TESTING MACHINE LEARNING ALGORITHMS WITHOUT ORACLE

A Thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfilment
of the Requirements for the Degree

Master of Science in Computer Science

University of Nebraska at Omaha

by

Abhishek Kumar

August, 2018

Supervisory Committee:

Harvey Siy, Ph.D.

Myoungkyu Song, Ph.D.

Matthew Hale, Ph.D.

TESTING MACHINE LEARNING ALGORITHMS WITHOUT ORACLE

Abhishek Kumar, M. S.

University of Nebraska, 2018

Advisor: Harvey Siy, Ph.D.

Abstract here

ACKNOWLEDGMENTS

Acknowledgments here

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine learning has gained rapid popularity in the past decade and different sectors like healthcare, finance, retail, etc. are using machine learning to provide better products and services. This rising popularity has created a demand for better implementation of the state-of-the-art algorithms in order to implement more complex and sophisticated use cases. Machine learning has been around for a long time now and several developers have written a number of tools and libraries to help others learn and use these algorithms in their own projects. While developing and training a machine learning model, the developers rely on the accuracy of the libraries to produce best results. The aim is to create a model that makes the best predictions. Conventionally, oracles have been used to test the correctness of a program. Oracles provide expected values against which the output from the program can be compared and validated. Lack of reliable oracles for testing machine learning algorithms makes it very hard to test the accuracy of such programs. This problem is called the oracle problem[12]. Oracle problem arises when either:

- An oracle does not exist, or,

- An oracle can theoretically exist but it is computationally too expensive to determine the output.

Such set of programs which does not have a test oracle to predict the output on a set of inputs are called "non-testable programs"[8]. Davis and Weyuker describe these set of programs as "Programs which were written in order to determine the answer in the first place. There would be no need to write such programs if the correct answer were known"[3]. Most machine learning programs fall under this category as they are written in order to predict the correct answers in the first place. Several techniques can be employed to verify the correctness of such programs. One of the most popular methods being the use of pseudo-oracles. To use pseudo-oracle two or more implementations of the algorithm are independently developed to fulfill the same specification. They are run on the same input test data and the outputs from them are then compared. If the outputs match it can be asserted that the original results are according to the specification. This kind of testing is often referred to as dual coding. However, this technique introduces a significant overhead in terms of implementing two or more versions of the same algorithm and testing their outputs and is more suitable for mission-critical systems[12].

Another testing methodology called Metamorphic testing introduced by Chen et. al can be used for testing ML programs instead. It addresses some of these problems while testing the "non-testable programs" without significant overhead. The idea behind Metamorphic testing is that it easier to compare and understand the relationship between outputs than to compare and understand input-output behavior. For a prototype example: To test a program which implements the *sin* function, the output from the program for value of *sin* at x could be compare to the real value

of $sin(x)$ or, the mathematical property of $sin(x) = sin(\pi - x)$ can be exploited to verify the correct implementation of sin function. Such relation ($sin(x) = sin(\pi - x)$) are called Metamorphic Relations. Metamorphic testing makes use of metamorphic relations where an input relation is used to generate new input test cases from existing test data, and an output relation is used to compare the outputs produced by the test cases.

In this paper, we will explore the types of guarantees one can expect a machine learning model to possess because of the properties that the underlying algorithm of the implementation.

In this study we will answer the following research questions:

1. How does metamorphic testing compare to other testing methodologies for testing machine learning algorithms?

2. How sensitive are the predictions made by ML algorithms to the change in input data?

3. At what point does the algorithm fail to produce correct classifications on distorted inputs?

Chapter 2

Literature Review

2.1 Testing in the Presence of Uncertainty [4]

Uncertainty is present in most systems that are built today, whether introduced by human decisions, machine learning algorithms, external libraries, or sensing variability mostly due to the need to support deep interactions between interconnected software systems and their users and execution environments. In the context of software testing, uncertainty increases the ambiguity of what constitutes the input space and what is deemed as acceptable behavior, which are two key attributes of a test. In this paper, the authors explore the potential directions for dealing with uncertainty during testing. Two ways to deal with uncertainty are to exercise more control over the inputs provided to the unit under test and to constrain the testing environment. Some uncertainties like those present in systems that deal with the physical world are aleatoric; that is, they cannot be known precisely because of their inherent variability and noise. Epistemic uncertainties are those that could be resolved with enough effort (e.g., over-engineering a sensor-based system to minimize inference errors about the environment)— and resolving them are becoming more numerous and costly, as the

systems are becoming increasingly complex. A complementary way to deal with uncertainty at the other end of the testing process is to develop more sophisticated oracles. If the oracles are relaxed too much to tolerate the variations introduced by uncertainties, there is an inherent risk for the uncertain behavior to mask a fault. On the other hand, if the oracles are too narrow it may cause them to generate false positives, resulting in wasted developer's time. The authors provide a set of requirements for adequate handling of uncertainty in testing:

1. Specifying input distributions and generating inputs with the help of the distributions instead of using discrete inputs.

2. Probabilistic oracles that can help distinguish acceptable from unacceptable misbehaviors. These models will also provide the specification of the likelihood of results.

3. Richer models to represent system and environment uncertainty, so that uncertainties can be connected to test requirements and outcomes. This will help with automated uncertainty quantification.

Not all systems will impose these requirements, and not all techniques will satisfy them. In fact, they expect it will be necessary to develop a suite of techniques for dealing with uncertainty, selected according to system characteristics and the particular forms of uncertainty the system embodies and based on the use of stochastic models and quantitative analyses.

2.2 On Testing Non-testable Programs [12]

The current testing research activities fall into three categories:

1. Develop a sound theoretical basis for testing.
2. Devise and improve testing methodologies, especially the mechanizable ones.
3. Define accurate measurement criteria for testing data adequately.

An oracle is a system that determines the correctness of the solution by comparing the system's output to the one that it predicts. Programs for which such oracles do not exist are called 'non-testable'. The term non-testable is used, from the point of view of correctness testing. If one cannot decide whether or not the output is correct or have to spend some extraordinary amount of resources to do so, testing those systems may not be worth it. Non-testable programs can be usually classified into three categories:

1. Programs that were written to determine the correct answer.
2. Programs that produce a lot of outputs such that it is hard to verify all of them.
3. Programs where tester have a misconception (tester believes that he has the oracle even though he might not).

In absence of oracles, the ideal way to test a program is through Pseudo Oracles/Dual Coding. But, due to a great deal of overhead involved pseudo-oracles may not be practical for every situation. A different, and frequently employed course of action is to run the program on 'simplified' data for which the correctness of the results can be accurately and readily determined. The tester then extrapolates from the correctness of the test results on these simple cases to correctness for more complicated cases. In this case, we are deliberately omitting test cases even though these cases may have been identified as important. They are not being omitted because it is not expected that they will yield substantial additional information, but rather because they are

impossible, too difficult, or too expensive to check. The problem with using simple test cases is very obvious i.e. it is common for central cases to work perfectly whereas boundary cases to cause errors. Although non-testable programs occur in all areas of data processing, the problem is undoubtedly most acute in the area of numerical computations, particularly where floating point arithmetic is used. While performing mathematical computations, errors from three sources can creep in:

1. The mathematical model used to do the computations.
2. Programs written to implement the computation.
3. The features of the environment like round-off, floating point operations etc.

Even in the absence of oracles the users often have a ballpark idea of what the correct answer would look like without knowing the correct answer. In such cases, we make use of partial oracles. It is relatively easier to test the systems on simpler inputs for which the output is known. There is rarely a single correct answer in these types of computations. Rather, the goal is generally an approximation which is within a designated tolerance of the exact solution. The authors finally make five recommendation for items to be considered as a part of documentation during testing.

1. The criteria used to select the test data.
2. The degree to which the criteria were fulfilled.
3. The test data, the program ran on.
4. The output of each of each test datum.
5. How the results were determined to be correct or acceptable.

