Leveraging Mutants for Automatic Prediction of Metamorphic Relations using Machine Learning

Aravind Nair
aanair@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Karl Meinke
karlm@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Sigrid Eldh
sigrid.eldh@ericsson.com
Ericsson AB
Stockholm, Sweden

ABSTRACT

An oracle is used in software testing to derive the verdict (pass/fail) for a test case. Lack of precise test oracles is one of the major problems in software testing which can hinder judgements about quality. Metamorphic testing is an emerging technique which solves both the oracle problem and the test case generation problem by testing special forms of software requirements known as metamorphic requirements. However, manually deriving the metamorphic requirements for a given program requires a high level of domain expertise, is labor intensive and error prone. As an alternative, we consider the problem of automatic detection of metamorphic requirements using machine learning (ML). For this problem we can apply graph kernels and support vector machines (SVM). A significant problem for any ML approach is to obtain a large labeled training set of data (in this case programs) that generalises well. The main contribution of this paper is a general method to generate large volumes of synthetic training data which can improve ML assisted detection of metamorphic requirements. For training data synthesis we adopt mutation testing techniques. This research is the first to explore the area of data augmentation techniques for ML-based analysis of software code. We also have the goal to enhance black-box testing using white-box methodologies. Our results show that the mutants incorporated into the source code corpus not only efficiently scale the dataset size, but they can also improve the accuracy of classification models.

CCS CONCEPTS

• **Computing methodologies → Support vector machines**; • **Software and its engineering → Software reliability**.

KEYWORDS

Data augmentation, Metamorphic Testing, Machine Learning, Source Code Analysis, Mutation Testing, Test Case Generation, Fault Identification

ACM Reference Format:
Aravind Nair, Karl Meinke, and Sigrid Eldh. 2019. Leveraging Mutants for Automatic Prediction of Metamorphic Relations using Machine Learning. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE '19), August 27, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3340482.3342741

1 INTRODUCTION

Oracles are used in software testing to derive the verdict (pass/fail) for a test case. A program is considered to be non-testable if either an oracle does not exist or the tester has to spend an enormous amount of time to determine its output [21]. Lack of adequate, precise and automated oracles is one of the major problems in software testing [2]. It reduces the possibility for fully automated software testing, as manual verdict construction cannot cope with large test suites. Metamorphic testing [5] is a requirement-based testing technique that has been proposed to alleviate the oracle problem. The metamorphic approach can even be used to automatically generate black-box test cases.

Despite its promise as a fully automated testing technique, manually identifying the metamorphic requirements satisfied by a given program p often requires a high level of domain expertise, is labor intensive and error prone. It is thus natural to try to automate discovery of the metamorphic requirements satisfied by p. Recently in [13], attention has been devoted to discovering metamorphic requirements through machine learning (ML). Techniques from ML can be used for pattern recognition on control flow graph structures and hence can implement static code analysis. In particular, we can try to learn the metamorphic requirements satisfied by p by comparing its control flow structure with the features of known programs p_i from a given ML training set $T = \{p_1, ..., p_n\}$. Metamorphic requirements are often generic and shared by many different programs. So there is some hope to find a sufficiently large value n that would avoid the dangers of poor generalisation and over-fitting from T. The main problem that will be addressed in this paper is: *how to extend a training set T by methods of data augmentation to avoid these dangers.*

By Rice's Theorem [8], the set $Prog(REQ)$ of all programs p satisfying a software requirement REQ is recursively undecidable unless REQ is trivial[1]. Therefore, it is not possible to learn this set exactly. Thus the VC dimension of learning $Prog(REQ)$ must be high, and only approximate learning is possible. Nevertheless, for specific and commonly occurring types of metamorphic requirements, ML appears to give good results, as first shown in [13]. These initial results are independently confirmed by our own work. Computability Theory suggests that large data sets are necessary to achieve high precision and high recall for learning. Herein lies one of the major challenges to leveraging ML methods for automated software

[1] REQ is trivial if it is true for all programs or false for all programs.

engineering. This difficulty is lack of access to appropriate large data sets of suitable program codes for training ML algorithms.

