Multiple-Implementation Testing of Supervised Learning Software

Siwakorn Srisakaokul, Zhengkai Wu, Angello Astorga, Oreoluwa Alebiosu, Tao Xie
University of Illinois at Urbana-Champaign
{srisaka2,zw3,aastorg2,alebios2,taoxie}@illinois.edu

Abstract

Machine Learning (ML) algorithms are now used in a wide range of application domains in society. Naturally, software implementations of these algorithms have become ubiquitous. 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, partly because producing test oracles used for checking behavior correctness (such as using expected properties or expected test outputs) is challenging. In this paper, we propose an approach of multiple-implementation testing to test supervised learning software, a major type of ML software. In particular, our approach derives a test input's proxy oracle from the majority-voted output running the test input of multiple implementations of the same algorithm (based on a pre-defined percentage threshold). Our approach reports likely those test inputs whose outputs (produced by an implementation under test) are different from the majority-voted outputs as failing tests. We evaluate our approach on two highly popular supervised learning algorithms: k-Nearest Neighbor (kNN) and Naive Bayes (NB). Our results show that our approach is highly effective in detecting faults in real-world supervised learning software. In particular, our approach detects 13 real faults and 1 potential fault from 19 kNN implementations and 16 real faults from 7 NB implementations. Our approach can even detect 7 real faults and 1 potential fault among the three popularly used open-source ML projects (Weka, RapidMiner, and KNIME).

Introduction

Machine Learning (ML) algorithms are now used in a wide range of application domains in society, such as marketing, stock trading, heart-failure identification, and fraud identification. Given such growing applications, faults in ML software can cause substantial losses in these application domains. However, faults in ML software commonly exist. An empirical study (Thung et al. 2012) of faults in ML software shows that a non-trivial percentage (22.6%) of faults are due to the implementations that do not follow the expected behavior.

To detect faults in ML software, software testing remains the most widely used mechanism, focusing on two major issues: test generation (i.e., generating sufficient test inputs) and test oracle (i.e., determining whether executing the program under test with the generated input produces the expected behavior). Generally, when the output of ML software is not as expected, there can be multiple likely reasons: (1) the training data is not sufficient; (2) the configuration of the algorithm is not desirable; (3) the implementation of the algorithm is faulty; (4) the design of the algorithm is undesirable. Our work in this paper focuses on test oracles for testing the implementation of ML software to detect the third preceding cause.

ML software is known to suffer from the "no oracle problem" (Murphy and Kaiser 2008). Supervised learning constructs a classification model from training data (i.e., labeled data) and then applies the classification model to predict the label for a future instance of unlabeled data. In the context of supervised learning, a test oracle is not easily obtainable. A future instance of data can be labeled (manually or automatically); however, using such label as the test oracle is not effective. One 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 (i.e., the classification model performs perfectly on the training data but undesirably on a future instance of unlabeled data).

To tackle the test-oracle problem for ML software, we present a novel black-box approach of multiple-implementation testing (Li, Hwang, and Xie 2008; Taneja et al. 2010; Choudhary, Versee, and Orso 2010) for supervised learning software. The insight underlying our approach is that there are multiple implementations available for a supervised learning algorithm, and the 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 proxy oracle for a test input by running the test input on n implementations of the same supervised learning algorithm, and then using the common test output produced by a *majority* (determined by a predefined percentage threshold) of these n implementations. Our empirical investigation shows that using majority voting effectively approximates a test oracle.

This paper makes the following main contributions:

- A novel approach of multiple-implementation testing for supervised learning software.

- Empirical evaluations showing that our approach detects faults in real-world ML software. In particular, our approach detects 13 real faults and 1 potential fault from 19 k-Nearest Neighbor (kNN) implementations and 16 real faults from 7 Naive Bayes (NB) implementations. Our approach can detect 7 real faults and 1 potential fault even among the three popularly used open-source ML projects (Weka, RapidMiner, and KNIME).
- Empirical comparison between the majority-voted oracle produced by our approach and the benchmark-listed oracle (i.e., using the labels from benchmark data sets as expected outputs).

Testing Fundamentals for Supervised learning Software

In this section, we first define terminology used to explain testing of supervised learning software. Then, we present an example to illustrate our multiple-implementation testing approach.

