2018 IEEE International Conference on Software Quality, Reliability and Security

Manifesting Bugs in Machine Learning Code: An Explorative Study with Mutation Testing

Dawei Cheng, Chun Cao, Chang Xu and Xiaoxing Ma
State Key Laboratory of Novel Software Technology
Institute of Computer Software, Nanjing University, China
Email: daweic@yeah.net, {caochun, changxu, xxm}@nju.edu.cn

***Abstract*—Nowadays statistical machine learning is widely adopted in various domains such as data mining, image recognition and automated driving. However, software quality assurance for machine learning is still in its infancy. While recent efforts have been put into improving the quality of training data and trained models, this paper focuses on code-level bugs in the implementations of machine learning algorithms. In this explorative study we simulated program bugs by mutating Weka implementations of several classification algorithms. We observed that 8%-40% of the logically non-equivalent executable mutants were statistically indistinguishable from their golden versions. Moreover, other 15%-36% of the mutants were stubborn, as they performed not significantly worse than a reference classifier on at least one natural data set. We also experimented with several approaches to killing those stubborn mutants. Preliminary results indicate that bugs in machine learning code may have negative impacts on statistical properties such as robustness and learning curves, but they could be very difficult to detect, due to the lack of effective oracles.**

***Index Terms*—machine learning programs, mutation testing, explorative study.**

I. Introduction

Over the past years, the adoption of (statistical) machine learning (ML) in practice has been accelerating. In addition to those everyday applications such as spam email filtering and online shopping recommendation, ML is also used in mission critical scenarios such as biometric authentication and automated driving. As a consequence the need for quality assurance of ML programs is increasing. However, practitioners usually do little more than testing trained models with some samples left from the training data and validating them with few samples newly obtained in field.

Recently, the problem that ML models are vulnerable to adversarial examples has attracted a lot of researches [1], [2]. There is also interesting work on reducing the noises in the training data [3]. These efforts focus on the quality of ML applications, but not directly on the quality of ML code. In other words, the aim of them is to get a better trained model fit for the current application, but not a less buggy learning program.

In this paper we focus on potential bugs in the *implementation* of ML algorithms. This problem is seldom considered by ML researchers and practitioners, and they tend not to distinguish ML algorithms and the code implementing these algorithms. However, to err is human, and there is no reason that ML code is bug-free. It is interesting to consider the impact of bugs in ML code. For example, in a prediction task, your ML program got the accuracy of 95%, which seemed like a good result. Nevertheless, assume that there was a bug in the implementation, and if the bug was fixed, the result of the program might achieve 98%. On the other hand, we cannot exclude the possibility that an unintended mistake may improve the performance in some circumstance.

Detecting bugs in ML program is challenging. A deep reason is "the lack of *a priori* distinctions between learning algorithms", a.k.a. the no-free-lunch theorem [4], which implies that in theory we cannot specify a test oracle for a learning program by predicting on its performance, without assumptions on the problem to be solved. Another reason is that the statistical nature of ML programs can make them robust to bugs [5]. In practice a programmer usually tests his ML program against a reference learner. If the result is unexpectedly worse than the reference, the programmer may manually review his program. However it is difficult to tell whether the bad performance is caused by the uncompetitiveness of the algorithm, the insufficiency of the training data, or a bug in the implementation.

We conducted an explorative study to understand the impact of bugs in ML programs and the effectiveness of several testing approaches. We used the mutation testing technique [6] to simulate bugs in ML programs. Weka [7] implementation of Naive Bayes, C4.5, k-NN and SVM algorithms were used as golden versions, and totally 31450 compilable mutants were generated with μJava[1]. To focus on mutants that do have an impact on performance, we excluded mutants that caused fatal errors in execution, and those logically equivalent to golden versions. Surprisingly, in the remaining 4382 logically non-equivalent executable mutants, 8%-40% of them were statistically indistinguishable from their golden versions. Moreover, other 15%-36% of the mutants were stubborn, as they performed not significantly worse than a reference classifier on at least one natural data set. This result indicates that bugs in the implementation of ML algorithms could be a real but unaware problem.

Preliminary experiments were carried out to examine the effectiveness of testing approaches using metamorphic relations and other partial oracles such as learning curves [8], [9] and robustness [10], [11]. Unfortunately they were not effective

[1]https://cs.gmu.edu/ offutt/mujava/

978-1-5386-7757-5/18/$31.00 ©2018 IEEE
DOI 10.1109/QRS.2018.00044

enough to kill stubborn mutants. These stubborn mutants did cause observable negative impacts on some statistical properties such as robustness and learning curves when compared with their golden versions. However the impacts were not significant enough to distinguish them from the reference classifier. Note that golden versions are unavailable in practice and only a reference classifier can be used instead.

The rest of the paper is organized as follows. Section II introduces machine learning programs and mutation testing. We classify mutants based on their performance in Section III and examine the effectiveness of testing approaches in Section IV. After briefly discussing the threads to validity of our study in Section V and related work in Section VI, we conclude the paper in Section VII.

II. Background

A. Machine Learning Foundation

Unlike the traditional programs, ML explores the study and construction of algorithms, then learns from data and makes predictions [12]. Each element in the data is called a sample, which is represented as a vector $s = < a_0, a_1, ..., a_m, l >$. Each sample has its attributes a_i $(i = 1, 2, ..., m)$ and label l. Usually data set can be divided into training set, validation set and test set [13]. The learner uses training data to learn, which is used to fit the parameters. The validation set plays a role in model selection. The parameters could be tuned in this phase, e.g. the learning rate of a deep neural network. The test set is only used for evaluating the performance of a fully specified classifier.

As shown in Figure 1, it is a flow chart about the procedure of ML. The learner generates a classifier with training data. And the classifier gives the predictions of new data.

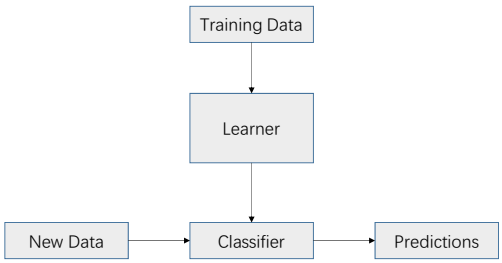


Fig. 1: Procedure of machine learning

In this paper, we selected four classification algorithms for investigation, which were Naive Bayes [14], C4.5 [15], k-NN [16] and SVM [17].

1) Naive Bayes: Naive Bayes is a classification algorithm based on Bayes theorem and conditional independence assumption between the features.

2) C4.5: C4.5 is an algorithm which generates a decision tree and is developed by Ross Quinlan [18]. It uses information gain ratio to select on which attribute to split and can deal with numeric or enumerated data.

3) k-NN: k-NN is an algorithm which outputs a class membership. A sample is classified by a majority vote of its neighbours, i.e., being assigned to the most common class among its k nearest neighbours.

4) SVM: SVM is an algorithm which constructs a hyperplane in a high or infinite dimensional space. It can be represented as a convex optimization problem and the objective function is to find the global maximum margin.

B. Mutation Testing

Mutation testing is a fault-based technique in software testing. It is not used to find the bugs in software but evaluate the quality of existing test approaches. It was proposed by Richard Lipton [19] originally, and was first published by DeMillo et al. [20].

Mutation testing is done by selecting a set of mutation operators such as operator replacement, operator increase and statement deletion. One mutation operator is applied to the available part of the source code, which is called golden version, at a time and the result is called a mutant. Once a test case fails on a mutant, the mutant is said to be killed. Sometimes a mutant could be semantically consistent with golden version, then we call it an equivalent mutant.