Although these recommendations do not solve the problem of non-testable programs, however, they do provide information on whether the program should be considered adequately tested or not.

2.3 Metamorphic Testing

2.3.1 A Survey on Metamorphic Testing [11]

Segura et al. did an extensive review of metamorphic testing with 119 papers published between 1998 and 2015 to answer four important research questions:

1. RQ1: What improvements to the technique have been made?
2. RQ2: What are its known application domains?
3. RQ3: How are experimental evaluations performed?
4. RQ4: What are the future research challenges?

2.3.1.1 Properties of good metamorphic relations

To answer the first question they first studied the properties of effective metamorphic relations. To select the most effective metamorphic relations to detect faults one must have.

- A good understanding of the problem domain since good metamorphic relations are usually strongly inspired by the semantics of the program under test.
- Metamorphic relations that make execution of the follow–up test case as different as possible from the source test case.

- Metamorphic relations derived from specific parts of the system since they are more effective than those targeting the whole system.

- Formally described metamorphic relations. In particular, a metamorphic relation should be a 3-tuple composed of *i*) relation between the inputs of the source and follow–up test cases, *ii*) relation between the outputs of source and follow-up test cases, and *iii*) program function.

2.3.1.2 Construction of metamorphic relations

Composition of Metamorphic Relations (CMR) can be used to construct new metamorphic relations by combining several existing relations. The rationale behind this method is that the resulting relations should embed all properties of the original metamorphic relations, and thus they should provide similar effectiveness with a fewer number of metamorphic relations and test executions. Two metamorphic relations are considered "compositable" if the follow-up test cases of one of the relations can always be used as source test case of the other. The composition is sensitive to the order of metamorphic relations and generalizable to any number of them. Determining whether two metamorphic relations are composable is a manual task. Chen et al. [2] presented a specification–based tool called METRIC for the identification of metamorphic relations. In this framework, the program specification is used to partition the input domain in terms of categories, choices and complete test frames. The results of an empirical study with 19 participants suggest that METRIC is effective and efficient at identifying metamorphic relations.

2.3.1.3 Generation of source test cases

Gotlieb and Botella [5] presented a framework called Automated Metamorphic Testing (AMT) to automatically generate test data for metamorphic relations. Given the source code of a program written in C and a metamorphic relation, AMT tries to find test cases that violate the relation. The underlying method is based on the translation of the code into an equivalent constraint logic program over finite domains. Other techniques like "special values" and random testing can also be used as source test cases for metamorphic testing. Genetic algorithms have also been used for the selection of source test cases, to maximize the paths traversed in the program under test.

2.3.1.4 Execution of metamorphic test cases

The execution of a metamorphic test case is typically performed in two steps. First, a follow–up test case is generated by applying a transformation to the inputs of a source test case. Second, source and follow–up test cases are executed, checking whether their outputs violate the metamorphic relation.

Iterative Metamorphic Testing (IMT) can be used to systematically exploit more information from metamorphic tests, by applying metamorphic relations iteratively. In IMT, a sequence of metamorphic relations is applied in a chain style, by reusing the follow–up test case of each metamorphic relation as the source test case of the next metamorphic relation.

Murphy et al. [9] presented a framework named Amsterdam for the automated application of metamorphic testing. The tool takes as inputs the program under test and a set of metamorphic relations, defined in an XML file. Then, Amsterdam automatically runs the program, applies the metamorphic relations and checks the

results.

2.3.1.5 Application domains

To answer the second question the authors selected the papers where the main objective was a case study. They identified that the most popular use of metamorphic testing was in web services, followed by computer graphics, simulation and modelling, and, embedded systems. They also found some other domains of application like financial software, optimization programs, and encryption programs. Some of the research challenges identified by the authors are:

- Lack of guidelines, with step–by–step process to guide testers, experts, and beginners, in the construction of good metamorphic relations.
- Prioritisation and minimisation of metamorphic relations: It is worth mentioning that test case minimisation is a NP–hard problem and therefore heuristic techniques should be explored.
- Generation of likely metamorphic relations.
- Combination of metamorphic relations.
- Automated generation of best possible source test cases.
- Lack of metamorphic testing tools.

2.3.2 Automatic System Testing of Programs without Test Oracles [9]

In this paper the authors have demonstrated the usefulness of metamorphic testing in assessing the quality of applications without test oracles. Comparing the outputs

of the morphed data still remains a challenge especially if the data set is large or not in human readable format. The authors presented an approach called "Automated Metamorphic System Testing" to automate the metamorphic testing by considering the system as a blackbox and checking if the metamorphic properties holds after execution of the system. They also present another approach "Heuristic Metamorphic Testing" to reduce false positives and address some non-determinism. Unlike in the previous papers, here, the authors are focusing to improve the metamorphic testing technique itself. They list some benefits of using metamorphic testing: it can be used on broader domain of applications that display metamorphic properties, and it treats the application under test as a black box and does not require detailed understanding of the source code. They then list some of the limitations of using metamorphic testing:

- Manual transformation of large input data can be laborious and error-prone. They need special tools to transform the input.
- Comparing the outputs(some of which may be very large and/or in not human-readable format) of the input data can be tedious.
- Floating point calculations can also lead to imprecision even though the calculations are programmatically correct.
- Coming up with the initial test-cases is also a challenge as some defects may only occur under certain inputs.

Automated Metamorphic System Testing: This technique can be used to test the application in development environment as well as in production as long as the users are only provided the output from the original execution and not the result from transformed input. In this model i) Metamorphic properties are specified by

the tester and applied to the input. *ii*) The original input is fed into the application which is treated as a black-box and a transformation of the input is also generated. *iii*) That transformed input is fed into a separate instance of the application running in a separate sandbox. *iv*) When the invocations are finished, the results are compared and if they do not match according to the specifications, there is an error. Tester need not write any code and only needs to specify the metamorphic properties. They don't need to know the source code or other implementation details. Amsterdam framework: The metamorphic properties are specified using XML file. The specification consists of three parts: *i*) how to transform the input, *ii*) how to execute the program, and, *iii*) how to compare the outputs.

Heuristic Metamorphic testing: This method allows for small differences in outputs, in a meaningful way according to the application being used to address the problems of false positives and non-determinism. Imprecisions in floating point calculation and representation of irrational number such as may result in failure of metamorphic testing even if the implementation is correct. If two outputs are close enough they are considered the same. The definition of close enough depends on the application and in complex applications checking semantic similarity may also be required.

2.4 Overview of Machine Learning Algorithms

2.4.1 Supervised Sequence Labelling with Recurrent Neural Networks [6]

Artificial neural networks (ANNs) were originally developed as mathematical models of the information processing capabilities of biological brains. Although it is now clear

that ANNs bear little resemblance to real biological neurons, they enjoy continuing popularity as pattern classifiers. The basic structure of an ANN is a network of small processing units, or nodes, joined to each other by weighted connections. In terms of the original biological model, the nodes represent neurons, and the connection weights represent the strength of the synapses between the neurons. The network is activated by providing an input to some or all of the nodes, and this activation then spreads throughout the network along the weighted connections. ANNs without cycles are referred to as feedforward neural networks (FNNs). Well known examples of FNNs include perceptrons, radial basis function networks, Kohonen maps and Hopfield nets. The most widely used form of FNN is the multilayer perceptron(MLP). It has been proven that an MLP with a single hidden layer containing a sufficient number of nonlinear units can approximate any continuous function on a compact input domain to arbitrary precision. For this reason MLPs are said to be universal function approximators. At each unit in a layer the activation function θ_h is applied, yielding the final activation b_h of the unit. The most common choices for neural network activation function are the hyperbolic tangent and the logistic sigmoid. An important feature of both tanh and the logistic sigmoid is their nonlinearity. Nonlinear neural networks are more powerful than linear ones since they can, for example, find nonlinear classification boundaries and model nonlinear equations. Another key property is that both functions are differentiable, which allows the network to be trained with gradient descent. Because of the way they reduce an infinite input domain to a finite output range, neural network activation functions are sometimes referred to as squashing functions. The output vector y of an MLP is given by the activation of the units in the output layer. The network input a_k to each output unit k is calculated by summing over the units connected to it, exactly as for a hidden unit. Both the number of units in the output layer and the choice of output activation

function depend on the task the network is applied to. For classification problems with $K > 2$ classes, the convention is to have K output units, and normalise the output activations with the softmax function to obtain the class probabilities which is also known as a multinomial logit model.

