2019 IEEE International Conference on Artificial Intelligence Testing (AITest)

Fault Detection Effectiveness of Metamorphic Relations Developed for Testing Supervised Classifiers

Prashanta Saha
School of Computing, Montana State University
Bozeman, USA
prashantasaha@montana.edu

Upulee Kanewala*
School of Computing, Montana State University
Bozeman, USA
upulee.kanewala@montana.edu

***Abstract*—In machine learning, supervised classifiers are used to obtain predictions for unlabeled data by inferring prediction functions using labeled data. Supervised classifiers are widely applied in domains such as computational biology, computational physics and healthcare to make critical decisions. However, it is often hard to test supervised classifiers since the expected answers are unknown. This is commonly known as the *oracle problem* and metamorphic testing (MT) has been used to test such programs. In MT, metamorphic relations (MRs) are developed from intrinsic characteristics of the software under test (SUT). These MRs are used to generate test data and to verify the correctness of the test results without the presence of a test oracle. Effectiveness of MT heavily depends on the MRs used for testing. In this paper we have conducted an extensive empirical study to evaluate the fault detection effectiveness of MRs that have been used in multiple previous studies to test supervised classifiers. Our study uses a total of 709 reachable mutants generated by multiple mutation engines and uses data sets with varying characteristics to test the SUT. Our results reveal that only 14.8% of these mutants are detected using the MRs and that the fault detection effectiveness of these MRs do not scale with the increased number of mutants when compared to what was reported in previous studies.**

***Index Terms*—Metamorphic testing, Random testing, Supervised classifiers, Metamorphic Relations, Mutation Analysis, Machine Learning**

I. Introduction

Supervised classifiers are widely used for making predictions in diverse domains. For instance, over fifty real world computational applications use support vector machines for classification [1]. As these types of applications are becoming part of our daily life, ensuring their quality becomes even more important [2]. In such applications, formal proofs of the underlying algorithm does not always guarantee that it implements that algorithm correctly. Therefore, software testing is imperative to assure the quality of these systems.

Often, conventional software testing approaches may not be feasible for assuring the quality of supervised classifiers because of the absence of a test oracle that determines the correctness of produced test outputs. This class of software applications is often referred to as "non-testable programs" [3]. Further, usually supervised classifiers are not 100% accurate. Thus, an incorrect prediction does not necessarily mean that there is a fault in the program. These characteristics of supervised classifiers make it hard to detect subtle faults in these applications.

To date, limited work has been done on systematic testing of software systems that incorporate machine learning. Among them, metamorphic testing (MT) has been used widely for testing software applications that uses supervised machine learning algorithms [4]–[7]. MT uses metamorphic relations (MRs) for testing the software under test (SUT), where MRs act as partial oracle [8], [9]. A MR specifies how the outputs should change according to a specific change made to the input. Thus, from existing test cases (named as *source test cases*) MRs are used to generate new test cases (named as *follow-up test cases*). If the changes found between the outputs of the source and follow-up test cases are not as expected according to the MR, then there is a defect in the SUT. Thus, using MRs we can address the oracle problem presented by supervised machine learning classifiers.

Several previous studies have defined MRs for testing supervised classifiers. Xie et al. proposed a set of MRs based on user expectations to validate supervised classifiers [5]. Dwarakanath et al. developed MRs for two image classifiers that are based on support vector machines and deep learning [6]. Ding et al. developed three levels of MRs to test and validate a deep learning framework [7]. The evaluations conducted in these studies to measure the fault detection effectiveness of the developed MRs is fairly limited due to the number of mutants used. For example Xie et al. used 24 mutants and Dwarakanath et al. used 22 mutants in total. These numbers are significantly low especially considering the number of classes and the number of lines of source code involved with these SUTs.

To overcome this limitation, in this paper, we report the findings of a large scale experiment that we conducted to evaluate the fault detection effectiveness of MRs developed for supervised classifiers. In this experiment we used a total of 709 reachable mutants (i.e. the mutated statement in the mutant was executed with the test cases) that were generated for a real world supervised classifier implementation from Weka [10].

*Corresponding author

978-1-7281-0492-8/19/$31.00 ©2019 IEEE
DOI 10.1109/AITest.2019.00019

Our results show a significant reduction of the fault detection effectiveness with the increased number of mutants.

Rest of the paper organized as follows: Section II describes the background of this work, including an overview of supervised machine learning and the k-Nearest Neighbors algorithm, which is used as the SUT in this study. Section III discusses more about MT and the MRs used for testing. In Section IV we discuss the details of our experimental approach and mutation analysis. Section V presents the results and their analysis. Section VI identifies the related work and Section VII contains our conclusions and future work.