III. Classifying Enormous Mutants

In this section, we tried to classify the bugs in ML programs through statistical hypothesis testing.

Assuming that the algorithms and data sets are both correct, i.e., we only consider the bugs in the implementation of ML algorithms. We want to figure out what their behaviours are like. If a bug always leads to a crash or an extreme decrease in performance, the programmers could soon perceive this by running with some common data sets. However, if there exist bugs which are well-hidden, things become different. Below is the research question.

RQ1: How do these bugs in machine learning programs behave differently against the golden version and the reference?

In this question, we investigated four common ML algorithms using 5x2cv t-test [21] to compare the performance in a statistical way.

A. Methodology

In order to comprehend the properties of bugs in ML programs, we used mutants to simulate the real mistakes the programmers made in their code. Andrews et al. [6] found that the mutants and real mistakes have no significant difference statistically. We selected μjava to generate mutants, which was a mutation testing tool for Java language programs. And we used Weka as our subject, which was a well-known ML toolkit implemented with Java.

The experiment consisted of several steps. First we generated mutants for each algorithm, using both the class level

and the method level mutation operators. The source code in Weka is called a golden version, while the generated code by μJava is called a mutant version. Each algorithm has a series of mutant versions and each mutant version only has one mutant. Second, we compared golden version with its each mutant version through 5x2cv t-test [21]. We used weighted f_1-measure to evaluate the performance of classifier. For each pair of programs and each time of cross validation, we calculated the f_1-measure differences on the 1st fold Δ_i^1 and the 2nd fold Δ_i^2. In order to alleviate the independence of f_1-measure, only the average of the difference on the first time of cross validation was calculated.

$$\mu = 0.5(\Delta_1^1 + \Delta_1^2) \tag{1}$$

The variance is

$$\sigma_i^2 = (\Delta_i^1 - \frac{\Delta_i^1 + \Delta_i^2}{2})^2 + (\Delta_i^2 - \frac{\Delta_i^1 + \Delta_i^2}{2})^2 \tag{2}$$

and the variable

$$\tau_t = \frac{\mu}{\sqrt{0.2\sum_{i=1}^{5}\sigma^2}} \tag{3}$$

obeys the t distribution with 5 DOF (degree of freedom), which critical value of two-sided test $t_{\alpha/2,5}$ is 2.5706 when $\alpha = 0.05$.

We assumed that the performance of golden version and mutant version had no significant difference (H_0). If the absolute value of test statistic is less than the critical value, then we can not reject H_0. It means that the two classifiers have no significant difference in performance. Otherwise one is quite different from another in performance.

B. Against Golden Version

In order to investigate the performance difference between mutant version and golden version, we carried out four sub-experiments with different combinations of data sets.

1) Different Distribution: (sub-experiment 1) To begin with, we chose three data sets in different distribution from UCI ML repository[2]. The basic idea is that different distribution may lead to different traces in the execution. Table I shows the details.

TABLE I: DATA SETS IN DIFFERENT DISTRIBUTION

Data set	Size	Attr num	Class num
German	1000	24	2
Kr-vs-kp	3196	36	2
Waveform	5000	40	2

2) Different Size: (sub-experiment 2) Apart from the distribution, we also considered the influence on the size of data set. We hoped the distinction in performance could be revealed through data sets in difference size even if they obeyed same distribution. The original data set was *covtype* from UCI ML repository. We selected two most frequent classes and did some reduction on attributes. The number of samples in each class was equal. Table II shows the details.

[2]http://archive.ics.uci.edu/ml/datasets.html

TABLE II: DATA SETS IN DIFFERENT SIZE

Data set	Size	Attr num	Class num
covtype$_s$	1000	11	2
covtype$_m$	10000	11	2
covtype$_l$	100000	11	2

3) Imbalanced Data: (sub-experiment 3) Imbalanced data was another aspect we wanted to explore. Here we still chose *covtype* data set, changing the amount of samples labelled 0 while fixing the amount of samples labelled 1. Table III shows the details.

TABLE III: IMBALANCED DATA SETS

Data set	Size	Attr num	Class 0 num	Class 1 num
covtype$_{s,im}$	7500	11	2500	5000
covtype$_{m,im}$	10000	11	5000	5000
covtype$_{l,im}$	15000	11	10000	5000

4) Special Data: (sub-experiment 4) In real world, noise always exists in data. Thus we considered some special data sets which were obviously separable. A mutant might be insensitive to a real world data set, but sensitive to a special data set. The insight is that the low entropy of special data set can be easily enlarged due to any tiny perturbation, e.g. mutant. Table IV shows the details. For instance, data set sp_1 is generated for SVM. It has two numeric attributes (a_1, a_2) and two classes (0,1). All the samples are generated randomly or based on some distributions.

TABLE IV: SPECIAL DATA SETS

Data set	Size	Algorithm	Class 0	Class 1
sp$_1$	22000	SVM	$a_2 > a_1 + 200$	$a_2 < a_1 - 140$
sp$_2$	20000	Naive Bayes, C4.5	$a_1 \in [0, 2\times10^4)$, $a_2 \sim N(-3,0)$, $a_3 \in [0,4]$ and is integer[1]	$a_1 \in [2\times10^4, 4\times10^4)$, $a_2 \sim N(3,0)$, $a_3 \in [5,9]$
sp$_3$	10000	k-NN	$a_1 \in (-5\times10^3, 5\times10^3)$, $a_2 \in (5\times10^3, 1.5\times10^4)$	$a_1 \in (5\times10^3, 1.5\times10^4)$, $a_2 \in (-5\times10^3, 5\times10^3)$

[1] the default type of attribute is double

In each sub-experiment, we could see that some mutants led to a crash or time out. Usually these mutants threw runtime exceptions then crashed. We set the maximum elapsed time for each mutant to avoid a dead loop. Besides, we observed that a lot of mutants were not executed. It is because Weka is an integration tool, and some code segments are not relevant to the core part of the algorithms. In addition, an algorithm might have a variety of models like attribute type in C4.5 or have many parameters. If only the numeric part is focused on, the correctness of enumerated part will be missed validating.

Definition 1: (Statistically equivalent mutant). *Given a set of data sets $\mathbb{D} = \{D_1, D_2, .., D_n\}$, if a mutant version has no significant difference with its golden version on any data sets $D_i (i = 1, 2, ..., n)$, we call it a statistically equivalent mutant. The equivalent and not executed mutants are not included.*

After eliminating the two kinds of mutants above, we observed that three circumstances appeared in each sub-experiment: 1) passing t-test on all data sets, 2) passing t-test on no data sets, 3) passing t-test on a part of data sets.

The first circumstance meant that these mutants had no significant difference with golden version. If they were equivalent mutants, then the mutant versions were exactly same as golden version in performance and we ignored them. Some of them might perform worse on a new data set even if they had good results here. We didn't care about this situation because we couldn't possess all the data sets in the real testing environment. The rest of them were statistically equivalent mutants which were defined above. We thought they were not bugs at a view of statistics.

The second circumstance meant that these mutants had great impact on learner and led to bad performance. Usually testing against the reference could find their existence.

The third circumstance meant that these mutants had different responses to different data sets. Data sets in different distribution might lead to different traces in the program. For instance, the calculations of information gain were different when taking the type of attributes into account in C4.5. The numeric attributes and the enumerated attributes should be treated differently. The attributes in $Kr\text{-}vs\text{-}kp$ were all enumerated but the ones in $Waveform$ were all numeric, so the traces of two training process had divergence. The mutants in the relevant part of numeric attributes had no affect on the part of enumerated attributes and vice versa.

