Towards Effective Metamorphic Testing by Algorithm Stability for Linear Classification Programs

Yingzhuo Yang[a,b], Zenan Li[a,b], Huiyan Wang[a,b], Chang Xu[a,b,*] and Xiaoxing Ma[a,b]

[a]State Key Lab for Novel Software Technology, Nanjing University, Nanjing, China
[b]Department of Computer Science and Technology, Nanjing University, Nanjing, China

ARTICLE INFO

Keywords:
Metamorphic testing
Metamorphic relation
Machine learning program
Linear classifier
Algorithm stability

ABSTRACT

The quality assurance for machine learning systems is becoming increasingly critical nowadays. While many efforts have been paid on trained models from such systems, we focus on the quality of these systems themselves, as the latter essentially decides the quality of numerous models thus trained. In this article, we focus particularly on detecting bugs in implementing one class of model-training systems, namely, linear classification algorithms, which are known to be challenging due to the lack of test oracle. Existing work has attempted to use metamorphic testing to alleviate the oracle problem, but fallen short on overlooking the statistical nature of such learning algorithms, leading to premature metamorphic relations (MRs) suffering efficacy and necessity issues. To address this problem, we first derive MRs from a fundamental property of linear classification algorithms, i.e., algorithm stability, with the soundness guarantee. We then formulate such MRs in a way that is rare in usage but could be more effective according to our field study and analysis, i.e., Past-execution Dependent MR (PD-MR), as contrast to the traditional way, i.e., Past-execution Independent MR (PI-MR), which has been extensively studied. We experimentally evaluated our new MRs upon nine well-known linear classification algorithms. The results reported that the new MRs detected 37.6-329.2% more bugs than existing benchmark MRs.

1. Introduction

Machine learning has been widely applied in a great number of computational fields over the past few years. In addition to those successful applications, such as spam email filtering, item recommendation, and image recognition, machine learning has also been intensively applied recently to some mission-critical tasks, e.g., medical diagnosis, financial distress forecasting, and autonomous driving [12, 36, 43]. However, despite its popularity in application, the quality assurance of machine learning, especially for its kernel learning programs, is still missing adequate attention, while such assurance plays a vital role in the life cycle of deploying machine learning systems.

Machine learning systems typically deploy previously trained models from learning programs to provide smart-decision services. Much recent research has targeted on the deployed machine learning models, and found that they could be vulnerable to adversarial attacks [8]. In order to validate the reliability of such machine learning systems, existing work has mostly emphasized on testing those trained machine learning models for deployment. For example, a series of testing techniques based on neuron coverage have been proposed for examining whether a machine learning model, e.g., deep neural networks [39, 40, 50, 70], could make wrong predications upon dedicatedly designed inputs. With quite a few reported successful cases, existing work, however, lacks enough attention on the quality assurance for learning programs themselves, which is actually the foundation for the quality of thus trained models. Even worse, such an overlooking of the learning programs could cause its generated models to keep suffering from unknown quality problems.

To this end, in this article, we focus particularly on the quality of machine learning programs themselves, and aim to effectively detect potential bugs in them (such programs typically refer to the implementations of relevant machine learning algorithms). We say that this problem has been more or less overlooked by many machine learning researchers and developers. The reason is that they tend to naturally believe that implementing machine learning programs faithfully according to respective algorithms should not be a big problem. Besides, in practice, when the accuracy of a trained model is somewhat low, developers could tend to attribute the problem to their incorrect settings of specific algorithm hyperparameters rather than to possible implementation bugs. On the other hand, since a trained model's quality problems are likely to be caused by the potential bugs of its corresponding learning program, fixing these bugs would be extremely useful and of great advantage to avoiding future problematic trained models. Nevertheless, although testing such machine learning programs is vital and critical, it is not that easy.

One of the key obstacles to testing machine learning programs is the oracle problem [2]. Considering that a typical machine learning program contains two parts, namely, training and predicting, we focus mainly on the training part, which is typically more complex and tend to contain bugs, while the remaining predicting part commonly refers to classic searching and reporting functionalities upon trained models from the training part. With this setting, the input of a learning program is the training set, and its out-

*Corresponding author
yingzhuoy@smail.nju.edu.cn (Y. Yang); lizenan@smail.nju.edu.cn (Z. Li); cocowhy1013@gmail.com (H. Wang); changxu@nju.edu.cn (C. Xu); xxm@nju.edu.cn (X. Ma)
ORCID(s): 0000-0002-2897-7548 (Y. Yang)

put is the model parameter for the thus trained model. The training set usually consists of instances sampled from an unknown distribution, and the model parameter is usually a high-dimensional vector. Hence, it is impossible to automatically compute a correct model parameter on a dataset randomly sampled from this unknown distribution based on the learning algorithm (a.k.a. oracle problem). In other words, one cannot easily obtain the expected output of a learning algorithm for any given input. Even if considering the overall accuracy for the model instead of giving detailed model parameters, one can only obtain a rough estimate (or with) of the accuracy [63].

The other key obstacle to testing a machine learning program is the statistical nature of its implemented machine learning algorithm. A machine learning algorithm is usually designed on the basis of statistics, which should be inherently capable of hiding errors [14]. For example, in a classification task, suppose that the model obtained by a machine learning program may get an accuracy of 90%. One may confidently consider that this learning program is good (even bug-free) due to this acceptable accuracy. However, there is still a possibility that a bug resides in its implementation, and fixing this bug may further improve the accuracy to 95%. Note that this is an interesting observation that bugs of learning programs may not always cause an accuracy reduction and they could instead increase the accuracy in some cases [1] . Therefore, the accuracy itself can give wrong hints on the existence of bugs in machine learning programs. This observation thoroughly reveals the difference between traditional program bugs and machine learning program bugs, and it also challenges the efficacy of traditional testing techniques that detect program bugs through observing a trained model's performance. Besides, it also reminds us of considering what properties of learning algorithms should be used to effectively detect bugs.

In this article, we focus on testing the programs of linear machine learning algorithms, which have been widely applied in popular machine learning applications, such as logistic regression, support vector machine, and linear discriminant analysis [1, 18, 26]. Some existing work [23, 68, 69] has proposed to apply *metamorphic testing* to alleviate the oracle problem in testing learning programs. However, they are restricted by the following limitations: (1) their selected properties (i.e., metamorphic relations or MRs), e.g., shuffling the training data, do not produce different models, thus would not reveal the kernel statistical nature of learning programs, and are likely to be not that effective on detecting bugs in machine learning programs; (2) their considered properties for metamorphic testing sometimes lack theoretical guarantee and can be accidentally violated. Therefore, we in this article aim for a nice property that should touch the kernel statistical nature of learning programs, and provide a theoretical guarantee for the effectiveness of such MR-based testing.

At first glance, it is a very simple and even trivial issue. But in fact, the existing methods are still not sufficiently effective. The main reason is due to the target property of metamorphic testing, i.e., metamorphic property. The metamorphic property fundamentally determines the efficacy of methods, but the currently selected properties lack statistical characteristics and theoretical guarantee. Thus it still needs further exploration for the metamorphic testing of machine learning programs.

Moreover, by investigating existing MR usages across different fields, we observe that MRs could be divided into two categories according to how they are formulated, namely, *Past-execution Independent MR* (PI-MR) and *Past-execution Dependent MR* (PD-MR). They differ in whether the follow-up input generation in MR depends on specific characteristics in past executions (e.g., output in the last execution). For example, in PI-MR, the generation of a follow-up input, a.k.a. *metamorphic transformation*, would depend only on its corresponding source input, while in PD-MR, the dependency would expand to both the source input and source output, making the transformation non-trivial. According to our observations and experiences with MR-based testing, our second focus in this article is to exercise and explore PD-MR for better effectiveness in testing machine learning programs.

To sum up, we in this article aim to (1) *find a fundamental property associated with the kernel statistical natural of machine learning algorithms*, (2) *formula the property as the form of PD-MR*, and (3) *enable theoretical guarantee on the formulated MRs*. We would consider the kernel stability nature of linear classification algorithms, and encode it into two PD-MRs with theoretical guarantee, aiming to verify the fundamental property of machine learning programs.

Systematic empirical evaluations show the high effectiveness of our proposed MRs on bug detection. We conducted experiments on programs of nine well-known linear classification algorithms. To simulate bugs in learning programs, we generated 1,265 mutants through an improved version of the Python mutation tool mutmut. The extensive experiments showed the nice effectiveness of our MRs on detecting bugs that disturb the algorithm stability. Compared with the six state-of-the-art MRs proposed by existing work [23, 69, 72], our MRs can be much more effective in detecting bugs, with an improved mutant killing rate ranging from 37.6% to 329.2%.

The rest of this article is organized as follows. Section 2 introduces background knowledge used in this work, including machine learning foundations, linear classification algorithms, and metamorphic testing. Section 3 elaborates on how we encode the stability of linear classification algorithms into two PD-MRs. Section 4 evaluates our proposed MRs with nine algorithm implementations. Finally, Section 5 discusses related work in recent years, and Section 6 concludes this ariticle.

[1] Here are two typical examples:

- https://github.com/BVLC/caffe/issues/4202
- https://github.com/Britefury/self-ensemble-visual-domain-adapt-photo

2. Background

In this section, we introduce some background knowledge involved with our work, including the machine learning foundations, linear classification algorithms, and metamorphic testing.

2.1. Machine learning foundations

Machine learning is the study of computer algorithms that improve system efficiency through experience [45]. Typically, a machine learning algorithm is designed to build a model that gives predictions or makes decisions based on the previous observation. For details, it would train a model based on a labeled dataset (called training set) and expect such a trained model to generalize its prediction ability to data that it has not seen before. Such generalization ability is the key property of a machine learning algorithm.

In machine learning fields, the probably approximately correct (PAC) learning theory is used to provide a formal description of an algorithm's generalization ability [63], by calculating a boundary of a trained model's generalization error. For example, in a binary classification task, assume that a training set $D = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^m$ is sampled from a distribution $\mathcal{D}$ independently. In this case, given a trained classification model or classifier h, its empirical error denotes the accuracy of this classifier h with respect to the training set D:

$$\widehat{E}(h; D) = \frac{1}{m}\sum_{i=1}^{m} \mathbb{I}(h(\boldsymbol{x}_i) \neq y_i), \tag{1}$$

where $\mathbb{I}(\boldsymbol{x}_i, y_i)$ is an indicator function that takes value 1 when $h(\boldsymbol{x}_i) = y_i$ holds and takes value 0 if otherwise. Then, the generalization error is calculated by the accuracy of classifier h on the distribution $\mathcal{D}$ as follows:

$$E(h; \mathcal{D}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}}\left[\mathbb{I}(h(\boldsymbol{x}) \neq y)\right]. \tag{2}$$

Thus, we can bound the gap between the empirical error $\widehat{E}(h; D)$ and the generalization error $E(h; \mathcal{D})$ through the HoefIding inequality [27]:

$$P(|E(h) - \widehat{E}(h)| \geq \epsilon) \leq 2\exp\left(-2m\epsilon^2\right), \quad \forall \epsilon \in (0, 1). \tag{3}$$

Although there have been several theoretical studies similar to HoefIding inequality in 3 when estimating the generalization errors, they cannot give analysis results for specific algorithms. To overcome this weakness, one starts to explore the stability of algorithms [58] to describe how the output of a specific algorithm changes when modifying its input. For a machine learning algorithm, its input refers to a training set, and the modifications upon it can be normally divided into two categories:

- Obtaining a new training set $\widehat{D}$ by **replacing** i-th example in the training set D:

$$D = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^m \Rightarrow \widehat{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1, i\neq j}^m \cup \{(\widehat{\boldsymbol{x}}_j, y_j)\}.$$

- Obtaining a new training set $\widehat{D}$ by **removing** i-th example from the training set D:

$$D = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^m \Rightarrow \widehat{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1, i\neq j}^m.$$

Therefore, as can be seen from the above equations, another advantage of stability analyses is that only the training set needs to be used.

