Improving the Dependability of Machine Learning Applications

Christian Murphy and Gail Kaiser, *Member, IEEE*

Abstract—As machine learning (ML) applications become prevalent in various aspects of everyday life, their dependability takes on increasing importance. It is challenging to test such applications, however, because they are intended to learn properties of data sets where the correct answers are not already known. Our work is not concerned with testing how well an ML algorithm learns, but rather seeks to ensure that an application using the algorithm implements the specification correctly and fulfills the users' expectations. These are critical to ensuring the application's dependability. This paper presents three approaches to testing these types of applications. In the first, we create a set of limited test cases for which it is, in fact, possible to predict what the correct output should be. In the second approach, we use random testing to generate large data sets according to parameterization based on the application's equivalence classes. Our third approach is based on metamorphic testing, in which properties of the application are exploited to define transformation functions on the input, such that the new output can easily be predicted based on the original output. Here we discuss these approaches, and our findings from testing the dependability of three real-world ML applications.

Index Terms—Machine Learning, Software Dependability, Software Testing, Quality Assurance, Metamorphic Testing, Random Testing, Non-Testable Programs, Oracle Problem.

1 Introduction

MAKING machine learning (ML) applications dependable presents a challenge because conventional software testing processes do not always apply: in particular, it is difficult to detect subtle errors, faults, defects or anomalies in the ML applications of interest because there is no reliable "test oracle" to indicate what the correct output should be for arbitrary input. The general class of software systems with no reliable test oracle available is sometimes known as "non-testable programs" [1]. These ML applications fall into a category of software that Davis and Weyuker describe 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"* [2]. Formal proofs of an ML algorithm's optimal quality (*e.g.*, its ability to predict well) do not guarantee that an application implements or uses the algorithm correctly, and therefore software testing is needed. Thus, testing these types of applications does not seek to determine whether an ML algorithm learns well, but rather to ensure that an application using the algorithm correctly implements the specification and fulfills the users' expectations.

As machine learning applications become more and more prevalent in various aspects of everyday life [3], it is clear that the dependability of machine learning software takes on increasing importance, whether the tasks are simple things like getting a recommendation for a DVD, or critical tasks like helping doctors perform a medical diagnosis or enabling weather forecasters to more accurately predict the paths of hurricanes. Our concern, then, is in improving the dependability of such applications through software testing. Improved testing of these types of applications can increase availability and reliability, not to mention confidence in correctness, and thus make the applications more dependable.

Here we discuss a methodology for testing ML applications. Of course, in any software testing, it is possible only to show the presence of bugs but not their absence. Usually when input or output equivalence classes are applied to developing test cases, however, the expected output for a given input is known in advance. Our research seeks to address the issue of how to devise test cases that are likely to reveal bugs, and how one can indeed know whether a test actually is revealing a bug, given that we do not know what the output should be in the general case. In other words, although we cannot know for sure when the output is correct, we seek to generate tests cases where we can easily detect in most cases if the output is wrong.

Our methodology consists of three approaches. In the first approach, we hand-craft simple data sets for which we can, in fact, know whether the output that is produced is correct. This is a very limited approach, of course, given that there is no general test oracle for these applications, but by creating a *"niche oracle"* for a small subset of the input domain, we can create a baseline of test cases that at least must be passed before proceeding on to any other testing.

The second approach is based on the notion of random testing [4] [5]. In our approach, we use randomness to generate large data sets, but use paramaterization to guide it towards different equivalence classes for which we can predict aspects of the application's expected

• *The authors are members of the Programming Systems Lab, Department of Computer Science, Columbia University, New York NY 10027. E-mail: {cmurphy, kaiser}@cs.columbia.edu*

behavior, though not the final output. The contribution of this approach is a hybrid that couples the benefits of using randomness with the necessary control over the properties of the testing data.

In the third and final approach, we explore the use of metamorphic testing [6] [7], which is a methodology of reusing input test data to create additional test cases whose outputs can be predicted. In metamorphic testing, the input from test cases that did not reveal any defects is modified in such a manner that it should produce an expected output based on the original, and if it does not, then a defect must exist. In our approach, although we cannot know whether the initial test cases did or did not reveal defects, we can in some circumstances modify the input to create new test cases and then check whether the new output matches our expectations, by use of a built-in "pseudo-oracle" [2].

In this paper, we describe our methodology, and present our findings from applying the approaches to three machine learning applications: MartiRank [8], a ranking implementation of the Martingale Boosting algorithm [9]; an implementation of Support Vector Machines (SVM) [10] called SVM-Light [11]; and the anomaly-based network intrusion detection system PAYL [12]. We have found previously-unknown defects and/or inconsistencies in all three applications, and demonstrate that our work has potential to help improve their dependability.

1.1 Motivation

This line of research began with work in which we addressed the dependability of an ML application commissioned by a company for potential future experimental use in predicting impending electrical device failures, using historic data of past failures as well as static and dynamic information about the current devices. Classification in the binary sense ("will fail" vs. "will not fail") is not sufficient because, after enough time, every device will eventually fail. Instead, a ranking of the propensity of failure with respect to all other devices is more appropriate. The application uses both the MartiRank and SVM algorithms in its implementation. We do not discuss the full application further in this paper; see [8] for details.

The dependability of the implementation of this system addresses real-world concerns, rather than just academic interest. Although it may be impossible to accurately predict all power outages (which can be due to weather, human error, *etc.*) there have been cases in which outages might be prevented via timely maintenance or replacement of devices that are likely to fail, such as the 2008 blackout in Miami [1] and the 2005 blackout in Java and Bali.[2] A dependable application in this domain may save money and even lives if it can accurately predict which devices are most likely to fail.

To generalize our findings to other types of machine learning besides ranking algorithms, we also applied our techniques to PAYL, which is an anomaly-based intrusion detection system (IDS) that is currently used in numerous real-world deployments. Systems such as PAYL and other machine learning applications in the security domain must be dependable to prevent everything from annoyances like spam mail and phishing attempts to serious threats like denial-of-service attacks or system break-ins.

Thus, it is clear that the implications of this work are significant. Machine learning applications are widespread in modern society, and to ensure their dependability, we must first address their correctness. Therein lies the challenge.

1.2 Forecast

The rest of this paper is organized as follows. Section 2 provides a brief background, including an overview of the fundamentals of supervised machine learning and ranking algorithms (for non-ML readership), and an introduction to MartiRank and SVM-Light. In Section 3 we describe our methodology, including the analyses we performed and the tools we created in order to facilitate our testing. We then discuss in detail each of our three approaches to testing machine learning applications: Section 4 describes our testing based on a "niche oracle" for limited test cases; Section 5 deals with random testing and larger data sets; and Section 6 covers metamorphic testing. To demonstrate that our approaches apply to unsupervised machine learning applications, in Section 7 we address the results of our testing of PAYL. Next we discuss and compare these three approaches in Section 8, and Section 9 describes related work in this field. Last, in Section 10 we describe some of the limitations of our approaches and possible future work, and conclude in Section 11.

2 Background

One complication in our work arose due to conflicting technical nomenclature: "testing", "regression", "validation", "model" and other relevant terms have very different meanings to machine learning experts than they do to software engineers. Here we employ the terms "testing" and "regression testing" as appropriate for a software engineering audience, but we adopt the machine learning sense of "model", as defined below.

2.1 Machine learning fundamentals

In general, data sets used in machine learning consist of a collection of *examples*, each of which has a number of *attribute* values. The examples can be thought of as rows in a table, each of which represents one item from which to learn, and the attributes are the columns of the table.

[1] http://www.cnn.com/2008/US/02/26/florida.power/index.html

[2] http://www.thejakartapost.com/news/2005/08/19/massive-blackout-hits-java-bali.html

In **supervised** ML, a *label* indicates how the example is categorized. In some cases the labels are binary: a label of 1 is considered a *positive example*, and a 0 represents a *negative example*. In the motivating device failure application described above, though, the labels could be any non-negative integer, indicating how may times the device failed over a given period of time (devices may fail, be repaired, and then fail again). Figure 1 shows a small portion of a data set that could be used by such applications. The rows represent examples from which to learn, as comma-separated attribute values; the last number in each row is the label.

Supervised ML applications execute in two phases. The first phase (called the *training phase*) analyzes a set of *training data*; the result of this analysis is a *model* that attempts to make generalizations about how the attributes relate to the label. In the second phase (called the *testing phase*), the model is applied to another, previously-unseen data set (the *testing data*) where the labels are unknown. In a classification algorithm, the system attempts to predict the label of each individual example; in a ranking algorithm, the output of this phase is a ranking such that, when the labels become known, it is intended that the highest valued labels are at or near the top of the ranking, with the lowest valued labels at or near the bottom.

```
27,81,88,59,15,16,88,82,41,17,81,98,42, ..., 0
15,70,91,41, 5, 3,65,27,82,64,58,29,19, ..., 0
22,72,11,92,96,24,44,92,55,11,12,44,84, ..., 1
82, 3,51,47,73, 4, 1,99, 1,51,84, 1,41, ..., 0
57,77,33,86,89,77,61,76,96,98,99,21,62, ..., 1
...
```

Fig. 1. Example of part of a data set used by supervised ML ranking algorithms such as MartiRank and SVM

2.2 Supervised ML applications investigated

Our testing involved two supervised machine learning applications, both of which implement ranking algorithms.

2.2.1 MartiRank

MartiRank [8] was developed by researchers at Columbia University's Center for Computational Learning Systems (CCLS) as a ranking implementation of the Martingale Boosting algorithm [9] specifically with the device failure application in mind.

In the training phase, MartiRank executes a number of "rounds". In each round the set of training data is broken into sub-lists; there are N sub-lists in the Nth round, each containing $1/N$th of the total number of positive labels. For each sub-list, MartiRank sorts that segment by each attribute, ascending and descending, and chooses the attribute that gives the best "quality". The quality is assessed using a variant of the Area Under the Curve (AUC) [13] calculation that is adapted to ranking rather than binary classification. The model, then, describes for each round how to split the data set and on which attribute and direction to sort each segment for that round. In the second (testing) phase, MartiRank applies the segmentation and sorting rules from the model to the testing data set to produce the ranking (the final sorted order).

```
1.0000,61,d
0.4000,32,a;1.0000,12,d
0.2500,18,d;0.5555,55,d;1.0000,41,d
```

Fig. 2. Sample MartiRank model

Figure 2 shows a sample model created by MartiRank. In the first "round", shown on the first line, all of the examples are sorted by attribute 61 (indicated by the "61") in descending order (indicated by the "d"). In the second round, shown on the second line, the result of the first round is then segmented. The first segment contains 40% of the examples in the data set (indicated by the "0.4000") and sorts them on attribute 32, ascending. The rest of the data set is sorted on attribute 12, descending. The two segments are then concatenated to reform the data set, which is then segmented and sorted according to the next line of the model, and so on. A typical MartiRank model in the device failure application may have anywhere from four to ten rounds.

2.2.2 SVM-Light

SVM-Light [11] is a C implementation of the Support Vector Machines (SVM) algorithm [10] and was developed at University of Dortmund's Fakultät für Informatik; along with MartiRank, this program has been applied by researchers at CCLS to address the motivating problem described above.

Support Vector Machines attempt to find a hyperplane that separates examples from different classes. In the learning phase, SVM treats each example from the training data as a vector of N dimensions (since it has N attributes), and attempts to segregate the examples with a hyperplane of N-1 dimensions. Figure 3 demonstrates a simple example.[3] Hyperplane H3 does not separate the two classes, but H1 does, with a small margin (average distance of each data point from the hyperplane), and H2 does with the maximum margin.