Here are some typical statistically equivalent mutants in sub-experiment 1.

Variable precision: The algorithm in Weka has some procedure to determine variable precision. For example, if the precision of a variable is 0.01, then the values in [0.085, 0.095) are regard as 0.09. Some mutants just modify the precision, might changing it from 0.01 to 0.02, and have slight influence on the decision logic.

Sample reduction: Some mutants could modify the utility of data set. In Listing 1, if we change i into $++i$ at line 2, the iterator variable increase one when executing this statement. Finally half of the data are never used but it still has no significant difference in performance because the amount of data set is enough to learn.

```
1 //for(int i=0; i<m_Instances.numInstances();
      i++){
2 for(int i=0; ++i<m_Instances.numInstances();
      i++){
3   inst = m_Instances.instance(i);
4   ...
5 }
```

Listing 1: Example of sample reduction

Model selection: Some algorithms have more than one model in some procedure. In C4.5, the maximum information gain ratio of numeric attributes can be calculated under Single Gaussian

$$p(X = x|C = c) = g(x, \mu_c, \sigma_c), \text{ where} \tag{4}$$

$$g(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{5}$$

or Kernel Density Gaussian

$$p(X = x|C = c) = \frac{1}{n} \sum_i g(x, \mu_i, \sigma_c) \tag{6}$$

The results of two model have no significant difference in the three data sets, but we cannot promise that it still holds for any other data sets.

Approximation: The approximation procedure is common in ML. In C4.5 algorithm, we need a normal distribution approximation to binomial distribution and continuity correction when computing estimated extra error. In a binomial distribution $B \sim (n, p)$, if $np \geq 4$ and $n(1-p) \geq 4$, its probability is close to normal distribution N. The correction factory is 0.5 because if we calculate $P_{x\sim B}(2)$, we cannot simply use $P_{x\sim N}(2)$. $P_{x\sim N}(2.5)$ is proper because the rectangle area of interval [2, 2.5] is need, otherwise we miss it. However, the mutant changes $2 + 0.5$ to $2 - 0.5$ and passes t-test. Approximation itself is not exactly accurate. Perhaps the sum of original value and offset caused by mutant is more close to the real value. In addition, the gradients of probability changes are same for all the samples, and we are more concerned about the relative values.

We selected all the three data sets in Table I and $covtype_s$ in Table II as the set of data sets $\mathbb{D}$ mentioned in the definition and classified the mutants. Table V below shows the details.

TABLE V: MUTANT CLASSIFICATION AGAINST GOLDEN VERSION

Type	C4.5	k-NN	Naive Bayes	SVM
fatal	1110	209	279	2009
equivalent or not executed	12467	2693	4225	4076
statistically equivalent	643	27	182	502
non statistically equivalent	962	315	606	1145
total	15182	3244	5292	7732

We could see that 8% to 40% of the logically non-equivalent mutants were statistically indistinguishable from their golden versions, e.g., the rate for C4.5 was $643/(643+962) \approx 40\%$. The "fatal" meant the mutant versions crashed or caused time out on all data sets. Some of them were time-consuming and some of them even entered a dead loop. "Equivalent or not executed" meant the performance was exactly the same as its golden version on all data sets. A small part of them were equivalent mutants, and most of them (over 90%) were not executed on any data sets. "Non statistically equivalent" mutants and the statistically equivalent mutants formed logically non-equivalent executable mutants.

Beside the classification results, we had some other observations. Two special circumstances occurred in sub-experiment 2. One was passing t-test only on small data sets, and another was passing t-test only on large data sets.

The former indicates that the increasing size of data set might enlarge the influence of mutants in the programs gradually. There are so many loop statements in ML programs. If the loop count has positive correlation with the size of data and the mutants precisely locate here, this situation is more likely

to happen. More generally, if the execution number of some calculation has positive correlation with size, e.g. incremental learning, this situation could arise too. Listing 2 shows the calculation of the standard deviation in normal distribution. The mutant uses denominator to replace the fraction in line 2. The execution number is increasing when the size is increasing. The error is accumulated gradually and is exposed when the size is large to a certain extent.

```
1 m_Mean = m_SumOfValues / m_SumOfWeights;
2 double stdDev = Math.sqrt( m_SumOfWeights );
3 /*double stdDev = Math.sqrt( Math.abs(
      m_SumOfValuesSq - m_Mean * m_SumOfValues ) /
4  m_SumOfWeights ); */
5 if (stdDev > 1e-10) {
6   m_StandardDev = Math.max( m_Precision / (2 *
        3), stdDev );
7 }
```

Listing 2: Example of first special circumstance

```
1 //for (int i = 0; i < numSymbols; i++) {
2 for (int i = 0; i < --numSymbols; i++) {
3   m_Counts[i] = 1;
4 }
5 m_SumOfCounts = (double) numSymbols;
```

Listing 3: Example of second special circumstance

The later indicates that the increase of the size may conceal some mutants. One proper explanation is that these mutants are only executed within fixed times, and the influence are covered by the large size finally, e.g., a mutant changes the variable v from 1 to 2 before calculating $v + s$ where s is data size. If $s = 9$, v will increase by 10%. However, v will only increase by 0.1% if $s = 999$. Listing 3 is a specific example where the mutant changed $numSymbols$ to $--numSymbols$ at line 2. This is Laplace calibration in Naive Bayes which is used to avoid the posterior probability of certain class becoming zero. The mutant adds a decrement operator ahead of variable $numSymbols$. If $numSymbols = 2n$, the last n elements of array will not be initialized to 1. If $numSymbols = 2n + 1$, the last $n + 1$ elements will not be initialized. When the size is large enough, the effect of "1" will vanish (assuming that no posterior probability is zero).

We compared the result of sub-experiment 3 with the former two sub-experiments. It was quite different from sub-experiment 1 but close to sub-experiment 2. Thus, we thought the imbalanced data sets caused slight perturbation in original distribution but influenced the size more significantly.

TABLE VI: RESULT OF SPECIAL DATA SET

Algorithm	Total num	Run successfully	Crash or time out
C4.5	6	3	3
k-NN	0	0	0
Naive Bayes	0	0	0
SVM	15	6	9

In sub-experiment 4, we generated some data sets under special distributions for the specific algorithms. We intended to know whether the special data sets had the ability to enlarge the impact of the mutants when the former three sub-experiments cannot. We concerned about the mutants that cannot pass t-test on special data sets but passed t-test on all other data sets. Except for Naive Bayes and k-NN, we got some expectant mutants and Table VI shows the results. "Run successfully" meant mutant versions ran successfully on all data sets include special data sets, and "crash or time out" meant mutant version crashed or caused time out only on special data sets. Below are the details.

```
1  //if (validModels == 0)
2  if (--validModels == 0)
3    return noSplitModel;
4  averageInfoGain = averageInfoGain / validModels;
5  ...
6    if (...) {
7      if (currentModel[i].infoGain() >=
           averageInfoGain - 1E-3 &&
           Utils.gr(currentModel[i].gainRatio(),
           minResult )) {
8        bestModel = currentModel[i];
9        minResult = currentModel[i].gainRatio();
10     }
11   }
12 if (Utils.eq( minResult, 0 ))
13   return noSplitModel;
```

Listing 4: A mutant version of C4.5 in sub-experiment 4

Three mutant versions ran successfully in C4.5, all decreasing the variable $validModels$ which represented the number of valid split points. An example is in Listing 4. The variable value is 3 in sp_2, and the mutant decreases it to 2. The best model must meet the condition that its information gain is larger than the average minus 0.001 (line 7). However, the mutant increases the average information gain more than one, and there is no model which information gain is larger than the average, leading to no model being selected. In general data sets this could not happen because the number of valid split points is much more than the one in special data sets. The range of information gain is large enough to cover the influence caused by the mutants, i.e., the difference between the information gain of valid split point and the average is far greater than the increase on the average. Besides, one mutant version crashed and two caused time out. The former flipped the sign of the probability in the odd times of recursion and generated negative probabilities, thus crashed. The other two could not stop splitting where they were supposed to. They did extra split work and wasted a lot of time.