2.5 Testing Machine Learning Programs

2.5.1 Properties of Machine Learning Applications for Use in Metamorphic Testing [8]

In the absence of a reliable oracle to indicate the correct output for arbitrary inputs, machine learning programs are often very hard to test. Non-testable programs can be tested in one of the two ways:

- Creating multiple implementations of the same program and testing them on same inputs and comparing the results. If the outputs are not same then either of the implementations can contain error. This approach is called pseudo-oracle.
- In absence of multiple oracle, metamorphic testing can be used. In metamorphic testing, the input is modified using a metamorphic relation such that the two sets of input will generate similar outputs. If similar outputs are not observed then there must be a defect.

The main challenge with metamorphic testing is to come up with the metamorphic relations to transform inputs since such coming up with such relations require domain knowledge and/or familiarity with the implementation. In this paper the authors seek to create a taxonomy of metamorphic relationships that can be applied to the input data for both supervised and unsupervised machine learning softwares. These set of

properties can be applied to define the metamorphic relationships so that metamorphic testing can be used as a general testing method for machine learning applications. The problem with some of the current machine learning frameworks like: Weka and Orange is that they compare the quality of results but don't evaluate the correctness of the results. The authors apply metamorphic testing to three ML applications: MartiRank, SVM-Light, PAYL. MartiRank is a supervised ML algorithm that applies segmentation and sorting of the input data to create a model. The algorithm then performs similar operations from the model on the test data to produce a ranking list. SVM-Light is an open-source implementation of SVM that also has a ranking mode. The authors also investigated an intrusion detection system called PAYL. PAYL is an unsupervised machine learning system. It's dataset simply consist of TCP/IP network payloads(stream of bytes) without any label or classification. Based on the analysis of MartiRank algorithm, the authors realized that the actual values of the attributes were not very important but their relative values determined the model. Thus, adding a constant value to every attribute or multiplying each attribute with a positive number, should not affect the model and generate the same ranking as before. Thus, the metamorphic properties identified were: addition and multiplication. Applying the model on two sets of data, one of which created from the other, either by multiplying a positive number or, adding a constant number, should not change the ranking. Changing the order of examples should not affect the model or ranking since the algorithm sorts the inputs thus, MartiRank also has permutative metamorphic property. Multiplying the data by a negative constant value will create a new sorting order which can easily predicted. The only change to the model will be the sorting direction i.e. the algorithm will change the sorting direction but keep the sorting order intact. Thus, MartiRank also displays an invertive metamorphic property where the output can be predicted by taking the opposite of input. Mar-

tiRank also includes inclusive and exclusive metamorphic properties. Knowing the model can help predict the position of any new elements.

2.5.2 Application of Metamorphic Testing to Supervised Classifiers [13]

Building on the previous paper, the authors explore the metamorphic relations based on expected behavior of given machine learning problems. They present a case study on Weka, a popular machine learning framework, which is also the foundation for computational science tools such as BioWeka in bioinformatics. In this paper the authors explore k-Nearest Neighbors and Naive Bayes classifier algorithms. Previously, they researched on Support Vector Machines. In this paper the authors seek to identify the metamorphic relations for the two algorithms (kNN and NBC). NBC and kNN both calculates the mean and standard deviation of the input data. Thus, the metamorphic relations identified are: Permuting the order of input data does not affect the mean or standard deviation. Multiplying the data with -1 does not affect the standard deviation since, the deviation from the mean will still be the same. Multiplying the data with some other positive number will increase the standard deviation by the same amount. Thus, the output will still be predictable. The authors then, define the metamorphic relations that a classification algorithm is expected to exhibit:

1. Consistency with affine transformation.
2. Permutation of class labels.
3. Permutation of attributes.
4. Addition of uninformative/informative attributes.

5. Consistency with re-prediction.

6. Addition of training samples.

7. Addition of classes by duplicating/re-labeling samples.

8. Removal of classes/samples.

Next, the authors introduce the notion of validation and verification. Validation refers to choosing the most appropriate algorithm to solve a problem. Verification refers to whether the implemented algorithm is correct or not. Current, software testing methods have not addressed the problem of validation and only focus on verification. The authors then performed an experiment to verify the correctness of Weka. They created a set of random input data and used the above metamorphic relationships to generate another set of inputs. Upon running the inputs on both the algorithms they realized only a subset of MRs were a necessary property of the corresponding algorithm. It was observed that several MRs violated NBC algorithms. Violations in the MRs that are necessary properties imply defects in implementation. In the case of kNN algorithm, none of the necessary MRs were violated which means that there are no implementation error as per the testing.

2.5.3 Dataset Coverage for Testing Machine Learning Computer Programs [10]

Recently, computer programs for Big Data analytics or statistical machine learning have become essential components of intelligent software systems. Test oracles are rarely available for them, and this unavailability of test oracles is known as the oracle problem. Machine learning programs are a typical instance of non-testable programs and are of the known unknowns type. Metamorphic testing (MT) is a method for

tackling the oracle problem. Metamorphic relations (MR) play a role as pseudo oracles to check whether executions of the same program differ for two different test inputs. The test inputs are related by translation functions derived from metamorphic properties so that the relationship between the two results is predictable. If the results coincide with each other, the program behavior is relatively correct. This paper studies the characteristics of the supervised learning classifiers (SUT). Identifying Quasi-testable Core: A program component, function or procedure, is quasi-testable if we have appropriate pseudo oracles or metamorphic relations. The result of the program execution embodies uncertainty because the output is accompanied by the statistical classification performance. The classifier itself is non-testable. However, pseudo oracles with a MT can be used for testing.

Dataset Coverage: Test coverage is essential in software testing because it is a basis to measure how much of the SUT is checked with a set of the input test data. The graph coverage is the most popular model for software testing because it captures the structural characteristics of software artifacts, such as control-flows or data-flows of a computer program. The paper introduces the notion of dataset coverage to focus on the characteristics of the population distribution in the training dataset. However, complete coverage is not possible. The number of possible populations in datasets is also infinite.

SVM: A support vector machine (SVM) is a supervised machine learning classifier. The support vectors lie on the dotted hyperplanes parallel to the separating hyperplane. The margin, the minimum gap between the supporting hyperplane and the separating hyperplane, is chosen to be maximum. The pseudo code is a common, abstract description of implemented SMO computer programs. Because SMO is an algorithm for solving the SVM optimization problem, it corresponds to the model constructor and is an abstract version of the quasi-testable core.

Main tasks:

- Obtain pseudo oracles.
  - MR for Pseudo Oracles: Various combinations of the dataset is obtained by reordering the dataset.
  - MR for Dataset Generation: Dataset is increased to increase the population of the input dataset.
- Generate data points that achieve the required dataset coverage: In order that the result is predictable, the population distribution of the initial dataset is simple enough to contain linearly separable data points. Then a series of tests with pseudo oracles that are obtained based on appropriate metamorphic properties is conducted. Then dataset is extended by adding new data points to calculate a new hyperplane.

A similar metamorphic testing approach can be applied to K-nearest neighbors and naive Bayes classifiers. Since testing the whole program at once is not always possible choosing a right SUT from a non-testable program has a large impact on testing activities.