The type and shape of the hyperplane is determined by the SVM's "kernel": here, we investigate the linear, polynomial, and radial basis kernels. The goal is to find the maximum margin (distance) between the "support vectors", which are the examples that lie closest to the surface of the hyperplane; the resulting hyperplane is the model. As SVM is typically used for binary classification, ranking is done by classifying each individual example (irrespective of the others) from the testing data according to the model, and then recording its distance from the hyperplane. The examples are then ranked according to this distance. SVM-Light [11], which we used in our testing, is an open-source implementation of SVM, and also has a ranking mode.

[3] http://en.wikipedia.org/wiki/Support_vector_machine

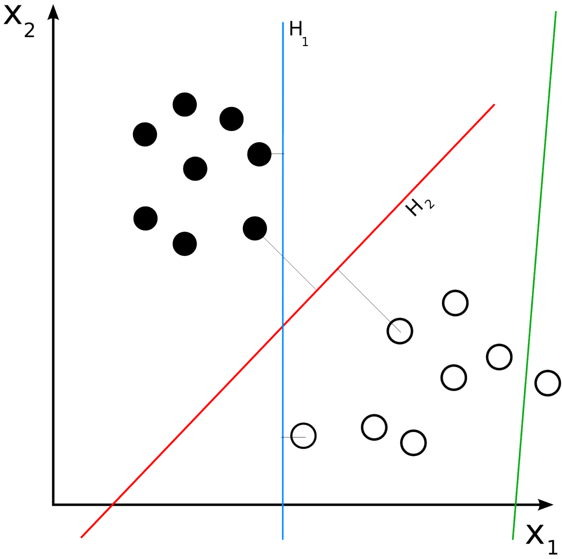

Fig. 3. Data points separated by hyperplanes in SVM

2.3 Potential impact

The impact of our research goes beyond the particular application for which our investigations began. Over fifty different real-world applications, ranging from facial recognition to computational biology, use SVM implementations alone.[4] Additionally, ranking is widely used by Internet search engines (*e.g.*, [14]), also apparently using similarly non-testable algorithms. Thus, ensuring the dependability of these sorts of applications takes on importance even beyond our initial work.

3 Methodology

As is standard in software testing approaches, we begin by performing an analysis of the algorithms and applications being tested, in order to determine equivalence classes and to get an idea for the type of data sets that will be needed. In this section, we describe the particular types of analysis that we perform for the two applications we tested in the domain of supervised machine learning. We then briefly describe our three approaches that are based on this analysis (these are explained in much more detail in Sections 4, 5, and 6, including our findings when applied to MartiRank and SVM-Light), and the tools that we developed to enable these approaches.

3.1 Analyses

All of the approaches described in this paper depend on initial analysis of the algorithms and their respective implementations. While this step is conventional, not novel, a number of interesting issues arise when applied to determining equivalence classes and creating data sets for testing ML applications.

We first analyzed the problem domain (in this case, the prediction of electrical device failures) and the corresponding data sets. We then considered the algorithm as it is defined, and sought out imprecisions or likely places where developers could make errors. Finally, we looked at the implementation's runtime options. We initially outlined these analyses in [15]; here, we provide more detail and describe how they are used to guide our testing approaches.

3.1.1 Analyzing the problem domain

The first step is to consider the problem domain and try to determine equivalence classes based on the properties of real-world data sets. We particularly look for traits that may not have been considered by the algorithm designers, such as data set size, the potential ranges of attribute and label values, *etc.*

The data sets of interest in our particular domain are very large, both in terms of the number of attributes (hundreds) and the number of examples (tens of thousands). In the device failure application, the label could be any non-negative integer, although it is typically a 0 (indicating that there was no device failure) or 1 (indicating that there was), and rarely was higher than 5 (indicating five failures over a given period of time). As would be expected in any large data set, many examples may share the same value for the same attribute, *i.e.*, the attribute value may be repeated, such as the year in which the electrical device was installed; however, in the real-world data, many values are also missing for various reasons, raising the issues of handling unknowns as well as breaking "ties" during sorting.

Though much of the real-world data of interest consists of numerical values - including floating point decimals, dates and integers - some of the data are instead alphanumeric. Some ML ranking algorithms rely on sorting, and while in principle lexicographic sorts could be employed, non-numerical sorts do not seem intuitively appealing as ML predictors; for instance, it may not be meaningful to think of an electrical device manufactured by "Westinghouse" as more or less likely to fail than something made by "General Electric" just because of their alphabetical ordering, and we certainly cannot expect a device manufactured by "Honeywell" as having a propensity to failure being between those two, just because H is between W and G.

To solve this problem, the data sets are expanded by the application during pre-processing to use categorical data. Categorical data refers to attributes in which there are K different distinct (non-numeric) values, but there is no sorting order that would be appropriate for the ranking algorithm. In these cases, a given attribute with K distinct values is expanded to K different attributes, each with two possible values: a 1 if the example has the corresponding attribute value, and a 0 if it does not. That is, amongst the K attributes, each example should have exactly one 1 and K-1 0s. For instance, rather than having an alphanumeric attribute called

[4] http://www.clopinet.com/isabelle/Projects/SVM/applist.html

"manufacturer", multiple attributes would be created, with names "manufacturer_is_Honeywell" or "manufacturer_is_Westinghouse". A given example will have a value of 1 for only one of these attributes, and the rest will be 0s.

By understanding the traits of the data sets in this problem domain, we are able to devise equivalence classes that will be used in all aspects of our testing, as described below.

3.1.2 Analyzing the algorithm as defined

The second element to our analysis was to look at the algorithm as it is defined (in pseudocode, for instance) and inspect it carefully for imprecisions, particularly given what we knew about the real-world data sets as well as plausible "synthetic" data sets. This would allow us to speculate on areas in which flaws might be found, so that we could create test sets to try to reveal those flaws. Here, we are looking for imprecisions in the specification, not so much bugs in the implementation. For instance, the algorithm may not explicitly explain how to handle missing attribute values or labels, negative attribute values or labels, *etc.*

Also, by inspecting the algorithm carefully, one can determine how to construct "predictable" training and testing data sets that should (if the implementation follows the algorithm correctly) yield a "predictable" model or ranking; this is further explored in Section 4. This is peculiar to non-testable programs, since normally all outputs would be predictable from their inputs via the test oracle. This analysis also provides us with insight as to how the application should react when its input data is modified, leading to the creation of the "metamorphic properties" that are used in Section 6.

3.1.3 Analyzing the runtime options

The last part of the analysis is to look at the applications' runtime options and see if those give any indication of how the implementation may actually manipulate the input data, and try to design data sets and tests that might reveal flaws or inconsistencies in that manipulation.

For example, the MartiRank implementation that we analyzed by default randomly permutes the order of the examples in the input data so that it would not be subject to the order in which the data happened to be constructed; it was, however, possible to turn this permutation off with a command-line option. We realized, though, that in the case where none of the attribute values are repeating, the input order should not matter at all because all sorting would necessarily be deterministic. This type of analysis allowed us to create further "metamorphic properties" for use in Section 6.

3.2 Overview of testing approaches

After analyzing the MartiRank and SVM algorithms as described above, we devised the following basic equivalence classes: small vs. large data sets; repeating vs. non-repeating attribute values; missing vs. non-missing attribute values; repeating vs. non-repeating labels; negative labels vs. non-negative-only labels; predictable vs. non-predictable data sets; and combinations thereof. These equivalence classes were then used to guide the generation of appropriate input data sets for the three approaches.

Our first approach uses what we call a "niche oracle". This is a test oracle that only applies to a very small subset of the input domain, for which the expected output can, in fact, be known in advance. In this approach, we hand-craft small "predictable" data sets, such that we know that, if the application is correctly implementing the algorithm, we should get a particular model or ranking. Although these data sets are typically quite trivial, an application must at the very minimum pass these tests before any other testing can proceed; indeed, this type of testing actually did reveal defects in both MartiRank and SVM-Light. This approach is described further in Section 4.

The second approach is called "parameterized random testing". In the absence of sufficient real-world data sets for testing, random testing [4] [5] is an easy way to generate large sets of input. This addresses a limitation of the first approach (in that data sets were necessarily small and often generated by hand), but without an oracle, it is impossible to know what the expected output should be. In this approach, we parameterize the randomness that is used to generate large data sets, based on the different equivalence classes that we intend to test. Although there is no oracle, this approach allows us to still reveal defects and inconsistencies in some cases. We explain this in more detail in Section 5.

The third approach is based on the idea of "metamorphic testing" [6] [7]. Metamorphic testing is designed as a general technique for creating follow-up test cases based on existing ones, particularly those that have not revealed any failure, in order to try to find uncovered flaws. It is a methodology of reusing input test data to create additional test cases whose outputs can be predicted. This allows the application to act as a "pseudo-oracle" [2] for itself, by specifying the behavior that is expected upon changes to the input. Of course, as with the other approaches, this can only show the existence of defects and cannot demonstrate their absence, since the correct output cannot be known in advance (and even if the outputs are as expected, both could be incorrect), but metamorphic testing provides a powerful technique to reveal defects in applications in this domain. Section 6 provides additional explanation.

It is important to note that we consider these approaches in combination, not as alternatives, as each seeks to perform testing in a different manner. The first approach is oracle-based and can be used to demonstrate correctness on the limited set of inputs. The second approach uses no oracle but can exercise different equivalent classes and look for obvious errors. The last approach cannot determine correctness but, by use of a pseudo-oracle, can reveal defects that are not intuitively recognizable. By using different testing approaches, we hope to reveal different types of defects, and thus provide a more powerful technique for improving the dependability of ML applications.

3.3 Tools

To automate some parts of our testing, we devised a framework that includes tools for generating data sets and comparing results.

3.3.1 Data set generation

Although in our particular case real-world data sets were available in abundance, in the general case these data sets may not always be accessible and, even when they are, may not necessarily contain all the equivalence classes that proper testing demands. Hand-generation of data is an option but is only useful for small tests, in particular the types used in the niche oracle approach; however, the applications we tested are used in the real world on extremely large data sets.

In order to facilitate our testing, we developed a tool that allowed us to create a suite of tests that separately addressed different equivalence classes, or combinations of classes. The tool allowed us to specify parameters such as the number of examples, the number of attributes, and the names of the output test data set files, and then used a pseudo-random number generator to create data sets in the appropriate file format. The motivation for this tool is expanded upon in Section 5, in which we further describe our parameterized random testing approach.

The data generation tool can be run with a flag that ensures that no attribute values are repeated within the data set. This option was motivated by the need to run simple tests in which all values are different, so that sorting would necessarily be deterministic (no "ties"). It works as follows: for *M* attributes and *N* examples, generate a list of integers from 1 to *M*N* and then randomly permute them. The numbers are then placed into the data set. If the flag is not used, then each value in the data set is simply a random integer between 1 and *M*N*; the tool also ensures at least one set of repeating numbers. The tool currently only generates positive integers.

The utility is also given the percentage of "positive examples" to include in the data set; positive examples have a label of 1, and negative examples have a label of 0. Similarly, a parameter specifies the percentage of missing values. Our data generation framework has been designed to guarantee that the number of positive examples and the number of missing values come out to be the right number, even though the values are randomly placed (or omitted) throughout the data set.