Terminology

Multiple-implementation testing. Multiple-implementation testing (Li, Hwang, and Xie 2008; Taneja et al. 2010) is a technique for addressing the "no oracle problem" (Murphy and Kaiser 2008). It is based on the insight that multiple implementations of the same functionality may be available to leverage. Some of these implementations can contain different faults leading to unexpected behaviors for particular inputs. However, the same output across a majority of the executed implementations is likely to be correct. Such a majority can be determined by a predefined percentage threshold denoted as pt. For a given input, when the percentage of the implementations sharing the same output is greater than pt, the result is considered the majority output. It can be used as a proxy for the expected output. More details are discussed later in this section.

Test Input. For testing supervised learning algorithms, we define a test input as (1) a tuple of values of parameters denoted as P, (2) a training data set denoted as D, and (3) an "unlabeled" testing instance denoted as x', which has not been classified. The training data set D consists of multiple instances, each of which has an assigned class label ($D = \{(x_1, c_1), \ldots, (x_N, c_N)\}$). An instance x_i or x' is a tuple of t attributes: $(a_1, a_2, \ldots, a_t)$. In summary, a test input is:

$$ip = (P, D : \{(x_1, c_1), \ldots, (x_N, c_N)\}, x')$$

For the kNN algorithm, P contains only one parameter, k. The NB algorithm has no parameter, so P is empty.

Usually, either existing or automatically generated data sets (e.g., UCI benchmark data sets (Lichman 2013)) can be used to form test inputs. A data set can be randomly split into sets of training data denoted as D and test data. Then each instance in the test data set forms a testing instance x'. For some machine learning algorithms that require tuning parameters (e.g., the learning rate) of the learned model in the training phase, a validation data set may be needed. For simplicity, we omit the use of a validation data set in our setting. Neither kNN nor NB requires a validation data set.

Test Oracle and Proxy Oracle. A test oracle (Baresi and Young 2001) is a mechanism for determining whether the result of executing an implementation passes or fails. For a given test input and an implementation under test (IUT), we compare the output of executing the IUT against the expected output, which is determined by the oracle. Testing supervised learning software faces the "no oracle problem" (Murphy and Kaiser 2008). In other words, it is challenging to come up with an algorithm-expected output with 100% confidence. Thus, in practice, one can use a *proxy oracle*, a mechanism that determines the algorithm-expected output with a certain confidence level of but not equal 100%.

There exist other alternatives that can be used as a proxy oracle. In particular, one can create test inputs from the existing benchmark data sets as discussed earlier when defining test inputs. The benchmark-listed class label for each testing instance x' in a test input can be used as the expected output. In this case, the proxy oracle is named as Benchmark-listed Oracle (in short as Bench-Oracle).

In multiple-implementation testing of supervised learning software, the majority output, selected among all the outputs from multiple implementations, is considered as the expected output. In this case, the proxy oracle is named the Majority-voted Oracle (Major-Oracle for short). Formally, let $imp_i(ip)$ denote the output from running imp_i on the input ip, and $Majority$ is a function that takes a list of outputs and returns the majority output according to the predefined percentage threshold. Therefore, Major-Oracle is defined as below:

$$MajorOracle(ip) = Majority(imp_1(ip), \ldots, imp_n(ip))$$

Test Case. A test case (Ammann and Offutt 2008) consists of a test input and the expected output determined by the test oracle (or proxy oracle). For example, for multiple-implementation testing, a test case is defined as below:

$$tc = \{P, D, x', MajorOracle(P, D, x')\}$$

Failing Test and Deviating Test. A *failing* test case (failing test for short) of the implementation under test is a test case whose actual output of the implementation is different from the algorithm-expected output, which is determined by the test oracle.

We also define a *deviating* test case (deviating test for short) of the implementation under test as a test case whose actual output of the implementation is different from the expected output determined by a proxy oracle (e.g., Bench-Oracle and Major-Oracle). Note that when the expected output determined by a proxy oracle is different from the algorithm-expected output determined by a test oracle, a deviating test may not necessarily be a failing test. Similarly, a failing test may not necessarily be a deviating test if the proxy and test oracles are in agreement.