Chapter 3

Proposed Work

3.1 Setting up the Test Environment

3.1.1 Jupyter

The Jupyter Notebook is an open-source web application that supports data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, etc. by allowing us to create and share documents that contain live code, equations, visualizations and narrative text. Jupyter allows for displaying the result of computation inline in the form of rich media (SVG, LaTeX, etc.). The Jupyter notebook combines two components:

- **A web application**: a browser-based tool for interactive authoring of documents which combine explanatory text, mathematics, computations and their rich media output.
- **Notebook documents**: a representation of all content visible in the web application, including inputs and outputs of the computations, explanatory text, mathematics, images, and rich media representations of objects.

Jupyter notebook is gaining rapid popularity in the field of data science for making sharing documentation and codes for replication very easy. In this project, the codes are written in python notebook which can be accessed from http://github.com/.

3.1.2 Tensorflow

TensorFlow™ is an open source software library developed within Google's AI organization by the Google Brain team with a strong support for machine learning and deep learning. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. It is being used at a number of well known companies like Uber, Google, AMD, etc. for high performance numerical computation and machine learning. While TensorFlow is capable of handling a wide range of tasks, it is mainly designed for deep neural network models. It will serve as baseline to test the metamorphic relations identified in section 3.3.

3.1.3 MNIST Dataset

The MNIST database of handwritten digits maintained by Yann LeCun [7], has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting. The MNIST database was constructed from NIST's Special Database 3 and Special Database 1 which contain binary images of handwritten digits. The original black and white (bilevel) images from NIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The

images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field.

3.1.3.1 Format of Dataset

The data is stored in a very simple file format designed for storing vectors and multidimensional matrices. All the integers in the files are stored in the MSB first (high endian) format used by most non-Intel processors. There are 4 files:

- train-images-idx3-ubyte: training set images
- train-labels-idx1-ubyte: training set labels
- t10k-images-idx3-ubyte: test set images
- t10k-labels-idx1-ubyte: test set labels

The format of training and test files are described in the following table.

offset	**type**	**value**	**description**
0000	32 bit integer	0x00000801(2049)	magic number (MSB first)
0004	32 bit integer	60000	number of items
0008	unsigned byte	??	label
0009	unsigned byte	??	label
.....			
xxxx	unsigned byte	??	label
The label values are 0 to 9.			

Table 31: Training data file format.

3.1.4 EMNIST Dataset

The EMNIST dataset is a set of handwritten character digits derived from the NIST Special Database 19 and converted to a 28x28 pixel image format and dataset structure that directly matches the MNIST dataset.

offset	type	value	description
0000	32 bit integer	0x00000803(2051)	magic number (MSB first)
0004	32 bit integer	60000	number of images
0008	unsigned byte	28	number of rows
0012	unsigned byte	28	number of columns
0016	unsigned byte	??	pixel
0017	unsigned byte	??	pixel
.....			
xxxx	unsigned byte	??	pixel
Pixels are organized row-wise. Pixel values are 0 to 255. 0 means background (white), 255 means foreground (black).			

Table 32: Test data file format.

3.1.4.1 Dataset Summary

There are six different splits provided in this dataset. A short summary of the dataset is provided below:

- EMNIST ByClass: 814,255 characters. 62 unbalanced classes.
- EMNIST ByMerge: 814,255 characters. 47 unbalanced classes.
- EMNIST Balanced: 131,600 characters. 47 balanced classes.
- EMNIST Letters: 145,600 characters. 26 balanced classes.
- EMNIST Digits: 280,000 characters. 10 balanced classes.
- EMNIST MNIST: 70,000 characters. 10 balanced classes.

The full complement of the NIST Special Database 19 is available in the ByClass and ByMerge splits. The EMNIST Balanced dataset contains a set of characters with an equal number of samples per class. The EMNIST Letters dataset merges a balanced set of the uppercase and lowercase letters into a single 26-class task. The EMNIST Digits and EMNIST MNIST dataset provide balanced handwritten digit datasets directly compatible with the original MNIST dataset.

3.2 Subject Programs

Various models from TensorFlow is being used at different organizations like Mozilla, Google, Stanford University, etc. in different domains extending from speech recognition to computer vision. To evaluate the accuracy of some of the popular algorithms used in supervised classification we decided to implement metamorphic testing of:

- K-Nearest Neighbors
- Neural Networks

3.3 Identification of Metamorphic Relations

Murphy et al. [8] identified six metamorphic properties that they believe exist in many machine learning applications: additive, multiplicative, permutative, invertive, inclusive, and exclusive. We expect the neural networks to be resistant to these transformations and produce the same classifications before and after applying tranformations.

3.4 Generation of source test cases

Source test data

3.5 Generation of follow-up test cases

Follow-up data after applying MR

3.6 Evaluation metrics

How to evaluate

Chapter 4

Work Plan

Gantt Chart	**2018**						
	Jan	Feb	Mar	Apr	May	June	July
Environment Setup							
Jupyter Notebook							
TensorFlow							
Dataset							
Properties selection							
Algorithm							
Metamorphic Relations							
Implementations							
K-NN							
Neural Networks							
Data collection and analysis							
Writing							
Literature Review							
Proposal							
Final Report							

Table 41: Project schedule as of May 2018.

The colors indicate the status of the task which can be one of the following: *completed* (green), *in progress* (yellow), *not started* (red), and *as needed* (blue).

Chapter 5

Results

5.1 Overall Accuracy

5.2 Digit-by Accuracy

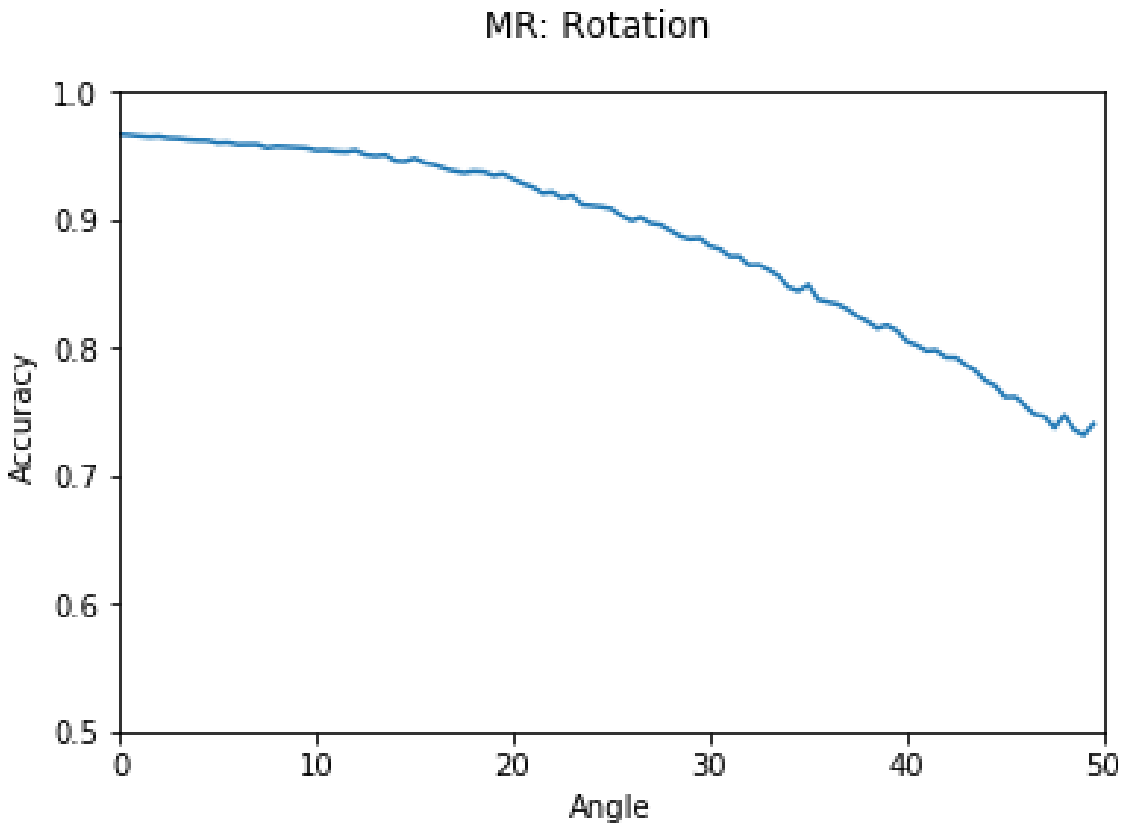

Figure 51: MR: Rotation.