In this paper we consider the problem of using ML to automatically detect metamorphic requirements for testing. Specifically, we consider graph kernels and SVM learning. We can provide an ML algorithm with a training set of interesting and common codes, e.g. from a software repository. However, such training sets always have limited size and scope. Therefore, the resulting discriminator will be prone to over-fitting and poor generalisation. Our goal is to develop general solutions to training set limitations by constructing additional synthetic data using a variety of code transformation techniques. Our approach is similar to data augmentation techniques used successfully in multimedia analysis[22]. One potential source of transformations are program mutations arising from software testing theory [1]. We present results showing how test mutants can generate synthetic training data which can reduce the problem of over-fitting on small code collections.

2 BACKGROUND

2.1 Metamorphic Testing

Metamorphic testing [5] offers an elegant way to solve the oracle and test case generation problems for requirements testing. To apply this testing technique however, requires some knowledge of the requirements satisfied by the program to be tested (aka. the *system under test* SUT). These requirements are known as metamorphic relations (MRs). A metamorphic relation (MR) is a relation between two program executions that specifies how a change or transformation of the input determines a change in the output [7].

In metamorphic testing, given a program p, the results $p(t_1)$ and $p(t_2)$ of executing p on two test cases t_1, t_2 are compared post-execution. The idea is to evaluate the SUT behavior against a *black-box requirement REQ* such as an *equation* $exp_1(x, y) = exp_2(x, y)$ or an *inequality* $exp_1(x, y) < exp_2(x, y)$.

We say that the p *satisfies* the requirement REQ on tests t_1, t_2 if

$$exp_1(t_1, t_2)\ R\ exp_2(t_1, t_2),$$

otherwise[2] p *fails REQ* on t_1, t_2.

If we are interested to test how the output of p changes, as we vary its inputs, we may introduce a *transformation* between t_1 and t_2. e.g. $t_2 = T(t_1)$. If such a transformation T can be found for any x and y then we have a general *metamorphic requirement* or *metamorphic relation* (MR)

$$exp_1(x, T(x))\ R\ exp_2(x, T(x)).$$

As an example, in Section 4.3 we consider the *permutative* MR, which is formally a set of equations[3]:

$$p(x_1, ..., x_n) = p(T_\pi(x_1, ..., x_n)),$$

where for each permutation $\pi : \{1, ..., n\} \to \{1, ..., n\}$,

$$T_\pi(x_1, ..., x_n) = (x_{\pi(1)}, ..., x_{\pi(n)}).$$

For a metamorphic requirement, repeated applications of the transformation T on a *seed test case* t generate a family of test

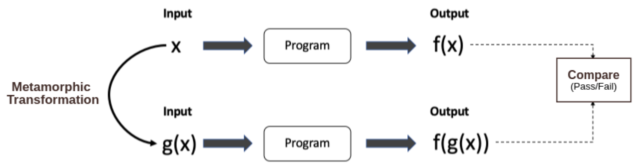


Figure 1: Overview of the metamorphic testing process

cases $t = T^0(t), T^1(t), ...T^n(t)$ that all satisfy the same metamorphic requirement $exp_1(T^n(t))\ R\ exp_2(T^{n+1}(t))$. Figure 1 depicts the metamorphic test process.

In general it is difficult to know what requirements (metamorphic or otherwise) should satisfied by a given program p. This makes requirements testing a difficult and often neglected activity. We are interested to automatically detect metamorphic requirements through ML. Such requirements can then be automatically verified by metamorphic testing. In this way we can easily and efficiently gain confidence about code quality through a highly automated test process.