II. Background

A. Supervised Machine Learning Classifiers

Supervised classification is the task of deducing a function from labeled training data such that it can be used to predict unknown labels on test data. Training data can be represented by two vectors of size k. One vector is the training samples $S =< s_0, s_1, ..., s_{k-1} >$ and the other one is the class labels $C =< c_0, c_1, .., c_{k-1} >$ where, c_i is the class label for s_i. Each sample s_i has m features from which the prediction function will be learned. Class labels are a finite set and each class label c_i is an element of it, i.e. $c \in L =< l_0, l_1, ..., l_{n-1} >$, where n is the number of class labels [5].

Supervised machine learning applications execute in two phases: the *training phase* and *testing phase*. In the *training phase*, a set of training samples are used by a supervised classification algorithm to learn a prediction function. To develop the prediction function, the supervised learning algorithm would analyze how the attributes relate to the class label. In the *testing phase*, the prediction function is applied to unseen data known as the *test set*, where the class labels are unknown. The application attempts to predict the class label for each instance in the test set using the learned prediction function [5]. Some of the commonly used supervised classification algorithms are K-Nearest Neighbors [11], Naive Bayes [12] and Support Vector Machine [1].

B. K-Nearest Neighbors

This study uses an open-source implementation of the K-Nearest Neighbors (kNN) algorithm as the SUT. kNN is particularly chosen due to its popularity in the machine learning community and is used in domains such as recommendation systems, semantic searching and anomaly detection etc. Further, Xie et al. used kNN in their study and using the same algorithm would allow us to do a comparison with their results [5]. However, the MT approach discussed here should be applicable to other supervised learning algorithm implementations.

In kNN, for a sample training set S, each sample set has m attributes,$< att_0, att_1, ..., att_{m-1} >$, and also n classes,$< l_o, l_1, ..., l_{n-1} >$. The sample test data is t_s,$< a_0, a_1, ..., a_{m-1} >$. kNN computes the distance between each sample training set and the test case. Euclidean distance metric is one of the most popular approach to measure distance. For sample $s_i \in S$, the value for each attribute is $< sa_0, sa_1, ..., sa_{m-1} >$. And the euclidean distance formula is:

$$dist(s_i, t_s) = \sqrt{\sum_{j}^{m-1} (sa_j - a_j)^2}$$

Once the distance is calculated, kNN selects the k nearest training samples for the test data after sorting all the distances. These k samples from the training set are considered as the *k-nearest neighbors* of the test case. Then, kNN calculates the proportion of each label in the selected k-nearest neighbors. The class label with the highest proportion is predicted as the label for the test data.

III. Metamorphic Testing for supervised classifiers

Often, programs exhibit properties such that if the test input is changed in a way that the new output can be predicted based on the original output. In MT, such properties (known as MRs) are used as partial oracles to conduct testing [13], [14]. In practice one can easily apply MT. As the first step, it is necessary to identify MRs that can relate multiple pairs of the inputs and outputs of the SUT. Then, source test cases are generated using techniques like random testing, structural testing or search based testing and the corresponding follow-up test cases are constructed based on the MRs. In our previous studies we investigated how the fault detection effectiveness of MT varies with various source test case generation techniques such as different structural coverage based approaches and our results show that coverage based source test case generation outperforms randomly generated source test cases [15]. After executing the source and the follow-up test cases on the SUT we can check if there is a change in the output that matches the MR, if not the MR is considered as violated. Violation of a MR during testing indicates faults in the SUT. Since MT checks relationship between inputs and outputs of a test program, we can use this technique when the expected result of individual test output is unknown.

For example, *Consistence with affine transformation* MR described in Section III-A can be used to test a kNN classifier. Source test case for kNN can be randomly generated (see Table I for an example - training data on the left and test data on the right). After executing this source test case, the output will be the class label predicted for that test case which is 0 for this example. To generate follow-up test case, we apply the input transformation described in the above MR, where an arbitrary affine transformation function is applied to the attributes of both the training data and test data. After executing the follow-up test case the output is 0 which is the predicted class label for the transformed test data. To satisfy this MR both the source and follow-up test case outputs should be same. Therefore, in this example, the considered MR is satisfied for the given source and follow-up test cases.

A. Identified MRs for Testing kNN

Murphy et al. [4] suggested six MRs (additive, multiplicative, permutative, invertive, inclusive and exclusive) that can

be applied to machine learning applications including both supervised and unsupervised ML. Xie et al. developed a set of MRs based on the user expectations of supervised classifiers [5]. In our study, we mainly use the MRs developed by Xie et al. to test kNN. Some variations of the same MRs are used in some recent studies as well [6]. Below we provide a brief description of these MRs (formal definitions can be found in [5]).

MR1: Consistence with affine transformation. If we apply some affine transformation function, $f(x) = kx + b, (k \neq 0)$, to every value x in some subset of features in the training and testing data, and then create a new model using this data, the predictions made by the model should be unchanged.

MR2: Permutation of the attribute. If we permute the order of the attributes, or features, of all the samples in the training and testing data, the result of the predictions of the test data should not change.