Parameters can be provided for generating categorical data (with *K* distinct values expanded to *K* attributes as described above). For creating categorical data, the input parameter to the data generation utility is of the format (a_1, a_2, ..., $a_{K\text{-}1}$, a_K, b), where a_1 through a_K represent the percentage distribution of those values for the categorical attribute, and b is the percentage of unknown values. The utility also allows for having multiple categorical attributes, or for having none at all.

Figure 4 shows two data sets created by the data generation tool, using the same parameters (10 examples, 10 attributes, 40% positive examples, 20% missing, repeats allowed); question marks indicate missing values, and the last number in each row is the label. Although the parameters for generating the data are the same, randomness comes into play in the placement of missing values, deciding which examples have labels of 1, *etc.*

```
27,81,88,59, ?,16,88, ?,41, ?,0
15,70,91,41, ?, 3, ? ,?, ?,64,0
82, ?,51,47, ?, 4, 1,99, ?,51,0
22,72,11, ?,96,24,44,92, ?,11,1
57,77, ?,86,89,77,61,76,96,98,1
76,11, 4,51,43, ?,79,21,28, ?,0
 6,33, ?, ?,52,63,94,75, 8,26,0
77,36,91, ?,47, 3,85,71,35,45,1
 ?,17,15, 2,90,70, ?, 7,41,42,0
 8,58,42,41,74,87,68,68, 1,15,1

35, 3,20,41,91, ?,32,11,43, ?,1
19,50,11,57,36,94, ?,96, 7,23,1
24,36,36,79,78,33,34, ?,32, ?,0
 ?,15, ?,19,65,80,17,78,43, ?,0
40,31,89,50,83,55,25, ?, ?,45,1
52, ?, ?, ?, ?,39,79,82,94, ?,0
86,45, ?, ?,74,68,13,66,42,56,0
 ?,53,91,23,11, ?,47,61,79, 8,0
77,11,34,44,92, ?,63,62,51,51,1
21, 1,70,14,16,40,63,94,69,83,0
```

Fig. 4. Two data sets generated with same parameters

The data generation tool also includes a function to permute the order of the examples in an existing data set; this was used in the metamorphic testing approach.

3.3.2 Comparing models and rankings

For situations in which we wanted to compare two outputs, for instance in the metamorphic testing approach (Section 6), we developed a tool that would tell us if two models were the same, and would also compare the final rankings of the testing data.

A simple tool like "diff" could be used to look for differences in the models, since the models are just text files, but for MartiRank we created a utility that compares the models and reports on the differences in each round: specifically, where the segment boundaries are drawn, the attribute chosen to sort on, and the direction. Typically, however, any difference between models in an earlier round would necessarily affect the rest of the models, so only the first difference is of much practical importance. For SVM, the model file contains (among other things not relevant to our testing) the coordinates of the support vectors, and we created a tool that would report any differences between them.

Note that the metamorphic testing approach does not necessarily call for the outputs (models or ranking) to be exactly the same. In some cases, changes to the output are expected. At this point, however, the comparison tool only checks for equality, but we are currently devising a technique for specifying at a high level what the expected changes would be, so that they may be checked automatically.

3.3.3 *Tracing options*

The final part of the testing framework is a tool for examining the differences in the trace outputs produced by different test runs. Because we had access to the source code of both MartiRank and SVM-Light, we were able to add runtime options to report significant intermittent values that arise during the algorithm's execution. In MartiRank, for example, this included the ordering of the examples before and after attempting to sort each attribute for a given segment, and the AUC calculated upon doing so; this would then allow us to see how the examples are being sorted (there may be bugs in the sorting code), what AUC values are determined (there may be bugs in the calculations), and which attribute the code is choosing as best for each segment/round (there may be bugs in the comparisons). This is extremely useful in debugging differences in the creation of models and rankings in the cases when they are not as expected (as in the "niche oracle" approach) or where two outputs are expected to be the same, as in the metamorphic testing approach.

4 Niche Oracle-Based Testing

It is impossible to know whether the output of these applications is correct for arbitrary input, because there is no general test oracle to cover all cases. However, in some trivial cases, it is possible to know what the correct output should be, based on analysis of the algorithm and understanding how it should perform under certain conditions. In this section, we define a "*niche oracle*" as one that will indicate correctness or incorrectness in an otherwise non-testable program, but only for a limited set of test cases. This is the only area of our work in which we can say that there is a correct output that should be produced by the ML algorithm.

As a simple example purely for demonstrative purposes, consider a classification algorithm in which all the examples in the training data have a label of 1. Any example in the testing data, regardless of its attribute values, should also be classified as having a label of 1, because there is no reason for the algorithm to think otherwise; all it knows is that everything has a label of 1. Thus, in this case, it is possible to predict what the correct output should be.

For SVM, we know that in the training phase it seeks to separate the examples into categories. In the simplest case, we could have labels of only 1s and 0s, and then construct a data set such that, for example, every example with a given attribute equal to a specific value has a label of 1, and every example with that attribute equal to any other value has a label of 0. Another approach would be to have a set or a region of attribute values mapped to a label of 1, for instance "anything with the attribute set to 5, 6 or 7" or "anything with the attribute between 3 and 8" or "anything with the attribute above 6". Depending on the kernel that is used, if the testing data then exhibits similar properties, it should be possible to predict which examples will be ranked at the top, and which will be at the bottom.

Creating predictable data sets for MartiRank is a bit more complicated because of the sorting and segmentation. We created each predictable data set by setting values in such a way that the algorithm should choose a specific attribute on which to sort for each segment for each round, and then divided the distribution of labels such that the data set will be segmented as we would expect; this should generate a model that, when applied to another data set showing the same characteristics, would yield the expected ranking.

Figure 5 shows an example of a data set for which it should be possible to predict the "correct" model. In each round of execution, MartiRank seeks to find an attribute to sort, either ascending or descending, so that most of the positive examples (examples with a label of 1; the label is the last value in each row) are towards the top, *i.e.*, it has the highest AUC value. In this case, it should choose to sort the first attribute in ascending order; that gives a better result than sorting it in descending order, or sorting either of the other two attributes in either order.

```
0,3,1,1
1,5,4,1
5,4,5,0
7,6,6,0
2,7,7,1
3,0,9,1
4,2,3,0
6,1,2,1
8,8,0,1
9,9,8,0
```

Fig. 5. Data set that should yield a predictable model in MartiRank, based on the relationship between the attributes and the labels

Although these test cases may seem trivial, they have merit in determining the dependability of the application, because they provide a baseline for minimum requirements that must be passed before any other, more rigorous testing can proceed.

4.1 Findings

This section describes the testing we performed on MartiRank and SVM-Light, in which we manually created simple data sets for which the correct output could be predicted. The creation of these data sets was guided by the analysis we performed (as described above in Section 3) and the ensuing equivalence classes and test cases.

4.1.1 Testing of MartiRank

By inspecting the MartiRank algorithm and considering any potential vagueness, we developed test cases that showed that different interpretations could lead to different results. Specifically, because MartiRank is based on sorting, we questioned what would happen in the case of repeating values; in particular, we were interested to see whether "stable" sorting was used, so that the original order of elements with the same value would be maintained. Although we did have access to the source code and to the MartiRank developers, the sorting routine used a third-party library that had been developed elsewhere.

We constructed data sets such that, if a stable sort were used, a predictable ranking would be achieved because examples in the testing data with the same value for a particular attribute would be left in their original order; however, if the sort were not stable, then the ranking would not necessarily be predictable because the examples could be out of order. Our testing showed that the sorting routine was not, in fact, stable. Though this was not specified in the algorithm, the developers agreed that it would be preferable to have a stable sort for deterministic results - so they substituted another, "stable" sorting routine.

In another simple test, we wanted to see what would happen if sorting on two different attributes gave the same AUC. For instance, if sorting on attribute 3 ascending would give the same AUC as sorting on attribute 10 descending, and either provided the best AUC for this segment, which would the code pick? Our assumption was that the implementation should choose an attribute/direction for sorting only when it produces a better AUC than the best so far, starting with attribute 0 (leftmost in the data file) and going up to attribute N-1 (rightmost), as specified in MartiRank. We thus created test data input such that two attributes, when sorted, gave the same AUC, and we expected in this case that the model would include the leftmost one.

In some of these test cases, the MartiRank implementation acted as expected, but in others it did not, particularly when the two attributes that should provide the same AUC upon sorting were to be chosen after the first round. We designed a data set such that we expected MartiRank to choose one particular attribute in the first round, and then in the second round, for one of the segments, there would be two attributes that yielded the same AUC upon sorting. We expected the model to contain the leftmost attribute, but in some cases it did not.

This led to the interesting discovery that the MartiRank implementation was doing the segmentation (sublist splits) differently from our expectations. By using the framework's model analysis tool, we found that in the second round it was not choosing the attribute we thought it would because at the end of the first round the percentage of the data set in each segment was not as we expected.

It appeared (and we confirmed using the tracing analysis tool) that the difference was that the implementation was taking enough failure examples (labeled as 1s) to fill the segment with the appropriate number, and then taking all non-failure examples (0s) up to the next failure (1). In contrast, the algorithm's designers (who were not the same people who developed the implementation) told us that it would only take enough failures to fill the segment and stop there.

For example, Figure 6(a) shows a sequence of labels that would appear after the first round of sorting. In the second round, two segments would be created, each having 1/2 of the failures. The algorithm designers intended the segmentation to appear as in Figure 6(b), in which the pipe represents the break between the two segments; however, the implementation segmented as in Figure 6(c).

```
(a) 1 1 0 0 1 0 0 1 0 0
(b) 1 1|0 0 1 0 0 1 0 0
(c) 1 1 0 0|1 0 0 1 0 0
```

Fig. 6. (a) A set of labels to be split such that 1/2 of the 1s are in each segment. (b) The expected segmentation, indicated by the pipe. (c) The actual segmentation.

Both are "correct" because the algorithm merely says that, in the *N*th round, each segment should contain 1/*N*th of the failures, and here each segment indeed contains two of the four. The algorithm does not specify where to draw the boundaries between the non-failures. This was one instance we found in which the MartiRank algorithm did not address an implementation-specific issue, thus affecting the result.

In another test case, we sought to explore the possibility of having predictable rankings, as opposed to predictable models as described above. We created a data set with some repeating attribute values and applied it to a particular model that sorted based on that attribute, expecting that the examples with the same attribute values would stay in the same relative order. In some cases, though, the final ranking was not as expected.

Using the tracing utility to see how the examples were being ordered during each sorting round, we found that the "stability" of the sorting (by which we mean, items with the same value stay in their original order with respect to one another after the sorting is complete) in the MartiRank implementation was based on the *initial* ordering from the original data set, and not from the sorted ordering at the end of the previous round. That is, when a list that contained repeating values was to be sorted, we expected that MartiRank would leave those examples in their relative order as they stood at the end of the previous round, but the implementation left them in the relative order as they stood in the original data set. The developer informed us that it was designed this way to make it faster, by "remembering" the sort order for each attribute at the very beginning of the execution, and not having to re-sort in each round.

Since this was not explicitly addressed in the MartiRank algorithm, we contacted the original algorith designers, who decided that remembering the order from the previous round was more in the spirit of the algorithm since it would take into account its execution history, rather than just the somewhat-randomness of how the examples were ordered in the original data set. As in the case of the segmentation issue, this is not a defect in the implementation per se, but the output of the application deviates from what would be expected, and could certainly affect the results in a real-world application.

4.1.2 Testing of SVM-Light

We also created small data sets by hand that should yield a predictable ranking in SVM-Light. In one case, for the first attribute, every example that had a value less than X (where X is some integer) had a label of one; everything else had a label of zero. There were two other columns of random noise. All three kernels we tested (linear, polynomial, and radial basis) correctly ranked the examples.