In some cases of multiple-implementation testing, there may not exist a proxy oracle. Such cases occur when the number of votes for each distinct label (e.g,. output of the implementation under test) do not surpass the predefined

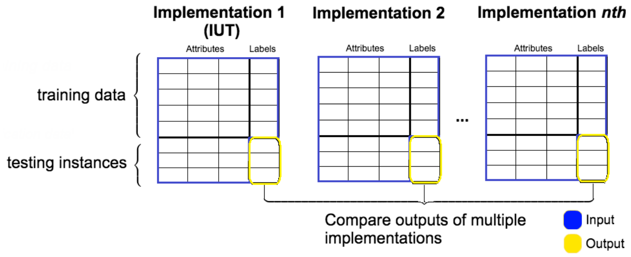

Figure 1: Test Case for Multiple-Implementation Testing

threshold. Therefore, a majority-voted label cannot be derived. In such cases, we assign a special value "undecided" as the majority-voted label. Then a particular test case is a deviating test on an implementation if the actual output label of this test case is different from the majority-voted label, and the majority-voted label is **not** "undecided".

Example. Figure 1 shows an example test case used in our multiple-implementation testing. There are n implementations of the same algorithm. Implementation 1 is the IUT. There are three test cases where each testing instance is listed as one of the three data entries at the bottom of the column. We use the same training data, testing instances, and parameter values for all implementations, including the IUT when applying our approach. The three majority-voted labels for all the three test inputs are determined by comparing outputs from these n implementations with the predefined percentage threshold of 50%.

Approach

Our multiple-implementation testing approach runs n implementations of the same supervised-learning algorithm with the same test input and algorithm configurations. Then, it derives and uses the majority-voted output of these n implementations as the expected output. In particular, our approach includes four main steps:

- **Step 1:** Determine parameters for the Implementation Under Test (IUT)
- **Step 2:** Collect multiple implementations
- **Step 3:** Collect data sets and create test inputs
- **Step 4:** Run test inputs on the multiple implementations to detect deviation

Overall, there are two scenarios of performing multiple-implementation testing: testing an implementation under test (IUT) and testing all the collected implementations. Both scenarios share these four steps with some small variations in details. We next discuss these four steps in the first scenario, and then discuss variations on these steps in the second scenario.

Scenario 1: Testing an Implementation Under Test (IUT)

In this scenario, we perform multiple-implementation testing on the given IUT of an algorithm in order to detect faults in the IUT.

Step 1. The algorithm that the IUT implements may have user-defined parameters, and thus the IUT must have an API that allows to set the values for these parameters. Moreover, the IUT may have additional required parameters (beyond the algorithm parameters) that are needed in order to be run. We have to consider those parameters as well. The IUT documentation may specify the ranges of those parameters. If not, we can inspect the algorithm pseudocode and figure out the ranges.

For example, according to the Wikipedia (Wikipedia 2017), parameter k in kNN usually starts from 1; however, there is no clear definition on the upper bound of k. With deep understanding of the kNN algorithm, one can assume that the upper bound of k is the number of training instances, since we want to find the k nearest instances in the algorithm. Considering Weka's kNN implementation as the IUT, we find no additional required parameters for the IUT in order to run it, since the documentation of Weka indicates an kNN object constructor that takes only k as the parameter (i.e., `new IBk(int k)`).

Step 2. We collect multiple implementations of the same algorithm by searching for implementations on well-known project repositories (e.g., GitHub). For the kNN algorithm, we can use the name of the algorithm as the keywords for searching, such as "kNN" and "k nearest neighbor". Then we can filter out and collect only the implementations whose set of parameters (in their API or setter methods) is the superset of the set of the IUT parameters found in Step 1. For an implementation with required parameters that cannot be set to equivalent values as in the IUT (there are no setter methods for them or the documentation does not state that they are already hard-coded), we discard this implementation, since we cannot run the IUT with different parameter values.

Step 3. Collect a number of data sets from existing online repositories. Usually a data set's documentation states what types of machine learning tasks the data set is suitable for. For example, in our evaluations, we collect data sets that are made for classification algorithms from the UCI repository (Lichman 2013) to test kNN and NB, which are classification algorithms. After that, there are two tasks that may be needed as shown below:

- **Data Transformation.** Some implementations may have different input formats because they were independently-written. In order to run them with our collected data sets, we may need to transform the data-set format. Some commonly used formats are *arff*, *csv*, and *libsvm*. In addition, some implementations may still require minor changes to the data set file, e.g., moving the column containing class labels to the first (or last) column in a csv file or renaming class labels.

- **Generating Test Inputs.** We can randomly split a benchmark data set into two sets of instances: the training data set and test data set. In our approach, the ratio of the number of training instances to the number of testing instances is $4 : 1$. Then, each instance in the test data set is a test input. In our evaluations, neither kNN nor NB needs validation data sets. Note that we can also employ an automatic test generation tool to generate testing instances without labels.

