8 concludes.

2 Examples

In this section, we present illustrating examples for multiple-implementation testing.

2.1 Multiple-Implementation Testing

We use an example to illustrate the process of applying our approach on a small sample data set. As shown in Table 1, a specific training data set[3] (with the label of -1 or +1) is fed to 6 different implementations: A, B, C, D, E, F (one of which is the implementation under test, in short as IUT) of the same classification algorithm (e.g., the kNN algorithm) to build 6 different classification models. In software testing, a test case consists of test input (also called test data) and test oracle. Here for testing supervised learning software, the test input in a test case consists of the training data set and an unlabeled entry of the application data. The unlabeled application data entries are also fed to these 6 different classification models (produced with the training data set) to produce the predicted labels. A predicted label from an implementation is the actual label. Note that these data sets can be manually or automatically constructed, or borrowed from some ML benchmark datasets such as the UCI benchmarks [5, 6, 7, 8].

[1] Application data refer to the data whose labels are to be predicted by the classification model.

[2] Overfitting generally occurs when a model is excessively complex, such as having too many parameters relative to the number of observations.

[3] To simplify the illustration, we omit the use of a validation or test data set.

Table 1: Classification Model Predictions from all implementations

App Data Entry ID	A	B	C	D	E	F
1	-1	-1	-1	+1	+1	+1
2	-1	-1	+1	-1	-1	+1
3	-1	-1	+1	-1	-1	-1
4	+1	+1	+1	+1	+1	+1
5	-1	-1	-1	-1	-1	-1
6	+1	+1	-1	+1	+1	+1

```
1. Input: D = (x1, c1),.., (xN, cN)
2. x = (x1,.., xn) new entry to be classified
3. FOR each labelled entry (xi,ci) calculate d(xi,x)
4. Order d(xi, x) from lowest to highest, (i = 1,..,N)
5. Select the K nearest instances to x: D^K_x
6. Assign to x the most frequent class in D^K_x
```

Figure 1: The kNN Algorithm Pseudocode[9]

2.2 Consistent Configuration Parameters

We ensure that each implementation makes use of the same configuration parameters before performing multiple-implementation testing. Below is the kNN algorithm along with the configuration parameters that need to be set for all implementations before performing multiple-implementation testing.

Line 3 from the kNN algorithm pseudocode above includes a step that requires the use of a distance function to calculate distance between $x1$ and x, it provides a variation of the algorithm and is not configurable. In which case, we describe this as an implicit setting. We only perform multiple-implementation testing on multiple implementations with the same implicit and explicit configuration parameter. Line 5 includes a step that selects K, which is an explicit parameter that has to be set during the execution of the implementation. We ensure that all implementations have consistent explicit settings. Each implementation builds a classification model.

Suppose that each classification model makes predictions for 6 application data entries from the application data set as shown in Table 1 with each row representing the predicted labels (i.e actual output) for an entry by a classification model produced by each implementation (denoted as Implementations A to F). The first column of the table denotes the application data entry's unique ID from the application data set.

Each implementation produces a predicted label, treated as test actual output, for each data entry in the application data set. The final output of our approach, as shown in Table 1, is the majority-voted label predicted for an application data entry and the names of implementations producing classification models that predict a label different from the majority-voted label. The majority-voted label predicted for an application data entry is treated as a proxy for the algorithm-expected label for that application data entry. Note that the number of votes for each predicted label for the first application data entry in Table 1 is equally split. In such case, we say that the majority-voted label is "undecidable". We then treat the majority-voted and "undecideable" labels as the correct predictions and add the appropriate implementations that produce classification models with deviated predictions to Table 1.

From Table 1, we can see mis-classifications (i.e., deviate predictions) for both implementation C and implementation F.

3 Approach

To solve the test-oracle issue in testing of supervised learning software, we propose a test case formulation setup for Multiple-Implementation Testing (a black-box approach) for supervised learning software. Our approach runs independently-written programs that follow the same specifications