Six mutant versions ran successfully and nine crashed or caused time out in SVM. One of nine entered a dead loop and the left eight threw the "ArrayIndexOutOfBoundsException" because of modifying the index of the array.

Another observation was that the code coverage still counted in testing machine learning program. There were so many mutants that their results were exactly equal to the result of golden version (with accuracy of 15 decimal points). The most important reason was many mutants had not yet been executed. These mutants could be equivalent, trivial or stubborn. Another reason was some mutants were semantically equivalent to

golden version. For example, mutants modified string type variables only had impact on the result of output rather than the result of classification.

Apart from code coverage, we captured that some mutant versions ran much slowly than golden version, especially in SVM. Some of them even caused a time out. This is why the rate of fatal mutants in SVM is much higher than other algorithms in Table V. We guessed that these mutants modified the condition of loops, and slowed down the convergence rate. Noticing that some of them even performed closely to golden version. Besides, we saw some bugs could be detected by unit testing but could not through general data sets. For instance, some mutants changed the behaviour of maximum heap. It should return the k nearest samples but returned the $k+1$ nearest ones. These two outputs were closed and we couldn't catch this by the statistical results of whole data sets. When it came to unit testing, the correctness of maximum heap could be tested easily.

In summary, we classified the mutants of ML programs against golden version and saw that 8% to 40% of the logically non-equivalent mutants were statistically indistinguishable from their golden versions. The different forms of the data set combinations have benefit to classifying mutants, e.g., data sets with different distribution could explore distinct traces of programs, and might behave differently. In addition, the code coverage is still important. We should cover the programs as much as possible in testing. We also observed some mutants might do not influence the accuracy but were time-consuming. In the end, the unit testing still works in some situation.

C. Against Reference

We tried to make a further classification of the mutants in this subsection. In the real testing environment, we only have the programs to be tested. There is no such golden version and mutant version. Usually programmers compare the target with reference. We supposed that when the programmers noticed the performance of their programs was no significantly worse than the reference, they could ignore the opportunity for code to go wrong. Here we treated Naive Bayes as reference because it was simple and naive enough to be a benchmark. We applied 5x2cv t-test again on reference and target, i.e. the golden version of Naive Bayes and the mutant version of the other algorithms. We defined the statistically stubborn mutant below.

Definition 2: (Statistically stubborn mutant). *Given a set of data sets $\mathbb{D} = \{D_1, D_2, .., D_n\}$, if a mutant version performs no significantly worse than the reference at least on one data set $D_i (i = 1, 2, ..., n)$, we call it a statistically stubborn mutant. The statistically equivalent mutants are excluded.*

Referring to last subsection, we chose same set of data sets as $\mathbb{D}$ mentioned in the definition. According to the result of 5x2cv t-test between reference and mutant version, we classified the mutants of other three algorithms. Details are shown in Table VII. Besides, the influence caused by statistically stubborn mutants is shown in Table VIII. The values are the absolute differences in f_1-measure between mutant version and golden version. Max value is on the left, min value is in the middle, and average value is on the right.

TABLE VII: MUTANT CLASSIFICATION AGAINST REFERENCE

Type	C4.5	k-NN	SVM
worse than reference	724	236	558
statistically stubborn	238	79	587
statistically equivalent	643	27	502
total	1605	342	1647

TABLE VIII: IMPACT CAUSED BY STATISTICALLY STUBBORN MUTANT

Data set	C4.5	k-NN	SVM
German(%)	1.13/-47.24/-4.68	1.94/-22.10/-2.20	0.91/-53.59/-7.57
Kr-vs-kp(%)	0.08/-63.07/-8.98	0.92/-59.51/-13.17	2.25/-87.37/-14.65
Waveform(%)	0.93/-58.32/-12.88	7.47/-54.53/-12.66	0.03/-85.16/-19.72
covtype$_s$(%)	1.20/-40.31/-10.07	0/-35.07/-13.01	0.42/-41.62/-13.19

We observed that 15% to 36% of the logically non-equivalent executable mutants were statistically stubborn, e.g., the rate for C4.5 was $238/1605 \approx 15\%$. The fatal and logically equivalent mutants were eliminated before. "Worse than reference" meant the performance was worse than reference on all data sets, crashing or logically equivalent performance on a part of data sets was allowed. Then we eliminated the statistically equivalent mutants from the rest mutants and got the statistically stubborn mutants. Surprisingly, some of them could improve the performance (differences are large than 0).

In summary, we classified the non worse than reference mutants into two classes. The first was the statistically stubborn mutants, which could mislead the programmers and should be killed. The second was the statistically equivalent mutants, which should not be killed because they were not belong to bugs statistically.

IV. Killing Stubborn Mutants

In last section, we used mutants to simulate the real bugs and found there existed some statistically stubborn mutants in ML programs which were difficult to be killed and could mislead the programmers. Thus we began our exploration on the methods of killing stubborn mutants.

We proposed three methods to be tried, which were metamorphic testing, learning curve and the robustness to adversarial examples. Below are the research questions.

RQ2: How effective is metamorphic testing in killing statistically stubborn mutants?

RQ3: How effective is using learning curves to kill statistically stubborn mutants?

RQ4: How effective is the robustness and stability to adversarial examples in killing statistically stubborn mutants?

In the above questions, we tried to classify the statistically stubborn mutants and statistically equivalent mutants with each method.

A. Metamorphic Testing

Metamorphic testing is widely used in testing programs without oracles. Its basic idea is that given some test cases and their expected outputs, we can compare the results of generated test cases with the expected ones and then verify the properties of the programs. Some researches [22], [23], [24] have focused on this aspect. Similar to Xie et. al. [22], we used metamorphic testing to kill mutants. However, we believe that we should use statistical significance difference to judge the existence of the violation of metamorphic relations (MRs). It is because ML programs are statistical and tolerant of errors. Even some calculation procedures are carried out in an approximate way and using strict equality is not suitable here. Thus, we used four statistical MRs, i.e. the properties and transformations of MRs were related to the whole data sets, to kill statistically stubborn mutants.

We performed a binary classification task with C4.5, k-NN and SVM on the first time cross validation data of $covtype_m$ data set (first fold as training set and second as test set). We used weighted f_1-measure to evaluate the performance on whole data sets and the parameters were all default values for each algorithms if there were any. After the model had been trained, we evaluated it on both training set and test set, and then did that again after applying the transformation of the MRs on both data sets.

Let $f_{tr,o}$ represent the original f_1-measure, $f_{tr,e}$ represent the expected value and $f_{tr,a}$ represent the actual value after applying the MRs on training set, while $f_{te,o}$, $f_{te,e}$ and $f_{te,a}$ are for test set similarly. Below are the MRs and their properties.