Step 4. Run all the implementations on our data sets. For each test input, we determine its majority-voted output. If the multiple implementations' outputs vary a lot, we may need to add more implementations or decrease the percentage threshold so that we do not have too many "undecided" labels. In our evaluations, we set the percentage threshold to be 50%. This heuristic threshold is adapted from previous work of multiple-implementation testing (Choudhary, Versee, and Orso 2010; Li, Hwang, and Xie 2008). Next, we determine the deviating tests for the IUT, and debug each deviating test by tracing the execution to find faults in a similar way as how we usually do when debugging a faulty program. Note that a test input can contain parameters that we have to supply parameter values. Thus, we use the information about the ranges of parameters (from Step 1) for both functional testing and robustness testing. For example, in kNN, we may set the parameter k to be 1, n (the number of training instances), or -1 (out of range).

Scenario 2: Testing all implementations

In this scenario, given a set of implementations of the same algorithm, we test each of them. The approach is still quite similar to the first scenario. The differences are listed below:

- In **Step 1**, we have to determine the required parameters of each implementation.
- In **Step 2**, using the results from Step 1, we can partition the implementations into several partitions so that each partition contains all the implementations that have the same set of required parameters. Then we perform **Step 3** and **Step 4** on each partition separately.

Evaluations

To assess the effectiveness of our multiple-implementation testing in detecting faults in supervised learning software, we conduct evaluations on 23 open-source projects. In these projects, 3 are among the most popularly used open-source ML projects: Weka, RapidMiner, and KNIME. From these 23 open-source projects, we test 19 implementations of the k-nearest neighbor (kNN) algorithm and 7 implementations of the Naive Bayes (NB) algorithm. In our evaluations, we investigate the following research questions:

- **RQ1:** How effectively can Majority-voted Oracle (Major-Oracle) help detect faults?
- **RQ2:** How more effectively can Major-Oracle help detect faults compared to Benchmark-listed Oracle (Bench-Oracle)?
- **RQ3:** How does the choice of data sets impact fault-detection effectiveness when using Major-Oracle or Bench-Oracle?

Note that although we empirically compare our approach (i.e., Major-Oracle) and Bench-Oracle, Bench-Oracle is **not** applicable when testing instances are automatically generated instead of being drawn from benchmark data whereas our approach is applicable in such setting.

Evaluation Setup

Evaluation Subjects We select the kNN algorithm and the NB algorithm because these two algorithms are among the most popular supervised learning algorithms and they are well documented. We conduct the evaluations by following Scenario 2. To select the implementations for our evaluations, we search for implementations of the kNN and NB algorithm on the well-known project repository service - GitHub (2016). Keywords "K Nearest", "kNN", and "Naive Bayes" are used as search queries on GitHub. From our kNN queries, most of the implementations use the Euclidean-distance metric. After partitioning all the implementations according to the distance metric, we decide to evaluate only the implementations that use the Euclidean-distance metric, since other partitions contain only few implementations. It turns out that there are 5 C# and 11 Java kNN implementations that use the Euclidean distance and that we are able to run. For NB, there are 4 Java implementations that we are able to run and work with a training data set containing more than two class labels. Furthermore, we add implementations from the three popular open-source ML projects: Weka, RapidMiner, and KNIME since they are likely well maintained and likely fault-free. Thus, there are 19 kNN and 7 NB implementations in total that we use as our evaluation subjects.

To perform multiple-implementation testing for kNN implementations, we run our evaluations by setting the value of the parameter k to 1. For NB, we do not identify any parameters so those implementations are run as they are.

Data-set generation We obtain our benchmark data from the UCI machine learning repository (Lichman 2013). This repository is used by the ML community for empirical analysis of ML algorithms. Furthermore, the data in this repository are representative of real-world situations. Our approach creates partitions from the benchmark data. In the evaluations, we treat each implementation in our evaluation subjects as the implementation under test (IUT) one at a time. For testing kNN implementations, we apply our approach on each IUT using 3 data sets from the UCI data sets: Iris (1988), Breast Cancer Wisconsin (BCW) (1992), Glass Identification (Glass) (1987). For testing NB implementations, we also use 3 data sets from the UCI data sets: Breast Cancer Wisconsin (BCW) (1992), Haberman's Survival Data (Haberman) (1999), Hayes-Roth (Hayes) (1989). These data sets are used for testing classification algorithms. We do not use the same three data sets for both algorithms since the Glass and Iris data sets contain some attributes that are floating numbers, so the values in those attributes are unique, not suitable for testing the NB algorithm. The class labels in the data sets are used as benchmark-listed labels. To make the ratio between the numbers of training and test instances 4:1, we randomly choose one fifth of all instances in a data set to form a test data set. The remaining instances are used to form a training data set. In the Iris data set, 30 out of 150 instances are randomly chosen to form a test data set. In the BCW data set, 137 out of 676 instances are randomly chosen to form a test data set. In the Glass data set, 44 out of 211 instances are randomly chosen to form a test data set. In the Haberman data set, 61 out of 306 instances are randomly chosen to form a test data set. In the Hayes data set, 27 out of 132 instances are randomly chosen to form a test data set.