In another test, however, we changed the labels so that they were all different - simply equal to the value of that example's first attribute incremented by 1. We expected this to be predictable because the examples with larger values for that attribute would be farther away from the hyperplane, and thus would be ranked accordingly. The linear and radial basis kernels found the predictable ranking but the polynomial kernel did not. This difference is, after all, the motivation for multiple kernels, but from our perspective it shows that what is predictable for one kernel is not always predictable for another.

4.2 Discussion

Despite being limited to a very small subset of test cases, the "niche oracle" approach was successful in that it helped us discover discrepancies from the stated algorithms and deviations from user expectations, improving the dependability of MartiRank in particular (all the issues were fixed by the developers). By inspecting the algorithms, we could create predictable data sets that should yield expected results and indicate whether the algorithm was implemented correctly for those basic cases. Even though this approach cannot aid in providing a general test oracle for these non-testable programs, demonstrating that the implementations pass these tests at least increases confidence and provides a foundation for further testing of dependability.

We also discovered that tracing of intermediate state can be useful, because even though we may not know what the final output should be, inspection of the algorithm could indicate what to expect from certain intermediate results. In the case of MartiRank, we could inspect the rankings at the end of each round and see how the examples were being sorted; this led us to discover the segmentation issue.

Possibly the most important thing we discovered is that what is "predictable" for one algorithm will not necessarily lead to a predictable ranking in another. For instance, in some (but not all) SVM kernels it is possible to achieve a predictable outcome in cases when the examples with a 1 label have a particular attribute whose value is in the middle of a range. However, this is difficult in MartiRank because it sorts the attributes ascending and descending in linear order, so values in the middle of a range will never be at the very top or the very bottom. [5] The impact of this observation is that predictable data sets may not necessarily be useful across different algorithms, or even amongst all variations of the same algorithm. However, the use of predictable data sets, even for a single application, does help to address issues related to its dependability.

5 Parameterized Random Testing

Although the "predictable" data sets were useful in revealing some inconsistencies in the applications that we tested, these data sets needed to be created by hand, and creating anything more than small, trivial sets proved to be quite laborious. Additionally, we were not able to cover all the different equivalence classes: for instance, none of the tests in the "niche oracle" approach included missing attribute values, because we could not predict how they would be handled by SVM-Light.

Random testing [4] [5] could be an alternative in these cases: in random testing, rather than create test data sets based on equivalence partitions, the data sets are generated randomly, the idea being that it is fast and easy to create numerous and large test input data sets this way, as long as there is a reliable oracle to determine whether the outputs are correct. However, given that machine learning applications fall under the category of "non-testable programs", this approach may not be suitable. In fact, Hamlet even points out that *"Random testing cannot be attempted without an effective oracle. A vast number of test points are required, and they cannot be trivialized to make things easier for a human oracle."*[5] Thus pure random testing may not be completely applicable in these cases.

To address this limitation, we introduce a technique that we call "parameterized random test data generation", which we originally proposed in [17]. In order to obtain data sets that provide the different combinations of equivalence classes, or the desired separation and isolation of equivalence classes, it is necessary to automatically generate random data sets, but parameterized to control the range and characteristics of those random values. This hybrid testing approach couples the benefits

[5] In fact, this very issue actually came up as a hypothetical problem in the device failure application. Some devices installed in the 1970s tended to fail more than those installed in the 1960s or 1980s, and it was observed that the installation date attribute may conceivably not be selected in the model because of the linear sorting [16].

of using randomness with the necessary control over the properties of the testing data, in order to create data sets that can be used in testing the dependability of these types of applications.

5.1 Findings

It is important to note that, in the general case, without a reliable test oracle, it is only possible to test for obvious and egregious errors, such as core dumps and runtime errors. However, it is not possible to detect minor issues or even tell if the output is correct. Although this seems to limit the effectiveness of this particular approach, it does allow us to exercise different equivalence classes to look for these types of defects, and create data sets that could be used in other oracle or pseudo-oracle based approaches. Here, we discuss these types of findings for MartiRank and SVM.

5.1.1 *Testing of MartiRank*

Some of our test cases for MartiRank simply dealt with data set size, regardless of whether the values in the data set were repeating, missing, categorical, *etc.* The MartiRank implementation did not have any difficulty handling large numbers of examples (hundreds of thousands), but for more than 200 or so attributes, the program reproducibly crashed. Analyzing the tracing output and then inspecting the code, we found that some code that was only required for one of the runtime options was still being called even when that flag was turned off - but the internal state was inappropriate for that execution path, and some pointers were being overwritten. When the implementation attempted to calculate the AUC at the end of each round and there was a large number of attributes, these pointer values were referring to garbage, thus causing the segmentation violation. The MartiRank developers refactored the code and the failures disappeared. Although for these large data sets, we could not tell if the output was actually correct (due to the absence of a general test oracle), the fact that the application was crashing was clearly not the intended behavior.

We also considered test data sets that had missing values. We used the data generation framework to create large, randomly-generated (but non-repeating) data sets, this time with the percent of missing values specified as a parameter. We noticed that multiple invocations of MartiRank, even with the same input data, were yielding different results, which should not be the case because we had explicitly disabled the runtime options dealing with randomization and expected the output to be deterministic (if not predictable).

Upon analyzing the tracing outputs with the framework tools, we noticed that the implementation - even with all optimizations turned off - was still performing randomizations in the case of missing values. In particular, when trying to sort a list of numbers that contained some missing values, it kept the missing values in the same relative order but placed them randomly throughout the list. For instance, Figure 7(a) shows a series of numbers, with A, B, and C representing the missing values; Figure 7(b) shows possible results of this technique of sorting the known values and randomly placing the missing ones amongst them.

(a)	4	A	5	6	2	1	0	B	C	3
(b)	0	1	2	A	3	B	4	C	5	6
(b)	0	A	1	2	B	C	3	4	5	6
(b)	A	0	1	2	3	B	4	5	C	6
(c)	0	A	1	2	3	4	5	B	C	6

Fig. 7. (a) A set of data to be sorted, with A, B, and C representing the missing values. (b) Three possible outputs in which the missing values stay in their relative order but are randomly placed in the set. (c) A deterministic output in which the missing values stay in place.

After we brought up this issue with the ML researchers who had devised the algorithm, they decided that the sorting should be "stable" with respect to missing values in that examples with a missing attribute value should remain in the same position, with the other examples (with known values) sorted "around" them, as shown in Figure 7(c) [18]. Other deterministic options for handling this case (such as putting all missing values at the end) were considered, but keeping the examples with missing values consistent with respect to any previous sort order was deemed to be most in the MartiRank spirit. The developers then changed the implementation, to allow for determinism that may be needed to assist later testing.

Because categorical data provides a combination of necessarily repeating (all values are either 0s or 1s, so if there are more than two non-missing, there will necessarily be a repeat) and sometimes missing values, we created data sets with categorical attributes as part of our test data. We used a data set that included categorical data to discover a bug in the implementation whereby the incorrect use of a global variable in the calculation of the AUC led to a reported value that was greater than 1, which is clearly wrong (since the AUC is normalized). More importantly, though, this bug did not surface when testing only with repeating values or only with missing values; it was the data sets that combined these two equivalence classes that allowed the bug to be revealed.

5.1.2 *Testing of SVM-Light*

Whereas MartiRank crashed when given a training data set with a large number of attributes, SVM-Light would crash when given a training data set with a large number of examples. Data sets of about 10,000 examples caused SVM-Light to give an "out of memory" error, even when there were very few attributes and the system had 16GB of free memory. Ironically, SVM-Light had no problem handling the cases that had few examples but many attributes (over 10,000, which is more than two orders of magnitude greater than what caused the analogous failure in MartiRank).

5.2 Discussion

Our contribution here is to have demonstrated how random testing and partition testing can work together, instead of as alternatives, and also to show that this hybrid approach can be useful in applying the principles of random testing to "non-testable programs", thus improving their dependability. We have demonstrated defects in both MartiRank and SVM, as well as further issues related to the interpretation of the MartiRank algorithm. In addition, in this approach we created numerous test data sets that could be used in the other approaches, as well as regression testing, too.

By combining parameterization and randomness, we gained the ability to control the properties of very large data sets, which was critical for limiting the scope of individual tests and for pinpointing specific issues in how the code was handling different equivalence classes. The data generation tool proved to be preferable to alternative approaches we considered, such as culling real-world data, which would be time consuming and more prone to error.

Additionally, the tool could be used for the testing of any ML ranking algorithm, not just MartiRank or SVM; it could also be used for supervised ML classification algorithms. The data generator supports plug-replaceable modules for creating data set files in whatever format is needed. Two such modules are currently implemented, one for MartiRank implementations (csv files) and the other for SVM-Light (a "sparse" attribute-value pair representation for data sets with a high ratio of missing values).

6 Metamorphic Testing

Despite the usefulness of the first two approaches in detecting defects and inconsistencies in the implementations we tested, they are both somewhat limited by design. Although the "niche oracle" approach is able to demonstrate correctness to a (very) limited degree, the creation of data sets is time-consuming and only a small number of test cases can be used, and generally only for a specific application. The parameterized random testing approach eases the creation of the data sets and of covering different equivalence classes, but without a general test oracle, only obvious errors can be detected.

One approach to investigating the dependability of such "non-testable programs" has been to use a pseudo-oracle [2], in which multiple implementations of an algorithm process an input and the results are compared, as in N-Version Programming [19]; if the results are not the same, then at least one of the implementations contains a defect. This is not always feasible, though, since multiple implementations may not exist, or they may have been created by the same developers, or by groups of developers who are prone to making the same types of mistakes [20]. In the absence of multiple implementations [6], however, metamorphic testing [6] can be used to produce a similar effect. The approach we describe in this section is based on metamorphic testing, applied to this particular domain.

6.1 Background

Metamorphic testing is a methodology of reusing input test data to create additional test cases whose outputs can be predicted. The definition of metamorphic testing is as follows: if input *x* produces an output *f(x)*, the function's so-called "metamorphic properties" can then be used to guide the creation of a transformation function *t*, which can then be applied to the input to produce *t(x)*; this transformation then allows us to predict the output *f(t(x))*, based on the (already known) value of *f(x)*. If the output is not as expected, then a defect must exist. We demonstrate that, in particular, this approach can be applied with success to the domain of machine learning applications.

A simple example (outside the domain of ML applications) of a function to which metamorphic testing could be applied would be one that calculates the standard deviation of a set of numbers. Certain transformations of the set would be expected to produce the same result. For instance, permuting the order of the elements should not affect the calculation; nor would multiplying each value by -1, since the devation from the mean would still be the same (think about the values being "flipped" around the origin on the number line).

Furthermore, we know that there are other transformations that will alter the output, but in a predictable way. For instance, if each value in the set is multipled by 2, then the standard deviation should be twice as much as that of the original set, since the values on the number line are just "stretched out" and their deviation from the mean becomes twice as great. Thus, given one set of numbers, we can create three more sets (one with the elements permuted, one with each multiplied by -1, and another with each multiplied by 2), and get a total of four test cases; moreover, given the output of only the first test case, we can predict what the other three should be.