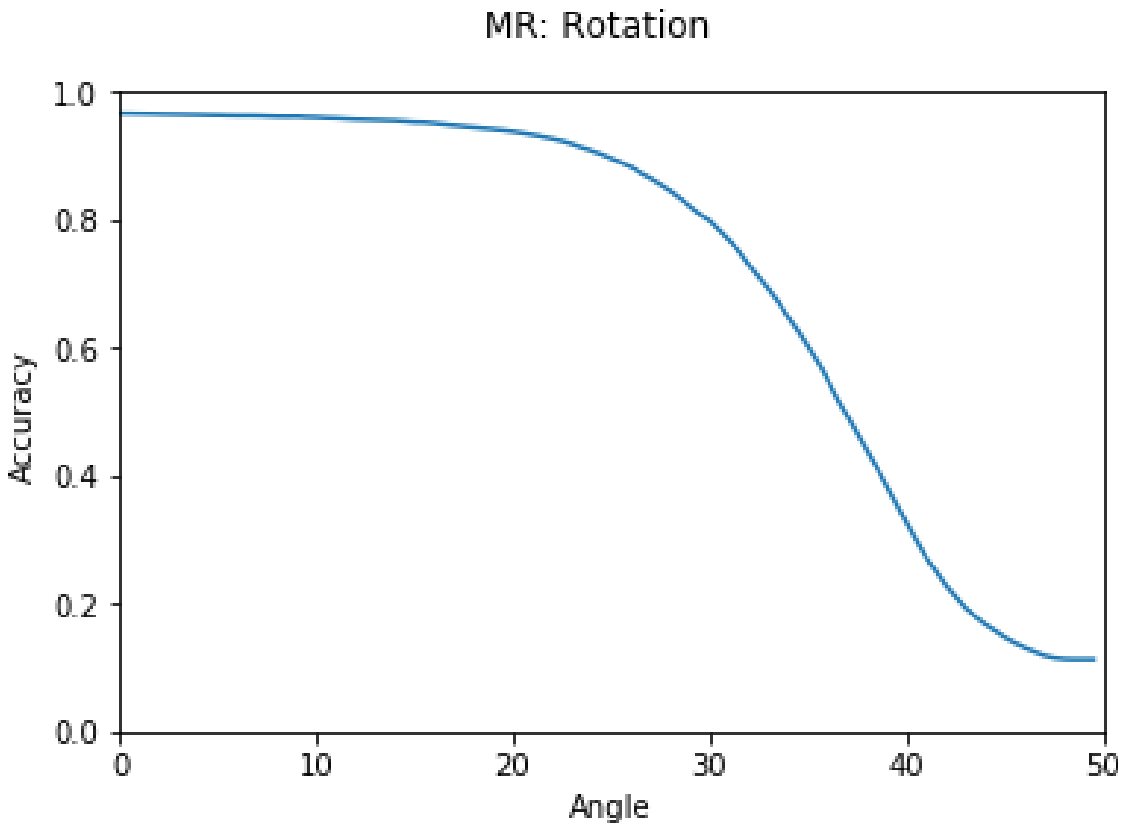

Figure 52: MR: Shading.

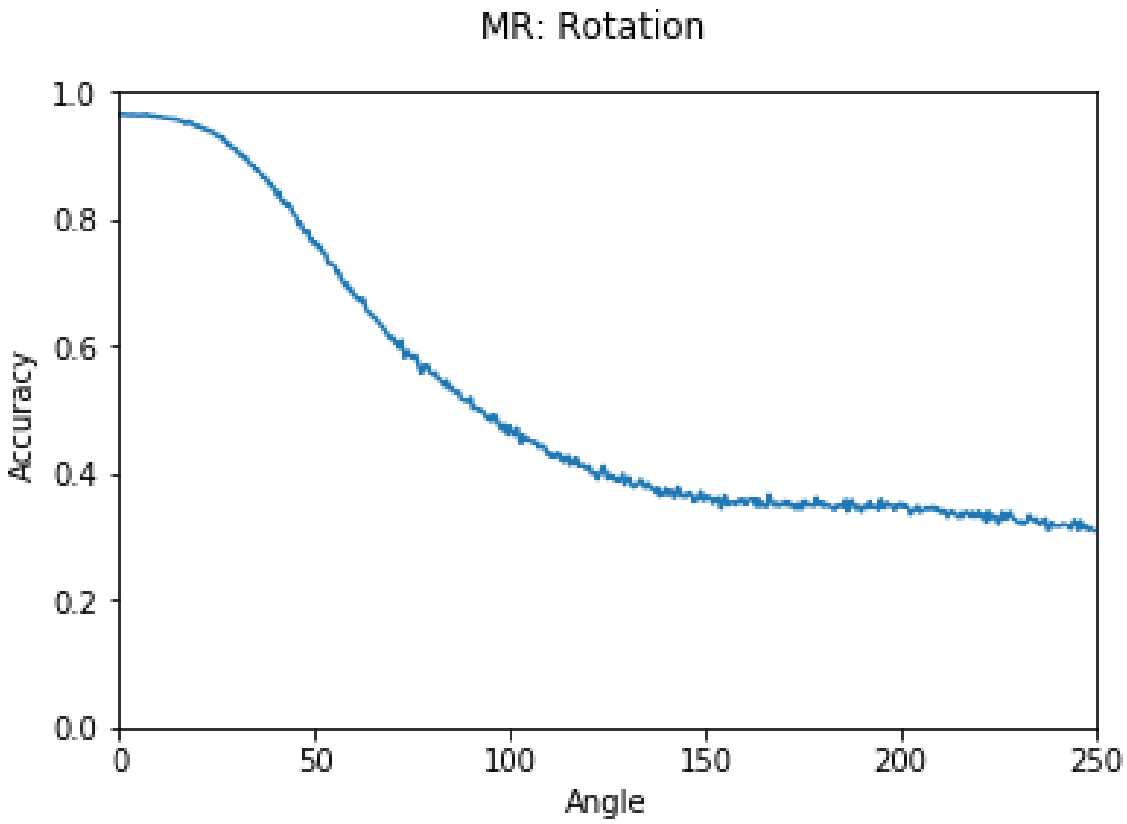

Figure 53: MR: Sheering.