Fault Detection To assess the effectiveness of our approach in detecting faults in ML applications, we measure the number of deviating tests for each IUT (i.e., the actual output is different from the expected output) according to Major-Oracle and Bench-Oracle, respectively. Furthermore, we count the number of deviating tests that lead to a real fault. We also report the number of real faults in these IUTs. These real faults are determined by us by tracing the execution and manually debugging each deviating test to find unexpected behaviors in the IUT. Then we attempt to fix the identified faults and rerun the deviating test and confirm that it becomes a passing test while previously passing tests still pass.

Evaluation Results

Table 1 shows the fault detection results on each implementation for both algorithms. Column **#Faults** indicates the number of real faults revealed from the deviating tests for each implementation under test. We also create a bug report for each implementation involved. The report includes a solution as to how to fix the reported faults. More detailed results can be found on our project website (Srisakaokul et al. 2017).

RQ1: Effectiveness of Major-Oracle in Fault Detection We intend to investigate the effectiveness of Major-Oracle in detecting faults. We investigate all the deviating tests and see how many of them actually reveal a real fault. Table 1 shows that 20.5% of the tests are deviating tests, and almost all the deviating tests (97.5%) based on Major-Oracle reveal a fault. According to both proxy oracles, 10 out of 26 implementations do not have any deviating tests.

Table 2 shows the distribution of faults in the kNN algorithm. Column **#FaultRevealingTests** indicates how many tests reveal the corresponding faults in the kNN implementations. 41% of all the fault-revealing tests reveal that normalization is the main cause of the deviations/failures. For example, the normalize method in kNN13 returns `NaN` on some tests. The implementation kNN8 has a fault due to distance sorting: when $k = 1$, it returns the class label of the first instance in the training data set without sorting all the instances by distance first. One fault that our approach detects in Weka is that Weka does some preprocessing on instances before running its kNN. Although we do not classify this fault as a real fault, we classify it as a potential fault because Weka developers make an “unpopular” design decision, different from most of other implementations for the same algorithm. Such behavior shall draw the attention of Weka developers or even users to assess whether such design decision is a desirable one.

We also categorize the types of faults detected by our approach in the NB implementations into three categories as shown in Table 3. The main fault is that different implementations calculate the probability differently: we find that only NB1 and RapidMiner calculate the probability in the standard NB algorithm. The second fault is the case that all classes have zero probability to contain the given instance. The NB algorithm does not clearly define how to handle this case. Different implementations seem to handle this case differently. For example, NB1 sets the probability to contain the given instance for a class c_i to be `1/(#training instances with class label` c_i`)` for the case that it is actually zero. Doing so is incorrect because different classes may have different values. Setting the zero conditional probability to `1/(#training instances with class label` c_i`)`, instead of a constant, is inconsistent across multiple classes (c_i). The third fault is the case that some classes have zero probability to contain the given instance. For this case, the implementations should still output the classes that have the highest probability. Only NB2 and RapidMiner handle this case correctly. The other implementations output an arbitrary class label instead.

RQ2: Effectiveness of Major-Oracle compared to Bench-Oracle Table 1 shows the comparison result between Major-Oracle and Bench-Oracle with respect to effectiveness of detecting real faults. The result shows that both proxy oracles reveal the same number of real faults in kNN implementations, but Major-Oracle reveals two more faults in NB implementations (one fault is in Weka and the other is in KNIME). Moreover, Major-Oracle has lower false positive (the percentage of deviating tests that do not reveal faults) being (100% - 97.5%) = 2.5%, whereas Bench-Oracle has higher false positive being (100% - 48.4%) = 51.6%. So Major-Oracle is a lot more effective than Bench-Oracle. One possible reason is that Bench-Oracle may not reflect the expected behaviors of the algorithm (which is often designed to avoid overfitting and thus has $< 100\%$ prediction accuracy), whereas Major-Oracle often captures the expected behaviors of the algorithm.