Furthermore, the equivalence between stability and generalization error bounds has also been extensively studied [4, 21, 34, 46]. Therefore, based on such equivalence studies, stability can be considered as a core property of machine learning algorithms, and it may be more ideal in application.

2.2. Linear classification algorithms

Since a multi-class classification task can be decomposed into multiple binary classification tasks, regardless of whether *one-vs.-one* or *one-vs.-all* strategy is applied, we simply consider a binary classification task here. Given any example $\boldsymbol{x} = (x_1, \ldots, x_n)$, the goal of classification is to use its features (or attributes), i.e., $\{x_1, \ldots, x_n\}$, to identify which class it belongs to. A linear classifier attempts to make such classification decisions based on the value of a linear combination of these features. For details, the classifier f provides a weight vector denoted by $\boldsymbol{w}$, and a bias term denoted by b. The decision logic of classifier f is

$$f(\boldsymbol{x}; \boldsymbol{w}) = \begin{cases} +1, & \boldsymbol{w}^\mathsf{T}\boldsymbol{x} + b \geq 0, \\ -1, & \boldsymbol{w}^\mathsf{T}\boldsymbol{x} + b < 0. \end{cases} \tag{4}$$

The weight vector $\boldsymbol{w}$ and bias term b are computed based on the training set. Let $D = \{(\boldsymbol{x}_i, y_i) \mid y_i \in \{-1, +1\}, i = 1, \ldots, m\}$ be the training set. Then, the classifier parameters $(\boldsymbol{w}, b)$ are obtained by solving the following optimization problem:

$$(\boldsymbol{w}^*, b^*) = \underset{\boldsymbol{w}\in\mathbb{R}^n,\, b\in\mathbb{R}}{\arg\min} \sum_{i=1}^{m} \ell(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_i + b, y_i) + R(\boldsymbol{w}), \tag{5}$$

where function $\ell(\cdot,\cdot)\colon \mathbb{R}\times\mathbb{R} \to \mathbb{R}$ is the (surrogate) loss function, such that $(\boldsymbol{w}, b)$ can achieve the lowest misclassification error on the training set, and function $R(\cdot)\colon \mathbb{R}^n \to \mathbb{R}$ is the regularization to avoid overfitting [6] during the optimization.

Besides, in some machine learning literatures [15, 29], the bias term b may be formulated into the weight vector $\boldsymbol{w}$ such that $\widehat{\boldsymbol{w}} = (\boldsymbol{w}, b) \in \mathbb{R}^{n+1}$ for simpler presentation. Then, correspondingly, we add a constant feature to $\boldsymbol{x}$, i.e., take $\widehat{\boldsymbol{x}} = (\boldsymbol{x}, 1) \in \mathbb{R}^{n+1}$. In this sense, the problem 5 can be written as

$$\widehat{\boldsymbol{w}}^* = \underset{\widehat{\boldsymbol{w}}\in\mathbb{R}^{n+1}}{\arg\min} \sum_{i=1}^{m} \ell(\widehat{\boldsymbol{w}}^\mathsf{T}\widehat{\boldsymbol{x}}_i, y_i) + R(\widehat{\boldsymbol{w}}). \tag{6}$$

Actually, there is a slight difference between the problems 5 and 6. That is, the former will not impose constraints on the bias b, while the latter will indeed give some penalty to the bias b.

Despite its relatively simple usage, the linear classification model is still a preferred choice in many fields due to its adequate inter-predictability and superior efficiency. Especially, as the scale of tasks increases, a series of algorithms are proposed to solve the optimization problem 5 [16, 19, 29, 31, 55, 74]. Unfortunately, these well-designed algorithms often are very complicated and involve massive hyper-parameters, and different tricks in the implementation might make it even worse. Therefore, it can be extremely difficult to guarantee the correctness of the corresponding programs.

It is also worth noting that, the essence of a machine learning algorithm is basically an optimization algorithm, i.e., solving an optimization problem to obtain parameters of its expected model. However, the purpose is radically different from the conventional optimization algorithm, which aims to find an optimal solution, while the goal of a machine learning algorithm is actually to reduce the generalization error. Such fact also renders the analysis of machine learning programs more challenging, since certain assumptions may no longer be held, e.g., with the parameter $(\boldsymbol{w}, b)$ not being the optimal solution of problem 5, with the gradient of loss function w.r.t $(\boldsymbol{w}, b)$ vanishing, and so on.

2.3. Metamorphic testing

Metamorphic testing is an effective software testing method against oracle problems [10]. From its first being published in 1998 until now, metamorphic testing has been well-adopted in various real-life applications [9, 37, 71]. It successfully helped to detect a massive number of software bugs.

The most critical step of applying metamorphic testing is to find an effective metamorphic relation (MR), which is a functional relation established among multiple inputs and outputs of the tested program [10]. Instead of testing by validating the output for a given single input, metamorphic testing tries to test the program by checking whether the relation among several input-output pairs is held or not. An MR indeed encodes a *necessary* property of program (also called metamorphic property in some literature [47, 57, 59]) into a relation among several inputs and outputs. Specifically, these inputs include source inputs and follow-up inputs, with their corresponding outputs being source outputs and follow-up outputs. Generally, source inputs are given by the testing program and the follow-up inputs are generally according to the source inputs and the source outputs. Take the *sine* function as an example. There is a relation that requires $\sin(x) = -\sin(-x)$. To test a program P which realizes function $\sin(x)$, let input x be the source input. Then, the follow-up input $-x$ is calculated by x. Their corresponding outputs, $P(x)$ and $P(-x)$, naturally become source output and follow-up output. They should satisfy the relation that $P(x) = -P(-x)$. In this case, the relation $P(x) = -P(-x)$ is an MR, and it is easily derived from the oddness of the sine function.

As machine learning programs often lack oracle, metamorphic testing also showed its prowess in previous researches. Some metamorphic relations have been carefully designed to test machine learning programs [23, 47, 69, 72]. In our work, we conduct a more in-depth study on the application of MR in testing machine learning programs.

2.4. PI-MR and PD-MR

In previous work, a few MRs were designed for testing machine learning programs. However, these MRs still not be effective enough. We focus on linear classification algorithms in this paper. We conclude the two main weaknesses of existing MRs as follows:

(1) *Lack of efficacy.* The targeted metamorphic property in their designed MR is not sound for testing machine learning programs, such that they cannot detect bugs very effectively. For example, according to the mutation analysis results in [69], some MRs can not detect any bug. Furthermore, the proposed metamorphic relations are too weak to touch the key property of machine learning algorithms. For example, the MR that shuffling the training data does not make use of the statistical properties of machine learning programs, which leads to its poor performance [72, 23].

(2) *Lack of necessity.* The metamorphic relations are too intuitive and lack a theoretical guarantee. For example, some MRs were demonstrated not necessary for the algorithms being implemented [69], and some MRs were defined based on the users' intuitive expectations and specific requirements [72]. As we discussed earlier, the metamorphic relation should encode the necessary property of the algorithm. However, even though the tested learning program is bug-free, the current relations still may not hold. We analyzed this kind of MR in detail in section 4.6.2.

To overcome these weaknesses, we try to borrow the wisdom from the proposed MRs applied to other fields. Through summarizing existing MRs in various fields [9, 76, 37], we find that the current MRs can be divided into two categories, i.e., PI-MR and PD-MR. When an MR is unrelated to the past execution result, we call it *Past-execution Independent MR*, referred to as PI-MR. Contrarily, when an MR utilizes the past execution result, we call it *Past-execution Dependent MR*, referred to as PD-MR. Figure 1 and Figure 2 give the main procedures of PI-MR and PD-MR, respectively. As shown in these figures, the only variance between the two MRs is *how to generate the follow-up input,* i.e., the so-called metamorphic transformation $\mathcal{T}$ in the figures.

We use a typical example (i.e., shortest path problem) to illustrate the difference between PI-MR and PD-MR. Suppose the program $P(x, y)$ implements a search of the shortest path from x to y in a given graph, we can design the following two typical MRs:

(I) Find the shortest paths from y to x, and check whether $P(y, x)$ is the reverse of $P(x, y)$.

(II) Let $(x, v_1, \ldots, v_N, y)$ is the output of $P(x, y)$, and choose any integer k, where $1 \leq k \leq N$. Then, check whether $P(x, y) = P(x, v_k) + P(v_k, y)$.

Figure 1: Past-execution independent metamorphic relation

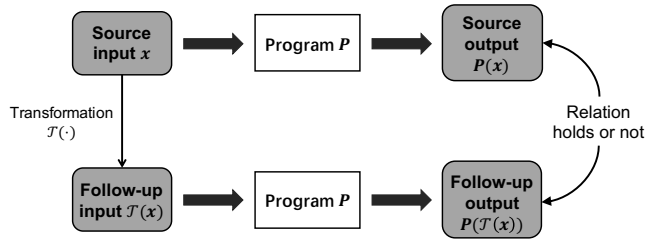
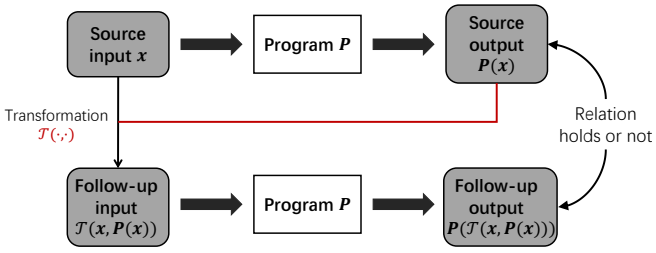


Figure 2: Past-execution dependent metamorphic relation



In MR (I), the corresponding metamorphic transformation $\mathcal{T}(\cdot)$ is $\mathcal{T}(x, y) = (y, x)$ that only requires the source input, thus being PI-MR. However, in MR (II), the transformation $\mathcal{T}(\cdot,\cdot)$ is $\mathcal{T}((x, y), P(x, y)) = (x, v_k), (v_k, y)$ that needs both source input and source output, thus being PD-MR.

Through some existing works that investigate what kind of MRs were effective [11, 7] and the efficacy of MRs across several fields, such as compilers [37, 61], bio-informatics [9, 53], and so on. We conjecture that *PD-MR can perform better than PI-MR*, which is also reflected in some existing research [37, 38, 7]. Based on this conjecture, we later indeed attempt to translate our designed metamorphic property into PD-MRs.

3. Methodology

3.1. Encode stability into PD-MR

When applying metamorphic testing to machine learning programs, we first need to choose a reasonable property as metamorphic property. Since the generalization property is the key property of a machine learning algorithm, we take it as the most straightforward and hopefully effective choice. However, encoding the algorithm generalization into a suitable MR is impractical, because we do not naturally own the true generalization error (to some extent, it is an oracle) of the algorithm. Although one often uses the test accuracy as an estimate of the generalization error, this is just an estimate rather than the true generalization error, as we discussed in Section 2. Based on this consideration, we hereby use algorithm stability instead of generalization as metamorphic property, since the equivalence between stability and generalization error bounds has also been extensively studied [4, 21, 34, 46]. In addition to the feasibility of encoding this property into the MR, stability is also algorithm-specific to be more optimized to analyze the specific machine learning program.

