Properties of Machine Learning Applications for Use in Metamorphic Testing

Christian Murphy, Gail Kaiser, Lifeng Hu
Department of Computer Science, Columbia University, New York NY 10027
{cmurphy, kaiser, lh2342}@cs.columbia.edu

Abstract

It is challenging to test machine learning (ML) applications, which are intended to learn properties of data sets where the correct answers are not already known. In the absence of a test oracle, one approach to testing these applications is to use metamorphic testing, in which properties of the application are exploited to define transformation functions on the input, such that the new output will be unchanged or can easily be predicted based on the original output; if the output is not as expected, then a defect must exist in the application. Here, we seek to enumerate and classify the metamorphic properties of some machine learning algorithms, and demonstrate how these can be applied to reveal defects in the applications of interest. In addition to the results of our testing, we present a set of properties that can be used to define these metamorphic relationships so that metamorphic testing can be used as a general approach to testing machine learning applications.

1 Introduction

Making machine learning (ML) applications dependable presents a particular 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" [20].

One approach to testing such applications is to use a pseudo-oracle [7], in which multiple implementations of an algorithm process an input and the results are compared; if the results are not the same, then one or both of the implementations contains a defect. In the absence of multiple implementations, however, metamorphic testing [2] [22] can be used to produce a similar effect: input can be modified in such a manner that it should produce the same output as the original, and if it does not, then a defect must exist. Of course, this can only show that a defect does exist and cannot demonstrate the absence of defects, since the correct output cannot be known in advance, but metamorphic testing provides a powerful technique to testing such "non-testable programs" by use of a built-in pseudo-oracle.

A challenge of metamorphic testing is to determine the so-called metamorphic relationships that can be used to transform an input such that its new output will be predictable, given the output produced by the original input. This generally requires domain knowledge and/or familiarity with the algorithm's implementation, and these relationships may not necessarily apply to other applications. In this paper, we seek to create a taxonomy of metamorphic relationships that are applicable to input data of both supervised and unsupervised machine learning applications, including the inclusion and omission of data, permutation, and modification of numerical values. Our contribution is a set of properties that can be used to define these relationships so that metamorphic testing can be used as a general approach to testing machine learning applications.

Previously we have investigated approaches to testing such applications by considering properties of their data sets [14] and by using random testing [15]. In this paper, we first present our analysis of the metamorphic properties of MartiRank [9], a ranking implementation of the Martingale Boosting algorithm [12]. The result of this investigation is then used to guide the creation of metamorphic relationships that can be used in testing. We apply metamorphic testing to MartiRank, as well as to two other machine learning applications: an implementation of Support Vector Machines (SVM) [18] called SVM-Light [11], and the anomaly-based intrusion detection system PAYL [19], and report our findings.

2 Background

2.1 Metamorphic testing

Metamorphic testing [2] [22] 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. Instead of being an approach for test case selection, it is a methodology of reusing input test data to create additional test cases whose outputs can be predicted. In metamorphic testing, if input x produces an output $f(x)$, a transformation function T can then be applied to the input to produce $T(x)$; this transformation is based on a metamorphic property of the function, such that the output $f(T(x))$ can then be predicted, based on the (already known) value of $f(x)$.

A classic example is the sine function. If we have built a function to compute sine, and for some selected input x we have computed $\sin(x) = y$, then we can create the test input $(x + 2\pi)$ and expect that $\sin(x + 2\pi)$ will also equal y, based on the metamorphic property of sine that $\sin(\alpha) = \sin(\alpha + 2\pi)$. Similarly, given $\sin(x) = y$, we can create the test input $-x$ and expect that $\sin(-x)$ should be $-y$, based on the metamorphic property of sine that $\sin(-\alpha) = -\sin(\alpha)$.

It is clear that this approach is very useful in the absence of an oracle. Regardless of the values of x and y, if $\sin(-x)$ does not equal $-\sin(x)$, then there must be a defect in the implementation of the sine function. Although the use of these simple identities for testing numerical functions is not unique to metamorphic testing [6], the approach can be used on a broader domain of any functions that display metamorphic relationships, including machine learning applications.

2.2 Related work

Applying metamorphic testing to situations in which there is no test oracle has been studied in great detail by Chen *et al.* [4] [5]. 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 [3], we are considering inputs that consist of larger (possibly alphanumeric) data sets, as a result of the types of applications we are investigating.

```
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
...
```

Figure 1. Example of part of a data set used by supervised ML ranking algorithms

Although there has been much work that applies machine learning techniques to software engineering in general and software testing in particular (e.g., [1]), there has thus far been very little published work in the reverse sense: applying software testing techniques to ML applications that have no reliable test oracle. Orange [8] and Weka [21] are two of several frameworks that aid ML developers, but the testing functionality they provide is focused on comparing the quality of the results, and not evaluating the "correctness" of the implementations. Similarly, testing of intrusion detection systems [13] [16] 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.