MR3: Addition of uninformative attributes. If we add some new feature that is equally associated with all classes, the predictions of the test data should not be changed.

MR4: Consistence with re-prediction. Suppose we predict some test case t as class l_i. If we append t to our training data and re-create the model, t should still be classified as class l_i.

MR5: Additional training sample. Suppose we predict some test case t as class l_i. If we duplicate all samples of class l_i in our training data and re-classify our test data, t should still be classified as l_i. More generally, every test case predicted as class l_i should still be predicted as class l_i with the duplicated samples.

MR6: Addition of classes by re-labeling samples. For some test cases not of class l_i, we switch the class label from x to x^*. Then every test case predicted as class l_i should still be predicted as class l_i with the re-labeled samples.

MR7: Permutation of class labels. If we permute the order of the class labels with some random permutation $p(l_i)$ where l_i is a class label, all test cases which were predicted as l_i should now be predicted as $p(l_i)$.

MR8: Addition of informative attribute. If we add some new feature that is strongly associated with one class, l_i, then for every prediction that was class l_i, the prediction with this new attribute should also be class l_i.

MR9: Addition of classes by duplicating samples. Suppose we duplicate every class except for n, and give them all a new class. For example, if we originally had class labels of 1, 2, 3, and 4, then we would create class labels of 1, 1*, 2, 2*, 3, 4, 4*. Then every test case predicted as class l_i (class 3 in this example) should still be predicted as class l_i with the duplicated samples.

MR10: Removal of classes. If we remove some class l_i, the remaining predictions should remain unchanged.

MR11: Removal of samples. If we remove samples that have not been predicted as class l_i, then all cases which were predicted as l_i should remain unchanged.

When using these MRs for testing kNN, it is important to select the appropriate value for k (i.e. number of nearest neighbours) such that the MR becomes a necessary property for kNN. Table II shows the k values used with each MR for verification of kNN [5].

TABLE I
SAMPLE DATA SET

@attribute pictures numeric @attribute paragraphs numeric @attribute files numeric @attribute files2 numeric @attribute profit {0,1,2,3,4} @data 45,3,16,38,0 15,87,89,46,4 59,77,94,11,0 86,89,94,15,2 80,28,94,11,4 23,12,47,41,1 94,15,22,15,0 95,26,97,76,3 50,90,0,72,2 33,46,47,95,0	@attribute pictures numeric @attribute paragraphs numeric @attribute files numeric @attribute files2 numeric @attribute profit {0,1,2,3,4} @data 6,40,8,89,0

IV. EXPERIMENTAL STUDY

The goal of this experimental study is to conduct an in-depth evaluation of the fault detection effectiveness of the MRs listed in Section III-A. We used the kNN implementation in Weka 3.5.7 as the SUT [10].

A. Research Questions

We conducted a set of experiments to answer the following research questions:

1) **How does the fault detection rate of MRs vary as the number of mutants increase?** As we mentioned above the fault detection effectiveness of these MRs were measured using a limited set of mutants in previous studies [5]. But it is important to use a reasonable number of mutants to evaluate the fault detection effectiveness using the mutation killing rate. Therefore we increase the number of mutants significantly and measure the killing rate.
2) **Does the fault detection rate of MRs change with varying the data set size in the source test cases?** There are two major components in MT that determines its fault detection effectiveness: the MRs and the source test cases. We examined whether varying the data set size of the source test case can effect the mutant killing rate for a given MR.
3) **Does the fault detection effectiveness vary with the mutation engine used to generate the mutants?** As we discussed above the underlying process used by MuJava and Major for generating mutants is different. The purpose of this research question is to see whether that one category of mutants are hard to kill than the other.

B. Source and Follow-up Test Cases

In an individual source test case there is a training set and a test set. Similar to Xie et al. [5], we used a random

approach to generate these source test cases. In each training and test set, there are four numerical attributes that are named as: $\{pictures, paragraphs, files, files2\}$. The class label $profit$ can have five values $\{0, 1, 2, 3, 4\}$. The value of each attribute is randomly selected and ranges within [0, 100]. The value of the class labels are also selected randomly. The training set size ranges within [10, 200]. Table I shows a sample source test case, with the training data set on the left and test data set on the right.

We transform the source test cases to obtain the corresponding follow-up test cases according to the MRs described in Section III-A. Multiple source and follow-up test case pairs are generated for each MR by varying the number of samples in the training data set as well as the size of source test case.

We executed all the source and follow-up test case pairs on the original kNN implementation and validated the outputs against each MR before proceeding to the mutation analysis described below. The original kNN implementation did not report any MR violations.

