# IMPROVING THE CONFIDENCE OF MACHINE LEARNING MODELS THROUGH IMPROVED SOFTWARE TESTING APPROACHES

by

Faqeer ur Rehman

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

Doctor of Philosophy

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

Dec 2022

# DEDICATION

I dedicate this dissertation to my mother and father, my wife and son, and to my siblings.

# ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Clemente Izurieta for his continuous support, motivation, and nice guidance that helped me in achieving this grand milestone. I also appreciate his efforts in providing his valuable feedback for this dissertation report. Thanks to Dr. John Paxton, Dr. Mike Wittie, and Dr. Travis Peters for being my valuable committee members and their continuous support throughout the program.

Among my lab fellows and friends, I feel lucky to have Derek Reimanis and Madhu Srinivasan in my circle, always available to guide me and helping me to stay focused, deterministic and motivated.

I would like to thank Dr. John Paxton for providing me the TA ship opportunity throughout the program, Dr. Craig Olive for awarding me the Ph.D. dissertation completion award, and Donna Negaard for the best guidance and support.

I would like to extend my gratitude to the MSU for providing me with the amazing educational opportunity that allowed me to fulfill my Ph.D. dream from one of the prestigious institutes in the country. Last but not the least, special thanks to the Gianforte School of Computing, and the Graduate School for providing me all the resources and help I ever needed, resulting in completion of this work.

TABLE OF CONTENTS

TABLE OF CONTENTS – CONTINUED

TABLE OF CONTENTS – CONTINUED

TABLE OF CONTENTS – CONTINUED

# LIST OF TABLES

LIST OF TABLES – CONTINUED

Table Page

# LIST OF FIGURES

# ABSTRACT

Machine learning is gaining popularity in transforming and improving a number of different domains e.g., self-driving cars, natural language processing, healthcare, manufacturing, retail, banking, and cybersecurity. However, knowing the fact that machine learning algorithms are computationally complex, it becomes a challenging task to verify their correctness when either the oracle is not available or is available but too expensive to apply.

Software Engineering for Machine Learning (SE4ML) is an emerging research area that focuses on applying the SE best practices and methods for better development, testing, operation, and maintenance of ML models. The focus of this work is on the *testing* aspect of ML applications by adapting the traditional software testing approaches for improving the confidence in them.

First, a statistical metamorphic testing technique is proposed to test Neural Network (NN)-based classifiers in a non-deterministic environment. Furthermore, an MRs minimization algorithm is proposed for the program under test; thus, saving computational costs and organizational testing resources.

Second, a Metamorphic Relation (MR) is proposed to address a data generation/labeling problem; that is, enhancing the test inputs effectiveness by extending the prioritized test set with new tests without incurring additional labeling costs. Further, the prioritized test inputs are leveraged to propose a statistical hypothesis testing (for detection) and machine learning-based approach (for prediction) of faulty behavior in two other machine learning classifiers i.e., NN-based Intrusion Detection Systems.

Finally, to test unsupervised ML models, the metamorphic testing approach is utilized to make some insightful contributions that include: i) proposing a broader set of 22 MRs for assessing the behavior of clustering algorithms under test, ii) providing a detailed analysis/reasoning to show how the proposed MRs can be used to target both the verification and validation aspects of testing the programs under investigation, and iii) showing that verification of MR using multiple criteria is more beneficial than relying on using just a single criterion (i.e., clusters assigned).

Thus, the work presented here results in providing a significant contribution to address the gaps found in the field, which enhances the body of knowledge in the emergent SE4ML field.

# 1

# INTRODUCTION

In such a tech-driven age, Machine learning (ML) has become an integral part of a broad range of domains like finance, marketing, transportation, machine translation, object detection, cybersecurity, and self-driving cars. Most of the time a software engineer focuses more on exploring and implementing different approaches to develop accurate models but much less on ensuring their quality and correctness. Thung et al. [73] conducted a study in which they reported that 22.6% of faults they found in ML applications are due to the incorrect implementation that caused them to produce unexpected and incorrect results. A small bug in the system may have disastrous consequences and can pose serious threats to both property and human lives. Some of the recent failures we have seen include, an incident on 14 Feb 2016, when a Google self-driving car crashed in an attempt to avoid sandbags [90], in May 2016, a Tesla Model S crashed when it did not treat the trailer as an obstacle [40], in March 2018, an Uber self-driving car hit and killed a woman at night in Arizona [52], and in 2017, a Palestinian man was arrested when he posted 'good morning' on Facebook that was wrongly translated as 'attack them' [28]. These incidents demand that we seriously think and realize how important it is to test the correctness and robustness of ML applications from multiple perspectives before moving them to production environments. Although this is a very challenging task, it is an equally essential problem for researchers and practitioners to address.

It is important to highlight that in general, the research community is focusing more on the development of high-performance ML models (which is critical) but much less on performing their quality assurance (which is also very important but unfortunately much ignored). It can be argued that ML engineers already use different performance evaluation

metrics (i.e., accuracy, precision, recall, F1-score, etc.) during the development of ML models to perform some testing activities. However, these evaluation metrics require a test oracle in the form of labelled data (which is not always available) and are not meant to test ML-based models for finding bugs in them, instead, they are used to evaluate which ML algorithm is best suited for the underlying data/problem. Furthermore, models with low scores in performance metrics are often representative of ML problems related to the availability or feeding of insufficient data. However, if the problem is not related to the insufficiency of data, and instead, there is an implementation fault in the ML-based classifier then we need to understand the causes for the low performance in the algorithm. Collecting more images, labeling them (a resource-intensive task), and feeding them to the same buggy algorithm will be of no advantage to further improving the model's performance unless we focus on new verification techniques. Ultimately, the ML engineer will start looking for alternative algorithms, wrongly assuming that the current algorithm used is not appropriate for addressing the underlying problem. Therefore, it is essential to have a testing component as a critical part of the ML pipeline.

It is important to know that the nature of ML software is different from traditional software. This is the reason that in comparison to traditional software, testing ML models bring its own challenges. First, when testing traditional software, the focus of testing is usually the source code. However, when testing ML-based models, besides testing the source code, the addition of data adds an extra layer of complexity. Second, traditional software is manually programmed (by developers) to perform the desired functionality. The code written is usually fixed and the output generated by the program under test is based on a predefined set of rules. However, the decision logic derived in ML applications is not explicitly hard-coded, instead, the logic surfaces from the data used to train these models. Third, the low accuracy of ML model can be attributed to the composite effect of the data, program (code written by the programmer), and the underlying framework/library

(e.g., Weka, Pytorch, TensorFlow). Any of these components may contain bugs, so, it is important to verify that each of them is correct and meets the desired expectations. Last but not the least, ML applications may have to be verified for a large set of input scenarios. As an example, features like date of birth, distance, speed, and road conditions (in an image) may contain a large range of valid values. As a result, it becomes difficult to verify their correctness for all possible scenarios that ultimately leads to the exhaustive testing problem and places these systems into the category of non-testable programs, suffering from the oracle problem [80]. An oracle is a mechanism where a program is verified by comparing its output with the expected outcome. To test such complex systems, either the oracle is unavailable or it is too expensive to apply. As an example, suppose that there is a ML model for classifying executable files, either as malicious or benign. To verify the model output, a security professional would have to first execute all those programs in a sandbox environment and then use the obtained results to verify the model outputs. It is a time-consuming and resource-intensive task to first execute and then compare the results manually, especially, when there is a large number of programs to verify.

Recently, we have seen tremendous advances in machine learning due to the availability of high computing power and better algorithms but relative to Software Engineering it is still a less mature field. In this context, Software Engineering for Machine Learning (SE4ML) is an emerging research area that focuses on applying SE best practices and methods for better development, testing, operation, and maintenance of ML-based systems [12, 38, 45, 46]. Our focus in this work is on the *testing* aspect of ML models that show traditional software testing approaches can be adapted to perform better quality assurance of these models. Although, some of the existing literature focuses on testing ML applications using traditional software testing techniques i.e., Metamorphic testing, Differential testing, and Combinatorial testing, these traditional software testing techniques have their own limitations when it comes to their direct applicability when testing ML models (some of the challenges are discussed above).

This motivated us to explore the gaps in the existing literature and utilize the SE approaches (from a quality assurance perspective) to not only better test such computationally complex ML-based systems for enhancing their quality but also to raise our trust in them. The work conducted seeks to further the understanding of the applicability of SE testing techniques for the quality assurance of ML-based models (supervised and unsupervised ML models).

The ML space is a broad domain, covering a large number of ML algorithms that can broadly be classified into supervised learning, unsupervised learning, and reinforcement learning algorithms. In this work, we focus only on the quality assurance of supervised and unsupervised learning algorithms because these types are the most common in use. Yet another challenge is that the corpus of both supervised and unsupervised learning contains a plethora of algorithms, so testing all possible algorithms is beyond the scope of this research. Therefore, in order to make a meaningful contribution, I have narrowed the problem space to i) focus on supervised learning, where I propose testing approaches to target NN based models (i.e., Fully connected NN and CNN based classifiers), and ii) focus on unsupervised learning, where I identify the gaps in the literature for multiple types of algorithms i.e., partitioning-based, hierarchical-based, and density-based clustering, and address them using the metamorphic testing approach.

The rest of the dissertation is organized as follows. The chapter titled 'Background & Related Work' discusses the necessary background knowledge related to software testing, mutation testing, machine learning, supervised and unsupervised ML algorithms, oracle problem and metamorphic testing, along with related research work done in the space of testing supervised and unsupervised ML models. Next, in the chapter titled 'Research Objectives', the motivators for this dissertation are discussed and the Goal Question Metric (GQM) is presented to frame the research work. In the chapter titled 'Statistical Metamorphic Testing of Neural Network Based Intrusion Detection Systems', a statistical metamorphic testing technique is proposed to test Neural Network (NN)-based classifiers in

a non-deterministic environment. In the chapter titled 'Testing Deep Learning Systems: A Statistical Metamorphic Approach', the proposed statistical metamorphic testing technique is further validated by testing a different type of deep learning model (i.e., CNN based pneumonia detection classifier) in the healthcare space. A Metamorphic Relations (MRs) minimization algorithm is also proposed that helps in saving organizational resources and performing testing with fewer MRs (especially in a regression testing environment) without compromising the overall fault detection effectiveness of the proposed approach. In the chapter titled 'A Hybridized Approach for Testing Neural Network Based Intrusion Detection Systems', an MR is proposed to address a data generation/labeling problem; that is, enhancing the test inputs effectiveness by extending the prioritized test set with new tests without incurring additional labeling costs. Further, the prioritized test inputs are leveraged to propose a statistical hypothesis testing (for detection) and machine learning-based approach (for prediction) of faulty behavior in two other machine learning classifiers i.e., NN-based Intrusion Detection Systems. In the chapter titled 'MT4UML: Metamorphic Testing for Unsupervised Machine Learning', the metamorphic testing approach is utilized to make some insightful contributions that include: i) proposing a broader set of 22 MRs for assessing the behavior of clustering algorithms i.e., k-means and agglomerative clustering algorithms under test, ii) providing a detailed analysis/reasoning to show how the proposed MRs can be used to target both the verification and validation aspects of testing the programs under investigation, and iii) showing that verification of MRs using multiple criteria is more beneficial than relying on using just a single criterion (i.e., clusters assigned). In the chapter titled 'An Approach For Verifying And Validating Clustering Based Anomaly Detection Systems Using Metamorphic Testing', a diverse set of MRs are proposed to test the implementation of the award-winning density-based clustering algorithm (at the leading data mining conference, ACM SIGKDD [70]): Density-based Spatial Clustering of Applications with Noise (DBSCAN) from both the verification and validation perspective. Lastly, the

chapter titled ‘Conclusion and Future Work’, concludes the dissertation along with potential future work and direction.

# 7

# BACKGROUND & RELATED WORK

## Background

This section covers core concepts and the terminologies most frequently used in this work. It encompasses an introduction to software testing, machine learning, supervised and unsupervised classification fundamentals, fully connected deep neural network, convolutional neural networks, k-means (partitioning based), agglomerative (hierarchical based), and Density-based spatial clustering of applications with noise (a density based) clustering methods, the oracle problem, metamorphic testing, metamorphic relations, source test case, follow-up test case, and mutation testing.

### Software Testing

Software testing is considered one of the most important phases in the Software Development Life Cycle (SDLC). It is a set of activities exercised to check the intended functionality of the program under test. A program is said to be faulty if the output produced by a program is different from the expected one. Some of the important program characteristics that practitioners focus on during testing include security, correctness, robustness, and performance. There is a plethora of testing techniques available but the popular ones include differential testing, unit testing, integration testing, combinatorial testing, metamorphic testing and adversarial testing.

### Mutation Testing

Mutation testing is a fault-based testing technique that is not only used to check the effectiveness of test cases but is also used in simulating the real faults made by developers [13]. An artificial bug/mutant is injected into the source code of a program to check whether the test cases can kill the mutant or not. A mutant is said to be killed if the output of

the mutant program is different from the output of the original program. The effectiveness of a test case is determined based on the mutation score (number of killed mutants / total number of mutants). Mutation testing can also be thought of as a technique to test the adequacy of the data available for testing. If a mutant cannot be killed with the existing data set, then the data set needs to be extended.

Machine Learning

Machine learning can be broadly defined as the study of computational methods that learn from the data (by extracting the hidden patterns) and use this information for future predictions [47]. For example, given a set of text documents each labeled with a topic, a known machine learning problem is how to use and extract the knowledge from these documents to accurately predict the topic for yet unseen documents. Machine learning can be broadly categorized into supervised learning, unsupervised learning, and reinforcement learning. The following subsections provide a brief overview of supervised and unsupervised machine learning.

Supervised Machine Learning

Supervised learning is the task of learning the patterns from training data and mapping the input features to the output/class labels. This extracted knowledge is then used to predict the output for an instance for which the class label was previously unknown. In a supervised ML task, training data can be represented with two $k$ size vectors. The first vector represents the training samples $S =< s_0, s_1, ..., s_{k-1} >$ and the second vector represents the class labels $C =< c_0, c_1, ..., c_{k-1} >$. Each sample $s_i$ (where $i = 0, 1, 2, ..., k-1$) has $m$ attributes which are used by the classifier for learning purposes. Each label $c_i$ (of sample $s_i$) corresponds to an element taken from a finite set of class labels, i.e. $c_i \in L = \{l_0, l_1, ..., l_{n-1}\}$, where $n$ denotes the number of class labels [83].

Supervised ML algorithms work in two phases namely - the *training phase* and the *testing phase*. In the training phase, a model tries to analyze and learn how the attributes are related to the class label. In the next phase (known as the testing phase), the trained model is used to predict output for the test instances. The model is then evaluated based on the predictions made for the test data. Some of the widely used ML algorithms include K-Nearest Neighbors, Naive Bayes, Support Vector Machine, Random Forest, and Neural Networks.

Examples of supervised learning problems include text classification [8], image classification [19] and spam filtering [26].

Random Forest Random Forest [11] is a supervised machine learning ensemble technique that uses multiple decision trees to make more stable and accurate predictions. The development of each tree is based on a random selection of data with replacement (known as bagging) and feature randomness, which makes the forest of trees diverse and less correlated. This property protects us from the potential effects of individual errors in a single tree provided that the same decision mistake is not present in all trees. The class label assigned to the test instance is based on the *maximum voting* concept i.e., the class label for which the maximum number of trees voted.

Support Vector Machine (SVM) SVM [11] is a supervised ML algorithm that tries to find the best hyperplane in N-dimensional space. The best hyperplane is the one that has maximum margin and can best separate the class instances. The hyperplane is the decision boundary that represents a different class on either side of it. SVM supports linear and non-linear kernels to separate both types of linear and non-linearly separable data. In order to separate the non-linear data, SVM takes advantage of kernel tricks (e.g., RBF kernel) that first transforms the data to a higher-dimensional space and then tries to find the hyperplane that can best linearly separate the data.

k-NN The k-NN algorithm [11] works on the assumption that similar objects tend to stay closer to each other. For a given test instance, it finds the distance (using Euclidean distance or Manhattan distance) between the test instance and all the training set instances, and selects the $k$ closest neighbors. The class label being dominant in the filtered $k$ nearest neighbors is assigned to the test instance.

Fully Connected Neural Network (NN) As shown in Figure 2.1, a fully connected NN is comprised of a series of fully connected layers containing multiple neurons, where each neuron in a layer $l$ is connected with the neurons in the next layer *l+1*. A simple neural network is comprised of an input layer, a hidden layer, and an output layer. A neural network is said to be deep if it contains multiple hidden layers. A neuron in a neural network is a basic computing unit that receives an input, performs a dot product (of input and corresponding weight parameter), applies the activation function, and then forwards the result to the neurons connected in the next layer. Some of the common activation functions used in DNNs include sigmoid, ReLU, and Tanh [56]. The output layer has either the softmax (if the problem is a multi-class classification problem) or the sigmoid activation function (if the problem is a binary class classification problem). The output layer produces the probability vector and the class for which the probability score is higher is treated as a predicted output for the given test instance.

Each layer in a neural network extracts and learns the patterns in the training data and then uses this knowledge for future prediction. The neurons in each layer are connected with other neurons using a weight parameter that represents the strength of the connection. The weights are learned during the training process with an aim to minimize the loss function (e.g., cross-entropy used for the classification problem, and mean square error used for the regression problem). The most popular algorithm used to update the weights and train the neural network is the gradient descent with back propagation [64].

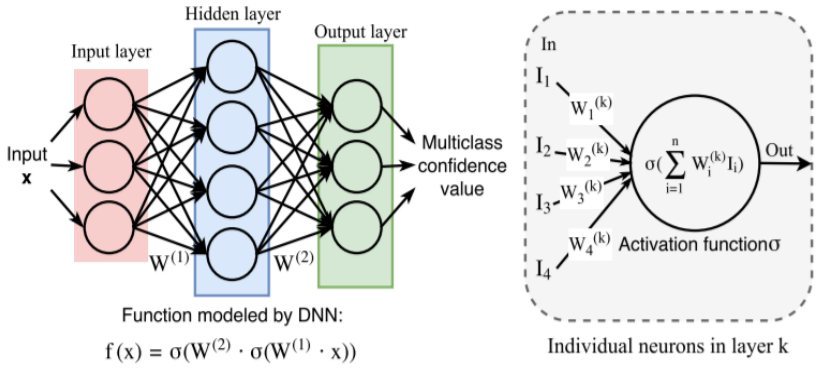


Figure 2.1: Neural Network Architecture [56]

Convolutional Neural Network (CNN) CNN is a type of feed-forward neural network which is widely used in solving image-related problems [51]. Its architecture consists of an input layer, hidden layers, and an output layer. The hidden layers may contain a series of convolution, pooling, and fully-connected layers, as shown in Figure 2.2. Each convolution layer contains several filters where each filter is used to extract specific patterns from the input data. As an example, given the images of animals, filters in the first layer may look for the low-level texture features e.g., edges, colors, etc., whereas, the filters in the next layer may extract the high-level features i.e., nose, eyes, etc. The filter is like a window that slides over the input image and calculates the inner product of weights parameters and the pixels of the input image. The activation function is then applied to the calculated dot product and forward this information as an input to the next layer. It is important to note that as the training progresses, the weights parameters and the image filters will be adjusted during the backpropagation process (using gradient-descent optimization algorithm). The pooling layer performs dimensionality reduction by sliding the window over the output received from the convolutional layer and obtain the pooled value (either maximum or average value) for that sliding window. Lastly, the multidimensional data is flattened using the fully connected layer before feeding it to the output layer of the network. The output layer will have either

the softmax (if the problem is a multi-class classification problem) or the sigmoid activation function (if the problem is a binary class classification problem). The output layer produces the probability vector and the class for which the probability score is higher is treated as a predicted output for the given test instance.

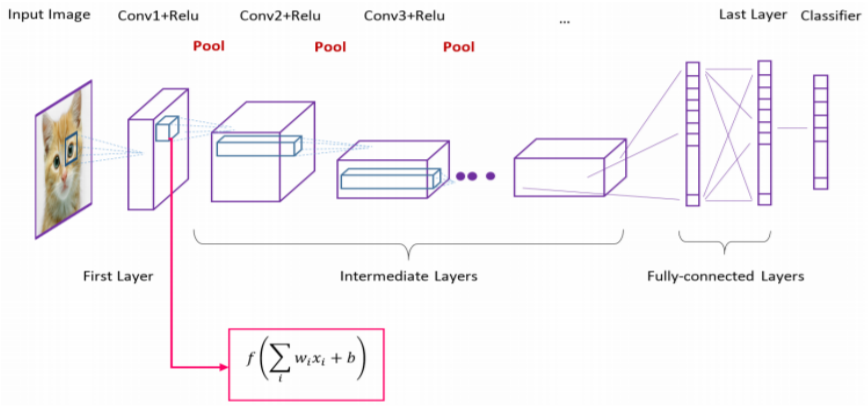


Figure 2.2: Convolutional Neural Network Architecture [51]

Unsupervised Machine Learning

In unsupervised machine learning, given the unlabeled data set $D$ having $x$ inputs with attributes $< A_o, A_1, A_2, ..., A_n >$, the goal is to partition the data set D into different groups/clusters such as $< C_0, C_1, C_2, ..., C_n >$, in such a way that the instances within the cluster $C_i$ are highly similar to each other, whereas, the instances between the clusters $C_i$ and $C_j$ are highly dissimilar. Some of the widely used clustering techniques include k-means clustering (a prototype-based approach), Agglomerative clustering (a hierarchy-based approach), and DBSCAN clustering (a density-based approach).

Examples of unsupervised learning problems are clustering DNA patterns [9], customer segmentation [29], and document clustering [63].

K-Means Clustering The k-means clustering algorithm belongs to the category of partitioning based clustering algorithms that works as follows [27]:

1. Select the initial 'k' centroids randomly.

2. Iterate over all the data points, calculate the distance between them and assign each of them to the nearest cluster centroid using the following formula.

$$C_i^{(t)} = \{x_p : ||x_p - c_i^{(t)}|| \leq ||x_p - c_j^{(t)}||\}$$

   where, $C_i^{(t)}$ represents the $i^{th}$ cluster (to which data instances are assigned) during the $t^{th}$ iteration, $c_i$ and $c_j$ denotes the centroids, and $x_p$ denotes the data instance that will be assigned to the cluster whose centroid has a minimum distance to it.

3. Recalculate the new centroids (it will be the mean of data points that are in the cluster $C_i$) as follows.

$$c_i^{(t+1)} = \frac{1}{|C_i^{(t)}|} \sum_{j=1}^{n} x_j$$

   where, $c_i^{(t+1)}$ represents the $i^{th}$ new centroid found, $x_j$ represents the $j^{th}$ instance (where j=1,2,...,n) belonging to the cluster $C_i$.

4. Repeat steps 2 and 3 until no change in centroids is found.

Agglomerative Clustering Agglomerative clustering belongs to the category of hierarchical-based clustering algorithms that follows a bottom-up approach to cluster the data. In this algorithm, each data point is initially considered as a single cluster and then the most similar pair of clusters are merged together forming a new cluster. The following are the main steps performed [84]:

1. Treat each data point as an individual cluster.

2. Evaluate and merge the two clusters that are most similar. The similarity between clusters is calculated using a distance measure (i.e., Euclidean distance) and a linkage criterion (i.e., single-linkage, complete-linkage, and average-linkage) and the two clusters which are more similar are merged together. As an example, for the average-linkage method, the following formula is used to calculate the distance between two clusters which is defined as the average distance between each of the data points (belonging to one cluster i.e., $C_i$) to every other data point (belonging to another cluster i.e., $C_j$).

$$d(C_i, C_j) = \frac{1}{|C_i||C_j|} \sum_{x_r \in C_i} \sum_{x_s \in C_j} d(x_r, x_s)$$

where, $d(C_i, C_j)$ represents the distance between cluster $C_i$ and cluster $C_j$, $x_r$ represents the $r^{th}$ data point (where r = 1,2,...,n) belonging to the cluster $C_i$, and $x_s$ represents the $s^{th}$ data point (where s = 1,2,...,n) belonging to the cluster $C_j$.

3. Repeat step 2 until all the similar data points are merged into a single cluster.

The result of agglomerative clustering can be visualized as a dendrogram, as shown in Figure 2.3. Although this algorithm does not take a number of clusters 'k' as input, the dendrogram should be broken at some point using some criteria (i.e., similarity level or a number of desired 'k' clusters) to obtain a disjoint set of clusters.

Density-based Spatial Clustering of Applications with Noise (DBSCAN) The DBSCAN algorithm proposed by Ester et al. [24] is designed for a scenario when distributions contain groups of arbitrary shapes. The algorithm has the ability to separate the noise and outliers; which are treated as anomalies. As shown in Figure 2.4, this algorithm finds the *core points* that have at least *min_samples* (minimum number of data points) in their neighborhood

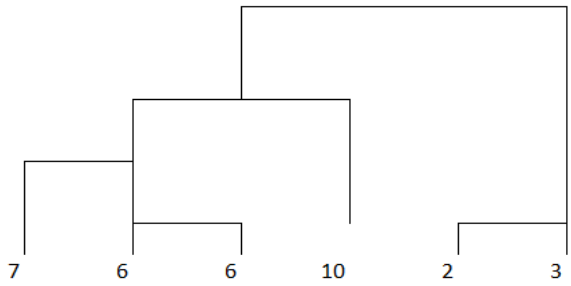


Figure 2.3: Agglomerative Clustering Example

within the radius *epsilon*. The *border points* are within the radius of some core point(s) but do not have the *min_samples* in their neighborhood to become a core point. The point is treated as a *noise point*, if there is no core point in its neighborhood within the radius *epsilon*.

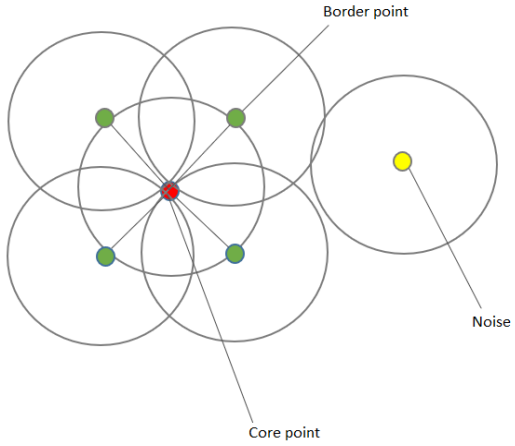


Figure 2.4: DBSCAN Algorithm with min_samples=5

The algorithm starts with an arbitrary core point $c_i$ and it grows the cluster by adding all its neighbor core points (that are within its radius *epsilon*). The cluster continues growing until all the core points and border points (reachable to the nearest core point) are assigned to a single cluster. The same process is repeated and an arbitrary new unvisited core point is selected to form the second cluster, then the third cluster, and so on. The points that are

not in the neighborhood of any *core points* are treated as *noise or anomalies*.

Oracle Problem

Oracle is a mechanism where a program is verified by comparing its output with the expected outcome. An example can be a *user authentication process* in which a valid users list is already available. A software tester knows beforehand that either the system should grant access to the given user or not (also known as the expected behavior of a system). However, in ML applications, the parameters are learned from the data and the results produced by an algorithm involve a complex logical and computational process. Apart from that, the input space for testing ML applications is so large that this brings difficulties in figuring out the expected output for the program under test and causes ML applications to suffer from a serious problem, known as *Oracle Problem* [80].

Metamorphic Testing

Metamorphic Testing (MT), a software testing method first proposed to alleviate the oracle problem in testing traditional software [20], has also been shown to be an effective approach in alleviating the oracle problem in testing ML applications [33, 54, 72, 79, 83]. The first step in MT is to identify the set of necessary properties/characteristics of a system known as metamorphic relations. Second, verify, if the behavior of a program (on both the source and follow-up test cases) is not in accordance with the identified metamorphic relation(s), this represents the potential presence of a fault in the system.

Metamorphic Relations (MRs) The Metamorphic Relation (MR) describes the relationship between the input and the related output that specifies how the output of a program should be changed when changing the input. A simple example can be a program calculating the standard deviation of a list of numbers. The following MRs can be formulated to check the correctness of this program.

- Shuffling the order of elements in a list should not change the final output.
- Multiplying each element in the list with -1 should not change the final output.

Source and Follow-up Test Cases Once the MRs have been identified, *source* and *follow-up* test cases are generated. The *source test case* (treated as original data) can either be domain-specific or can be randomly generated, whereas, the *follow-up* test case is generated by performing some valid transformation to the source test case (as specified in the related MR). As shown in Figure 2.5, both the *source* and *follow-up* test cases are then executed on the target program under test. If the results generated for the source and follow-up executions do not adhere to the relation specified in MR, this would indicate that there is some potential bug in the application.

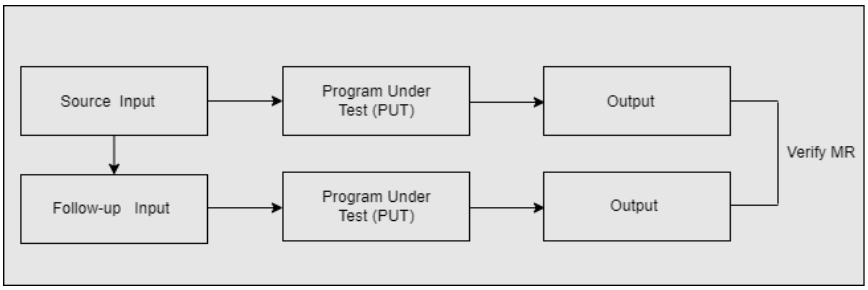


Figure 2.5: Metamorphic Testing Approach

## Related Work

A large number of contributions is available in the space of applying machine learning to software engineering problems (especially related to software testing) [50, 57, 67, 68, 69] but much less in the reverse direction i.e., software engineering for machine learning [12, 39, 66]. Characterizing the software engineering practices to machine learning is a very broad field, therefore, we decide to focus only on the quality assurance aspect of machine learning models i.e., identification, utilization, and adaptation of traditional software testing techniques in

machine learning space. We intend to explore the effective software testing techniques, their applicability in the machine learning space, challenges faced, and proposing potential solutions.