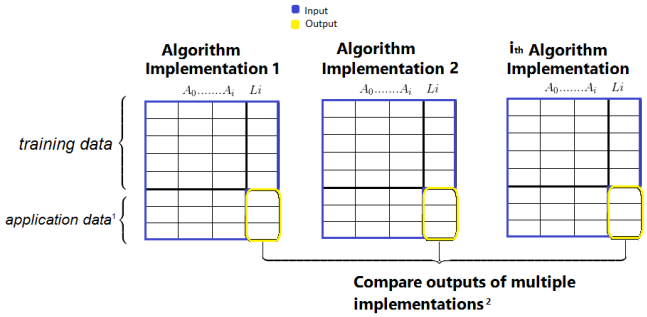

Figure 2: Test Input and Output for Multiple-Implementation Testing of Supervised Learning Software

with the same test input, configurations, and parameters. We derive a pseudo-oracle for a test input by running such input on n implementations of the same algorithm, and then use the majority-voted output (determined by a percentage threshold) of these n implementations as the "expected-algorithm label". Next, we discuss our test case formulation in particular how we generate a test oracle. Then, we describe our application of Multiple-Implementation testing.

3.1 General Framework

We detail how we form a test case for multiple-implementation testing of ML software. To test an implementation (i.e implementation under test- IUT), we must use the same test inputs for the n implementations (independently written, including IUT) of a particular algorithm. Additionally, we test these n-implementation under the same conditions and setting. Then, we derive a pseudo-oracle by using the common test output produced by a majority (determined by a percentage threshold) of these n implementations.

Test Case. A test case entails the combination of test inputs and expected results needed to evaluate the software under test[10]. In multiple-implementation testing, we set a test case to be n implementations of the same specifications, one of which is the implementation under test (IUT). The sequence of steps is to obtain a test input, run every implementation, including the IUT with the given test input, then determine if the IUT produces outputs that deviate from the majority outputs from all the n implementations. If the majority cannot be determined on this test input (the output value that has the high percentage is below the percentage threshold), we will say that the IUT's output does no deviate from the majority output on this test input.

Test Input. In testing supervised learning algorithms, we define test input as the training data set and an unlabeled application[1] entry. We use the same training data set and an unlabeled application[1] entry for all implementations in multiple-implementation testing. Each entry in the training data set has an assigned label.

Test Oracle. The test oracle is a mechanism for determining whether a test has passed or failed. We use test oracles in comparing the output of the IUT, for a given test input, to the output that the oracle determines that implementation should have. The expected output is the output an implementation should have. One or more ML algorithms are implemented in supervised learning software. In supervised learning, the algorithm-expected label is treated as the expected output for a given test input. In multiple-implementation testing of supervised learning software, the majority-voted label is used as a proxy for the algorithm-expected label. A percentage threshold is used to decide the majority-voted label. The actual output, given a test input, is the predicted application entry label of the IUT. A failed test is a condition when the actual output is inconsistent with the expected output. There are three test cases in Figure 2, a test input corresponds to the training data and a single application entry. To create a more test inputs, we create additional application entries. The actual output(s) corresponds to the three application entry labels. In Figure 2, we use the same training data and application entries for all implementations, including IUT.

3.2 Consistent Configuration Parameters

All implementations should be executed under the same conditions, therefore, in addition to giving them the same test input data, we provide the implementations with the same test input configuration parameters. Software may implement one or more configuration options, in which case configurations are said to be implicit. Software may have explicit configuration parameter values that should be set before or during its execution.

Implicit Settings. Prior to performing multiple implementation testing, we group implementations by there different configurations. We perform multiple implementation testing on the group with the same configuration as the IUT. A common configuration is the normalization procedure used by the implementations (e.g., z-score normalization and min-max normalization).

Explicit Settings. Input configuration parameters are algorithm specific (e.g., k in the kNN algorithm). We set the same value for all input parameters including k for all kNN implementations. We performed an empirical study used to understand the symptoms for the deviations and determine whether the deviations are fault symptoms or due to inconsistent configuration parameter values. Due to page limit of this paper, the result of our study is on our project website[5]. Our evaluation results in this paper (RQ1.3 and RQ2.1), also describes how majority-voted oracle perform, in multiple-implementation testing, when all the implementations still have there default (non-modified) implicit and explicit settings, and when all the implementations have consistent implicit and explicit settings. Furthermore, we present results on the effectiveness of our learning-based multiple-implementation monitoring technique in detecting deviations of the implementation under test's predicted class label from the majority-voted class label (when implementation's default (non-modified) implicit and explicit settings are used, and when all the implementations have consistent implicit and explicit settings).

4 Implementation

We have implemented the techniques in the proposed multiple-implementation testing framework so they may be applied on supervised learning algorithms. Our framework partitions different data sets and randomly selected entries from the different partitions. Our framework performed input document and data set transformation so as to use the same input data for all implementations. We run the different implementations with test input. Our framework then computes the majority-voted output with the output of each implementation while also reporting implementations deviating output.

