MULTIPLE-IMPLEMENTATION TESTING OF SUPERVISED LEARNING SOFTWARE

BY

OREOLUWA ALEBIOSU

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Associate Professor Tao Xie
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, this thesis presents a novel black-box approach of multiple-implementation testing for supervised learning software. The insight underlying the 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, the proposed 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 \textit{majority} (determined by a percentage threshold) of these \( n \) implementations. The proposed approach includes techniques to address challenges in multiple-implementation testing (or generally testing) of supervised learning software: the definition of test cases in testing supervised learning software, along with resolution of inconsistent algorithm configurations across implementations. In addition, to improve dependability of supervised learning software during in-field usage while incurring low runtime overhead, The approach includes a multiple-implementation monitoring technique. The evaluations on the proposed approach show that 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, and the proposed technique of multiple-implementation monitoring substantially reduces the need of running mul-
tiple implementations with high prediction accuracy.
ACKNOWLEDGMENTS

I would especially like to thank Professor Tao Xie for serving as my adviser, and for his committed guidance and assistance during the research and preparation of my thesis. I would also like to thank Professor Darko Marinov for providing comments on the initial results from this project in his CS 527 class. Thanks are also extended to my fellow graduate students who contributed their time and effort to this project.
# TABLE OF CONTENTS

| LIST OF TABLES .................................................. | vi  |
| LIST OF FIGURES ................................................ | vii |
| LIST OF ABBREVIATIONS ........................................ | viii|
| CHAPTER 1 INTRODUCTION ....................................... | 1  |
| CHAPTER 2 BACKGROUND ......................................... | 5  |
| 2.1 Machine Learning (ML) ..................................... | 5  |
| 2.2 NaiveBayes Algorithm ...................................... | 6  |
| 2.3 kNN Algorithm .............................................. | 7  |
| CHAPTER 3 EXAMPLES ............................................. | 9  |
| 3.1 Multiple-Implementation Testing .......................... | 9  |
| 3.2 Multiple-Implementation Monitoring ...................... | 11 |
| CHAPTER 4 APPROACH ............................................. | 13 |
| 4.1 General Framework .......................................... | 15 |
| 4.2 Consistent Configuration Parameters .................... | 16 |
| 4.3 Multiple-Implementation Monitoring ...................... | 17 |
| CHAPTER 5 IMPLEMENTATION ...................................... | 19 |
| CHAPTER 6 EVALUATION .......................................... | 21 |
| 6.1 Evaluation Setup .......................................... | 21 |
| 6.2 Evaluation Results ........................................ | 24 |
| CHAPTER 7 DISCUSSION ........................................... | 33 |
| CHAPTER 8 RELATED WORK ....................................... | 34 |
| CHAPTER 9 CONCLUSION .......................................... | 36 |
| REFERENCES ...................................................... | 37 |
LIST OF TABLES

3.1 Classification Model Predictions from all implementations . . 10
3.2 Majority predictions, and implementations with deviating
predictions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
3.3 Multiple-Implementation Monitoring for Implementation C . 12

6.1 Evaluation Results of Fault Detection . . . . . . . . . . . . 24
6.2 The number of correct predictions from the majority-voted
oracle and the majority-voted oracle (without fault-free
implementations), respectively. . . . . . . . . . . . . . . . . . . 25
6.3 NaiveBayes13 Application Data . . . . . . . . . . . . . . . . 28
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>kNN Example</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>The kNN Algorithm Pseudocode [1]</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>Multiple-Implementation Monitoring Tree for IUT</td>
<td>12</td>
</tr>
<tr>
<td>4.1</td>
<td>Multiple-Implementation Testing Framework for Supervised Learning Software</td>
<td>14</td>
</tr>
<tr>
<td>4.2</td>
<td>Test Input and Output for Multiple-Implementation Testing of Supervised</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>Learning Software</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>CSV and LIBSVM Input Formats</td>
<td>20</td>
</tr>
</tbody>
</table>
# LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Deviate</td>
</tr>
<tr>
<td>FP</td>
<td>False Positive</td>
</tr>
<tr>
<td>FN</td>
<td>False Negative</td>
</tr>
<tr>
<td>IUT</td>
<td>Implementation Under Test</td>
</tr>
<tr>
<td>kNN</td>
<td>k-Nearest Neighbor</td>
</tr>
<tr>
<td>ML</td>
<td>Machine Learning</td>
</tr>
<tr>
<td>NB</td>
<td>Naive Bayes</td>
</tr>
<tr>
<td>N</td>
<td>Not Deviate</td>
</tr>
<tr>
<td>RQ</td>
<td>Research Questions</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

The importance of machine learning (ML) has soared in the field of computer science and beyond as ML software, used to implement an ML algorithm, is widely used in many application domains such as financial, business, and engineering domains. ML software is being used in advertisement placement, financial trading, heart-failure identification, fraud detection, etc.

Given the growing importance of ML software in our society, the quality assurance of ML software is also becoming increasingly important. Subtle faults in ML software can remain undetected during quality assurance activities such as testing, and later cause the deployed ML software to fail. A software failure occurs when the delivered service no longer complies with the specifications. A software failure in ML software can lead to substantial losses. For example, algorithmic trading is a common financial field where ML software is used. 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 three popularly used real-world ML software systems (Apache Mahout, Lucene, and OpenNLP) 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, testing ML software is difficult, especially on producing test oracles used for checking behavior correctness (such as using expected properties or expected test outputs). ML software is known to suffer from the “no oracle” problem [4]: there is an absence of a test oracle or it is too expensive to produce or apply the oracle. A sub-category of ML, supervised learning, learns a classification model from training data (labeled data) and then applies the classification model to predict the label for a future application data\(^1\) entry. In the context of supervised learning,

\(^{1}\) Application data refer to the data whose labels are to be predicted by the classification model.
a test oracle is not always obtainable. A future application data entry 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 often desirable to avoid the overfitting problem\(^2\) (i.e., the classification model performs perfectly on the training data but undesirably in future application data).

Existing approaches have tried to address the “no oracle” problem but still have their own limitations. Property-based testing can be used to assert certain properties of ML software e.g., “for a range of certain input, some condition should hold.” Metamorphic testing [5] is an instance of property-based testing. Metamorphic testing requires non-trivial human efforts for writing properties, i.e., having high expectation on the knowledge, skill, and expertise of the property writers in the domains of both property writing and ML. Assertion-based testing requires both test input and expected output, e.g., “assert that the output for a given input is correct”. For example, in supervised learning (a sub-category of ML), it is not uncommon for testers to use validation/test\(^3\) data to perform assertion-based testing. Simply using the validation/test data as “test cases” where expected outputs are the labels in the validation/test data is inadequate in software testing. The validation/test data is generally a small number of test cases, and do not represent the large input space of which the ML software will be later applied. More importantly, supervised learning algorithms are typically designed not to overfit and thus mis-prediction in the test data does not indicate faults or weaknesses in the algorithm or its software implementation.