Most of the time software engineers focus more on the development of accurate models and rely on using an accuracy measure to check the performance and appropriateness of the proposed ML model for the underlying problem. Zheng et al. [42] has shown that using an accuracy measure may sometimes be misleading and may not be a good measure to use for checking the correctness of the program under test. The authors used the MT technique and proposed several MRs to test the neural network-based classifier. To show the effectiveness of the proposed MRs, they took advantage of mutation testing. The results obtained show that the proposed MT-based approach is able to kill the high accuracy producing mutants, knowing the fact that they represent the faulty implementations of the program under test. However, one of the limitations of the proposed approach is that the software tester needs to fix the random seeds (because of the stochastic nature in the training of neural networks) for getting deterministic results for both the source and follow-up executions. This can be very challenging, especially in an environment when the application under test is very complex and is using a large number of third-party libraries and some of them do not provide any option at all for fixing their random seeds.

In the context of white-box testing methods, the implementation details of a program are available to the software tester which can be used to test the internal implementation of a program under test. Pei et al. [56] propose *DeepXplore*, a white-box testing framework that uncovers thousands of erroneous behaviors in 5 Deep Neural Networks (DNNs). The following are the main contributions of this paper:

- First, a white box testing metric known as neuron coverage (number of neurons activated) is introduced. It is used to measure the extent to which a Deep Learning (DL) application's logic is exercised by the test inputs. A neuron is said to be

covered/activated if its value is above a certain set threshold.

- Second, the authors have demonstrated that finding the behavioral differences between multiple similar systems and obtaining a high neuron coverage can be formulated as a joint optimization problem. A gradient-based approach is presented to solve this joint optimization problem.

- Third, a white-box testing framework known as *DeepXplore* is proposed which implements the above ideas and uncovers thousands of erroneous corner case behaviors in 15 state-of-the-art DL models.

- Fourth, the authors showed that if the training data is augmented with the difference-inducing inputs (inputs that triggered erroneous behaviors in DL models), the classification accuracy of corresponding DL models can improve by up to 3%.

The proposed approach shows some promising results in uncovering the faults in the programs under test but one of the limitations of *DeepXplore* is that it strongly relies on the differential testing technique, which requires the availability of at least two similar implementations of a system. An error is reported, if the output produced by at least one of the DNNs is different from the others. In the real world, it is either hard to get similar multiple implementations or too costly to implement multiple copies of the same system. It is also possible that the developers may make similar types of coding mistakes in all those multiple copies [37]. It will resultantly cause all the DNNs to produce the same (but incorrect) results, thus failing to detect erroneous behaviors in the programs under test. In the presence of these limitations, it may be difficult to apply the proposed approach to real-world DNNs, especially when we do not have multiple implementations available.

In the space of testing supervised ML and DL-based applications, *DeepTest* [74] proposed a greedy search approach that uses multiple transformations (i.e., brightness,

contrast, fog effect, rotation etc.) to create realistic images that can be used to test self driving cars more effectively. The authors evaluated the proposed approach on testing CNNs and RNNs using the Udacity self-driving car challenge dataset [4] and found more than 1000 inconsistent behaviours. Zhou et al. [88] proposed *DeepBillboard*, a testing approach that adds perturbations to generate real world billboards for misleading the steering decisions in autonomous vehicles. The authors applied the *DeepBillboard* on testing 4 CNNs (used in autonomous driving testing [44, 56, 74, 87]) and uncovered numerous erroneous behaviours in them. Zhou et al. [89] reported serious defects in the Uber system 8 days before when on March 18, 2018, a fatal accident took place that killed the pedestrian at night in Tempe, AZ. In their proposed work, a combination of testing techniques (Fuzz testing and Metamorphic testing) are used to test LiDAR (light detection and ranging), an obstacle perception module used in Uber, and revealed several previously unknown serious issues. Ding et al. [21] applied the MT approach to validate a deep learning framework that has been used for the classification of biomedical images. The proposed approach was shown to be effective in validating the framework that includes, i) the architecture of a convolutional neural network, ii) the execution environment Caffe, and iii) the data set comprised of cellular images. Dwarakanath et al. [23] work applies MT for finding the implementation bugs in SVM and DNN based image classifier. The authors identified 4 MRs for testing an SVM-based model (used for classification of hand-written digit images), whereas 4 MRs were proposed to test a DNN-based image classifier called a Residual Network (ResNet), used for addressing a multi-class classification problem. The results obtained show that, on average, their approach is able to uncover 71% of implementation faults. They proposed an idea of initializing the network weights with some constant values and use maximum standard deviation as a threshold to verify the MRs. The reason to initialize the network weights with constant values is to get the deterministic results for multiple runs, whereas, using maximum standard deviation as a threshold will help in checking whether the MR is

violated or not. However, the authors have mentioned that they were able to get deterministic results when the DNN under test was trained on CPUs but not on GPUs due to the inherent non-determinism introduced by NVidia CUDA libraries. Therefore, the proposed approach will not work (as intended) in a non-deterministic environment. To address this problem, the most recent work [60] [76] includes taking advantage of machine learning and statistical-based MT techniques to uncover the implementation faults in a DNN-based intrusion detection system and a cancer prediction system. Jarman et al., [32] applied MT on Adobe data analytics software, which is used to detect trends in marketing data. The proposed MRs are based on geometric transformations (affine transformation matrices) and are able to successfully reveal 3 previously unknown bugs in the application. It is important to note that the authors' work primarily focuses on testing Adobe's software, not the quality of data, which is consumed by Adobe software for identification of data trends. Auer et al., [14] come up with this novel idea of applying MT technique to address data-related issues in a big data domain. The authors suggested that just like the necessary attributes used for verifying the correctness of an algorithm, the data also inherits intrinsic data quality characteristics e.g. accuracy, completeness, consistency, credibility, and correctness, which can be used to define metamorphic data relations for verifying its quality. To show the applicability of the proposed approach, the authors tested an open-source big data application *DBpedia* (https://wiki.dbpedia.org/) and found 30 incorrect entries which violated the metamorphic data relation.

ML Frameworks such as WEKA [81] are frequently used by technical and non-technical users for addressing both types of supervised and unsupervised ML problems. WEKA is a popular open-source framework that provides a large number of features i.e., data loading and pre-processing, supervised and unsupervised algorithms and their performance evaluation, different features selection techniques, and libraries to help in data visualization related tasks. It allows non-technical ML enthusiasts/beginners to develop and test ML models

without writing a single line of code. Santos et al. [65] utilized MT to validate a k-NN based breast cancer classifier and showed the applicability of MRs in the healthcare domain. Xie et al. [83] proposed 6 types of MRs for testing a couple of well-known supervised ML algorithms in WEKA i.e., Naive Bayes and k-NN. The proposed MRs are not only able to uncover some of the implementation bugs (injected via mutation testing technique) in these two algorithms (targeting program verification) but some of them can also be used to serve validation purposes.

Similar to supervised algorithms, testing unsupervised algorithms also suffer from the oracle problem. The metamorphic testing technique is considered an effective approach in alleviating the oracle problem in testing unsupervised clustering algorithms [84, 85]. Yang et al. [85] proposed 7 MRs to test the k-Means algorithm that target the algorithm's correctness from a user perspective (validation) to check whether the user expectations from the algorithm are satisfied or not. Their results show that two of the MRs are violated but it does not necessarily mean that there is some implementation defect in the algorithm under test. Xie et al. [84] proposed 11 generic MRs that assess and validate the characteristics of different clustering algorithms from a user perspective. The authors conducted an experiment to test 6 clustering algorithms and compared them using the proposed MRs. It thus helps the end-users (not technically expert and do not possess a solid theoretical foundation of clustering algorithms) coming from diverse fields i.e, bioinformatics, finance, and electrical engineering to choose a specific type of algorithm from a large set of available algorithms that can best fit their needs. However, the following are the limitations that we have found for the proposed works:

- They lack targeting the verification aspect of testing the clustering algorithms under test.
- A low dimensional synthetic data is used which does not represent real world data.

- The proposed approaches are limited to testing clustering algorithms provided by the WEKA tool.

These limitations make it an interesting area for investigation and making new contributions. We therefore, address them by proposing the MT-based technique that targets testing of clustering algorithms from both the verification and validation perspective using real world multidimensional datasets.

# RESEARCH OBJECTIVES

## Motivation

Most of the time, a software engineer focuses more on the development of accurate ML models but much less on ensuring their quality (in terms of correctness and robustness), which is equally very important for enhancing trust in them. When the ML model produces an incorrect output(s) or shows low performance, it is often attributed to deficient training data and the developer is asked to retrain the model on more data. However, there could be implementation bugs in the model, in which case, getting more data to train the model will not help. Apart from that, the performance evaluation metrics used frequently by ML engineers (i.e., accuracy, precision, recall, and F1-score) are not meant to test ML models for finding bugs in them, instead, they are used to evaluate which algorithm is better suited (i.e., giving better performance) for the underlying problem/data.

ML models are now becoming an integral component of security and safety-critical applications i.e., healthcare and autonomous driving, therefore, it is very important to make sure that these systems are correct and are working as intended. Some recent incidents like [28, 40, 52, 90] caused serious harm to both the humans and property, which raised serious concerns over their reliability in a real environment. Thus, this demands that researchers propose effective testing strategies for both the verification and validation of such critical ML-based systems.

SE4ML is an emerging research area that focuses on applying the SE best practices and methods for better development, testing, operation, and maintenance of ML models. The focus of this work is on the *testing* aspect of ML applications by utilizing and adapting the traditional software testing approaches in performing better quality assurance of ML models. Among the available software testing techniques, we have found MT as an effective technique in alleviating the oracle problem in testing ML applications. However, the

following identified gaps in the literature (related to applicability of MT in testing ML applications) are interesting areas to explore and also the motivators for this work:

- The first motivator for this work is that the traditional MT technique is not directly applicable to test neural network-based models due to their stochastic nature in the predicted outputs (i.e., if we provide the same training data to retrain the neural network, it may produce slightly different results for the same test data). Although few researchers [23, 42] tried to address this problem by either initializing the network weights with some constant values or fixing the random seeds, their proposed approaches of applying MT in such an environment has their own limitations [60].

- The second motivator for this work is that in classification problems, the data labeling task is considered an expensive and resource-intensive task. In the existing literature, MT has only been used mainly for testing the ML applications but none of the work focuses on using MT for solving the data collection/labeling problem. We aim to use its effectiveness in enhancing the prioritized test input size without incurring additional labeling costs. It can thus save a large amount of organizational cost in dedicating the resources for labeling the data instances manually.

- The third motivator for this work is that the existing literature focuses more on leveraging ML approaches in the prediction of faulty behavior in traditional software [50, 57, 67, 68] but to the best of our knowledge, we are unable to find any work that harnesses ML-based approach in the prediction of faulty behavior in ML-based applications. Therefore, we aim to explore this interesting area and making fruitful contributions.

- The fourth motivator for this work is that although the existing literature discusses taking advantage of using MT for testing unsupervised algorithms [84, 85], the work

proposed by them has its own limitations i.e, (i) instead of using a real data set, their work uses a randomly generated low dimensional (2D) data, (ii) their work focuses only on testing the algorithms provided by the WEKA tool, and (iii) their proposed MRs target only the validation aspect of the clustering algorithms.

In this study, we aim to explore and adapt the SE testing techniques to address the gaps (highlighted above) for producing better quality ML models.

## GQM

The GQM (Goal Question Metric) is a goal oriented approach [15] that we have used in order to lay out our research plan and guide the research. As shown in Figure 3.1, in the GQM approach, a set of **goal(s)** are identified, each of them is further refined using the **questions** to address the corresponding research goal, and then the **metrics** are outlined to answer the questions in a quantifiable manner. In our research work, we identify the following list of Research Goals (RGs), Research Questions (RQs), and the metrics used to answer the RQs in a quantifiable manner. Further, in Table 3.1, we mention the chapters addressing each of the raised RQs.

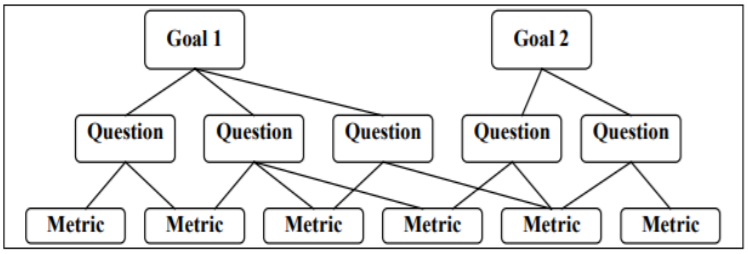


Figure 3.1: GQM Approach [18]

**RG1:** To detect implementation bugs in supervised ML models *for the purpose of* improving their quality *from the perspective of* software developers in *the context of* testing Fully Connected NN based models in a non-deterministic environment.

**RQ1.1:** How effective is the proposed approach in the identification of implementation bugs in NN-based classifiers under test?

**RQ1.2:** Do all MRs have the same defect detection ability?

**RG2:** To detect buggy behavior in supervised ML models *for the purpose of* improving their quality *from the perspective of* software engineers in *the context of* testing CNN-based models in a non-deterministic environment.

**RQ2.1:** How effective is the proposed statistical metamorphic testing approach in the identification of implementation bugs in CNN-based deep learning classifier under test?

**RQ2.2:** Do all MRs (verified through different statistical methods) show the same fault detection ability?

**RQ2.3:** Using the proposed approach, which MR(s) have high fault detection effectiveness, and which among them has the least?

**RQ2.4:** How can the proposed MRs (verified through the proposed statistical methods) be minimized for the CNN-based model under test?

**RG3:** To investigate the MT technique *for the purpose of* enhancing the test inputs size *from the perspective of* an organization in *the context of* reducing the labeling cost in testing NN-based classifiers.

**RQ3.1:** Can we enhance the test set size (using MT technique) without incurring any additional labeling cost?

**RQ3.2:** To what extent the given test set can be increased?

**RG4:** To detect and predict buggy behaviour in the new release of a classifier *for the purpose of* improving its quality *from the perspective of* a quality assurance team in *the context of* testing NN-based classifiers.

**Detection:**

**RQ4.1:** Is the proposed statistical hypothesis testing technique effective in detection of buggy behavior in the classifiers under test?

**Prediction:**

**RQ4.2:** Is the proposed ML-based approach effective and which ML model is more suitable for the problem under investigation?

**RQ4.3:** Does the addition of metadata features increase the performance of proposed ML models?

**RG5:** To investigate the MT technique for testing unsupervised algorithms *for the purpose of* improving their quality *from the perspective of* both the end user and a developer in *the context of* testing k-Means, DBSCAN, and Agglomerative clustering algorithms.

**RQ5.1:** How effective are the proposed MRs in testing the clustering algorithms under test?

**RQ5.2:** How to evaluate which algorithm is more stable in comparison to other(s) for performing clustering-related tasks (from both the end-user and developer perspective)?

**RQ5.3**: How effective is the proposed MT approach in testing the applications under test?

**RQ5.4:** Do all MRs have the same ability to detect the violations?

**Research Metrics (RM)**: The following metrics will be used to answer the research questions.

**RM1.1:** *Mutation Score* - Percentage of mutants killed. This metric will be used to answer RQ1 and RQ2.

**RM1.2:** *Yes/No* - This measure tries to answer RQs in Yes/No. It will be used to answer RQ1.2, RQ2.2, RQ3.1, RQ4.3, and RQ5.4.

**RM1.3:** *Test Set Size* - A measure to show that to what extent the test set size has been increased. This metric will be used to answer RQ3.2.

**RM1.4:** *Number of classes (or percentage) for which a buggy behavior is detected* - A count of the number of classes (or percentage) for which the program under test shows a buggy behavior. This metric will be used to answer RQ4.1.

**RM1.5:** *Accuracy, Precision, Recall, and F1* - These performance measures will be used to answer RQ4.2 and RQ4.3.

**RM1.6:** *Violation Rate* - Percentage of test instances for which the model shows inconsistent behaviour. This metric will be used to answer RQ5.

**RM1.7:** *Number of violated MRs* - A count of the number of MRs that are violated by the program under test. This metric will be used to answer RQ5.

Table 3.1: Chapters Addressing Research Questions (RQs)

| **RQ#** | **Paper/Chapter Title** |
|---|---|
| RQ1.1 | Statistical Metamorphic Testing of Neural Network Based Intrusion Detection Systems |
| RQ1.2 | Statistical Metamorphic Testing of Neural Network Based Intrusion Detection Systems |
| RQ2.1 | Testing Deep Learning Systems: A Statistical Metamorphic Approach |
| RQ2.2 | Testing Deep Learning Systems: A Statistical Metamorphic Approach |
| RQ2.3 | Testing Deep Learning Systems: A Statistical Metamorphic Approach |
| RQ2.4 | Testing Deep Learning Systems: A Statistical Metamorphic Approach |
| RQ3.1 | A Hybridized Approach for Testing Neural Network Based Intrusion Detection Systems |
| RQ3.2 | A Hybridized Approach for Testing Neural Network Based Intrusion Detection Systems |

| | |
|---|---|
| RQ4.1 | A Hybridized Approach for Testing Neural Network Based Intrusion Detection Systems |
| RQ4.2 | A Hybridized Approach for Testing Neural Network Based Intrusion Detection Systems |
| RQ4.3 | A Hybridized Approach for Testing Neural Network Based Intrusion Detection Systems |
| RQ5.1 | MT4UML: Metamorphic Testing for Unsupervised Machine Learning |
| RQ5.2 | MT4UML: Metamorphic Testing for Unsupervised Machine Learning |
| RQ5.3 | An Approach For Verifying And Validating Clustering Based Anomaly Detection Systems Using Metamorphic Testing |
| RQ5.4 | An Approach For Verifying And Validating Clustering Based Anomaly Detection Systems Using Metamorphic Testing |

# STATISTICAL METAMORPHIC TESTING OF NEURAL NETWORK BASED INTRUSION DETECTION SYSTEMS

## Contribution of Authors and Co-Authors

Manuscript in Chapter titled 'Statistical Metamorphic Testing of Neural Network Based Intrusion Detection Systems'

Author: Faqeer ur Rehman

Contributions: Problem identification and proposing solution, running experiment, manuscript writing, creating tables and figures. Primary writer

Co-Author: Dr. Clemente Izurieta

Contributions: Contribution in manuscript editing/writing, provided feedback, guidance and advice.

Manuscript Information Page

Faqeer ur Rehman and Dr. Clemente Izurieta

IEEE International Conference on Cyber Security and Resilience (CSR)

Status of Manuscript:
___ Prepared for submission to a peer-reviewed journal
___ Officially submitted to a peer-reviewed journal
___ Accepted by a peer-reviewed journal
_×_ Published in a peer-reviewed journal

IEEE
06 September 2021
10.1109/CSR51186.2021.9527993

## Abstract

Testing computationally complex neural network-based applications (i.e. network intrusion detection systems) is a challenging task due to the absence of a test oracle. Metamorphic testing is a method to potentially address the oracle problem when the correctness of individual output is difficult to determine. However, due to the stochastic nature of these applications, multiple runs with the same input can produce slightly different results; thus rendering traditional metamorphic testing technique inadequate. To address this problem, this paper proposes a statistical metamorphic testing technique to test neural network based Network Intrusion Detection Systems (N-IDSs) in a non-deterministic environment. We also performed mutation analysis to show the effectiveness of the proposed approach. The results show that the proposed method has a strong defect detection capability and is able to kill 100% implementation bugs in two neural network-based N-IDSs, and 66.66% in a neural network-based cancer prediction system.


## Introduction

Information Technology (IT) practitioners grapple on a daily basis with how to maintain their networks secure from malicious adversaries. A large number of tools exist today that can help with identifying potential weaknesses and vulnerabilities regarding all types of Cybersecurity concerns. The ISO 25k standard [7] characteristics helps us partition such threats into different categories. However, operational solutions to theoretical characterizations are not contextual, and require significant manual efforts from practitioners to identify relevant attacks. To aid practitioners and prevent such disastrous threats proactively, one possible solution would be to choose and deploy an intelligent machine learning based Network Intrusion Detection System (N-IDS). These automated techniques act in context and remove significant manual efforts. A challenge faced by these systems,

however; is how can we trust and rely on the correctness of such computationally complex machine learning based N-IDSs?, especially, when the organization has purchased it from a new vendor or built on top of some open source libraries.

Machine learning (ML) is heavily used in solving real-world problems in many application domains like voice recognition [10], transportation [16], safety-critical applications (e.g., self-driving cars and self-flying drones) [30], machine translation [36], healthcare [53], finance [78], and as exemplified above, the Cybersecurity domain [25]. Normally, we focus more on the development of accurate models but much less on ensuring their quality. Thung et al. [73] showed in their study that 22.6% of faults in ML applications are due to incorrect implementation that caused them to produce inconsistent and unexpected results. A small bug in the system may lead to catastrophic failure which can result in both financial and human loss. For example, on 14 Feb 2016, a Google self-driving car crashed due to misjudgment and putting itself into the path of an oncoming bus, in an attempt to avoid sandbags [90]. In May 2016, a Tesla Model S crashed when the autonomous driving system hit the trailer and did not treat it as an obstacle [40]. In March 2018, an Uber self-driving car killed a woman in Arizona, when at night it failed to recognize a pedestrian on the road [52]. In the face of these disasters, ensuring the correctness of ML-based systems is very challenging but equally an essential problem to be addressed.

Testing ML-based applications seriously suffer from the *Oracle problem* due to the difficulty in assessing whether the generated output is correct [80]. An oracle is a mechanism where a program is verified by comparing its output with the expected outcome. To test such complex systems, either the oracle is unavailable or too expensive to apply. A major approach used to address the oracle problem in such non-testable programs is known as Metamorphic Testing (MT) [20]. MT has been proven to be an effective approach in alleviating the oracle problem in testing ML applications, for which the correctness of individual output is difficult to determine [83, 89]. The first step in metamorphic testing is to identify the set of necessary

properties/characteristics of a system known as Metamorphic Relations *(MRs)*. Each MR describes the relationship between the input and the related output that specifies how the output of the program should be changed when changing the input. A simple example of an MR for a program calculating the standard deviation of a list of numbers can be stated as *'shuffling the order of elements in a list should not change the final output'*. Once the MRs have been identified, *source* and *follow-up* test cases are generated. The *source test case* (treated as original data) can either be domain-specific or can be randomly generated, whereas the *follow-up* test case is generated by performing some valid transformation to the original data (as specified in the related MR). Both the *source* and *follow-up* test cases are then executed on the target program under test. If the results generated by the source and follow-up executions do not adhere to the relation specified in the MR, then this would indicate a potential bug in the system.

One of the challenges faced in applying MT to Neural Network (NN) based applications is their stochastic nature (due to random initialization of weights) where they produce slightly different outputs for multiple runs with the same inputs. In order to get consistent results for both the source and follow-up test cases, a couple of researchers have proposed the solution of fixing the random seeds [23, 42]. The first paper is focused on applying MT to uncover implementation bugs in a Deep Learning (DL) based image classifier [23]. The authors used maximum standard deviation $\sigma_{max}$ (based on variation in a loss on the test data) as a threshold to verify if the program under test adhered to MRs or not. In order to alleviate stochasticity, the authors fixed the random seed to get deterministic results (obtaining consistent $\sigma_{max}$) for the program under test. The authors highlighted that they were able to get deterministic results on CPUs but not on GPUs due to inherent non-determinism introduced by NVidia CUDA libraries. Hence, their approach was limited to work only on CPUs. The second paper deals with applying MT to test an Artificial Neural Network (ANN) based classifier, taken from Stanford's cs231n course [42]. In order to get

deterministic results and to verify the MRs using the equality operator, the authors initialized the weights of a classifier with fixed values. The problem with this approach is two-fold, (i) it will not work in an environment where either the weights of the ANN cannot be fixed or where getting non-deterministic results is unavoidable, and (ii) it may not be applicable in a scenario when a model needs to be trained on GPUs to accelerate the training time.

In order to address the above highlighted problems, this paper proposes a statistical-based MT technique to unveil implementation bugs in NN-based N-IDSs, especially, in an environment where neither the random seeds nor the weights can be fixed in order to get deterministic results. Apart from that, in real-world it may not be possible for a software tester to fix the random seeds explicitly in a project that has millions lines of code and is also using a large number of third-party libraries. Therefore, instead of relying on a single run, the proposed approach statically analyses the results over multiple iterations (each iteration denotes a trained NN classifier) because a correct NN classifier should converge to almost the same solution most of the time, if not always [23]. To show the applicability of the proposed approach, we have worked with three ML applications. Two applications belong to Cybersecurity space i.e. N-IDSs, whereas the third one is from the healthcare domain that classifies cancer types among patients.

The following are the main contributions of this paper [60]:

- Three metamorphic relations are proposed to uncover implementation bugs in ML-based applications (i.e., N-IDSs and Cancer prediction system).
- Four statistical measures are used that will allow software testers to verify the correctness of a program under test (especially in a non-deterministic environment) using a combination of statistical hypothesis testing and MT technique.
- Mutation testing is applied to show the effectiveness of the proposed MT-based approach. The results show that the proposed method is able to kill 100% implementation

bugs in the two N-IDSs, whereas 66.66% in the cancer prediction system.

## Related Work

Deploying applications that are not fully tested can have disastrous consequences in the real world. Zhou et al. [89] reported serious defects in the Uber system 8 days prior to when the autonomous Uber killed Elaine Herzberg (a pedestrian) on March 18, 2018, in Tempe, AZ. The authors applied Fuzz testing in combination with MT to test LiDAR (light detection and ranging), an obstacle perception module used in Uber, and revealed several previously unknown fatal errors. Xie et al. [83] applied MT (as a test oracle) to test a popular open-source ML tool, known as Weka. Weka provides a large number of algorithms for data-prepossessing, classification, clustering, prediction, feature selection, and visualization [81]. The proposed MRs are not only able to find implementation bugs in K-Nearest Neighbors and Naive Bayes classification algorithms (treated as a verification step) but are also helpful in serving as validation steps. Pei et al. [56] proposed DeepXplore, a white-box testing framework to address two main challenges faced in the automated testing of a large-scale DL system: (i) identification of erroneous behavior(s) in the system without a need to labeling each test instance manually, and (ii) generation of inputs that can exercise different parts of a DL system's logic (known as neuron coverage) to uncover hidden defects in it. The proposed approach has shown promising results in uncovering thousands of erroneous behaviors in 5 DNNs. However, one of the limitations of the proposed approach is its strong dependency on a differential testing technique, which requires at least two similar implementations of a system. Apart from that, the authors did not provide any details regarding how they verified that the generated images truly represent real-world scenarios.

Normally, practitioners rely on using accuracy measures to check the appropriateness of an algorithm for the underlying problem. Li et al. [42] showed that higher accuracy does not necessarily mean that the trained model is free of bugs. The authors applied the MT

technique to test NN classifiers. The proposed MRs first transform the original inputs to generate follow-up test cases and then check, whether the produced output (for both source execution and follow-up execution) adheres to the corresponding MRs or not. To check the effectiveness of the proposed MRs, mutation testing is applied, and artificial bugs are injected into the source code of a program under test. The authors highlighted a few mutants that produced the same high accuracy as that of the original program, knowing the fact that they represent the faulty implementations. The results obtained show that the proposed MT-based approach is more effective in detecting faults than using the accuracy measure. However, the proposed approach is only applicable in an environment where the software tester can fix the random seeds to get deterministic results, which is not always possible.

It is important to note that despite efforts to find relevant references in the literature that address how MT could be used to test N-IDSs, we were unsuccessful, and although beyond the scope of this paper, we believe this is an important gap in the body of knowledge that needs further investigation.

## Proposed Approach

In this section, we present the proposed statistical-based MT technique in detail. First, we provide three MRs which are used to find implementation bugs in the applications under test. Next, instead of checking the correctness of MRs over a single source and follow-up execution (not possible due to random initialization of weights causing the non-deterministic behavior), we obtain results over multiple iterations and analyze them statistically (using the proposed statistical measures) to verify whether the outputs adhere to the relation specified in the MRs or not. The violation of an MR will be an indication of a potential bug in the program under test. Lastly, a mutation testing technique is applied on the following N-IDSs to check the effectiveness of proposed MRs. In order to show the relevance of the approach in other domains (i.e., health care), we also include a DNN-based cancer prediction system.

The details for each application are presented in Table 4.1.

- **Application#1:** A shallow NN-based N-IDS used for the detection of malicious attacks in an OpenStack environment. It is a multi-class classification problem that classifies the network request among 3 class labels (*normal*, *attacker*, and *victim*).
- **Application#2:** A DNN-based N-IDS for intrusion detection in a network that targets a binary class classification problem (i.e., either the request is *attack* or *benign*).
- **Application#3:** A DNN-based cancer prediction system used to identify cancer types among patients. It is a multi-class classification problem that classifies the patient among one of 10 cancer types.

Table 4.1: Neural Network Model Architectures

| **Application** | **# Hidden Layers** | **Hidden Layer(s) Type** | **Output Layer** |
|---|---|---|---|
| Application#1 (ANN N-IDS) | 1 | Fully Connected + ReLU | Softmax |
| Application#2 (DNN N-IDS) | 3 | Fully Connected + sigmoid | Sigmoid |
| Application#3 (DNN Cancer identification) | 3 | Fully Connected + sigmoid | Softmax |

Metamorphic Relations (MRs)

We propose the following three MRs that are applicable to all three ML based applications under test:

MR-1:-Changing the order of features (of both training and test data) Let $X_{train}$ be the training data and $X_{test}$ be the test data. After training the neural network, let a specific test instance $x^i_{test}$ be classified as class *'a'*. MR-1 says that if we change the order of attributes in both the training and test data, the output for the test instance $x^i_{test}$ should remain same (i.e., class *'a'*).

MR-2:-Addition of uninformative attribute to both training and test data Let $X_{train}$ be the training data and $X_{test}$ be the test data. After training the neural network, let a specific test instance $x^i_{test}$ be classified as class *'a'*. MR-2 says that if we add an uninformative attribute (attribute having value 6) to all the instances of both the training and test data, the output for the test instance $x^i_{test}$ should remain same.