Data set generation. Our benchmark data were obtained from the UCI machine learning repository. This repository is used by the ML community for the empirical analysis of ML algorithms[4]. Furthermore, the data in this repository is representative of real world situations. Our framework creates partitions from the benchmark data. Each partition contains data with a certain equivalence class. Equivalence classes such as small vs. large data sets; missing vs. non-missing attribute values; repeating vs non-repeating attribute values; and a combination thereof. These equivalence classes can be used to guide the generation of appropriate input data sets [11]. In particular, we chose the Iris, Adult and Poker data sets that created partitions of these data sets so they address different equivalence classes. Finally, our framework randomly selects entries from the partitioned data set. The randomly selected entries from the Iris data set is used as the input data set (training and application) in our evaluation.

Threshold for computing majority-voted output. We set a threshold of 0.5 in our test result analysis, i.e., The majority-voted label L is treated as the expected output, if at least half of the implementations produce the same L. This heuristic is adapted from previous work of multiple-implementation testing [12]. We ignore test cases (and treat the test result as uncertain) when there are two candidates for the expected output or if we were unable to reach a threshold of 0.5. Our framework obtains and compares the predicted labels for all entries in application data [4] set of each implementation. Our framework then computes the expected output based on the threshold and informs us if the output of the IUT matches with the expected output.

[4]In this paper, Application data may be referred to as validation/test data set in ML community, we address it as application data set to prevent confusion with Test Input, which is used in the traditional software testing.

Data set transformation. Some implementations may have varying input document and data set formats because they were independently-written. In order to run the generated data set on each implementation we have to perform document and data set format transformations. Some commonly used document formats are arff, csv, and libsvm. In addition to creating such documents from the initially generated data sets, some implementations may still require minor changes to the data in the document, e.g., moving the column containing all labels to become first (or last) column in a csv document. Our framework performs the input document and data set format transformation.

5 Evaluations

To assess the effectiveness of our multiple-implementation testing in detecting faults in supervised learning software, we conduct evaluations on 30 open-source projects. In our evaluations, we investigate the following research questions:

- **RQ1**: How does the majority-voted oracle perform compared with an alternate approach (using the actual output from the benchmark dataset as the oracle)?
- **RQ2**: How does the majority-voted oracle perform when all the implementations are faulty?
- **RQ3**: How does the majority-voted oracle perform when all the implementations still have there $default$ (non-modified) explicit parameter settings?

5.1 Evaluation Setup

5.1.1 Evaluation Subjects

We selected popular supervised learning algorithms for our evaluation. A ranked list of supervised learning algorithms was constructed based on the algorithms popularity on the well-known project repository service - Github [13]. We searched for implementations of the several supervised learning algorithms listed on Wikipedia [14] on the repository service website (Github [13]) and ranked each algorithm by the number of implementations available on the website. The keywords used to search for each algorithm are its name, as seen, on Wikipedia [14]. The ranked list is available on our project website [5]. Additionally, we filtered the implementations based on its programming language, selecting only subjects implemented with Java, C#, and Python. This filtering requirement was devised because the researchers possess debugging tools that support these programming languages.

We selected 30 implementations of each of the top two algorithms from our filtered ranked list as our evaluation subjects. In particular, we gathered 30 implementations of the kNN algorithm and 30 implementations of the Naive Bayes algorithm. All the available implementations from Github [13] were selected as our evaluation subjects, in addition, to reach the aim of 30 implementations, we searched the algorithm keyword found on Wikipedia [14] (plus the word 'implementation') on Google.com [15]; selecting the top indexed results that included an implementation of the searched algorithm.

Of the 30 kNN implementations, we were unable to run 10 as the execution of such implementations required non-trivial code changes to its source code in order to run them with our datasets. For example, one of our subject (kNN2) accepts only input datasets with two class labels (i.e., it accepts only binary datasets). Our black-box approach maintains the integrity of the implementation, and for evaluation purposes our source code modification was to only set the explicit parameter k for kNN implementations. To perform multiple-implemenation testing for kNN implementations, we ran our evaluation with explicit parameter k as 1. Of the 30 NaiveBayes implementations, 20 implementations required non-trivial code changes to its source code in order to run them with our datasets.