MR0: Duplicating all the samples in training set, then $f_{tr,o}$ = $f_{tr,e}$ and $f_{te,o}$ = $f_{te,e}$.

MR1: Flipping the label of samples in both data sets (changing label 0 to 1 or 1 to 0), then $f_{tr,o}$ = $f_{tr,e}$ and $f_{te,o}$ = $f_{te,e}$.

MR2: Multiplying all the attribute values with a positive integer of all samples in both data sets, then $f_{tr,o}$ = $f_{tr,e}$ and $f_{te,o}$ = $f_{te,e}$.

MR3: Multiplying all the attribute values with a negative integer of all samples in both data sets, then $f_{tr,o}$ = $f_{tr,e}$ and $f_{te,o}$ = $f_{te,e}$.

Taking the case of training set and MR0, it is easy to infer $f_{tr,e}$ from the property of MR0 on the algorithm. After executing the program twice, we got $f_{tr,o}$ and $f_{tr,a}$. We defined that if $\delta = |f_{tr,e} - f_{tr,a}| > \tau$ where δ was the absolute difference of two f_1-measure and τ was the threshold, then MR0 was violated. Error rate was calculated under different τ, e.g., if we got 40% of statistically stubborn mutants and 30% of statistically equivalent mutants violating MR0 at $\tau = 1\%$, the error rates of them were 60% and 30% accordingly with such threshold.

Finally we observed that the result of metamorphic testing was not good enough. Table IX and X show the details. We noticed that sometimes even the δ of golden version was not zero. It indicates that our metrics using threshold to judge whether the MR is violated is suitable for ML programs. There are lots of reasons causing this phenomenon. For example, if a variable x obeys Gaussian distribution, we get $P(-x) = 1-P(x)$. However, when x equals some specific values, the equation becomes invalid due to the precision of computer, leading to violating MR3. We saw that the error rates of statistically stubborn mutants were quite high because of the poor ability of metamorphic testing. Most of the error rates of statistically equivalent mutants are 0%, and seems to be a good result. However, some of them are above 0% except those in k-NN.

```
// Compute minimum number of Instances required
    in each subset.
minSplit = 0.1 * (m_distribution.total()) /
    ((double) trainInstances.numClasses());
if (Utils.smOrEq(minSplit, m_minNoObj))
  minSplit = m_minNoObj;
else if (Utils.gr(minSplit, 25))
  minSplit = 25;
// Enough Instances with known values?
if (Utils.sm((double) firstMiss, 2 * minSplit))
  return;
```

Listing 5: Example of splitting on leaves

Besides, we observed that a MR might not be appropriate for all the algorithms, due to the different learning process. C4.5 algorithm does not satisfy MR0 when it comes to splitting on leaves rather than the calculation of information gain ratio. Listing 5 shows the reason. The size of samples in each subset influences the value of $minSplit$, hence deciding whether to spilt or not. When the size is doubled, the decision logic could be transformed. Further more, even if the MR was fixed, the programs with different parameters could behave differently.

In summary, our experiment result of metamorphic testing is not satisfactory. There are problems with the universality and the ability of MRs. Finding powerful MRs is still an enormous hardship. And the threshold should be set when determining whether the violation is caused by the statistical properties of ML programs.

B. Learning Curve

Learning curve represents the measure of the predictive generalization performance in ML field and is a function of the size of training set [8]. Figure 2 shows the ideal curves. One curve is for training accuracy and another is for validation accuracy. The property is that training accuracy and validation accuracy will converge to the same value with the increase of training set size.

First we chose the Kr-vs-kp as the data set and Naive Bayes as algorithm to see how it performed. The size of data set was starting from 10 and then doubled until reaching its max size (10,20,40,...,3196). For each line symbol graph, we used B-spline to fit them. Some mutants caused low performance, e.g., the training accuracy is lower than 50% even at the size of 3196, and were easily killed. Besides, we captured some buggy mutant versions (Figure 3c) with acceptable accuracy but they had significant difference with golden version (Figure 3a). However, the unclear mutant

TABLE IX: ERROR RATE OF STATISTICALLY STUBBORN MUTANTS

MR(%)	τ=10	τ=5	τ=3	τ=2	τ=1	δ of g.v.[c]
MR0(%)	-/86.14/95.65[a]	-/77.56/82.61	-/65.02/82.61	-/61.72/82.61	-/57.76/82.61	-/0.06/0
	-/81.85/100[b]	-/73.27/100	-/65.35/86.96	-/62.38/82.61	-/50.17/82.61	-/0.02/0
MR1(%)	100/70.30/100	100/61.39/100	100/52.81/86.96	100/52.48/86.96	98.46/48.51/86.96	0/0.02/0
	100/67.00/100	100/54.13/100	100/47.85/86.96	100/44.55/86.96	100/41.91/86.96	0/0/0
MR2(%)	100/100/100	100/100/100	100/100/100	100/100/100	100/100/100	0/0/0
	100/100/100	100/100/100	100/100/100	100/100/100	100/100/100	0/0/0
MR3(%)	100/85.81/100	100/79.21/100	100/73.27/100	98.46/70.63/100	95.38/62.71/100	0/0/0
	100/86.47/100	100/76.90/100	100/73.60/100	96.15/70.30/100	95.38/56.11/100	0.04/0/0

[a] the error rate on training set(left for C4.5, middle for SVM, right for k-NN)
[b] the error rate on test set
[c] the δ of golden version rather than error rate

TABLE X: ERROR RATE OF STATISTICALLY EQUIVALENT MUTANTS

MR(%)	τ=10	τ=5	τ=3	τ=2	τ=1	δ of g.v.
MR0(%)	-/0.49/0	-/1.95/0	-/3.41/0	-/4.87/0	-/8.03/0	-/0.06/0
	-/0.24/0	-/2.68/0	-/4.62/0	-/4.87/0	-/12.90/0	-/0.02/0
MR1(%)	0/1.46/0	0/6.57/0	0/6.57/0	0/8.03/0	0.42/9.25/0	0/0.02/0
	0/1.46/0	0/3.41/0	0/6.81/0	0/8.52/0	0.21/12.17/0	0/0/0
MR2(%)	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
MR3(%)	0/0.73/0	0/1.46/0	0/2.92/0	1.05/3.65/0	2.10/4.14/0	0/0/0
	0/0.73/0	0/0.97/0	0/2.19/0	0.42/4.14/0	1.05/5.84/0	0.04/0/0

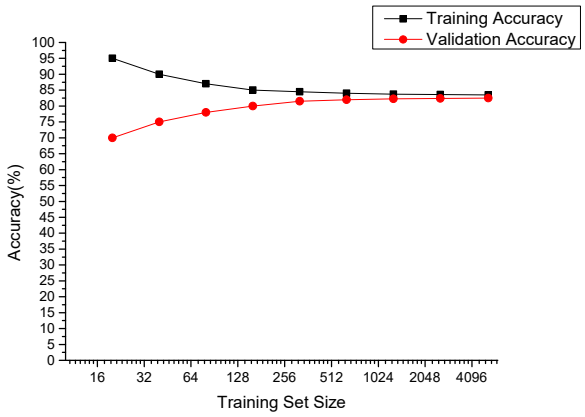


Fig. 2: Example of ideal learning curves

versions (Figure 3b) exist and it is hard to distinguish them from golden version by naked eye. In intermediate and large size, the curves of unclear mutant versions extremely resemble the curves of golden version, but have a bit of deviation in small size. We would like to learn more about their difference in small size.