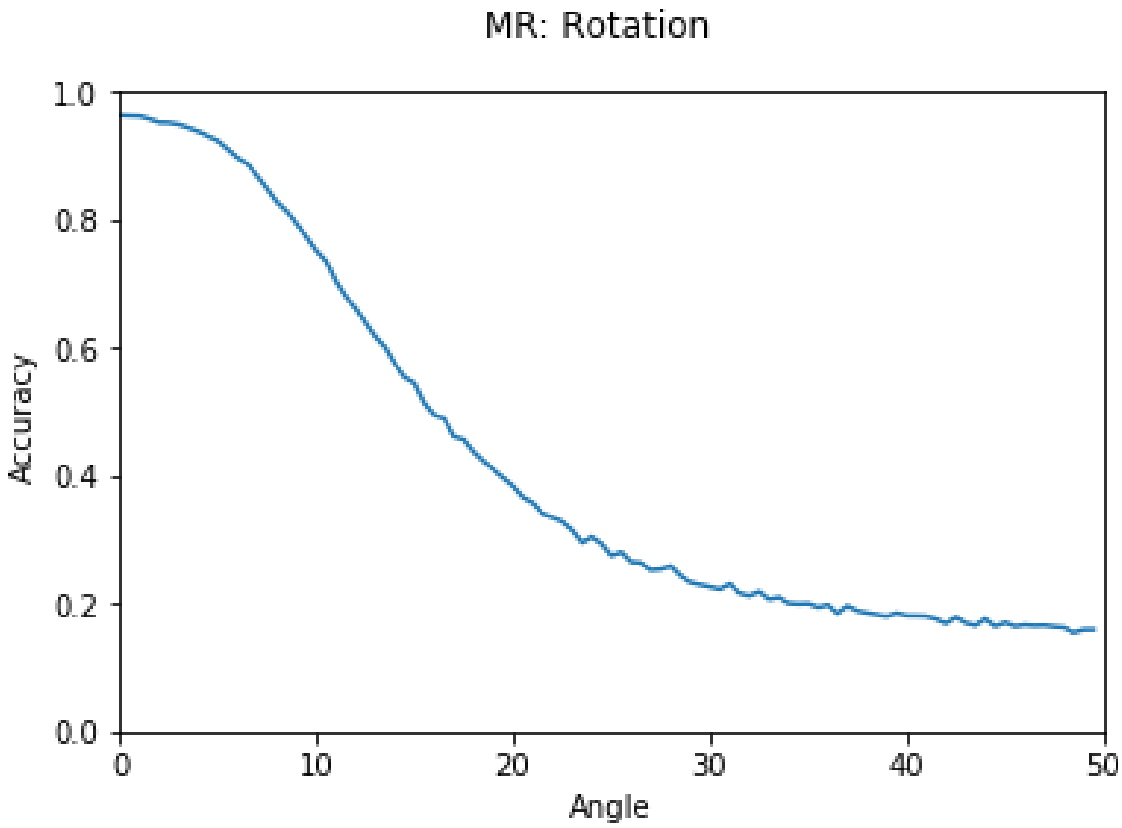

Figure 54: MR: Shifting.

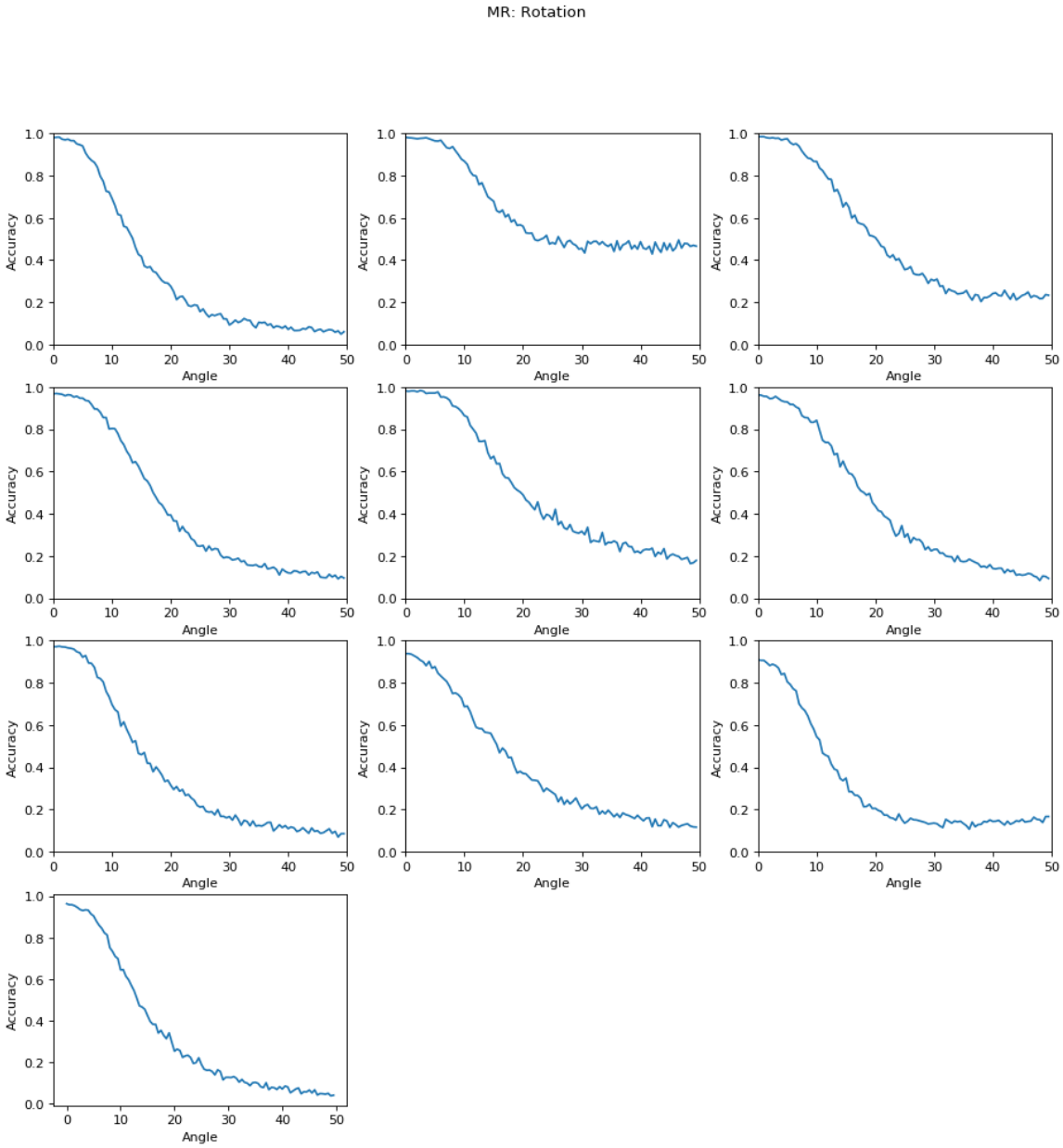

Figure 55: MR: Rotation.