However, there is another problem remaining to be solved. As we discussed earlier in Section 2, a PI-MR is deficient inability to encode the critical property of the algorithm, and it may also destroy the necessity of metamorphic property. Thus, establishing a PD-MR of the algorithm is undoubtedly a better choice. Furthermore, the output of machine learning programs is usually a set of parameters, which is actually a vector. In this case, it is extremely difficult to find and construct a relation between (source and follow-up) outputs. To address this issue, we attempt to build a relation between inference results of a given example. For example, the source and follow-up output of linear classifier are $(\boldsymbol{w}, b)$ and $(\widehat{\boldsymbol{w}}, \widehat{b})$, respectively. Therefore, for a given example $\boldsymbol{x}$, we can identify an equality (or inequality) relation between $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b$ and $\widehat{f}(\boldsymbol{x}) = \widehat{\boldsymbol{w}}^\top \boldsymbol{x} + \widehat{b}$.

Combining the above two points, we can formally define our problem by the following.

Problem 1. *Given source input D, an example $\boldsymbol{x}$, and a linear classification program P, how to generate the follow-up input $\widehat{D}$ based on the algorithm stability, such that we can obtain a metamorphic relation $R(f(\boldsymbol{x}), \widehat{f}(\boldsymbol{x}))$, where*

$$\begin{aligned} f(\boldsymbol{x}) &= \boldsymbol{w}^\top \boldsymbol{x} + b, \quad (\boldsymbol{w}, b) = P(D), \\ \widehat{f}(\boldsymbol{x}) &= \widehat{\boldsymbol{w}}^\top \boldsymbol{x} + \widehat{b}, \quad (\widehat{\boldsymbol{w}}, \widehat{b}) = P(\widehat{D}). \end{aligned} \tag{7}$$

3.2. Two stability-based MRs

At the first glance, to encode the stability only requires the replacement or removal of the original training set (i.e., the source input). In other words, such modifications based on stability is not directly related to the model parameter (i.e., the source output). From this perspective, it may be more appropriate for establishing a PI-MR. But in fact, *How to replace or remove an example of training set* depends on the source output. To further explain this statement, we provide the following two propositions.

Proposition 1. *Given a dataset $D = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^m$, an additional example $\boldsymbol{x}$, and a linear classification algorithm Φ, let the output of the algorithm be $(\boldsymbol{w}, b) = \Phi(D)$. Perturb i-th example of D by*

$$\widehat{\boldsymbol{x}}_i = \boldsymbol{x}_i + \boldsymbol{\xi}, \tag{8}$$

where $\boldsymbol{\xi}$ is any vector orthogonal to $\boldsymbol{w}$ (i.e., $\boldsymbol{w}^\top \boldsymbol{\xi} = 0$), and obtain

$$\widehat{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1, i\neq j}^m \cup \{(\widehat{\boldsymbol{x}}_j, y_j)\}. \tag{9}$$

Suppose the output of the algorithm to be $(\widehat{\boldsymbol{w}}, b) = \Phi(\widehat{D})$. Then, we have that

$$y_i \left(f(\boldsymbol{x}) - \widehat{f}(\boldsymbol{x}) \right) = y_i \left((\boldsymbol{w} - \widehat{\boldsymbol{w}})^\top \boldsymbol{x} + (b - \widehat{b}) \right) \tag{10}$$

is monotone with respect to $y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b)$.

Proposition 2. *Given a dataset $D = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^m$, an additional example $\boldsymbol{x}$, and a linear classification algorithm Φ,*

let the output of the algorithm be $(\boldsymbol{w}, b) = \Phi(D)$. *Remove i-th example of* D, *and obtain*

$$\widehat{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1, i\neq j}^{m} \backslash \{(\boldsymbol{x}_j, y_j)\}. \tag{11}$$

Suppose the output of the algorithm to be $(\widehat{\boldsymbol{w}}, b) = \Phi(\widehat{D})$. *Then, we have that*

$$y_i \left(f(\boldsymbol{x}) - \widehat{f}(\boldsymbol{x}) \right) = y_i \left((\boldsymbol{w} - \widehat{\boldsymbol{w}})^{\mathsf{T}} \boldsymbol{x} + (b - \widehat{b}) \right) \tag{12}$$

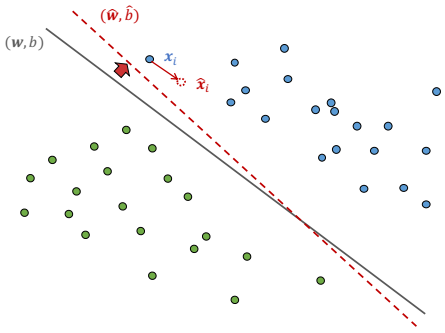
is monotone with respect to $y_i(\boldsymbol{w}^{\mathsf{T}} \boldsymbol{x}_i + b)$.

The proofs of these two propositions are simple and we include them in Appendix A for completeness.

Through the above two propositions, we can comfortably translate the stability (replacement and removal) into two PD-MRs, respectively. Specifically, based on the stability of replacing/removing an example of the training set, we can derive a monotone function with respect to the modified example, which can be naturally encoded into a partial order relation.

As shown in Figure 3 and Figure 4, we provide a visual illustration for each proposition and its corresponding MR. The green points and blue points represent positive and negative instances used for training, and the source input D is made up of these instances. By executing the given linear classification program P with input D, we attain the output $(\boldsymbol{w}, b) = P(D)$. The black line is the linear classification hyperplane determined by $(\boldsymbol{w}, b)$. After replacing/removing an example $\boldsymbol{x}_i$ from dataset, we got the follow-up input $\hat{D}$. The follow-up output $(\widehat{\boldsymbol{w}}, \widehat{b})$ yields the next linear classification hyperplane, which is indicated by the red dashed line. Actually, the proposed two MRs observe the hyperplane movements when replacing/removing different examples. Instead of directly monitoring the hyperplane (i.e., the changes of $(\boldsymbol{w}, b)$), MRs use the result $f(\boldsymbol{x}) = \boldsymbol{w}^{\mathsf{T}} \boldsymbol{x} + b$ for a given example $\boldsymbol{x}$ to build a partial order relation.

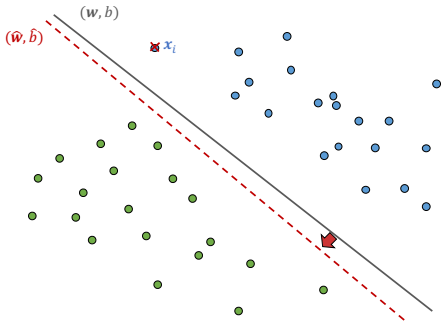
Figure 3: An illustrative example of Proposition 1



3.3. Algorithms

In order to obtain a robust decision on whether the given program violates the stability of the algorithm, we repeat the replacement/removal of the training set D multiple times. The MR of replacement stability is shown in Algorithm 1. The MR of removal stability is shown in Algorithm 2.

Figure 4: An illustrative example of Proposition 2



Algorithm 1 The MR of stability of replacement

Input: Given linear classification program P, source input $D = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{m}$, an additional example $\boldsymbol{x}$.
Output: Whether the MR holds or not.

1: Initialize the sequence $T = []$.
2: Execute program P with input D to get output $(\boldsymbol{w}, b)$.
3: Compute $t_i = y_i(\boldsymbol{w}^{\mathsf{T}} \boldsymbol{x}_i + b)$ for $i = 1, \ldots, m$.
4: Get the index set I of the sorted $t_i, i = 1, \ldots, m$.
5: **for** $k = 1, 2, \ldots, L$ **do**
6: Set $i = I_k$, and choose i-th example $\boldsymbol{x}_i$ of D.
7: Compute a vector $\boldsymbol{\xi}$ such that $\boldsymbol{w}^{\mathsf{T}} \boldsymbol{\xi} = 0$.
8: Add a perturbation to $\boldsymbol{x}_i$, i.e., $\widehat{\boldsymbol{x}}_i = \boldsymbol{x}_i + \boldsymbol{\xi}$.
9: Obtain follow-up input $\widehat{D}$ through replacing $\boldsymbol{x}_i$ by $\widehat{\boldsymbol{x}}_i$, i.e., $\widehat{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1, i\neq j}^{m} \cup \{(\widehat{\boldsymbol{x}}_j, y_j)\}$.
10: Execute program P with follow-up input $\widehat{D}$ to attain follow-up output $(\widehat{\boldsymbol{w}}, \widehat{b})$.
11: Update T:
$T \leftarrow T.\text{append}\,(y_i(\boldsymbol{w}^{\mathsf{T}} \boldsymbol{x} - \widehat{\boldsymbol{w}}^{\mathsf{T}} \boldsymbol{x} + b - \widehat{b})$.
12: **end for**
13: **if** T is monotonically increasing/decreasing **then**
14: **return** True.
15: **else**
16: **return** False.
17: **end if**

In Algorithm 1, we use Gram–Schmidt process to yield the vector $\boldsymbol{\xi}$ that is orthogonal to the vector $\boldsymbol{w}$ [13]. To ensure the perturbation small enough, the Euclidean norm of $\boldsymbol{\xi}$ is fixed by 10^{-3}, i.e., $\|\boldsymbol{\xi}\|_2 = 10^{-3}$. We set the iteration number L to 30. Theoretically, on the one hand, too few iterations may make the lists T and $\widehat{T}$ too short, and thus not be able to detect inconsistencies between them. On the other hand, too large L will cause too many iterations, which costs a lot of time. We did some experiments on different L (i.e., 15, 30, 45, and 60) to verify this theory. The experimental results show that despite the algorithm with an iteration number of 15 performed a little worse, the algorithm with other iteration number performed exactly the same. These results are consistent with our theory. Therefore, setting L by a moderate value is a reasonable choice.

In algorithm 2, the rationale behind the setting of the iteration number L is similar to that in Algorithm 1, and we

Algorithm 2 The MR of stability of removal

Input: Given linear classification program P, source input $D = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{m}$, an additional example $\boldsymbol{x}$.
Output: Whether the MR holds or not.

1: Initialize the sequence $T = []$.
2: Execute program P with input D to get output $(\boldsymbol{w}, b)$.
3: Compute $t_i = y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b)$ for $i = 1, \ldots, m$.
4: Get the index set I of the sorted $t_i, i = 1, \ldots, m$.
5: **for** $k = 1, 2, \ldots, L$ **do**
6: Set $i = I_k$, and choose i-th example $\boldsymbol{x}_i$ of D.
7: Obtain follow-up input $\widehat{D}$ through removing $\boldsymbol{x}_i$ from D, i.e., $\widehat{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1, i\neq j}^{m}$.
8: Execute program P with follow-up input $\widehat{D}$ to attain follow-up output $(\widehat{\boldsymbol{w}}, \widehat{b})$.
9: Update T:
$T \leftarrow T.\text{append}\,(y_i(\boldsymbol{w}^\top \boldsymbol{x} - \widehat{\boldsymbol{w}}^\top \boldsymbol{x} + b - \widehat{b})$.
10: **end for**
11: **if** T is monotonically increasing/decreasing **then**
12: **return** True.
13: **else**
14: **return** False.
15: **end if**

Table 1
Algorithm list

Type	Name
Logistic Regression	NAG [54]
	Newton [3]
	LBFGS [77]
Support Vector Machine	APG_L1 [32]
	APG_L2 [32]
	ADMM_L1 [73]
	ADMM_L2 [73]
	SQP_L1 [15]
	SQP_L2 [15]

also set L to 30.

4. Evaluation

In this section, we evaluate our designed MRs, and compare them to MRs proposed by existing work on their bug detection effectiveness, for programs of nine well-known linear classification algorithms.