Our evaluation subjects also include three ML packages found from Google.com[15]: Weka, RapidMiner, and KNIME. Each of these three packages contains implementations for both the kNN and Naive Bayes algorithm. Our final evaluation subjects include 10 implementations for Naive Bayes and 20 implementations for kNN. The implementations included in our evaluation subjects range from well-tested applications developed by industry professionals to student projects, with varying

[5]The project website: https://sites.google.com/site/multimptest/

input data set formats and varying output formats. The programming languages used in the implementations cover Java, C#, and Python. Note that our multiple-implementation testing technique and the developed framework can be generalized for any language. Columns Algorithm and Implementation in Table 2 contain information about the implementations in our evaluation subjects. Detailed information of these implementations can be found on our project website[6].

In the evaluation, we treat each implementation in our evaluation subjects as the implementation under test (IUT) one at a time. We apply our multiple-implementation testing approach on each IUT using the UCI dataset mentioned in Section 4. Note that the application data entries are selected from the UCI dataset. The class labels for these selected entries are removed in our application data. Due to space limit, we report the evaluation result on the Iris dataset in this paper. Please refer to our project website for the evaluation results on the Adult and Poker dataset.

5.1.2 Fault Detection

To assess the effectiveness of our multiple-implementation testing approach in detecting faults in machine-learning applications, we measure the number of IUT that fail the test (i.e., the actual output is inconsistent with the expected output). We also report the number of faults in these IUT. These faults are detected in the diagnosis phase of our multiple-implementation testing approach.

For RQ1, we compare the majority-voted oracle for each entry of the application data to the expected labels of that entry. Note that the expected labels are obtained from the UCI benchmark dataset. We also compare each majority-voted oracle with the corresponding algorithm expected label.

5.2 Evaluation Results

5.2.1 Fault Detection

Among the 30 implementation under tests in our evaluation subjects, 14 implementations have failing test cases. Column **#Failure** in Table 2 indicates the number of failing test cases for each implementation under test. Among these 14 implementations, our approach can uncover 9 previously unknown faults. In particular, we detect 5 faults for the Naive Bayes implementations and 4 faults for the kNN implementations. Column **#Fault** in Table 2 indicates the number of detected faults for each implementation under test. We file a bug report for each implementation involved. The report includes solution as to how to fix the uncovered fault.

The fault detection capability of our multiple-implementation testing approach can depend on whether the majority-voted oracle (an approximation of the algorithm expected label) for an application data entry is consistent to the expected label for that entry. Our evaluation reveals that 95% of the majority-voted oracles are consistent with the expected labels. Please refer to our project website for more detailed results. Overall, using majority-voted oracles as pseudo test oracles can achieve consistent test results as using the expected labels or algorithm expected label for most of the cases.

For RQ2, we are interested to see how the majority-voted oracle would perform if we do not have any fault-free implementations in our multiple implementations. To see this, we compare both of the faulty majority-voted oracle (without fault-free implementations) and the majority-voted oracle for each application data entry to the expected prediction (ground-truth) of that entry by running the experiment on the kNN algorithm and NaiveBayes algorithm with the Iris dataset, and count the number of correct predictions (the predictions that match the expected predictions). Note that the majority-voted oracle is an approximation of the algorithm expected label. The result is shown in Table 3. Note that For NaiveBayes, there are four faulty implementations out of 10 implementations. For kNN, there are seven faulty implementations out of 20 implementations. We notice that for KNN, even though we remove all the fault-free implementations, the faulty majority-voted oracle has 96.67% accuracy, which is the same as the majority-voted oracle's accuracy. One reason is that there are not many overlappings of failing tests. The majority of the predictions are still correct. However, the faulty majority-voted oracle for NaiveBayes is less accurate, 76.67%, which is lower than that of the majority-voted oracle (93.33%). The reasons that faulty majority-voted oracle does not work well on this case are that we have only four implementations (without fault-free implementations) for NaiveBayes, and three of the four implementations output incorrect predic-