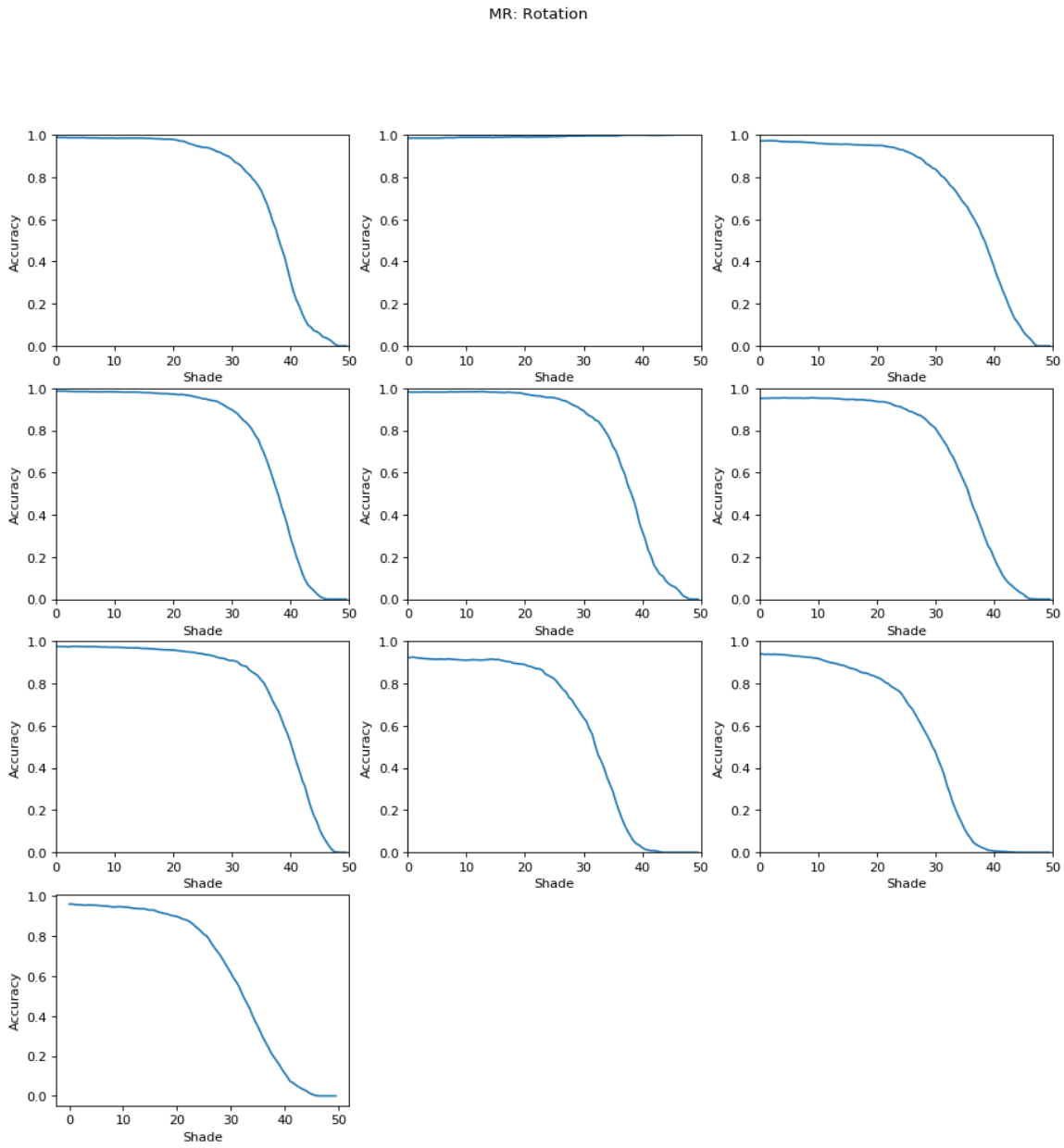

Figure 56: MR: Shading.

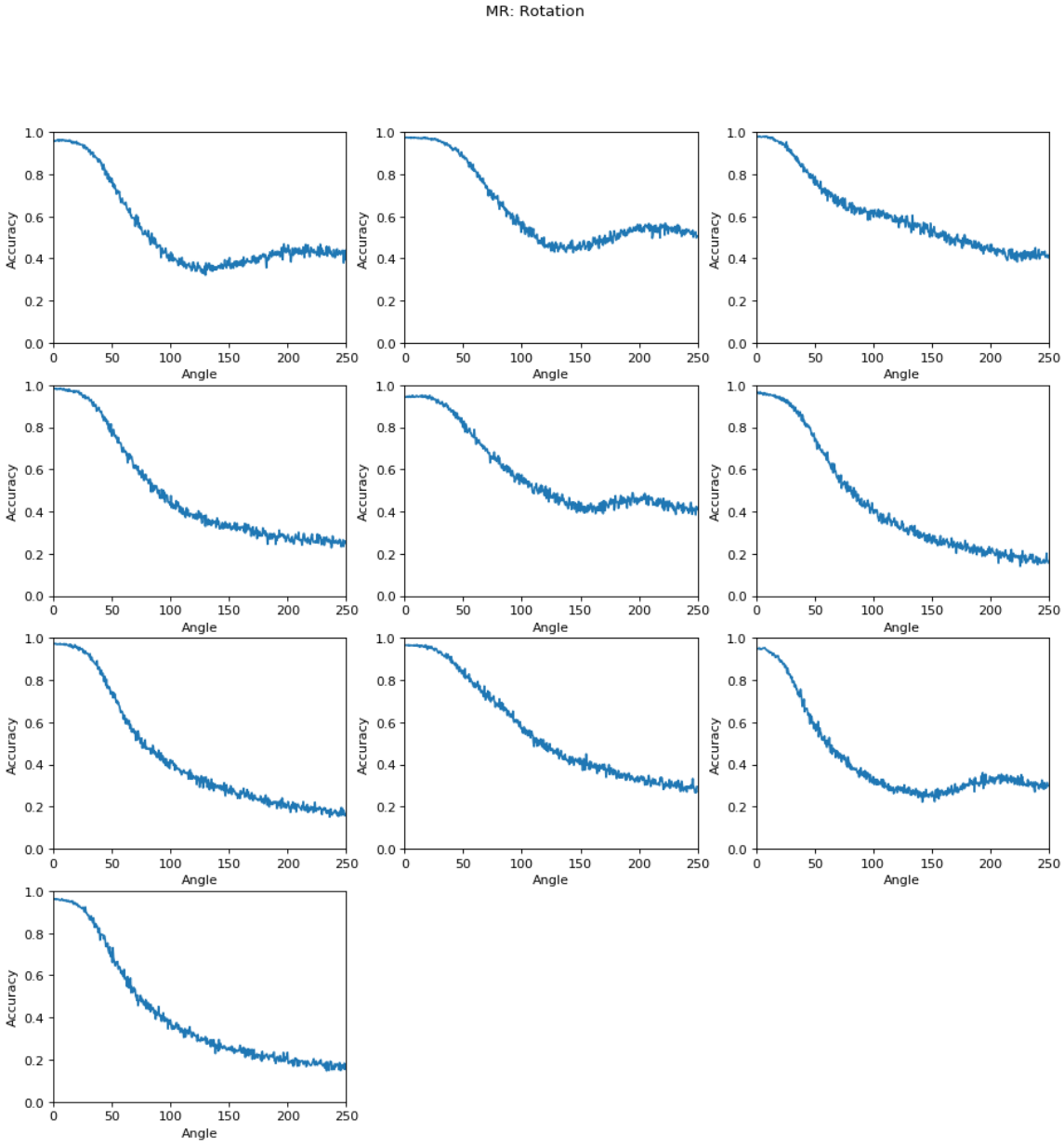

Figure 57: MR: Sheering.