2.3 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 and, in some cases, a *label*. 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. The label, if it exists, indicates how the example is categorized. In some cases a label of 1 is considered a *positive example*, and a 0 represents a *negative example*; without loss of generality, we only discuss these cases here. 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.

Unsupervised ML applications also execute in training and testing phases, but in these cases, the training data sets necessarily do not have labels. Rather, 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 if the same properties exist. Data mining and collaborative filtering are two well-known examples of unsupervised learning.

2.4 Applications investigated

In this work we looked at three ML applications: MartiRank [9], SVM-Light [11] and PAYL [19].

The development of MartiRank was commissioned by a company for potential future experimental use in predicting impending electrical device failures, using historic data of past device 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.

In the training phase, MartiRank, which is a supervised ML algorithm, 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/Nth 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 of an attribute is assessed using a variant of the Area Under the Curve (AUC) [10] 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 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
```

Figure 2. Sample MartiRank model

Figure 2 shows a sample model. 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.

The second supervised ML algorithm we invesigateed, SVM [18], belongs to the "linear classifier" family of ML algorithms that attempt to find a (linear) 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. The type of 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

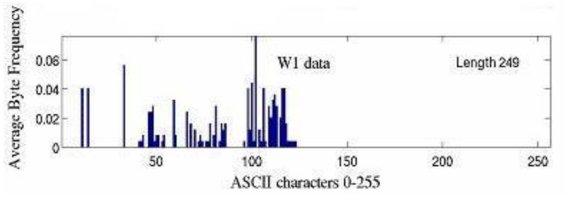

Figure 3. Sample payload byte distribution

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.

We also investigated an intrusion detection system called PAYL. 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 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 has been developed for research purposes at Columbia University.

As PAYL is an example of unsupervised machine learning, its training data simply consists of a set of TCP/IP network payloads (streams of bytes), without any associated lables or classificiation. During its training phase, it computes the mean and variance of the byte value distribution for each payload length in order to produce a model; Figure 3 shows an example of such a distribution. During the second ("detection") phase, each incoming payload is scanned and its byte value distribution is computed. This new payload distribution is then compared against the model (for that length) using the Mahalanobis distance, which is a way of comparing two sets of data but unlike Euclidean distance does not depend on the scale of the values; if the distribution of the new payload is above some threshold of difference from the norm, PAYL flags the packet as anomalous and generates an alert. PAYL may also raise an alert in other circumstances, for instance if the payload length had never been seen before in the training data.

3 Approach

We previously reported on our testing of ML ranking applications (MartiRank and SVM-Light) in which we developed test cases by analyzing the problem domain, analyzing the algorithms as defined in pseudo-code, and analyzing the runtime options [14]. This then allowed us to devise equivalence partitions that served as guidelines for the creation of datasets using random testing [15].

We then went back to these applications and used our knowledge of the algorithms to identify metamorphic relationships (previously unpublished) that would give us another way of testing such applications in the absence of an oracle. These properties are described in Section 4.

Once we had enumerated and categorized the different types of metamorphic properties, we used these principles in our testing. We first tested an implementation of MartiRank, and then sought to also apply these to SVM-Light (another ranking application) and the anomaly-based intrusion detection system PAYL, on which we conducted metamorphic testing using the same guidelines. Section 5 discusses the results of our testing.

4 Metamorphic properties

We begin by describing our observations of the metamorphic properties of MartiRank [9]. 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* so that input *T(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* so that the combinations *r(T(K), L)*, *r(K, T(L))* and *r(T(K), T(L))* all produce *R* as well.

Based on our analysis of the MartiRank algorithm, we noticed that it is not the actual values of the attributes that are important, but it is the *relative* values that determine the model. 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, and not the particular values of the examples' attributes. The model declares which attributes to sort to get the best ordering of the labels; in Figure 1, 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, 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 should not affect the output.

It should also 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). As MartiRank is based on sorting, in the cases where all the values for a given attribute are distinct, it is clear that the sorted order will still be the same regardless of the original input order. Thus, MartiRank also has a **permutative** metamorphic property, albeit only limited to certain inputs.

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* so that input *T(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* so that *r(T(K), L)*, *r(K, T(L))* and *r(T(K), T(L))* all can be predicted based on *R*. 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.

We mentioned above that multiplying all attributes by a positive constant should not affect the model. On the other hand, mulitplying 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 them all by a negative constant, and sort in descending order, the original sorted order would be kept intact. 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.

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 model shown in Figure 2. 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.

5 Case studies

As a result of our investigation of MartiRank, we have identified six metamorphic properties of supervised ML applications: additive, multiplicative, permutative, invertive, inclusive, and exclusive. Following that analysis, we conducted metamorphic testing using those properties.

5.1 Metamorphic testing of MartiRank

After identifying the metamorphic properties of MartiRank, we constructed corresponding test cases and were able to detect a defect in the implementation. Another of its invertive properties is that if all of the labels in the training data are negated (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 first with a simple toy data set and then 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 algorithm should allow for negative labels (-1 vs. +1 is sometimes used in other applications), of course.