Metamorphic testing generally would not be needed for this trivial example, but clearly can be very useful in the absence of an oracle: regardless of the values in the data set, and even if the correct output of an application or function could not be known in advance, if the different outputs for the different inputs are not as expected, then there must be a defect in the implementation. Although the use of these simple identities for testing numerical functions is not unique to metamorphic testing (*e.g.*, testing based on algebraic properties [21]), the approach can be used on a broader domain

[6] There are, in fact, numerous implementations of the SVM algorithm but we sought a generally-applicable approach that does not rely on having such pseudo-oracles.

of any functions that display metamorphic properties, including machine learning applications. Additionally, metamorphic testing can treat the application under test as a black box, and does not require access to source code.

6.2 Findings

In this approach, we enumerated and categorized the different types of metamorphic properties, and then used these principles in our testing. We used the analysis of both the MartiRank and SVM algorithms (from Section 3) to determine these properties, and then tested the implementations using the metamorphic properties as our guidelines. Note that this approach does not require access to the source code of the application under test, but rather only requires an understanding of the algorithm. Since it does not require the source, it could in principle be used by an organization that contracted out development of a system, to gain confidence in dependability during acceptance testing prior to releasing a product into production. We first outlined this approach in [22]; here, we present additional findings and detail.

6.2.1 Metamorphic properties

We begin by describing our observations of the metamorphic properties of both MartiRank and SVM. We first considered metamorphic relationships that should not affect the output: either the model that is created as a result of the training phase, or the ranking that is produced at the end of the testing phase. For the training phase, if training data set input D produces model M, then we looked for transformation functions T_d on the data sets, so that input $T_d(D)$ would also produce model M. Additionally, if testing data set input K and model L produce ranking $r(K, L) = R$, then we looked for transformation functions T_d on the data sets and transformation functions T_m on the model so that the combinations $r(T_d(K), L)$, $r(K, T_m(L))$ and $r(T_d(K), T_m(L))$ all produce R as well. That is, we identify transformations of the input data, the model, or both so that the ranking is unchanged.

For both MartiRank and SVM - as well as, based on our knowledge of the domain, most other ranking and classification algorithms - it should certainly be the case that changing the order of the examples should not affect the model (in the first phase) or the ranking (in the second); we describe this as having a **permutative** metamorphic property. For MartiRank, though, this is only true under certain conditions. As MartiRank is based on sorting of attributes, in the cases where all the values for a given attribute are distinct, it is clear that the sorted order should still be the same regardless of the original input order. However, since MartiRank uses stable sorting with respect to repeating and missing attribute values, and also using the sorted order from the previous "round" rather than the original (as a result of our findings in Sections 4 and 5), permuting the order may change the result.

Additionally, based on our analysis of the algorithms, we noticed that it is not the actual values of the attributes that are important, but it is the *relative* values that are used. In MartiRank, adding a constant value to every attribute, or multiplying each attribute by a positive constant value, should not affect the model because the model only concerns how the examples relate to each other (based on sorting), and not the particular values of the examples' attributes. The model declares which attributes to sort to get the best ordering of the labels: if the values in any column were all increased by a constant, or multiplied by a positive constant, then the sorted order of the examples would still be the same, so the same attribute would be chosen as the best to sort on, thus the model would not change. Additionally, applying a given model to two data sets, one of which has been created based on the other but with each attribute value increased by a constant, would generate the same ranking, based on the same line of reasoning. Thus, MartiRank exhibits metamorphic properties that we can classify as both **additive** and **multiplicative**: modifying the input data by addition or multiplication by a positive constant should not affect the output.

SVM displays these properties, too, though only in the ranking phase, and not in the generation of the model. If a training data set were transformed using an additive or multiplicative transformation, then the corresponding model (hyperplane) would be affected by being shifted or expanded in the N dimensions; however, if the testing data set also had the same transformation(s) applied, the resulting ranking of the new model applied to the new data set would be the same as the original model applied to the original data set, because each example (or point in N dimensions) would similarly be moved, and the relative distances from the hyperplane would stay the same.

We then considered metamorphic relationships that would affect the output, but in a predictable way. For the training phase, if training data set input D produces model M, then we looked for transformation functions T_d on the data set so that input $T_d(D)$ would produce model M', where M' could be predicted based on M. Additionally, if testing data set input K and model L produce ranking $r(K, L) = R$, then we looked for transformation functions T_d on the data set and transformation functions T_m on the model, so that $r(T_d(K), L)$, $r(K, T_m(L))$ and $r(T_d(K), T_m(L))$ all can be predicted based on R. That is, we identify transformations of the testing data, the model, or both so that the ranking can be predicted, but is not necessarily exactly the same as the original. Keep in mind that in order to perform testing, we need to be able to have a predictable output based on R because we cannot know it in advance otherwise, since there is no test oracle, aside from the very limited set of niche cases.

We mentioned above that multiplying all attributes by a positive constant should not affect the model in MartiRank. On the other hand, multiplying by a negative constant clearly would have an effect, because sorting would now result in the *opposite* ordering. The effect on the MartiRank model, however, could easily be predicted, because the model not only specifies which attribute to sort on, but which direction (ascending or descending) as well. Consider that, if one were to sort a group of numbers in ascending order, then multiply all the values in the original (unsorted) set by a negative constant, and sort in descending order, the resulting order of the examples in the ranking should be the same. In MartiRank, if in the original data set a particular attribute is deemed to be the best one to sort on, and a new data set is created by multiplying every attribute value by a negative constant, then that particular attribute will still be the best one to sort on, but in the opposite direction. The only change to the model will be the sorting direction. Thus, MartiRank displays an **invertive** metamorphic property, wherein it is possible to predict the output based on taking the "opposite" of the input. We mention here again that this property only holds in the case where all values are distinct.

This invertive property can also be seen in the testing phase. For data set input *K*, we define *K′* as its inverse, *i.e.*, all attribute values multiplied by a negative constant. For model *L*, we define *L′* as its inverse, *i.e.* the sorting directions all changed. We also define *R* = *r(K, L)* as the ranking produced on data set *K* and model *L*, and *R′* as the inverse ranking, where the examples are ranked in "backwards" order. Based on the explanation above, we can expect that if *r(K, L)* = *R*, then *r(K′, L′)* is also equal to *R*, because sorting the positive values ascending will yield the same ordering as sorting the negative values descending. It follows, then, that *r(K′, L)* and *r(K, L′)* should both be equal to *R′*, in which the ranking is the same but in the opposite direction.

SVM also demonstrates this invertive property, but only if it is applied to the entire data set, as opposed to an individual attribute. In the training phase, for example, if each value in the training data set is multipled by -1, then one can think of the model being "flipped" or "rotated" about the origin in N-dimensional space.

```
1.0000,61,d
0.4000,32,a;1.0000,12,d
0.2500,18,d;0.5555,55,d;1.0000,41,d
```

Fig. 8. Sample MartiRank model

Furthermore, once we know the model, it is easy to add an example to the set of testing data so that we can predict its final place in the ranking. Take, for example, the MartiRank model shown in Figure 8. In the first round, it sorts on attribute 61 in descending order; if we add an example to a testing data set such that the example has the greatest value in attribute 61, it will end up at the top of the sorted list. In the second round, the model sorts the top 40% (which would include our added example) on attribute 32 in ascending order; if we modify our added example so that it has the smallest value for attribute 32, it will stay at the top of the list. And so on. Knowing the model, we can thus construct an example, add it to the data set, and expect it to appear first in the ranking. We can thus say that MartiRank has an **inclusive** metamorphic property, meaning that a new element can be included in the input and the effect on the output is predictable. Similarly, MartiRank also shows an **exclusive** metamorphic property: if an example is excluded from the testing data, the resulting ranking should stay the same, but without that particular example, of course.

Because in its ranking mode, SVM considers each example in the testing data independently and ranks according to the distance from the hyperplane, SVM also demonstrates the exclusive property: if an example is removed, it would not affect the final ranking. Similarly, SVM demonstrates the inclusive property, though in a simpler form than MartiRank. In the ranking phase, regardless of the model, by looking at the numerical values in the testing data one can construct a new example with attribute values that are significantly greater than the others; thus, that example is going to be very far away from the hyperplane, and will be ranked highest.

6.2.2 Testing of MartiRank

After identifying the metamorphic properties of MartiRank, we used the equivalence classes from the analyses described in Section 3 and the test cases from the testing described in Section 5 to create data sets for use in this approach (in some cases we also used real-world data sets). We then applied the metamorphic transformations, and were able to detect a defect in the implementation that was not detected earlier using the other approaches. Another of MartiRank's invertive properties is that if all of the labels (as opposed to the attributes) in the training data are multiplied by -1, the final ranking of the testing data should be the same but in opposite order from the original, since what was the "worst" would now be considered "best". However, because the particular implementation we were testing was designed specifically to rank the likelihood of device failures, the labels in the training data (which represented the number of failures over a given period of time) would never be negative in practice, so this was not considered during development. During metamorphic testing, the implementation produced inconsistent results when a negative label existed, and we confirmed this bug upon inspection of the code, in which a logical flaw existed in the way the examples were being segmented during training. In principle a general-purpose ranking application should allow for negative labels (-1 vs. +1 is sometimes used in other applications).

6.2.3 Testing of SVM-Light

With respect to the SVM implementation we tested, the most relevant finding is that randomly permuting the order of the examples in the training data caused it to generate different results. The practical implication is that the order in which the data happens to be assembled can have an effect on the final outcome. The SVM algorithm theoretically should produce the same result regardless of the input data order; however, an ML researcher familiar with SVM-Light told us that because it is inefficient to run the quadratic optimization algorithm on the full data set all at once, the implementation performs “chunking” whereby the optimization algorithm runs on subsets of the data and then merges the results [18]. Numerical methods and heuristics are used to quickly converge toward the optimum; however, the optimum is not necessarily achieved, but instead this process stops after some threshold of improvement. Here the implementation deviates from the expected behavior.

6.3 Discussion

We have identified six metamorphic properties that we believe exist in many machine learning applications: additive, multiplicative, permutative, invertive, inclusive, and exclusive. Although these are likely not the *only* metamorphic properties that can exist in a machine learning algorithm, they provide a foundation for determining the relationships and transformations that can be used for conducting metamorphic testing.

Most importantly, we have demonstrated that this approach can reveal defects in the applications of interest, by allowing for the creation of built-in pseudo-oracles, even without access to the source code. As we have demonstrated in [23], these properties can also be applied to classification algorithms as well, and in fact we used these properties and this approach to discover defects in popular open-source ML libraries. When used in conjunction with the other two approaches described previously, this provides a powerful mechanism for evaluating the dependability of applications in this domain.

7 Applying to Unsupervised Learning

In addition to demonstrating the feasibility of our approach in the domain of supervised machine learning applications (particularly ranking algorithms), we also wanted to show its effectiveness in unsupervised machine learning as well.

7.1 Unsupervised ML and PAYL

Like supervised ML, **unsupervised** ML applications also execute in training and testing phases, but in these cases, the training data sets necessarily do not have labels (as opposed to supervised ML, in which the labels in the training data are known). Rather than attempt to find a relationship between attribute values and labels, as in supervised ML, an unsupervised ML application seeks to learn properties of the examples on its own, such as the numerical distribution of attribute values or how the attributes relate to each other. This model is then applied to testing data, to determine to what extent the same properties hold. Data mining [24] and collaborative filtering [25] are two well-known examples of unsupervised learning.

