Parameterizing Random Test Data According to Equivalence Classes

Christian Murphy
Dept. of Computer Science
Columbia University
New York, NY
cmurphy@cs.columbia.edu

Gail Kaiser
Dept. of Computer Science
Columbia University
New York, NY
kaiser@cs.columbia.edu

Marta Arias
Center for Computational Learning Systems
Columbia University
New York, NY
marta@ccls.columbia.edu

ABSTRACT
We are concerned with the problem of detecting bugs in machine learning applications. In the absence of sufficient real-world data, creating suitably large data sets for testing can be a difficult task. Random testing is one solution, but may have limited effectiveness in cases in which a reliable test oracle does not exist, as is the case of the machine learning applications of interest. To address this problem, we have developed an approach to creating data sets called "parameterized random data generation". Our data generation framework allows us to isolate or combine different equivalence classes as desired, and then randomly generate large data sets using the properties of those equivalence classes as parameters. This allows us to take advantage of randomness but still have control over test case selection at the system testing level. We present our findings from using the approach to test two different machine learning ranking applications.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification.
D.2.5 [**Software Engineering**]: Testing and Debugging.

General Terms
Reliability, Verification.

Keywords
Machine learning, Software testing, Random test data generation.

1. INTRODUCTION
During our investigation of the software testing of machine learning (ML) applications, we immediately encountered the problem of the availability of testing data. Real-world data sets are not always accessible and, even in the cases 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; however, these applications are used in the real world on extremely large data sets. Random testing [5, 7] could, of course, be an option in these situations, but is difficult given that machine learning applications fall under the category of "non-testable programs" [20]; that is, there is no reliable test oracle to ensure that the output is correct for the given random input. In fact, Hamlet [7] 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." Thus pure random testing may not be the best approach in these cases, either.

Our approach, then, has been to combine partition testing with what we call "parameterized random test data generation". 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. We present a hybrid testing approach that couples the benefits of using randomness with the necessary control over the properties of the testing data.

In this paper, we describe the manner in which we generate data for testing ML applications that implement ranking algorithms (a requirement of a real-world problem domain). We have developed a tool that, given certain parameters, will create randomly-generated data sets that are targeted at testing specific equivalence classes. We show how we used randomness in the test data generation to reveal bugs and inconsistencies in the ML applications that could not otherwise have been shown by real-world or hand-crafted data.

2. BACKGROUND
We are concerned with testing a decision support system that uses ranking of susceptibility to failure of some large number of electrical devices in a real-world industrial environment where preventative maintenance can potentially be guided by such susceptibility. 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 useful. The prototype application uses implementations of both the MartiRank [12] and SVM [18] algorithms to produce rankings; the dependability of the application has real-world industrial implications, rather than just academic interest. We do not discuss the full application further in this paper, see [6] for details.

2.1 Data Sets in the Problem Domain

In general, data sets used in ML (both ranking and classification) consist of a collection of *examples*, each of which has a number of *attribute* values and a *label*. The examples can be thought of as rows in a table, each of which represents one item from which to learn (in our case, electrical devices). The attributes are the columns of the table, and the label indicates how the example is categorized; for us, it is an indication of how many times the device failed in a given time period. In some phases of supervised ML learning there is a label, and in some there is none; without loss of generality, here we only discuss the former.

In previous work [14], we describe how we determined equivalence classes and performed test case selection for implementations of ML algorithms. The first part of our testing approach was to consider the problem domain and try to determine equivalence classes based on the properties of real-world data sets. We particularly looked for traits that may not have been considered by the algorithm designers, such as data set size, and the potential ranges (or even the existence) of values.

The data sets of interest are very large, both in terms of the number of attributes (hundreds) and the number of examples (tens of thousands). The label could be any non-negative integer, although it was 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).

Though much of the real-world data of interest indeed consists of numerical values – including floating point decimals, dates and integers – some of the data is instead alphanumeric. 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 a device manufactured by "General Electric" as more or less than something made by "Westinghouse" just because of their alphabetical ordering. To solve this problem, the data sets use categorical data. Categorical data refers to attributes in which there are K different distinct 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 0's.

As would be expected in any large data set, many non-categorical attributes had repeated values; however, in the real-world data, many values were missing for various reasons, raising the issues of breaking "ties" and handling unknowns during sorting.

2.2 Machine Learning Ranking Algorithms

We investigated two ranking algorithms. MartiRank [6] was specifically designed with the device failure application in mind. The algorithm looks for the combination of sorting and segmenting the data to try to achieve the best "quality". Our testing concerned three implementations of the MartiRank algorithm: the original, written in Perl; a C version, written to improve performance (speed) and introduce some experimental options to try to improve quality; and another implementation also written in C, designed to minimize the costly overhead of repeatedly sorting the attribute values. SVM [18] belongs to the "linear classifier" family of ML algorithms that attempt to find a (linear) hyperplane that separates examples from different classes. 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. For our testing, we investigated an implementation called SVM-Light [9].