[6]The project website: https://sites.google.com/site/multimptest/

Table 2: Evaluation Results on Subjects

Algorithm	Implementation	#Failure	#Fault	Acc(%)
kNN	kNN1	3	0	90.00
	kNN3	20	2	96.67
	kNN4	1	0	96.67
	kNN5	0	-	-
	kNN6	0	-	-
	kNN8	10	1	93.33
	kNN12	8	0	73.33
	kNN15	1	0	96.67
	kNN16	20	1	96.67
	kNN18	0	-	-
	kNN22	1	-	96.67
	kNN23	0	-	-
	kNN24	0	-	-
	kNN26	0	-	-
	kNN27	1	0	96.67
	kNN28	0	-	-
	kNN29	1	0	96.67
	Weka	0	-	-
	RapidMiner	0	-	-
	KNIME	0	-	-
NB	NaiveBayes1	10	2	60.00
	NaiveBayes5	4	1	86.67
	NaiveBayes13	10	1	73.33
	NaiveBayes17	0	-	-
	NaiveBayes18	7	1	76.67
	NaiveBayes19	0	-	-
	NaiveBayes21	0	-	-
	Weka	0	-	-
	RapidMiner	0	-	-
	KNIME	0	-	-

Table 3: The number of correct predictions of from the majority-voted oracle and the faulty majority-voted oracle.

Prediction	Algorithm	
	kNN	**NaiveBayes**
The majority-voted oracle		
Correct	29	28
Incorrect	1	2
%Correct Prediction	96.67%	93.33%
The faulty majority-voted oracle		
Correct	29	23
Incorrect	1	7
%Correct Prediction	96.67%	76.67%

tions. This implies that the multiple implementation oracle works reasonably well, when we have non-trivial number of implementations regardless of the number of fault-free implementations. The higher number of implementations we have (even though they are all faulty), the more effective the majority-vote oracle becomes. More statistics about the failing tests for each faulty implementations can be found on our project website [5].

For RQ3, we are interested to see how the majority-voted oracle would perform if we do not modify the explicit parameters of implementations when performing multiple-implementation testing. We aimed to investigate the impact of default (i.e non-modified) explicit parameters on our multiple-implementation testing. Table 2 presents evaluation subjects and results for multilple-implementation testing with modified configuration parameters. The table with evaluation results and subjects for multilple-implementation testing with default explicit parameters can be found on our project website [5]. A summary of our result is described here: (1) With our 20 kNN subjects, 11

of them required `k` as an explicit parameter before running the algorithm (i.e an exception is thrown if the input arguments are not set). Four of the subjects had a default value of '3'. Two of the subjects had value of '1'. There were two outliers, kNN4 with default value '10' and kNN6 with default value '90'. Keep in mind, when performing multiple-implementation testing in our approach (with consistent explicit parameter), we set the value of `k` to be 1 for all the implementations. The output for all subjects with their default explicit parameters are the same except with kNN6 which has an outlier value of `k` of '90'. Only 56.7% of the output label of the $default$ kNN6 is the same as the $modified$ kNN6. The majority-voted oracle produced by the subjects is the same when multiple-implementation testing is performed with $default$ explicit parameters and when it is performed with $modified$ explicit parameters. The NaiveBayes algorithm does not require any explicit parameter.

6 Discussion

Multiple-implementation testing only assumes that a "majority" of the implementations are correct for a given test input so there is not a guarantee that they are indeed correct. This problem is inherent to the general approach of multiple-implementation testing. Another issue is on the nature of multiple-implementation testing, of which has to do with the problem or cost of obtaining more than one implementation of a specification. Therefore, a concern is whether developers are able to obtain multiple (independently written) implementations in order to test a particular implementation. There are many implementations such as smaller open source projects, likewise larger ML packages (e.g., Weka, KNIME), and also ML libraries (e.g., scikit in python) that can used for multiple-implementation testing.

7 Related Work

Differential Testing [16] is a testing approach closely related to multiple implementation testing. During differential testing, developers would like to generate tests that exhibit the behavioral differences between two versions, if any differences exist, e.g., regression testing. As such, if developers choose a specific implementation as a reference implementation, then they are not doing multiple-implementation testing but just doing differential testing or testing against the reference implementation. In multiple-implementation testing, all implementations are treated equally and each places an equal vote to the test output.

Murphy and Kaiser [4] proposed an approach for testing ML applications based on metamorphic testing, parameterized random testing, and niche oracle based testing. Their approach conducts a set of analyses on the problem domain, the algorithm as defined, and runtime options. From the analyses, they derive equivalence classes to guide the aforementioned testing techniques.

Further related work includes the investigation of applying metamorphic testing to different domains such as testing epidemiological models by Pullum [17] for the verification and validation of disease spread models and testing of phylogenetic inference programs by Sadi [18] whereby the approach is used to test models that predict the evolutionary history of species. In addition, metamorphic testing has been investigated on specific ML algorithms such as kNN and NB [19]. Groce et al. [20] proposed test selection techniques that provide very good failure rates for end user interactive ML systems. This research focused on the problem of testing machine-generated programs when there exists an oracle, which is an end user.