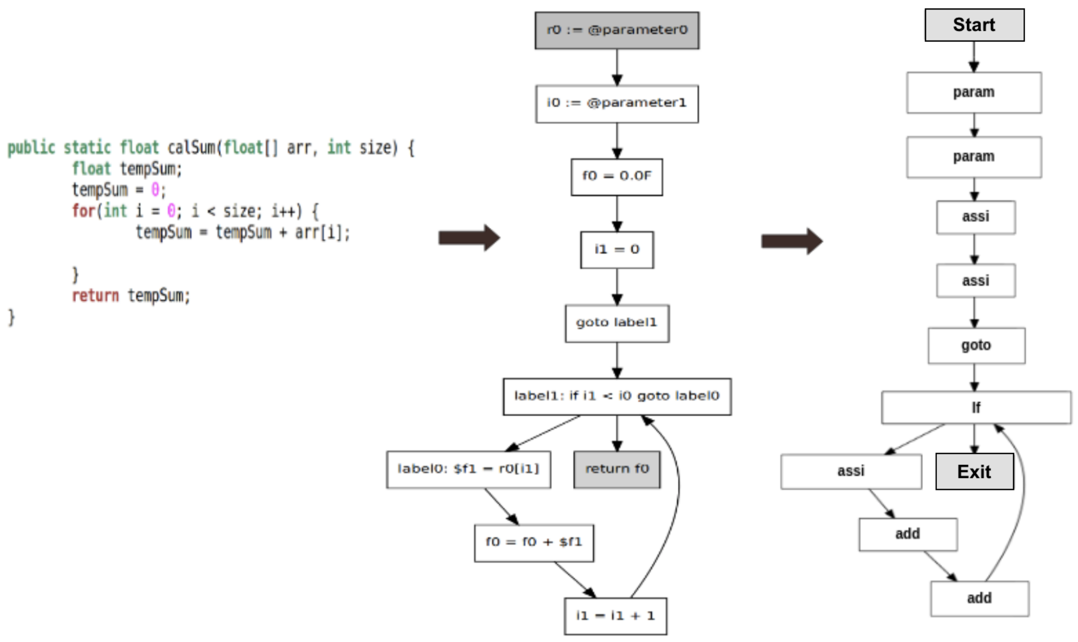
3 PROPOSED WORKFLOW

This section provides a detailed description of our methodology for identifying the metamorphic relations (*MRs*) satisfied by a program p using supervised machine learning techniques. Recent advances in machine learning like deep learning techniques have the capability to analyse and extract information from multimedia data such as text, image and videos. However, applying machine learning to program code is not straightforward due to its complex structure and the logic embedded in it. To analyse a program p, we transform it into a *control flow graph* (CFG). This represents the sequence of operations, cross references and the relations between variables in a graph-theoretic manner, while preserving syntactic and semantic properties of p. To generate a CFG from source code we used the open source Java tool Soot [9]. When a Java program p is passed to Soot, it internally converts the bytecode of p into the 3-address Jimple format [20]. This Jimple format is later used to construct the CFG at the atomic operation level.

The CFG thus generated is a directed graph $G_p = (V, E)$ representing p, where each vertex $v_x \in V$ denotes an atomic operation x in p. An edge $e \in E$ exists between two vertices (v_{x_1}, v_{x_2}) iff, x_2 is the immediately succeeding statement under execution after x_1. The labels on each vertex v_x are then processed to provide an abstract summary of the atomic operations in statement x. This abstraction simplifies the graph similarity measurement performed using graph kernels. Figure 2 shows the transformation of a Java method (calculating sum of elements in an array) to its corresponding *CFG* and the final *CFG* after label processing.

The next step involves analysing these *CFGs* using a graph kernel. We compute the similarity between two graphs (G_i, G_j) by passing them through the random walk graph kernel *RWK*. A random walk in the *CFG* corresponds to a sequential execution of atomic operations in the program p. Therefore the similarity score between two graphs (G_i, G_j) calculated by *RWK* is, the overall similarity between a random sample of sequential executions of operations in the graphs G_i and G_j. This similarity score is then

[2]Here, R is either the equality or inequality relation.
[3]All these equations must be satisfied for the MR to be satisfied.



```
public static float calSum(float[] arr, int size) {
        float tempSum;
        tempSum = 0;
        for(int i = 0; i < size; i++) {
                tempSum = tempSum + arr[i];

        }
        return tempSum;
}
```

Figure 2: Transformation process of a Java function to the processed Control Flow Graph

normalised using the following equation:

$$RWK(G_i, G_j) = \frac{RWK(G_i, G_j)}{\sqrt{RWK(G_i, G_i) * RWK(G_j, G_j)}} \quad (1)$$