2.3 Related Work

There has been much research into the generation of test data sets [4, 10], though whereas much of the early work in random test data generation [1, 8] 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 [11] 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 [21] proposes something similar to our 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 and not system-level testing.

Our work is also similar to that of Thénevod-Fosse *et al.* [17], who labeled their approach "structural statistical testing" in that random input are selected according to a 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 an equal distribution, we parameterize it according to equivalence classes.

Repositories of "reusable" ML real-world data sets have been collected (e.g., the UCI Machine Learning Repository [15]) for the purpose of comparing result quality, but not for testing in the software engineering sense. Of course, from an ML perspective, random test data is not generally as useful as real-world or hand-crafted data in testing the predictive power of an algorithm because it will not exhibit any trends, and therefore there is nothing for the algorithm to learn; however, random data can be used for testing the correctness of the results, if not the quality, and for detecting bugs.

Lastly, Mayer *et al.* [13] have investigated the use of random testing with applications that have no test oracle. However, their work is focused on randomized software, which is different from the case of the ML applications we investigate, which are deterministic but, of course, the results cannot be known *a priori*.

3. DATA GENERATION FRAMEWORK

Although real-world data sets were available in abundance, we required more control over the data so that we could ensure that we addressed all equivalence classes. We thus created a tool that

randomly generates values and puts them in the data set according to certain parameters. This allowed us to separately test different equivalence classes and ultimately create a suite of regression tests that addressed those classes, focusing on boundaries. The parameters also include the number of examples, the number of attributes, and the names of the output test data set files.

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 shuffle 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; there is thus a possibility that numbers may repeat, but originally this was not guaranteed. However, we have modified the tool so that it can ensure at least one set of repeating numbers. The tool currently only generates integers but could easily be modified to generate floating point numbers or dates, depending on the problem domain.

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

Parameters could 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-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.

In addition to the data generation tool, we created a complementary utility that randomly permutes the order of the examples (rows) and attributes (columns) of the data set. We expected (and later showed) that the order of the input data would affect the resulting output, even though in theory it should not.

4. 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, but it is not possible to detect minor issues or even tell if the output is correct. However, because we had three implementations of the MartiRank algorithm, we were able to use them as "pseudo-oracles" [3] for each other to see if we were getting the correct output; this led to most of our more interesting findings. In the case of SVM, though, for which we only tested one implementation, we looked for consistency more than correctness.

4.1 Testing with repeating values

Since any ML ranking algorithm will use sorting, our test case selection criteria led us to look at data sets with repeating values, to see how the software would deal with ties that may come from sorting examples with the same attribute values. In the real-world data sets, attributes like voltage level and activation date involve many repeating values.

Our initial testing with small hand-crafted data sets showed that two of the MartiRank implementations were producing the same results, but a third was not. We then used the parameterized random data generation tool to automatically create larger data sets with repeating attribute values, and confirmed our intuition that the two implementations that agreed were using sorting routines that were "stable" (i.e., they maintain the relative order of the examples from the previous step when the values are the same), whereas the other was using a faster sorting algorithm that was not a stable sort (in particular producing a different order than a stable sort in the case of "ties"). The MartiRank algorithm as defined in [12] did not address the specific implementation issue of which sorting approach to use, so different implementation decisions led to different results. This does not matter with respect to formal proofs, but does with respect to consistent testing.

4.2 Testing with sparse data sets

Our selection criteria then pointed us towards 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 as a parameter (0.5%, 1%, 5%, 10%, 20%, and 50%).

In these tests, all implementations were initially generating different results, and there was no way to know which was "correct" since the MartiRank algorithm does not dictate how to handle missing values. Consulting with the ML researchers, we 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. For instance, when the values "4 A 5 2 1 B C 3" are sorted in ascending order (with A, B and C being the missing values), the result should be "1 A 2 3 4 B C 5". Other deterministic options for handling this case (such as putting all missing values at the end) were considered, but this was deemed to be most in the MartiRank spirit since it kept the examples with missing values consistent with respect to any previous sort order.

4.3 Testing with categorical data

Because categorical data provides a combination of necessarily repeating (all 0s or 1s) and sometimes missing values, we created data sets with categorical attributes as part of our test data suite to be used for regression testing. We used a data set that included categorical data to discover that a bug had been introduced during refactoring of the Perl implementation by the incorrect use of a global variable in a crucial calculation. 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.

4.4 Testing effects of permuting input data

Among all our findings with respect to the SVM implementation we tested, the most relevant one here 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 (i.e., the same ranking) 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 [16]. 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. This is one important area in which the implementation deviates from the specification.