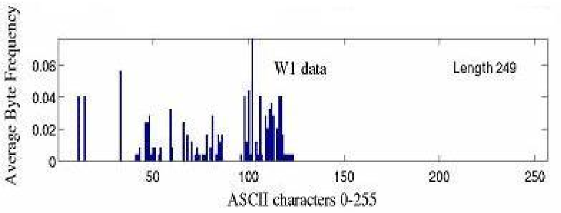

Fig. 9. Sample payload byte distribution

To apply our approach to an unsupervised machine learning application, we investigated PAYL [12], an anomaly-based network intrusion detection system (IDS) that was developed by members of Columbia University’s Intrusion Detection Systems Lab for purposes unrelated to the device failure application. Many such systems are primarily signature-based detectors, and while these are effective at detecting known intrusion attempts and exploits, they fail to recognize new attacks and some variants of old exploits. However, anomaly-based systems like PAYL are used to model normal or expected behavior in a system, and detect deviations of interest that may indicate a security breach or an attempted attack.

PAYL’s training data simply consists of a set of TCP/IP network packets (streams of bytes), without any associated labels or classification. During its training phase, it computes the mean and variance of the byte value distribution for each payload length (the payload can be thought of as the “message” inside the network packet) in order to produce a model; Figure 9 shows an example of such a distribution [12]. During the second (“detection”) phase, each incoming packet is scanned and the distribution of the byte values in its payload is computed. This new payload distribution is then compared against the model (for that payload length) using the Mahalanobis distance [26], which is a way of comparing two sets of data taking into account their correlations; if the distribution of the bytes in the new payload is above some configurable threshold of difference from the norm, PAYL flags it as anomalous and generates an alert.

Although PAYL can act as a standalone application, it has been incorporated within a commercial product that has been deployed in a number of corporate network environments [27]. Intrusion detection systems are clearly becoming more and more common as mission-critical data is stored online and attackers seek to access it or gain control of systems, so ensuring the dependability of such IDS applications has taken on immense importance.

7.2 Analysis of PAYL

Whereas we did have access to the MartiRank and SVM-Light source code, we did not have such a luxury for PAYL, and needed to treat it as a black box. However, based on our understanding of the algorithm from the literature and from discussing it with the developers, we were able to analyze the algorithm and determine its equivalence classes.

Unlike the data sets used for the ranking algorithms, particularly in the device failure application, the data sets used by PAYL had much less flexibility, which somewhat limited the number of test cases we could create by analyzing the problem domain. In the ranking algorithms, the data sets consisted of rows and columns of numbers, but in this domain, each data element needed to conform to the TCP/IP standard, otherwise it would be rejected as bad data by the underlying TCP/IP analysis tool (jpcap [28]) and would not be useful in testing. For example, if the value in the “packet length” field did not match the actual packet length, or if any part of the packet were missing, the packet would be ignored.

Also, in the data sets for the ranking algorithms, the number of attributes was configurable in the test data generation tool; in this case, though, there are only three attributes of importance: the length of the payload, the intended port, and the data inside the payload. Thus, the tool to generate random data would need to take only four parameters: the number of examples (packets); and whether or not to allow for repeated payload lengths, port numbers, and payload data values.

Our analysis of the algorithm as defined mostly focused on the types of alerts it could generate. As mentioned above, PAYL would raise an alert if the byte distribution of the payload of an incoming packet were anomalous according to the model (we refer to these as “anomalous-distribution alerts”). Additionally, PAYL may also raise an alert in other circumstances, for instance if the payload length had never been seen before in the training data (“length-never-before-seen alerts”), or if the packet was intended for a port that had never previously received any traffic (“port-never-before-seen alerts”). By understanding the various types of alerts, we could then create equivalence classes that sought to produce these different types of outputs.

While PAYL can perform “online learning”, in which the model is constructed and updated piecemeal in the training phase as new packets arrive on the network interface, as opposed to reading them all at once, it also has a batch mode in which it can read all the packets from a text file, created by a tool such as tcpdump [29]. This was possible for the detection phase, too, in which all the packets in the file would be flagged as anomalous if deemed as such. Thus, we did not need to simulate real input data by sending actual packets over the network, but rather we created data sets in plain text.

After our analysis of PAYL, we devised equivalence classes including the following: all payload lengths the same vs. distinct payload lengths vs. some repeating payload lengths; all port numbers the same vs. distinct port numbers vs. some repeating payload lengths; all payload values the same vs. distinct payload values vs. some repeating payload values; small vs. large data sets; and combinations thereof. These equivalence classes guided the testing we performed on PAYL, using the three approaches described previously, enabling us to detect defects and increase its dependability.

7.3 Niche oracle-based testing of PAYL

Using the results of our analysis, we devised simple test cases for which we could know the expected result, *i.e.*, that given a set of training data and/or a model, we would be able to know whether a particular incoming packet should be flagged as anomalous or not, and in particular which type of alert it should raise.

In one of the test cases in which the incoming packet was expected to be anomalous, the training data consisted of payloads where all the bytes were set to 0x00, and the payload in the packet in the testing data had all bytes set to 0xFF. Because this value is as “far away” as possible from what should be considered “normal”, the incoming packet should be flagged as anomalous. In test cases where we would *not* expect the incoming packet to be labeled as anomalous, the values in its payload would also be set to 0x00, so that it should exactly match what was in the training data and should definitely not be seen as different.

Another type of alert that PAYL could raise would be one in which the payload length in the incoming packet had not been seen in the training data. In a simple test, we would expect PAYL to raise this alert by creating training data in which all the payloads were of length X, and the payload in the testing data would be of length X+1; in the test in which we did not expect this type of alert, the testing data payload length would also be X.

We did not discover any defects in the PAYL implementation using this testing approach, but the fact that we were able to develop simple test cases shows that the “niche oracle” approach can apply to unsupervised machine learning applications as well as supervised.

7.4 Parameterized random testing of PAYL

Our testing using the parameterized random testing approach did reveal some unexpected behavior in PAYL. In one test case, PAYL raised *both* an anomalous-distribution alert *and* a length-never-seen-before alert for a payloads of a length 1448 bytes (the actual value is not important, but is used here for illustrative purposes), which theoretically should never happen, since the byte distribution can only be considered anomalous if a payload of that length had actually been seen before in the training data. Upon further investigation, we determined that PAYL actually should only have raised the length-never-seen-before alert, since there were no payloads of that length in the training data. Note that this only occurred intermittently (but was reproducible), and not for all test cases.

The developers of the software determined that this unexpected outcome was occurring because, depending on the amount of training data, PAYL would sometimes group packets of similar payload length together when determining the byte distribution in the model. That is, the model might be for a range of payload lengths, not just for one single value. In this case, there was no payload of length 1448 in the training data, so a length-never-seen-before alert was the correct response. The anomalous-distribution alert was raised because the packet was compared to a model for lengths "around" 1448 (for instance, in the range 1440-1454), and that particular packet was deemed to be anomalous compared to the normal distribution for payloads in that range. Although this behavior was by design, and is not a bug per se, the application did not behave as expected by raising two alerts that one would think would never appear in conjunction.

7.5 Metamorphic testing of PAYL

Last, we applied the metamorphic testing approach to PAYL. Because the model generated by PAYL in the training phase represents the distribution of byte values in the TCP/IP payload (see Figure 9), it is clear that it exhibits the additive and multiplicative properties, as described in Section 6. Adding a constant value to each byte would shift the distribution, and multiplying by a constant would stretch it. Therefore, it would be easy to predict the effect on the model. Additionally, the categorization (as anomalous or not) of a packet in the testing phase would not change if it, too, had its bytes modified in the same manner.

Much of our analysis of PAYL focused on its permutative properties, primarily because some attackers may try to hide a worm or virus by permuting the order of the bytes, so as to trick a signature-based intrusion detection system. Of course, the model created by PAYL does not consider the order of the bytes, only their distribution, so a permutation should still result in the same model. At a higher level, because the model is created from a number of packets (not just a single one), permuting the order of the packets in the training data stream should also result in the same model.

PAYL also has an invertive property. An "inverse" of the distribution can be obtained by subtracting each byte value from the maximum (0xFF), so that frequently-seen values become less frequent, and vice-versa. If the same treatment is then applied to the payloads in the testing data as well, then the same alerts should be raised as before, since these values will still appear to be anomalous.

Aside from considering the distribution of byte values in creating its model, PAYL also considers the existence (or absence) of payloads of certain lengths, and thus certainly has inclusive metamorphic properties. For instance, consider a model that generates an alert on a new payload because its length had never before been seen. If the particular payload were then included in the training data, it should no longer be considered anomalous. We would similarly expect PAYL to have exclusive metamorphic properties: if all payloads of a certain length were removed from the set of training data, then any messages of that length in the testing data would thus be considered anomalous because they had not previously been seen. The same holds true for port numbers, in addition to payload length.

After analyzing PAYL's metamorphic properties, we conducted testing of PAYL by generating data sets using parameterized random testing, and then modifying them according to these metamorphic relationships. By using the exclusive metamorphic property, we were able to detect another defect in PAYL. We started with training data that had payloads of various sizes, including 274 bytes (the actual value here too is used for illustrative purposes), and created a model that was applied to a set of testing data, which also included a payload of 274 bytes; we could not in advance know whether PAYL should raise an anomalous-distribution alert, but we expected from our "niche oracle" testing that it should not raise a length-never-seen-before alert. In this case, PAYL raised no alerts.

We then removed all payloads of 274 bytes from the training data and applied the new model to the same (unmodified) testing data, expecting that the payload of 274 bytes in the testing data would now cause PAYL to raise a length-never-seen-before alert, since it was not in the training data. However, PAYL only raised an anomalous-distribution alert instead. As we discovered previously, this could happen if the payload length is contained in a model for a range of lengths, but this turned out to be a different situation from the issue found in the parameterized random testing approach. In that case, *both* the anomalous-distribution and length-never-before-seen alerts were raised; in this case, only one alert was raised, and it was of the wrong type. Regardless of whether or not that payload had an anomalous distribution, the length had not been seen before and that alert should have been raised.

Our key result, though, was that we were able to verify that PAYL exhibits the same six metamorphic properties as do the supervised ML algorithms, and then use these properties to drive metamorphic testing and identify a previously-unknown defect in PAYL.

7.6 Discussion

We were very encouraged by the fact that our testing approaches translated very easily to the domain of unsupervised machine learning (or, at least, to this particular application in that domain). Although the data sets are structured differently from those in supervised machine learning, and there are no labels or classifications in the training data, it is still possible to develop equivalence classes based on analysis of the algorithm, and then design simple test cases that should yield predictable results, as well as more complex test cases that take advantage of parameterized random testing and demonstrate whether the implementation maintains its expected metamorphic properties. As PAYL is used in many real-world situations for important business purposes, helping to improve its dependability is indeed a welcome result.

8 Evaluation

Clearly the three approaches are designed to work in conjunction, as we have demonstrated here. However, on their own, the approaches have individual merits and drawbacks.

Although some issues could be detectable by all three approaches, metamorphic testing seemed to have the most potential for revealing more interesting and important defects, primarily because it most closely mimics the pseudo-oracle approach. In particular, the defect we discovered in PAYL most likely would not have been detected using the other approaches; it only came about when using more than one set of inputs and comparing their results. This observation may also be a result of the fact that we developed the metamorphic testing approach last, however, and used our experiences from the other testing approaches when devising the metamorphic properties.

All three approaches require some domain knowledge regarding the application under test, but testing can still be conducted by software test engineers who are not the developers (as in our case, since our background is in software engineering and not machine learning). In the analysis step, it was necessary to understand the problem domain and the corresponding data sets, the algorithm being implemented, and the implementation's runtime options, and our testing of MartiRank did benefit from working closely with the developers and having access to source code. However, this is not necessarily a requirement: although the SVM-Light implementation is open-source, we did not need to inspect the code in order to devise equivalence classes or test cases. In fact, we had very little knowledge of the SVM algorithm upon commencing our testing, nor did we communicate with its developers regarding our testing, and we did not even have access to the PAYL source.