Apart from finding the similarity, the kernel function also maps the inputs (CFGs in this case) implicitly into a feature space and finds an appropriate hypothesis for it. The feature map $\phi()$, for an input set $X = (x_1, x_2, ..., x_n)$ is defined by a kernel function k,

$$k(x_1, x_2) = \phi(x_1), \phi(x_2) \quad (2)$$

Let χ be the set of input data ($CFGs$) and by Y the set of labels (MRs satisfied by the program). Then, $\chi = \chi_1, ..., \chi_n$ denotes the training data and $Y = y_1, ..., y_n$, a set of its corresponding labels, jointly drawn independently from a probability distribution $P(\chi, y)$ on $\chi * Y$. The task now lies to create a function which could predict the label y_i for the input data χ_i. This is where the kernel function becomes relevant, as it enables computation of dot products in a feature space, for variables in the input space. This feature space could be utilized by other kernel methods like SVM for classification purposes.

The computed kernel RWK from the input $CFGs$ is then passed to a supervised machine learning classifier (SVM) to generate a predictive model capable of identifying the presence of certain MRs in an unseen program p'. Figure 3 represents the entire architectural workflow of our system.

4 IMPLEMENTATION RESULTS

This section describes in detail the dataset we used for training our model, the MRs we used in this study and discusses the results observed from our research.

4.1 Dataset Creation

Metamorphic testing is well suited for a large class of programs which have mathematical operations embedded into them. However, there is a lack of large and reliably labeled open-source program datasets to carry out ML research. Data augmentation techniques like, flipping, scaling and rotating, are used to scale datasets for image analysis. However, these image processing techniques cannot be directly applied to program code. Instead we must look for some analogous transformations which are applicable. In this study we try to answer two research questions:

- **RQ1:** Can mutants be effectively used as a data augmentation technique to scale the source code dataset?
- **RQ2:** Does a synthetically generated dataset help in creating a generalised model for metamorphic testing?

4.2 Mutants for Source Code Data Augmentation

Mutants are employed in mutation testing to characterise when a test suite is adequate to find faults in a program [10]. A mutant is created by purposefully embedding a small variant into a program p. If the newly created mutant p' always behaves in the same way as p, it is referred as an equivalent mutant or else as a non-equivalent mutant. For this study we manually generate both equivalent and non-equivalent mutants and use them as data augmentation techniques for scaling the data corpus. We succeeded to scale up our dataset 5-fold by adding 4 different sets (3 equivalent and 1 non-equivalent) of mutants.

- **Type 1 Mutant :** The first set of mutants generated are equivalent logical mutants - $EqLM1$. Every mutant $mu_i \in EqLM1$ is created by varying the logic of an atomic operation

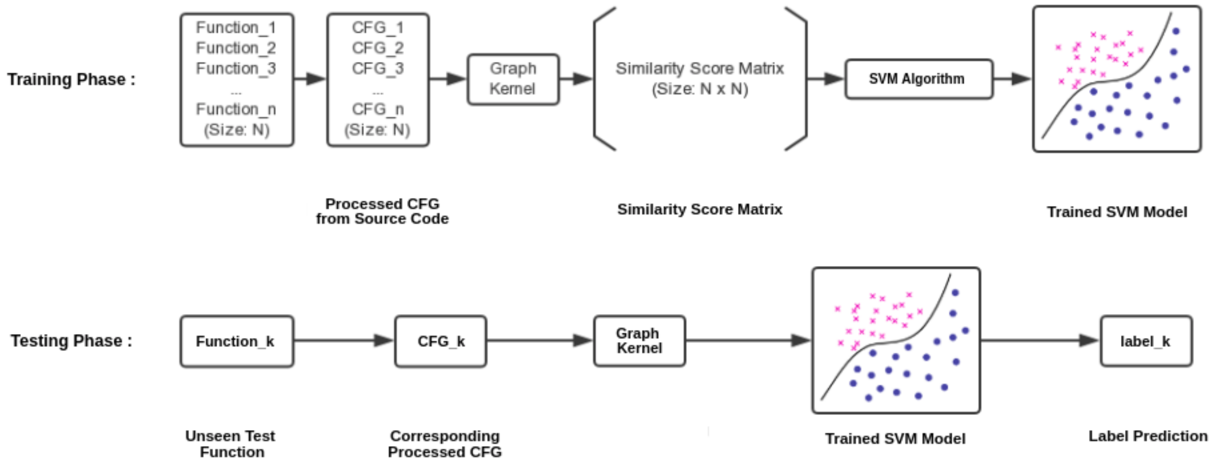


Figure 3: Architectural flow of the proposed approach

without changing the behaviour of the original program p. For example: an atomic operation: $(x = a + b)$ could be replaced by $(x = a - (-b))$