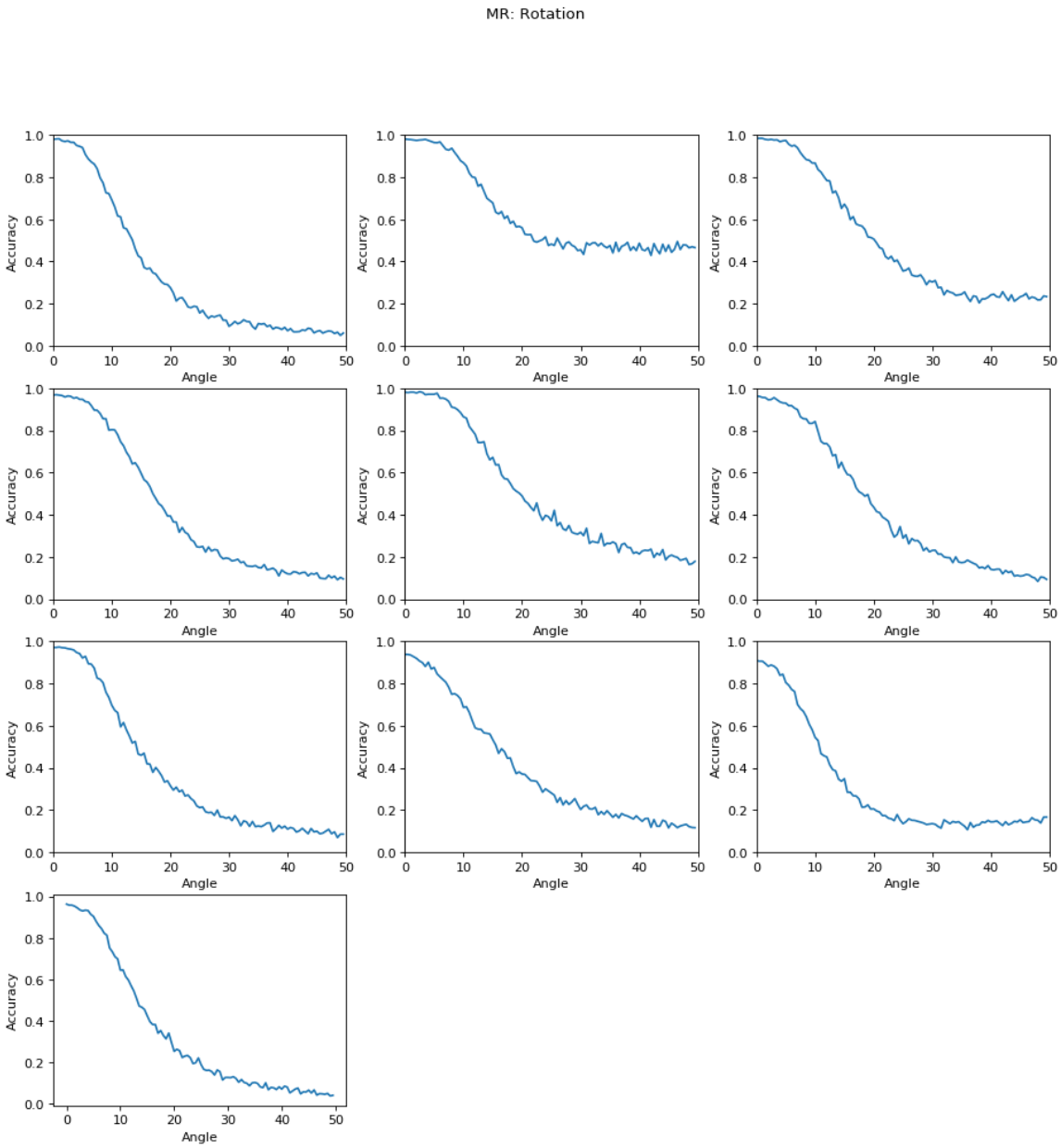

Figure 58: MR: Shifting.

Chapter 6

Synthesis Matrix

Paper	MRs used	Algorithms/Test Cases	Result of MT
Application of Metamorphic Testing to Supervised Classifiers	MRs for classification algorithms: 1. Consistency with affine transformation. 2. Permutation of class labels. 3. Permutation of attributes. 4. Addition of uninformative/informative attributes. 5. Consistency with re-prediction. 6. Addition of training samples. 7. Addition of classes by duplicating/re-labeling samples. 8. Removal of classes/samples.	Package: Weka. Test Case: Randomly generated training and test data. The randomly generated data model does not encapsulate any domain knowledge.	Only a subset of MRs were a necessary property of the corresponding algorithm. Several MRs violated NBC algorithms. In the case of kNN algorithm, none of the necessary MRs were violated.

Continuation of Table 62			
Paper	MRs used	Algorithms/Test Cases	Result of MT
Dataset Coverage for Testing Machine Learning Computer Programs	MR for Pseudo Oracles: 1. Reorder Data Points 2. Reverse Labels 3. Reorder Attributes 4. Add a Constant Attribute 5. Change Attribute Values MR for Dataset Generation: 1. Reduce Margin 2. Insert Noise 3. Insert Separable 4. Inconsistent Labels	Developed a Java program of an SVM classifier LIBSVM, used as a pseudo oracle. Test Case: Java pseudo random number generator.	The Java program passed the tests in terms of this criteria. When the model constructor component was tested it had some faults that the proposed method identifies.
Properties of Machine Learning Applications for Use in Metamorphic Testing	MartiRank: 1. Additive 2. Multiplicative 3. Permutative 4. Invertive 5. Inclusive 6. Exclusive 7. SVM-Light: 8. Exclusive 9. Inclusive 10. Permutative PAYL: 1. Additive 2. Multiplicative 3. Permutative 4. Invertive 5. Inclusive 6. Exclusive	Packages: MartiRank, SVM-Light, PAYL.	MartiRank: the implementation produced inconsistent results when a negative label existed. SVM-Light not tested because it is inefficient to run the quadratic optimization algorithm on the full data set. PAYL exhibits the same six metamorphic properties as MartiRank.

Table 61: Synthesis Matrix.

Paper	Domain and Algo tested	MRs investigated	New Innovative
Semi-Proving: An Integrated Method for Program Proving, Testing, and Debugging	"Siemens suite of programs" specifically "replace program": performs regular expression matching and substitutions.	Given the text 'ab', replacing 'a' with 'x' is equivalent to replacing non-'b' with 'x'. 'char' and '[char]' are equivalent with the exception of a few wildcards. Regular expressions that involve square brackets are equivalent. Use '?*' to match the entire input string.	Proof of concept. Supports automatic debugging through the identification of constraints for failure-causing inputs.
Metamorphic Testing Integer Overflow Faults of Mission Critical Program: A Case Study	Integer Overflow Faults in TCAS: tcas.c	Additive	Proof of concept. Formal definition of MR, original test case, follow-up test case, input relation and output relation. TCAS case study for demonstration.
Metamorphic Testing for Software Quality Assessment: A Study of Search Engines	Search engines: 1. Google search 2. Baidu 3. Bing	1. MPSite 2. MPTitle 3. MPReverseJD 4. SwapJD 5. Top1Absent	Validation, verification as well as quality assessment.

Table 62: Synthesis Matrix.

Continuation of Table 62			
Paper	Domain and Algo tested	MRs investigated	New Innovative
Automatic System Testing of Programs without Test Oracles	Machine learning. 1. SVM: Sequential Minimal Optimization (SMO). 2. C4.5 3. MartiRank: Area Under the Curve	1. Permute 2. Multiply 3. Add 4. Negate	Heuristic Metamorphic Testing. Automated Metamorphic System Testing.
Metamorphic Testing and Beyond	A laplace equation with Dirichlet boundary Conditions: "alternating direction implicit" method	Beyond identity relations: Convergence	Selecting useful MRs. Stronger may not necessarily be better than weaker ones. Provides guidelines.
A Metamorphic Testing Approach for Online Testing of Service-Oriented Software Applications	service-oriented calculator: arithmetic operators	1. Commutative 2. Associative	Methodological steps for online and offline testing. MT for service oriented applications.

Table 62: Synthesis Matrix.

Bibliography

[1] T. Y. Chen, F. C. Kuo, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing and beyond. In *Proceedings - 11th Annual International Workshop on Software Technology and Engineering Practice, STEP 2003*, pages 94–100, 2004.

[2] Tsong Yueh Chen, Pak-Lok Poon, and Xiaoyuan Xie. METRIC: METamorphic Relation Identification based on the Category-choice framework. *Journal of Systems and Software*, 116:177–190, jun 2016.

[3] Martin D. Davis and Elaine J. Weyuker. Pseudo-oracles for non-testable programs. In *Proceedings of the ACM '81 Conference*, ACM '81, pages 254–257, New York, NY, USA, 1981. ACM.

[4] Sebastian Elbaum and David S. Rosenblum. Known unknowns: Testing in the presence of uncertainty. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 833–836, New York, NY, USA, 2014. ACM.

[5] Arnaud Gotlieb and Bernard Botella. Automated metamorphic testing. In *Proceedings of the 27th Annual International Conference on Computer Software and Applications*, COMPSAC '03, pages 34–, Washington, DC, USA, 2003. IEEE Computer Society.

[6] Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*, volume 385 of *Studies in Computational Intelligence*. Springer, 2012.

[7] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`, 1999. Accessed: 2018-05-30.

[8] Christian Murphy, Gail Kaiser, and Lifeng Hu. Properties of Machine Learning Applications for Use in Metamorphic Testing.

[9] Christian Murphy, Kuang Shen, and Gail Kaiser. Automatic System Testing of Programs without Test Oracles. 2009.

[10] S Nakajima and H N Bui. Dataset Coverage for Testing Machine Learning Computer Programs. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 297–304, 2016.

[11] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortes. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, 2016.

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

[13] Xiaoyuan Xie, Joshua Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. Application of metamorphic testing to supervised classifiers. In *Proceedings - International Conference on Quality Software*, 2009.

[14] Xiaoyuan Xie, Joshua W K Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. Testing and Validating Machine Learning Classifiers by Metamorphic Testing. *J. Syst. Softw.*, 84(4):544–558, apr 2011.