5. EVALUATION

The testing framework facilitated our work by aiding us in the creation, execution and analysis of the test cases. 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 and their boundaries. The data generation tool proved to be simple and reliable, compared to alternative approaches we considered, such as culling real-world data.

Additionally, the tool could be used for the testing of any ML ranking algorithm, not just MartiRank and SVM; it could also easily 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 the MartiRank implementations (csv files) and the other for SVM-Light (a "sparse" attribute-value pair representation that enables more compact representation of data sets with a high proportion of missing values).

The framework does have some limitations. Ideally it should 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. But this may be impossible in the general case (we have noted in [14] that data sets that yield predictable rankings in MartiRank do not necessarily yield the same ranking in SVM). In addition, in order to create test cases reminiscent of real-world data, the framework should 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 [2] and [19].

6. ACKNOWLEDGMENTS

We would particularly like to thank David Waltz, Hila Becker, Wei Chu, John Gallagher, Philip Gross, Bert Huang, David Lee, Phil Long and Rocco Servedio for their assistance and encouragement. Murphy and Kaiser are members of the Programming Systems Lab, funded in part by NSF CNS-0627473, CNS-0426623 and EIA-0202063, NIH 1 U54 CA121852-01A1, and are also affiliated with the Center for Computational Learning Systems (CCLS). Arias is fully supported by CCLS, with funding in part by Consolidated Edison Company of New York.

7. REFERENCES

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

[2] H. Christiansen and C.M. Dahmke, *A Machine Learning Approach to Test Data Generation*, Roskilde University, Roskilde, Denmark.

[3] M.D. Davis and E.J. Weyuker, "Pseudo-Oracles for Non-Testable Programs", *ACM '81 Conference*, 1981, 254-257.

[4] R. A. DeMillo and A. J. Offutt, "Constraint-Based Automated Test Data Generation", *IEEE Trans. on Soft. Eng. vol 17, no. 9*, 1991, 900-910.

[5] J. Duran and S. Ntafos, "An Evaluation of Random Testing", *IEEE Trans. on Soft. Eng. vol 10*, 1984, 438-444.

[6] P. Gross *et al*.,"Predicting Electricity Distribution Feeder Failures Using Machine Learning Susceptibility Analysis", *Proc. of the 18th Conference on Innovative Applications in Artificial Intelligence*, Boston MA, 2006.

[7] D. Hamlet, "Random Testing", in *Encyclopedia of Software Engineering*, Wiley, New York, 1994, 970-978.

[8] K.V. Hanford, "Automatic Generation of Test Cases", *IBM Systems Journal* vol. 9 no. 4, 1970, 242-257.

[9] T. Joachims, *Making large-Scale SVM Learning Practical. Advances in Kernel Methods - Support Vector Learning*, B. Schölkopf, C. Burges and A. Smola (ed.), MIT-Press, 1999.

[10] B. Korel, "Automated Software Test Data Generation", *IEEE Trans. on Soft. Eng. vol.16 no.8*, August 1990, 870-879.

[11] K.J. Lieberherr, "Parameterized Random Testing", *Proc. of the 21st Design Automation Conference*, 1984.

[12] P. Long and R. Servedio, "Martingale Boosting", *Eighteenth Annual Conference on Computational Learning Theory*, Bertinoro, Italy, 2005, 79-94.

[13] J. Mayer, R. Guderlei, "Test Oracles Using Statistical Methods", *Proc. of the First International Workshop on Software Quality*, 2004, 179-189.

[14] C. Murphy, G. Kaiser, M. Arias, "An Approach to Quality Assurance of Machine Learning Applications", to appear in *Proc. of SEKE 2007*, Boston, July 2007.

[15] D.J. Newman, S. Hettich, C.L. Blake and C.J. Merz, UCI *Repository of machine learning databases* [http://www.ics.uci.edu/~mlearn/MLRepository.html], University of California, Department of Information and Computer Science, Irvine CA, 1998.

[16] R. Servedio, personal communication, 2006.

[17] P. Thévenod-Fosse, H. Waeselynck and Y. Crouzet, "An Experimental Study on Software Structural Testing: Deterministic Versus Random Input Generation", *Proc. Of the Twenty-First International Symposium on Fault-Tolerant Computing*, Montreal, June 1991, 410- 417.

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

[19] E. Walton, *Data Generation for Machine Learning Techniques*, University of Bristol, 2001.

[20] E.J. Weyuker, "On Testing Non-Testable Programs", *Computer Journal vol.25 no.4*, November 1982, 465-470.

[21] B.A. Wichmann, "Some Remarks About Random Testing", [http://www.npl.co.uk/scientific_software/publications/validation/random_testing.pdf