MR-3:-Shifting of both the training and test features Let $X_{train}$ be the training data and $X_{test}$ be the test data. After training the neural network, let a specific test instance $x^i_{test}$ be classified as class *'a'*. MR-3 says that if we shift the features in both the training and test data with some constant $c$, it will not change the existing relationship between the data points, so the output for the test instance $x^i_{test}$ should remain same.

Statistical Hypothesis Tests

In this section, four statistical measures are discussed that we used to perform statistical hypothesis testing and verification of a relation specified in MRs. For this purpose, we added custom code in the classifiers under test to store their predicted results in excel files. The generated excel files contain detailed information for each test instance e.g. expected class label, predicted class label, the probability distribution for multiple classes, the maximum probability for the predicted class label, etc. We then developed a C# utility to process those excel files and extract the needed information on which the proposed statistical measures are applied. If the results are found statistically significant e.g., if the probability (*p-value*)

is less than the set significance level ($\alpha$), we reject the *Null Hypothesis ($H_0$)*. Rejection of $H_0$ would suggest that the MR has been violated and that there is a potential bug in the system. The classifiers' original code, mutants generated, excel files (containing predictions) and their processed versions, the C# utility, and the results produced, all are open-sourced [1]. The statistical measures are as follows:

Maximum Voting To understand the proposed maximum voting concept, let $X_{sData}$ be the source data, and $X_{fData}$ be the follow-up data. We train the same classifier *'n'* times on the source data and *'m'* times on the follow-up data (where $n = m$). This results in $C_n$ trained classifiers on the source data and $C_m$ trained classifiers on the follow-up data. Those trained models ($C_n$ and $C_m$) are then used to predict the class label for the test instance $x^i_{test}$. The results obtained are accumulated and a frequency distribution table is prepared, as shown in Table 4.2.

For a given test instance $x^i_{test}$, the MR is said to be violated, if the maximum times of the class predicted for the source executions is different than the maximum times predicted for the follow-up executions.

Comparing Distributions Using Chi-square Test of Homogeneity & Fisher's Exact Test There may exist some scenarios where the *Maximum Voting* approach may not work. For example, in Table 4.2, it can be seen that for both the source and follow-up executions, the class which is predicted maximum times is *Class2*. So, the proposed measure would suggest that the MR is satisfied. However, one may argue that the difference between the *Class2* and *Class3* distribution (for follow-up executions) is not very high and that this difference could be treated as the identification of a potential bug in the system. This motivates us to analyze the distributions over multiple class labels for better identification of *true positive*

[1] `https://github.com/matifkhattak/StatisticalMT/tree/master`

alarms. For this purpose, we take advantage of the Chi-square test of homogeneity ($\tilde{\chi}^2$), which is used to compare two samples having unknown population distributions. Using the frequency table (as shown in Table 4.2), we formulate the underlying problem as one where we compare the source executions' distribution $S_i$ (treated as the expected distribution) with the follow-up executions' distribution $F_i$ (treated as the observed distribution).

$$\tilde{\chi}^2 = \sum_{i=1}^{n} \frac{(F_i - S_i)^2}{S_i}$$

However, one of the limitations of the Chi-square test is that it may produce inaccurate and unreliable results if any of the cell values is less than 5 [2]. To solve this problem for some of the distributions we obtained, we apply an alternative test known as Fisher's exact test, which works equally well for the distributions having small cell values [2]. Based on the results obtained, if the p-value is less than the set significance level, *null hypothesis* is rejected, which ultimately means that the MR is violated and there is some potential bug in the system.

The following are the proposed *null* and *alternative hypothesis* used for both the $\tilde{\chi}^2$ and Fisher's exact test.

- $H_0$*:* The distributions for both the source and follow-up executions are same.
- $H_a$*:* The distribution for the source executions is different from the follow-up executions.

Table 4.2: Frequency Distribution Table

| **Execution Type** | **Class1** | **Class2** | **Class3** |
|---|---|---|---|
| Source (n=30) | 3 | 25 | 2 |
| Follow-up (m=30) | 2 | 15 | 13 |

Comparing Distributions Using Two Sample t-Test & Permutation Test In an NN-based classifier, an activation function (e.g., sigmoid or softmax) is used in the output layer that generates a probability vector, providing the probability for each of the class labels. A class with the highest probability is treated as the predicted class label for the given test instance. We take advantage of using these probabilities and perform statistical analysis to check how close the probability distributions are for both the source and follow-up executions for any given instance.

We treat this problem as comparing the two sample means using the t-Test. First, the results in excel files are processed (using the C# program) and the class label predicted maximum times during the source executions is identified. For the given test instance $x^i_{test}$, the purpose is to first find the class label for which the model is more confident and then extracting the probability of that specific class for both the source and follow-up executions. This will result in the generation of two samples for each MR, one for the source and other for the follow-up executions, as shown in Table 4.3.

Table 4.3: Exemplary Probabilities

| **Source Executions** | **Follow-up Executions** |
|---|---|
| 0.63 | 0.61 |
| 0.64 | 0.60 |
| 0.61 | 0.62 |
| 0.61 | 0.65 |
| 0.60 | 0.59 |
| 0.61 | 0.60 |

Let $x = (x_1, x_2,...,x_n)$ and $y = (y_1, y_2,...,y_n)$ represent two samples, one corresponding to source executions and the other corresponding to follow-up executions. In order to conduct

the statistical t-Test, we need to find the mean $\bar{x_n} = \sum_{i=1}^{n} x_i$ and the variance $s^2_{x,n} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x_n})^2$ for both samples. The t-score is calculated as,

$$t = \frac{\bar{x_n} - \bar{y_n}}{\sqrt{\frac{s^2_{x,n}}{n} + \frac{s^2_{y,n}}{n}}}$$

After applying the t-Test, the p-value obtained is compared with the significance level. If it is less than the set significance level, $H_0$ will be rejected, which ultimately means that the MR is violated and that there is a potential bug in the system. The following are the *null* and *alternative hypothesis* to check whether the MRs have been satisfied or not.

- $H_0$*:* There is no difference in the sample means of both the source and follow-up executions.
- $H_a$*:* The sample mean of source executions is different from the sample mean of follow-up executions.

It is important to note that before applying the two-sample t-Test, we have analyzed the data to check whether the assumptions are fully satisfied. During the analysis (using a Q-Q plot), we found that for some of the MRs, the normality assumption is slightly violated.

However, the t-Test is robust against such violations and can still be applied [59]. To perform better analysis and support decision-making, we show our results using both the t-Test and the permutation test. The permutation test is a non-parametric test that does not rely on the normality assumption. For more details about the permutation test, we refer the interested readers to read Chapter 1 [59].

## Empirical Results

All three applications (N-IDSs and Cancer identification system) under test are built on Keras 2.3.1 and TensorFlow 2.0. We used the MutPy [3] tool to generate mutated versions

Table 4.4: Results for Application#1: Shallow Neural Network Based N-IDS

| **Significance Level ($\alpha$) = 0.05** | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Mutants** | **Maximum Voting** | | | **$\tilde{\chi}^2$ / Fisher Exact Test** | | | **t-Test** | | | **Permutation Test** | | | **Overall** |
| | MR-1 | MR-2 | MR-3 | MR-1 | MR-2 | MR-3 | MR-1 | MR-2 | MR-3 | MR-1 | MR-2 | MR-3 | |
| Mutant#1 | Y | | | Y | | | Y | | | Y | | | Y |
| Mutant#2 | | | | | Y | | | Y | | | Y | | Y |
| Mutant#3 | | Y | Y | | | | | | | | | | Y |
| **MS** | **33.33%** | **33.33%** | **33.33%** | **33.33%** | **33.33%** | **0%** | **33.33%** | **33.33%** | **0%** | **33.33%** | **33.33%** | **0%** | |
| | **66.66%** | | | **66.66%** | | | **66.66%** | | | **66.66%** | | | **100%** |
| **Significance Level ($\alpha$) = 0.1** | | | | | | | | | | | | | |
| **Mutants** | **Maximum Voting** | | | **$\tilde{\chi}^2$ / Fisher Exact Test** | | | **t-Test** | | | **Permutation Test** | | | **Overall** |
| | MR-1 | MR-2 | MR-3 | MR-1 | MR-2 | MR-3 | MR-1 | MR-2 | MR-3 | MR-1 | MR-2 | MR-3 | |
| Mutant#1 | Y | | | Y | | | Y | | | Y | | | Y |
| Mutant#2 | | | | Y | Y | | | Y | | Y | Y | | Y |
| Mutant#3 | | Y | Y | | Y | | | | | | | | Y |
| **MS** | **33.33%** | **33.33%** | **33.33%** | **66.66%** | **66.66%** | **0%** | **33.33%** | **33.33%** | **0%** | **66.66%** | **33.33%** | **0%** | |
| | **66.66%** | | | **100%** | | | **66.66%** | | | **66.66%** | | | **100%** |

Y denotes the mutant is killed, and MS represents the Mutation Score

Table 4.5: Results for Application#2: Deep Neural Network Based N-IDS

| **Significance Level ($\alpha$) = 0.05** | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Mutants** | **Maximum Voting** | | | **$\tilde{\chi}^2$ / Fisher Exact Test** | | | **t-Test** | | | **Permutation Test** | | | **Overall** |
| | MR-1 | MR-2 | MR-3 | MR-1 | MR-2 | MR-3 | MR-1 | MR-2 | MR-3 | MR-1 | MR-2 | MR-3 | |
| Mutant#1 | Y | | | Y | | | Y | | Y | Y | | Y | Y |
| Mutant#2 | | | | | | | Y | | | Y | | | Y |
| Mutant#3 | | Y | Y | | | | | | | | | | Y |
| **MS** | **33.33%** | **33.33%** | **33.33%** | **33.33%** | **0%** | **0%** | **66.66%** | **0%** | **33.33%** | **66.66%** | **0%** | **33.33%** | |
| | **66.66%** | | | **33.33%** | | | **66.66%** | | | **66.66%** | | | **100%** |
| **Significance Level ($\alpha$) = 0.1** | | | | | | | | | | | | | |
| **Mutants** | **Maximum Voting** | | | **$\tilde{\chi}^2$ / Fisher Exact Test** | | | **t-Test** | | | **Permutation Test** | | | **Overall** |
| | MR-1 | MR-2 | MR-3 | MR-1 | MR-2 | MR-3 | MR-1 | MR-2 | MR-3 | MR-1 | MR-2 | MR-3 | |
| Mutant#1 | Y | | | Y | | Y | Y | | Y | Y | | Y | Y |
| Mutant#2 | | | | | | | Y | | | Y | | | Y |
| Mutant#3 | | Y | Y | | Y | Y | | | | | | | Y |
| **MS** | **33.33%** | **33.33%** | **33.33%** | **33.33%** | **33.33%** | **66.66%** | **66.66%** | **0%** | **33.33%** | **66.66%** | **0%** | **33.33%** | |
| | **66.66%** | | | **66.66%** | | | **66.66%** | | | **66.66%** | | | **100%** |

Y denotes the mutant is killed, and MS represents the Mutation Score

Table 4.6: Results for Application#3: Deep Neural Network Based Cancer Prediction System

| **Significance Level ($\alpha$) = 0.05** | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Mutants** | **Maximum Voting** | | | **$\tilde{\chi}^2$ / Fisher Exact Test** | | | **t-Test** | | | **Permutation Test** | | | **Overall** |
| | MR-1 | MR-2 | MR-3 | MR-1 | MR-2 | MR-3 | MR-1 | MR-2 | MR-3 | MR-1 | MR-2 | MR-3 | |
| Mutant#1 | Y | Y | Y | Y | | | Y | | | Y | | | Y |
| Mutant#2 | | | | | | | | | | | | | |
| Mutant#3 | Y | | | Y | Y | | Y | Y | | Y | | | Y |
| **MS** | **66.66%** | **33.33%** | **33.33%** | **66.66%** | **33.33%** | **0%** | **66.66%** | **33.33%** | **0%** | **66.66%** | **0%** | **0%** | |
| | **66.66%** | | | **66.66%** | | | **66.66%** | | | **66.66%** | | | **66.66%** |
| **Significance Level ($\alpha$) = 0.1** | | | | | | | | | | | | | |
| **Mutants** | **Maximum Voting** | | | **$\tilde{\chi}^2$ / Fisher Exact Test** | | | **t-Test** | | | **Permutation Test** | | | **Overall** |
| | MR-1 | MR-2 | MR-3 | MR-1 | MR-2 | MR-3 | MR-1 | MR-2 | MR-3 | MR-1 | MR-2 | MR-3 | |
| Mutant#1 | Y | Y | Y | Y | | | Y | | | Y | | | Y |
| Mutant#2 | | | | | | | | | | | | | |
| Mutant#3 | Y | | | Y | Y | Y | Y | Y | Y | Y | | | Y |
| **MS** | **66.66%** | **33.33%** | **33.33%** | **66.66%** | **33.33%** | **33.33%** | **66.66%** | **33.33%** | **33.33%** | **66.66%** | **0%** | **0%** | |
| | **66.66%** | | | **66.66%** | | | **66.66%** | | | **66.66%** | | | **66.66%** |

Y denotes the mutant is killed, and MS represents the Mutation Score

of the classifiers under test. After excluding the mutants, which were either changing the architecture of the neural network or causing the program to crash, we selected 3 valid mutants for each application (details are available online [2]). One of the generated mutants is shown in Figure 4.1, which will cause the program to over-fit. A mutant is said to be killed if the results do not adhere to the relation specified in the MR. The effectiveness of MR is determined based on the mutation score (number of killed mutants / total number of mutants).

```
total_loss = loss_weight * output_loss

total_loss = loss_weight / output_loss
```

Figure 4.1: Original Code (top) and Mutant (below).

The main objective of this study is to test the NN based N-IDSs in a non-deterministic environment. To show whether the results are statistically significant, we analyzed the

[2] `https://github.com/matifkhattak/StatisticalMT/tree/master`

results over multiple iterations. The results for Application#1 (shown in Table 4.4) and Application#3 (shown in Table 4.6) are obtained based on 200 trained models for each MR (100 models trained on source data and 100 trained on follow-up data). However, due to time resource constraints, for Application#2, we obtained the results based on 60 trained models for each MR (30 models trained on source data and 30 trained on follow-up data), as shown in Table 4.5. The results presented show the effectiveness of each MR using the mutation score (as a %). We present the mutation scores at two levels, (i) of each individual MR, and (ii) of each statistical measure (with combined MRs).

It is important to note that all three applications under investigation are critical systems of high consequence, so a Type-II error has more severity because the acceptance of a *false Null hypothesis* (in case of using low $\alpha$ level) will suggest that there is no bug in the system when actually there is. To perform better analysis and support decision-making, we reported the results with both $\alpha = 0.05$ and $\alpha = 0.1$. Results presented in Table 4.4, 4.5, and 4.6 show that most of the time using the proposed statistical measures, the given MRs are able to kill at least 66% of the mutants. Further, if we have enough resources to apply all four statistical measures, the results (under column name *Overall*) show that the proposed approach is able to kill 100% of mutants in two N-IDSs (Application#1 and Application#2), and 66% of mutants in the cancer prediction system (Application#3), which shows its strong capability to detect defects. **Therefore, we find that the proposed statistical-based MT technique is effective in finding the implementation bugs in NN-based applications in a non-deterministic environment, which answers RQ1.1 of RG1**.

However, it can be seen that the proposed MRs have different fault detection capability for each application under test. For example, upon closer inspection of the results in Table 4.5, when statistically analyzed using the *t-Test*, we can see that MR-1 has a strong defect detection capability of 66.66%, whereas MR-2 has the lowest mutant killing rate of 0%. We can use the same knowledge to check the effectiveness of different MRs for different

applications, as shown in Table 4.4, 4.5, and 4.6. The results in Table 4.6 (for Application#3) show that mutant#2 is survived and none of the MRs are able to kill it (for both $\alpha = 0.05$ and $\alpha = 0.1$). Upon further analysis, we find that each time a DNN is trained, it predicts the same class label for all the test instances, which shows that there is some potential bug in the classifier under test. **Hence, we come to the conclusion that different MRs have different fault detection ability for the NN-based applications under test, which answers RQ1.2 of RG1**.

## Conclusion

We proposed a statistical-based Metamorphic Testing (MT) technique for testing a class of Machine Learning (ML) applications (i.e, network intrusion detection systems) that have stochastic behaviour in their results. Having this property, traditional MT approaches are not applicable because they rely on using an equality operator to verify the relation specified in Metamorphic Relations (MRs). We introduce three MRs for uncovering implementation bugs in the applications under test. Furthermore, we propose four statistical measures that allows us to statistically analyze the predicted outputs and to verify the relation specified in MRs. The effectiveness of our proposed method is show with the identification of bugs (injected through mutation testing technique) in three ML-based applications. This research is focused on showing the applicability of statistical-based MT techniques with sample MRs for neural network-based network intrusion detection systems. Furthermore, we also show the usefulness of the proposed approach in the healthcare domain (e.g., Cancer identification system). A more comprehensive study on formulating new MRs and evaluating their performance using the proposed approach is in progress.

# TESTING DEEP LEARNING SYSTEMS: A STATISTICAL METAMORPHIC APPROACH

## Contribution of Authors and Co-Authors

Manuscript in Chapter titled 'Testing Deep Learning Systems: A Statistical Metamorphic Approach'

Author: Faqeer ur Rehman

Contributions: Problem identification and proposing solution, conducting experiment, manuscript writing, creating tables and figures. Primary writer

Co-Author: Dr. Clemente Izurieta

Contributions: Contribution in manuscript editing/writing, provided feedback, guidance and advice.

Manuscript Information Page

Faqeer ur Rehman and Dr. Clemente Izurieta

IEEE Transactions on Software Engineering

Status of Manuscript:
___ Prepared for submission to a peer-reviewed journal
_×_ Officially submitted to a peer-reviewed journal
___ Accepted by a peer-reviewed journal
___ Published in a peer-reviewed journal

IEEE
17 September 2022

## Abstract

Machine learning technology spans many areas and today plays a significant role in addressing a wide range of problems in critical domains i.e., healthcare, autonomous driving, finance, manufacturing, cybersecurity, etc. Metamorphic Testing (MT) is considered a simpler but very powerful approach in testing such computationally complex systems for which either an oracle is not available or is available but difficult to apply. Traditional metamorphic testing techniques are not applicable for testing deep learning-based models (i.e., CNNs) that have a stochastic nature in their training (the stochastic nature (because of randomly initializing the network weights)). In this paper, we make an attempt to address this problem by proposing a statistical metamorphic testing technique that does not require software testers to worry about fixing the random seeds (for getting deterministic results) in order to verify the Metamorphic Relations (MRs). We propose 7 MRs in combination with different statistical methods to statistically verify whether the program under test adheres to the relation(s) specified in the MR(s) or not. We further use mutation testing techniques to show the usefulness of the proposed approach in the healthcare space and test an open-source CNN-based deep learning model (used for pneumonia detection among patients). The empirical results show that our proposed approach has been able to uncover 85.71% of the implementation faults in the Classifier Under Test (CUT). Furthermore, we also propose an MR minimization algorithm for the CUT; thus, saving computational costs and organizational testing resources.


## Introduction

According to the World Health Organization (WHO), in 2019 alone, pneumonia (a pulmonary infection illness) was responsible worldwide for almost 740180 (14%) of all deaths in children under the age of 5 [6]. It has affected children and families all around the globe,

but the worst among them are mostly from sub-Saharan Africa and South Asia.

The lungs of the human body are composed of sacs (also known as alveoli), which are filled with air when a healthy individual breathes. However, when suffering from pneumonia (caused by viruses, bacteria, or fungi), these sacs are filled with pus/fluid which limits oxygen intake, causes pain, and makes it hard for a person to breathe. Therefore, it is critical to detect and diagnose infections at the earliest possible time to help prevent them from becoming pathogenic. However, the difficulty of clinical detection remains a big challenge due to the potential variability of symptoms, and their resemblance to those observed in other types of illnesses such as influenza, colds, and asthma.

One of the promising diagnostic tools witnessed in medical science includes taking advantage of chest radiography but it remains a time-consuming and resource-intensive task for radiologists to manually examine and diagnose chest X-rays [41]. At the same time, manual approaches are error-prone and subjective. Therefore, this raises a need for an efficient method that can automatically detect the existence of pneumonia correctly.

Artificial Intelligence (AI) is playing a pivotal role in empowering and building solutions in critical domains, such as medical science, in which mostly, Deep Learning (DL) based models i.e., Convolutional Neural Networks (CNN), have become the state-of-the-art technique. Recent work has shown their usefulness/effectiveness in the detection of pneumonia using chest X-ray images [31, 49, 58, 75]. It is important to highlight that in the healthcare space (and in general), the research community is focusing more on the development of high-performance DL-based models (which is critical) but much less on performing their quality assurance (which is also very important but unfortunately much ignored). It can be argued that ML engineers already use different performance evaluation metrics (i.e., accuracy, precision, recall, F1-score, etc.) during the development of ML models to perform some testing activities. However, these evaluation metrics are not meant to test (i.e., verify and validate) ML-based models for finding implementation bugs in them, instead,

they are used to evaluate which ML algorithm is best suited for the underlying data/problem.

Furthermore, models with low scores in performance metrics are often representative of ML problems related to the availability or feeding of insufficient data. However, if the problem is not related to the insufficiency of data, and instead, there is an implementation fault in the DL-based classifier (in our case, used for detection of pneumonia in chest X-ray images) then we need to understand the causes for the low performance in the algorithm. Collecting more images, labeling them (a resource-intensive task), and feeding them to the same buggy algorithm will be of no advantage to further improving the model's performance unless we focus on new verification techniques. Ultimately, the ML engineer will start looking for alternative algorithms, wrongly assuming that the current algorithm used is not appropriate for addressing the underlying problem. Therefore, it is essential to have a testing component as a critical part of the ML pipeline.

Among the traditional software testing techniques, Metamorphic Testing (MT) [20] is considered a simpler but very powerful approach to alleviating the oracle problem when testing ML-based applications. Although MT techniques can be applied directly (without any modifications) to test traditional ML-based algorithms that have a deterministic nature in their predictions for multiple runs (i.e., SVM, Decision Trees, KNN, Logistic Regression, etc.), they are not applicable when testing DL-based classifiers (i.e., CNNs). Testing CNN-based DL models bring an added challenge (i.e., the stochastic nature of their training because of randomly initializing the network weights) making the application of traditional MT techniques infeasible. For example, a CNN-based model trained on the same training data (during the source execution) may produce slightly inconsistent/different outputs for the same test inputs (i.e., trained on the same data during the follow-up execution); thus, consistent results may not be feasible to verify the MRs. One possible approach to obtain deterministic results is to fix the random seeds but this may not work when, i) it is hard for the software tester to manually identify and fix the random seed for a large number of

libraries, ii) a third party library used in the project does not provide options to fix random seeds, and, iii) the underlying hardware architecture uses GPUs (that may induce non-determinism in calculations when leveraged by libraries i.e., Nvidia CUDA Libraries [23]). To address these challenges, our previous research work focused on adapting the traditional MT approach and proposed statistical MT techniques for testing non-deterministic neural network-based intrusion detection systems [60]. This research work is an extension of our previous work in order to exemplify its further applicability to a different domain and to present a new MR minimization algorithm. It is important to highlight that to the best of our knowledge, we have not found any research work that focuses on using MT for testing CNN-based image classifiers in a non-deterministic environment (especially in the healthcare domain), which is equally, a motivator for this work.

In this paper [77], we make the following contributions:

- A statistical MT technique to test DL-based image classifiers (i.e., a CNN-based model) that have a stochastic nature in their training.
- 7 MRs that both researchers and practitioners can leverage (in combination with statistical methods that we proposed in our previous work [60]) to test the correctness and robustness of CNN-based image classifiers in a stochastic environment.
- We show the applicability of the proposed approach in the healthcare space by testing an open-source CNN-based deep learning model which is used to detect pneumonia among patients.
- We use the mutation testing technique to show the effectiveness of the proposed MRs (i.e., their ability to detect implementation faults in the PUT). The results obtained show that using the proposed approach we are able to uncover 85.71% of the implementation faults in the PUT.

- In the context of statistical metamorphic testing approach, we take advantage of using multiple concepts i.e., mutation scores, the type of mutants killed by each MR, and the difference of sets (borrowed from set theory) to propose an MR minimization algorithm; thus, saving computational costs and organizational testing resources.

## Related Work

The traditional testing activities performed while developing ML models are cross-validation and evaluating the performance metrics (i.e., accuracy, precision, recall, F1-score, etc.). However, they are not aimed to test ML-based models for finding bugs in them, instead, they are used to evaluate which ML algorithm is best suited for the underlying data/problem. Imagine, there is an implementation fault in the underlying ML algorithm used for prediction. In that case, the aforementioned evaluation metrics will not provide any useful information about the existence of such fault. Li et al. [42] and ur Rehman et al. [76] utilized mutation testing techniques to generate mutated program versions (for NN-based classifiers) that had almost the same high accuracy as that produced by the original program (i.e., mutant free), knowing *a priori* that they represented the buggy versions of the PUT; thus, showing that i) using high accuracy as a testing criterion is not a reliable choice, and ii) a program producing high accuracy (used as a performance metric) is not an indication that the PUT is free from bugs. Furthermore, the work in [42] proposes a set of MRs to kill high accuracy producing mutants, whereas, the work in [76] used a statistical hypothesis testing technique for identification (of high accuracy producing mutants) and an ML-based approach for prediction in the next release of a Classifier Under Test (CUT).

The MT technique has shown its usefulness in alleviating the oracle problem in testing both supervised and unsupervised ML algorithms/applications. For testing unsupervised ML algorithms, Yang et al. [85] proposed 7 MRs to assess the behavior of the k-means clustering algorithm, provided by the WEKA tool [5], whereas, Xie et al., [84] proposed 11 MRs

targeting users' general expectations from six unsupervised ML algorithms. However, the aforementioned research only used simple synthetic 2D data and only targeted the validation aspects of testing clustering algorithms, which have been recently addressed in [61, 62]. In the latter, the authors proposed a diverse set of MRs targeting both the verification and validation aspects of testing clustering algorithms using multi-dimensional real-world data sets that fall into three distinct categories i.e., density-based, hierarchy-based, and prototype-based unsupervised ML algorithms.

In contrast to testing unsupervised ML applications, the MT technique has received more attention when testing supervised ML applications. Santos et al. [65] utilized MT to validate a k-NN based breast cancer classifier and showed the applicability of MRs (originally proposed by Xie et al. [83], to test k-NN and Naive Bayes algorithms (provided by the WEKA tool) in the healthcare domain. Dwarakanath et al. [23] proposed 4 MRs for testing an SVM-based model (used for classification of hand-written digit images), whereas 4 MRs were proposed to test a DNN-based image classifier called a Residual Network (ResNet), used for addressing a multi-class classification problem. The results obtained show that, on average, their approach is able to uncover 71% of implementation faults. However, the limitation of using the traditional MT technique is that it does not work when there is non-determinism in the outputs of a program (such as DNNs) for source and follow-up inputs. For obtaining deterministic results, the authors fixed the random seed for each execution to verify whether the relation provided in the MR is satisfied or not. Furthermore, Li et al., [42] handled the same stochastic behavior in testing the NN-based classifier by initializing the random weights with constant values, so that consistent results could be obtained (for both the source/original and the generated follow-up inputs) to verify the proposed MRs. However, this research [23, 42] leaves serious questions unaddressed, i.e., what if, i) it is not possible/feasible for a software tester to manually identify and fix the random seed for a large number of libraries integrated with the project, ii) a third party library does not provide

any options to fix random seeds, and, iii) the underlying hardware architecture uses GPUs (that may induce non-determinism in calculations when leveraged by libraries i.e., Nvidia CUDA Libraries [23]). To address these challenges when testing DNN-based image classifiers with their stochastic nature (which are equally the motivators for this work), we propose a statistical metamorphic testing technique that can be used for testing such computationally complex programs without worrying about either fixing the random seed or initializing the weights (of neural network) with constant/fixed values.

Approach To Identify Implementation Bugs in DNN-based Applications

In this section, we provide in-depth analysis of our proposed Statistical Metamorphic Testing (SMT) technique for testing an open source Deep Learning (DL) classifier (i.e., a CNN-based model), used for identification of pneumonia among patients using chest X-ray images. We propose a set of 7 MRs that have been used in combination with statistical methods (that we proposed in our previous work [60] for testing a different class of neural networks, known as, fully connected neural networks) for uncovering implementation bugs in a CNN-based image CUT. Among the contributions of this work, we validate our approach and show the applicability of the proposed statistical methods in a different domain (i.e., image classification in the healthcare space). Due to the stochastic nature of training CNNs (because of random initialization of weights), a CNN trained on source data may produce slightly inconsistent/different final outputs (for the same test inputs) when trained on follow-up data. For this reason, a single source and follow-up program execution cannot be used to verify an MR. Therefore, instead of comparing the outputs for a single source and follow-up execution, we statistically compare the results for multiple source and follow-up executions. As shown in Figure 5.1, we use the MR to generate the follow-up training/test data from the source training/test data. Then, using the source training data and the follow-up training data, we train the CUT multiple times and use all trained models to predict the outputs for

the same source and follow-up test data. The results obtained are statistically analyzed (with the help of proposed statistical methods) to verify the satisfiability of the relation captured in the MR. Empirical evidence from [23] shows that for different deep learning architectures, a correct NN classifier (ran multiple times) might have different converge points but those will be very close to each other and will not differ significantly in terms of overall loss. An MR is said to be violated if there is a significant difference in the outputs of multiple source and follow-up executions, which ultimately would be a sign of a potential bug in the CUT. To show the effectiveness of the proposed MRs, we further use the mutation testing technique for injecting artificial bugs in the CNN-based CUT, and then empirically evaluate the number of bugs uncovered by the proposed approach.