To tackle the test-oracle issue for supervised learning software, we present a novel black-box approach of multiple-implementation testing and monitoring for supervised learning software. The insight underlying our approach is that there can be multiple implementations available 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 ML algorithm, and then using the common test output produced by a \textit{majority} (determined by a percentage threshold) of these \(n\) implementations.

In our approach, we detail how to form a test case for testing supervised learning software.\(^2\)

\(^2\)Overfitting generally occurs when a model is excessively complex, such as having too many parameters relative to the number of observations (i.e., samples).

\(^3\)Here the term “test data” has a different meaning than the one used in the software testing community as mentioned earlier.
learning software (including multiple-implementation testing). Despite fundamental, defining what a test case is when testing ML software remains an open problem. In the software testing community, a test case consists of a test input\(^4\) and a test oracle (often in the form of assertions used to check against expected properties or expected test output). In the ML community, for supervised learning software, data used to construct and assess classification models (before applying the models on application data in in-field usage) are split into training data, validation data, and test data\(^3\). Training data are used to learn classification models. Validation data and test data are used to assess and compare different models. Thus, it is natural and common for the ML community to define the test input in a test case as a data entry (without its label) in the validation/test data or even in the application data, and the test oracle in the test case as the label for the data entry. However, such definition is undesirable because the label to be predicted is dependent on three parts: the supervised learning software under test, the training data, and the data entry (without its label). In our approach, we define the test input in a test case as two parts: (1) the whole training data and (2) a data entry (without its label) in the validation/test/application data, and the test oracle in the test case as the algorithm-expected label for the data entry. Note that here we use “algorithm-expected label” to differentiate such label from “expected label”, which is used to refer to the label in the labeled data.

To improve dependability of an ML implementation during in-field usage while incurring low runtime cost, our approach includes a multiple-implementation monitoring technique, to complement multiple-implementation testing (which is typically used for offline in-house testing). In particular, during in-field usage of an implementation \(I\) for a classification algorithm, besides applying the classification model produced by \(I\) (based on the given training data) against an application data entry, we also apply the classification models produced by other implementations of the same classification algorithm (based on the given training data). During in-field usage of an implementation \(I\), we detect whether \(I\) produces outputs that deviate from the pseudo oracle obtained from running multiple-implementation testing. To reduce the runtime cost of monitoring with multiple implementations, our technique (1) predicts whether, for an application data entry, the ML implementation will produce an output deviated from the pseudo oracle based on the multiple implementations, and (2) run multiple imple-

\(^4\)A test input is also named as a test data or a test value in the software testing community.
mentations against the data entry *only* when such deviation is predicted.

This thesis makes the following main contributions:

- The first approach of multiple-implementation testing and monitoring for supervised learning software.
- The first definition of test case in testing ML software for unifying fundamental concepts from the ML and software testing communities.
- A technique for multiple-implementation monitoring for ML software without incurring high runtime cost.
- Evaluations for showing that our multiple-implementation testing 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, and our multiple-implementation monitoring substantially reduces the need of running multiple implementations with high prediction accuracy.

The rest of this thesis is organized as follows. Chapters 2 and 3 provide illustrating examples and background information, respectively. Chapters 4 and 5 present our approach and its implementation, respectively. Chapter 6 presents our evaluations. Chapter 7 discusses some issues. Chapter 8 presents related work, and Chapter 9 concludes.
Software programs such as ML applications do not have a reliable test oracle available and are sometimes known as “non-testable programs.” Weyuker [6] describes such programs as “Programs which were written in order to determine the answer in the first place. There would be no need to write such programs, if the correct answer were known”.

2.1 Machine Learning (ML)

A data set in ML can be considered as a table representing collections of entries. Each entry has attributes of which can be thought as the columns in a table. A label describes the class or category of an entry. For binary labels, each entry may pose a class label of 0 or 1 of which can be thought of as negative or positive labels. Model and classifier are used synonymously.

In this paper, to prevent confusion between test data set and test inputs in traditional software testing, application data is used to denote “test data”, as it is commonly used in the ML community.

Multiple-implementation testing may be useful for just about any algorithm with multiple implementations; however, it is much more useful for ML algorithms of which the output is usually unknown and there is not an oracle to validate the outcome of the algorithm’s implementation. As stated in Section 3.1, multiple-implementation testing does not need an expected label for the application¹ data set because the label with the majority vote for each entry in our data set is treated as a proxy to the algorithm-expected label.

The accuracy ratio is the percentage of entries that are correctly classified by the model. The accuracy ratio is used to summarize the number of correct classifications or mis-classifications when comparing different implementations. We only study the k-nearest neighbors (kNN) classifier and the Naive Bayes (NB) classifier, two of the most popular ML algorithm repository on
2.2 NaiveBayes Algorithm

Naive Bayes (NB) classifier [7] is based on the Bayes theorem and assumption of independence between predictor variables. For each entry, NB classifier calculates the probability of each class label based on the predictor variables $X$. In particular, the probability of class label $C_i$ is calculated by:

$$P(C_i|X) = \prod_{k=1}^{n} P(x_k|C_i) \times P(C_i)$$

where $P(C_i)$ is the prior probability of class label $C_i$, $P(X)$ is the probability of predictor variables, $n$ is the number of attributes for this entry, and $x_k$ represents the value of a single attribute of this entry.

**Theorem 1** Let $P(C)$ be the prior probability of class, and $P(X)$ be the probability of predictor variables. By having the likelihood $P(X|C)$ which is the probability of predictor given class, we can calculate the posterior probability of class given predictor variable:

$$P(C|X) = \frac{P(X|C)P(C)}{P(X)}$$

As the naive assumption, attributes are conditionally independent. The likelihood can be calculated by

$$P(X|C_i) = \prod_{k=1}^{n} P(x_k|C_i)$$

Since $P(X)$ is constant for all classes, we can use the following equation:

$$P(C_i|X) = \prod_{k=1}^{n} P(x_k|C_i) \times P(C_i)$$

Where $n$ is the number of variables/attributes.

The classification rules: $X$ belongs to $C_i$ if the probability $P(C_i|X)$ is the highest among all the $P(C_m|X)$ for all the $m$ class.

---

6To prevent confusion between test data and test inputs in traditional software testing, application data is used to denote “Validation/Test data”, as it is commonly used in the ML community.
2.3 kNN Algorithm

K-nearest neighbor (kNN) algorithm is a lazy learner whereby it learns by new cases as it stores all available cases and classifies new cases based on a similarity measure such as distance functions. A case is classified by a majority vote of its neighbors, with the case being assigned to the class amongst its kNN measured by a distance function. If \( K = 1 \), then the case is simply assigned to the class of its nearest neighbor.

The distance functions that may be used are below.