RQ3: Impact of Data-Set Choices on Fault-Detection Effectiveness One may wonder how many data sets are needed to test supervised learning software, and whether using only one data set is sufficient to reveal all the faults revealed when all the data sets are used. Therefore, we next investigate how the choice of data sets impacts fault-detection effectiveness when using Major-Oracle or Bench-Oracle. Due to the page limit, we create a table showing the number of faults revealed by each data set on our project website (Srisakaokul et al. 2017). The result shows that a single data set does not reveal all the real faults in some implementations. For example, kNN11 has three real faults in total, but none of the data sets reveals all the three real faults. Moreover, only **Data Set 3 (Glass)** reveals a real fault in Weka. When comparing the effectiveness of the two proxy oracles, we can see that Major-Oracle reveals more real faults than Bench-Oracle when using only the Iris data set or the BCW data set. In addition, Bench-Oracle has false negatives as it does not reveal a real fault in kNN1 and kNN14, when the Iris data set is used, and some real faults in NB implementations are detected by Major-Oracle, but are not detected by Bench-Oracle.

In summary, Bench-Oracle is not effective in detecting faults compared to Major-Oracle, when we do not have sufficient data sets to use for testing. One implication is that

Table 1: Fault Detection Effectiveness of Major-Oracle and Bench-Oracle for kNN and NB Algorithms

Impl	**Major-Oracle**					**Bench-Oracle**				
	DeviatingTests		**FaultRevealTests**		**#Faults**	**DeviatingTests**		**FaultRevealTests**		**#Faults**
	#	**%**	**#**	**%**		**#**	**%**	**#**	**%**	
kNN1	18	8.5%	18	100.0%	1	32	15.2%	11	34.4%	1
kNN2	0	0.0%	0	N/A	0	30	14.2%	0	0.0%	0
kNN3	0	0.0%	0	N/A	0	30	14.2%	0	0.0%	0
kNN4	0	0.0%	0	N/A	0	30	14.2%	0	0.0%	0
kNN5	0	0.0%	0	N/A	0	30	14.2%	0	0.0%	0
kNN6	0	0.0%	0	N/A	0	30	14.2%	0	0.0%	0
kNN7	0	0.0%	0	N/A	0	30	14.2%	0	0.0%	0
kNN8	100	47.4%	100	100.0%	2	98	46.4%	88	89.8%	2
kNN9	0	0.0%	0	N/A	0	30	14.2%	0	0.0%	0
kNN10	28	13.3%	28	100.0%	1	39	18.5%	22	56.4%	1
kNN11	15	7.1%	15	100.0%	3	34	16.1%	14	41.2%	3
kNN12	6	2.8%	6	100.0%	1	34	16.1%	6	17.6%	1
kNN13	211	100.0%	211	100.0%	1	211	100.0%	211	100.0%	1
kNN14	39	18.5%	39	100.0%	2	39	18.5%	26	66.7%	2
kNN15	26	12.3%	26	100.0%	2	36	17.1%	19	52.8%	2
kNN16	0	0.0%	0	N/A	0	30	14.2%	0	0.0%	0
kNN17 (Weka)	10	4.7%	10	100.0%	*1	30	14.2%	8	26.7%	*1
kNN18 (RapidMiner)	0	0.0%	0	N/A	0	30	14.2%	0	0.0%	0
kNN19 (KNIME)	0	0.0%	0	N/A	0	30	14.2%	0	0.0%	0
NB1	23	10.3%	23	100.0%	2	42	18.8%	19	45.2%	2
NB2	27	12.1%	23	85.2%	2	50	22.3%	23	46.0%	2
NB3	160	71.4%	160	100.0%	3	154	68.8%	152	98.7%	3
NB4	14	6.3%	14	100.0%	2	27	12.1%	7	25.9%	2
NB5 (Weka)	6	2.7%	6	100.0%	3	29	12.9%	4	13.8%	2
NB6 (RapidMiner)	4	1.8%	3	75.0%	1	29	12.9%	3	10.3%	1
NB7 (KNIME)	21	9.4%	21	100.0%	3	39	17.4%	19	48.7%	2
Average		**20.5%**		**97.5%**			**21.9%**		**48.4%**	

Bench-Oracle does not reflect how the algorithm is expected to work. Also, using more data sets usually helps reveal more real faults.