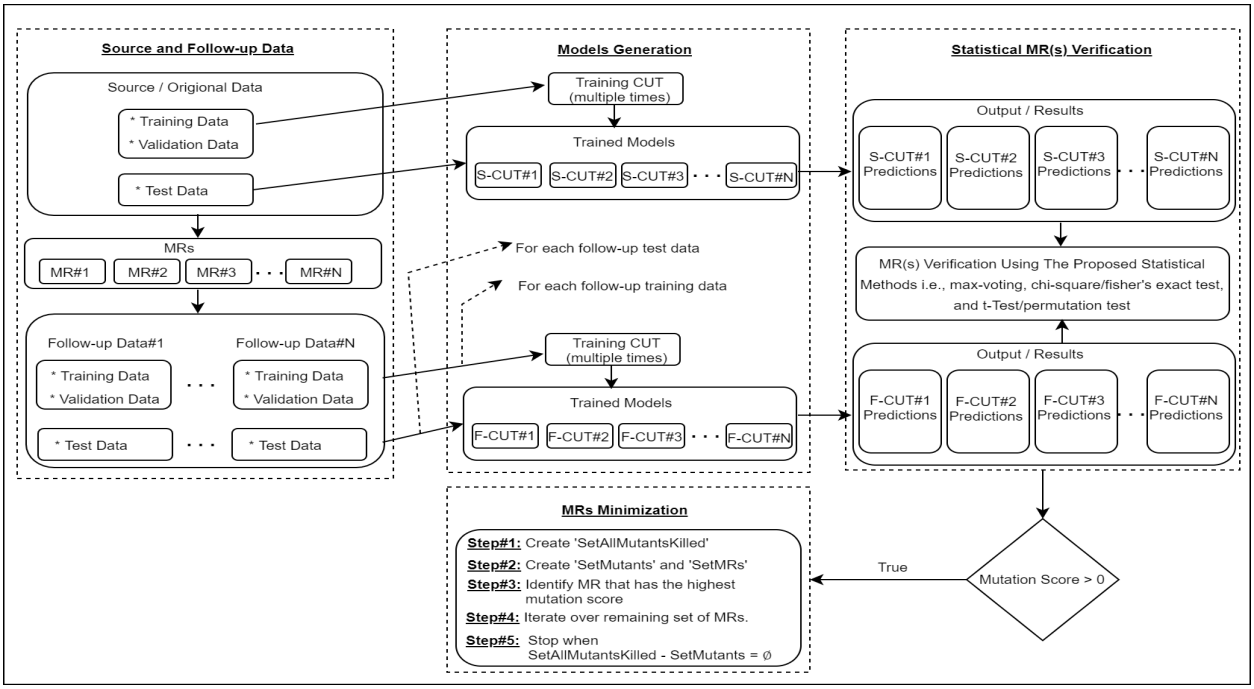


Figure 5.1: Proposed Statistical Metamorphic Testing Approach

Metamorphic Relations (MRs)

We use the following set of seven MRs to test critical image processing solutions (i.e., the pneumonia detection application in our case) in the health care domain. It is important to mention that we first performed a verification step to check that the transformation proposed in each MR is not changing the overall performance (i.e., accuracy and F1-score) of the original model (which is a mutant free program); thus, all MRs can be considered valid transformations to test the image CUT. In Figure 5.2, we show the source/original image and the follow-up/transformed images generated using the relation captured in the specified MRs.

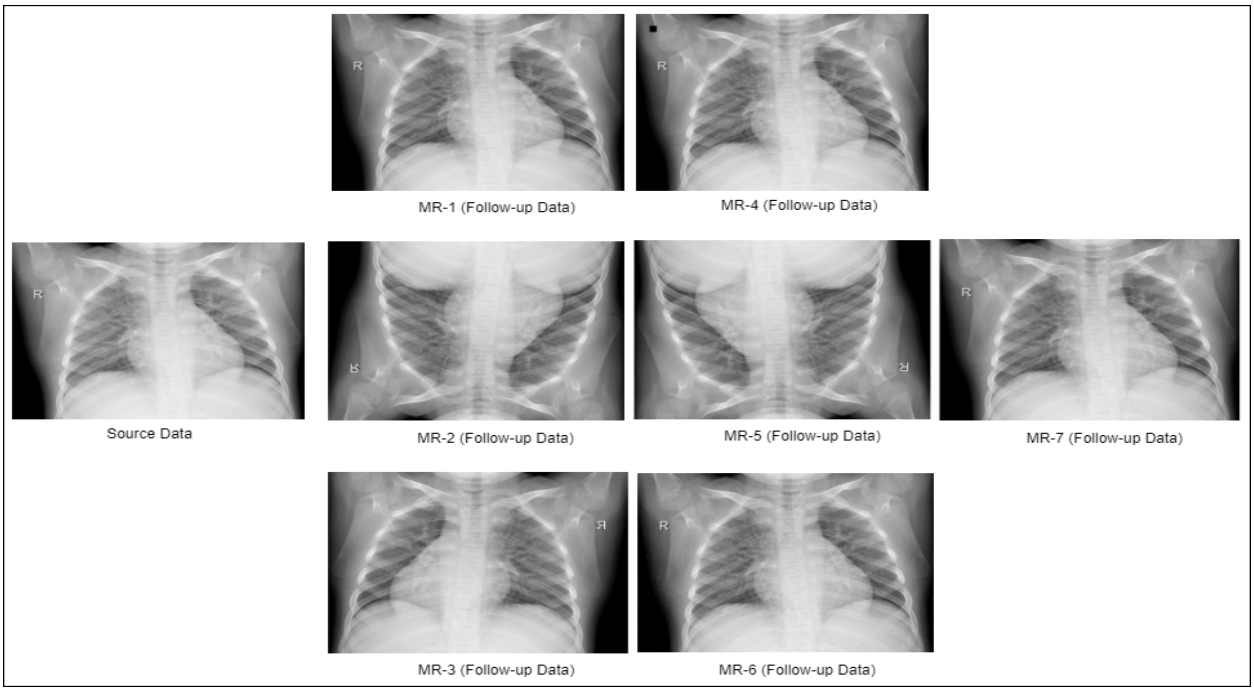


Figure 5.2: Real examples of the source and follow-up data

MR-1:-Blurring the training and test X-ray images Suppose, we have the training data represented as $X_{train}$ and the test data as $X_{test}$. We train the CUT on the source data $X_{train}$ and use it for prediction on the test data $X_{test}$. Let the output/class label obtained

for a given test instance $x^i_{test}$ be represented as 'c'. This MR specifies that for the follow-up execution, if we slightly blur the X-ray images in the given training and test data, the result produced by the program for the given test instance $x^i_{test}$ should stay the same i.e., class label 'c'.

MR-2:-Flipping the training and test X-ray images Suppose, we have the training data represented as $X_{train}$ and the test data as $X_{test}$. We train the CUT on the source data $X_{train}$ and use it for prediction on the test data $X_{test}$. Let the output/class label obtained for a given test instance $x^i_{test}$ be represented as 'c'. This MR specifies that for the follow-up execution, if we flip the X-ray images in the given training and test data, the result produced by the program for the given test instance $x^i_{test}$ should stay the same i.e., class label 'c'.

MR-3:-Mirroring the training and test X-ray images Suppose, we have the training data represented as $X_{train}$ and the test data as $X_{test}$. We train the CUT on the source data $X_{train}$ and use it for prediction on the test data $X_{test}$. Let the output/class label obtained for a given test instance $x^i_{test}$ be represented as 'c'. This MR specifies that for the follow-up execution, if we mirror the X-ray images in the given training and test data, the result produced by the program for the given test instance $x^i_{test}$ should stay the same i.e., class label 'c'.

MR-4:-Adding a small rectangle (outside the region of interest) to the training and test X-ray images Suppose, we have the training data represented as $X_{train}$ and the test data as $X_{test}$. We train the CUT on the source data $X_{train}$ and use it for prediction on the test data $X_{test}$. Let the output/class label obtained for a given test instance $x^i_{test}$ be represented as 'c'. This MR specifies that for the follow-up execution, if we add a small rectangle near the border of an X-ray image, i.e., (outside the region of interest i.e., chest ) in the given training and test data, the result produced by the program for the given test instance $x^i_{test}$ should stay the same i.e., class label 'c'.

MR-5:-Rotating the training and test X-ray images Suppose, we have the training data represented as $X_{train}$ and the test data as $X_{test}$. We train the CUT on the source data $X_{train}$ and use it for prediction on the test data $X_{test}$. Let the output/class label obtained for a given test instance $x^i_{test}$ be represented as 'c'. This MR specifies that for the follow-up execution, if we rotate the X-ray images (i.e., by $180^o$) in the given training and test data, the result produced by the program for the given test instance $x^i_{test}$ should stay the same i.e., class label 'c'.

MR-6:-Adding scattered dots to the training and test X-ray images This MR can be used to simulate the scenario when an X-ray machine has some dust on it. Suppose, we have the training data represented as $X_{train}$ and the test data as $X_{test}$. We train the CUT on the source data $X_{train}$ and use it for prediction on the test data $X_{test}$. Let the output/class label obtained for a given test instance $x^i_{test}$ be represented as 'c'. This MR specifies that for the follow-up execution, if we scatter dots over X-ray images in the given training and test data, the result produced by the program for the given test instance $x^i_{test}$ should stay the same i.e., class label 'c'.

MR-7:-Sharpening the training and test X-ray images Suppose, we have the training data represented as $X_{train}$ and the test data as $X_{test}$. We train the CUT on the source data $X_{train}$ and use it for prediction on the test data $X_{test}$. Let the output/class label obtained for a given test instance $x^i_{test}$ be represented as 'c'. This MR specifies that for the follow-up execution, if we slightly sharpen the X-ray images in the given training and test data, the result produced by the program for the given test instance $x^i_{test}$ should stay the same i.e., class label 'c'.

Statistical Verification Method

In this section, we discuss the proposed statistical approach we used to verify/check the relation(s) captured in the proposed MR(s). We leverage the existing statistical methods that we proposed in our prior work [60] for testing a fully connected neural network-based anomaly detection system. We further validate and show the applicability of the proposed statistical methods in a different domain (i.e., image classification in the healthcare space). To establish a baseline, we briefly discuss each statistical method here (additional details are available at [60]). As previously mentioned, the limitation of using the traditional MT technique is that it does not work well when there is non-determinism in the outputs of a program (such as DNNs, because of randomly initializing the network weights). Therefore, we adapt the traditional MT technique and propose a statistical MT technique that statistically compares the outputs over multiple iterations (obtained for both the source/original and the generated follow-up test cases) and verify whether the difference in the results is statistically significant or not. If it is significant, the *Null Hypothesis* stating that *'For a given test instance $x^i_{test}$, the difference in the outputs for the source and follow-up executions is not statistically significant'* is rejected. This ultimately suggests that the relation captured in the corresponding MR is violated and indicates the existence of a potential bug in the application under test.

Maximum Voting In order to apply the maximum voting concept, we train the model 'u' times on the given source data $X_{sd}$ and 'v' times on the generated follow-up data $X_{fd}$ (where, u = v). This results in producing $M_u$ number of trained models on $X_{sd}$ and $M_v$ number of trained models on $X_{fd}$. We then compare the results to see whether, for the given test instance $x^i_{test}$, the class that is predicted a maximum number of times for source executions (by $M_u$ models) is different from the class predicted for the follow-up executions (by $M_v$ models). If so, the MR is said to be violated.

Comparing Distributions Over All Class Labels One of the limitations we observed for the maximum voting approach is that there may exist some cases where the class predicted a maximum number of times for both the source/original and the follow-up executions is the same but there is a significant difference in the frequency distribution over all predicted class labels. In such types of scenarios, the maximum voting concept may not provide realistic results in the identification of bugs. Therefore, this provides further motivation to compare the distributions over all class labels (instead of just one) for better identification of faults in the PUT. We use the Chi-square parametric test of homogeneity ($\tilde{\chi}^2$) and formulate this problem as comparing the class labels' distribution obtained from the source execution (used as the expected distribution) with the class labels' distribution obtained from the follow-up execution (treated as the observed distribution). However, one of the assumptions for the Chi-square test of homogeneity is that each cell value (in a frequency distribution table) should have a reading of at least five [2]. In the cases where this assumption was violated, we also applied Fisher's non-parametric exact test, which does not require this assumption to be satisfied. In order to compare the distributions using the Chi-square test of homogeneity/Fisher's exact test, we propose the following null and alternative hypothesis:

- $H_0$*:* The class distributions obtained for the source and follow-up executions are similar.
- $H_a$*:* The class distributions obtained for the source executions are different from the class distributions obtained for the follow-up executions.

If the *p-value* obtained is below the set significance level, the null hypothesis is said to be rejected, which will suggest that there is a significant difference in the class distributions obtained for the source and follow-up executions. Thus, the MR is said to be violated.

Comparing Distributions Over Probability Scores Another method that we propose to verify MRs is to statistically compare the predicted probability scores (instead of final

output/class labels). In the CNN-based CUT, the last layer (which is an output layer) uses the sigmoid activation function which produces a probability vector containing probability scores for all the class labels. For the given test instance $x^i_{test}$, the class for which the probability score is higher is treated as the final output of the model. We leverage these probability scores and perform statistical analysis to check whether for the given test instance $x^i_{test}$, the probability scores predicted during source executions are close to the scores predicted for the follow-up executions. If there is a significant difference, the MR is said to be violated.

In order to compare the probability scores, we developed a Windows-based desktop utility (using C#) that processes the raw results (obtained in the form of probability scores for each class label) and transform them into a form in which statistical analysis could be performed. We first identify the class label that is predicted a maximum number of times (i.e., identifying the class for which the model is most likely to be sure) during source executions. We then leverage this knowledge and extract the probability scores for the same class predicted during follow-up executions. As an example, Table 5.1 shows the probability scores extracted for a specific class for both the source/original and the follow-up executions.

Table 5.1: Exemplary Probabilities

| **Source Executions** | **Follow-up Executions** |
|---|---|
| 0.77 | 0.57 |
| 0.71 | 0.51 |
| 0.79 | 0.53 |
| 0.72 | 0.52 |
| 0.75 | 0.53 |
| 0.72 | 0.53 |

We treat this problem as comparing two sample means and propose the following null and alternative hypotheses:

- $H_0$*:* The sample mean of probability scores for the source executions is similar to the sample mean of probability scores for the follow-up executions.
- $H_a$*:* There is a significant difference in the sample means of probability scores for the source and follow-up executions.

In order to statistically compare the two sample means, we use the parametric t-Test to compare the means of the two groups. One of the assumptions for the t-Test is the satisfiability of the normality assumption. Although the t-Test is considered robust to slight violations in the normality assumption, during our analysis, we found some cases where this assumption is badly violated. For example, Figure 5.3 shows the Q-Q plot (for MR-5) that has long tails at the end, showing a slight violation of the normality assumption. Therefore, for making better analysis and decision-making purposes, we also applied the non-parametric 'permutation test' to present our results. For the given test instance $x^i_{test}$, if the difference between the probability scores is found to be statistically significant (i.e., less than the set significance level), the null hypothesis $H_0$ is rejected; thus, the MR is said to be violated, which signifies the existence of a potential bug in the PUT.

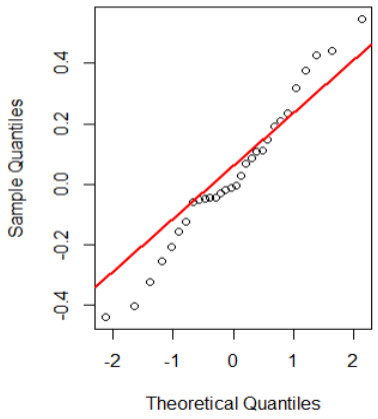


Figure 5.3: Q-Q plot

## Empirical Results

We evaluate our proposed statistical metamorphic testing technique for testing an open source deep learning-based pneumonia detection model [1]. It uses a CNN (as a deep learning neural network) built on top of Keras 2.8.0 and TensorFlow 2.8.0, trained on a data set having 5216 chest X-ray images. We further use the mutation testing technique to inject 7 valid mutants that are either of 'statement replacement, statement removal, or constant replacement' type. For example, one of the mutants (of type 'statement removal' mutant) is shown in Figure 5.4 which prevents the model from aggregating the calculated gradients and may cause multiple models to converge at significantly different points.

Due to the stochastic nature of training CNN-based CUTs, we use multiple iterations of source and follow-up executions to statistically verify the MRs. For each MR, we train the model 'u' times on the given source/original data and 'v' times on the generated follow-up data (where, u=30 and v=30). We then use these trained models to predict the output for the given test data. All the necessary information about the predicted outputs (i.e., test instance, predicted class label, expected class label, probability vector, and max probability

[1]https://github.com/sanghvirajit/Medical-Image-Classification-using-CNN

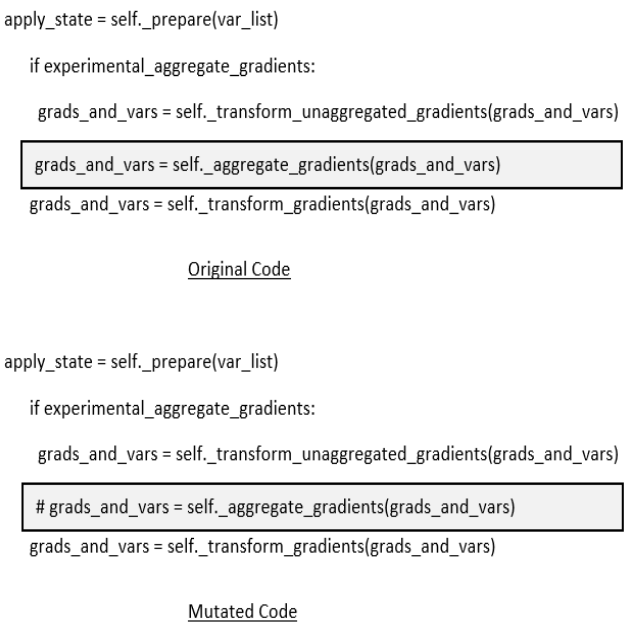


Figure 5.4: An example of the 'statement removal' mutant

for the predicted class) is logged into excel files. We then develop a .Net Framework-based desktop utility (using C#) to further process this raw data and transform it into a form (i.e., storing the processed results into new excel files) on which we apply the proposed statistical methods (implemented in R) to verify the relation specified in the MRs. The violation of the MR will suggest that the mutant is killed. The application code, details about the injected mutants, the MRs implementation, the raw excel files, the processed versions of raw files, the .Net utility, R scripts, and the final results (in excel format), all are publicly available in a shared GitHub repository[2].

We use the Mutation Score, which is 'Number of mutants killed / Total number of injected mutants,' as a measure to show the effectiveness of our proposed approach in the identification of faults injected in the CUT. As shown in Tables 5.2, 5.3, 5.4, and 5.5, we use this measure to not only show the results of each MR at an individual level but also for

[2]https://github.com/matifkhattak/SMT4DL

combined MRs for each of the proposed statistical methods. In order to identify whether the MR is able to kill the mutant or not, we use a standard significance level ($\alpha$) of 0.05 and statistically evaluate the results (for both the source/original and the follow-up executions) to check whether the difference is statistically significant or not. If it is, the null hypothesis is said to be rejected, suggesting that the MR has been violated and the mutant is said to be killed.

Table 5.2: Results For Maximum Voting Concept, ✓ Denotes The Mutant Is Killed

| | MR1 | MR2 | MR3 | MR4 | MR5 | MR6 | MR7 |
|---|---|---|---|---|---|---|---|
| Mutant1 | ✓ | | | | | ✓ | |
| Mutant2 | | | | ✓ | | | |
| Mutant3 | | | | | | | |
| Mutant4 | | | | ✓ | | | |
| Mutant5 | | ✓ | ✓ | ✓ | | ✓ | ✓ |
| Mutant6 | | | | ✓ | | ✓ | |
| Mutant7 | ✓ | | | | | | |
| **Mutation Score** | 28.57% | 14.29% | 14.29% | 57.14% | 0% | 42.86% | 14.29% |
| | **85.71%** | | | | | | |

**RQ2.1 (of RG2): How effective is the proposed approach (i.e, Statistical Metamorphic Testing) in the identification of implementation bugs in the CUT?**

In Table 5.2, we provide the mutation score (used as a metric to answer the research questions in a quantifiable manner) for all the combined MRs verified through the maximum voting concept, Table 5.3 provides the mutation score for all the combined MRs verified through the chi-square test/fisher exact test (significance level = 0.05), Table 5.4 provides the mutation score for all the combined MRs verified through the t-Test (significance level =

Table 5.3: Results For Chi-square test/Fisher Exact Test (Significance Level ($\alpha$) = 0.05), ✓ Denotes The Mutant Is Killed

| | MR1 | MR2 | MR3 | MR4 | MR5 | MR6 | MR7 |
|---|---|---|---|---|---|---|---|
| Mutant1 | | ✓ | ✓ | ✓ | ✓ | ✓ | |
| Mutant2 | | | ✓ | ✓ | | | |
| Mutant3 | | | | | | | |
| Mutant4 | | ✓ | ✓ | ✓ | | | |
| Mutant5 | ✓ | | | | | ✓ | |
| Mutant6 | | | | ✓ | | ✓ | |
| Mutant7 | | | ✓ | | | | |
| **Mutation Score** | 14.29% | 28.57% | 57.14% | 57.14% | 14.29% | 42.86% | 0% |
| | **85.71%** | | | | | | |

0.05), and Table 5.5 provides the mutation score for all the combined MRs verified through the permutation test (significance level = 0.05). In total, we inject 7 valid mutants into the CUT, among which 6 (85.71%) have been killed by the proposed MRs (for all the statistical methods). So, the overall fault detection effectiveness of the proposed MRs is 85.71%, suggesting that the proposed statistical metamorphic testing approach is effective in the identification of faults in the CUT, **which answers RQ2.1 (of RG2)**.

**RQ2.2 (of RG2): Do all MRs (verified through different statistical methods) have the same fault detection effectivenss?**

In Tables 5.2, 5.3, 5.4, and 5.5, we also present the results of each MR at the individual level. In Table 5.2, it can be seen that when the maximum voting approach is used for verification of MRs, MR4 has the highest mutation score (57.14%), whereas, MR5 has the lowest i.e., 0%. Similarly, results in Table 5.3 show that when MRs are evaluated

Table 5.4: Results For t-Test (Significance Level ($\alpha$) = 0.05), ✓ Denotes The Mutant Is Killed

| | MR1 | MR2 | MR3 | MR4 | MR5 | MR6 | MR7 |
|---|---|---|---|---|---|---|---|
| Mutant1 | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Mutant2 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Mutant3 | | | | | | | |
| Mutant4 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Mutant5 | ✓ | | | ✓ | | ✓ | |
| Mutant6 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Mutant7 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **Mutation Score** | 71.43% | 71.43% | 71.43% | 85.71% | 71.43% | 85.71% | 71.43% |
| | **85.71%** | | | | | | |

statistically using the chi-square test/fisher exact test, both MR3 and MR4 have the highest fault detection effectiveness (57.14%), whereas, MR7 has the lowest (0%). Lastly, results in Table 5.4, and 5.5 show that MRs statistically verified through the t-Test and permutation test give better results (in terms of their fault detection effectiveness) and show that for both methods, MR4 and MR6 have the highest fault detection effectiveness (85.71%), whereas, the rest of the MRs have the same (71.43%). **This answers RQ2.2 (of RG2)**, that most of the MRs when verified through a t-Test and a permutation test have the same defect detection ability, whereas, for maximum voting, MR1, MR4, MR5, and MR6 have different fault detection effectiveness, and for the chi-square test/fisher exact test, MR2, MR3, MR6, and MR7 uncover a different number of faults in the CUT.

**RQ2.3 (of RG2): Using the proposed approach, which MR(s) have high fault detection effectiveness and which among them have the least?**

Table 5.5: Results For Permutation (Significance Level ($\alpha$) = 0.05), ✓ Denotes The Mutant Is Killed

| | MR1 | MR2 | MR3 | MR4 | MR5 | MR6 | MR7 |
|---|---|---|---|---|---|---|---|
| Mutant1 | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Mutant2 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Mutant3 | | | | | | | |
| Mutant4 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Mutant5 | ✓ | | | ✓ | | ✓ | |
| Mutant6 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Mutant7 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **Mutation Score** | 71.43% | 71.43% | 71.43% | 85.71% | 71.43% | 85.71% | 71.43% |
| | **85.71%** | | | | | | |

To answer this research question, we analyze the results at a broader level. We collectively analyze the results provided in Tables 5.2, 5.3, 5.4, and 5.5 in order to understand how many of the total mutants have been killed by each of the individual MRs, assuming that the organization has enough testing resources to verify the MRs using all the proposed statistical methods. In this case, a mutant is said to be killed if any of the MRs (for any of the proposed statistical methods) has been violated. After close review, we identify that every MR is able to kill a sufficient number of mutants (i.e., 6 out of 7), and noticed a couple of interesting cases, i) MR5 is unable to kill mutant5 for any of the proposed statistical methods, and ii) none of the proposed MRs is able to kill mutant3. We perform further analysis to check why mutant3 has survived. Upon close analysis, we identify that every CNN trained (for all the source and follow-up executions) has predicted the same class label and produced the same probability vector for all the given test instances; thus, none of the MRs (using the proposed statistical methods) was able to kill the mutant. Therefore, based

on our analysis, we conclude that among all the MRs, MR5 has the least fault detection effectiveness (killing 5 out of 7 mutants), whereas, the rest of the MRs have the highest fault detection effectiveness (killing 6 out of 7 mutants), **which answers RQ2.3 (of RG2)**.

**RQ2.4 (of RG2): How can the proposed MRs (verified through the proposed statistical methods) be minimized for the DNN-based pneumonia detection model under test?**

To answer this research question (in the context of the proposed statistical metamorphic testing approach), we take advantage of using a few concepts together, i.e., the mutation score, the type of mutants killed by each MR, and the difference of sets (from set theory) to propose an MR minimization algorithm. The new set of minimized MRs (i.e., represented as C) is expected to have the same fault detection effectiveness (i.e., the same mutation score) as that of using the original set of 7 MRs (i.e., represented as D), where $|C| <= |D|$; thus, saving computational costs and organizational testing resources. As an example, to minimize the list of MRs verified through the maximum voting concept (as shown in Table. 5.2), the following steps are followed. This algorithm will execute when the base condition (i.e., mutation score $> 0$) is true.

*Step#1:* Create a set called *SetAllMutantsKilled* that stores information about all the mutants that have been killed by the proposed method i.e. for the maximum voting approach, the set would be comprised of {Mutant1, Mutant2, Mutant4, Mutant5, Mutant6, Mutant7}.

*Step#2:* Create two empty sets, i) *SetMutants* (that progressively keeps track of killed mutants), and ii) *SetMRs* (that progressively keeps track of MRs that have killed the mutants i.e., mutants existing in *SetMutants*).

*Step#3:* Start with the identification of an MR that has the highest mutation score i.e. MR4 (for maximum voting approach). Then, add this MR to *SetMRs* i.e, {MR4}, and the mutants it killed to *SetMutants* i.e, {Mutant2,Mutant4,Mutant5,Mutant6}. In case, there are multiple MRs having the same mutation score, two possibilities can exist, i) they killed

the same type of mutants. If so, then randomly select one of them (since using one of them is enough to kill all those mutants), ii) they killed different types of mutants. In this case, we treat each of them as a separate case and follow the remaining steps for each of the cases. As an example, in Table 5.3, it can be seen that for the Chi-square test/fisher exact test, MR3 and MR4 have the same mutation score but have killed different types of mutants. Therefore, two possible MRs prioritization paths could be initiated (i.e., one with MR3, and the other with MR4) that will end up in the identification of two MRs minimization sets i.e., {**MR3**, MR6} and {**MR4**, MR3, MR6} (as shown in Table 5.6).

*Step#4:* Sequentially, iterate over the remaining set of MRs (not yet added to *SetMRs*). For each MR, identify the mutant(s) it killed, add them to a separate set i.e, *SetTemp* (that will keep track of the mutants killed by the selected MR), and calculate the difference between the two sets i.e., *SetTemp* - *SetMutants*. The MR for which the difference is largest (i.e., killing the largest number of mutants missing in *SetMutants*) is added to further expand the *SetMRs*, and its corresponding killed mutants to *SetMutants*. As an example, for the maximum voting approach, the next MR added to the *SetMRs* is MR1 i.e., {MR4, MR1}, and its killed mutants to the *SetMutants* = {Mutant2, Mutant4, Mutant5, Mutant6, Mutant1, Mutant7}.

*Step#5:* The algorithm will stop when the difference between the two sets i.e., *SetAllMutantsKilled* - *SetMutants* = $\emptyset$, which means there are no more mutants left for the algorithm to identify and add the MR to further expand the minimized set of MRs (i.e., *SetMRs*). As an example, the final minimized set of MRs identified for max-voting is comprised of {MR4, MR1}. Similarly, for the Chi-square test, we used the same steps (defined above) and identified two MRs minimization sets i.e., {MR3, MR6} and {MR4, MR3, MR6}. However, the set containing a minimal set of MRs will be selected i.e., {MR3, MR6} as a final set, because this set contains fewer MRs (hence, less testing resources will be consumed) but has the ability to kill the same number of mutants which are either killable

through {MR4, MR3, MR6} or using the complete set of 7 MRs.

Table 5.6: MRs Minimization

| Priority | Max-voting | Chi-square/Fisher Exact Test | t-Test | Permutation Test |
|---|---|---|---|---|
| 1 | MR4 | MR3 —— MR4 | MR4 or MR6 | MR4 or MR6 |
| 2 | MR1 | MR6 —— MR3 | | |
| 3 | | —— MR6 | | |

We follow the algorithm (proposed above) to minimize the set of MRs for rest of the statistical methods and present the results in Table 5.6. It can be seen that for maximum voting, just two MRs with their prioritization order MR4, and M1 are enough to attain a mutation score of 85.71% (i.e., the overall mutation score achievable through all the 7 MRs). Similarly, for the chi-square/fisher exact test, MR3, and MR6 are enough to attain the overall mutation score, whereas, for the t-Test and permutation test, only one MR (either of MR4 or MR6) is enough to kill the 6 mutants, **which answers RQ2.4 (of RG2)**. Such a minimized set of MRs (for each of the proposed statistical methods) helps in saving organizational resources and performing testing with fewer MRs (especially in a regression testing environment) without compromising the overall fault detection effectiveness of the proposed approach.