- **Euclidean Distance:**
  \[
  \sqrt{\sum_{i=1}^{k} (x_i - y_i)^2}
  \]

- **Manhattan Distance:**
  \[
  \sum_{i=1}^{k} |x_i - y_i|
  \]

- **Minkowski Distance:**
  \[
  \left[ \sum_{i=1}^{k} (|x_i - y_i|)^q \right]^{\frac{1}{q}}
  \]

  The Hamming distance is used for categorical variables.

  - **Hamming Distance:**
    \[
    D_H = \sum_{i=1}^{k} |x_i - y_i|
    \]

    \[
    \begin{cases}
    x = y \Rightarrow D = 0 \\
    x \neq y \Rightarrow D = 1
    \end{cases}
    \]

As seen in Figure 2.1, If \( K = 5 \), then query instance \( x_q \) will be classified as negative since three of its nearest neighbors are classified as negative.

Choosing the optimal value for \( k \) is best done by first inspecting the data.
In general, a large $k$ value is more precise as it reduces the overall noise but there is no guarantee. A good $k$ can be selected by various heuristic techniques, however, we will not present such techniques because as multiple implementation suggests, we shall use the same $k$ parameter for all implementations.

**Theorem 1** For each training example $< x, f(x) >$, add the example to the list of training examples. Given a query instance $x_q$ to be classified:

- Let $x_1, x_2, \ldots, x_k$ denote the $k$ instances from training examples that are nearest to $x_q$.
- Return the class that represents the maximum of the $k$ instances.

As seen in Figure 2.1, If $K = 5$, then query instance $x_q$ will be classified as negative since three of its nearest neighbors are classified as negative.
In this section, we present illustrating examples for multiple-implementation testing and monitoring.

3.1 Multiple-Implementation Testing

We use an example to illustrate the process of our multiple-implementation testing approach on a small sample data set. From Table 3.1, a training data set\(^7\) (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 (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. In this thesis, 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 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 [8, 9, 10, 11].

3.1.1 Consistent Configuration Parameters

Figure 3.1 presents an illustration of the kNN algorithm, along with the configuration parameters that need to be set for all implementations before performing multiple-implementation testing.

Line 3 from Figure 3.1 includes a step that requires the use of a distance function to calculate distance between \(x_1\) and \(x\); this step provides a variation of the algorithm and is not configurable. We refer to such situation as

\(^7\)To simplify the illustration, we omit the use of a validation or test data set.
1. Input: $D = (x_1, c_1), \ldots, (x_N, c_N)$
2. $x = (x_1, \ldots, x_n)$ new entry to be classified
3. FOR each labelled entry $(x_i, c_i)$ calculate $d(x_i, x)$
4. Order $d(x_i, x)$ from lowest to highest, $(i = 1, \ldots, N)$
5. Select the $K$ nearest instances to $x$: $D^*_x$
6. Assign to $x$ the most frequent class in $D^*_x$

Figure 3.1: The kNN Algorithm Pseudocode [1]

Table 3.1: Classification Model Predictions from all implementations

<table>
<thead>
<tr>
<th>App Data Entry ID</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>+1</td>
<td>+1</td>
<td>-1</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
</tr>
</tbody>
</table>

an implicit setting. Line 5 includes a step that selects $K$, which is an explicit parameter that has to be set during the execution of the implementation.

In our example, each classification model makes predictions for 6 application data entries from the application data set as shown in Table 3.1 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 the actual output for each data entry in the application data set. The final output of our approach, as shown in Table 3.2, 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 3.2 is equally split. In such case, we say that the majority-voted label is “undecidable”. We then treat the majority-voted and “undecidable” labels as the correct predictions and add the appropriate implementations that produce classification models with deviated predictions to Table 3.2.

From Table 3.2, we can see mis-classifications (i.e., deviating predictions) for both implementation $C$ and implementation $F$. 

10
Table 3.2: Majority predictions, and implementations with deviating predictions

<table>
<thead>
<tr>
<th>App Data Entry ID</th>
<th>Majority prediction</th>
<th>Implementation with deviating prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>“undecidable”</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>C, F</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>+1</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>+1</td>
<td>C</td>
</tr>
</tbody>
</table>

3.2 Multiple-Implementation Monitoring

The monitoring example illustrates the technique used to predict whether an implementation will show deviating behaviors during in-field usage. A deviating behavior is determined by the pseudo-oracle referred to as majority-voted label described in Section 3.1. The majority-voted label is computed from multiple-implementation testing, and is used here for multiple-implementation monitoring.

During multiple-implementation testing, if predictions from the implementation under test (IUT) deviate from the majority-voted predictions, we apply the monitoring technique to determine when IUT will likely predict a deviating label for future application data entries (during in-field usage). When the majority-voted label is “undecidable”, we conclude that IUT does not produce predictions that deviate from the majority-voted predictions.

Implementation C from Section 3.1 is used as IUT. Suppose that the application data entries come with attributes a1 to a3, along with their label -1 or +1. For an application data entry against IUT, we assign a NotDeviate or Deviate status to expose the behavior of IUT. A NotDeviate status is assigned when IUT’s prediction is consistent with the majority-voted prediction or the majority-voted prediction is “undecidable”. Note that when the majority-voted prediction is “undecidable”, executing additional implementations would not help predict a label on the particular application data entry. Otherwise, a Deviate status is assigned. Table 3.3 shows each application data entry, its corresponding predicted label, its majority-voted label from Section 3.1 and whether a predicted label is deviated from the majority-voted label.

We feed a1 to a3, along with the Deviate or NotDeviate status (as a label assigned to each entry), to a decision-tree learner. Figure 3.2 shows a monitoring tree. D corresponds to the Deviate status, and N corresponds
Table 3.3: Multiple-Implementation Monitoring for Implementation C

<table>
<thead>
<tr>
<th>App ID</th>
<th>Attr</th>
<th>Predicted Label</th>
<th>Majority-voted Label</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 1, 0</td>
<td>-1</td>
<td>“undecidable”</td>
<td>N</td>
</tr>
<tr>
<td>2</td>
<td>0, 0, 0</td>
<td>+1</td>
<td>-1</td>
<td>D</td>
</tr>
<tr>
<td>3</td>
<td>0, 0, 0</td>
<td>+1</td>
<td>-1</td>
<td>D</td>
</tr>
<tr>
<td>4</td>
<td>0, 1, 0</td>
<td>+1</td>
<td>+1</td>
<td>N</td>
</tr>
<tr>
<td>5</td>
<td>0, 1, 1</td>
<td>-1</td>
<td>-1</td>
<td>N</td>
</tr>
<tr>
<td>6</td>
<td>0, 0, 1</td>
<td>-1</td>
<td>+1</td>
<td>D</td>
</tr>
</tbody>
</table>

Figure 3.2: Multiple-Implementation Monitoring Tree for IUT
to the *NotDeviate* status.