Threats to Validity

The threats to external validity primarily include the degree to which the subject programs, faults, or test cases are representative of true practice. Our evaluations use only two algorithms, their open-source implementations, and three benchmark data sets. These threats could be reduced by more experiments on wider types of subjects in future work. The threats to internal validity are instrumentation effects that can bias our results. Faults in our prototype and human investigation might cause such effects. To reduce these threats, we manually inspect the execution traces and provide fix to confirm real faults.

Discussion

Multiple-implementation testing assumes that a "majority" of the implementations is correct for a given test input but there is no guarantee that they are indeed correct. This problem is inherent to the general approach of multiple-implementation testing.

We notice that in some ML algorithms, there are some unspecified behaviors when dealing with some corner cases. The developers for these ML algorithms should pay enough attention to these situations to make it clear what result their implementation should produce. For example, we can find from Table 1 that compared to the KNN implementations, the NB implementations all have a non-trivial number of deviating tests, even among the three popular implementations. The reason for such result is that, for the NB algorithm, we need to calculate the conditional probability $p(x_i|C_i)$ for each attribute x_i under the condition that the result label is C_i for one test data entry x. And for the specific label C_i, if for some attribute x_i, $p(x_i|C_i) = 0$, the probability $p(C_i|x) = p(C_i)/p(x) * \prod p(x_i|C_i)$ will also be 0. Thus, if all labels have the probability $p(C_i|x) = 0$, the NB algorithm cannot determine a proper label for the data entry. The Haberman dataset contains some attributes that never appear in the training data, making the conditional probabilities for all labels to be 0. The NB algorithm can output any arbitrary label. Different implementations handle this case differently. Such factor causes these implementations to have a non-trivial number of deviating tests. However, for the implementation $NB4$, there is a real fault in dealing with 0 probability. The implementation $NB4$ uses a smoothing

Table 2: Fault Categorization for kNN Algorithm

Fault Types	**# Fault RevealTest**	**%**
Normalization	220	41.0%
Support fixed # of attributes	97	18.1%
Sorting distance	95	17.7%
Euclidean-distance calculation	65	12.1%
Preprocessing data	43	8.0%
Loop iteration	14	2.6%
Arithmetic computation overflow	2	0.4%

Table 3: Fault Categorization for NB Algorithm

Fault Types	**# Fault RevealTest**	**%**
Probability calculation	107	42.1%
Some classes with probability 0	97	38.2%
All classes with probability 0	50	19.7%

function designed to adjust the probability to be larger than a small threshold such as 0.01 so that the probability will not get too close to 0. But the function implementation is incorrect, so that the adjusted probability would be much larger than 0 and exceed some non-zero probabilities (however, testing with the three data sets does not reveal this fault; we find this fault during code inspection when investigating what cause the deviating tests in $NB4$).

One may wonder about the effectiveness of multiple implementation testing when there are no deviation-free implementations, i.e., those whose results are the same as the majority output across all test cases. Due to the page limit, we discuss this aspect with empirical results on our project website (Srisakaokul et al. 2017).

Related Work

Differential testing (Xie et al. 2007) 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 of a program, 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 (2008) 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. In addition, metamorphic testing has been investigated on specific ML algorithms such as kNN and NB (Xie et al. 2009). Metamorphic testing adopted in the preceding previous work requires high human cost and skill to formally specify metamorphic properties; in contrast, our approach of multiple-implementation testing does not require any formal specifications.

Tian et al. (2017) proposed an approach for creating a test oracle for automated testing of Deep-Neural-Networks (DNN) driven autonomous cars. Their idea is to leverage metamorphic relations between the car behaviors across different input images to the DNN model. However, defining the relations over-strictly may result in a large number of false positives. Their approach addresses this issue by allowing variations within some error ranges. Their approach cannot be used for a supervised learning algorithm (e.g., kNN) that outputs only one expected value for a given input.

Pei et al. (2017) proposed an approach for automatically testing deep learning systems. Their approach leverages multiple deep learning systems with similar functionalities as cross-referencing oracles. The goal is to find new inputs from the seed inputs that cause different behaviors of those deep learning systems; such different behaviors may reveal incorrect behaviors. Their approach requires those systems to behave differently (since they are trained independently) so that the approach can find new inputs that cause different behaviors. However, our approach expects multiple implementations to behave consistently by training them with the same data set. If an implementation behaves differently from the majority, it is likely to contain a fault.