## Conclusion

Manual examination of chest X-rays images (for detection of pneumonia among patients) is not only a time-consuming and resource-intensive task but is also subjective and prone to errors. This raises a need for an efficient method that can automatically detect

the existence of pneumonia among patients correctly. ML-based solutions are playing a significant role in the automatic and timely detection of pneumonia among patients using chest X-ray images. To test such critical systems, the ML community frequently uses performance evaluation metrics i.e., accuracy, precision, recall, F1-score, etc. However, the fact is that these evaluation metrics are not meant to test (i.e., verify and validate) ML-based models for finding bugs in them, instead, they are used to evaluate which ML algorithm is best suited for the underlying data/problem. Apart from that, these evaluation metrics say nothing about the existence of faults in the model or the underlying used algorithm. Therefore, the use of a systematic testing strategy is essential to test such critical systems in order to verify their correctness and enhance trust in them.

The metamorphic testing technique is a simple but effective testing strategy to alleviate the oracle problem in testing such computationally complex programs where the output for individual input is difficult to verify. However, the stochastic nature (because of randomly initializing the network weights) in the training of deep learning-based models, (i.e., the CNN-based pneumonia detection model in our case) adds an extra challenge and makes the traditional metamorphic testing technique infeasible to apply. In this research, we address this problem by proposing a statistical metamorphic testing technique that does not require fixing the random seeds to get deterministic results for verification of the MRs. We propose 7 MRs in combination with statistical methods (for MRs verification) to identify implementation faults in an open-source CNN-based image classifier that has stochastic nature in its results. The empirical results obtained show that our proposed method is able to uncover 85.71% of the implementation faults in the CUT. Furthermore, we also propose an algorithm for minimization of the proposed MRs; thus, saving computational costs and organizational testing resources.

In the future, we aim to show further applicability of the proposed approach in testing CNNs-based ML models used for addressing multi-class classification problems in

the healthcare space. We also aim to further expand the MRs repository and leverage machine learning techniques for their effective prioritization and minimization.

# A HYBRIDIZED APPROACH FOR TESTING NEURAL NETWORK BASED INTRUSION DETECTION SYSTEMS

## Contribution of Authors and Co-Authors

Manuscript in Chapter titled 'A Hybridized Approach for Testing Neural Network Based Intrusion Detection Systems'

Author: Faqeer ur Rehman

Contributions: Problem identification and proposing solution, running experiment, manuscript writing, creating tables and figures. Primary writer

Co-Author: Dr. Clemente Izurieta

Contributions: Contribution in manuscript editing/writing, provided feedback, guidance and advice.

Manuscript Information Page

Faqeer ur Rehman and Dr. Clemente Izurieta

IEEE International Conference on Smart Applications, Communications and Networking (SmartNets)

Status of Manuscript:
___ Prepared for submission to a peer-reviewed journal
___ Officially submitted to a peer-reviewed journal
___ Accepted by a peer-reviewed journal
_×_ Published in a peer-reviewed journal

IEEE
05 October 2021
10.1109/SmartNets50376.2021

## Abstract

Enhancing the trust of machine learning-based classifiers with large input spaces is a desirable goal; however, due to high labeling costs and limited resources, this is a challenging task. One solution is to use test input prioritization techniques that aim to identify the most effective test instances. These prioritized test inputs can then be used with some popular testing techniques e.g., Metamorphic testing (MT) to test and uncover implementation bugs in computationally complex machine learning classifiers that suffer from the oracle problem. However, there are certain limitations involved with this approach, (i) using a small number of prioritized test inputs may not be enough to check the program correctness over a large variety of input scenarios, and (ii) traditional MT approaches become infeasible when the programs under test exhibit a non-deterministic behavior during training e.g., Neural Network-based classifiers. Therefore, instead of using MT for testing purposes, we propose a metamorphic relation to address a data generation/labeling problem; that is, enhancing the test inputs effectiveness by extending the prioritized test set with new tests without incurring additional labeling costs. Further, we leverage the prioritized test inputs (both source and follow-up data sets) and propose a statistical hypothesis testing (for detection) and machine learning-based approach (for prediction) of faulty behavior in two other machine learning classifiers (Neural Network-based Intrusion Detection Systems). In our case, the problem is interesting in the sense that injected bugs represent the high accuracy producing mutated program versions that may be difficult to detect by a software developer. The results indicate that (i) the proposed statistical hypothesis testing is able to identify the induced buggy behavior, and (ii) Random Forest outperforms and achieves the best performance over SVM and k-NN algorithms.

## Introduction

Machine Learning (ML) provides core functionality to many critical application domains, such as bioinformatics, network security, collaborative robots, and self-driving vehicles. Developing high-quality ML models to solve real-world classification, prediction, and clustering problems is always desirable because a small bug in these critical high-consequence applications can lead to disastrous consequences and can pose serious threats to life and property [40, 52, 90]. Therefore, it is very important to verify their correctness before deploying them in a production environment. However, the computational complexity, large input space, and lack of a test oracle raise serious challenges in order to verify their correct functionality. Consequently, these applications end up in a category of non-testable programs, also known as programs suffering from the *Oracle Problem* [80].

From a quality assurance perspective, it is always desirable to verify the correctness of ML-based classifiers over a large range of test scenarios. However, the high cost of obtaining a test oracle, a.k.a. labeling a large number of test instances, makes it a resource-intensive and infeasible task. One of the solutions proposed in the literature is to use test input prioritization techniques that aim to identify and label only the most effective test instances [17, 86]. However, concerns like, whether the small number of prioritized test inputs are enough to test the correctness of critical ML applications over a diverse set of data, are unaddressed. We propose a *Metamorphic Relation* (MR) that aims to target this data collection/labeling problem by extending the prioritized test set with new tests without incurring additional labeling costs. It thus allows the software tester to check the program correctness over a diverse set of input scenarios.

Metamorphic Testing (MT) is considered an effective testing technique to alleviate the oracle problem in testing ML applications [20, 23, 42, 72]. MT is not only a testing technique but can also be used to generate domain specific data using valid transformations to the input

data e.g., rotation, scaling, reflection etc. At the heart of MT are MRs that are derived from the necessary characteristics of the program under test. Each MR is composed of a source test case and a follow-up test case. A valid transformation performed on the source test-case generates a new test case, known as a follow-up test case. Instead of verifying the output for individual inputs, the relation between the input and its associated output is used to verify the program correctness. The MR is said to be violated if the result of the source and follow-up test case does not adhere to the relation specified in the corresponding MR. A violation of a MR will indicate that there is some potential bug in the system.

One of the challenges faced in applying tradition MT approach to test ML applications is when the program under test exhibits a stochastic behavior in training e.g., Artificial Neural Network (ANN) based classifiers (due to random initialization of weights). For example, given the same inputs, the NN-based application may produce slightly different output in each run; thus rendering traditional MT techniques inadequate if they rely on using the equality operator to check the relation specified in the MR. One of the solutions proposed in the literature (for getting deterministic results) is either to fix the random seed(s) or initialize the network weights with the same constant values [23, 42]. However, the proposed solution has its own limitations i.e., it is only applicable in an environment where either the random seeds or the weights can be fixed, which is not always possible. Furthermore, even if somehow the random seeds have successfully been fixed, it is not guaranteed that we can obtain deterministic results on GPUs because of inherent non-determinism introduced by NVidia CUDA libraries [23]. Also, in practice, it may not be viable for a software tester to identify and then fix the seeds in a highly complex system internally using a large number of third-party libraries, especially when some of the libraries do not provide any option at all. To alleviate this problem, our prior work focuses on proposing statistical measures for testing deep neural network based applications in a non-deterministic environment [60].

The above-highlighted limitations motivate us to propose an approach that can not

only be used to extend the prioritized test set with new test inputs (using the proposed MR) but can also leverage these inputs (both source and follow-up test sets) to test NN-based classifiers (in both deterministic and non-deterministic environments) using a combination of statistical and ML techniques. The purpose of performing statistical analysis is to check whether the difference between the results produced by original and mutated program versions is statistically significant. If so, the buggy behavior is said to be detected. Next, we use this knowledge to build ML classifier that can be used to predict whether, for the given prioritized test inputs, the output (probability distribution over classes) produced by an NN-based classifiers under test exhibits a 'buggy' or 'non-buggy' behavior. These prioritized test inputs can be a part of permanent test case repository that the organization frequently uses to test the new release in regression testing environment and reducing the overall cost of software testing phase.

To show the effectiveness of the proposed approach, we have worked with detecting and predicting buggy behavior in two NN-based Intrusion Detection Systems (N-IDSs). The following are the main contributions made in this paper [76]:

- Mutants generation to obtain the faulty versions of the program which give the same high accuracy as that of the original program, hence difficult to identify the buggy behavior induced by them.

- Instead of relying on a synthetic data set (generated randomly) that may contain noise and missing values, we propose an MR to generate a new set of prioritized data without incurring additional labeling costs. Such MRs can be very useful in a scenario when we do not have enough test instances available but are still interested in verifying the system correctness over a diverse set of data (other than the data already available).

- Statistical analysis of probability scores over predicted classes (predicted by original and mutated programs) using a statistical hypothesis testing technique to check

whether the difference is significant. If so, the buggy behavior is said to be successfully detected.

- Proposed an approach that takes advantage of ML techniques to test and predict faults in NN-based classifiers using prioritized test inputs.
- Conducted one more independent experiment to show that addition of an informative metadata features can further enhance the performance of proposed ML classifiers.

## Related Work

Testing ML applications is getting considerable attention in the research community, however, couple of challenges make it a harder task, i.e., (i) high labelling costs, and (ii) susceptibility to the oracle problem due to large input space. Byun et al. [17] focused on proposing a test prioritization technique to reduce the cost associated with labeling the data instances. They took advantage of the probability vectors (produced by either *softmax* or *sigmoid* output layers in neural networks) and proposed three sentiment measures (confidence,uncertainty, and surprise) to prioritize the inputs that are more likely to reveal faults in a trained model. Zhang et al. [86] used probability vectors to prioritize the uncertain or sensitive inputs that have the high potential to become adversarial examples using small perturbations. Dwarakanath et al. [23] proposed MT based approach to alleviate the oracle problem in testing DNN based image classifiers but the proposed approach is only applicable in an environment where the random seeds can be fixed to get deterministic results.

Shaikh et al. [68] introduced ML-based classification models (LibSVM and LinearLib) that use NASA dataset models to foresee the faults in a software defect-prone model. Based on comparative analysis performed, the results show that overall LibSVM attains high accuracy and is more efficient than LinearLib. Prabha et al., [57] first performed dataset dimensionality reduction (using PCA) and then applied Random Forest, Naïve Bayes, SVM,

and Neural Networks to predict software defects that can help software developers to identify and correct a program's buggy code before deploying them in the actual environment. It also allows the software development team to prioritize and focus more on the modules that are problematic. Sethi [67] proposed Artificial Neural Network (a feed-forward back propagation model) to predict defects in twenty software projects. It was found that the proposed model provides better performance than the classic fuzzy-based approach. Nehi et al., [50] took advantage of the history available in version control systems to perform software defect prediction. The authors used the code history information (e.g., bug reported by the client, code defects determined, etc.) extracted from several open-source projects and prepared a benchmark data set. To show its usefulness, the data set is used to evaluate the performance of different defect prediction approaches. The authors applied Artificial Neural Networks, Random Forest, K-Nearest Neighbor, Naïve Bayes, and Support Vector Machine and performed their evaluation using different measures e.g., Precision, Recall, F-measure, G-mean, and AUC. The results show that Random Forest outperforms and has higher predictive power in the identification of faults in the programs under test.

It is important to mention that the available literature primarily focuses on using the public data sets and harnessing ML approaches in the prediction of faults in traditional software, not specifically in ML applications. This makes it a potential area for exploration and making for fruitful contributions; thus we propose an approach that takes advantage of statistical and ML techniques to detect and predict buggy behavior in ML classifiers using prioritized test inputs.

### Motivation To Use Probability Vectors / Scores

In this study, we took an advantage of using the probability vectors produced by the output layer in NN-based classifiers under test. Therefore, we provide a brief motivation regarding why we think that probability vectors contain additional information about the

computation performed in an NN-based application and can be used for detecting and predicting buggy behavior in the programs under test. The probability vectors have been used to prioritize test inputs that are more likely to uncover faulty behavior in deep neural networks so that efforts can focus only on labeling the prioritized inputs [17]. These vectors help in deriving sentiment-based metrics (confidence, uncertainty, and surprise) that capture extra information about the computation performed inside deep neural networks on the given data input(s). The higher priority is given to the data inputs that are more uncertain or surprising (i.e., the inputs for which the probability distribution is spread out) and likely to reveal the faulty behavior in the trained model. The probability vectors have also been used to detect high-noise sensitive inputs that have a high potential to become adversarial examples, such that if a small perturbation is added to them, they can fool the DNN [86]. Alternatively, low noise-sensitive inputs will require a significant perturbation that will easily be detected by defensive models and hence is not effective.

Knowing that probability vectors have been used in addressing the test input prioritization problem, we are interested in exploring their usefulness in the detection of buggy behavior and the development of ML models for its prediction in NN-based classifiers.

## Proposed Approach

This section discusses the proposed approach used for detection (using statistical hypothesis testing) and prediction (using ML-based approach) of implementation bugs in two NN-based classifiers under test, as shown in Figure 6.1.

In order to detect and predict defects in NN-based classifiers under test, we took advantage of multiple techniques namely, (i) random sampling method [59] for selection of prioritized test inputs (ii) *MR*: applying valid transformation to the source/original prioritized inputs to generate new (follow-up) prioritized inputs without incurring additional labeling costs, (iii) *Mutants generation*: injecting high accuracy producing mutants into the

NN-based classifiers under test, so that for the prioritized test inputs (both source and follow-up), the behavior of non-buggy/original and mutated program versions can be recorded and analyzed, (iv) *Statistical hypothesis testing*: to check statistically, whether the difference in the results produced by multiple program versions is statistically significant. If so, the buggy behavior is said to be detected, and (v) *ML*: automatic detection of bugs by leveraging ML techniques to learn from this difference and to predict whether, for the same prioritized test inputs, the new release under test shows the buggy or non-buggy behavior.

The effectiveness of the proposed approach is shown by testing two NN-based Network Intrusion Detection Systems (N-IDSs). The details for each classifier are shown in Table 6.1.

- **Application#1:** This application is a shallow NN-based N-IDS (having one hidden layer, an input layer, and an output layer) that deals with a multi-class classification problem in an Open Stack environment and predicts whether the request is normal, by an attacker, or by a victim.

- **Application#2:** This application is a deep neural network-based N-IDS (having three hidden layers, an input layer, and an output layer) that deals with a binary class classification problem and predicts whether the network request is malicious or benign.

Table 6.1: Classifiers Under Test Architecture

| **Classifiers Under Test** | **No of Layers** | **Hidden Layer(s) Type** | **Output Layer** |
|---|---|---|---|
| Application#1 | 3 | Fully Connected + ReLU | Softmax |
| Application#2 | 5 | Fully Connected + sigmoid | Sigmoid |

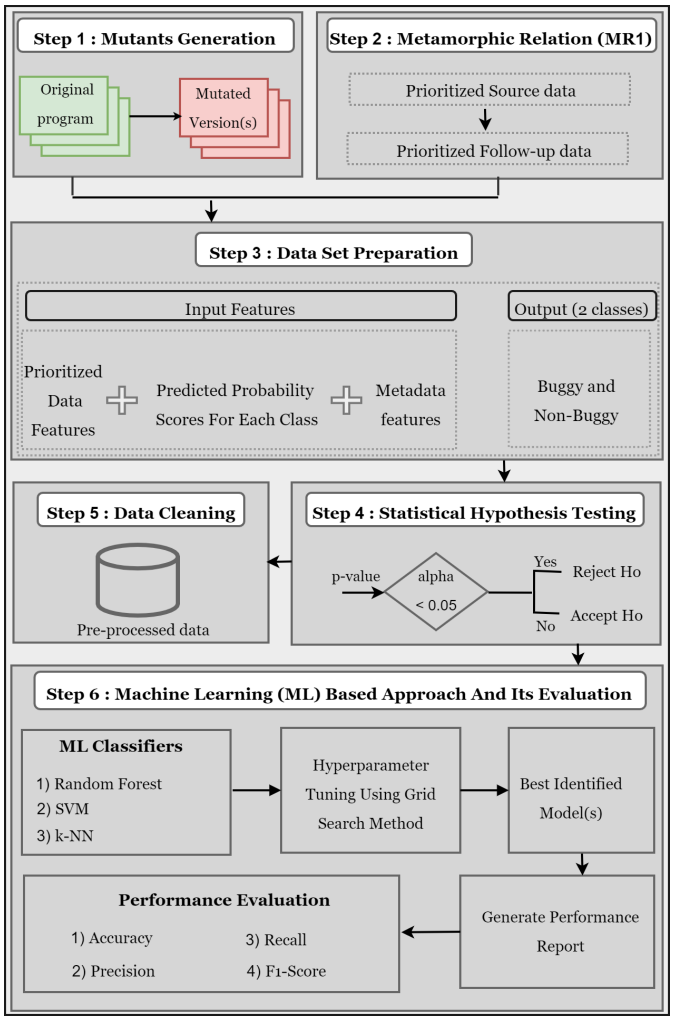


Figure 6.1: Proposed Approach

Step 1:-Mutants Generation

For the development of a machine learning model, a data set containing information about the buggy/mutated and non-buggy/original program versions is needed. So, the first step is to obtain the buggy code for the classifiers under test. As we do not have access to actual buggy versions of the classifiers under test, we took advantage of mutants generation process used in mutation testing technique [34], that is considered an effective approach in simulating the real faults made by programmers [71]. We injected 4 valid mutants into the

programs under test that give the same high accuracy as that of the original program. These mutants belong to the category of CRP-constant replacement type mutants. The behavior recorded for all these mutated versions is treated as buggy behavior. As shown in Table 6.2, it can be seen that these mutated versions achieve the same high accuracy as that of the original program, knowing the fact that they represent the faulty implementations. The goal is to apply statistical and ML techniques for the identification of buggy behavior induced by such high accuracy producing faulty programs; which otherwise may not be possible for software engineers who primarily focus on using accuracy measure to develop accurate models. Figure 6.2 represents one of the injected mutants to simulate the mistake made by a software developer. It will cause the model to learn from a lesser number of informative features but it has been observed that this mutated model has produced almost the same high accuracy as that of the original program. The details of other mutants, the python code for the NN-based classifiers under test, the data used for training the models, the R-code for performing statistical analysis, and the proposed ML models' Python code, are all open-source and available online[1].

Table 6.2: Original And Mutated Versions Average Accuracy (%)

| **Application** | **Original** | **Mutant#1** | **Mutant#2** | **Mutant#3** | **Mutant#4** |
|---|---|---|---|---|---|
| App#1 | 98.9% | 98.3% | 98.9% | N/A | N/A |
| App#2 | 92.5% | N/A | N/A | 92.5% | 92.9% |

```
total_loss = loss_weight * output_loss

total_loss = loss_weight / output_loss
```

Figure 6.2: Mutant#4: Original Code (top) and Mutant (below).

[1] https://github.com/matifkhattak/MLTesting/tree/master

Step 2:-Metamorphic Relation (MR-1) For New Test Inputs Generation- Shifting the features by constant $k$

Let $X_{train}$ denote the training data and $X_{test}$ denote the prioritized test data. In order to select the prioritized test instances, we have used the random sampling method [59]. After training the NN classifier, suppose it classifies a specific test instance $x^i_{test}$ as belonging to class *'a'* (known as source execution). MR-1 says that if the features in both the training and test data are shifted by constant *k; where* $k = 0.3$, the output for the test instance $x^i_{test}$ should remain the same i.e., class *'a'* (known as the follow-up execution). Such transformation will not change the existing relationship between the data points, and a correct NN model should learn from the relationship among the data points, not from their position in the space [23]. The follow-up transformation will help in the generation of new *'m'* prioritized follow-up instances from the given *'n'* number of prioritized original/source data instances (where n=m), thus extending the prioritized test set size from *'n'* to *'n+m'*, **which answers RQ3.1 and RQ3.2 of RG3.**

The proposed MR1 will facilitate in enhancing the testing phase effectiveness by doubling the size of prioritized test inputs, so that the behavior of NN-classifiers under test can be observed over a diverse set of data (both source and follow-up data). Another advantage of the proposed MR is that it also helps in removing the labeling costs for the newly generated test instances (follow-up data instances) that might be expensive in some cases.

Step 3:-Dataset Preparation

Due to the stochastic nature of NN-based classifiers, a single run may not be enough to verify their correctness, therefore, we obtain the results over multiple iterations. In this step, we record the results (probability distribution over classes predicted by original and mutated versions) and metadata information of NN-based classifiers under test separately

for both the prioritized source and follow-up test instances. As shown in Figure 6.1, each instance in the prepared data set is comprised of a prioritized test instance features, the probability scores over predicted classes, the trained NN Model's metadata information, and the class label (Buggy, Non-Buggy). As mentioned earlier, a correct NN classifier (retrained on the same data) may produce slightly different results for the given test instances; which does not necessarily mean that there is a bug in the system. Those multiple trained NN classifiers (if correct) may have different initialization points but will have almost the same convergence points [23]. By utilizing this property of NN classifiers, we obtain the results for prioritized test inputs over multiple iterations, whereas each iteration denotes a trained NN classifier. This allows us to capture the probability distribution for each prioritized test instance for a sufficient number of times that have been predicted by both the mutated and original NN-based classifiers under test.

Step 4:-Statistical Hypothesis Testing

Once the probability scores (over predicted classes) for both the original/non-buggy and mutated/buggy programs are obtained, we compare the probability scores for each of the $class_i$ (produced by the mutated version) with the probability scores of the same $class_j$ (produced by the original version). This is important because before developing the ML model, it first needs to be confirmed that for the given prioritized test inputs, the difference between the probability distributions over predicted classes for both types of programs is statistically significant, otherwise, there is no information available for the ML model to distinguish the buggy program version's behavior from the non-buggy one. It is important to mention that the proposed approach is not meant to identify the individual injected mutants, instead we are interested to identify whether in general, the program behavior can be characterized as buggy or not. This is the reason that we treat the behavior produced by all the mutated versions as 'buggy' and formulate this problem as statistically comparing the

two samples (buggy vs non-buggy). The following are the *Null and Alternative hypothesis*.

- $H_0$*:* There is no difference in the probability scores predicted by the original and the mutated programs.
- $H_a$*:* The probability scores predicted by the mutated programs are different from the original program.

After applying the test-statistic, the p-value obtained is compared with the set standard significance level ($\alpha = 0.05$). If it is less than the value of $\alpha$, $H_0$ is rejected. The rejection of $H_0$ would suggest that (i) the difference between two samples is significant, showing that there is likely a bug in the system, and (ii) the probability scores do have some information available to distinguish both types of programs (buggy and non-buggy).

Step 5:-Data Cleaning

It is possible that for multiple iterations the same probability distribution over classes is predicted for a specific test instance, which will result in the addition of duplicate observations in the data set. Such duplicate instances may bias the results of the ML models and can lead to incorrect conclusions. Therefore, a data cleaning process is an essential step in making sure that the observations are unique and the results produced by the models are unbiased. Thus, we performed the data pre-processing step and removed the duplicate observations before training the models.

Step 6:-Proposed Machine Learning Based Approach

The last step is to harness ML techniques for predicting buggy behavior in NN-based classifiers using prioritized test inputs. For this purpose, we select three widely used ML algorithms in the literature (i.e., Random Forest, SVM, and k-NN) to extract the hidden knowledge captured in the data set, as shown in Figure 6.1. The data set is split into 80%-20%, where 80% is used for training purposes and 20% is used for testing. We use

the k-fold cross-validated grid-search method to find the best hyperparameter settings for the classifiers. The best classifier (with optimal hyperparameter settings) is identified and is used as a final model to predict the output for the test data. In the end, a performance report is generated for each of the classifiers, and results are compared using the evaluation metrics e.g., accuracy, precision, recall, and F1-score (harmonic mean of precision and recall) [11], to select the best model having higher prediction power.

## Experimentation and Evaluation

A faulty new program release giving the same high accuracy as that of the original/correct program may classify a test instance with the same class label 'a' that was predicted by the original/correct program but may affect the probability distribution over the predicted classes. As an example, Figure 6.3 shows that both the original and high accuracy producing mutated programs have predicted the same class label 'attack' but the probability distributions over predicted classes have significantly changed for the mutated version. Our experiment shows that unlike predicted class labels, the probability scores provide more granular details and can be analyzed statistically to detect the buggy behavior induced by such high accuracy producing mutated program versions. In total, we injected 4 valid mutants into the programs under test and performed multiple iterations to obtain the results for 200 prioritized test instances (100 source and 100 follow-up instances), which resulted in a total of 8000 data instances (50% of the data is labeled as buggy and the other 50% is labeled as non-buggy). The data set for App#1 contains 37 features, whereas, for App#2, it contains 46 features.

***RQ4.1 (targeting RG4): Is the proposed statistical hypothesis testing technique effective in detection of buggy behavior (produced by the high accuracy producing mutated program versions) in the classifiers under test?***

Before training the ML model, it is important to check whether the probability

distribution over predicted classes for both types of programs (original and mutated) is statistically significant. Therefore, we perform statistical hypothesis testing to check whether, for the given prioritized test inputs, the injected mutants have significantly changed the probability distributions over predicted classes. Comparing the distributions of two classes (e.g., $class_i$ probability scores produced by mutated code vs the same $class_j$ probability scores produced by the original code) can be treated as a problem of statistically comparing two samples. During a single run, the same model is used for predictions for multiple prioritized test inputs, so they all are connected to the same model. For this reason, applying a paired t-Test will be an appropriate choice. However, during analysis, we found that the normality assumption was badly violated, hence applying the paired t-Test, although known to be a robust test, may not provide reliable results. For example, Figure 6.4 provides evidence that the normality assumption is violated for one of the classes (because of large tails at both ends). For this reason, the *Wilcoxon signed-rank test* is applied, which is a non-parametric test used for paired data and does not rely on the satisfiability of the normality assumption [59].

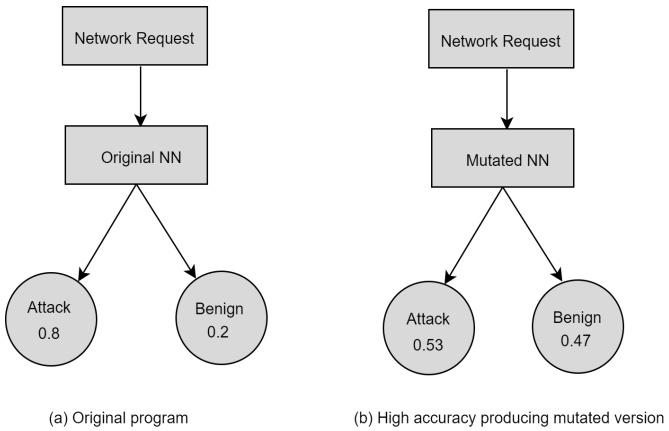


Figure 6.3: Final predicted output (i.e., attack) is same but probability distributions over predicted classes have significantly changed for the mutated program

After performing the statistical analysis on the probability distributions over predicted

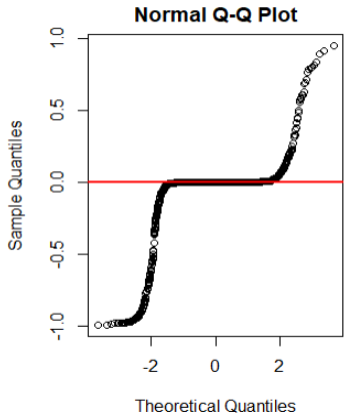

Figure 6.4: Q-Q plot

classes, results in Table 6.3 and Table 6.4 show that we have strong evidence to reject $H_0$ for all the classes (except for class2 in App#1) for both the classifiers under test (p-value $< 0.05$). Rejection of $H_0$ means that the difference is significant and there is likely some bug in the system, hence the buggy behavior is said to be detected. It is important to note that the mutated program versions under investigation produce the same high accuracy as the original program, knowing the fact that they represent the faulty implementation of the programs under test. Such mutants may otherwise be difficult to detect and can mislead if the software developers solely rely on observing the accuracy of the ML model in a new release. **Therefore, we conclude that high accuracy does not necessarily mean that the program is correct/free of bugs, and the proposed statistical hypothesis testing technique is effective in the detection of buggy behavior produced by such high accuracy producing program versions, which answers RQ4.1 of RG4**.