Fortunately, we got a method [9] which could transform the figure of learning curve to some parameters. The validation and training error are represented as

$$\varepsilon_{validate} = a + \frac{b}{s^{\alpha}} \tag{7}$$

$$\varepsilon_{train} = a - \frac{c}{s^{\beta}} \tag{8}$$

where a is asymptote, b and c are amplitudes, α and β are exponents and s is the size of training set. If we make the assumption that $b = c$ and $\alpha = \beta$, then the sum and the difference of $\varepsilon_{validate}$ and ε_{train} are

$$\varepsilon_{validate} + \varepsilon_{train} = 2a \tag{9}$$

$$\varepsilon_{validate} - \varepsilon_{train} = \frac{2b}{s^{\alpha}} \tag{10}$$

If we use a $log{-}log$ representation of the sum and difference of the validation and training error as a function of the the training set size s, the sum will equal a constant $log(2a)$ while the difference will form a line which intersection is $log(2b)$ with the slope -α.

Before we used this method, we had done some validation experiments on the figure of learning curve. Sometimes the curve was not smooth enough for a single execution, thus we verified the effect of averaging the results after multiple execution and balanced data to overcome the disturbance in small size on several other data sets from UCI ML repository. We observed that both of them were effective and then adopted them into followed experiments.

We used line $y = y_0$ and $y = kx + t$ in $log - log$ figure to fit the learning curves, then the three parameters were

$$a = 0.5 \times 10^{y_0} \tag{11}$$

$$b = 0.5 \times 10^{t} \tag{12}$$

$$\alpha = -k \tag{13}$$

To eliminate the influence of certain data set, we first ran all the mutant versions on another data set $magic$ with SVM, then chose the mutant versions that passed t-test except the ones which were exactly equal to golden version in performance as the unclear part. We got 562 mutants which belonged to unclear type. The size series was from 10 to 100 with the interval of 10. In each size, we sampled ten times randomly and calculated the average value. Tabel XI shows the execution

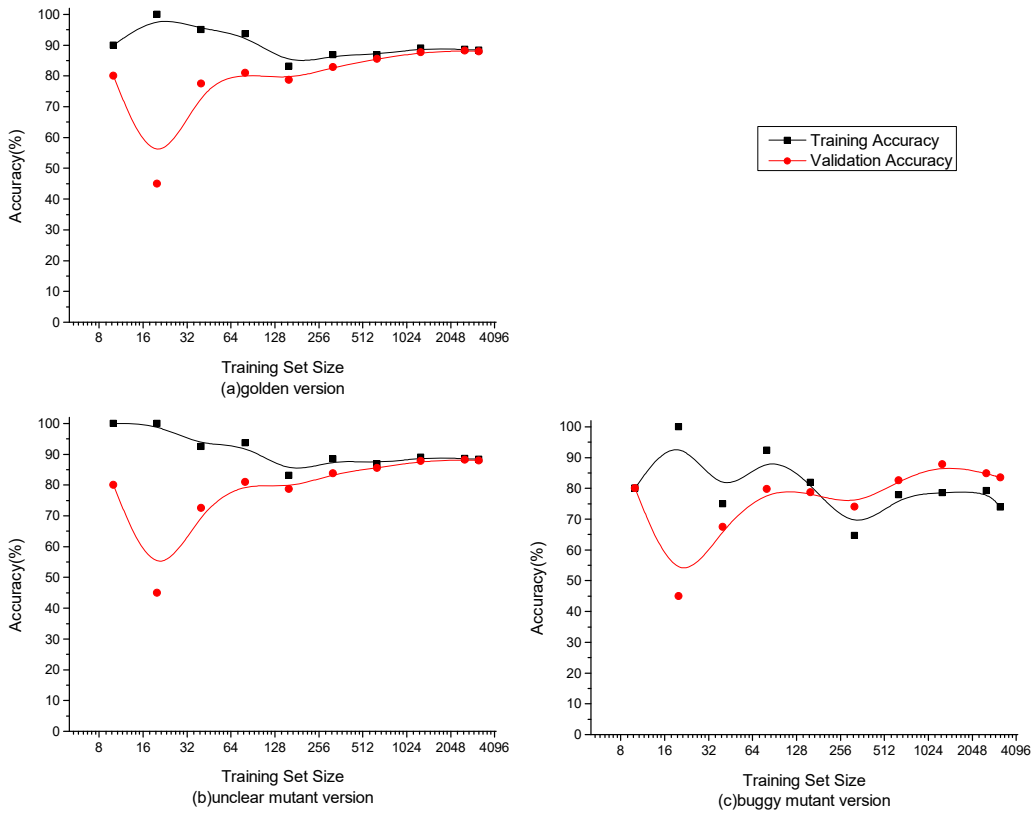


Fig. 3: Three typical results in learning curve

result. We got 463 mutant versions ran successfully on all small data sets. When calculating the parameters, 64 mutants were killed because the antilogarithms were not positive. Finally we owned 399 valid mutant versions. Figure 4 shows the distribution of three parameters. The results of golden version are $a \approx 24.670$, $b \approx 107.566$ and $\alpha \approx 0.996$, marked as dot lines in the figure.

TABLE XI: RESULT ON SMALL DATA SET

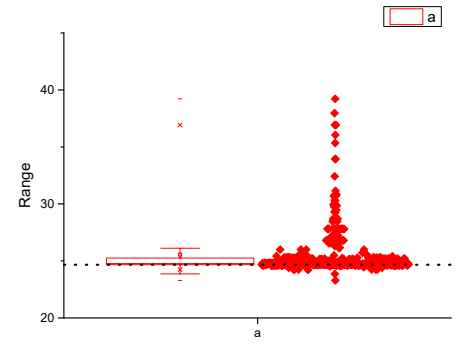
Size	10	20	30	40	50	60	70	80	90	100
Suc. num	518	513	502	511	484	514	505	509	507	522
Fail num	44	49	50	51	78	48	57	53	55	40

We observed that some mutant versions performed worse than golden version and a lot of outliers existed. However, it is still hard to distinguish the unclear type from golden version because some of them are quite close to golden version on parameters and some are even better. a is the asymptotic value, the smaller it is the lower the generalization error is. 285 mutant versions are larger than golden version on a and 104 ones are smaller. b is the amplitude, representing the distance to asymptote. 147 mutant versions are larger than golden version on b and 244 ones are smaller. α is the converge rate, the larger it is the faster the converge rate is. 239 mutant versions are larger than golden version on α and 152 ones are smaller.

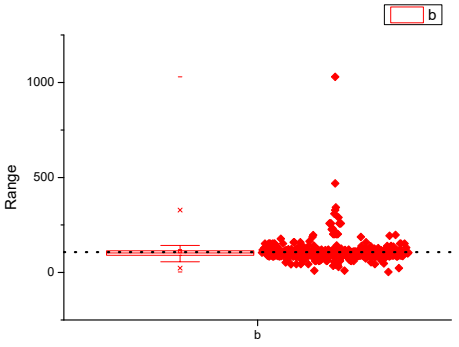
Then we tried to kill the stubborn mutants. The basic idea is learning from the curves and classify the curves of new program. In fact, it is the problem of time series classification. Below is the definition.