Multiple-implementation testing has been used for non-ML application domains, e.g., in detecting faults in XACML implementations (Li, Hwang, and Xie 2008), web input validators (Taneja et al. 2010), and cross-browser issues (Choudhary, Versee, and Orso 2010). In this paper, we show that by using multiple-implementation testing, we detect real faults in ML software with high effectiveness.

Conclusion

In this paper, we have proposed an approach of multiple-implementation testing for supervised learning software. Our evaluations on two popular ML algorithms, k-Nearest Neighbor (kNN) and Naive Bayes (NB), have shown that our majority-voted oracle, produced by multiple-implementation testing, is an effective proxy of a test oracle. The majority-voted oracle has low false positives and can detect more real faults than the benchmark-listed oracle. In particular, our approach detects 13 real faults and 1 potential fault from 19 kNN implementations and 16 real faults from 7 NB implementations. Our approach can even detect 7 real faults and 1 potential fault among the three popularly used open-source ML projects (Weka, RapidMiner, and KNIME).

Acknowledgments

This work was supported in part by National Science Foundation under grants no. CCF-1409423, CNS-1434582, CNS-1513939, CNS-1564274, and an award from the Zhejiang University-University of Illinois at Urbana-Champaign Institute Research Program.

References

Ammann, P., and Offutt, J. 2008. *Introduction to Software Testing*. New York, NY, USA: Cambridge University Press, 1 edition.

archive.ics.uci.edu. 1987. Glass Identification Data Set.

archive.ics.uci.edu. 1988. Iris Data Set.

archive.ics.uci.edu. 1989. Hayes-Roth Data Set.

archive.ics.uci.edu. 1992. Breast Cancer Wisconsin (Original) Data Set.

archive.ics.uci.edu. 1999. Haberman's Survival Data Set.

Baresi, L., and Young, M. 2001. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A. http://www.cs.uoregon.edu/~michal/pubs/oracles.html.

Choudhary, S. R.; Versee, H.; and Orso, A. 2010. WEBDIFF: Automated identification of cross-browser issues in web applications. In *Proc. IEEE International Conference on Software Maintenance (ICSM 2010)*, 1–10.

GitHub. 2016. GitHub. https://github.com.

Li, N.; Hwang, J.; and Xie, T. 2008. Multiple-implementation testing for XACML implementations. In *Proc. Workshop on Testing, Analysis, and Verification of Web Services and Applications (TAV-WEB 2008)*, 27–33.

Lichman, M. 2013. UCI machine learning repository. http://archive.ics.uci.edu/ml.

Murphy, C., and Kaiser, G. E. 2008. Improving the dependability of machine learning applications. Technical report, Department of Computer Science, Columbia University.

Pei, K.; Cao, Y.; Yang, J.; and Jana, S. 2017. DeepXplore: Automated whitebox testing of deep learning systems. In *Proc. Symposium on Operating Systems Principles (SOSP 2017)*, 1–18.

Srisakaokul, S.; Wu, Z.; Astorga, A.; Alebiosu, O.; ; and Xie, T. 2017. Multiple-Implementation Testing of Supervised Learning Software – Project Website. https://sites.google.com/site/mltestproj/.

Taneja, K.; Li, N.; Marri, M. R.; Xie, T.; and Tillmann, N. 2010. MiTV: multiple-implementation testing of user-input validators for web applications. In *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*, 131–134.

Thung, F.; Wang, S.; Lo, D.; and Jiang, L. 2012. An empirical study of bugs in machine learning systems. In *Proc. IEEE International Symposium on Software Reliability Engineering (ISSRE 2012)*, 271–280.

Tian, Y.; Pei, K.; Jana, S.; and Ray, B. 2017. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. *CoRR* abs/1708.08559.

Wikipedia. 2017. k-Nearest Neighbors algorithm.

Xie, T.; Taneja, K.; Kale, S.; and Marinov, D. 2007. Towards a framework for differential unit testing of object-oriented programs. In *Proc. International Workshop on Automation of Software Test (AST 2007)*, 5.

Xie, X.; Ho, J.; Murphy, C.; Kaiser, G.; Xu, B.; and Chen, T. Y. 2009. Application of metamorphic testing to supervised classifiers. In *Proc. International Conference on Quality Software (QSIC 2009)*, 135–144.