***RQ4.2 (targeting RG4): Is the proposed ML-based approach effective and which ML model is more suitable for the problem under investigation?***

After answering RQ#4.1 and finding that the difference between the probability distribution over predicted classes is statistically significant, the next step is to use this knowledge for developing the ML models, so that the 'buggy' and 'non-buggy' behavior

Table 6.3: Wilcoxon signed rank test results for App#1 ($\alpha = 0.05$)

| **Class Label** | **p-value** | **Reject $H_0$** | **Buggy behavior Detected?** |
|---|---|---|---|
| Class1 (Normal) | $< 0.0001$ | Yes | Yes |
| Class2 (Attacker) | 0.63 | No | No |
| Class3 (Victim) | $< 0.0001$ | Yes | Yes |

Table 6.4: Wilcoxon signed rank test results for App#2 ($\alpha = 0.05$)

| **Class Label** | **p-value** | **Reject $H_0$** | **Buggy behavior Detected?** |
|---|---|---|---|
| Class1 (Attack) | $< 0.0001$ | Yes | Yes |
| Class2 (Benign) | $< 0.0001$ | Yes | Yes |

for the prioritized test inputs can be learned and then using this knowledge to predict the faulty behavior in the new release (using the same prioritized test inputs). Table 6.5 and Table 6.6 shows the performance of Random Forest, SVM (with RBF kernel) and k-NN algorithms on App#1 and App#2 data sets respectively. We used a fixed random seed to make sure that the data set is split in the same way for all the algorithms, so that, every algorithm is trained and evaluated on the same data instances. The evaluation metrics used to evaluate the performance of models include accuracy, precision, recall, and the F1-score. The results obtained show that for both the classifiers under test, random forest outperforms, having higher scores for all four measures. For App#1, SVM and k-NN provide satisfactory performance but not very well for App#2. Apart from that, SVM attains a lower F1-score than k-NN for App#2 but performs better than k-NN for App#1. We also observed that for the given data set, even the best identified k-NN model (using 10-fold cross-validated grid-

search method) is over-fitting, thus may not be an appropriate choice for predicting buggy behavior in App#2. Based on the results shown in Table 6.5 and Table 6.6, we conclude that overall, the proposed ML models have performed well in extracting the hidden patterns of 'buggy' and 'non-buggy' program versions. Furthermore, among the proposed models, random forest outperforms and attains higher performance (high accuracy and F1-score) than SVM and k-NN. **Hence, it answers our RQ4.2 (of RG4) that the proposed ML based approach is effective in predicting the faulty behavior in the programs under test (especially for App#1), and among the proposed models, Random Forest achieves the best performance for both the NN based N-IDSs under test**.

Table 6.5: Performance Report on App#1 Data set

| **Classifier** | **Accuracy** | **Precision** | **Recall** | **F1** |
|---|---|---|---|---|
| SVM | 0.87 | 0.86 | 0.88 | 0.87 |
| Random Forest | **0.93 ± 0.0** | **0.93 ± 0.0** | **0.93 ± 0.001** | **0.93 ± 0.0** |
| k-NN | 0.84 | 0.83 | 0.86 | 0.85 |

Table 6.6: Performance Report on App#2 Data set

| **Classifier** | **Accuracy** | **Precision** | **Recall** | **F1** |
|---|---|---|---|---|
| SVM | 0.60 | 0.67 | 0.47 | 0.55 |
| Random Forest | **0.83 ± 0.004** | **0.86 ± 0.008** | **0.81 ± 0.008** | **0.83 ± 0.005** |
| k-NN | 0.59 | 0.62 | 0.54 | 0.58 |

***RQ4.3 (targeting RG4): Does the addition of metadata features increase the performance of proposed ML models?***

The results in Table 6.6 show that for App#2, the SVM, and k-NN classifiers do not achieve very good performance. This motivates us to conduct one more experiment and to add metadata features extracted during training of both types of programs (original and mutated ones), expecting the addition of new informative features to increase the performance of models. The two metadata features extracted include *'training time taken in minutes'*, and *'training time taken in seconds'*. We are interested in observing whether the addition of metadata features can further discriminate both classes (high accuracy producing buggy code and non-buggy code) and whether they can help to enhance the ML models' performance. This model can be used in a scenario where the developer has submitted the change documentation mentioning that the change is neither related to enhancing/reducing the model complexity nor significantly expanding the training set size. An example can be: changing the Min-Max Normalization method to z-Score Normalization. Such change should not have a significant impact on either enhancing or reducing the model's training time. However, if the model mistakenly becomes overly or insufficiently complex, it will have an impact on the model training time which can be a useful feature in capturing the information about the trained NN-based classifier under test. Based on the results obtained on a new data set having metadata features, if the results in Table 6.7 are compared with Table 6.6, it can be seen that the performance of all the models has significantly improved. In comparison to k-NN, SVM attains high accuracy and precision but has a low recall value. However, random forest still outperforms and achieves the best predictive power. **Therefore, we conclude that the addition of metadata features is useful in enhancing the ML models' performance and can play a handy role in developing accurate ML models, which answers RQ4.3 (of RG4)**.

Table 6.7: Performance Report on App#2 Data set After Adding Metadata features

| **Classifier** | **Accuracy** | **Precision** | **Recall** | **F1** |
|---|---|---|---|---|
| SVM | 0.76 | 0.96 | 0.54 | 0.69 |
| Random Forest | **0.90 ± 0.0** | **0.96 ± 0.0** | **0.93 ± 0.01** | **0.94 ± 0.01** |
| k-NN | 0.70 | 0.72 | 0.65 | 0.68 |

## Threats To Validity

In this study, we do not aim to identify the individual injected mutants, instead we are interested in identifying whether, in general, the program behavior can be characterized as buggy or not. This is the reason that we treat the behavior produced by all the mutated versions as 'buggy' and by the original program as 'non-buggy'. The proposed approach serves its purpose well in meeting this objective. Nevertheless, we identify the following threats to validity:

- Although we propose only a single MR, it worked sufficiently well to show the applicability of the MT approach, (i) in extending the prioritized test set with additional tests without incurring additional labeling costs, and (ii) using the prioritized test inputs (both source and follow-up data) to record and observe the behavior of the classifiers under test over a diverse set of inputs.

- Due to time and resource constraints, we are only able to find and inject 4 valid high accuracy producing mutants in the NN-based N-IDSs under test. Although the number of mutants is small, it fulfills the objective and applicability of the proposed approach in detecting their buggy behavior that may otherwise be difficult to detect by a software developer. Second, the main reason to use only the high accuracy producing mutants is that when the software developer sees the low accuracy of the model, the developer

gets an alert indicating that some problem in the model requires further investigation. However, when the model produces high accuracy, that may be skeptical and mislead the developer, assuming that everything is fine and model is ready to be deployed in the production environment.

- It can be argued that some of the mutants will produce a different probability class distribution than the original/correct program version, and hence it can be easy to identify them. This poses a potential threat to the construct validity of this study, however, this observation is subjective and is error prone. Also, it may not hold true without providing some solid statistical evidence. Second, in order to automate the testing process, there must exist a systematic approach to automatically detect the buggy behavior in such high accuracy producing faulty program versions.

- The NN-based N-IDSs under test belong to the class of fully connected neural networks, thus generalizing the results to other types of DNNs e.g. CNNs, RNNs, would be speculative and reveals a threat to external validity.

## Conclusion And Future Work

The high labeling cost associated with data instances and testing computationally complex machine learning classifiers that have stochastic behavior are both challenging and resource-intensive tasks. To address the first challenge, we propose a Metamorphic Relation (MR) that effectively addresses the data generation/labeling problem without any need to label the new test instances manually. To target the second challenge, we propose a statistical hypothesis testing (for detection) and machine learning-based approach (for prediction) of faulty behavior in NN-based classifiers using the prioritized test inputs. The proposed statistical and ML-based approach is applicable for testing NN-based classifiers in an environment where the random seeds can not be fixed for getting deterministic results

and checking the program correctness. The usefulness of our proposed approach is shown in detecting and predicting buggy behavior in two NN-based network intrusion detection systems i.e., one based on Shallow NN, whereas the other is a DNN-based classifier. The results obtained show that, (i) the proposed statistical hypothesis testing is effective in detecting the buggy behavior, and (ii) among the proposed ML models, random forest outperforms and achieves better performance than SVM and k-NN algorithms.

In this paper, we propose a sample MR to show the effectiveness of MT in addressing the data generation/labeling problem. In future research work, we intend to propose more effective MRs that can be used to generate additional representative data; thus reducing the labeling cost further and enabling organizations to check program correctness on large input scenarios for enhancing their trust. It will also be interesting to explore what other types of useful features can be added to enhance the performance of ML models and to show the applicability of the proposed approach in general. The results obtained in this preliminary study are encouraging, and a comprehensive study on applying the proposed approach on a larger number of mutants is in progress.

MT4UML: METAMORPHIC TESTING FOR UNSUPERVISED MACHINE LEARNING

Contribution of Authors and Co-Authors

Manuscript in Chapter titled 'MT4UML: Metamorphic Testing for Unsupervised Machine Learning'

Author: Faqeer ur Rehman

Contributions: Problem identification and proposing solution, conducting experiment, manuscript writing, creating tables and figures. Primary writer

Co-Author: Dr. Clemente Izurieta

Contributions: Contribution in manuscript editing/writing, provided feedback, guidance and advice.

Manuscript Information Page

Faqeer ur Rehman and Dr. Clemente Izurieta

IEEE Swiss Conference on Data Science (SDS)

Status of Manuscript:
___ Prepared for submission to a peer-reviewed journal
___ Officially submitted to a peer-reviewed journal
___ Accepted by a peer-reviewed journal
_×_ Published in a peer-reviewed journal

IEEE
14 October 2022
10.1109/SDS54800.2022.00012

## Abstract

One of the advantages of using unsupervised machine learning algorithms is that they don't need labeled data; thus, ultimately saving higher labeling costs for an organization. However, the computational complexity and large input space put these algorithms into the category of non-testable programs, which also suffer from the oracle problem. One popular testing approach, borrowed from the Software Engineering (SE) domain is the Metamorphic Testing (MT) technique that has been proven to be an effective approach in alleviating the oracle problem in testing such non-testable programs. We take advantage of this MT approach to make some insightful contributions that include: i) proposing a broader set of 22 Metamorphic Relations (MRs) for assessing the behavior of the K-means clustering algorithm (a prototype-based approach) and the Agglomerative clustering algorithm (a hierarchy-based approach), provided by the leading scikit-learn Python library, ii) providing a detailed analysis/reasoning to show how the proposed MRs can be used to target both the verification and validation aspects of testing the clustering algorithms under investigation, and iii) showing that verification of MRs using multiple criteria is more beneficial than relying on using just a single criterion (i.e., clusters assigned). We further applied the proposed approach to test an open source customer segmentation application and the results obtained show that, i) 10 MRs have been violated for both the K-means and Agglomerative clustering algorithms, and ii) in comparison to K-means, the Agglomerative clustering algorithm is highly susceptible to small changes in inputs and may not offer a better alternative to scenarios captured by the violated MRs.


## Introduction

Machine Learning (ML) can broadly be classified into supervised and unsupervised machine learning algorithms. Supervised machine learning learns patterns from the labeled

data, whereas, there is no class label available for the data points when the problem under investigation falls under the unsupervised machine learning category. Some of the widely used unsupervised machine learning algorithms include K-means clustering (a prototype-based approach) and Agglomerative clustering (a hierarchy-based approach). These algorithms can be used to address a wide range of real world problems i.e., customer segmentation, document clustering, clustering DNA patterns, recommendation systems, and anomaly detection. It is thus important that such applications are tested properly to ensure their quality before moving them to production environments. Further, one should also be aware of the possible changes in clustering results when the data itself undergoes changes in the future. However, similar to supervised ML algorithms, one of the challenges faced in testing unsupervised ML algorithms is their complexity and their exposure to the oracle problem. The oracle is a mechanism that a software tester uses to verify the output of the program under test. When the oracle is not available (or is available but infeasible to apply) we call a program suffering from the oracle problem.

Software Engineering for Machine Learning (SE4ML) is an emerging research area that focuses on applying SE best practices and methods for better development, testing, operation, and maintenance of ML-based systems [12, 38, 45, 46]. Our focus in this work is on the testing aspect of unsupervised ML algorithms and how a traditional software testing approach i.e., Metamorphic Testing (MT) can be utilized to perform better quality assurance from both the verification and validation perspective.

MT is considered an effective testing strategy in alleviating the oracle problem in testing both type of supervised and unsupervised ML algorithms. In MT, Metamorphic Relations (MRs) are proposed to test the program under test. Each MR is composed of a source test case and a follow-up test case. An MR is said to be violated if the result obtained for the source test case is different from the follow-up test case. The MRs can be used to target either i) the necessary characteristics (related to implementation), or ii)

expected characteristics (related to user expectations) of the program under test. MT is different from using the classical evaluation methods i.e., residual sum-of-squares, silhouette coefficient, Davies–Bouldin index, etc. (frequently used to evaluate the clustering results of different algorithms) in the sense that MT is a testing technique, whereas, these evaluation methods aim to identify the algorithm more suitable for the problem under investigation. It is important to note that much of the research work has focused on utilizing the power of MT for testing supervised ML algorithms [48, 56, 60, 74, 76] but much less work has been done in using MT for testing unsupervised algorithms [84, 85]. Prior work (related to testing unsupervised ML algorithms) uses MT to test clustering algorithms, provided by the WEKA tool [81], only from the validation perspective. This motivates us to take MT one step further and utilize its power to test some widely used clustering algorithms (i.e., K-means clustering and Agglomerative clustering), provided by a popular and widely used Python library i.e., scikit-learn [55] from both the verification and validation perspective.

The following presents the main contributions made in this paper [62]:

- We propose an MT based approach for verification and validation of two popular clustering algorithms, provided by the leading Python library known as scikit-learn. These includes K-means (a prototype-based approach) and Agglomerative clustering (a hierarchy-based approach) algorithms.

- We propose 22 MRs to assess the behaviour (from both the user's and developer's/implementation perspective) of the clustering algorithms under test.

- The proposed MRs are further analyzed, necessary reasoning is provided, and MRs are then categorized to show whether each of those MRs targets the verification or the validation aspect of testing the two algorithms under investigation.

- The effectiveness of the proposed approach is demonstrated by applying it to testing

an open source customer segmentation application [1]. The results show that among the proposed MRs, 10 MRs are violated for both the K-means and the Agglomerative clustering algorithm.

## Related Work

The MT technique has been shown to be an effective approach in alleviating the oracle problem in computationally complex machine learning based classifiers [48, 56, 60, 74, 76]. To the best of our knowledge we are able to find only two research papers in which MT has been utilized to test unsupervised clustering algorithms [84, 85], which is equally the motivator for this work and for making some beneficial research contributions. Yang et al. [85] proposed 7 MRs to test the K-means algorithm (in WEKA tool) that target the algorithm's correctness from a user perspective (validation) to check whether the user expectations from the algorithm are satisfied or not. Their results show that two of the MRs are violated but this does not necessarily mean that there is some implementation defect in the algorithm under test. Xie et al. [84] proposed 11 generic MRs that assess and validate the characteristics of different clustering algorithms from a user perspective. The authors conducted an experiment to test 6 clustering algorithms (provided by WEKA) and compared them using the proposed MRs. This research helps end-users (non-technical users) coming from diverse fields such as bioinformatics, finance, and electrical engineering to choose a specific type of algorithm from a large set of available algorithms that can best fit their needs. However, the following are the limitations we have found in their work:

- The proposed approaches only serve validation purposes, and only check whether the algorithms under test meet the user expectations.

[1] `https://github.com/matifkhattak/MT4UML`

- The proposed MRs only target the algorithms provided by the WEKA tool. It is equally worth exploring to test the behavior of other notable clustering algorithms provided by widely used Python libraries i.e., Scikit-learn. It will not only help the end-users in choosing the most suitable clustering algorithm but it will also help in selecting the right ML library for their problem under investigation.

- The proposed approaches use synthetic 2D data (i.e., not real data). It is worth exploring whether the proposed MRs are effective in testing the models that use multi-dimensional real-world data sets as well.

## Our Approach

Our approach for testing the K-means and Agglomerative clustering algorithms is based on the Metamorphic Testing (MT) technique [20]. In **K-means algorithm**, the following equation is used for calculation of new centroids [27], in which $c_i^{(t+1)}$ represents the $i^{th}$ new centroid found, and $x_j$ represents the $j^{th}$ instance (where $j = 1, 2, ..., n$) belonging to the cluster $C_i$.

$$c_i^{(t+1)} = \frac{1}{|C_i^{(t)}|} \sum_{j=1}^{n} x_j \tag{7.1}$$

In **Agglomerative clustering**, the following equation represents the average linkage method used to merge two similar clusters [84]:

$$d(C_i, C_j) = \frac{1}{|C_i||C_j|} \sum_{x_r \in C_i} \sum_{x_s \in C_j} d(x_r, x_s) \tag{7.2}$$

where, $d(C_i, C_j)$ represents the distance between cluster $C_i$ and cluster $C_j$, $x_r$ represents the $r^{th}$ data point (where r = 1,2,...,n) belonging to the cluster $C_i$, and $x_s$ represents the $s^{th}$ data point (where s = 1,2,...,n) belonging to the cluster $C_j$.

In the rest of this section, we define the proposed MRs along with the analysis/reasoning to show how the proposed MRs target both the *verification* and *validation* aspect of testing the clustering algorithms under investigation.

Proposed Metamorphic Relations for Unsupervised Algorithms

We provide a set of 22 MRs that can be broadly classified into 14 categories. Each of the proposed MRs either targets the *verification* or the *validation* aspect of testing the clustering algorithms under test. The MRs targeting the *verification* aspect aim to check whether the algorithms under test adhere to the necessary characteristics (from the implementation perspective) expected from the algorithms, whereas, the MRs targeting the *validation* aspect aim to check whether the algorithms under test meet the general user expectations or not. This large set of MRs are not just limited to the two algorithms under investigation (provided by the scikit-learn library), instead, they can also be used by naive users, developers and professionals to assess any clustering algorithm(s) they are interested in using for their respective problems. For better support and guidance, as shown in Tables 7.1 and 7.2, we further provide the verification and validation analysis and reasoning for the proposed MRs. This analysis gives users a better idea in understanding the algorithms and in choosing the appropriate solution that best fits their needs.

**MR1 - Duplication of Data Instance(s):**

**MR1.1 - Duplication of single instance:** For a given source input $s$, with associated data instances assigned to clusters $c_i$ (where $i = 1, 2, 3, ..., n$), we denote the output as $O_s$. If we duplicate a single instance in the follow-up input $f$, the output $O_f$ should remain consistent i.e., $O_s = O_f$.

**MR1.2 - Duplication of multiple instances:** For a given source input $s$, with associated data instances assigned to clusters $c_i$ (where $i = 1, 2, 3, ..., n$), we denote the

output as $O_s$. If we duplicate multiple instances (i.e., each belonging to a different cluster) in the follow-up input, the output $O_f$ should remain unchanged i.e., $O_s = O_f$.

**MR1.3 - Duplication of cluster centroids:** For a given source input $s$, we denote a set of centroids found as $t_i$ (where $i = 1, 2, 3, ..., n$). If we duplicate these centroids in the follow-up input, the output $O_f$ should remain unchanged i.e., $O_s = O_f$.

**MR2 - Data Standardization:** If the existing standardized data is once again standardized, the output for both the source and follow-up inputs should remain the same i.e., $O_s = O_f$.

**MR3 - Duplication of Features:** For a given source input $s$, we denote the output as $O_s$. For the follow-up input, if new features are added by duplicating existing features, the output $O_f$ should remain unchanged i.e., $O_s = O_f$.

**MR4 - Removal of Instance(s):**

**MR4.1 - Removal of instance from one cluster:** For a given source input $s$, we denote the result as $O_s$. If an instance from a cluster $c_i$ is removed for the follow-up input, it should not have any effect on changing the results for the remaining inputs, so the output $O_f$ should remain the same i.e., $O_s = O_f$.

**MR4.2 - Removal of instance from different clusters:** For the follow-up input, if an instance from each of the clusters $c_i$ (found during the source execution, where $i = 1, 2, 3, ..., n$) is removed, the output should remain consistent.

**MR4.3 - Removal of multiple instances from a single cluster:** For the follow-up input, if some random number of $n$ instances are removed from a single cluster $c_i$, it should not have any effect on changing the results for the remaining inputs.

**MR5 - Addition of Uninformative Attribute:** For the follow-up input, if a new uninformative feature (i.e., a feature having the same value for all the instances) is added, the output should remain unchanged.

**MR6 - Deterministic Output Across Multiple Runs:** If a new data point is added, it should be assigned to the same cluster no matter how many times the algorithm under test is executed, i.e., the output for the execution at time $time_i$ (where i $= 1, 2, 3, ..., n$) and $time_{i+1}$ should remain consistent for both the source and follow-up inputs.

**MR7 - Shifting Features With constant *k*:** For the follow-up input, if the feature(s) for all the instances are shifted with some constant $k$, the output should remain the same for both the source and follow-up inputs.

**MR8 - Scaling Features With Constant *k*:** If the feature(s) for all the instances are scaled with some constant $k$, the output should remain unchanged for both the source and follow-up inputs.

**MR9 - Replacement of Instance(s):**

**MR9.1 - Replacement of single instance:** If a single instance belonging to a cluster $c_i$ is replaced with some other instance $x$ (belonging to the same cluster $c_i$), it should not have any impact on changing the clustering results i.e., the output $O_f$ should remain the same for both the source and follow-up inputs.

**MR9.2 - Replacement of multiple instances:** If multiple instances belonging to a cluster $c_i$ are replaced with some other instance $x$ (belonging to the same cluster), the output $O_f$ should remain consistent for both the source and follow-up inputs.

**MR9.3 - Replacement of all instances:** If all the instances belonging to a cluster $c_i$ are replaced with some other instance $x$ (belonging to the same cluster), the output $O_f$ should remain consistent for both the source and follow-up inputs.

**MR10 - Changing the Location of Features:** If we change the order of features, the clustering result should remain unchanged for both the source and follow-up inputs.

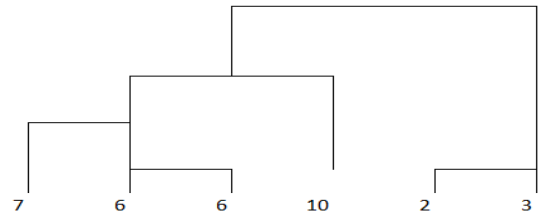


Figure 7.1: Agglomerative Clustering Example

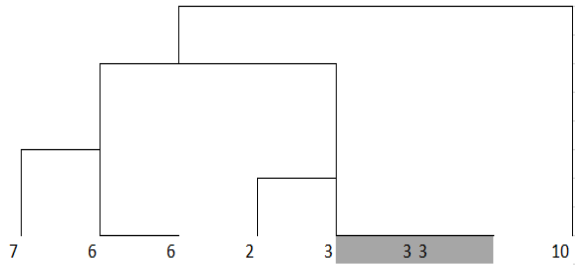


Figure 7.2: MR1 for agglomerative clustering: Added 3 as a duplicate instance

**MR11 - Adding an Informative Attribute:** For a given source input $s$, we denote the identified set of clusters as $C$=$\{c_1, c_2, ..., c_n\}$. For the follow-up input, if a new attribute whose value is strongly associated with each of the clusters i.e., value $x_1$ with $c_1$, $x_2$ with $c_2$,..., and $x_n$ with $c_n$, is added to the original data instances, the clustering result should remain the same for both the source and follow-up inputs.

**MR12 - Rows Transformation:**

**MR12.1 - Reversing the order:** If we reverse the order of data points/rows, the clustering result should remain consistent for both the source and follow-up inputs.

**MR12.2 - Random shuffling:** If we randomly shuffle the order of data points/rows, the clustering result should remain unchanged for both the source and follow-up inputs.

**MR13 - Reflection Transformation:** For a given source input $s$, we denote the output as $O_s$. For the follow-up input, if we multiply all the features with -1 (performing the data reflection), the output $O_f$ should remain the same i.e., $O_s = O_f$.

**MR14 - Addition of New Instance(s):**

**MR14.1 - Addition of instance with informative attributes:** If we increase the cluster $c_i$ density by adding a new data-point(s) in the middle of two existing data points (i.e., instance x and y belonging to the same cluster $c_i$), it should not have any effect on changing clustering results, i.e., the output should remain consistent for both the source and follow-up inputs i.e., $O_s = O_f$.

**MR14.2 - Addition of instance with uninformative attributes:** If we increase the cluster $c_i$ density by adding a new data-point(s) such that all the features have 0 values in them, it should not have any effect on the clustering results, i.e., the output should remain consistent for both the source and follow-up inputs.

Table 7.1: k-Means Algorithm: Verification (VR) And Validation (VD) Analysis For The Proposed MRs

| # | VR? | VD? | Reasoning |
|---|---|---|---|
| MR1 | × | ✓ | For the follow-up input, if we add a duplicate instance(s) to any of the clusters, this will result in different cluster centroid(s) (calculated using Equation 7.1) which may cause the original data points to get assigned to different clusters; thus, changing the output for the follow-up input. It is important to note that this is how (as shown in Equation 7.1) the centroid/mean calculation is implemented in the K-means algorithm under test, which ultimately means that the violation of this MR can not be characterized as violating the necessary characteristics (related to implementation) of the algorithm under test. Therefore, this MR can not be used for **verification** purposes (because its violation can not be characterized as the bug in the implementation) but can be used for **validation** purposes (because this is what the user's general expectation would be from this algorithm).<br>***Note**: Readers can use the same reasoning for the verification and validation aspect of testing the algorithms under test for rest of the proposed MRs.* |
| MR2 | ✓ | ✓ | For the follow-up input, if we re-apply the standardization step, i) it will not change the mean and variance of the data points, and ii) it will maintain the same distance among the data points (similar to the source-input); thus, it should not not change the results. |

| | | | |
|---|---|---|---|
| MR3 | × | ✓ | For the follow-up intput, if we add a duplicate attribute(s) to the original data points, they may have a strong influence on changing the distance between the data points and their existing centroids; thus, assigning the data points to different clusters. An example is provided (inside excel sheet available in GitHub repo), where, a violation of this MR can be seen. |
| MR4 | × | ✓ | For the follow-up input, if we remove any instance(s), Equation 7.1 will result in the calculation of different centroids which ultimately may lead to changing the final output i.e., data points assigned to different clusters. Since this violation can not be characterized to the wrong implementation, it can only be used for validation purposes. |
| MR5 | ✓ | ✓ | The addition of uninformative attributes (e.g., 0 or some other constant) will not change the existing relationship between the data points and will also maintain the same relationship between the data points and the initial centroids. Therefore, the output for the follow-up input should remain the same. |
| MR6 | ✓ | ✓ | Running the algorithm at different times (keeping the initial centroids the same) should not have any effect on how the centroids are calculated. Further, as there is no change made in the data points, the output should remain the same. If the output changes, it means that there is some implementation bug in the algorithm under test. |

| | | | |
|---|---|---|---|
| MR7 | ✓ | ✓ | If we shift all the features with some constant $k$, it will maintain the same distance between the data instances. So, Equation 7.1 will produce the same centroids as the one found during source execution; thus, not changing the final output. Let $x$ be the centroid, $y$ be the data point and the distance $d(x, y)$ found between them during source execution is $z$. Now, if we shift the features of both $x$ and $y$ with some constant (i.e., $k$) then the distance between them would be $d(x, y) = \sqrt{((x + c) - (y + c))^2} => \sqrt{(x - y)^2} => x - y => z$ (i.e., will remain the same). Therefore, the output for both the source and follow-up inputs should remain consistent. |
| MR8 | ✓ | ✓ | If we scale all the features with some constant $k$, it will maintain the same relationship between the data points and the centroids. Let $x$ and $y$ be the two data points and during the source execution their relationship to the centroid $c$ is $x - c < y - c$. During the follow-up execution, after scaling the features with constant $k$ i.e., $k(x - c) < k(y - c) => x - c < y - c$, the relationship (i.e., greater than, less than, and equal) remains the same. Therefore, if the output for the follow-up input changes, that would suggest that there is some bug in the algorithm under test. |
| MR9 | × | ✓ | If we replace any of the instances with some other instance (belonging to the same cluster), it may result in the calculation of different centroids (as per Equation 7.1); thus an instance $x_i$ (where $i = 1, 2, .., n$) may get assigned to a different cluster. |

| | | | |
|---|---|---|---|
| MR10 | ✓ | ✓ | For the follow-up input, if we change the location of features, it will not have any affect on the relationship between the data points and calculation of centroids (using Equation 7.1). So, the output should remain the same. |
| MR11 | × | ✓ | For the follow-up input, if we add an informative attribute to each of the clusters instances, it may result in changing the centroids that can assign the instances to different clusters; thus, leading to a different final output. |
| MR12 | ✓ | ✓ | For the follow-up input, if we change the order of rows/data points, it will not have any affect on existing relationship between the data points and will lead to the calculation of same centroids (similar to the one found during source execution). Thus, the output for the source and follow-up inputs should remain consistent. |
| MR13 | ✓ | ✓ | For the follow-input, if we apply reflection transformation, the distance between the data points will remain the same thus leading to the calculation of same centroids (using Equation 7.1). This should result in consistent output for both the source and follow-up inputs. |
| MR14 | × | ✓ | If we add a new instance(s) to any of the clusters, this addition of new instance(s) may result in the change of centroids (different from the one found during source execution); thus, changing the final output. |