5.2 Analysis of SVM-Light

Our testing demonstrated that SVM exhibits the same metamorphic properties shown by MartiRank. Almost all of the transformations that we tested based on these metamorphic properties resulted in a modification of the output compared to the original, but this modification was always predictable or could be converted to the original output with additional transformations. For instance, if a training data set were transformed using an additive, multiplicative, and/or invertive relationship, then the corresponding model (hyperplane) would be affected by being shifted, expanded, or inverted 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 would stay the same. Our testing revealed this to be the case in SVM-Light.

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. Lastly, SVM has the permutative property because the ordering of the examples in the training data should not affect the resulting hyperplane that separates them.

5.3 Metamorphic testing of SVM-Light

As we originally reported in [14], the SVM-Light implementation has a bug in which permuting the training data caused it to create different models for different input orders. This occurred even when all attributes and labels were distinct - thus removing the possibility that ties between equal values would be broken depending on the input order.

Our analysis of the SVM algorithm indicates that it theoretically should produce the same model 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 [17]. Numerical methods and heuristics are used to quickly converge to the optimum. However, the optimum is not necessarily achieved, but instead this process stops after some threshold of improvement. This is one important area in which the implementation deviates from the specification, as revealed by metamorphic testing.

Although we have only considered ranking algorithms thus far, we believe that classification algorithms would display the same properties because of the similarity of the algorithms in terms of the ways in which they treat the data; this is left as future work.

5.4 Analysis of PAYL

We next sought to determine whether the properties we used to guide metamorphic testing of MartiRank could also be applied to a different type of ML application. The application we chose was PAYL [19], an anomaly-based intrusion detection system.

Because the model generated by PAYL in the training phase represents the distribution of byte values in the TCP/IP payload (see Figure 3), it is clear that it exhibits the additive and multiplicative properties. 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.

PAYL also has an invertive property. An "inverse" of the distribution can be obtained by subtracting each byte value from the maximum (255, or 0xFF), so that frequently-seen values become less frequent, and vice-versa. If the same treatment is applied to the payloads in the testing data, then the same alerts should be raised, 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.

5.5 Metamorphic testing of PAYL

We then conducted testing of PAYL by using data sets generated via these metamorphic relationships. By using the exclusive metamorphic property, we were able to detect two defects in PAYL. We started with training data that had payloads of various sizes, including 274 bytes, and created a model that was applied to a set of testing data, which also included a payload of 274 bytes; PAYL raised no alerts. We then removed all payloads of 274 bytes from the training data and applied the model to the same (unmodified) testing data, expecting that the payload of 274 bytes in the testing data would cause PAYL to raise a "length-never-seen-before" alert. However, PAYL raised an anomaly alert for the payload of length 274, even though there was no payload of that length in the training data. An alert was correctly being raised, but it was the wrong type.

Additionally, PAYL unexpectedly raised *both* anomaly alerts *and* "length-never-seen-before" alerts for payloads of 1448 bytes, which theoretically should never happen (since it can only be anomalous if that length had actually been seen before). Upon further investigation, we determined that PAYL actually should have raised the "length-never-seen-before" alert from the first set of training data, since there were no payloads of that length. So not only were the alerts not being raised in the first place, but false positives were then being raised in the second.

Our key result, though, was that we were able to verify that PAYL exhibits the same six metamorphic properties as MartiRank, and then use these properties to drive metamorphic testing and find important defects in PAYL.

6 Conclusion and future work

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, which we have shown to reveal defects in the applications of interest.

Further investigation would involve applying these metamorphic properties to other ML applications, 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.

We have found metamorphic testing to be an efficient and effective approach to testing ML applications. We hope that our findings here and the identification of metamorphic properties help others who are also concerned with the quality of non-testable programs.

7 Acknowledgments

The authors would like to thank T.Y. Chen, Marta Arias, Hila Becker, Gabriela Cretu, Phil Gross, and David Waltz for their assistance. Murphy and Kaiser are members of the Programming Systems Lab, funded in part by NSF CNS-0717544, CNS-0627473, CNS-0426623 and EIA-0202063, and NIH 1 U54 CA121852-01A1.

References

[1] 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*, pages 135–141, 1995.

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

[3] 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)*, pages 94–100, 2004.

[4] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Fault-based testing without the need of oracles. *Information and Software Technology*, 44(15):923–931, 2002.

[5] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. 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)*, pages 191–195, 2002.

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

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

[8] 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.

[9] 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.

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

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

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

[13] P. Mell et al. An overview of issues in testing intrusion detection systems. Tech. Report NIST IR 7007, National Institute of Standard and Technology.

[14] 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)*, pages 167–172, 2007.

[15] 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*, pages 38–41, 2007.

[16] 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*, 22(10):719–729, 1996.

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

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

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

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

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

[22] 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.