4.1. Research questions

We aim to answer the following two research questions:

RQ1 (Effectiveness): How effective are our designed MRs on detecting bugs for linear classification programs, as compared to MRs proposed by existing work?

RQ2 (Sensitivity): How sensitive are our designed MRs on detecting bugs with respect to their different stability consequences for linear classification programs, as compared to MRs proposed by existing work?

Table 2
Mutation operators of mutmut

Name	Description	Example
AOR	Replace arithmetic operator	'+' to '-'
LOR	Replace logical operatior	'and' to 'or'
ROR	Replace relational operator	'==' to '!='
SOR	Replace shift operator	'<<' to '>>'
ASR	Replace shortcut assignment operator	'+=' to '='
KR	Replace keyword	'break' to 'continue'
AVR	Replace assignment value	'x = 1' to 'x = None'

Table 3
The newly added mutation operator

Name	Description	Example
POA	Add parameter operation	'1+x' to '1+0.3*x'

4.2. Experimental subjects

We selected nine well-known linear classification algorithms as shown in Table 1, including three logistic regression algorithms and six support vector machine algorithms. They are either classic algorithms in the linear classification field or recently published in top journals/conferences with nice credits. Since their source codes are not directly accessible, we chose to implement them by ourselves, according to their algorithms presented in corresponding documents or papers. By a careful manual verification process, we utilized our implementation as corresponding golden versions. To ensure the credibility of the golden versions, we also compared the execution results of our implemented program and the linear classification program in the scikit-learn library [49]. The error (i.e., the Euclidean norm of two outputs) of the output results (i.e., a pair of $(\boldsymbol{w}, b)$) of these two programs is less than 10^{-6}, suggesting the nice quality of our implementation.

In order to better evaluate different MRs' bug detection effectiveness, we adopted popular *mutation analyses* [48, 20, 28] in software engineering, and generated mutants for those programs, each of which contains one inserted syntactic bug. Mutation analyses aim to generate mutants with syntactic bugs for simulating realistic bugs in practice [33, 64], and are used as a relatively promising way to evaluate different treatments' bug detection effectiveness. We used a well-established mutation tool for Python, i.e., mutmut [28], and Table 2 presents its supported seven mutation operators for generating mutants.

However, we observe that mutmut's supported mutation operators have quite a limited ability to generate numerical bugs, which are extremely common in machine learning programs due to massive mathematical calculations. To be specific, the only operator to generate numerical bugs in mutmut is AOR, which substitutes an arithmetic operator + to −, tending to largely interrupt a program's logic and trigger a program crash, thus easily detected. Therefore, in order to better simulate such common numerical computational bugs in machine learning, we additionally designed mutation operation POA as in Table 3, which inserts a random constant as coefficient c to programs' numeric calculations, including

Table 4
Details of generated mutants

Mutants (#)	Algorithms									Sum
	NAG	Newton	LBFGS	ADMM_L1	ADMM_L2	APG_L1	APG_L2	SQP_L1	SQP_L2	
Num	101	96	240	199	205	123	97	88	116	**1,265**

a calculation expression, an assignment expression, a function parameter, and so on. This coefficient follows a normal distribution $\mathcal{N}(0, 10)$.

For example, the Code Listing 1 shows a practical numerical calculation in the ADMM algorithm, and line 2 contains a square root calculation. Simply altering the symbol '+' to '−' will cause illegal calculation, and thus lead to the program exception (e.g., RuntimeWarning) or even a program crash, which can be easily discarded, thus being relatively worthless. By using POA, Code Listing 2 can be generated by adding a coefficient of 0.5 to the formula in line 2. This type of bug often appears when misreading the algorithm formula in practice, and we believe it can be indeed more relatively worthy in bug detection evaluation.

After that, combining operators in both Table 2 and Table 3, we eventually generated 1,265 mutations in total. Details are in Table 4. We discarded not executable mutants or some clearly equivalent ones with manual checking. Considering that we only focus on generating mutants for machine learning programs' training part, which is relatively short in code length, we believe such mutants can be already adequate for evaluating MRs.

Code Listing 1
A code piece of numerical calculation in ADMM algorithm

```
1 theta = np.linalg.norm(u,'fro') / tau_old
2 c = 1 / np.sqrt(1+theta*theta)
3 tau = tau_old * theta * c
```

Code Listing 2
An example of numerical computation bug

```
1 theta = np.linalg.norm(u,'fro') / tau_old
2 c = 1 / np.sqrt(1+0.5*theta*theta)
3 tau = tau_old * theta * c
```

4.3. Experimental design

Then, we introduce our experimental design, including dataset preparation for feeding into the former machine learning programs, and descriptions about our considered MRs for comparisons later.

Datasets. We randomly generate datasets (i.e., the original dataset D) due to the following reasons. Firstly, our designed MRs are indeed dataset-independent, not specifically designed for a specific task of the dataset, thus making dataset selection unrestricted. Secondly, existing studies have also pointed out that the randomly generated dataset can work better in detecting bugs [22]. Although several existing real-world datasets are applicable for testing the effectiveness of machine learning algorithms, they maybe not as sensitive as the randomly generated datasets in detecting bugs of the machine learning program. Therefore, we adopted scikit-learn [49] to generate synthetic datasets randomly.

Moreover, in order to reduce the running time of our experiment to save time, we restricted the scale of randomly generated datasets by 300 and its concerned feature number between two to ten. Then, for any generated dataset, we also divided it into two parts for our latter usage, a training set (240 in size) and a test set (60 in size). The scale of our datasets is much larger than some previous work [69, 72], and the feature number also covers feature numbers used in these work. Since larger datasets are usually more statistically significant, we believe our generated datasets are not only enough for the linear classification task, but also produce feasible running time for training in experiments. Furthermore, as we discussed earlier, a multi-class classification task can be converted easily into multiple binary classification tasks, thus we directly apply these algorithms to a binary classification task. Thus, only two labels are considered in generating datasets.

Table 5
Six MRs in previous work

No	Description
MR-1	Shuffle of training data
MR-2	Permutation of training and testing features
MR-3	Permutation of class labels
MR-4	Addition of uninformative attributes
MR-5	Consistence with re-prediction
MR-6	Additional training samples

MRs. To evaluate our designed MRs' bug detection effectiveness, we have prepared both clean buggy programs as long as randomly generated datasets. For comparisons, we aim to use both our designed MRs and those proposed by existing work for bug detection later. Here, we briefly introduce them as follows.

We chose six classical MRs (listed in Table 5) proposed in previous work for comparisons. All of these six MRs can be applied directly to the linear classification programs, and have been evaluated with superior results in previous work. For example, MR-2 is previously found to find all 12 bugs of linear-SVM and RBFSVM [23], and MR-3 achieved a high killing rate of 71.4% (15 out of 21) [69]. The details of these MRs are as follows. In these MRs, dataset D is source input,

and dataset $\widehat{D}$ is follow-up input.

- **MR-1: Shuffle of training data** [23, 72]. Dataset $\widehat{D}$ is obtained by shuffling D. The accuracy of models trained by D and $\widehat{D}$ should be the same.
- **MR-2: Permutation of data features** [69, 23]. Dataset $\widehat{D}$ is obtained by permuting the features of D. The accuracy of models trained by D and $\widehat{D}$ should be the same.
- **MR-3: Permutation of class labels** [69]. Dataset $\widehat{D}$ is obtained by permuting the class labels of D. The accuracy of models trained by D and $\widehat{D}$ should be the same.
- **MR-4: Addition of uninformative attributes** [69, 72]. Dataset $\widehat{D}$ is obtained by adding an uninformative attribute to all data in D. The accuracy of models trained by D and $\widehat{D}$ should be the same.
- **MR-5: Re-prediction** [69]. Dataset $\widehat{D}$ is obtained by adding a random sample to the testing dataset to D for re-prediction. The prediction results of models trained by D and $\widehat{D}$ should be the same.
- **MR-6: Additional training samples** [69]. Dataset $\widehat{D}$ is obtained by duplicating samples with a specific label in D. The prediction results of the samples with the specific label of models trained by D and $\widehat{D}$ should be the same.

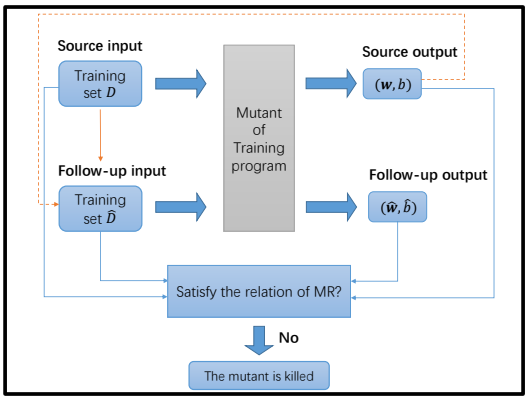
4.4. Experimental process and setup

We now introduce our experimental process with prepared resources and set up to answer the aforementioned two research questions.

Process. With subjects (golden versions and corresponding mutants), datasets (both training and testing sets), and MRs prepared, we now introduce our experimental process. For any considered MR, we aim to check whether it is satisfied or not upon any considered mutant fed by generated datasets, according to a typical procedure presented as in Figure 5. We call a mutant can be "killed" by an MR when this MR is eventually violated upon this mutant following this procedure. Note that, due to some statistical and numerical bias during machine learning programs' execution, we choose randomly repeat the procedure for any mutant and MR 100 times, only an MR is violated by a mutant more than five times out of 100, we recognized as its killing upon this mutant successfully.

In this procedure, a training set D is generated as the source input, and then fed into a program (clean or buggy). To be specific, the program actually points to a program written to train a linear classifier and produce weight vector and bias as $(\boldsymbol{w}, b)$, possibly with an unknown bug. Then, if one uses this classifier to predict, the output $(\boldsymbol{w}, b)$ would be the model parameter, and used to give predictive results for the given testing set. The predictive results and model parameter $(\boldsymbol{w}, b)$ are collectively referred to as the source output. Based on the source output and source input, the follow-up input $\widehat{D}$ can be generated. Similarly, the follow-up output consists with output $(\widehat{\boldsymbol{w}}, \widehat{b})$ that is the output of executing program with input $\widehat{D}$, and its corresponding predictive results of the same test set. Any MR actually points out some relationships among source input and output, follow-up input and output, that must be satisfied, otherwise violated.

Figure 5: Procedure of MR killing mutant



Setup. We designed the following independent variables to control the experiments:

- *Subject.* We generate more than 1,265 mutants and used them as our subjects for evaluating MR's bug detection effectiveness.
- *MR.* We controlled to select different MRs in experiments. For selection, we consider both our designed MRs as MR-P1 (for Proposition 1) and MR-P2 (for Proposition 2), and six MRs proposed in existing work, i.e., from MR-1 to MR-6 as in Table 5.
- *Algorithm.* We used a total of nine different linear classification algorithms for generating mutants. Different concerned algorithms in mutants may lead to different experimental results when evaluating MR's bug detection effectiveness.
- *Stability degree.* We study how different MR can detect bugs with respect to different stability degree interruption to existing programs. We divided them into three categories by considering their associated accuracy consequences. For any mutant leading to a large accuracy change (more than an absolute value of 0.05), we consider its stability degree to be "*severe*". For any leading to a relatively small accuracy change (less than an absolute value of 0.05), we consider its stability degree to be "*light*". For those otherwise, we consider it to be "*negligible*" or simply refer to it as "*other*" category.