During in-field usage, the monitoring tree receives an application data entry as input and produces an output to predict the *Deviate* or *NotDeviate* status of the application data entry. If the *NotDeviate* status is predicted, our approach does not execute additional implementations of the same ML algorithm on the application data entry. If the *Deviate* status is predicted, our approach executes additional implementations on the application data entry. In this way, our approach incurs much reduced monitoring cost than typical multiple-implementation monitoring, which runs multiple implementations on all application data entries.
CHAPTER 4

APPROACH

To address the test-oracle issue in testing supervised learning software, we propose formulation of a test case for multiple-implementation testing (a black-box approach) of supervised learning software. Our approach runs independently-written programs that follow the same specifications with the same test input and configuration 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 “algorithm-expected label”. We also propose the technique of multiple-implementation monitoring. The runtime cost of re-applying multiple-implementation testing to implementation $I$ (during in-field usage) can be significantly reduced by our technique of multiple-implementation monitoring. This technique uses a decision tree (described as a monitoring tree) to inform users when the result of a future input for implementation $I$ is likely to deviate from the pseudo-oracle. The overview of our approach is shown in Figure 4.1.

Next, we discuss our test case formulation, in particular, how we formulate a test oracle. Then, we describe our application of multiple-implementation testing. We then conclude by showing how multiple-implementation monitoring improves the dependability and reduces the overhead of multiple-implementation testing.
Figure 4.1: Multiple-Implementation Testing Framework for Supervised Learning Software
4.1 General Framework

We next describe how we form a test case for multiple-implementation testing of ML software. To test an implementation (i.e., implementation under test in short as IUT), we must use the same test inputs for the $n$ implementations (independently written, including IUT) of a particular algorithm. Additionally, we test these $n$ implementations under the same condition 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 is the combination of a test input and expected result needed to evaluate the software under test [12]. In multiple-implementation testing, we run a test case for each of the $n$ 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 it on every implementation, including the IUT with the given test input, and then determine whether the IUT produces outputs that deviate from the majority outputs from all the $n$ implementations. If a majority-voted output cannot be determined on a test input, we refer to it as “undecidable”, and we conclude that the IUT’s output does not deviate from the majority-voted output on for that test input.

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

**Test Oracle.** A test oracle is a mechanism for determining whether running a test input has passed or failed. We use test oracles in comparing the output of the IUT, for a given test input, to the expected output determined by the test oracles. In multiple-implementation testing of ML 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 case is a test case whose actual output is inconsistent (i.e., different) with the expected output. There are three test cases in Figure 4.2. Each test case’s test input corresponds to the training data and a single application entry. To create more test inputs, we create additional application entries. The actual output(s) corresponds to the three application entry labels. In Figure 4.2, we use the same training data and application
4.2 Consistent Configuration Parameters

All implementations should be run under the same conditions; therefore, in addition to giving these implementations the same test inputs, we provide the implementations with the same configuration parameter values. Software may implement (i.e., hardcode) 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 Configurations.** Prior to performing multiple-implementation testing, we group together implementations that share the same implicit configurations. We perform multiple-implementation testing on the group with the same implicit configuration as the IUT. A common implicit configuration in supervised learning is the normalization procedure used by the implementations (e.g., the z-score normalization and min-max normalization).

**Explicit Configurations.** Explicit configuration parameters are typically algorithm specific (e.g., $k$ in the kNN algorithm). We set the same value for all explicit configuration parameters including $k$ for all kNN implementations. Our evaluation results in this thesis (RQ1.3 and RQ2.1), also show how the majority-voted oracle performs, in multiple-implementation testing, when all the implementations still have their default (non-modified) implicit
and explicit configurations, and when all the implementations have consistent implicit and explicit configurations. Furthermore, we present results on the effectiveness of our learning-based technique of multiple-implementation monitoring in detecting deviations of the implementation under test’s predicted class label from the majority-voted class label, when the implementations’ default (non-modified) implicit and explicit configurations are used, and when all the implementations have consistent implicit and explicit configurations, respectively.

4.3 Multiple-Implementation Monitoring

The technique of multiple-implementation monitoring is used to improve dependability of an ML implementation during in-field usage while incurring low runtime cost. It thereby complements the multiple-implementation testing approach (which is typically used for offline in-house testing). The technique uses a decision tree (which we refer to as monitoring tree) to inform users when the output of a test input for an implementation is likely to deviate from the majority-voted output. In this way, users would use the ML software normally while receiving warnings of situations when the output of the software will likely deviate from most other implementations.

The multiple-implementation monitoring pays the price of additional runtime cost: when there are $n - 1$ additional different implementations to be used, the runtime cost for each application data entry (for classification) or data set (for clustering) would be $n$ times of the original runtime cost. To reduce such runtime cost, our technique (1) predicts whether, for an application data entry (or data set in the case of clustering), the ML implementation will produce an output deviated from the pseudo-oracle based on the multiple implementations, and (2) runs multiple implementations against the data entry (or data set in the case of clustering) only when such deviation is predicted.

In supervised learning, after performing multiple-implementation testing, we assign a NotDeviate or Deviate label to each application entry used for an IUT. A NotDeviate label is assigned if the actual output of the IUT is consistent with the expected output (i.e., the majority-voted output). A Deviate label is assigned if they are inconsistent. We then create a decision tree out of the unlabeled application data set and an assigned $N$ (NotDeviate) or $D$ (Deviate) label for each entry. The decision tree contains conditions on which the IUT will likely deviate from majority imple-
ments (based on the pseudo-oracle in multiple-implementation testing). A *NotDeviate* label reflects that a deviation is unlikely, while a *Deviate* label reflects that a deviation is likely. Given an unlabeled application entry, the decision tree produces a label of *NotDeviate* or *Deviate*, informing a user that the output of IUT may (or may not) deviate from majority implementations.

Let $A$ be a sequence of attribute values $(a_1, a_2, \ldots, a_n)$ where $n$ is the number of attributes and $T$ be a training data set of $A$. Every sequence in $T$ has a corresponding label. We therefore denote every sequence $A$ in $T$ as a labeled entry in the training data and $a_i$ as an attribute value in $A$. Let $X = (x_1, x_2, \ldots, x_n)$ be an unlabeled sequence. Application data $U$ is a set of $X$. For the IUT, we build a model, $M$, with $T$. This $M$ is used to assign labels to each entry in $U$. If the assigned label is consistent with the expected label, then we rename the label as *NotDeviate* (*Deviate* otherwise), denoting that the model, $M$ correctly (or incorrectly) assigns a label to $X$. We create a new training data $T'$ that replaces the initial corresponding labels with new labels denoting $N$ (*NotDeviate*) or $D$ (*Deviate*). We build a decision tree from the new training data. The decision tree is used to inform users whether the output of the IUT will deviate from majority implementations.

---

8In this thesis, application data may be referred to as validation/test data set in the ML community; we address it as application data set to prevent confusion with Test Input, which is used in the traditional software testing.