TABLE II
METAMORPHIC RELATIONS FOR KNN USED IN MUTATION ANALYSIS

k=3	MR1 Consistence with affine transformation MR2 Permutation of the attribute MR3 Addition of uninformative attributes MR4 Consistence with re-prediction MR5 Additional training sample MR6 Addition of classes by re-labeling samples
k=1	MR7 Permutation of class labels MR8 Addition of informative attribute MR9 Addition of classes by duplicating samples MR10 Removal of classes MR11 Removal of samples

C. Mutation Analysis

To evaluate the fault detection effectiveness of the MRs described in Section III-A, we use a mutation engine to systematically inject defects into the SUT. Mutation testing has been extensively used to evaluate fault detection effectiveness, as many experiments suggest that mutants are a proxy to the real faults for comparing testing techniques [16]. As we mentioned above, previous studies used mutation analysis to evaluate the fault detection effectiveness of MRs [5]. But, the number of mutants used in the mutation analysis is quite low compared to the size of the SUT's used in those experiments.

In our evaluation, we applied MuJava [17] and Major [18] tools to systematically generate mutants for kNN in Weka-3.5.7. MuJava is a powerful and automatic mutation analysis system, which can support both method-level and class-level mutation operators. MuJava provides various types of mutants, including inter-class, intra-class, inter-method and intra-method level of mutants. In this experiment, we only included the intra-method level of mutants. Major is a mutation testing framework which manipulates the abstract syntax tree of the SUT. Similar to MuJava, we only used the intra-method level mutant operators from the Major tool.

MuJava and Major allows users to define which parts of the source code needs to be mutated. Since, Weka is a large scale software with about 16.4M source code and our experiments only focuses on the functionality of the kNN classifier, we only selected the class files which are directly related to the kNN classifier according to its hierarchy structure. Table III shows the names of the selected class files in our mutation analysis. We generated all possible mutants for the 6 class files in Table III. After excluding the mutants that caused compilation errors, runtime exception as well as equivalent mutants, we have obtained a total of 1500 mutants from the MuJava and Major mutation tools. From those mutants we identified 609 mutants generated by MuJava that are reachable by the test cases that we described above. From the mutants generated by Major, we randomly selected 100 mutants that are reachable by the test cases due to time limitations. The distribution of mutants between the two tools are described in Table IV.

TABLE III
SELECTED FILES FOR MUTATION ANALYSIS

kNN
weka.classifiers.lazy.IBk.java
weka.core.Attribute.java
weka.core.neighboursearch.LinearNNSearch.java
weka.core.neighboursearch.NearestNeighbourSearch.java
weka.core.NormalizableDistance.java
weka.core.EuclideanDistance.java

TABLE IV
DETAILS OF MUTANTS

Tool Name	Total # of mutants generated	# of mutants used
MuJava	2383	609
Major	987	100

V. RESULTS AND DISCUSSIONS

Below we discuss the results of our experiments and provide answers to our research questions.

1. How does the fault detection rate of MRs vary as the number of mutants increase?

Out of the 709 mutants (609 MuJava + 100 Major) used in this experiment, only 105 (14.8%) mutants could be killed using the MRs. This is a significant decrease in the mutation killing rate compared to what is reported in Xie et al., where 19 out of the 24 (79%) mutants were killed by the same set of MRs [5]. We think that the reason for this significant decrease in the mutation killing rate is due to the fact that Xie et al. used a selected set of mutation operators to generate the mutants used in their experiment and those mutants do not provide a good representation of the potential faults in the SUT.

To further evaluate how the mutation killing rate varies when increasing the number of mutants, we executed the MRs with mutant sets of size 100, 400 and 600 that are randomly selected from the mutants generated by the MuJava tool. We used 10 randomly generated source test cases for executing each of the MRs. Figure 1 shows the mutant kill rates for each MR for the three mutant sets. As, shown in