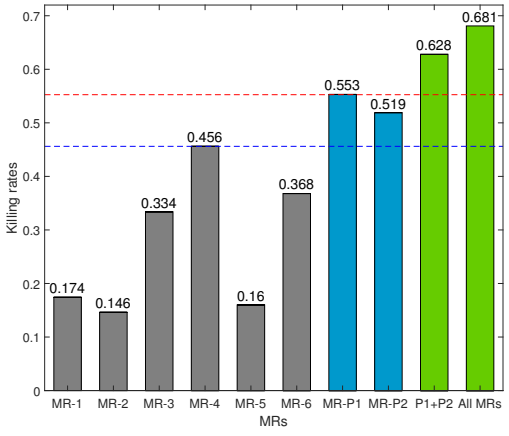
To measure the bug detection effectiveness, we focus on the following dependent variable:

- *Killing rate.* It refers to the proportion of killed mutants among all mutants considering any specific MR, which is for measuring this MR's bug detection effectiveness.

Table 6
Killed mutants when applying different MRs on experimental subjects

MRs	Subjects									Sum
	NAG	Newton	LBFGS	ADMM_L1	ADMM_L2	APG_L1	APG_L2	SQP_L1	SQP_L2	
MR-1	1	2	21	55	70	4	7	30	30	**220**
MR-2	2	2	24	56	67	3	7	3	21	**185**
MR-3	74	27	139	53	68	13	16	16	17	**423**
MR-4	73	13	133	83	104	65	44	34	28	**577**
MR-5	7	4	28	57	68	6	8	4	20	**202**
MR-6	53	10	103	88	89	32	27	40	24	**466**
MR-P1	**82**	23	**144**	**115**	**113**	52	**53**	**69**	**48**	**699**
MR-P2	54	**31**	128	94	109	**110**	47	38	45	**656**
P1+P2	**82**	**38**	**145**	**120**	**121**	**110**	**54**	**73**	**51**	**794**
All MRs	**84**	**46**	**158**	**136**	**123**	**121**	**64**	**79**	**51**	**862**

Figure 6: Ovarall killing rate of MRs



All experiments were conducted on a CPU server with AMD EPYC 7401 24-Core processor and 128G of memory, running Ubuntu 20.04.1 with GNU/Linux kernel 5.4.0. Our code and experimental data are also available at `https://github.com/yingzhuoy/MRs-of-linear-models`.

To answer RQ1 (Effectiveness). We conducted experiments for our designed MRs upon all generated mutants for programs of different linear classification algorithms, as compared to existing MR-1 to MR-6, and study their effectiveness on bug detection, suggested by their corresponding killing number of mutants and killing rates.

To answer RQ2 (Sensitivity). We divided mutants according to their different interruption degrees to programs' stability, and study their effectiveness upon mutants concerning different stability degrees, i.e., "severe", "light", and "other", as mentioned before to see how sensitive our designed MRs can be on detecting bugs that lead to different stability interruption degrees.

4.5. Experimental results and analyses

We report and analyze experimental results, and answer the preceding research questions in turn.

4.5.1. RQ1: effectiveness

Figure 6 shows the total killing rate of concerned MRs on all subjects (i.e., 1,265 mutants). We use MR-1 to MR-6 to represent studied MRs proposed by existing work, whose details can also be found in Table 5, while MR-P1 (or P1) and MR-P2 (or P2) represent our proposed MRs in this work, i.e., aforementioned Proposition 1 and 2 in Section 3. MR-P1 and MR-P2 can achieve a nice killing rate of 55.3% and 51.9% mutants among all 1,265 ones, much more than those of existing MRs (MR-1 to MR-6), i.e., only 14.6%–45.6% in killing rates per MR. Moreover, when combing our MR-P1 and MR-P2 together, the killing rate (mutants that are killed by either MR-P1 or MR-P2) can be up to 62.8% among all 1,265 mutants, already covering 92.2% of all detected mutants by all studied eight MRs (68.1% mutants). Still, our MRs' killing rate indeed seems to be not so high, it may be due to the following reasons: (1) there may exist some statistically equivalent mutants in fact, and this denotes the inevitable problem of mutation analyses [42], and has not been perfectly addressed yet, (2) there may also exist some mutants that do not destroy the program stability, which are not specifically designed in our MRs' consideration, i.e., out of scope. We also give a few examples in our case study section (section 4.6), in order to look a little deeper into those survived mutants for your reference.

Table 6 shows some details of killed mutants when different MRs are applied to subjects associated with different LC algorithms. As shown in Table 6, our approach can kill the most mutants for all subjects, with either MR-P1 or MR-P2 achieving the largest number of killed mutants. Especially, for most subjects, MR-P1 or MR-P2 killed far more mutants than the other six MRs. For example, for the ADMM_L1 subject, although MR-6 killed 88 mutants successfully, which has already been the most effective one among all the six MRs (MR-1 to MR-6) for comparison, our proposed MR-P1 can kill 115 mutants, with 27 more mutants (with a 38.5% degree larger in the killing number) than MR-4. Furthermore, for the APG_L1 subject, MR-P2 killed 110 mutants (69.2% larger in number), while the most effective MR-4 in existing work only killed 65 mutants.

Table 7
The number of mutants of three stability degree categories

Category	Subjects								
	NAG	Newton	LBFGS	ADMM_L1	ADMM_L2	APG_L1	APG_L2	SQP_L1	SQP_L2
Light	13	10	25	47	20	90	16	14	12
Severe	59	28	105	80	104	30	48	35	36
Other	29	58	110	72	81	3	33	39	68

Table 8
Killing rate of MRs on with respect to mutants in the "light" category

MRs	Subjects									All Subjects
	NAG	Newton	LBFGS	ADMM_L1	ADMM_L2	APG_L1	APG_L2	SQP_L1	SQP_L2	
MR-1	0.0%	0.0%	20.0%	2.1%	10.0%	1.1%	0.0%	100.0%	66.7%	12.6%
MR-2	7.7%	0.0%	24.0%	2.1%	5.0%	1.1%	0.0%	0.0%	8.3%	4.5%
MR-3	69.2%	30.0%	100.0%	4.3%	0.0%	0.0%	6.3%	0.0%	8.3%	16.6%
MR-4	76.9%	30.0%	100.0%	29.8%	70.0%	47.8%	31.3%	100.0%	83.3%	55.9%
MR-5	0.0%	0.0%	24.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	2.4%
MR-6	46.2%	0.0%	60.0%	17.0%	15.0%	11.1%	12.5%	78.6%	41.7%	24.3%
MR-P1	92.3%	30.0%	100.0%	59.6%	80.0%	27.8%	100.0%	100.0%	83.3%	60.3%
MR-P2	53.8%	30.0%	100.0%	38.3%	65.0%	85.6%	87.5%	100.0%	66.7%	72.5%
P1+P2	92.3%	40.0%	100.0%	66.0%	85.0%	85.6%	100.0%	100.0%	83.3%	83.4%
All MRs	92.3%	80.0%	100.0%	72.3%	95.0%	97.8%	100.0%	100.0%	83.3%	91.5%

When further combining the killed mutants of different subjects together for an MR individually, our proposed MR-P1 and MR-P2 also show their significant superiorities with being the best two MRs clearly, i.e., 699 for MR-P1, and 656 for MR-P2. in total. Statistically, MR-P1 and MR-P2 killed 122–514 and 79–471 more mutants than any existing MR, with an increasing rate of 21.1%–277.8% and 13.7%–254.6%, respectively. If one compares them to the best MR in existing work, i.e., MR-4, which killed 577 mutants in total, our proposed MRs (P1+P2) indeed killed 37.6% (MR-4) to 329.2% (MR-2) more mutants, suggesting their great effectiveness in detecting buggy programs, as well as their significant superiorities over existing MRs.

Therefore, we answer RQ1 as follows: *our proposed MRs (MR-P1 and MR-P2) show nice effectiveness on their bug detection. When compared to existing MRs, they also achieve significant superiorities (21.1%–277.8% and 13.7%–254.6% improvement on killing rates) over the studied six MRs proposed by existing work.*

4.5.2. RQ2: sensitivity

As aforementioned, in order to better evaluate our proposed MRs' effectiveness and their sensitivity on bug detection with respect to different stability interruptions, we divided all 1,265 mutants into three categories: "severe", "light", and "other", according to their stability interruption degree. To do so, we hereby used *accuracy* as the standard and then partitioned all mutants as follows: (1) mutants leading to a large accuracy change into the "severe" category, (2) mutants leading to a relatively small accuracy change into the "light" category, (3) the remaining ones into the "other" category.

Figure 7: Killing rate under different thresholds

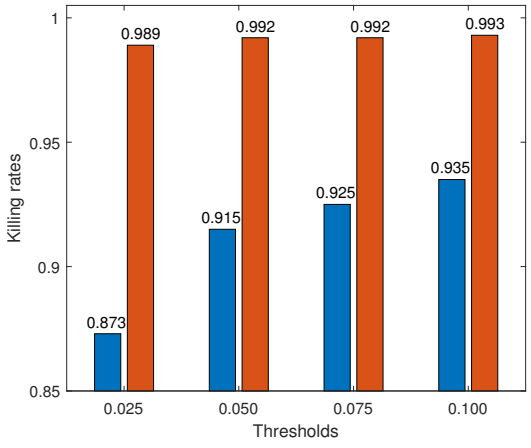


Figure 7 shows the killing rate of all MRs of "severe" and "light" categories under different thresholds (i.e., 0.025, 0.05, 0.075, and 0.1). According to the figure, the killing rate of all MRs increased with the increase of thresholds in both "severe" and "light" categories and the killing rate of "severe" category are higher than "light" category. These results indicate when accuracy changes more, the stability is more damaged and bugs can be easier detected, which shows the rationality of using accuracy as the standard. According to our statistics, different thresholds have little effect on the results of MR evaluation, so we randomly choose 0.05 as the threshold to determine stability degree in our evaluation.

By investigating different effectiveness to different categories, we aim to evaluate MR's effectiveness sensitivities

Table 9
Killing rate of MRs with respect to mutants in the "severe" category

MRs	Subjects									All Subjects
	NAG	Newton	LBFGS	ADMM_L1	ADMM_L2	APG_L1	APG_L2	SQP_L1	SQP_L2	
MR-1	1.7%	7.1%	12.4%	67.5%	65.4%	10.0%	14.6%	45.7%	58.3%	35.2%
MR-2	1.7%	7.1%	13.3%	68.8%	63.5%	6.7%	14.6%	8.6%	52.8%	32.2%
MR-3	96.6%	82.1%	96.2%	63.8%	65.4%	43.3%	31.3%	45.7%	41.7%	68.4%
MR-4	96.6%	25.0%	93.3%	82.5%	86.5%	73.3%	81.3%	54.3%	44.4%	78.9%
MR-5	11.9%	10.7%	17.1%	71.3%	65.4%	20.0%	16.7%	11.4%	52.8%	36.2%
MR-6	78.0%	28.6%	75.2%	81.3%	82.7%	73.3%	52.1%	57.1%	44.4%	69.9%
MR-P1	96.6%	53.6%	97.1%	95.0%	92.3%	90.0%	75.0%	85.7%	91.7%	89.9%
MR-P2	64.4%	85.7%	84.8%	92.5%	91.3%	100.0%	68.8%	68.6%	91.7%	83.8%
P1+P2	96.6%	96.4%	97.1%	97.5%	99.0%	100.0%	77.1%	97.1%	100.0%	96.0%
All MRs	100.0%	100.0%	99.0%	97.5%	100.0%	100.0%	97.9%	100.0%	100.0%	99.2%

on bugs with different stability interruptions. We specifically look into the experimental results for category "severe" and "light", as shown in Table 8 and Table 9.