- **Type 2 Mutant :** The second set of mutants generated are non-equivalent logical mutants - $NEqLM2$. Every mutant $mu_i \in NEqLM2$ is created by varying the logic of an atomic operation which results in changing the behaviour of the original program p. For example: an atomic operation: $(x = a + b)$ could be replaced by $(x = a - b)$
- **Type 3 Mutant :** The third set of mutants generated are equivalent logical mutants - $EqLM3$. Every mutant $mu_i \in EqLM3$ is created by varying the logic of the loops present in a program p without changing the behaviour. For example: the loop logic:
$for(i = 0; i < size; i++)$ could be replaced by
$for(i = 0; i <= size - 1; i++)$
- **Type 4 Mutant :** The fourth and last set of mutants generated are equivalent dead code mutants - $DcEqM4$. Every mutant $mu_i \in DcEqM4$ is created by adding dead code assignments into the original program p. For example: initializing and assigning extra variables that have no impact on the behaviour of the program.

Figure 4 shows all the 4 types of mutants generated for the $calSum()$ function (calculating the sum of all elements in an array) described in Figure 2.

Equivalent mutants will have the same metamorphic properties as their parent program p, but a non-equivalent mutant might have different properties depending on how it has been changed. Thus, mutants could be used to generate distinct training datasets for both positive and negative classes. They also help to balance the number of samples in each class, giving the same distribution for positive and negative samples for both training and testing, thus avoiding the data bias.

4.3 Effectiveness of the Mutant Incorporated Dataset

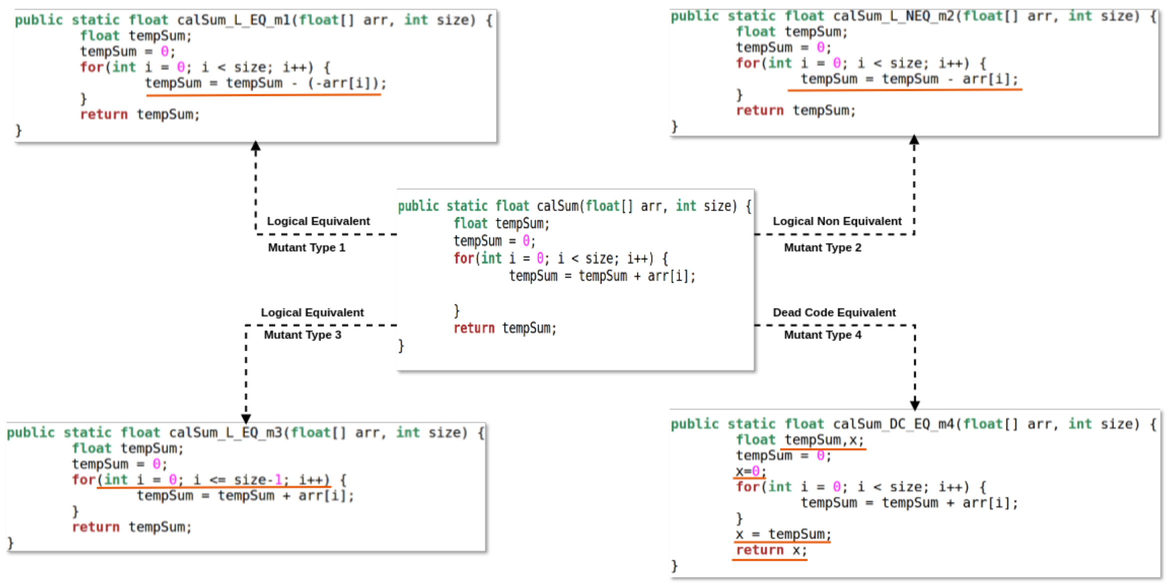
To begin with, we created a dataset[4] of 45 matrix calculation functions based on the research performed in [13] and hence, share its same scope, limitations and threats to validity. For this dataset we were interested to learn three metamorphic relations:

- **Permutative** $(P - MR)$: A change in the order of the input does not change the output of the program.
- **Additive** $(A - MR)$: By adding a positive scalar value to each unit in the input, the output of the program should increase or remain constant.
- **Inclusive** $(I - MR)$: By adding a new element in the input, the output of the program should increase or remain constant.

To calculate the effectiveness of the mutant augmented dataset, we generated both equivalent and non-equivalent mutants and labelled the MRs satisfied by them based on their behaviour. The full details of the dataset size and distribution is provided in Table 1. We then evaluated and compared the performance of the original dataset and the mutant augmented dataset, using the SVM algorithm. In this study, we used the SVM implementation of the scikit-learn library in python[18]. We used a stratified k-fold approach to evaluate our experiments. The stratified k-fold cross validator returns the training and test sets preserving the percentage of samples belonging to each class. This step is crucial to reduce the class imbalance in the dataset. To understand and evaluate the quality of the output provided by the classifier models we study the Receiver Operating Characteristic (ROC) metric [3]. ROC is the ratio of the True Positive Rate (TPR) against the False Positive Rate (FPR) and is considered as a measure of how well a trained model is capable of distinguishing between classes in the dataset.

$$TPR = \frac{TruePositive}{TruePositive + FalseNegative} \quad (3)$$

[4]https://github.com/aravi11/data-augmented-metamorphic-testing



```
public static float calSum(float[] arr, int size) {
        float tempSum;
        tempSum = 0;
        for(int i = 0; i < size; i++) {
                tempSum = tempSum + arr[i];

        }
        return tempSum;
}
```

```
public static float calSum_L_EQ_m1(float[] arr, int size) {
        float tempSum;
        tempSum = 0;
        for(int i = 0; i < size; i++) {
                tempSum = tempSum - (-arr[i]);
        }
        return tempSum;
}
```

```
public static float calSum_L_NEQ_m2(float[] arr, int size) {
        float tempSum;
        tempSum = 0;
        for(int i = 0; i < size; i++) {
                tempSum = tempSum - arr[i];
        }
        return tempSum;
}
```

```
public static float calSum_L_EQ_m3(float[] arr, int size) {
        float tempSum;
        tempSum = 0;
        for(int i = 0; i <= size-1; i++) {
                tempSum = tempSum + arr[i];
        }
        return tempSum;
}
```

```
public static float calSum_DC_EQ_m4(float[] arr, int size) {
        float tempSum,x;
        tempSum = 0;
        x=0;
        for(int i = 0; i < size; i++) {
                tempSum = tempSum + arr[i];
        }
        x = tempSum;
        return x;
}
```

Figure 4: Mutants generated for a function written to calculate the sum of all elements in an array

Table 1: Classwise dataset details before and after adding mutants

Metamorphic Relation	Original Dataset			After adding all Mutants		
	Total	Positive	Negative	Total	Positive	Negative
Permutative	**45**	19	26	**225**	101	124
Additive	**40**	23	17	**200**	96	104
Inclusive	**30**	19	11	**150**	76	74

$$FPR = \frac{FalsePositive}{FalsePositive + FalseNegative} \tag{4}$$

We evaluated the ROC performance of the three MRs for equivalent and non-equivalent mutant datasets separately. Figure 5 shows the ROC values obtained by the three MRs for equivalent datasets. The results show some increase in the *ROC* of the models for the permutative and inclusive MRs, but a small decrease for the additive MR. Higher values of *ROC* indicate the models are more capable of distinguishing between classes, i.e. whether a function satisfies a particular MR property or not. Thus it can be easily inferred that the corresponding mutant dataset helps to create a better model for detecting the MRs satisfied by a function.