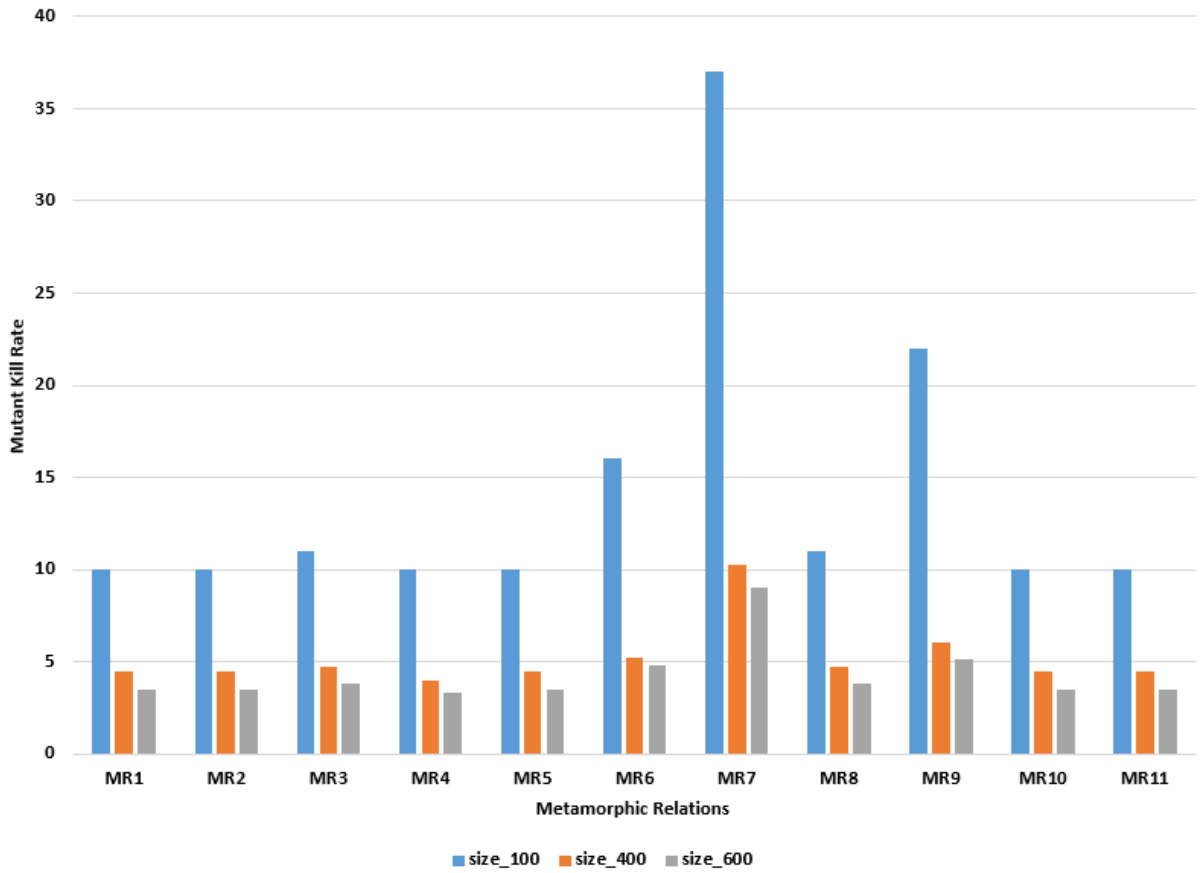

Fig. 1. Mutant kill rate for each MR by varying Mutant Size.

Figure 1, mutant kill rate for all the MRs reduced when the number of mutants were increased. In particular, the killing rates for sizes 400 and 600 are significantly lower compared to size 100 for MR6, MR7, and MR9.

A Significant decrease in the mutation killing rate with the increased number of mutants.

2. Does the fault detection rate of MRs change with varying the data set size in the source test cases?

The goal of this research question is to identify whether fault detection effectiveness of MRs vary with the size of the data sets used as the source test case. In this experiment, we created data sets of 18 different sizes where the number of samples vary from 30 to 200. These data sets were executed on 100 mutants that were randomly selected from the MuJava mutants.

Figure 2 shows the mutant killing rate for each MR with varying number of samples in the source test cases. It is interesting to note that mutants killing rate is low for all the MRs across the different data sets ranging between 10% and 37%. Only MR7 and MR9 is showing some variation in the killing rate which is 6% and 4%, respectively. On the other hand, the rest of MRs have a constant mutants killing rate despite the difference in data set sizes used as the source test cases. In summary, varying the size of the random sample data has no significant effect on the fault detection effectiveness of the considered MRs. But it might be possible to increase the fault detection effectiveness by generating test data based on some test coverage criterion as we discussed in our previous study [15].

No major changes in the kill rate with varying data set size.

3. Does the fault detection effectiveness vary with the mutation engine used to generate the mutants?

In order to answer this research question, we used mutants from MuJava and Major mutation tools. We used 10 randomly generated sample data sets as source test cases for each MR and executed them on a set of 100 randomly selected mutants from MuJava and a set of 100 randomly selected mutants from Major. We report the results of this evaluation in Figure 3.

As shown in Figure 3, the overall mutant killing rate on the MuJava and Major mutants is 43.6% and 35.1%, respectively. When comparing the results at the individual MR level, it is noticeable that there is some consistency in the killing rate for each MR between these two tools. For example, for both