9The majority-voted label for each entry is treated as a proxy of the algorithm-expected label.
CHAPTER 5
IMPLEMENTATION

We have implemented the techniques in the proposed framework of multiple-implementation testing in order to apply them on testing supervised learning algorithms. Our framework partitions different data sets (derived from benchmark data) and randomly selects entries from the different partitions. Our framework performs transformation on the input data to follow the input formats required by each implementation in order to use the same input data for all implementations. We then run the different implementations with a test input. Our framework then computes the majority-voted output with the output of each implementation while also reporting implementations that produce deviating output.

Data-set generation. We obtain our benchmark data from the UCI ML repository [8, 9, 10, 11]. This repository is used by the ML community for empirical analysis of ML algorithms [4]. Furthermore, the data in this repository are representative of real-world situations. In the evaluation, we treat each implementation in our evaluation subjects as the implementation under test (IUT) one at a time. We apply multiple-implementation testing on each IUT using the Iris [8], Adult [9], and Poker [11] data sets from the UCI ML repository. 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 [13]. Finally, our framework randomly selects entries from the partitioned data set. The randomly selected entries from the Iris, Adult, and Poker data sets are used as the input data set (training and application data) in our evaluation.

Threshold for computing majority-voted output. We set a threshold of 0.5 in our test-result analysis. In other words, 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 [14]. We ignore test cases (and treat the test result as uncertain) when there are two candidates for the expected output or if there is no same output produced by at least half of the implementations. Our framework obtains and compares the predicted labels for all entries in the application data set of each implementation. Our framework then computes the expected output based on the threshold and informs the testers whether the output of the IUT matches with the expected output.

**Data set transformation.** Since the implementations used in multiple-implementation testing often are independently written, some implementations may have different data-set formats than other implementations. In order to run the generated data sets on each implementation, we may need to perform transformations on the data-set formats. Some commonly used input-file formats are arff, csv, and libsvm. Figure 5.1 shows the difference between the libsvm and csv input formats. In addition to creating such input formats from the initially generated data sets, for some implementations we may still need to conduct minor changes to the data in the input file, e.g., moving the column containing all labels to become first (or last) column in a csv input file.

![Figure 5.1: CSV and LIBSVM Input Formats](image)
To assess the effectiveness of our approach, we conduct an evaluation on 30 open-source projects. In our evaluation, we investigate the following research questions:

• **RQ1**: How effective is our multiple-implementation testing in detecting faults in ML software?
  - **RQ1.1**: How does the majority-voted oracle perform compared with an alternative approach (using the actual output from the benchmark data set as the oracle)?
  - **RQ1.2**: How does the majority-voted oracle perform when all the implementations are faulty?
  - **RQ1.3**: How does the majority-voted oracle perform when all the implementations still have their default (non-modified) values for explicit configuration parameters?

• **RQ2**: How effective is our technique of multiple-implementation monitoring in detecting deviations of the implementation under test’s predicted class label from the majority-voted class label?
  - **RQ2.1**: How effective is our multiple-implementation monitoring technique in detecting deviations of the class label (predicted by the implementation under test) from the majority-voted class label (when the implementation’s default (non-modified) values for explicit configuration parameters are used)?

### 6.1 Evaluation Setup

We next discuss our procedure on (1) selecting evaluation subjects, (2) assessing the effectiveness of multiple-implementation testing (RQ1), and (3) assessing the effectiveness of multiple-implementation monitoring (RQ2).
6.1.1 Evaluation Subjects

We select popular supervised learning algorithms for our evaluation. We construct a ranked list of supervised learning algorithms based on the algorithm’s popularity on a well-known code repository website - GitHub [15]. We search for each supervised learning algorithm listed on Wikipedia [16]. We use the exact name of the algorithm from Wikipedia as the search keywords on GitHub. We then rank each algorithm by the number of implementations available on GitHub. The keywords used to search for each algorithm are its name, as seen, on Wikipedia. Additionally, we filter the implementations based on their programming language, selecting only subjects implemented with Java, C#, and Python. We devise this filtering requirement because debugging tools for these programming languages are available to us.

We select 30 implementations of each of the top two algorithms from our filtered ranked list as our evaluation subjects. In particular, we gather 30 implementations of the kNN algorithm and 30 implementations of the Naive Bayes algorithm. We select all the available implementations from GitHub as our evaluation subjects. In addition, to reach the goal of 30 implementations, we search the algorithm keyword found on Wikipedia (plus the word “implementation”) on Google.com [17], selecting the top indexed results that include an implementation of the searched algorithm.

Of the 30 kNN implementations, we are unable to run 10 because the execution of such implementations requires non-trivial code changes to their source code in order to run them with our data sets. For example, one of our subject (kNN2) accepts only input data sets with two class labels (i.e., it accepts only binary data sets). Our approach being black box maintains the integrity of the implementation, and for evaluation purposes our source-code modification is to only set the explicit configuration parameter \( k \) for kNN implementations. To perform multiple-implementation testing for kNN implementations, we run our evaluation with explicit configuration parameter \( k \) as 1. Of the 30 NaiveBayes implementations, 20 implementations require non-trivial code changes to their source code in order to run them with our data sets.

Our evaluation subjects also include three ML packages found from Google.com: Weka, RapidMiner, and KNIME. Each of these three packages contains implementations for both the kNN and Naive Bayes algorithms. 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 industrial professionals to student projects, with varying input formats and varying output formats. The programming languages used in the implementations are Java, C#, or Python. Note that our approach of multiple-implementation testing can be generalized for any language. Table 6.1 shows evaluation results for each evaluation subject. In the evaluation, we treat each implementation in our evaluation subjects as the implementation under test (IUT) one at a time. We apply our approach of multiple-implementation testing on each IUT using the UCI data set described in Section 5.

6.1.2 Fault Detection

To assess the effectiveness of our approach of multiple-implementation testing in detecting faults in ML implementations (RQ1), we measure the number of IUTs with at least one failing test case (i.e., the actual output is inconsistent with the expected output). We also report the number of faults in these IUTs. These faults are detected by tracing the execution of each failing test case to find unexpected behavior in the IUT. Then we try to fix the faults and rerun the failing test cases and see whether they pass.

For RQ1.1, we compare the majority-voted oracle for each entry of the application data to the expected labels of that entry. Note that we obtain the expected labels from the original data set: the expected labels are the removed class labels for the entries in the application data. We also compare each majority-voted oracle with the corresponding algorithm-expected label.