Figure 6 shows the ROC values obtained for non-equivalent mutants only. The graphs show a sharp dip in the *ROC* for all the three MR models after adding non-equivalent mutants to the original dataset. This could be due to the fact that programs having similar CFGs will satisfy the same metamorphic relation [13]. Non-equivalent mutants generate similar CFGs but these may not satisfy the same MRs as their parent functions. This is not the case for equivalent mutants, which always satisfy the same MRs as their parents. We also, conjecture that the learning problem has become

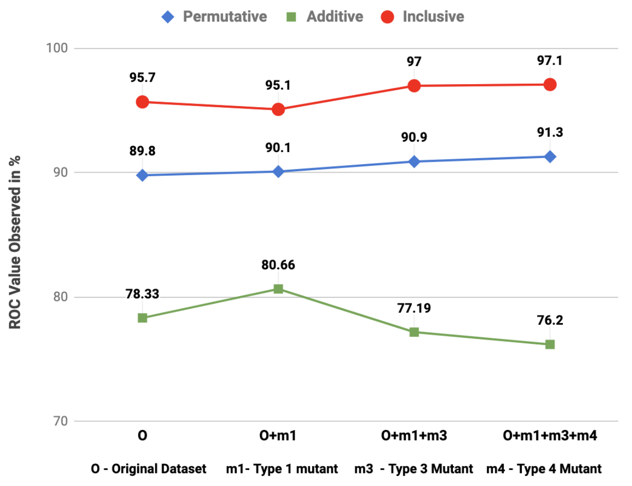


Figure 5: ROC Values observed for each MR for different equivalent mutant datasets

much harder, since the mutation distance[5] between the original

[5]We define the mutation distance between programs p and p' to be the minimum number of mutations needed to derive p' from p.

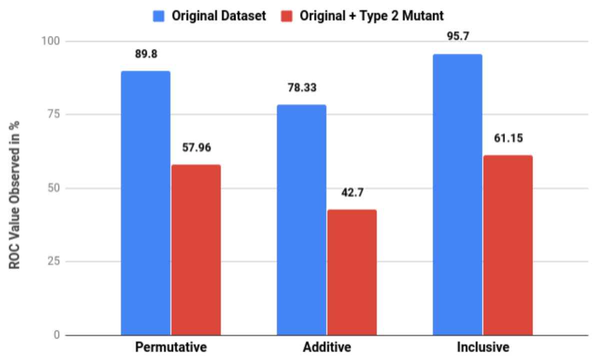


Figure 6: ROC Values observed for each MR for non-equivalent mutant datasets

positive examples is large, while the mutation distance of the new negative mutants to the original codes is quite small. This seems to impose a significant bias in the dataset. Thus adding non-equivalent mutants into the dataset decrements the efficiency of the models.

Our study shows a contrast in the role of equivalent and non-equivalent mutants in data augmentation. Equivalent mutants (often termed zombie mutants) obstruct the accurate measurement of test suite efficiency in software testing [15]. However for ML they turn out to be beneficial for generating new synthetic data. For non-equivalent mutants the benefit lies in the opposite direction.

5 RELATED WORK

Metamorphic testing (MT) was introduced as a novel technique to automate test case generation in [5]. MT has been proven effective to detect defects and has been used in different domains, including [4, 6, 12, 16]. The effectiveness of MT depends on the metamorphic relations satisfied and the quality of initial seed test cases. In [17], Murphy et. al. introduced a set of properties to define metamorphic relationships for testing machine learning applications. Kanewala et. al. conducted research to predict the metamorphic relations satisfied by a program automatically using both supervised and unsupervised machine learning techniques [11, 13, 14, 19]. However, in all these works, the dataset used has no augmentation, which leads to overfitted models having high variance. In our approach we have tried to solve this overfitting problem by scaling the dataset with data augmentation techniques.

6 CONCLUSION AND FUTURE WORK

Lack of adequate training data is a serious obstacle to using ML for many software engineering tasks. In this work, we have explored and compared equivalent and non-equivalent mutants as data augmentation techniques for enlarging a supervised training set. Our study has shown the effectiveness of equivalent mutants as a valid data augmentation technique to improve the detection rate for metamorphic relations satisfied by a program.