Table 8 shows the killing rate of MRs with respect to mutants in the "light" category ("light" mutants). We can observe that MR-P1 and MR-P2 can achieve nice 60.3% and 72.5% killing rates when applying to "light" mutants of all subjects, much higher than any MR from MR-1 to MR-6 (highest at 55.9%). Besides, for the APG_L2 subject, MR-P1 achieves a killing rate of 100% and MR-P2 achieves a killing rate of 87.5%, but the highest killing rate from MR-1 to MR-6 is only 31.3% (MR-4). Moreover, the joint killing rate of MR-P1 and MR-P2 is also very high on most of the subjects. In subject NAG, LBFGS, APG_L2 and SQP_L1, the joint killing rates all exceed 90.0% and are the same as the joint killing rate of all MRs. On some subjects, MR-P1 and MR-P2 behave not so well, especially the Newton subject (the killing rate of MR-P1 and MR-P2 are both 30.0%) and the ADMM_L1 subject (the killing rate of MR-P1 is 56.9%, the killing rate of MR-P2 is 38.3%). In these subjects, the killing rate of other MRs is also very low, indicating that bugs in these mutants are not easy to be detected. Still, our MR-P1 and MR-P2 can achieve the best killing rate in these cases (30.0% to Newton, and 59.6% to ADMM_L1).

Table 9 shows the killing rate of MRs with respect to mutants in the "severe" category ("severe" mutants). MR-P1 and MR-P2 achieve very high killing rates of 89.9% and 83.8% separately when applying to all "severe" mutants, which is also higher than any other MRs from MR-1 to MR-6. Besides, the killing rate of either MR-P1 and MR-P2 is undoubtedly the highest for almost every subject, except for the APG_L2 subject, whose killing rate of MR-P1 (75.0%) is slightly lower than that of MR-4 (81.3%). Moreover, MR-P1 and MR-P2 showed much higher killing rates than other MRs in many subjects. For example, in the SQP_L2 subject, both MR-P1 and MR-P2 have the killing rate 91.7%, while the killing rate from MR-1 to MR-6 is at most 58.3% (MR-1). The joint killing rates of MR-P1 and MR-P2 even exceed 95.0% in most subjects, suggesting its great effectiveness in detecting bugs of "severe" mutants.

When combining Table 8 and Table 9, the two categories of mutants indeed lead to varied stability interruptions to programs, as well as their different difficulties to be detected. The killing rates of "severe" mutants are relatively lower than those of "light" mutants. This also echoes our intuition that bugs that break properties more severely are relatively easier to be detected. We can still observe that our proposed MR-P1 and MR-P2 are stably effective and achieve nice superiorities for both "severe" and "light" mutants, suggesting their nice sensitivity with respect to bugs with different stability interruptions.

Therefore, we answer RQ2 as follows: *our proposed MR-P1 and MR-P2 show their great effectiveness with nice sensitivity. Its nice effectiveness holds with respect to bugs with varied stability interruptions to programs.*

4.6. Case studies

In this section, we present case studies in two aspects. First, we analyze some stubborn mutants with the same performance as the golden version. Bugs in these mutants are difficult to be detected by accuracy or some PI-MRs, but can be detected by our PD-MRs. Second, we discuss whether the previously proposed MRs successfully encode a necessary property of algorithms.

4.6.1. Stubborn mutants

A few generated mutants confirm the machine learning algorithms' capability of hiding errors. These mutants should have performed poorly, but the fact is just the opposite. This phenomenon not only explains the difficulty in testing machine learning programs, but also clarifies the gap between testing conventional software and machine learning programs.

Code Listing 3 provides a typical example. In subject NAG, a conditional statement `If` in code piece of BUG-73 conducts a line search to ensure the value of `Lw` (i.e., loss function) decreased. In this case, the mutation operator AOR alters the symbol '`*`' to '`/`', which causes the sufficient decrease of `Lw` to be destroyed. This mutant is highly inefficient in convergence or even cannot converge, but it achieved the same accuracy as the golden version on all ten datasets. While the mutant was not killed by any of the existing MRs (MR-1 to MR-6), it was killed by MR-P1 and MR-P2. Similarly, BUG-87 (shown in Code Listing 4) is a mutant that

modifies the calculation of the `Sigmoid` function. Actually, this mutant will be equivalent to the golden version if we applied this mutated `Sigmoid` in both training and testing process. In our experiment, we apply the mutated *sigmoid* function in the training process and the correct *sigmoid* function in the test process. Thus it should be recognized as a bug. MR-P1 and MR-P2 killed this mutant but the other six MRs did not.

Code Listing 3
BUG-73 in NAG

```
#----bug----
#if Lw < L0 - l * alpha * (g0.T*g0):
if Lw < L0 - l * alpha / (g0.T*g0):
  break
else:
  l = beta * l
```

Code Listing 4
BUG-87 in NAG

```
def sigmoid(x):
  # avoid overflow
  #----bug----
  #return .5 * (1 + np.tanh(.5 * x))
  return .5 * (1 + np.tanh(0.6698714043249333 *
      x))
```

Code Listing 5
BUG-97 in NAG

```
what = np.array(w_hat).flatten()
#----bug----
#b = w_hat[-1]
b = w_hat[+1]
w = w_hat[0:w.shape[0]-1]
```

A more interesting example is BUG-97 (as shown in Code Listing 5), which sets the value of the bias b by the first element of vector $\hat{\boldsymbol{w}}$ rather than the last element. Such mutation operator significantly changes the bias of the trained model, yet the model obtained by this mutant did not be impaired and even achieved higher accuracy than the model obtained by the golden version (with an increase from 88.3% to 93.3%). It should be more difficult to be killed, but four MRs (MR-2, MR-4, MR-6 and MR-P1) succeeded.

4.6.2. The necessity of MRs

Since most of MRs in previous work were intuitive, they are probably not consistent with machine learning theories sometimes. We take the MR-4, which attain the best performance among the previous six MRs, as an example to illustrate this issue. As in section 4.3, MR-4 adds an informative attribute to all examples of training set D to create the follow-up input $\hat{D}$. This transformation is actually equivalent to introducing an additional bias b' to model parameter, i.e., which makes the follow-up output be $(\boldsymbol{w}, b, b')$. In this case, the bias b will be added into the regularization function, but the additional bias b' will not. Thus, we can easily find the following two situations where MR-4 cannot hold.

(1) The algorithms that do not have the bias,

(2) The algorithms that apply the regularization.

In practice, the loss functions of most machine learning algorithms contain the regularization function, which severely limits the usage of MR-4. We also used scikit-learn to verify this analysis: When we set the bias to false (`intercept = False`), or set the hyperparameter `C` of regularization larger than three, MR-4 is violated.

Although we have tried our best to avoid the mutation operator on the algorithms' hyperparameters (e.g., mutate `C = 1.0` into `C = 1.5`) because the modification of hyperparameters should not be considered as a program bug, Some extreme examples are still inevitable. For example, BUG74 in Newton subject (as shown in Code Listing 6) is equal to modifying the coefficient of the regularizer. In this case, only MR-4 killed this mutant. The example was a little particular, but similar mutants may also exist in other algorithms of our experiment. Discarding such mutants will only reduce the performance of MR-4 and improve the efficacy results of other MRs. Incidentally, in our experiment, we adopted the same hyperparameter setting as scikit-learn, which applies the regularization by default. For the above reasons, the advantages of MR-4 in our experiment were also seriously suspected.

Code Listing 6
BUG-74 in Newton

```
tmp = beta.copy(); tmp[n-1] = 0;
#----bug----
#grad = - X.T * (y - self.p1(X * beta)) + 1*tmp
grad = - X.T * (y - self.p1(X * beta)) + 4.166*tmp
```

4.7. Threats analyses

The creation of datasets is one of *external threats* in our experiment. To alleviate this threat, we tried two aspects. (1). Refer to previous work to set the data scale. In two previous works, the size of the training dataset is a random number from 20 to 50 and the feature numbers are two and four respectively [69, 72]. Our dataset size, 300, is larger than the size in previous works and our feature number, randomly selected from two to ten, also covers the numbers in previous works. Thus, our generated datasets are enough for the linear classification task. (2). Randomly generate data values. Since our MRs are dataset-independent and randomly generated datasets can work better in detecting bugs [22], we randomly generated the data values to make the datasets of better diversity. We used the built-in library in scikit-learn [49] to randomly generate datasets, which makes the generated results with no bias.

The mutants generation is another *external threat*. We selected a widely used Python mutation tool *mutmut* [28], which was used to generate mutants for machine learning programs [23]. This tool supports many traditional mutation operators, which can create many different kinds of bugs. Besides, we added a new mutation operator POA to make this tool more adaptable to machine learning programs. We

generated over 1,000 mutants in total using this tool, which is enough for our experiment.

Possible bias in MR selection is also an *external threat*. To avoid bias, we selected six MRs for comparison from three previous works [69, 72, 23]. Some of these MRs appeared in more than one work, which indicates these MRs are classical and widely used. As for the performance, we selected both well-performed MRs and badly performed MRs in previous work to ensure the diversity of MRs.

For the above efforts we made to avoid external validity, our experiment settings should be appropriate.

The *internal threat* mainly comes from the implementation of nine linear classification algorithms we used. Although these algorithms were published in famous publications, they were not open source. We implemented these algorithms on our own according to their papers. To avoid possible faults we would make during the implementation, we carefully compared the training results of our implementation with the results of scikit-learn [49] and ensure the tolerance less than 10^{-6}. In this way, this internal threat is generally avoided.

5. Related work

5.1. Testing on machine learning program

Metamorphic testing has been one of the most popular techniques used for testing machine learning programs. Various MRs have been designed for different machine learning algorithms, e.g., k-nearest neighbor classifier, support vector machine, etc. In the following, we briefly overview related work in this direction.

Xie et al. proposed 11 MRs to test programs that implement k-nearest neighbor (*k*NN) algorithm or the Naïve Bayes (NBC) algorithm [69]. These MRs were divided into two types according to whether they encode the necessary properties of algorithms or not. MRs encoding necessary properties were typically used for (software) verification, and those remaining ones were typically used for software validation. These MRs were evaluated on Weka [25] and some introduced mutants of Weka.

Dwarakanath et al. designed four MRs for support vector machine (SVM) and Deep Residual Network (ResNet) [23], respectively. These MRs are specifically designed for the characteristics of the convolution operation or the RBF kernel function, including the permutation of input channels, the shift of training and test features, etc. Experimental results showed that such proposed MRs effectively killed 71% of generated mutants.

Xie et al. applied metamorphic testing to unsupervised machine learning fields [72]. They focused on the clustering algorithms and proposed 11 generic MRs of six kinds of data transformation. However, these MRs were designed according to user expectations instead of solid machine learning theories.

Cheng et al. mainly focused on the properties of implementation bugs in machine learning programs [14]. The Weka programs implement four algorithms (NBC, *k*NN, SVM, and Decision Tree) were selected as the golden version. The experiments were conducted on four different kinds of datasets, the dataset of different distribution, dataset of different sizes, imbalanced dataset, and special dataset. The experimental results showed that there are some logically nonequivalent but statistically equivalent mutants, and some of these mutants were very stubborn that they had the same results with golden versions on all datasets used in the experiment.

5.2. Testing on data, model, and framework

As in the survey of machine learning testing [75, 5], many studies focus on other kinds of defects in machine learning systems besides learning programs. We list some typical researches in the following.

Defects in data. Since data is one of the most critical elements in machine learning, there were many works aiming at detecting bugs in data. Some of those works aimed at debugging errors in polluted training datasets [30, 41], some focused on detecting improper model inputs [44, 66], and some focused on investigating the effects of dirty data [52].