6.1.3 Multiple-Implementation Monitoring

To assess the effectiveness of the technique of multiple-implementation monitoring (RQ2), we measure the overall accuracy, the number of false positives, and the number of false negatives produced by this technique. Note that a false positive means that our technique incorrectly determines that the implementation under test’s predicted class label deviates from the majority-voted oracle. In particular, we perform a 10-fold cross validation on the application data with labels to obtain these metrics. Also note that we evaluate on only the implementations with at least one failing test case. We also assess the effectiveness of the technique of multiple-implementation monitoring in reducing the cost of applying multiple-implementation testing for the implementation under test. In particular, we measure the percentage of application data entries that are predicted not to deviate from the
Table 6.1: Evaluation Results of Fault Detection

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Implementation</th>
<th># Failing</th>
<th># Fault</th>
<th>Accuracy (%)</th>
<th># FN</th>
<th># FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>kNN</td>
<td>kNN1</td>
<td>3</td>
<td>0</td>
<td>90</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>kNN</td>
<td>kNN3</td>
<td>20</td>
<td>2</td>
<td>96.67</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>kNN</td>
<td>kNN4</td>
<td>1</td>
<td>0</td>
<td>96.67</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>kNN</td>
<td>kNN5</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>kNN</td>
<td>kNN6</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>kNN</td>
<td>kNN8</td>
<td>10</td>
<td>1</td>
<td>93.33</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>kNN</td>
<td>kNN12</td>
<td>8</td>
<td>0</td>
<td>73.33</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>kNN</td>
<td>kNN15</td>
<td>1</td>
<td>0</td>
<td>96.67</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>kNN</td>
<td>kNN16</td>
<td>20</td>
<td>1</td>
<td>96.67</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>kNN</td>
<td>kNN18</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>kNN</td>
<td>kNN23</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>kNN</td>
<td>kNN24</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>kNN</td>
<td>kNN26</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>kNN</td>
<td>kNN27</td>
<td>1</td>
<td>0</td>
<td>96.67</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>kNN</td>
<td>kNN28</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>kNN</td>
<td>kNN29</td>
<td>1</td>
<td>0</td>
<td>96.67</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>NaiveBayes</td>
<td>NaiveBayes1</td>
<td>10</td>
<td>2</td>
<td>60</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>NaiveBayes</td>
<td>NaiveBayes5</td>
<td>4</td>
<td>1</td>
<td>86.67</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>NaiveBayes</td>
<td>NaiveBayes13</td>
<td>10</td>
<td>1</td>
<td>73.33</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>NaiveBayes</td>
<td>NaiveBayes17</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NaiveBayes</td>
<td>NaiveBayes18</td>
<td>7</td>
<td>1</td>
<td>76.67</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>NaiveBayes</td>
<td>NaiveBayes19</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NaiveBayes</td>
<td>NaiveBayes21</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Weka</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RapidMiner</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>KNIME</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

majority-voted oracle.

6.2 Evaluation Results

We next present our evaluation results for the effectiveness of our multiple-implementation testing (RQ1), and the effectiveness of our multiple-implementation monitoring (RQ2).
Table 6.2: The number of correct predictions from the majority-voted oracle and the majority-voted oracle (without fault-free implementations), respectively.

<table>
<thead>
<tr>
<th>Prediction</th>
<th>Algorithm</th>
<th>Majority-voted oracle</th>
<th>Majority-voted oracle (w/o fault-free implementations)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>kNN</td>
<td>Correct 29</td>
<td>Correct 29</td>
</tr>
<tr>
<td></td>
<td>NaiveBayes</td>
<td>Incorrect 1</td>
<td>Incorrect 2</td>
</tr>
<tr>
<td>%Correct Prediction</td>
<td>96.67%</td>
<td>93.33%</td>
<td>76.67%</td>
</tr>
</tbody>
</table>

6.2.1 Fault Detection

Among the 30 implementations under test in our evaluation subjects, 14 implementations have failing test cases. Column #Failure in Table 6.1 indicates the number of failing test cases for each implementation under test. Among these 14 implementations, our approach can detect 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 6.1 indicates the number of detected faults for each implementation under test. We file a bug report to the corresponding open-source project for each detected fault. In addition to describing the fault and induced failure in the bug report, we also make suggestions on how to fix the detected fault.

The fault-detection capability of our approach of multiple-implementation testing 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. 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 RQ1.2, 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. In particular, we compare both of the 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 evaluation 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 6.2. Note that For NaiveBayes, there are 4 faulty implementations out of 10 implementations. For kNN, there are 7 faulty implementations out of 20 implementations. We notice that for KNN, even though we remove all the fault-free implementations, the majority-voted oracle (without fault-free implementations) 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 test cases. The majority of the predictions are still correct. However, the majority-voted oracle (without fault-free implementations) for NaiveBayes is less accurate, 76.67%, which is lower than that of the majority-voted oracle (93.33%). The reasons that the majority-voted oracle (without fault-free implementations) does not work well on this case is that we have only 4 implementations (without fault-free implementations) for NaiveBayes, and 3 of the 4 implementations output incorrect predictions. Such case implies that the majority-voted oracle works reasonably well, when we have a 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.

For RQ1.3, we are interested to see how the majority-voted oracle would perform if we do not modify default values for the explicit configuration parameters of implementations when performing multiple-implementation testing. We aim to investigate the impact of default (i.e., non-modified) values of explicit configuration parameters on our multiple-implementation testing. Table 6.1 presents evaluation subjects and results for multiple-implementation testing with modified values for configuration parameters. We next summarize our findings. With our 20 kNN subjects, 11 of them require \( k \) as an explicit configuration parameter before running the algorithm (i.e., an exception is thrown if the parameter value is not set). 4 of the subjects have a default value of ‘3’. 2 of the subjects have value of ‘1’. There are two outliers, kNN4 with default value ‘10’ and kNN6 with default value ‘90’. Note that when performing multiple-implementation testing in our approach (with consistent values for explicit configuration parameters), we set the value of \( k \) to be 1 for all the implementations. The outputs for all subjects with their default explicit parameters are the same except with kNN6, which has an outlier value \( k \) of ‘90’. Only 56.7% of the output labels of the default kNN6 are 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 values for the explicit configuration parameters and when it is performed with modified values for the explicit configuration parameters. The NaiveBayes algorithm does not require any explicit configuration parameter.

Faults Detected in Naive Bayes (NB) Implementations

The first detected fault was from NaiveBayes13, an implementation of the NB algorithm. The NB algorithm predicts a class for an entry by selecting the greatest posterior probability of the entry being in a particular class. Posterior probability is the combination of the prior probability of a class and the likelihood of the entry given the class. To classify entry $X$ containing attributes $(A_1 \ldots A_n)$ given classes $(C_1 \ldots C_n)$, we calculate:

$$Posterior(C_1) = P(C_1)p(A_1|C_1) \ldots p(A_i|C_1)/evidence$$

$$\ldots$$

$$Posterior(C_n) = P(C_n)p(A_1|C_n) \ldots p(A_i|C_n)/evidence$$

The evidence (also termed normalizing constant) may be calculated below:

$$evidence = P(C_1)p(A_1|C_1) \ldots p(A_n|C_1) + \ldots + P(C_1)p(A_1|C_n) \ldots p(A_i|C_n)$$

Then the class with the highest posterior numerator is selected as the predicted label. In order to find $A_1$, we calculate the number of attribute values for its attribute. In a formal definition, we compare a correct implementation and NaiveBayes13:

Given $A$ as a sequence of attributes $(A_1, A_2, \ldots A_n)$ where $n$ is the number of attributes and $1 <= i <= n$, $|Ai| == m$ (denoting the number of rows in the training data).