In future work we intend to extend this study by considering significantly larger data sets and more MRs. The effects of equivalent and non-equivalent mutants needs further study. Also we need to generate better labels to capture the semantic information in each program, using the appropriate kernel.

ACKNOWLEDGMENTS

The authors gratefully acknowledge financial support from the ITEA3 TESTOMAT Project 16032 and Ericsson AB.

REFERENCES

[1] Paul Ammann and Jeff Offutt. 2008. *Introduction to software testing.* Cambridge University Press.
[2] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.
[3] Andrew P Bradley. 1997. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern recognition* 30, 7 (1997), 1145–1159.
[4] Joshua Brown, Zhi Quan Zhou, and Yang-Wai Chow. 2018. Metamorphic testing of navigation software: A pilot study with Google Maps. In *Proceedings of the 51st Hawaii International Conference on System Sciences.*
[5] Tsong Y Chen, Shing C Cheung, and Siu Ming Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases.* Technical Report. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong.
[6] Tsong Yueh Chen, Fei-Ching Kuo, Wenjuan Ma, Willy Susilo, Dave Towey, Jeffrey Voas, and Zhi Quan Zhou. 2016. Metamorphic testing for cybersecurity. *Computer* 49, 6 (2016), 48–55.
[7] Tsong Yueh Chen, Tsun Him Tse, and Z Quan Zhou. 2003. Fault-based testing without the need of oracles. *Information and Software Technology* 45, 1 (2003), 1–9.
[8] N.J. Cutland. 1980. *Computability: an Introduction to Recursive Function Theory.* Cambridge University Press, Cambridge.
[9] Arni Einarsson and Janus Dam Nielsen. 2008. A SurvivorâĂŹs guide to Java program analysis with soot. *BRICS, Department of Computer Science, University of Aarhus, Denmark* (2008), 17.
[10] Robert Geist, A. Jefferson Offutt, and Frederick C Harris. 1992. Estimation and enhancement of real-time software reliability through mutation analysis. *IEEE Trans. Comput.* 41, 5 (1992), 550–558.
[11] Bonnie Hardin and Upulee Kanewala. 2018. Using semi-supervised learning for predicting metamorphic relations. In *Proceedings of the 3rd International Workshop on Metamorphic Testing.* ACM, 14–17.
[12] Darryl C Jarman, Zhi Quan Zhou, and Tsong Yueh Chen. 2017. Metamorphic testing for Adobe data analytics software. In *Proceedings of the 2nd International Workshop on Metamorphic Testing.* IEEE Press, 21–27.
[13] Upulee Kanewala and James M Bieman. 2013. Using machine learning techniques to detect metamorphic relations for programs without test oracles. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE).* IEEE, 1–10.
[14] Upulee Kanewala, James M Bieman, and Asa Ben-Hur. 2016. Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels. *Software testing, verification and reliability* 26, 3 (2016), 245–269.
[15] Bob Kurtz, Paul Ammann, Jeff Offutt, and Mariet Kurtz. 2016. Are we there yet? How redundant and equivalent mutants affect determination of test completeness. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* IEEE, 142–151.
[16] Mikael Lindvall, Dharmalingam Ganesan, Ragnar Árdal, and Robert E Wiegand. 2015. Metamorphic model-based testing applied on NASA DAT: an experience report. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2.* IEEE Press, 129–138.
[17] Christian Murphy, Gail E Kaiser, and Lifeng Hu. 2008. Properties of machine learning applications for use in metamorphic testing. (2008).
[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
[19] Karishma Rahman and Upulee Kanewala. 2018. Predicting metamorphic relations for matrix calculation programs. In *Proceedings of the 3rd International Workshop on Metamorphic Testing.* ACM, 10–13.
[20] Raja Vallee-Rai and Laurie J Hendren. 1998. Jimple: Simplifying Java bytecode for analyses and transformations. (1998).
[21] Elaine J Weyuker. 1982. On testing non-testable programs. *Comput. J.* 25, 4 (1982), 465–470.
[22] Sebastien C Wong, Adam Gatt, Victor Stamatescu, and Mark D McDonnell. 2016. Understanding data augmentation for classification: when to warp?. In *2016 international conference on digital image computing: techniques and applications (DICTA).* IEEE, 1–6.