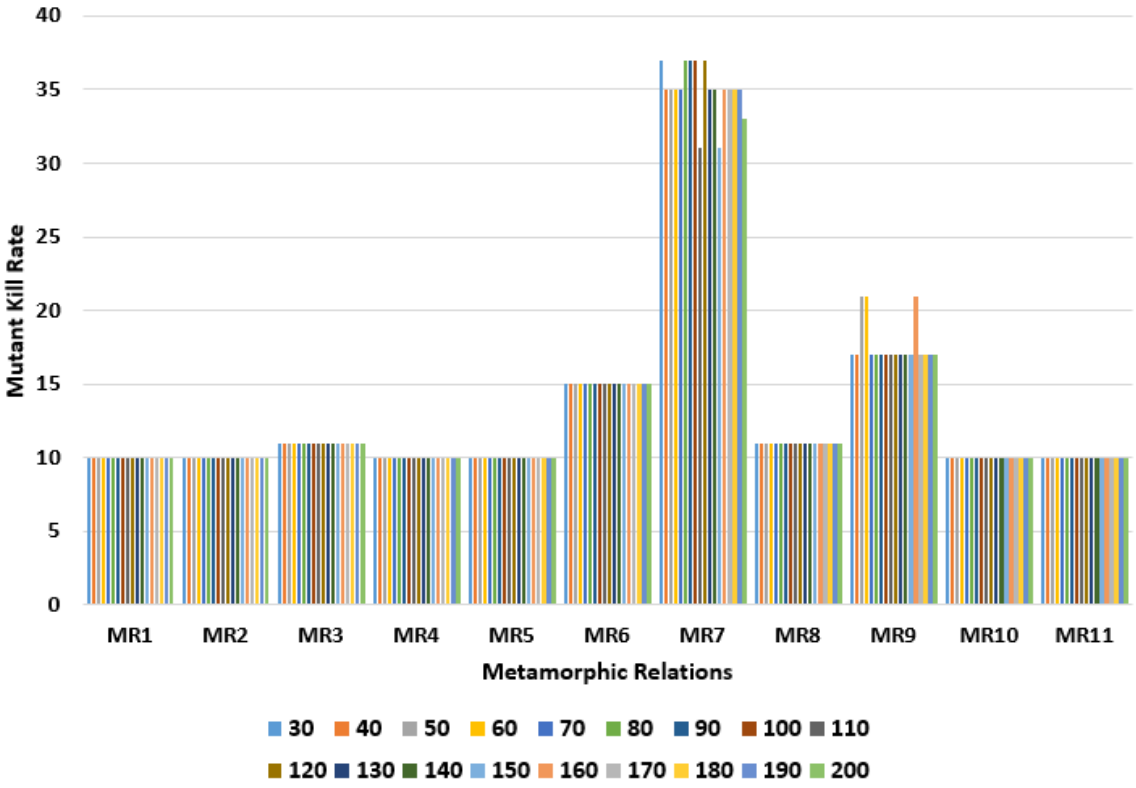

Fig. 2. Mutant kill rate by each MR in kNN with varying data Set.

tools, MR7 and MR9 have comparatively higher mutants killing rate than the other MRs. Also it is interesting to note that for all the MRs except MR7, killing rate of Major mutants is higher than that of the MuJava mutants even though the overall killing rate is higher for MuJava mutants. This indicates that the mutation killing is dominated by MR7. In summary, overall MuJava mutants are easier to kill, while with majority of MRs Major mutants are easier to kill.

Most effective MRs perform consistently across the two mutation tools.

VI. Related Work

There has been a significant amount of work done on applying machine learning to solve software testing problems compared to testing machine learning applications [19]. For example, machine learning has been used to predict likely MRs for a given programs [20]–[23].

MT has been applied to test different types of machine learning applications [4]. A case study done on real world machine learning application framework shows that MRs can effectively detect faults [5]. A recent work [6] investigated the application of metamorphic testing to test complex machine learning algorithms such as SVMs with non-linear kernels and deep residual neural networks (ResNET). The technique was able to successfully detect mutants in open-source machine learning applications.

MRs has been proven to be a core element of MT. In image processing applications MT was used by Tahir et al. [24]. They have shown that only few MRs that are related to specific images are more effective in detecting faults than others. Regardless of conducting MT, MRs have been used for the augmentation of the machine learning models [25]. Here MRs were identified based on properties of the input data and the usage of the binary classification model. Hui et al. [26] has proposed a semi automated MT approach for GIS testing that used the superficial area calculation program to illustrate the process of the testing approach. They have developed a MR model to generate compound MRs.

Some research efforts are reported on how to identify effective MRs. Asrafi et al. [27] have observed a correlation between the test code coverage achieved by an MR and its fault detection effectiveness. In object oriented software testing a method of constructing MRs based on algebraic specification has been proposed [28]. This method provides low MRs redundancy and improves the efficiency of software testing. μMT [29] a MR construction tool that uses data mutation to construct an input relation and the generic mapping rule