Definition 3: (Time series classification). *Given a set of samples* $\mathbb{X} = \{X_1, X_2, ..., X_m\}$*, each sample is a time series* $X_i = \{x_1, x_2, .., x_n, c\}$ *with a label* c *where* $x_j \in R^d$ *and* $c \in C$ *which is the set of labels. The task is predicting the label of a new sample.*

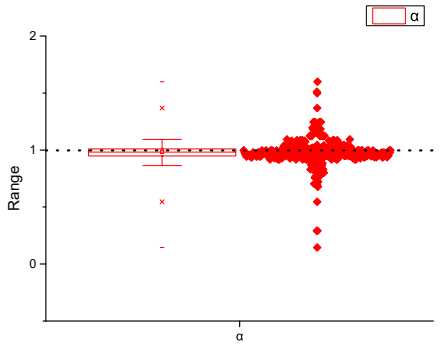
Here $|C| = 2$ and $d = 2$. The series are formed by the accuracy results of data sets in ascending order of size. We chose ten data sets from UCI ML repository. The range of sampling size is from 200 to 2000 with increase of 200, so that $n = 10$. The classifier is 1-NN which performs great in such task. We validated its ability on the series generated from same algorithm and same data set by cross-validation and it did work (f_1-measure was usually larger than 90%). Then for each data set, we trained the model with the series data of three of the four algorithms and tested on the left one. The



(a) Distribution of a



(b) Distribution of b



(c) Distribution of α

Fig. 4: Distribution of three parameters

details are shown in Table XII.

It is obvious that there is no data set which performs well with all algorithms. Thus we have tried changing the size of data set, the series length and the distribution of data set but the results are similar. One possible reason is that the series of different algorithms are quite different even on same data set and causes the non-i.i.d. samples.

In summary, we observed that different mutant versions had different shapes of learning curves and some could be distinguished from golden version quickly. And we tried to kill stubborn mutants by classifying the series through 1-NN but the result was not desired. Maybe we need a greater classifier or an ideal data set which can solve the non-i.i.d. problem.

TABLE XII: TIME SERIES CLASSIFICATION RESULT

Data set	C4.5	k-NN	Naive Bayes	SVM	Sum[a]
adult	n[b]	y[c]	n	n	1
bank	n	n	y	y	2
connect	n	n	n	y	1
EEG_Eye_State	n	y	n	n	1
HTRU_2	n	y	n	y	2
magic	n	y	n	y	2
nursery	y	y	n	y	3
occupancy_data	n	y	n	y	2
shuttle	n	y	y	y	3
Skin_NonSkin	n	n	n	n	0
Sum[d]	1	7	2	7	-

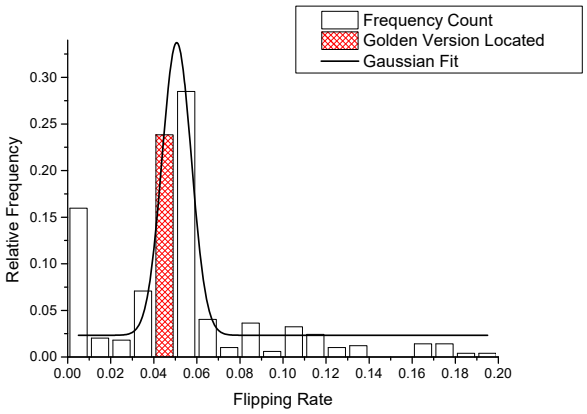
[a, d] the times of f_1-measure larger than 50% for each algorithm or data set
[b] f_1-measure $\leq 50\%$
[c] f_1-measure $> 50\%$

C. Robustness and Stability

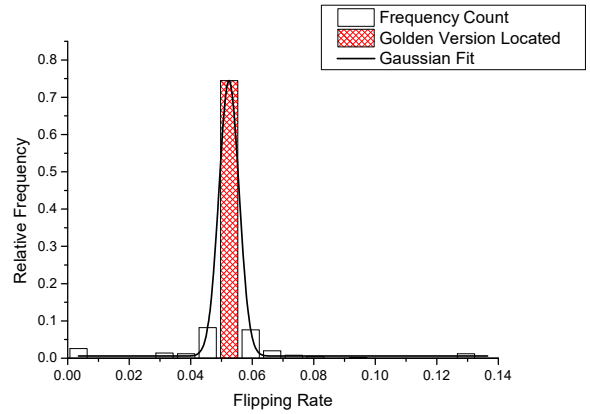
The insight is that the bugs in ML programs shall influence the robustness and stability of the programs. So we used a black-box attacking method based on difference of means to generate adversarial examples and tried to figure out their behaviours. An adversarial sample x_{adv} is generated in the form of $x_{adv} = x + \epsilon \cdot sign(\mu_t - \mu_o)$, where ϵ is the step, μ_t is the mean of the target class and μ_o is the mean of the original class. We can see that the difference does not play a part in the change of attribute value but supports the direction of gradient.

We accomplished this experiment on two-class data set $magic$ with C4.5, k-NN and SVM. The size of training set D_{tr} and test set D_{te} are both 1000. Then we generated the set of adversarial examples D'_{te} from D_{te}. Before that, we normalized the values of each attributes into [0,1]. The predicted labels of samples in two test sets are stored to calculate the flipping rate, e.g., if a mutant version generates $300 < x, x_{adv} >$ pairs which have opposite labels, the flipping rate is 0.3. ϵ ranges from 0.01 to 0.1 with the interval of 0.01. For the sake of limited page, we only show the result of SVM under $\epsilon = 0.01$ in Figure 5.

Figure 5a is the results for statistically stubborn mutants and Figure 5b is for statistically equivalent mutants. fr_{gv} is the flipping rate of golden version and the value is located in the bar with shadow. The flipping rates in each figure are divided into 20 boxes. The results with three algorithms are close and the higher the ϵ is, the higher the flipping rate is. The dispersion degree of statistically stubborn mutants is quite higher than that of statistically equivalent mutants. It shows that the definitions of two kinds of mutants are reasonable. There are many mutant versions which flipping rates are higher than golden version in statistically stubborn mutants. Surprisingly, some mutant versions have lower flipping rates than golden version and some rates even equal zero. It does not indicate that they have extremely high robustness and stability.



(a) statistically stubborn mutants, $\epsilon = 0.01$, $fr_{gv} = 0.049$



(b) statistically equivalent mutants, $\epsilon = 0.01$, $fr_{gv} = 0.049$

Fig. 5: Relative frequency of flipping rate with SVM

It is because they classify the most (sometimes all) of samples into same class falsely. And the small perturbations on data sets could not change the original classification results.

In summary, the mutants indeed changed the robustness and the stability of ML programs, and usually decreased them as expected. Besides, our definitions of two kinds of mutants are correct and appropriate. Even if we have tried a lot, the result is still negative and we need further exploration.

V. Threats to Validity

The statistical hypothesis testing was used in our study. However, two types of errors followed after we introduced it. Type I error means the rejection of H_0 mistakenly while type II error means the acceptance of H_0 when H_0 is invalid in fact. Here the 5x2cv t-test we used, which has these errors indeed, has been validated having lower probability of these errors than other testing method [21], e.g. McNemar's test.

We carried out experiments with four different ML algorithms and dozens of data sets. However, there are many other ML algorithms existing and some of them are quite different from we investigated, e.g. neural networks. Further more, the data sets we selected might not be representative of all natural data sets. Probably a data set following another distribution can lead to a new observation.

We sampled the data with different size in our experiment. The original data sets contains more samples and some are up to hundreds of thousands. A different sampling result may lead to different distribution, especially in small size, e.g. 200 samples.

The metamorphic relations we used are statistical but the amount is only four. Maybe there exist some relations which have more powerful abilities and can kill those stubborn mutants. The generation of metamorphic relations is a formidable task, especially for powerful ones. It requires the deep understanding of the algorithms and the knowledge of statistics in our task. In addition, due to the discrepancy in learning methods and loss functions, it is possible that a relation suitable for one ML algorithm is not good for another.