Given $X$ as our unlabeled entry with $X_i$ as an attribute value in $X$, the correct implementation is $\forall 1 <= i <= n$, # of occurrences of $X_i$ in $A_i/m$. But the incorrect NaiveBayes13 implementation is $\forall 1 <= i <= n$, # of occurrences of $X_i$ in $(A_1, A_2, \ldots A_n)/(n*m)$.

Therefore, NaiveBayes13 finds the likelihood probability (the probability of $X_i$, an entry value, occurring in class $C_i$) by finding the probability of $X_i$ occurring in all the attributes $(A_1, A_2, \ldots A_n)$. However, the correct implementation is to find the probability of $X_i$ in $A_i$.

As seen from the application data set in Table 6.3, in order to calculate the likelihood probability for attribute value ‘5.4’ for each label, we calculate the probability of ‘5.4’ in $A_1$ for the specified label (e.g., Iris-setosa). However, NaiveBayes13 calculates the probability of ‘5.4’ in all attributes $A_1$ to $A_4$.
Table 6.3: NaiveBayes13 Application Data

<table>
<thead>
<tr>
<th>a1</th>
<th>a2</th>
<th>a3</th>
<th>a4</th>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>3.0</td>
<td>1.1</td>
<td>0.1</td>
<td>iris-setosa</td>
</tr>
<tr>
<td>5.8</td>
<td>4.0</td>
<td>1.2</td>
<td>0.2</td>
<td>iris-setosa</td>
</tr>
<tr>
<td>5.7</td>
<td>5.4</td>
<td>1.5</td>
<td>0.4</td>
<td>iris-setosa</td>
</tr>
<tr>
<td>5.4</td>
<td>3.9</td>
<td>1.3</td>
<td>0.4</td>
<td>iris-setosa</td>
</tr>
<tr>
<td>5.0</td>
<td>2.0</td>
<td>3.5</td>
<td>1.0</td>
<td>iris-versicolor</td>
</tr>
<tr>
<td>5.6</td>
<td>2.9</td>
<td>3.6</td>
<td>1.3</td>
<td>iris-versicolor</td>
</tr>
<tr>
<td>6.7</td>
<td>3.1</td>
<td>4.4</td>
<td>5.4</td>
<td>iris-versicolor</td>
</tr>
<tr>
<td>5.6</td>
<td>3.0</td>
<td>4.5</td>
<td>1.5</td>
<td>iris-versicolor</td>
</tr>
<tr>
<td>6.4</td>
<td>3.2</td>
<td>5.3</td>
<td>2.3</td>
<td>iris-virginica</td>
</tr>
<tr>
<td>6.5</td>
<td>3.0</td>
<td>5.4</td>
<td>1.8</td>
<td>iris-virginica</td>
</tr>
<tr>
<td>7.7</td>
<td>3.8</td>
<td>6.7</td>
<td>2.2</td>
<td>iris-virginica</td>
</tr>
<tr>
<td>7.7</td>
<td>2.6</td>
<td>6.9</td>
<td>2.3</td>
<td>iris-virginica</td>
</tr>
<tr>
<td>4.7</td>
<td>3.2</td>
<td>1.6</td>
<td>0.2</td>
<td>iris-setosa</td>
</tr>
<tr>
<td>4.8</td>
<td>3.1</td>
<td>1.6</td>
<td>0.2</td>
<td>iris-setosa</td>
</tr>
<tr>
<td>5.4</td>
<td>3.4</td>
<td>1.5</td>
<td>0.4</td>
<td>iris-setosa</td>
</tr>
</tbody>
</table>

for the specified label.

Note that if we run a data set with different unique data types (i.e., a data set containing 1 attribute of categorical type, 1 attribute of text type, 1 attribute of integer) thereby limiting the occurrence of a value to only within its attribute. Then, NaiveBayes13’s classification results will be the same as other NB implementations. We run the Balloons data set [10] to confirm the preceding observation. However, if we run a data set where an attribute value occurs in more than one attribute, then NaiveBayes13 will have more misclassifications than other NB implementations. We run the Iris data set [8] to confirm the preceding observation.

The second fault occurs in both NaiveBayes5 and NaiveBayes1. We find that the implementations neglect the combination of the calculated final prediction by predicting only based on the likelihood probability. A core of the NB algorithm is to use the posterior probability for predictions. The posterior probability is a combination of both prior and likelihood probability.

To classify entry \( X \) containing attributes \( (A_1 \ldots A_n) \) given classes \( (C_1 \ldots C_n) \), the faulty implementations use the calculation below, of which neglects the prior probability, \( P(C_n) \):

\[
Posterior(C_1) = \frac{p(A_1|C_1) \ldots p(A_i|C_1)}{\text{evidence}} \\
\cdots \\
Posterior(C_n) = \frac{p(A_1|C_n) \ldots p(A_i|C_n)}{\text{evidence}}
\]

The correct implementation should include the prior probability:
\[ Posterior(C_1) = P(C_1)p(A_1|C_1)\ldots p(A_i|C_1)/evidence \]

\ldots
\[ Posterior(C_n) = P(C_n)p(A_1|C_n)\ldots p(A_i|C_n)/evidence \]

We confirm the preceding observation by comparing the final ranking of classes with and without adding the calculation of the prior probability. Adding the calculation of the prior probability results in a similar accuracy percentage achieved by other NB implementation in our evaluation.

Another fault is detected in NaiveBayes1. The implementation does not include a form of smoothing in its calculation, leading to a high number of failing test cases. If an entry \( X \) containing attributes \((A_1 \ldots A_n)\) has an attribute value that does not occur in any of \( X \)'s attribute for category \( C \), then \( P(X|C) \) will be estimated as 0. Given \( n \) as the number of attributes in \( X \), then the product \( P(C_i) \times P(X_1|C_i) \times P(X_2|C_i) \times \ldots \times P(X_n|C_i) \) will be equal to 0, no matter how much other evidence there is favoring \( C_i \). This fault results in the majority of failing test cases in NaiveBayes1 because when this situation occurs, the probability for each class being assigned to the test case is 0. The implementation then chooses the last class in the list of classes, of which is always 'Iris-virginica'. By simply adding laplacian correction, we are able to fix this issue.

NaiveBayes18 uses an arbitrary way for smoothing. It performs an estimation for every entry's attribute value with a zero probability value by assigning the entry's probability as \( 1/(\text{unique count of values for given attribute and class from the training data set}) \). This way results in failing test cases. By adding laplacian correction, we are able to fix this issue.

We have reported all the preceding detected faults to the developers of the corresponding open-source projects.