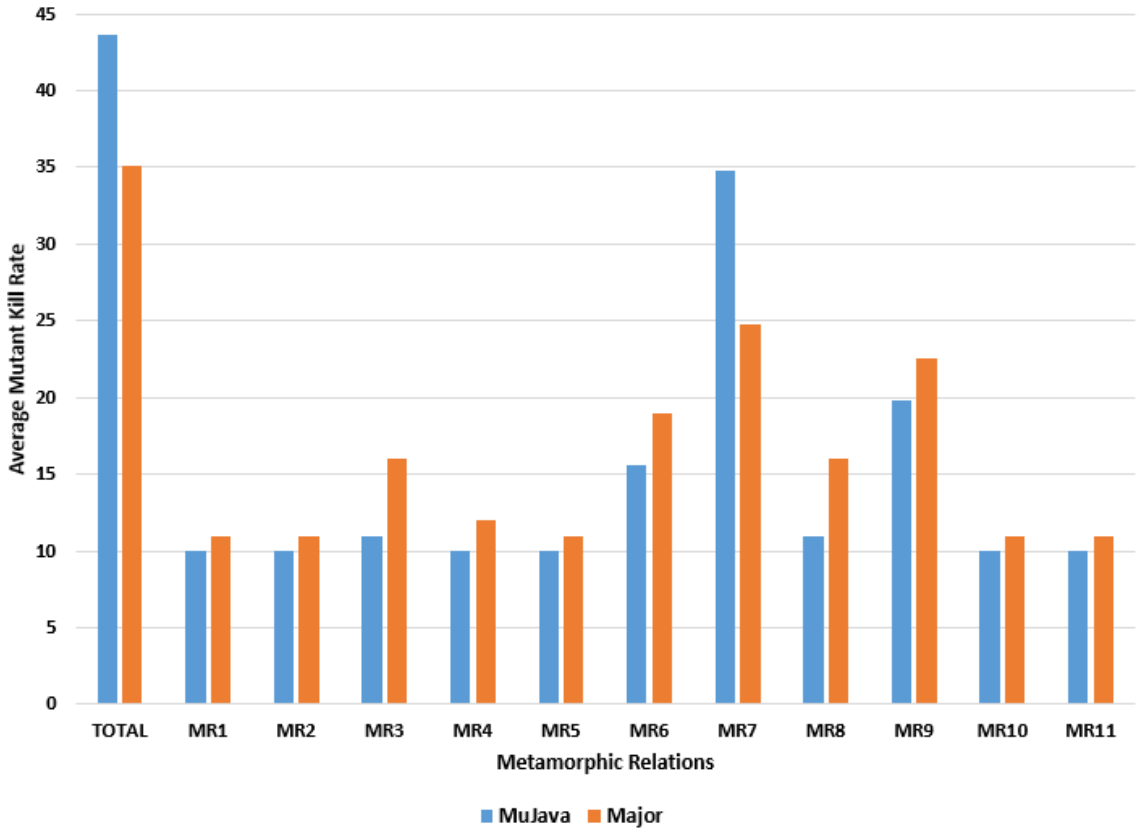

Fig. 3. Average mutants kill rate by each MR for MuJava and Major tool.

associated with each mutation operator to construct output relation.

VII. Conclusions and Future Work

Previous studies have developed MRs for conducting MT on supervised classifiers. But, a major drawback of these studies is the limited number of mutants used to evaluate their fault detection effectiveness. In this paper, we empirically evaluated the fault detection effectiveness of MRs developed for supervised classifiers using a set of 709 reachable mutants, which is a significant increase in the number of mutants compared to what is used in the previous studies.

Our study shows that the MRs identified based on user expectations of supervised classifiers are not as effective in detecting faults as claimed in previous studies. Out of the 709 mutants only 14.8% of mutants could be detected using these MRs. Our study also shows that changing the size of randomly generated data used as source test cases does not have an effect on the fault detection effectiveness of these MRs.

In the future, we plan to develop MRs based on specific algorithmic properties of commonly used supervised classifiers. We think such MRs will have higher fault detection effectiveness compared to the ones we investigated in this study. We also plan to investigate ways to develop more effective source test cases for this domain using various data distributions. Further, we plan to extend this experiment to other machine learning algorithms including deep learning algorithms.

Acknowledgment

This work is supported by award number 1656877 from the National Science Foundation. Any Opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the National Science Foundation.

References

[1] "Svm application list," http://www.clopinet.com/isabelle/Projects/SVM/applist.html.

[2] R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, *Machine Learning: An Artificial Intelligence Approach*. Springer Publishing Company, Incorporated, 2013.

[3] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982. [Online]. Available: http://dx.doi.org/10.1093/comjnl/25.4.465

[4] C. Murphy, G. E. Kaiser, L. Hu, and L. Wu, "Properties of machine learning applications for use in metamorphic testing," in *SEKE*, 2008.

[5] X. Xie, J. W. K. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *J. Syst. Softw.*, vol. 84, no. 4, pp. 544–558, Apr. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2010.11.920

[6] A. Dwarakanath, M. Ahuja, S. Sikand, R. M. Rao, R. P. J. C. Bose, N. Dubash, and S. Podder, "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*, ser. ISSTA 2018. New York, NY, USA: ACM, 2018, pp. 118–128. [Online]. Available: http://doi.acm.org/10.1145/3213846.3213858

[7] J. Ding, X. Kang, and X.-H. Hu, "Validating a deep learning framework by metamorphic testing," in *Proceedings of the 2Nd International Workshop on Metamorphic Testing*, ser. MET '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 28–34. [Online]. Available: https://doi.org/10.1109/MET.2017..2

[8] T. Y. Chen, T. H. Tse, and Z. Zhou, "Fault-based testing without the need of oracles," *Information Software Technology*, vol. 45, pp. 1–9, 2003.