VI. Related Work

A. Metamorphic Testing

Xie et al. [22] used metamorphic testing to test ML programs. They proposed several metamorphic relations for k-nearest neighbours and Naive Bayes classifier. They introduced mutants and tried to kill them. The mutants were all method level and they randomly chose dozens of them to be tested. They did killed most of the selected mutants and the rest were equivalent mutants. However, it seems that they haven't experiment in a statistical way.

Murphy et al. [24] identified six metamorphic properties which were used to define metamorphic relations. Inspired from their work, we analyzed and summarized four statistical MRs for our experiments.

Not only the ML domain, metamorphic testing can be used on other domains [25] such as web services [26], computer graphics [27], embedded systems [28] and simulation [29].

B. Errors in Data Set

Data is vital to ML programs. Improper data may lead to a poor performance. Chakarov et al. [3] considered the errors in training data causing incorrect classifiers in a classification task. They proposed an approach based on Peral's theory of causation [30]. They ranked those samples in training set which were more likely to cause an incorrect classifier according to the score of Probability of Sufficiency. They built a tool named PSI, which modelled the computation of the PS score as a probabilistic program and calculated the score. Their tool is able to pick up a subset of training samples and fix their labels.

C. Testing Classifier

Classifier is the product of ML learner. Grosse et al. [5] helped end-user to test ML system, where users were assumed to lack software engineer skills. They provided three test selection methods which were fast enough to run in the interactive environment which the user was accustomed to using. The failure detection could be achieved in a small test suite with the help of test coverage.

D. Testing Deep Learning Systems

Pei et al. [2] implemented DeepXplore, which was the first white-box system for systematically testing deep learning systems. They used differential testing within a set of neural networks to find as many corner cases as possible. They designed a metric named neuron coverage, measuring how many neurons in a deep neural network were activated by inputs.

Sun et al. [31] proposed a set of four novel test criteria for deep neural networks. For each criterion, they developed an algorithm to generate test suite. The algorithm is based on linear programming.

Tian et at. [32] designed a systematic testing tool for automatically detecting erroneous behaviours of DNN-driven vehicles. It could explore different parts of the DNN logic systematically. They really found many behaviours which could lead to fatal crashes.

VII. Conclusion and Future Work

Our explorative study exposes some important aspects of the problem of bugs in the implementation of ML algorithms. First, we guess that bugs hidden in ML programs are not rare. This is based on the result that 8%-40% of the logically non-equivalent mutants are statistically indistinguishable from their golden versions, and other 15%-36% of the mutants have similar performance with the reference classifier on at least one natural data set. Second, these bugs could be very difficult to detect. None of the approaches including metamorphic testing, automated judging with learning curves and robustness to adversarial samples worked effectively in killing the statistically stubborn mutants.

Although these observations are not conclusive, we believe they are convincing enough to call for serious further research on testing ML programs.

Acknowledgement

This work is supported in part by National Key R&D Program of China under Grant No. 2017YFB1001801, National Natural Science Foundation under Grant Nos. 61690204, 61472177, and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

[1] I. Goodfellow and N. Papernot, "The challenge of verification and testing of machine learning," http://www.cleverhans.io/security/privacy/ml/2017/06/14/verification.html, June 2017.

[2] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," *SOSP*, 2017.

[3] A. Chakarov, A. Nori, S. Rajamani, S. Sen, and D. Vijaykeerthy, "Debugging machine learning tasks," *arXiv preprint arXiv:1603.07292*, 2016.

[4] D. H. Wolpert, "The lack of a priori distinctions between learning algorithms," *Neural Comput.*, vol. 8, no. 7, pp. 1341–1390, Oct. 1996. [Online]. Available: http://dx.doi.org/10.1162/neco.1996.8.7.1341

[5] R. B. Grosse and D. K. Duvenaud, "Testing mcmc code," *arXiv preprint arXiv:1412.5218*, 2014.

[6] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 402–411.

[7] E. Frank, M. A. Hall, and I. H. Witten, "The weka workbench," in *Data Mining: Practical Machine Learning Tools and Techniques*, 4th ed. Morgan Kaufmann, 2016, ch. Online Appendix.

[8] C. Perlich, *Learning Curves in Machine Learning*. Springer US, 2011.

[9] C. Cortes, L. D. Jackel, S. A. Solla, V. Vapnik, and J. S. Denker, "Learning curves: Asymptotic values and rate of convergence," in *Advances in Neural Information Processing Systems*, 1994, pp. 327–334.

[10] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of machine learning*. MIT press, 2012.

[11] S. Shalevshwartz, O. Shamir, N. Srebro, and K. Sridharan, "Learnability and stability in the general learning setting." 2009.

[12] R. Kohavi, "Glossary of terms," *Machine Learning*, vol. 30, pp. 271–274, 1998.

[13] B. D. Ripley, *Pattern recognition and neural networks*. Cambridge university press, 2007.

[14] G. H. John and P. Langley, "Estimating continuous distributions in bayesian classifiers," in *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 1995, pp. 338–345.

[15] S. L. Salzberg, "C4. 5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993," *Machine Learning*, vol. 16, no. 3, pp. 235–240, 1994.

[16] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[17] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy, "Improvements to platt's smo algorithm for svm classifier design," *Neural computation*, vol. 13, no. 3, pp. 637–649, 2001.

[18] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[19] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," in *Mutation testing for the new century*. Springer, 2001, pp. 34–44.

[20] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[21] T. G. Dietterich, "Approximate statistical tests for comparing supervised classification learning algorithms," *Neural Computation*, vol. 10, no. 7, pp. 1895–1923, 1998.

[22] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software*, vol. 84, no. 4, pp. 544–558, 2011.

[23] J. Ding, D. Zhang, and X.-H. Hu, "An application of metamorphic testing for testing scientific software," in *Proceedings of the 1st International Workshop on Metamorphic Testing*. ACM, 2016, pp. 37–43.

[24] C. Murphy, G. E. Kaiser, L. Hu, and L. Wu, "Properties of machine learning applications for use in metamorphic testing." in *SEKE*, vol. 8, 2008, pp. 867–872.

[25] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[26] C.-a. Sun, G. Wang, B. Mu, H. Liu, Z. Wang, and T. Y. Chen, "Metamorphic testing for web services: Framework and a case study," in *Web Services (ICWS), 2011 IEEE International Conference on*. IEEE, 2011, pp. 283–290.

[27] J. Mayer and R. Guderlei, "On random testing of image processing applications," in *Quality Software, 2006. QSIC 2006. Sixth International Conference on*. IEEE, 2006, pp. 85–92.

[28] T. Tse and S. S. Yau, "Testing context-sensitive middleware-based software applications," in *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*. IEEE, 2004, pp. 458–466.

[29] K. Sim, W. Pao, and C. Lin, "Metamorphic testing using geometric interrogation technique and its application," *ECTI Transactions on Computer and Information Technology (ECTI-CIT)*, vol. 1, no. 2, pp. 91–95, 2005.

[30] J. Pearl, "Probabilities of causation : Three counterfactual interpretations and their identification," *Synthese*, vol. 121, pp. 93–149, 1999.

[31] Y. Sun, X. Huang, and D. Kroening, "Testing deep neural networks." *arXiv: Learning*, 2018.

[32] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," *arXiv preprint arXiv:1708.08559*, 2017.