In the metamorphic testing approach, the tester would need to know the algorithm well enough to understand the effects of changes on input, but these could conceivably be specified by the algorithm designer, and not the developer who codes the implementation. It may also be possible to detect these metamorphic properties automatically, in a similar way to how program invariants [30] and algebraic specifications [31] are, though this would require access to the source code. In our own testing of PAYL, we did not have access to the source code, and as described here, the metamorphic testing approach is purely "black-box" and can be conducted by anyone who is familiar with how the application should react upon changes to its input.

Although we did find discrepancies (differences from expectations) that might or might not be considered defects, we did not encounter any false positives in our testing. Since each approach relies heavily on analysis of the algorithm, however, if the tester's analysis is incorrect then defects could be reported erroneously. In the case of metamorphic testing, though, it may be acceptable if the set of specified metamorphic properties is not sound, *i.e.*, not true for *all* inputs. Others have demonstrated that, at the risk of false positives, when using model-based testing approaches, an unsound model (or, in our case, metamorphic properties) may reveal defects that more restrictive sound properties would not [32]. For instance, in MartiRank, we pointed out that permuting the order of the input data should not affect the output, but only assuming that the values in the input are all distinct (because MartiRank now uses stable sorting). However, we can remove this assumption and concede that although this metamorphic property is not sound (because for some inputs, it will not be true), the new output will in general be *approximately* equal to the original, based on some metric of comparing rankings, such as the number of elements ranked differently, the Manhattan distance, or the Euclidean distance in N-dimensional space. At the expense of revealing false positives, this property may also reveal actual defects that may not be detected if we included the original constraint that all values must be distinct.

Of the three approaches, the metamorphic testing approach is the only one suitable for runtime testing in the production environment ("the field"). As we have explored in [23], the metamorphic properties could conceivably be checked as the program is running in the live environment, assuming they are expressed in some executable specification format.

A desirable side effect of our testing has been to create a suite of data sets that can then be used for regression testing purposes, further helping to increase the dependability of the applications. The development of the device failure application was done in conjunction with our testing, and our input data sets together with previously recorded outputs were used successfully to find newly-introduced bugs. For example, after a developer refactored some repeated code and put it into a new subroutine, regression testing showed that the resulting models were different than in the previous version. Inspection of the code revealed that a global variable was incorrectly being overwritten, and after the bug was fixed, regression testing showed that the same results (prior to refactoring) were once again being generated.

Last, all three approaches could be used for other types of machine learning applications: we have demonstrated their feasibility for ranking algorithms and anomaly-based intrusion detection systems, but they could also be used with classification algorithms, as we have explored in [23]. Although new plug-in modules would need to be created to tailor the testing framework to particular input data formats and model formats, the approaches themselves would still be able to reveal defects, even without test oracles.

9 Related work

9.1 Testing ML applications

Although there has been much work that applies machine learning techniques to software engineering in general and software testing in particular (*e.g.*, [33], [34], [35], *etc*.), we are not currently aware of any work in the reverse sense: applying software testing techniques to machine learning applications, particularly those that have no reliable test oracle.

Orange [36] and WEKA [37] are two of several frameworks that aid ML developers, but the testing functionality they provide is focused on comparing the quality of the results in terms of how well the application can learn or predict, and not evaluating the "correctness" of the implementations. An ML application may appear to be predicting well but still have defects, of course. Repositories of "reusable" data sets have been collected (e.g., the UCI Machine Learning Repository [38]) for the purpose of comparing result quality, but not for the software engineering sense of testing and for ensuring dependability.

Testing of intrusion detection systems [39] [40], intrusion tolerant systems [41], and other security systems [42] has typically addressed quantitative measurements like overhead, false alarm rates, or ability to detect zero-day attacks, but does not seek to ensure that the implementation is free of defects, as we do here. An IDS with very few or no false alarms could still have bugs that prevent it from detecting many (or any) actual intrusions, making it completely undependable.

9.2 Random testing

There has been past research into the generation of test data sets [43] [44]; however, whereas much of the early work in random test data generation [45] [46] started in the area of compilers, we are looking at a way of creating parameterized random test data specifically for ML algorithms. Of course, even purely random test data is somehow "parameterized", but this generally refers to specifying the data type or range of acceptable values. The term "parameterized random testing" appears in circuit design literature [47] but refers to parameterizing the distribution of input values, and does not address parameterizing according to equivalence classes or partitions, which we present here.

Wichmann [48] proposes something similar to our random testing-based approach in his recommendations to the British Computer Society Specialist Group in Software Testing. He notes the role that randomization can have even within the "limitations" of partition testing when it comes to randomly selecting testing data for a given equivalence class. However, his work in this area has only focused on software components, whereas we are investigating approaches to system-level testing. More importantly, our work addresses the particular issues that arise in the domain of non-testable programs.

Our work in random testing is also similar to that of Thévenod-Fosse *et al.* [49], who labeled their approach "structural statistical testing" in that random input are selected according to given criteria, particularly related to path selection. Our approach differs, though, in that we are focused on the equivalence classes of the input data for black-box system testing, and not for coverage testing. Our work also differs from what they call "uniform statistical testing" because although we do select random data over a uniform distribution, we parameterize it according to equivalence classes.

Mayer *et al.* [50] have investigated the use of random testing with applications that have no test oracle, and Wildman *et al.* [51] have looked at testing Java components in the face of non-determinism caused by concurrency issues. However, while these two works deal with randomization in the software (either by design or by concurrency), the ML applications we investigate are deterministic (when non-determinism options are disabled) but, of course, the correct results still cannot be known *a priori*.

9.3 Metamorphic testing

Applying metamorphic testing to situations in which there is no test oracle has previously been studied by Chen *et al* [52]. In some cases, these works have looked at situations in which there cannot be an oracle for a particular application [53]; in others, the work has considered the situation in which the oracle is simply absent or difficult to implement [54]. Our work builds on theirs by applying metamorphic testing to a specific application domain (machine learning) and looking for the metamorphic relationships within those types of applications. Additionally, whereas their work has primarily focused on functions with simple numerical input domains [55], we are considering inputs that consist of larger data sets, as a result of the types of applications we are investigating.

Metamorphic properties are similar in some ways to algebraic specifications [21], though algebraic specifications often declare legal sequences of function calls that will produce a known result, typically within a given data structure (*e.g. pop(push(X))* == *X* in a Stack), but do not describe how an arbitrary function or application should react when its input is changed. The runtime checking of algebraic specifications has been explored in [56] and [57], though neither work considered the specification of metamorphic properties, and the particular issues that arise from testing without oracles. Others have looked at the automatic detection of algebraic specifications, in particular [31], and also of program invariants (*e.g.* DIDUCE [58], Daikon [30], Houdini [59], *etc.*).

Even in the cases in which program invariants, algebraic specifications, or formal specifications (using languages such as Alloy [60], ANNA [61], Larch [62], Z [63], *etc.*) are used to act as oracles (assuming they are complete, which may be an undecidable problem [64]), work to date has focused primarily on the creation of testing frameworks [65] and on consistency checking of abstract data types [64], and has not sought to create oracles for applications and functions that do not otherwise have them.

10 LIMITATIONS AND FUTURE WORK

Although our testing approach has been successful in finding defects that had not been found by the developers prior to release of their systems, we have not yet made efforts to determine the *adequacy* [66] of our testing approach, particularly for any object-oriented implementations [67] using inheritance [68], perhaps by measuring path/statement coverage or percentage of defects reliably found, and establishing success criteria. Additionally, other areas of future work remain.

10.1 Expansion to complete ML applications

Our research to date has not yet addressed the use of ML algorithm implementations in the context of overall "systems". That is, we have started looking at the dependability of ML applications but so far have studied only the "ML Engine". Future work in this area is to expand the methodology to encompass the entire system, including for instance the decision support treatment of ranking (or classification) results. The emerging generation of ML ranking systems could take various implementations of MartiRank, SVM and/or other ML algorithms, generate multiple models, and then determine which is currently "best" for the dynamic data sets at hand, and use that model for making real-time predictions or rankings; this type of system is envisioned for the motivating example (the device failure prediction application) [69], which must be dependable. Similarly, an anomaly-based IDS like PAYL could be incorporated into a larger security system that might also include rule-based intrusion detection.

It may be possible to extend the approaches to test these kinds of large systems. Our ultimate goal is to make our current and later expanded testing methodology useful outside this particular problem domain, particularly to other ML researchers who rarely cross paths with the software engineering community.

10.2 Addressing non-determinism

Some ML algorithms are intentionally non-deterministic and necessarily rely on randomization, which makes testing very difficult; our testing was assisted by the fact that it is possible to "turn off" any randomization options in the applications we investigated. More detailed trace analysis may be needed for future work on algorithms that depend on randomization.

10.3 Improvements to random data generation

In order to create test cases reminiscent of real-world data, the test data generation framework could be extended to generate data sets that exhibit the same correlations among attributes and between attributes and labels as do real-world data, building upon [70]. Ideally it could also be extended to generate arbitrarily large data sets with repeating, missing and/or categorical data such that an arbitrary ML ranking algorithm could definitively enable a "predictable" ranking, *i.e.* where there is a clear-cut "correct" output, in order to expand the "niche oracle" approach. But this may be impossible in the general case (we have noted in Section 4 that data sets that yield predictable rankings in MartiRank do not necessarily yield the same ranking in SVM).

10.4 Additional metamorphic properties

Further investigation could involve applying the metamorphic properties to other ML applications, as we initially investigated in [23], and looking to classify other properties. Additionally, as we have defined our properties independent of the actual numerical values used in the data sets, future work could consider how to initially create new data sets such that further application-specific metamorphic properties can also be revealed.

11 CONCLUSION

We have presented a methodology consisting of three approaches for testing a particular class of algorithm implementations for which there is no reliable test oracle applying to all inputs in the general case. Particularly in machine learning applications, where there is often no precise input/output specification, it can be very difficult to determine the "correct" answer. But the methodology and approaches make it possible to detect defects in the implementations and discrepancies in interpretations of the algorithm, even without relying on multiple implementations.

Addressing the testing of applications without oracles has been identified as a future challenge for the software testing community [71], and as ML applications that fall into this category become more and more prevalent and mission-critical, ensuring their dependability gains the utmost importance. We hope that our methodology and results here help others who are also concerned with the quality and dependability of such non-testable programs.

12 ACKNOWLEDGMENTS

The authors would like to thank Marta Arias, Hila Becker, T.Y. Chen, Wei Chu, Gabriela Cretu, Phil Gross, Bert Huang, Phil Long, Rocco Servedio, Swapneel Sheth, Yingbo Song, Sal Stolfo, David Waltz, and Leon Wu for their guidance and assistance. John Gallagher, Lifeng Hu, and Dokyun Lee all contributed to the development of the testing frameworks. The authors are members of the

Programming Systems Lab, funded in part by NSF CNS-0717544, CNS-0627473 and CNS-0426623, and NIH 1 U54 CA121852-01A1.

References

[1] E. J. Weyuker, "On testing non-testable programs," *Computer Journal*, vol. 25, no. 4, pp. 465–470, November 1982.

[2] M. D. Davis and E. J. Weyuker, "Pseudo-oracles for non-testable programs," in *Proc. of the ACM '81 Conference*, 1981, pp. 254–257.