[9] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 4:1–4:27, Jan. 2018. [Online]. Available: http://doi.acm.org/10.1145/3143561

[10] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd ed. San Francisco: Morgan Kaufmann, 2005.

[11] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Trans. Inf. Theor.*, vol. 13, no. 1, pp. 21–27, Sep. 2006. [Online]. Available: https://doi.org/10.1109/TIT.1967.1053964

[12] I. Rish, "An empirical study of the naive bayes classifier," Tech. Rep., 2001.

[13] T. Y. Chen, "Metamorphic testing: A simple method for alleviating the test oracle problem," in *Proceedings of the 10th International Workshop on Automation of Software Test*, ser. AST '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 53–54. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819261.2819278

[14] T. Y. Chen, F. C. Kuo, D. Towey, and Z. Q. Zhou, "Metamorphic testing: Applications and integration with other methods: Tutorial synopsis," in *2012 12th International Conference on Quality Software*, Aug 2012, pp. 285–288.

[15] P. Saha and U. Kanewala, "Fault detection effectiveness of source test case generation strategies for metamorphic testing," in *Proceedings of the 3rd International Workshop on Metamorphic Testing*, ser. MET '18. New York, NY, USA: ACM, 2018, pp. 2–9. [Online]. Available: http://doi.acm.org/10.1145/3193977.3193982

[16] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments? [software testing]," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, May 2005, pp. 402–411.

[17] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: An automated class mutation system: Research articles," *Softw. Test. Verif. Reliab.*, vol. 15, no. 2, pp. 97–133, Jun. 2005. [Online]. Available: http://dx.doi.org/10.1002/stvr.v15:2

[18] R. Just, "The major mutation framework: Efficient and scalable mutation analysis for java," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 433–436. [Online]. Available: http://doi.acm.org/10.1145/2610384.2628053

[19] L. C. Briand, "Novel applications of machine learning in software testing," in *2008 The Eighth International Conference on Quality Software*, Aug 2008, pp. 3–10.

[20] U. Kanewala, J. M. Bieman, and A. Ben-Hur, "Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels," *Softw. Test., Verif. Reliab.*, vol. 26, no. 3, pp. 245–269, 2016. [Online]. Available: http://dx.doi.org/10.1002/stvr.1594

[21] U. Kanewala and J. Bieman, "Using machine learning techniques to detect metamorphic relations for programs without test oracles," in *In Proc. 24th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, Pasadena, California, USA, Nov. 2013, pp. 1–10.

[22] K. Rahman and U. Kanewala, "Predicting metamorphic relations for matrix calculation programs," in *Proceedings of the 3rd International Workshop on Metamorphic Testing*, ser. MET '18. New York, NY, USA: ACM, 2018, pp. 10–13. [Online]. Available: http://doi.acm.org/10.1145/3193977.3193983

[23] B. Hardin and U. Kanewala, "Using semi-supervised learning for predicting metamorphic relations," in *Proceedings of the 3rd International Workshop on Metamorphic Testing*, ser. MET '18. New York, NY, USA: ACM, 2018, pp. 14–17. [Online]. Available: http://doi.acm.org/10.1145/3193977.3193985

[24] T. Jameel, M. Lin, and L. Chao, "Test oracles based on metamorphic relations for image processing applications," in *2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, June 2015, pp. 1–6.

[25] L. Xu, D. Towey, A. P. French, S. Benford, Z. Q. Zhou, and T. Y. Chen, "Enhancing supervised classifications with metamorphic relations," in *2018 IEEE/ACM 3rd International Workshop on Metamorphic Testing (MET)*, May 2018, pp. 46–53.

[26] Z. Hui and S. Huang, "Experience report: How do metamorphic relations perform in geographic information systems testing," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, June 2016, pp. 598–599.

[27] M. Asrafi, H. Liu, and F. Kuo, "On testing effectiveness of metamorphic relations: A case study," in *2011 Fifth International Conference on Secure Software Integration and Reliability Improvement*, June 2011, pp. 147–156.

[28] X. Zhang, L. Yu, and X. Hou, "A method of metamorphic relations constructing for object-oriented software testing," in *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, May 2016, pp. 399–406.

[29] C. Sun, Y. Liu, Z. Wang, and W. K. Chan, "mt: A data mutation directed metamorphic relation acquisition methodology," in *2016 IEEE/ACM 1st International Workshop on Metamorphic Testing (MET)*, May 2016, pp. 12–18.