Defects in models. Besides machine learning programs, the trained model is another key component of the machine learning system. In recent years, lots of work has been proposed to test deep learning models.

Some of these studies borrowed the idea of structural coverage in conventional software testing and proposed a series of coverage criteria for deep neural networks, such as DeepXplore [50], DeepGauge [40], DeepTest [62], etc. Despite using the concept of coverage, many other methods were proposed for testing deep neural networks. For example, Surprise Adequacy [35] measured how many surprises did give input give compared to the training dataset. DeepCheck [24] used the idea of program analysis, especially symbolic execution to test deep neural networks.

Defects in frameworks. This line of work mainly concentrates on the bugs in the code of deep learning frameworks. For example, Pham et al. proposed to test the implementation of deep learning libraries (TensorFlow, CNTK and Theano) through differential testing [51]. There were also many other works in testing machine learning framework [60, 67].

6. Conclusion and future work

The quality assurance for machine learning systems has gained growing popularity. However, existing researchers have mainly focused on the quality of the machine learning models deployed in the systems, and neglected the quality of the learning programs, which actually the foundation for the quality of thus trained models. We, in this paper, focus particularly on the quality of machine learning programs, especially those implementing linear classification algorithms. To do so, we derive fundamental stability properties from linear classification algorithms' kernel statistical nature and propose two PD-MRs accordingly for effectively detecting bugs in learning programs potentially.

We also experimentally evaluated our proposed MRs upon nine well-known linear classification algorithms, and observe a significant improvement over six benchmark MRs proposed by existing work, with 37.6–329.2% bugs being detected. Our experimental results also somehow confirm our conjecture that PD-MRs can be more effective than PI-MRs, which is also reflected in some other research [37].

Our research also deserves further research along this line. On one hand, our proposed MRs' specialties and their effectiveness in detecting different types of machine learning bugs might deserve more researches. On the other hand, we also expect more effective PD-MRs to be explored in the future, for different machine learning algorithms besides linear classification ones, in order to better maintain the quality assurance of machine learning systems.

References

[1] Balakrishnama, S., Ganapathiraju, A., 1998. Linear discriminant analysis-a brief tutorial, in: Institute for Signal and information Processing, pp. 1–8.

[2] Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S., 2014. The oracle problem in software testing: A survey. IEEE transactions on software engineering 41, 507–525.

[3] Bishop, C.M., 2006. Pattern recognition and machine learning. springer.

[4] Bousquet, O., Elisseeff, A., 2002. Stability and generalization. Journal of machine learning research 2, 499–526.

[5] Braiek, H.B., Khomh, F., 2020. On testing machine learning programs. Journal of Systems and Software 164, 110542.

[6] Bühlmann, P., Van De Geer, S., 2011. Statistics for high-dimensional data: methods, theory and applications. Springer Science & Business Media.

[7] Cao, Y., Zhou, Z.Q., Chen, T.Y., 2013. On the correlation between the effectiveness of metamorphic relations and dissimilarities of test case executions, in: 2013 13th International Conference on Quality Software, IEEE. pp. 153–162.

[8] Chakraborty, A., Alam, M., Dey, V., Chattopadhyay, A., Mukhopadhyay, D., 2018. Adversarial attacks and defences: A survey. arXiv preprint arXiv:1810.00069 .

[9] Chen, T.Y., Ho, J.W., Liu, H., Xie, X., 2009. An innovative approach for testing bioinformatics programs using metamorphic testing. BMC bioinformatics 10, 24.

[10] Chen, T.Y., Kuo, F.C., Liu, H., Poon, P.L., Towey, D., Tse, T., Zhou, Z.Q., 2018. Metamorphic testing: A review of challenges and opportunities. ACM Computing Surveys (CSUR) 51, 1–27.

[11] Chen, T.Y., Tse, T., Zhou, Z.Q., 2010. Semi-proving: An integrated method for program proving, testing, and debugging. IEEE Transactions on Software Engineering 37, 109–125.

[12] Chen, W.S., Du, Y.K., 2009. Using neural networks and data mining techniques for the financial distress prediction model. Expert systems with applications 36, 4075–4086.

[13] Cheney, W., Kincaid, D., 2009. Linear algebra: Theory and applications. The Australian Mathematical Society 110.

[14] Cheng, D., Cao, C., Xu, C., Ma, X., 2018. Manifesting bugs in machine learning code: An explorative study with mutation testing, in: 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), IEEE. pp. 313–324.

[15] Chiu, C.C., Lin, P.Y., Lin, C.J., 2020. Two-variable dual coordinate descent methods for linear svm with/without the bias term, in: Proceedings of the 2020 SIAM International Conference on Data Mining, SIAM. pp. 163–171.

[16] Chou, H.Y., Lin, P.Y., Lin, C.J., 2020. Dual coordinate-descent methods for linear one-class svm and svdd, in: Proceedings of the 2020 SIAM International Conference on Data Mining, SIAM. pp. 181–189.

[17] Cohen, A., 2014. Surrogate Loss Minimization. Ph.D. thesis.

[18] Cortes, C., 1995. Support vector machine .

[19] Defazio, A., Bach, F., Lacoste-Julien, S., 2014. Saga: A fast incremental gradient method with support for non-strongly convex composite objectives, in: Advances in neural information processing systems, pp. 1646–1654.

[20] Denisov, A., Pankevich, S., 2018. Mull it over: Mutation testing based on llvm, in: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 25–31. doi:10.1109/ICSTW.2018.00024.

[21] Devroye, L., Wagner, T., 1979. Distribution-free performance bounds for potential function rules. IEEE Transactions on Information Theory 25, 601–604.

[22] Duran, J.W., Ntafos, S.C., 1984. An evaluation of random testing. IEEE Transactions on Software Engineering SE-10, 438–444. doi:10.1109/TSE.1984.5010257.

[23] Dwarakanath, A., Ahuja, M., Sikand, S., Rao, R.M., Bose, R.J.C., Dubash, N., Podder, S., 2018. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing, in: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 118–128.

[24] Gopinath, D., Pasareanu, C.S., Wang, K., Zhang, M., Khurshid, S., 2019. Symbolic execution for attribution and attack synthesis in neural networks, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), IEEE. pp. 282–283.

[25] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H., 2009. The weka data mining software: an update. ACM SIGKDD explorations newsletter 11, 10–18.

[26] Hilbe, J.M., 2009. Logistic regression models. CRC press.

[27] Hoeffding, W., 1994. Probability inequalities for sums of bounded random variables, in: The Collected Works of Wassily Hoeffding. Springer, pp. 409–426.

[28] Hovmöller, A., . mutmut · pypi. https://pypi.org/project/mutmut/.

[29] Hsieh, C.J., Chang, K.W., Lin, C.J., Keerthi, S.S., Sundararajan, S., 2008. A dual coordinate descent method for large-scale linear svm, in: Proceedings of the 25th international conference on Machine learning, pp. 408–415.

[30] Hynes, N., Sculley, D., Terry, M., 2017. The data linter: Lightweight, automated sanity checking for ml data sets, in: NIPS MLSys Workshop.

[31] Ito, N., Takeda, A., Toh, K.C., 2017a. A unified formulation and fast accelerated proximal gradient method for classification. The Journal of Machine Learning Research 18, 510–558.

[32] Ito, N., Takeda, A., Toh, K.C., 2017b. A unified formulation and fast accelerated proximal gradient method for classification. Journal of Machine Learning Research 18, 1–49. URL: http://jmlr.org/papers/v18/16-274.html.

[33] Jia, Y., Harman, M., 2010. An analysis and survey of the development of mutation testing. IEEE transactions on software engineering 37, 649–678.

[34] Kearns, M., Ron, D., 1999. Algorithmic stability and sanity-check bounds for leave-one-out cross-validation. Neural computation 11, 1427–1453.

[35] Kim, J., Feldt, R., Yoo, S., 2019. Guiding deep learning system testing using surprise adequacy, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE. pp. 1039–1049.

[36] Kononenko, I., 2001. Machine learning for medical diagnosis: history, state of the art and perspective. Artificial Intelligence in medicine 23, 89–109.

[37] Le, V., Afshari, M., Su, Z., 2014. Compiler validation via equivalence modulo inputs. ACM SIGPLAN Notices 49, 216–226.

[38] Le, V., Sun, C., Su, Z., 2015. Finding deep compiler bugs via guided stochastic program mutation. ACM SIGPLAN Notices 50, 386–399.

[39] Ma, L., Juefei-Xu, F., Xue, M., Li, B., Li, L., Liu, Y., Zhao, J., 2019. Deepct: Tomographic combinatorial testing for deep learning systems, in: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE. pp. 614–618.

[40] Ma, L., Juefei-Xu, F., Zhang, F., Sun, J., Xue, M., Li, B., Chen, C., Su, T., Li, L., Liu, Y., et al., 2018a. Deepgauge: Multi-granularity testing criteria for deep learning systems, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 120–131.

[41] Ma, S., Liu, Y., Lee, W.C., Zhang, X., Grama, A., 2018b. Mode: automated neural network model debugging via state differential analysis and input selection, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 175–186.

[42] Madeyski, L., Orzeszyna, W., Torkar, R., Jozala, M., 2013. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. IEEE Transactions on Software Engineering 40, 23–42.

[43] Menze, M., Geiger, A., 2015. Object scene flow for autonomous vehicles, in: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 3061–3070.

[44] Metzen, J.H., Genewein, T., Fischer, V., Bischoff, B., 2017. On detecting adversarial perturbations. arXiv preprint arXiv:1702.04267 .

[45] Mitchell, T.M., et al., 1997. Machine learning. Burr Ridge, IL: McGraw Hill 45, 870–877.

[46] Mukherjee, S., Niyogi, P., Poggio, T., Rifkin, R., 2006. Learning theory: stability is sufficient for generalization and necessary and sufficient for consistency of empirical risk minimization. Advances in Computational Mathematics 25, 161–193.

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

[48] Offutt, J., Li, N., . mujava home page. https://cs.gmu.edu/~offutt/mujava/ Accessed June 2013.

[49] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E., 2011. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 12, 2825–2830.

[50] Pei, K., Cao, Y., Yang, J., Jana, S., 2017. Deepxplore: Automated whitebox testing of deep learning systems, in: proceedings of the 26th Symposium on Operating Systems Principles, pp. 1–18.

[51] Pham, H.V., Lutellier, T., Qi, W., Tan, L., 2019. Cradle: cross-backend validation to detect and localize bugs in deep learning libraries, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE. pp. 1027–1038.

[52] Qi, Z., Wang, H., Li, J., Gao, H., 2018. Impacts of dirty data: and experimental evaluation. arXiv preprint arXiv:1803.06071 .

[53] Ramanathan, A., Steed, C.A., Pullum, L.L., 2012. Verification of compartmental epidemiological models using metamorphic testing, model checking and visual analytics, in: 2012 ASE/IEEE International Conference on BioMedical Computing (BioMedCom), IEEE. pp. 68–73.

[54] Ruder, S., 2016. An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747 .

[55] Schmidt, M., Le Roux, N., Bach, F., 2017. Minimizing finite sums with the stochastic average gradient. Mathematical Programming 162, 83–112.

[56] Schulam, P., Saria, S., 2019. Can you trust this prediction? auditing pointwise reliability after learning. arXiv preprint arXiv:1901.00403 .

[57] Segura, S., Fraser, G., Sanchez, A.B., Ruiz-Cortés, A., 2016. A survey on metamorphic testing. IEEE Transactions on software engineering 42, 805–824.