[3] T. Mitchell, *Machine Learning: An Artificial Intelligence Approach, Vol. III*. Morgan Kaufmann, 1983.

[4] J. Duran and S. Ntafos, "An evaluation of random testing," *IEEE Trans. on Soft. Eng.*, vol. 10, pp. 438–444, 1984.

[5] D. Hamlet, "Random testing," Encyclopedia of Software Engineering, pp. 970–978, 1994.

[6] T. Y. Chen, S. C. Cheung, and S. Yiu, "Metamorphic testing: a new approach for generating next test cases," Department of Computer Science, Hong Kong University of Science and Technology, Tech. Rep. HKUST-CS98-01, 1998.

[7] Z. Q. Zhou, D. H. Huang, T. H. Tse, Z. Yang, H. Huang, and T. Y. Chen, "Metamorphic testing and its applications," in *Proc. of the 8th International Symposium on Future Software Technology (ISFST 2004)*, 2004.

[8] P. Gross et al., "Predicting electricity distribution feeder failures using machine learning susceptibility analysis," in *Proc. of the 18th Conference on Innovative Applications in Artificial Intelligence*, 2006.

[9] P. Long and R. Servedio, "Martingale boosting," in *Proc. of the 18th Annual Conference on Computational Learning Theory (COLT)*, 2005, pp. 79–84.

[10] V. N. Vapnik, *The Nature of Statistical Learning Theory*. Springer, 1995.

[11] T. Joachims, *Making large-Scale SVM Learning Practical. Advances in Kernel Methods - Support Vector Learning*. MIT Press, 1999.

[12] K. Wang and S. Stolfo, "Anomalous payload-based network intrusion detection," in *Proc. of Recent Advances in Intrusion Detection (RAID)*, Sept. 2004.

[13] J. A. Hanley and B. J. McNeil, "The meaning and use of the area under a receiver operating characteristic (ROC) curve," *Radiology*, vol. 143, pp. 29–36, 1982.

[14] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proc of the seventh international conference on World Wide Web*, April 1998, pp. 107–117.

[15] C. Murphy, G. Kaiser, and M. Arias, "An approach to software testing of machine learning applications," in *Proc. of the 19th international conference on software engineering and knowledge engineering (SEKE)*, 2007, pp. 167–172.

[16] P. Gross, Personal communication, 2008.

[17] C. Murphy, G. Kaiser, and M. Arias, "Parameterizing random test data according to equivalence classes," in *Proc of the 2nd international workshop on random testing*, 2007, pp. 38–41.

[18] R. Servedio, Personal communication, 2006.

[19] K. Goševa-Popstojanova and A. Grnarov, "N version programming: An unified modeling approach."

[20] J. Knight and N. Leveson, "An experimental evaluation of the assumption of independence in multi-version programming," *IEEE Transactions on Software Engineering*, vol. 12, no. 1, pp. 96–109, 1986.

[21] W. J. Cody Jr. and W. Waite, *Software Manual for the Elementary Functions*. Prentice Hall, 1980.

[22] C. Murphy, G. Kaiser, L. Hu, and L. Wu, "Properties of machine learning applications for use in metamorphic testing," in *Proc of the 20th international conference on software engineering and knowledge engineering (SEKE)*, 2008, pp. 867–872.

[23] C. Murphy, K. Shen, and G. Kaiser, "Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles," Columbia University Dept of Computer Science, Tech. Rep. cucs-044-08, 2008.

[24] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2006.

[25] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl, "GroupLens: an open architecture for collaborative filtering of netnews," in *Proc of the 1994 ACM conference on computer supported cooperative work (CSCW)*, 1994, pp. 175–186.

[26] P. C. Mahalanobis, "On the generalised distance in statistics," *Proceedings of the National Institute of Science of India*, vol. 12, pp. 49–55, 1936.

[27] S. Stolfo, Personal communication, 2008.

[28] P. Charles, "jpcap: Network packet capture facility for Java," http://sourceforge.net/projects/jpcap.

[29] V. Jacobson, C. Leres, and S. McCanne, "tcpdump," http://www.tcpdump.org.

[30] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely programming invariants to support program evolution," in *Proc. of the 21st International Conference on Software Engineering (ICSE)*, 1999, pp. 213–224.

[31] J. Henkel and A. Diwan, "Discovering algebraic specifications from Java classes," in *Proc. of the 17th European Conference on Object-Oriented Programming ECOOP*, 2003.

[32] S. Hallem, B. Chelf, Y. Xie, and D. Engler, "A system and language for building system-specific, static analyses," in *Proc of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002, pp. 69–82.

[33] L. Briand, "Novel applications of machine learning in software testing," in *Proc of the Eighth International Conference on Quality Software*, 2008, pp. 3–10.

[34] T. J. Cheatham, J. P. Yoo, and N. J. Wahl, "Software testing: a machine learning experiment," in *Proc. of the ACM 23rd Annual Conference on Computer Science*, 1995, pp. 135–141.

[35] D. Zhang and J. J. P. Tsai, "Machine learning and software engineering," *Software Quality Control*, vol. 11, no. 2, pp. 87–119, June 2003.

[36] J. Demsar, B. Zupan, and G. Leban, "Orange: From experimental machine learning to interactive data mining," [www.ailab.si/orange], Faculty of Computer and Information Science, University of Ljubljana.

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

[38] D. J. Newman, S. Hettich, C. L. Blake, and C. J. Merz, "UCI repository of machine learning databases," University of California, Dept of Information and Computer Science, 1998.

[39] P. Mell, V. Hu, R. Lippmann, J. Haines, and M. Zissman, "An overview of issues in testing intrusion detection systems," Tech. Report NIST IR 7007, National Institute of Standard and Technology.

[40] J. P. Nicholas, K. Zhang, M. Chung, B. Mukherjee, and R. A. Olsson, "A methodology for testing intrusion detection systems," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 719–729, 1996.

[41] K. V. B. Madan, K. Goševa-Popstojanova and K. S. Trivedi, "A method for modeling and quantifying the security attributes of intrusion tolerant systems," *Performance Evaluation Journal*, vol. 56, no. 1-4, pp. 167–186, 2004.

[42] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *2008 IEEE Symposium on Security and Privacy*, 2008, pp. 387–401.

[43] R. A. DeMillo and A. J. Offutt, "Constraint-based automated test data generation," *IEEE Trans. on Soft. Eng.*, vol. 17, no. 9, pp. 900–910, 1991.

[44] B. Korel, "Automated software test data generation," *IEEE Trans. on Soft. Eng.*, vol. 16, no. 8, pp. 870–879, 1990.

[45] D. Bird and C. Munoz, "Automatic generation of random self-checking test cases," *IBM Systems Journal*, vol. 22, no. 3, pp. 229–245, 1983.

[46] K. V. Hanford, "Automatic generation of test cases," *IBM Systems Journal*, vol. 9, no. 4, pp. 242–257, 1970.

[47] K. J. Lieberherr, "Parameterized random testing," in *Proc. of the 21st Design Automation Conference*, 1984.

[48] B. A. Wichmann, "Some remarks about random testing."

[49] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet, "An experimental study on software structural testing," in *Proc. of the Twenty-First International Symposium on Fault-Tolerant Computing*, June 1991, pp. 410–417.

[50] J. Mayer and R. Guderlei, "Test oracles using statistical methods," in *Proc. of the First International Workshop on Software Quality*, 2004, pp. 179–189.

[51] B. L. Luke Wildman and P. Strooper, "Dealing with non-determinism in testing concurrent java components," in *Proc of the 12th Asia-Pacific Software Engineering Conference*, 2005, pp. 393–400.

[52] T. Y. Chen, T. H. Tse, and Z. Q. Zhou, "Fault-based testing without the need of oracles," *Information and Software Technology*, vol. 44, no. 15, pp. 923–931, 2002.

[53] ——, "Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing," in *Proc. of the 2002 ACM SIGSOFT international symposium on software testing and analysis (ISSTA)*, 2002, pp. 191–195.

[54] W. K. Chan, S. C. Cheung, and K. R. P. H. Leung, "A metamorphic testing approach for online testing of service-oriented software applications," *International Journal of Web Services Research*, vol. 4, no. 1, pp. 60–80, April-June 2007.

[55] T. Y. Chen, F.-C. Kuo, T. H. Tse, and Z. Q. Zhou, "Metamorphic testing and beyond," in *Proc. of the International Workshop on Software Technology and Engineering Practice (STEP)*, 2004, pp. 94–100.

[56] I. Nunes, A. Lopes, V. Vasconcelos, J. Abreu, and L. S. Reis, "Checking the conformance of java classes against algebraic specifications," in *In Proceedings of ICFEM06, volume 4260 of LNCS*. Springer-Verlag, 2006, pp. 494–513.

[57] S. Sankar, "Run-time consistency checking of algebraic specifications," in *Proceedings of the 1991 international symposium on software testing, analysis, and verification*, 1991, pp. 123–129.

[58] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proc. of the 24th International Conference on Software Engineering (ICSE)*, 2002, pp. 291–301.

[59] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for ESC/Java," in *Proc of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, 2001, pp. 500–517.

[60] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 2, pp. 256–290, 2002.

[61] D. Luckham and F. W. Henke, "An overview of ANNA - a specification language for ADA," Stanford Univ, Tech. Rep. CSL-TR-84-265, 1984.

[62] J. V. Guttag and J. J. Horning, *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[63] J. R. Abrial, *Specification Language Z*. Oxford Univ Press, 1980.

[64] S. Sankar, A. Goyal, and P. Sikchi, "Software testing using algebraic specification based test oracles," Stanford Univ., Tech. Rep. CSL-TR-93-566, 1993.

[65] T. Miller and P. Strooper, "A framework and tool support for the systematic testing of model-based specifications," *ACM Transactions on Software Engineering and Methodology*, vol. 12, no. 4, pp. 409–439, 2003.

[66] E. Weyuker, "Axiomatizing software test data adequacy," *IEEE Trans. Software Eng., SE-12*, pp. 1128–1138, December 1986.

[67] N. L. Hashim, H. W. Schmidt, and S. Ramakrishnan, "Test order for class-based integration testing of Java applications," in *Proc of the 5th International Conference on Quality Software*, 2005.

[68] D. E. Perry and G. E. Kaiser, "Adequate testing and object-oriented programming," *Journal of Object-Oriented Programming*, vol. 2, no. 5, pp. 13–19, 1990.

[69] H. Becker and M. Arias, "Real-time ranking with concept drift using expert advice," in *Proc. of the 13th International Conference on Knowledge Discovery and Data Mining*, Aug 2007.

[70] E. Walton, "Data generation for machine learning techniques," University of Bristol, 2001.

[71] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Proc. of ICSE Future of Software Engineering (FOSE)*, May 2007, pp. 85–103.

Christian Murphy is a PhD Candidate in the Computer Science department at Columbia University. He is a member of the Programming Systems Lab, and his research focuses on software testing, computer science education, and computer-supported cooperative work. He earned a BS (*summa cum laude*) in Computer Engineering from Boston University in 1995, and an MS in Computer Science from Columbia University in 2006.

Gail Kaiser is a Professor of Computer Science and the Director of the Programming Systems Laboratory in the Computer Science Department at Columbia University. She was named an NSF Presidential Young Investigator in Software Engineering and Software Systems in 1988, and has published over 100 refereed papers in a range of software areas. Prof. Kaiser received her PhD and MS from CMU and her SB from MIT.