Faults Detected in k-Nearest Neighbors (kNN) Implementations

Two faults are detected in kNN3. The first fault is due to casting double values to integers. We detect this fault after running the Iris data set (with all its attribute types as doubles) on the implementation. After reading the attribute values as double, the implementation then casts the attribute values to integers. Attribute values such as '5.4' and '5.8' are all treated as '5' in the implementation. The second fault is that the application data set is normalized differently from the training data. Both test and training data sets should use the same ranges for normalization. The developers of the kNN3 implementation have acknowledged this fault as a comment in code.

We find an if-else-condition fault in kNN8. The developers of the kNN8
implementation use if-else statements to find the largest count of labels among several class labels in order to select the predicted label. The if-else branch is not well written to handle the third case. We simply fix this fault by restructuring the if-else statement in the code.

The final fault is due to a re-initialization fault. The developers of the kNN16 implementation do not re-initialize a variable before finding the closest label distance to a new application entry. The kNN algorithm requires that the distances of a new application entry and each point in the training data set to be computed. Then, we classify the new application entry based on $k$ closest entries from the training data set. In order to perform this calculation, we have to re-initialize the distance variable used in the code to find each distance. We fix this fault by simply re-initializing the distance variable.

The majority of the faults detected for the NB subjects are at the core of the algorithm while the faults detected for the kNN subjects are less severe, and experienced developers could have avoided such faults in the kNN subjects. We have filed bug reports for all the preceding faults and are waiting for confirmation from the developers of the corresponding open-source projects.

6.2.2 Multiple-Implementation Monitoring

Columns **Accuracy**, **# FN**, and **# FP** of Table 6.1 show the accuracy, the number of false negatives, and the number of false positives of applying the multiple-implementation monitoring technique on each implementation that has at least one failing test case.

On average, the multiple-implementation monitoring technique achieves 87.18% accuracy. For kNN implementations, the technique of multiple-implementation monitoring achieves very high accuracy: out of the 9 implementations that each have at least one failing test case, multiple-implementation testing achieves greater than 95% accuracy for 6 implementations. The average accuracy for the kNN implementations is 92.96% while the average accuracy for the Naive Bayes implementations is 74.17%.

We analyze the cost and benefit of applying multiple-implementation monitoring for potential in-field usage. The benefit is the reduced runtime cost compared to applying full multiple-implementation testing for each application data entry. The cost of multiple-implementation monitoring is two-fold: (1) multiple-implementation monitoring may mis-predict and run unnecessary multiple-implementation testing, increasing the runtime cost, and
(2) multiple-implementation monitoring may mis-predict and fail to run multiple-implementation testing for an application data entry that causes deviation.

Our technique of multiple-implementation monitoring can substantially reduce the runtime cost by reducing the number of application data entries that need multiple-implementation testing. On average, the technique of multiple-implementation monitoring requires running multiple-implementation testing on only 20.5% of the application data entries for each implementation under test. For 6 implementations, multiple-implementation monitoring requires no runs of multiple-implementation testing at all.

The number of false positives in multiple-implementation monitoring represents the number of application data entries for which multiple-implementation monitoring mis-predicts and runs unnecessary multiple-implementation testing. On average, the multiple-implementation monitoring technique produces only 4.37% false positives for each implementation under test. Multiple-implementation monitoring produces 0 false positive for 7 out of 14 implementations under test that each have at least one failing test case.

The number of false negatives in multiple-implementation monitoring represents the number of application data entries for which multiple-implementation monitoring mis-predicts and fails to run multiple-implementation testing. Such application data entries are likely to produce incorrect results. Mis-predicting such cases may result in failing to validate these application data entries and missing debugging opportunities. On average, the technique of multiple-implementation monitoring produces only 8.43% false negatives for each implementation under test. Multiple-implementation monitoring produces 0 false negative for 2 implementations under test.

Overall, our technique of multiple-implementation monitoring reduces the number of runs of multiple-implementation testing substantially, while maintaining high accuracy, as well as low false positives and low false negatives, for its prediction. To address RQ2.1, our evaluation results show that performing the monitoring technique with default values for explicit configuration parameters of the implementations is the same as performing with modified values for explicit configuration parameters except for the outlier described in Section 6.2.1: kNN6 with the value of k as 90. The monitoring technique achieves an accuracy of 85.71%. Multiple-implementation monitoring produces 2 false negatives and 2 false positives. There is no need to perform the monitoring technique on kNN6 with modified parameter values, because kNN6 is found to be a fault-free implementation.

Overall, our technique of multiple-implementation monitoring reduces the
number of runs of multiple-implementation testing substantially, while maintaining high accuracy, as well as low false positives and low false negatives, for its prediction.
CHAPTER 7

DISCUSSION

Multiple-implementation testing 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 issue 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 difficulty 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-learn in Python) that can be used for multiple-implementation testing.

We address the runtime cost of re-applying multiple-implementation testing with the technique of multiple-implementation monitoring. With the technique, developers have to create a classification model (i.e., monitoring tree) once, as opposed to creating models for all different \( n \) implementations during multiple-implementation testing.
Differential testing [18] is a testing approach closely related to multiple-implementation testing. During differential testing, developers would like to generate tests that exhibit 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 conducting multiple-implementation testing but just conducting differential testing or testing against the reference implementation. In multiple-implementation testing, all implementations are treated equally and each implementation places an equal vote towards the test oracle.

Murphy and Kaiser [4] proposed an approach for testing ML applications based on metamorphic testing [5], 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 [19] for the verification and validation of disease-spread models and testing of phylogenetic-inference programs by Sadi et al. [20] where metamorphic testing 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 [21]. Groce et al. [22] proposed test-selection techniques that provide very good failure rates for end-user interactive ML systems. Their 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 [14], web input validators [23], and cross-browser issues [24]. Our work differs from these previous approaches as we adapt multiple-implementation testing by introducing a mon-
onitoring mechanism with reduced cost in the context of ML software. By using multiple-implementation testing, we detect faults in ML software as shown in our evaluation.

Our work on multiple-implementation monitoring is related to failure-avoidance research. Aviso [25] includes a mechanism for failure avoidance in concurrent programs. Aviso monitors the execution of a multi-threaded program to collect runtime information and a history of events from failing runs. Then it generates constraints that will force future executions to execute a different schedule of events avoiding the failures. Similarly, in GUI applications, Michail and Xie [26] used the nearest-neighbor algorithm on bounded execution histories to predict future failures.
In this thesis, we have presented a novel black-box approach of multiple-implementation testing for supervised learning software. Our approach includes techniques to address challenges in multiple-implementation testing and monitoring: the definition of test case, resolution of inconsistent algorithm configurations across implementations, and reduction of monitoring cost. The evaluation results have shown that our approach is effective in detecting real faults in real-world ML software (even popularly used ones): 5 faults from 10 NaiveBayes implementations and 4 faults from 20 k-nearest neighbor implementations, and our approach incurs much reduced monitoring cost than typical multiple-implementation monitoring.
REFERENCES