Table 7.2: Agglomerative Clustering Algorithm: Verification (VR) And Validation (VD) Analysis For The Proposed MRs

| # | VR? | VD? | Reasoning |
|---|---|---|---|
| MR1 | × | ✓ | For the follow-up input, if we add a duplicate instance(s) to any of the clusters, this may change the average distance between the two clusters (calculated using Equation 7.2), thus ending up with changing the clusters for the original data points. For understanding purposes, suppose in Figure 7.1, the dendogram for the source input is cut to obtain the two clusters. The data points 7,6,6,10 will be assigned to one cluster, whereas, the data points 2,3 will be assigned to a second cluster. Now, if we duplicate the data point 3 and cut the dendogram to obtain the two clusters, it can be seen in Figure 7.2 that the original data points 7,6,6,2,3 are now assigned to one cluster, whereas, the data point 10 is assigned to the second cluster; thus, changing the output for the follow-up inputs. |
| MR2 | ✓ | ✓ | Same reasoning as provided for MR2 in Table 7.1 |
| MR3 | × | ✓ | For the follow-up intput, if we add a duplicate attribute(s) to the original data points, they may have a strong influence on changing the average distance between the clusters; thus, changing the overall result. An example is provided (inside excel sheet available in GitHub repo), where, a violation of this MR can be seen. |

| | | | |
|---|---|---|---|
| MR4 | × | ✓ | For the follow-up input, if we remove an instance(s), Equation 7.2 will result in changing the average distance between the clusters. As an example, in Figure 7.1, if we remove the data point '2', the data point '3' (instead of data point '10') will get assigned to the cluster '7,6,6'. Now, if the dendogram is cut to obtain the two clusters, one cluster will have data points '7,6,6,3', whereas, the second one will have only '10', thus changing the clustering result for the follow-up inputs. |
| MR5 | ✓ | ✓ | The addition of uninformative attribute (e.g., 0 or some other constant) will neither change the existing relationship between the data points nor the average distance between the clusters. Therefore, the output for the follow-up input should remain the same. |
| MR6 | ✓ | ✓ | Running the algorithm at different times should not have any affect on how the average distance between the clusters is calculated. Apart from that, as there is no change made in the data points, so the output should remain consistent. |
| MR7 | ✓ | ✓ | If we shift all the features with some constant $k$, it will maintain the same distance between all the data instances. So, Equation 7.2 will result in merging the same clusters that were merged during the source execution. As an example, let $x$ and $y$ be the two data-points merged together during the source execution. Now, if we shift the features of both the $x$ and $y$ with some constant 'c' then the distance between them i.e., $d(x,y) = \sqrt{((x+c)-(y+c))^2} => \sqrt{(x-y)^2}$ , would remain similar to the one calculated during the source execution. Therefore, the output for both the source and follow-up inputs should remain the same. |

| | | | |
|---|---|---|---|
| MR8 | ✓ | ✓ | If we scale all the features with some constant $k$, it will maintain the same relationship (i.e., greater than, less than, and equal) between the data points. As an example, let suppose that we have three data points $x, y$, and $z$ that we are interested to group them into two clusters. During the source execution, let $x$ and $y$ are merged together to form one cluster, and $z$ in another cluster. The current relationship between them is $x-y < z-y$. During the follow-up execution, after scaling the features with a constant $k$ i.e., $k(x-y) < k(z-y) => x-y < y-z$, the relationship remains the same. Therefore, the output should remain consistent for both the source and follow-up executions. |
| MR9 | × | ✓ | For the follow-up input, if we replace any of the instances with some other instance (that belongs to the same cluster), it may change the average distance between them (calculated using Equation 7.2) which can result in assigning the original input to different clusters. As an example, as shown in Figure 7.1, if in cluster#2 (which contain data points 2 and 3), we replace the instance 2 with instance 3, it will assign them to the cluster#1 which contains the data points 7,6,6. Now if the dendogram is cut to obtain the two clusters, one cluster will have the data-points 7,6,6,3,3, whereas, the other will have only 10; thus, the original data-point (which is 3) has been assigned to the cluster#1 (instead of cluster#2). This will result in violation of this MR. |

| | | | |
|---|---|---|---|
| MR10 | ✓ | ✓ | For the follow-up input, if we change the location of features, it will not have any affect on the relationship between the data points and calculation of average distance between the clusters (calculated using Equation 7.2). So, the output should remain unchanged for both the source and follow-up executions. |
| MR11 | ✓ | ✓ | If we add an informative attribute such that it is strongly associated with each of the clusters, it will not change the existing relationship between the data points assigned to each clusters. As an example, let suppose that the dendogram for the source execution (as shown in Figure 7.1) is cut to form two clusters, one cluster will have the data points 7,6,6,10, whereas, the other cluster will have the data points 2,3. Now, for the follow-up execution, if we add a new informative attribute which has the value 3 for all the instances in cluster#1 i.e., (7,3),(6,3),(6,3),(10,3) and value 4 for all the instances in cluster#2 i.e., (2,4),(3,4), it will not change the existing relationship between the clusters; thus, the output should remain the same. |
| MR12 | ✓ | ✓ | For the follow-up input, if we change the order of rows/data instances, it will not have any affect on the way the calculation is made (using Equation 7.2) to merge the two clusters. Thus, the output for both the source and follow-up inputs should remain the same. |
| MR13 | ✓ | ✓ | For the follow-input, if we apply reflection transformation to all data points, the distance between them will remain the same; thus, leading to the identification of the same clusters found during source execution. |

| | | | |
|---|---|---|---|
| MR14 | × | ✓ | If we add a new instance(s) to any of the clusters, this addition of new instance(s) may result in change of average distance between the clusters; thus, changing the final output. An example is provided (inside excel sheet available in GitHub repo), where, a violation of this MR can be seen. |

Experimentation and Evaluation

To check the effectiveness of the proposed approach, we have selected an open source customer segmentation application that uses a real world multi-dimensional data-set [2]. The selected application uses K-means and Agglomerative clustering algorithms, provided by the leading Python library known as scikit-learn. It is worth mentioning that the proposed MRs are not just limited to this single application, instead, they can be used to test clustering algorithms in other domains as well (i.e., document clustering, clustering DNA patterns, and anomaly detection) in which the term 'data point/data instance' will represent either the document instance, DNA sequence, or the network traffic instance respectively.

In K-means, if the centroids are selected randomly, it will produce different results which can not be characterized as a violation of the MR. Therefore, we initialized the K-means algorithm with fixed centroids to make sure that it is converging to the same point for multiple iterations; thus, the focus is placed on testing the characteristics of the algorithms under test using the proposed MRs.

Table 7.3 summarizes the results obtained for both clustering algorithms under test. For testing the K-means algorithm, we present the results and verify the proposed MRs using multiple criteria e.g., whether the i) clusters, ii) centroids, and iii) nearest point to

[2] `https://github.com/matifkhattak/MT4UML`

Table 7.3: Results of testing k-Means and Agglomerative clustering algorithms

| | K-Means | | | | Agglomerative Clustering | |
|---|---|---|---|---|---|---|
| MR# | Same cluster assigned? | Centroids same? | Nearest point to centroid(s) same? | Violation? (Violation rate) | Same cluster assigned? | Violation? (Violation rate) |
| 1.1 | × | × | ✓ | ✓(0.12%) | × | ✓(0.05%) |
| 1.2 | × | × | ✓ | ✓(0.10%) | × | ✓(98.53%) |
| 1.3 | ✓ | ✓ | ✓ | × | N/A | N/A |
| 2 | ✓ | ✓ | ✓ | × | ✓ | × |
| 3 | ✓ | ✓ | ✓ | × | ✓ | × |
| 4.1 | × | × | ✓ | ✓(0.14%) | × | ✓(0.09%) |
| 4.2 | × | × | ✓ | ✓(0.23%) | × | ✓(0.23%) |
| 4.3 | × | × | × | ✓(57.82%) | × | ✓(93.40%) |
| 5 | ✓ | ✓ | ✓ | × | ✓ | × |
| 6 | ✓ | ✓ | ✓ | × | ✓ | × |
| 7 | ✓ | ✓ | ✓ | × | ✓ | × |
| 8 | ✓ | ✓ | ✓ | × | ✓ | × |
| 9.1 | ✓ | × | ✓ | ✓ | × | ✓(2.14%) |
| 9.2 | ✓ | ✓ | ✓ | × | × | ✓(98.07%) |
| 9.3 | × | × | ✓ | ✓(0.02%) | × | ✓(99.28%) |
| 10 | ✓ | ✓ | ✓ | × | ✓ | × |
| 11 | ✓ | × | × | ✓ | ✓ | × |
| 12.1 | ✓ | ✓ | ✓ | × | ✓ | × |
| 12.2 | ✓ | ✓ | ✓ | × | ✓ | × |
| 13 | ✓ | ✓ | ✓ | × | ✓ | × |
| 14.1 | × | × | ✓ | ✓(0.05%) | × | ✓(0.28) |
| 14.2 | ✓ | × | × | ✓ | × | ✓(94.90%) |

each centroid, are the same for both the source and follow-up inputs. This is beneficial, because if we are unlucky in identifying the violation(s) for MRs using the first criterion, we hope to uncover them using the other criteria. For example, in Table 7.3, it can be seen that for some of the MRs e.g., MR9.1, MR11, and MR14.2, the clusters assigned to data instances are the same for both the source and follow-up executions (thus, if only the first criterion is used, the MR is said to be satisfied) but those MRs were violated for the other criteria (e.g., verifying the centroids, and the nearest point to each of the centroids), thus showing the usefulness of using multiple criteria to verify the MRs. This also opens a new research direction for researchers to explore the type of different criteria that can be used for verification of MRs (for different clustering algorithms) instead of relying only on comparing the final output (i.e., clusters assigned), which may mislead the results.

The results presented in Table 7.3 show the violated MRs and their comparison for both algorithms under investigation. Each of the violated MRs either implies the implementation faults in the algorithm (verification), or its deviation from the user expectations (validation). For both the K-means and Agglomerative clustering alogrithms, 10 (out of 22) MRs (which quantifies RM1) have been violated. We also show the violation rate (i.e., RM2) for each of the violated MRs, and it can be seen that the K-means algorithm shows a higher violation rate for MR4.3, whereas, the Agglomerative clustering algorithm has a higher violation rate for MR1.2, MR4.3, MR9.2, MR9.3, and MR14.2. **This answers the RQ5.1 (of RG5) that the proposed MRs are effective in testing the clustering algorithms under test**. To answer the second RQ, it can be seen that both of the algorithms under investigation show the violations for the same number of MRs. However, agglomerative clustering seems to be more sensitive to smaller changes because a small change is causing a higher violation rate among the violated MRs. **Therefore, we conclude that in comparison to agglomerative clustering, the K-means algorithm is more stable for the scenarios captured in the proposed MRs, which answers RQ5.2 (of RG5)**.

## Conclusion And Future Work

Similar to supervised ML algorithms, one of the challenges faced in testing unsupervised algorithms is that they also suffer from the Oracle problem. Software Engineering for Machine Learning (SE4ML) is an emerging research area that focuses on applying SE best practices and methods for better development, testing, operation, and maintenance of ML-based systems. Our contribution in this work focuses on testing some popular unsupervised ML algorithms (i.e., K-means and Agglomerative clustering algorithms, provided by the leading Python library 'scikit-learn') and investigate how the traditional software testing approach i.e., Metamorphic Testing (MT) can be utilized to perform better quality assurance from both the verification and validation perspective. We propose a broader set of 22 MRs

that both researchers and practitioners can take advantage of to assess the behaviour of the clustering algorithms under test from both the user's general expectation (validation) and from the implementation perspective (verification). For testing the K-means algorithm, we also propose multiple criteria that can be used for verification of the MRs. Our results show that both the algorithms under test exhibit violations (from the validation perspective) for 10 MRs, which implies that the behaviour of the algorithms deviates from the general user expectations. Further, in comparison to K-means, the Agglomerative clustering algorithm is highly susceptible to small changes in inputs and may not offer a better alternative to scenarios captured by the violated MRs.

In the future, and to improve the testing of agglomerative clustering based applications, we intend to develop new criteria that can be used to verify MRs, and which will ultimately help in building trust in using critical application algorithms. Second, to show the general applicability of the proposed MRs, we intend to utilize the proposed approach by testing a broad range of other clustering algorithms that are popular among both researchers and practitioners of the ML community.

# AN APPROACH FOR VERIFYING AND VALIDATING CLUSTERING BASED ANOMALY DETECTION SYSTEMS USING METAMORPHIC TESTING

## Contribution of Authors and Co-Authors

Manuscript in Chapter titled 'An Approach For Verifying And Validating Clustering Based Anomaly Detection Systems Using Metamorphic Testing'

Author: Faqeer ur Rehman

Contributions: Problem identification and proposing solution, conducting experiment, manuscript writing, creating tables. Primary writer

Co-Author: Dr. Clemente Izurieta

Contributions: Contribution in manuscript editing/writing, provided feedback, guidance and advice.

Manuscript Information Page

Faqeer ur Rehman and Dr. Clemente Izurieta

IEEE International Conference On Artificial Intelligence Testing (AITest)

Status of Manuscript:
___ Prepared for submission to a peer-reviewed journal
___ Officially submitted to a peer-reviewed journal
___ Accepted by a peer-reviewed journal
_×_ Published in a peer-reviewed journal

IEEE
26 September 2022
10.1109/AITest55621.2022.00011

## Abstract

An oracle or test oracle is a mechanism that a software tester uses to verify the program output. In software testing, the oracle problem arises when either the oracle is not available or it may be available but is so expensive that it is infeasible to apply. To help address this problem in testing machine learning-based applications, we propose an approach for testing clustering algorithms. We exemplify this in the implementation of the award-winning density-based clustering algorithm i.e., Density-based Spatial Clustering of Applications with Noise (DBSCAN). Our proposed approach is based on the 'Metamorphic Testing' technique which is considered an effective approach in alleviating the oracle problem. Our contributions in this paper include, i) proposing and showing the applicability of a broader set of 21 Metamorphic Relations (belonging to 14 different categories), among which 7 type of MRs target the verification aspect of testing the algorithm under test, and ii) identifying and segregating the MRs (by providing a detailed analysis) to help both naive and expert users understand how the proposed MRs target both the verification and validation aspects of testing the DBSCAN algorithm. To show the effectiveness of the proposed approach, we further conduct a case study on an anomaly detection system. The results obtained show that, i) different MRs have the ability to reveal different violation rates (for the given data instances); thus, showing their effectiveness, and ii) although we have not found any implementation issues (through verification) in the algorithm under test (that further enhances our trust in the implementation), the results suggest that the DBSCAN algorithm may not be suitable for scenarios (meeting the user expectations a.k.a validation) captured by almost 52.4% of violated MRs; which show high susceptibility to small changes in the dataset.

## Introduction

The use of machine learning algorithms is growing fast in a number of application domains i.e., document clustering, image processing, social network analysis, recommendation systems, customer segmentation, and anomaly detection. As these applications become pervasive in real-world environments, it becomes crucial to verify their correct behavior.

Software testing is a common approach used to test and verify the quality of software. However, one of the problems it faces is the *oracle problem*. An oracle is a mechanism that a software tester uses to verify the program under test. Software is executed for a given test case and the output produced is compared with the expected outcome. A program is said to be buggy if the output produced by the program does not match the expected output. In real-life scenarios, the oracle may not be available or it may exist but is so expensive that it is infeasible to apply.

It always remains a challenging task to test unsupervised machine learning applications in the absence of a test oracle (i.e., in the form of ground truth/class label). To help address improving the quality of such machine learning-based applications, this work focuses on testing the following award-winning unsupervised clustering algorithm (at the leading data mining conference, ACM SIGKDD [70]): Density-based spatial clustering of applications with noise (DBSCAN) (provided by the leading python library *scikit-learn* [55]). Our approach uses the 'Metamorphic Testing (MT)' [20] technique that has been shown to be an effective approach in alleviating the oracle problem [35] [43]. It uses the necessary characteristics of the program as relations (known as Metamorphic Relations i.e., MRs) to check whether the program under test adheres to specified relations or not. Each MR uses a *source test case* and a *follow-up test case* to verify the output (instead of verifying the individual test case output). The violation of an MR is treated as a bug in the application.

To the best of our knowledge, very little effort has been placed in using the MT technique

for quality assurance of clustering algorithms [84] [85], and we are only able to find just one paper in which the authors leveraged the MT approach for testing the DBSCAN algorithm (provided by the WEKA tool) [84]. However, the limitations of their approach are that, i) it uses synthetic and unrealistic 2D data, ii) it lacks targeting the verification aspect of testing the DBSCAN algorithm, and iii) it does not provide any evidence to show whether their approach is also applicable to assess the behaviour of the DBSCAN algorithm provided by some popular open source python libraries i.e., sci-kit learn, which are equally the motivators for this work.

In this study [61], we make the following main contributions:

- We propose an MT-based approach for verification and validation of a density based clustering algorithm i.e., DBSCAN.
- We propose a collection of 21 diverse MRs that the clustering algorithm under test is expected to satisfy.
- We further provide the analysis and reasoning for the proposed MRs to show whether each of the MRs target the necessary characteristics of the program; and thus, the MR can be used to serve for *verification* purposes. If the program reveals a violation, it would suggest that there is some bug in the program under test. On the other hand, if the MR does not target the necessary characteristics of the program, it can still be used to show evidence for *validation* purposes (i.e., the 'expected' behaviour). In other words, the proposed MRs target both the verification and validation aspects of testing the DBSCAN algorithm under test.
- To show the applicability of the proposed MRs, we conduct a case study on an open-source anomaly detection system[1] that internally uses the DBSCAN algorithm (to

[1]https://github.com/matifkhattak/DBScanAnomalyDetection

isolate/detect the noise).

## Related Work

MT has been shown to be an effective approach in alleviating the oracle problem in a broad range of applications i.e., forecasting [22], image classification [23], acoustic scene classification [48], intrusion detection systems [60] [76], and machine translation [72]. Xie et al. [83] applied MT to test a category of supervised classifiers that includes K-means and Naive Bayes algorithms (provided by the WEKA tool). The authors proposed 11 MRs to test these two algorithms and identified real faults in the Naive Bayes classifier. Dwarakanath et al. [23] proposed 8 MRs to test an SVM-based digits image classifier. Their results show that the proposed approach is able to uncover 71% of the implementation faults in the applications under test. Santos et al. [65] applied MT on a breast cancer machine learning classifier and the results obtained show that MT can be considered a useful approach in testing the effectiveness of ML-based classifier in a medical domain.

With the availability of large data sets and higher computing power, deep learning solutions are getting popular and are becoming an integral part of critical applications e.g., autonomous vehicles, health care systems, and intrusion detection systems. Pei et al. [56] used the differential testing technique in combination with MT to test real-world deep learning systems. They leveraged the MRs to generate difference-causing images and found thousands of erroneous behaviors in the programs under test. Zhou et al. [89] used MT to test self-driving cars and found subtle errors eight days before the deadly Uber accident took place. Ding et al. [21] applied the MT approach to validate a deep learning framework that has been used for the classification of biomedical images. The proposed approach was shown to be effective in validating the framework that includes, i) the architecture of a convolutional neural network, ii) the execution environment Caffe, and iii) the data set comprised of cellular images. The most recent work [60] [76] includes taking advantage of

machine learning and statistical-based MT techniques to uncover the implementation faults in a DNN-based intrusion detection system and a cancer prediction system.

## Our Approach

In this study, we propose Metamorphic Testing (MT) based approach and show how can it be used to target both the *verification* and *validation* aspect of testing the clustering-based ML applications. To date, one of the main challenges faced in the software testing domain is the *oracle problem*. One possible approach to target the oracle problem is the *Differential Testing* technique that uses multiple implementations of the same program as a *pseudo oracle* to verify the output. However, the limitation of this approach that makes it unrealistic in real life is that it is very difficult to obtain multiple copies of the same system, and second, it is possible that multiple implementations have the same bug which causes them to produce the same output for the same given input.

Our approach for testing the DBSCAN algorithm uses the MT technique which is considered an effective approach in alleviating the oracle problem and it also does not require multiple implementations of the program under test. Instead of verifying the individual output, it uses multiple executions of the program to check whether the output is correct or not. The output should adhere to the relation known as *Metamorphic Relation (MR)* that targets the necessary characteristics of the program under test. Each MR uses the *source input* and some valid change/transformation made to the *source input* (using the relation specified in the MR) known as the *follow-up input*. The MR is said to be violated if the output produced for the *source input* does not match with the output produced for the *follow-up input*. As an example, suppose there is a program that calculates the standard deviation for the given inputs. An instance of an MR can be exemplified by multiplying the original inputs with -1 (this transformation will produce the follow-up inputs), the output of the program should remain the same because this change will have no effect on the deviation

from the mean.

First, we refer to the official documentation of python based implementation available for the DBSCAN algorithm [1] and propose a broader set of 21 MRs which will enable novice to expert data scientists the ability to test and assess DBSCAN's behavior from multiple perspectives. The list of all the proposed MRs along with their descriptions is provided in Table 8.1. Next, in Table 8.2, we further analyze the proposed MRs and provide the detailed reasoning to show whether each of the MRs targets the verification aspect or the validation aspect of testing the DBSCAN algorithm. This is important because in real-life projects, the data may undergo through some modification and if the user is not aware of the potential consequences on the clustering results, it may lead to misleading decisions and disastrous consequences. Lastly, we show the effectiveness of the proposed MRs on testing an anomaly detection system that utilizes the DBSCAN algorithm, provided by the scikit-learn python library. It is important to highlight that the proposed MRs are not just limited to testing the anomaly detection system under test, instead, they are applicable to number of other real-world applications (i.e., market and customer segmentation, document clustering, search results clustering, biological data clustering, etc.) as long as they use the DBSCAN algorithm for identification of clusters in the data.

Table 8.1: Proposed Metamorphic Relations

| **Metamorphic Relations (MRs)** | **Description** |
|---|---|
| **MR1 - Data Duplication** | **MR1.1 - Duplicating single instance:** For the source input $s$, we denote the output obtained as $O_s$. This MR says that if a single instance in the follow-up input $f$ is duplicated, the output $O_f$ should remain similar to the $O_s$ i.e., $O_s = O_f$. **MR1.2 - Duplicating multiple instances:** This MR says that if we duplicate multiple instances (i.e., each belonging to a different cluster) in the follow-up input, the output for both the source and follow-up inputs should remain the same. |
| **MR2 - Data Standardization** | For the follow-up input, if the existing standardized data is once again standardized, it should not have any effect on changing the final output i.e., $O_s = O_f$. |
| **MR3 - Features Duplication** | For the follow-up input, if the existing feature(s) are duplicated, the output should remain consistent for both the source and follow-up inputs i.e., $O_s = O_f$. |

| | |
|---|---|
| **MR4 - Removing Instance(s)** | **MR4.1 - Removing an instance from a single cluster:** For the source input $s$, we denote the clusters found as $C_0, C_1, C_2, ..., C_n$. This MR says that for the follow-up input if an instance is removed from just a single cluster i.e., $C_1$, the output should remain the same i.e., it should not have any effect on changing the results for the remaining instances.<br>**MR4.2 - Removing an instance from multiple clusters:** For the source input $s$, we denote the clusters found as $C_0, C_1, C_2, ..., C_n$. This MR says that for the follow-up input if an instance is removed from each of the obtained clusters i.e., $C_1, C_2, C_3, ..., C_n$, the output should remain the same.<br>**MR4.3 - Removing multiple instances from a single cluster:** For the source input $s$, we denote the clusters found as $C_0, C_1, C_2, ..., C_n$. This MR says that for the follow-up input if multiple instances are removed from just a single cluster i.e., $C_1$, it should not have any effect on changing the results for the remaining instances. |
| **MR5 - Adding an Uninformative Feature** | This MR says that for the follow-up input if we add a new uninformative feature (i.e., having a constant value for all the instances), the output should remain consistent for both the source and follow-up inputs i.e., $O_s = O_f$. |
| **MR6 - Deterministic Output Across Multiple Runs** | This MR says that if a new data point is added to the original data set, no matter how many times an algorithm under test is run, the output for this input and other inputs should remain consistent i.e., the output at $time_i$ and $time_{i+1}$ should remain the same. |

| | |
|---|---|
| **MR7 - Shifting of Data** | This MR says that if we shift the feature(s) with some constant $k$, the output for both the source and follow-up inputs should remain consistent i.e., $O_s = O_f$. |
| **MR8 - Scaling of Data** | This MR says that if we scale the feature(s) with some constant $k$, the output for both the source and follow-up inputs should remain unchanged i.e., $O_s = O_f$. |
| **MR9 - Data Replacement** | **MR9.1 - Replacing single instance:** This MR says that if we replace a single instance in cluster $c_i$ with another instance (belonging to the same cluster $c_i$), the output should remain unchanged.<br>**MR9.2 - Replacing multiple instances:** This MR says that if we replace multiple instances in cluster $c_i$ with another instance (belonging to the same cluster $c_i$), the output for both the source and follow-up inputs should remain the same.<br>**MR9.3 - Replacing all instances:** This MR says that if we replace all the instances in cluster $c_i$ with another instance (belonging to the same cluster $c_i$), it should not have any effect on changing the output for this, and the remaining instances. |
| **MR10 - Shuffling the Features** | This MR says that if we shuffle the data features to change their existing order, the output for both the source and follow-up inputs should remain the same. |
| **MR11 - Adding an Informative Attribute** | This MR says that for the follow-up input if we add a new informative feature (i.e., its value is strongly associated with each of the clusters), the output should remain consistent for both the source and follow-up inputs i.e., $O_s = O_f$. |

| | |
|---|---|
| **MR12 - Transformation of Rows** | **MR12.1 - Reversing the order:** This MR says that reversing the order of data instances should not have any effect on changing the final output i.e., the output for both the source and follow-up inputs should remain consistent.<br>**MR12.2 - Random shuffling:** This MR says that if we randomly shuffle the data instances, it should not have any effect on changing the final output i.e., the output for both the source and follow-up inputs should remain the same. |
| **MR13 - Data Reflection** | This MR says that if we perform the data reflection transformation on the original inputs i.e., multiply all the features with -1, the output should remain unchanged i.e., $O_s = O_f$. |
| **MR14 - Addition of Instance(s)** | **MR14.1 - Adding a new instance with informative attributes:** For the follow-up input, if we add a new data point in the middle of two existing data points of cluster $c_i$ (thus enhancing the density of the cluster), it should not have any effect on changing the results for remaining instances.<br>**MR14.2 - Adding a new instance with uninformative attributes** For the follow-up input, if we add a new data point with uninformative attributes i.e., features with 0 value, it should not have any effect on changing the results for the remaining instances. |

Table 8.2: DBSCAN Algorithm: Analysis For The Proposed MRs from Verification (VR) And Validation (VD) Perspective

| # | VR? | VD? | Reasoning |
|---|---|---|---|
| MR1 | × | ✓ | For the source input, we denote the clusters identified as $C_1$, $C_2$, $C_3$, ..., $C_n$. For the follow-up input, if we duplicate a data point $x$ (which is a border point) in cluster $C_1$, the addition of a new data instance may change this border point $x$ to become a core point. This core point will now be used by the DBSCAN algorithm to further grow the cluster, which can resultantly assign the other border points (belonging to different clusters) to this cluster $C_1$; thus, changing the final output for the follow-up inputs. A sample example is provided in the excel sheet (available in the shared GitHub repository) that shows the violation of this MR.<br>It is worth mentioning that this MR is not the necessary characteristic of the algorithm under test because the violation of this MR is not due to the bug in the implementation. Instead, the behavior of the program does adhere to the implementation but violates the user's potential requirements/expectations; thus, can not be used for verification but can definitely be used for validation purposes. Apart from that, an MR targeting the verification aspect can also be used for validation purposes but not always vice versa.<br>***Note:*** *We urge the readers to use the same reasoning (as mentioned above) for the rest of the proposed MRs in order to target the verification and validation aspect of testing the algorithm under test.* |

| | | | |
|---|---|---|---|
| MR2 | ✓ | ✓ | This MR can be used for verification (hence also for validation) purposes because the re-execution of the standardization step will neither change the mean and variance of data points nor the distance between the core points, border points and noisy points. Therefore, this transformation should not have any effect on how the core points, border points, and noisy points are identified by the algorithm to make the clusters. If this MR is violated, it would depict that there is some bug in the program under test. |
| MR3 | × | ✓ | For the follow-up input, if we duplicate all the existing attributes to the original data points, it will result in doubling the distance between them. As a result, for a data point $x_i$, the *min_samples (i.e., minimum no of data points)* within the *epsilon* distance may get fewer which will result in the identification of a different number of core points, border points, and noisy points; thus, may assign the data points to different clusters. |
| MR4 | × | ✓ | For the follow-up input, if we remove any data point(s), it may lead to i) changing the core point to become a border point, and ii) a border point to the noisy point; thus, changing the final output for the program under investigation. Since this violation can not be characterized as a violation of the program specification, so, this MR can be used for validation purposes (but not for verification purposes). |

| | | | |
|---|---|---|---|
| MR5 | ✓ | ✓ | For the follow-up input, if we add an uninformative feature i.e., a feature with value 0, to all the data points, it will not change the existing relationship between the core points, border points, and noisy points. Therefore, the output should remain consistent for both the source and follow-up inputs. |
| MR6 | ✓ | ✓ | If we run the program under test multiple times, it will not have any effect on how the algorithm identifies the core points, border points and noisy points. Furthermore, as we do not make any change in the data, so, the output should remain consistent. On the contrary, if the output for the follow-up input is different from the output obtained for the source input, it would depict that there is some bug in the implementation of the program under test. Therefore, this MR can be used for both verification and validation purposes. |
| MR7 | ✓ | ✓ | For the follow-up input, if we shift all the features of the data points with some constant $c$, it will not change the existing relationship/distance between the data points. To understand this concept, suppose $x$ represents the core point, $y$ represents the border point, and, the distance found between them during the source execution (i.e., $d(x, y)$) is represented as $z$. Now, after shifting the features for the follow-up inputs, the distance between these two points will remain the same i.e., $d(x, y) = \sqrt{((x + c) - (y + c))^2} => \sqrt{(x - y)^2} => x - y => z$. Therefore, the output should remain unchanged for both the source and follow-up inputs. |

| | | | |
|---|---|---|---|
| MR8 | × | ✓ | If all the features are scaled with some constant $c$, it will result in increasing the distance between the data points. As a result, for a data point $x_i$, the *min_samples* within the *epsilon* distance may get fewer which will result in the identification of a different number of core points, border points, and noisy points; thus, leading to the assignment of data points to different clusters. In our conducted experiment (results are shown in Table 8.3), the scaling of features moved the data points so farther away that majority of them got assigned to cluster -1 (treated as noise/anomaly). |
| MR9 | × | ✓ | For the follow-up input, if we replace the instance $x$ with $y$ (which is a border point), this will add a new data point within the *epislon* distance of $y$, which may make this instance to become a core point. This core point will now be used by the DBSCAN algorithm to further grow the cluster, which can resultantly assign the other border points (belonging to different clusters) to the cluster to which $y$ belongs; thus, changing the final output for the follow-up execution. |
| MR10 | ✓ | ✓ | This MR can be used for verification and validation purposes because changing the location of features will neither change the distance between the data points nor their location in space. Therefore, this transformation should not have any effect on how the core points, border points, and noisy points are identified to make the clusters. The violation of this MR would depict that there is some implementation bug in the program under test. |

| | | | |
|---|---|---|---|
| MR11 | ✓ | ✓ | For the source input, we denote the clusters identified as $C_1$, $C_2$, $C_3$, ..., $C_n$. This MR says that for the follow-up input, if we add an informative attribute such that the added attribute is strongly associated with each cluster's instances i.e., having value $s$ for $C_1$ instances, having value $t$ for $C_2$ instances, having value $u$ for $C_3$ instances, and similarly, having value $v$ for $C_n$ instances, this will not change the existing relationship between the data points belonging to the same cluster; thus, the instances should be assigned to the same clusters as identified for the source inputs. |
| MR12 | × | ✓ | For the follow-up input, if we change the order of data instances, the algorithm will assign the core data points to the same clusters but i) it may change their labels (based on the order in which they were provided to the algorithm under test [1]), and ii) it may assign the non-core points to different clusters [1]; thus, changing the final output for the follow-up inputs. |
| MR13 | ✓ | ✓ | This MR will also not change the existing relationship/distance between the data points. Therefore, it should not have any effect on changing the final output for the follow-up inputs. |
| MR14 | × | ✓ | Same reasoning as provided for MR9. If we add a new data point to any of the clusters (identified for the source inputs), it may change the border point $y$ to become a core point. This core point will now be used by the DBSCAN algorithm to further grow the cluster, which can resultantly assign the other border points (belonging to different clusters) to the cluster to which $y$ belongs; thus, changing the final output for the follow-up inputs. |

## Experimentation and Evaluation

To show the effectiveness of our proposed MT-based approach, we applied our technique to test an open-source anomaly detection system that internally uses the DBSCAN algorithm (provided by scikit-learn 0.24.1 [55]) to identify noise and anomalies. The scikit-learn is a popular python library that provides a large set of features for performing data pre-processing, classification, and clustering related tasks. Due to its wide range of capabilities, it is very popular among both researchers and practitioners for addressing ML-related problems.