Multiple-implementation testing has been used for non-ML subjects, e.g., in detecting faults in XACML implementations [12], web input validators [21], and cross-browser issues [22]. By using multiple-implementation testing, we detected faults in ML software as described in our evaluation section.

8 Conclusion

In this paper, we have proposed a novel black-box approach of multiple-implementation testing for supervised learning software. Our approach includes techniques to address challenges in multiple-implementation testing: the definition of test case, resolution of inconsistent algorithm configurations across implementations. The evaluation results show that our approach is effective in detecting

real faults in real-world supervised learning software (even popularly used ones): 5 faults from 10 NaiveBayes implementations and 4 faults from 20 k-nearest neighbor implementations.

References

[1] K. Leetaru, "How twitter corrupted microsoft's tay: A crash course in the dangers of ai in the real world," 2016.

[2] L. Muehlhauser and B. Hibbard, "Exploratory engineering in artificial intelligence," *Communications of the ACM*, vol. 57, no. 9, pp. 32–34, 2014.

[3] F. Thung, S. Wang, D. Lo, and L. Jiang, "An empirical study of bugs in machine learning systems," in *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*. IEEE, 2012, pp. 271–280.

[4] C. Murphy and G. E. Kaiser, "Improving the dependability of machine learning applications," 2008.

[5] archive.ics.uci.edu, "Iris Data Set." [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Iris

[6] archive.ics.uci.edu, "Adult data set." [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Adult

[7] archive.ics.uci.edu, "Balloons Data Set." [Online]. Available: https://archive.ics.uci.edu/ml/datasets/balloons

[8] archive.ics.uci.edu, "Poker data set." [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Poker+Hand

[9] Pedro Larranaga, "k-Nearest Neighbor." [Online]. Available: http://biocomp.cnb.csic.es/~coss/Docencia/ADAM/Sample/Sample_Classification.pdf

[10] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. New York, NY, USA: Cambridge University Press, 2008.

[11] C. Murphy, G. Kaiser, and M. Arias, "Parameterizing random test data according to equivalence classes," in *Proceedings of the 2Nd International Workshop on Random Testing: Co-located with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, ser. RT '07. New York, NY, USA: ACM, 2007, pp. 38–41. [Online]. Available: http://doi.acm.org/10.1145/1292414.1292425

[12] N. Li, J. Hwang, and T. Xie, "Multiple-implementation testing for xacml implementations," in *Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications*. ACM, 2008, pp. 27–33.

[13] Github, "Github." [Online]. Available: https://github.com

[14] Wikipedia, "Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/List_of_machine_learning_concepts

[15] Google, "Google." [Online]. Available: https://google.com

[16] T. Xie, K. Taneja, S. Kale, and D. Marinov, "Towards a framework for differential unit testing of object-oriented programs," in *Proceedings of the Second International Workshop on Automation of Software Test*. IEEE Computer Society, 2007, p. 5.

[17] L. L. Pullum and O. Ozmen, "Early results from metamorphic testing of epidemiological models," in *BioMedical Computing (BioMedCom), 2012 ASE/IEEE International Conference on*. IEEE, 2012, pp. 62–67.

[18] M. S. Sadi, "Testing and fault localization of phylogenetic inference programs using metamorphic technique."

[19] X. Xie, J. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Application of metamorphic testing to supervised classifiers," in *Quality Software, 2009. QSIC'09. 9th International Conference on*. IEEE, 2009, pp. 135–144.

[20] A. Groce, T. Kulesza, C. Zhang, S. Shamasunder, M. Burnett, W.-K. Wong, S. Stumpf, S. Das, A. Shinsel, F. Bice *et al.*, "You are the only possible oracle: Effective test selection for end users of interactive machine learning systems," *Software Engineering, IEEE Transactions on*, vol. 40, no. 3, pp. 307–323, 2014.

[21] K. Taneja, N. Li, M. R. Marri, T. Xie, and N. Tillmann, "Mitv: multiple-implementation testing of user-input validators for web applications," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 131–134.

[22] S. R. Choudhary, H. Versee, and A. Orso, "Webdiff: Automated identification of cross-browser issues in web applications," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–10.