[58] Shalev-Shwartz, S., Shamir, O., Srebro, N., Sridharan, K., 2010. Learnability, stability and uniform convergence. The Journal of Machine Learning Research 11, 2635–2670.

[59] Singh, G., et al., 2012. An automated metamorphic testing technique for designing effective metamorphic relations, in: International Conference on Contemporary Computing, Springer. pp. 152–163.

[60] Srisakaokul, S., Wu, Z., Astorga, A., Alebiosu, O., Xie, T., 2018. Multiple-implementation testing of supervised learning software., in: AAAI Workshops, pp. 384–391.

[61] Tao, Q., Wu, W., Zhao, C., Shen, W., 2010. An automatic testing approach for compiler based on metamorphic testing technique, in: 2010 Asia Pacific Software Engineering Conference, IEEE. pp. 270–279.

[62] Tian, Y., Pei, K., Jana, S., Ray, B., 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars, in: Proceedings of the 40th international conference on software engineering, pp. 303–314.

[63] Valiant, L.G., 1984. A theory of the learnable. Communications of the ACM 27, 1134–1142.

[64] Vu Do, H., Robach, C., Delaunay, M., 2006. Mutation analysis for reactive system environment properties, in: Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006), pp. 2–2. doi:10.1109/MUTATION.2006.9.

[65] Wainwright, K., et al., 2005. Fundamental methods of mathematical economics. McGraw-Hill/Irwin.

[66] Wang, J., Dong, G., Sun, J., Wang, X., Zhang, P., 2019. Adversarial sample detection for deep neural network through model mutation testing, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE. pp. 1245–1256.

[67] Wang, Z., Yan, M., Chen, J., Liu, S., Zhang, D., 2020. Deep learning library testing via effective model generation, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 788–799.

[68] Xie, X., Ho, J., Murphy, C., Kaiser, G., Xu, B., Chen, T.Y., 2009. Application of metamorphic testing to supervised classifiers, in: 2009 Ninth International Conference on Quality Software, IEEE. pp. 135–144.

[69] Xie, X., Ho, J.W., Murphy, C., Kaiser, G., Xu, B., Chen, T.Y., 2011. Testing and validating machine learning classifiers by metamorphic testing. Journal of Systems and Software 84, 544–558.

[70] Xie, X., Ma, L., Juefei-Xu, F., Xue, M., Chen, H., Liu, Y., Zhao, J., Li, B., Yin, J., See, S., 2019. Deephunter: A coverage-guided fuzz testing framework for deep neural networks, in: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 146–157.

[71] Xie, X., Wong, W.E., Chen, T.Y., Xu, B., 2013. Metamorphic slice: An application in spectrum-based fault localization. Information and Software Technology 55, 866–879.

[72] Xie, X., Zhang, Z., Chen, T.Y., Liu, Y., Poon, P.L., Xu, B., 2020. Mettle: a metamorphic testing approach to assessing and validating unsupervised machine learning systems. IEEE Transactions on Reliability .

[73] Ye, G.B., Chen, Y., Xie, X., 2011. Efficient variable selection in support vector machines via the alternating direction method of multipliers, in: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, pp. 832–840.

[74] Yu, H.F., Huang, F.L., Lin, C.J., 2011. Dual coordinate descent methods for logistic regression and maximum entropy models. Machine Learning 85, 41–75.

[75] Zhang, J.M., Harman, M., Ma, L., Liu, Y., 2020. Machine learning testing: Survey, landscapes and horizons. IEEE Transactions on Software Engineering .

[76] Zhou, Z.Q., Xiang, S., Chen, T.Y., 2015. Metamorphic testing for software quality assessment: A study of search engines. IEEE Transactions on Software Engineering 42, 264–284.

[77] Zhu, C., Byrd, R.H., Lu, P., Nocedal, J., 1997. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. ACM Transactions on Mathematical Software (TOMS) 23, 550–560.

A. Proof of Proposition 1 and Proposition 2

We start by defining some notation. Given dataset $D = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^m$, where $\boldsymbol{x}_i$ is an n-dimensional example, and $y_i \in \{-1, +1\}$. A linear classification algorithm attempts to solve the following optimization problem:

$$\min_{\boldsymbol{w}} J(\boldsymbol{w}; D) = L(\boldsymbol{w}; D) + R(\boldsymbol{w}),$$

where $\boldsymbol{w} \in \mathbb{R}^n$ is the vector of model parameter, function $L(\cdot,\cdot) : \mathbb{R}^n \times \mathbb{R}^{m\times n} \to \mathbb{R}$ is any given loss function (e.g., Cross Entropy, Hinge loss, modified Huber loss, and so on), and $R(\cdot) : \mathbb{R}^n \to \mathbb{R}$ is any regularization function (e.g., ℓ_1 norm, ℓ_2 norm, and even Elastic loss). Since the loss function L can be decomposed into the summation of losses for each example, the objective function can be reformed as

$$J(\boldsymbol{w}; D) = \frac{1}{n}\sum_{i=1}^{m} \ell(\boldsymbol{w}; (\boldsymbol{x}_i, y_i)) + R(\boldsymbol{w}).$$

Let $t_i = y_i(\boldsymbol{w}^\top \boldsymbol{x}_i), i = 1, \dots, m$, we can obtain that

$$\ell(\boldsymbol{w}; (\boldsymbol{x}_i, y_i)) = \ell(t_i), \quad i = 1, \dots, m. \tag{13}$$

The above equation holds due to that function $\ell(\cdot,\cdot)$ is the surrogate loss of $0/1$-loss [17]. Besides, it also needs to be mentioned is that the surrogate loss functions are all *convex* functions (with respect to variable t).

Similar to [56], we also assume that the output $\boldsymbol{w}^*$ of algorithm subjects to

$$\nabla J(\boldsymbol{w}^*) = \frac{1}{n}\sum_{i=1}^{m} \frac{\partial \ell}{\partial t}(t_i) \cdot y_i \boldsymbol{x}_i + \nabla(\boldsymbol{w}^*) = \boldsymbol{c}, \tag{14}$$

where $t_i = \boldsymbol{w}^{*\top} \boldsymbol{x}_i$, and $\boldsymbol{c} \in \mathbb{R}^n$ is any constant vector. For simplicity, we denote the $\nabla J(\boldsymbol{w})$ by $\phi(\boldsymbol{w})$. We can compute the Hessian matrix of $J(\boldsymbol{w})$ (i.e., the Jacobian of $\phi(\boldsymbol{w})$) by

$$\mathcal{J}\phi(\boldsymbol{w}) = \nabla^2_{\boldsymbol{w}} J(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{m} \frac{\partial^2 \ell}{\partial t^2}(t_i) \cdot \boldsymbol{x}_i \boldsymbol{x}_i^\top + \nabla^2(\boldsymbol{w}).$$

Next, we try to prove Proposition 1, and the proof of Proposition 2 is similar to the proof of Proposition 1. Without loss of generality, we consider that the n-th example $\boldsymbol{x}_n$ is replaced.

Proof. Let the perturbed example be $\widehat{\boldsymbol{x}}_n$, where $\widehat{\boldsymbol{x}}_m = \boldsymbol{x}_m + \boldsymbol{\xi}$. In this case, the modified dataset $\widehat{D}$ is $\widehat{D} = \{\boldsymbol{x}_i, y_i\}_{i=1}^{m-1} \cup \{\widehat{\boldsymbol{x}}_m, y_m\}$, and the corresponding loss function is

$$J(\boldsymbol{w}; \widehat{D}) = \frac{1}{n}\left(\sum_{i=1}^{m-1} \ell(\boldsymbol{w}; (\boldsymbol{x}_i, y_i)) + \ell(\boldsymbol{w}; (\boldsymbol{x}_m, y_m))\right) + R(\boldsymbol{w}).$$

Now that the loss function has changed, the output of the algorithm has also changed accordingly. We denote the changed output by $\widehat{\boldsymbol{w}}^*$, and it should also meet the Equation 14:

$$\nabla_{\boldsymbol{w}} J(\widehat{\boldsymbol{w}}^*) = \frac{1}{n}\sum_{i=1}^{m} \frac{\partial \ell}{\partial t}(\widehat{t}_i) \cdot y_i \boldsymbol{x}_i + \nabla(\widehat{\boldsymbol{w}}^*) = \boldsymbol{c},$$

where $\widehat{t}_i = \widehat{\boldsymbol{w}}^{*\top} \boldsymbol{x}_i$. Due to the (semi)-smoothness of loss and regularizer, we can use implicit function theorem [65] to conclude that there exists a local function such that $\psi(\boldsymbol{x}_n) = \boldsymbol{w}^*$. In other words, there is a map from example $\boldsymbol{x}_n$ to trained model's parameter $\boldsymbol{w}^*$. Hence, we can roughly compute the $\widehat{\boldsymbol{w}}^*$ through

$$\begin{aligned}
\widehat{\boldsymbol{w}}^* - \boldsymbol{w}^* &= \psi(\widehat{\boldsymbol{x}}_n) - \psi(\boldsymbol{x}_n) \\
&\approx \frac{\partial \psi}{\partial \boldsymbol{x}_n}(\boldsymbol{x}_n)(\widehat{\boldsymbol{x}}_n - \boldsymbol{x}_n) \\
&= -\left(\mathcal{J}_{\boldsymbol{w}}\phi(\boldsymbol{w}^*, \boldsymbol{x}_n)\right)^{-1} \mathcal{J}_{\boldsymbol{x}_n}\phi(\widehat{\boldsymbol{w}}, \boldsymbol{x}_n, y_n)(\widehat{\boldsymbol{x}}_n - \boldsymbol{x}_n) \\
&= -\frac{1}{n}\left(\mathcal{J}\phi(\boldsymbol{w})\right)^{-1}\left(y_n \frac{\partial \ell}{\partial t}(t_n) + \frac{\partial^2 \ell}{\partial t^2}(t_n)\boldsymbol{x}_n \widehat{\boldsymbol{w}}^\top\right)\boldsymbol{\xi}.
\end{aligned}$$

To avoid the computation of Jacobian, we set the perturbation $\boldsymbol{\xi}$ as a vector orthogonal to $\boldsymbol{w}$, i.e., $\boldsymbol{\xi}^\top \boldsymbol{w}$. In this sense, we have

$$\begin{aligned}
\widehat{f}(\boldsymbol{x}) - f(\boldsymbol{x}) &= (\widehat{\boldsymbol{w}}^* - \boldsymbol{w}^*)^\top \boldsymbol{x} \\
&= -\frac{1}{n} y_n \frac{\partial \ell}{\partial t}(t_n) \boldsymbol{x}^\top \left(\mathcal{J}_{\boldsymbol{w}}\phi(\boldsymbol{w}^*, \boldsymbol{x}_n)\right)^{-1} \boldsymbol{\xi} \\
&= -\frac{1}{n} y_n \frac{\partial \ell}{\partial t}(t_n) \cdot C,
\end{aligned}$$

where we denote $\boldsymbol{x}^\top \left(\nabla_{\boldsymbol{w}}\phi(\boldsymbol{w}^*, \boldsymbol{x}_n)\right)^{-1} \boldsymbol{\xi}$ by C since it is a constant scalar. Now, we can obtain that

$$y_n(\widehat{f}(\boldsymbol{x}) - f(\boldsymbol{x})) = -\frac{C}{n}\frac{\partial \ell}{\partial t}(t_n). \tag{15}$$

Since the $\ell(\cdot)$ is convex w.r.t. variable t, the first-derivation of $\ell(\cdot)$ should be non-decreasing, which indicates the monotonicity of function $y_n(\widehat{f}(\boldsymbol{x}) - f(\boldsymbol{x}))$. □