The ingested data to the application under test is comprised of 3093 instances (having 29 features). This data is used to cluster the input data into different groups. Once the clustering process is completed, the instances that are assigned to the -1 cluster are treated as noise/anomaly. All the application code, the data, and the MRs implementation are provided online[2].

After applying the proposed MRs to the anomaly detection application under test, we present the results in Tables 8.3 and 8.4. The results in Table 8.3 shows, i) whether the MR is rejected/violated, and ii) for the given violated MR, what number and percentage of data instances have shown violations. A higher percentage of violation rate suggests that the program is highly susceptible to small changes in the dataset, therefore, it should be avoided in the scenarios captured by the violated MRs. Each of the rejected MRs either shows the deviation from the program implementation (verification) or its deviation from general user expectations (validation). If the application under test violates the MR targeting the verification aspect, it would suggest that there is some bug in the program under test. On the other hand, if the MR that does not target the necessary characteristics of the program is violated, it can still be used to show evidence for *validation* purposes (i.e., the 'expected'

[2]https://github.com/matifkhattak/DBScanAnomalyDetection

behaviour). Further analysis of the results in Table 8.3 show that in total, the application under test has violated 11 out of 21 MRs (i.e., 52%).

In order to synthesize the granular information provided in Table 8.2, we present the results of violated MRs at two levels (as shown in Table 8.4):

i) MRs targeting the *verification* aspect, and

ii) MRs targeting the *validation* aspect.

Table 8.3: Results from testing the DBSCAN algorithm

| MR# | Violation? | No of Violated Instance(s) | Violation Rate |
|---|---|---|---|
| MR1.1 | ✓ | 5 | 0.16% |
| MR1.2 | ✓ | 7 | 0.23% |
| MR2 | × | | |
| MR3 | ✓ | 240 | 7.76% |
| MR4.1 | ✓ | 1 | 0.03% |
| MR4.2 | ✓ | 2 | 0.06% |
| MR4.3 | ✓ | 1937 | 62.63% |
| MR5 | × | | |
| MR6 | × | | |
| MR7 | × | | |
| MR8 | ✓ | 2926 | 94.60% |
| MR9.1 | ✓ | 1 | 0.03% |
| MR9.2 | ✓ | 4 | 0.13% |
| MR9.3 | ✓ | 171 | 5.53% |
| MR10 | × | | |
| MR11 | × | | |

| | | | |
|---|---|---|---|
| MR12.1 | × | | |
| MR12.2 | × | | |
| MR13 | × | | |
| MR14.1 | × | | |
| MR14.2 | ✓ | 1 | 0.03% |

Table 8.4: MRs Seggregation and Their Results

| | Total No. of MRs | No. of Violated MRs | % of Violated MRs |
|---|---|---|---|
| Verification | 7 | 0 | 0% |
| Validation | 21 | 11 | 52.4% |

In order to understand the possible reasons behind the violated MRs, we refer readers to the detailed analysis and reasoning provided in Table 8.2. The results provided in Table 8.4 show that none of the MRs targeting the *verification* aspect have been violated, which further enhance our trust on the implementation of this algorithm (i.e., its adherence of implementation to specification). However, it is important to note that failure to detect the violations for this type of MRs does not necessarily mean that the program under test is free from bugs. Instead, it is the inherent limitation of every testing technique (i.e., if a testing technique is unable to uncover bugs in a program, it does not necessarily mean that the program does not contain any defects at all), as is with ours. Therefore, we recommend that the proposed MRs should still be used as a supplementary testing technique in the organization's existing built-in testing pipeline. Apart from that, we see a higher number of violated MRs (i.e., 11 out of 21 => 52.4%) targeting the *validation* aspect, which ultimately suggests that the program under test is highly susceptible to small valid changes in the input data and may not be suitable for the scenarios (captured by the

violated MRs) to meet the general user expectations from this algorithm. This answers the RQ5.3 (of RG5) that **the proposed approach is effective in the detection of violations (targeting the validation aspect) in the program under test**. We hope that this type of useful information will help both researchers and practitioners to be aware of possible repercussions in the final results when the data itself may undergo changes in the future (which is very common in real world use cases). If they foresee the possible usage of application in the environment/scenarios captured by the violated MRs then this set of knowledge would significantly help them to perform a comparative analysis of the behaviour of different clustering algorithms, choosing the one best suited for their problem, and making well informed decisions.

**To answer the RQ5.4 (of RG5)**, the results obtained show that different MRs have detected a different percentage of violations for the given instances. It can be seen in Table 8.3 that MR8 has detected a higher percentage (94.60%) of violations, showing that the program under test is highly susceptible to small input changes (captured by this MR), whereas, MR4.1, MR9.1, and MR14.2 have the lowest percentage (0.3%) of violations (showing that among the violated MRs, for these three MRs, the program under test is least susceptible to small modifications). Apart from that, MR4.1, MR9.1, and MR14.2 have shown the violation for the same number of instances (i.e., one instance), thus; one may argue that since these all three MRs have the same test effectiveness, using only one of them would be enough. However, this reasoning (without making a deeper analysis) may be subjective and incorrect. Therefore, we further analysed the results for these three MRs and found that the instance for which the program shows a violation is different for each of the violated MRs; thus, revealing the diversity of the proposed MRs and suggesting that they all have different test effectiveness. This answers the raised research question that **different MRs have different ability to detect the violation for different number/percentage of data inputs.**

## Threats To Validity

In this section, we examine the threats to the validity of this research as described by Wohlin et al. [82]. Although the proposed approach seems to be effective in its ability to identify scenarios for which DBSCAN shows inconsistent behavior, we identify the following threats to validity:

- It can be argued that the proposed MRs may not be sufficient to validate a potentially large set of scenarios. This threat refers to the possibility of having unwanted or unanticipated causal relationships as a result of the MRs selected. This poses a potential threat to *internal validity* of this study. We minimize this threat by proposing a sufficient number of diverse MRs whose results show that even with this limited, but diverse set of MRs, we successfully identify the violations for 52.4% of the scenarios (captured by the MRs targeting the validation aspect).

- *Construct validity* refers to the meaningfulness of measurements made. The threat to *construct validity* may occur because of the selection of the algorithm for this study. We tried to minimize this threat by showing the effectiveness of the proposed MRs on an award-winning DBSCAN clustering algorithm, provided by the leading python library *scikit-learn*.

- *Content validity* refers to how complete the proposed MRs cover the content domain; which in our case is clustering algorithms. Although our focus is on DBSCAN, we have proposed a large set (i.e., 21) diverse MRs that are not only applicable to test this algorithm but has also been applied in our previous work [62] for testing *partitioning-based* and *hierarchical-based* clustering algorithms.

- The DBSCAN algorithm under test belongs to the category of density-based clustering

algorithms, thus generalizing the results (obtained from the proposed approach) to other type of density-based algorithms might be speculative and poses a threat to *external validity*. We aim to minimize this threat in the future by showing the applicability of the proposed approach on a collection of other density-based algorithms.

## Conclusion And Future Work

Our contribution with this research includes proposing an approach that both naive and expert users can benefit from for testing clustering based applications. This work shows the feasibility of the proposed MT based approach in both the verification and validation of the award-winning DBSCAN algorithm (a density based clustering algorithm), provided by the widely used python library i.e., scikit-learn. We proposed a broader set of 21 MRs and also provided a detailed analysis and reasoning to show whether each of the proposed MRs targets the verification or the validation aspect of testing the algorithm under test. Despite the fact that we have proposed a broader set of 21 MRs, this list is not exhaustive, and instead, we expect researchers to further expand this seminal list with additional MRs that can be leveraged to test a broader set of ML applications.

To show the effectiveness of the proposed approach, we further conducted a case study on an anomaly detection system. The results obtained show that 52.4% of the MRs (targeting the validation aspect) have been violated by the program under test, with one MR showing the highest violation rate of 94.5%. To show the applicability of the proposed MRs in general, we not only hope to apply them in testing other types of density based algorithms but also aim to further enhance the MR repository in the future.

# CONCLUSION AND FUTURE WORK

## Conclusion

This dissertation outlines current research in applying SE testing techniques for the quality assurance of machine learning models (supervised and unsupervised) and proposes an extension to this research by addressing existing gaps in the literature. This extension includes conducting multiple case studies to show the applicability of proposed approaches for testing both the supervised learning algorithms (i.e, neural network-based classifiers), and unsupervised learning algorithms (i.e., partitioning, hierarchical, and density-based clustering). To complete this dissertation, first, we adapted the traditional SE testing technique (metamorphic testing) and propose a statistical metamorphic testing technique for the identification of implementation bugs in NN-based classifiers that have a stochastic nature in training. To show the applicability of the proposed approach (in terms of further validation) in the image classification domain, we apply the proposed approach to test a CNN-based image classifier used for detecting pneumonia among patients. We also propose an MRs minimization algorithm that helps in saving organizational resources and performing testing with fewer MRs (especially in a regression testing environment) without compromising the overall fault detection effectiveness of the proposed approach. Second, we propose a hybridized testing approach in which, (i) instead of using MT for testing purposes, we utilize its power for addressing the data collection/labeling problem; that is, enhancing the prioritized test set size without incurring any additional labeling cost for an organization, and (ii) propose a statistical hypothesis testing technique (for detection) and ML-based approach (for prediction) of buggy behavior in the next release of the program under test. Other than the popularity of supervised learning algorithms, unsupervised learning algorithms i.e., partitioning-based, hierarchical-based, and density-based clustering, are also very common in solving data clustering related problems. However, to the best of

our knowledge, much less effort has been put in testing clustering algorithms. To contribute in this space, we use metamorphic testing technique and propose a diverse set of MRs to test such types of algorithms i.e., k-means (a partitioning-based algorithm), agglomerative clustering (a hierarchical-based clustering), and DBSCAN (a density-based clustering) from both the verification (targeting the necessary characteristics of the algorithm) and validation (targeting the user expectations from the algorithm) perspective. To the best of our knowledge, this work is novel in addressing the gaps identified in the literature and contributes to the body of knowledge of the SE4ML community.

## Future Work

This dissertation is focused on the *testing* aspect of ML applications and proposes multiple testing strategies for testing both supervised and unsupervised ML algorithms for improving confidence in them. In the future, we intend to make additional contributions in the emerging space of SE4ML by extending this work.

First, we intend to enhance the MRs repository for testing classification models dealing with both the structured and unstructured data, and evaluating their performance using the proposed statistical metamorphic testing technique. To show the applicability of the proposed approach in other domains, we intend to apply it in testing of DNNs used in speech recognition and natural language processing domains. Further, we aim to leverage machine learning techniques for effective MRs prioritization and their minimization.

Second, we intend to propose more effective MRs that can be used to leverage prioritized test inputs for generating additional representative data; thus reducing labeling costs further and enabling organizations to check program correctness on large input scenarios for enhancing their trust. This work will be an extension to the existing work proposed in the chapter titled 'A Hybridized Approach for Testing Neural Network Based Intrusion Detection Systems'. The results obtained in this preliminary study are encouraging, and a

comprehensive study on applying the proposed approach on a larger number of mutants is in progress. It will also be interesting to explore what other types of useful features can be added to enhance the performance of ML models for predicting bugs in the next release of the classifiers under test.

In the space of testing unsupervised ML algorithms, we have seen much less research work done. Therefore, we not only aim to enhance the MRs repository targeting clustering algorithms (from both the verification and validation perspective) but will also show the general applicability of the proposed MRs [61] [62] by utilizing the proposed approach in testing a broad range of other clustering algorithms that are popular among both researchers and practitioners of the ML community. Further, to improve the testing of agglomerative and DBSCAN clustering based applications, we intend to develop new criteria that can be used to verify MRs at multiple levels, which will ultimately help in building trust in using critical application algorithms.

Last but not least, we also see a research gap in the availability of open-source ML testing tools. Therefore, we aim to develop an open source testing tool for testing a variety of ML models in an automated fashion. First, we intend to unify the approaches (proposed in this dissertation) into a single tool and then expand it gradually with new research contributions. Second, we aim to integrate machine learning based techniques for predicting potential MRs best suited for testing the ML model(s) under test. Third, this tool will utilze the MRs prioritization/minimization component to prioritize/minimize MRs for the program(s) under test. This initiative will make this tool available for everyone and will also allow researchers across the globe to contribute and extend it for better serving the community.

We equally believe that the metamorphic testing is a simpler but a very powerful approach that can be used for testing a plethora of complex and critical programs; thus, more contributions in this space will further raise its awareness among both researchers and

practitioners. It is also worth exploring to check what other types of traditional software testing techniques can be leveraged and reused in the emerging ML space, thus; preventing reinvention of techniques.

# REFERENCES CITED


[1] https://scikit-learn.org/stable/modules/clustering.html#dbscan.

[2] John H.McDonald http://www.biostathandbook.com/small.html, last accessed: 04 Oct 2022.

[3] Python Software Foundation. MutPy 0.4.0. https://pypi.python.org/pypi/MutPy/0.4.0. [Online; accessed 04 Oct 2022].

[4] udacity, "Udacity challenge", 2017. *[online] Available at: https://github.com/udacity/self-driving-car [Accessed 04 Oct 2022].*

[5] WEKA, "Weka 3: Data mining software in java". *[online] Available at: https://www.cs.waikato.ac.nz/ml/weka/ [Accessed 04 Oct 2022].*

[6] World Health Organization. (2021), https://www.who.int/news-room/fact-sheets/detail/pneumonia.

[7] ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation SQuaRE) — System and software quality models. 2011.

[8] Basant Agarwal and Namita Mittal. Text classification using machine learning methods-a survey. In *Proceedings of the Second International Conference on Soft Computing for Problem Solving (SocProS 2012), December 28-30, 2012*, pages 701–709. Springer, 2014.

[9] Berlian Al Kindhi, Tri Arief Sardjono, Mauridhi Hery Purnomo, and Gijbertus Jacob Verkerke. Hybrid K-means, fuzzy C-means, and hierarchical clustering for DNA hepatitis C virus trend mutation analysis. *Expert systems with applications*, 121:373–381, 2019.

[10] Ashraf Tahseen Ali, Hasanen S Abdullah, and Mohammad N Fadhil. Voice recognition system using machine learning techniques. *Materials Today: Proceedings*, 2021.

[11] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2020.

[12] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300. IEEE, 2019.

[13] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, pages 402–411, 2005.

[14] Florian Auer and Michael Felderer. Addressing data quality problems with metamorphic data relations. In *2019 IEEE/ACM 4th International Workshop on Metamorphic Testing (MET)*, pages 76–83. IEEE, 2019.

[15] Victor R Basili. Software modeling and measurement: the Goal/Question/Metric paradigm. Technical report, 1992.

[16] Azzedine Boukerche and Jiahao Wang. Machine Learning-based traffic prediction models for Intelligent Transportation Systems. *Computer Networks*, 181:107530, 2020.

[17] Taejoon Byun, Vaibhav Sharma, Abhishek Vijayakumar, Sanjai Rayadurgam, and Darren Cofer. Input prioritization for testing neural networks. In *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 63–70. IEEE, 2019.

[18] Victor R Basili1 Gianluigi Caldiera and H Dieter Rombach. The goal question metric approach. *Encyclopedia of software engineering*, pages 528–532, 1994.

[19] Sai Yeshwanth Chaganti, Ipseeta Nanda, Koteswara Rao Pandi, Tavva GNRSN Prudhvith, and Niraj Kumar. Image Classification using SVM and CNN. In *2020 International Conference on Computer Science, Engineering and Applications (ICCSEA)*, pages 1–5. IEEE, 2020.

[20] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. *Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong*, 1998.

[21] Junhua Ding, Xiaojun Kang, and Xin-Hua Hu. Validating a deep learning framework by metamorphic testing. In *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*, pages 28–34. IEEE, 2017.

[22] Anurag Dwarakanath, Manish Ahuja, Sanjay Podder, Silja Vinu, Arijit Naskar, and MV Koushik. Metamorphic testing of a deep learning based forecaster. In *2019 IEEE/ACM 4th International Workshop on Metamorphic Testing (MET)*, pages 40–47. IEEE, 2019.

[23] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M Rao, RP Jagadeesh Chandra Bose, Neville Dubash, and Sanjay 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*, pages 118–128, 2018.

[24] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996.

[25] Mohamed Amine Ferrag, Leandros Maglaras, Sotiris Moschoyiannis, and Helge Janicke. Deep learning for cyber security intrusion detection: Approaches, datasets, and comparative study. *Journal of Information Security and Applications*, 50:102419, 2020.

[26] Thiago S Guzella and Walmir M Caminhas. A review of machine learning approaches to spam filtering. *Expert Systems with Applications*, 36(7):10206–10222, 2009.

[27] John A Hartigan and Manchek A Wong. Algorithm AS 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, 28(1):100–108, 1979.

[28] Alex Hern. Facebook translates 'good morning' into 'attack them', leading to arrest. *the Guardian*, 24, 2017.

[29] Phan Duy Hung, Nguyen Thi Thuy Lien, and Nguyen Duc Ngoc. Customer segmentation using hierarchical agglomerative clustering. In *Proceedings of the 2019 2nd International Conference on Information Science and Systems*, pages 33–37, 2019.

[30] Habiba A Ibrahim, Ahmad Taher Azar, Zahra Fathy Ibrahim, and Hossam Hassan Ammar. A hybrid deep learning based autonomous vehicle navigation and obstacles avoidance. In *The International Conference on Artificial Intelligence and Computer Vision*, pages 296–307. Springer, 2020.

[31] Amit Kumar Jaiswal, Prayag Tiwari, Sachin Kumar, Deepak Gupta, Ashish Khanna, and Joel JPC Rodrigues. Identifying pneumonia in chest X-rays: A deep learning approach. *Measurement*, 145:511–518, 2019.

[32] Darryl C Jarman, Zhi Quan Zhou, and Tsong Yueh Chen. Metamorphic testing for Adobe data analytics software. In *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*, pages 21–27. IEEE, 2017.

[33] Minghua Jia, Xiaodong Wang, Yue Xu, Zhanqi Cui, and Ruilin Xie. Testing Machine Learning Classifiers based on Compositional Metamorphic Relations. *International Journal of Performability Engineering*, 16(1), 2020.

[34] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[35] Mingyue Jiang, Tsong Yueh Chen, Fei-Ching Kuo, Dave Towey, and Zuohua Ding. A metamorphic testing approach for supporting program repair without the need for a test oracle. *Journal of systems and software*, 126:127–140, 2017.

[36] Zeeshan Khan, Muhammad Zakira, Wushour Slamu, and Nady Slam. A study of neural machine translation from Chinese to Urdu. *Journal of Autonomous Intelligence*, 2(4):29–36, 2020.

[37] John C Knight and Nancy G Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on software engineering*, (1):96–109, 1986.

[38] Fumihiro Kumeno. Sofware engneering challenges for machine learning applications: A literature review. *Intelligent Decision Technologies*, 13(4):463–476, 2019.

[39] Hiroshi Kuwajima, Hirotoshi Yasuoka, and Toshihiro Nakae. Engineering problems in machine learning systems. *Machine Learning*, 109(5):1103–1126, 2020.

[40] Fred Lambert. Understanding the fatal tesla accident on autopilot and the nhtsa probe. *Electrek, July*, 1, 2016.

[41] Matthew Lavine. The early clinical X-ray in the United States: patient experiences and public perceptions. *Journal of the history of medicine and allied sciences*, 67(4):587–625, 2012.

[42] Zheng Li, Zhanqi Cui, Jianbin Liu, Liwei Zheng, and Xiulei Liu. Testing Neural Network Classifiers Based on Metamorphic Relations. In *2019 6th International Conference on Dependable Systems and Their Applications (DSA)*, pages 389–394. IEEE, 2020.

[43] Huai Liu, Iman I Yusuf, Heinz W Schmidt, and Tsong Yueh Chen. Metamorphic fault tolerance: An automated and systematic methodology for fault tolerance in the absence of test oracle. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 420–423, 2014.

[44] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 120–131, 2018.

[45] Silverio Martínez-Fernández, Justus Bogner, Xavier Franch, Marc Oriol, Julien Siebert, Adam Trendowicz, Anna Maria Vollmer, and Stefan Wagner. Software Engineering for AI-Based Systems: A Survey. *arXiv preprint arXiv:2105.01984*, 2021.

[46] Satoshi Masuda, Kohichi Ono, Toshiaki Yasue, and Nobuhiro Hosokawa. A survey of software quality for machine learning applications. In *2018 IEEE International conference on software testing, verification and validation workshops (ICSTW)*, pages 279–284. IEEE, 2018.

[47] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[48] Diogo Moreira, Ana Paula Furtado, and Sidney Nogueira. Testing acoustic scene classifiers using Metamorphic Relations. In *2020 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 47–54. IEEE, 2020.

[49] Mriganka Nath and Chandrajit Choudhury. Automatic detection of pneumonia from chest X-Rays using deep learning. In *International Conference on Machine Learning, Image Processing, Network Security and Data Sciences*, pages 175–182. Springer, 2020.

[50] Molouk Mishmast Nehi, Zahra Fakhrpoor, and Mohammad R Moosavi. Defects in The Next Release; Software Defect Prediction Based on Source Code Versions. In *Electrical Engineering (ICEE), Iranian Conference on*, pages 1589–1594. IEEE, 2018.

[51] S Shajun Nisha and M Nagoor Meeral. Applications of deep learning in biomedical engineering. In *Handbook of Deep Learning in Biomedical Engineering*, pages 245–270. Elsevier, 2021.

[52] A Ohnsman. Lidar maker velodyne 'baffled' by self-driving uber's failure to avoid pedestrian. *Forbes, March*, 2018.

[53] Alireza Osareh and Bita Shadgar. Machine learning techniques to diagnose breast cancer. In *2010 5th international symposium on health informatics and bioinformatics*, pages 114–120. IEEE, 2010.

[54] Hyejin Park, Taaha Waseem, Wen Qi Teo, Ying Hwei Low, Mei Kuan Lim, and Chun Yong Chong. Robustness Evaluation of Stacked Generative Adversarial Networks using Metamorphic Testing. In *2021 IEEE/ACM 6th International Workshop on Metamorphic Testing (MET)*, pages 1–8. IEEE, 2021.

[55] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[56] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.

[57] C Lakshmi Prabha and N Shivakumar. Software defect prediction using machine learning techniques. In *2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)(48184)*, pages 728–733. IEEE, 2020.

[58] Pranav Rajpurkar, Jeremy Irvin, Kaylie Zhu, Brandon Yang, Hershel Mehta, Tony Duan, Daisy Ding, Aarti Bagul, Curtis Langlotz, Katie Shpanskaya, et al. Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning. *arXiv preprint arXiv:1711.05225*, 2017.

[59] Fred Ramsey and Daniel Schafer. *The statistical sleuth: a course in methods of data analysis*. Cengage Learning, 2012.

[60] Faqeer ur Rehman and Clemente Izurieta. Statistical Metamorphic Testing of Neural Network Based Intrusion Detection Systems. In *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 20–26, 2021.

[61] Faqeer Ur Rehman and Clemente Izurieta. An Approach For Verifying And Validating Clustering Based Anomaly Detection Systems Using Metamorphic Testing. In *2022 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 12–18. IEEE, 2022.

[62] Faqeer Ur Rehman and Clemente Izurieta. MT4UML: Metamorphic Testing for Unsupervised Machine Learning. In *2022 9th Swiss Conference on Data Science (SDS)*, pages 26–32. IEEE, 2022.

[63] Yazhou Ren, Kangrong Hu, Xinyi Dai, Lili Pan, Steven CH Hoi, and Zenglin Xu. Semi-supervised deep embedded clustering. *Neurocomputing*, 325:121–130, 2019.

[64] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[65] Sebastião HN Santos, Beatriz Nogueira Carvalho da Silveira, Stevão A Andrade, Márcio Delamaro, and Simone RS Souza. An experimental study on applying metamorphic testing in machine learning applications. In *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing*, pages 98–106, 2020.

[66] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. *Advances in neural information processing systems*, 28:2503–2511, 2015.

[67] Tanvi Sethi et al. Improved approach for software defect prediction using artificial neural networks. In *2016 5th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*, pages 480–485. IEEE, 2016.

[68] Salahuddin Shaikh, Liu Changan, Maaz Rasheed Malik, and Muhammad Asghar Khan. Software Defect-Prone Classification using Machine Learning: A Virtual Classification Study between LibSVM & LibLinear. In *2019 13th International Conference on Mathematics, Actuarial Science, Computer Science and Statistics (MACS)*, pages 1–6. IEEE, 2019.

[69] Krishna Pal Sharma et al. A Four-Stage dynamic approach for Software Defect Prediction. In *2021 2nd International Conference on Secure Cyber Computing and Communications (ICSCCC)*, pages 541–545. IEEE, 2021.

[70] A SIGKDD. sigkdd test of time award, https://www.kdd.org/News/view/2014-sigkdd-test-of-time-award, 2014.

[71] Ben H Smith and Laurie Williams. An empirical evaluation of the MuJava mutation operators. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 193–202. IEEE, 2007.

[72] Liqun Sun and Zhi Quan Zhou. Metamorphic testing for machine translations: MT4MT. In *2018 25th Australasian Software Engineering Conference (ASWEC)*, pages 96–100. IEEE, 2018.

[73] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 271–280. IEEE, 2012v.

[74] Yuchi Tian., Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, pages 303–314, 2018.

[75] Ashitosh Tilve, Shrameet Nayak, Saurabh Vernekar, Dhanashri Turi, Pratiksha R Shetgaonkar, and Shailendra Aswale. Pneumonia detection using deep learning approaches. In *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*, pages 1–8. IEEE, 2020.

[76] Faqeer ur Rehman and Clemente Izurieta. A Hybridized Approach for Testing Neural Network Based Intrusion Detection Systems. In *2021 International Conference on Smart Applications, Communications and Networking (SmartNets)*, pages 1–8. IEEE, 2021.

[77] Faqeer ur Rehman and Clemente Izurieta. Testing Deep Learning Systems: A Statistical Metamorphic Approach. In *IEEE Transactions on Software Engineering*. IEEE, 2022 (*Under Review*).

[78] Dejan Varmedja, Mirjana Karanovic, Srdjan Sladojevic, Marko Arsenovic, and Andras Anderla. Credit card fraud detection-machine learning methods. In *2019 18th International Symposium INFOTEH-JAHORINA (INFOTEH)*, pages 1–5. IEEE, 2019.

[79] Shuai Wang and Zhendong Su. Metamorphic testing for object detection systems. *arXiv preprint arXiv:1912.12162*, 2019.

[80] Elaine J Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[81] Ian H Witten and Eibe Frank. Data mining: practical machine learning tools and techniques with Java implementations. *Acm Sigmod Record*, 31(1):76–77, 2002.

[82] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[83] Xiaoyuan Xie, Joshua WK Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software*, 84(4):544–558, 2011.

[84] Xiaoyuan Xie, Zhiyi Zhang, Tsong Yueh Chen, Yang Liu, Pak-Lok Poon, and Baowen Xu. METTLE: a METamorphic testing approach to assessing and validating unsupervised machine LEarning systems. *IEEE Transactions on Reliability*, 69(4):1293–1322, 2020.

[85] Sen Yang, Dave Towey, and Zhi Quan Zhou. Metamorphic exploration of an unsupervised clustering program. In *2019 IEEE/ACM 4th International Workshop on Metamorphic Testing (MET)*, pages 48–54. IEEE, 2019.

[86] Long Zhang, Xuechao Sun, Yong Li, and Zhenyu Zhang. A noise-sensitivity-analysis-based test prioritization technique for deep neural networks. *arXiv preprint arXiv:1901.00054*, 2019.

[87] M Zhang, Y Zhang, L Zhang, C Liu, and S Khurshid. DeepRoad: GAN-based metamorphic autonomous driving system. *Research Gate Publication*, 2018.

[88] Husheng Zhou, Wei Li, Zelun Kong, Junfeng Guo, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. Deepbillboard: Systematic physical-world testing of autonomous driving systems. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 347–358. IEEE, 2020.

[89] Zhi Quan Zhou and Liqun Sun. Metamorphic testing of driverless cars. *Communications of the ACM*, 62(3):61–67, 2019.

[90] Chris Ziegler. A google self-driving car caused a crash for the first time. *The Verge*, 2016.