An Approach to Software Testing of Machine Learning Applications

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

Some machine learning applications are intended to learn properties of data sets where the correct answers are not already known to human users. It is challenging to test such ML software, because there is no reliable test oracle. We describe a software testing approach aimed at addressing this problem. We present our findings from testing implementations of two different ML ranking algorithms: Support Vector Machines and MartiRank.

1. Introduction

We investigate the problem of making machine learning (ML) applications dependable, focusing on software testing. Conventional software engineering processes and tools do not neatly apply: in particular, it is challenging to detect subtle errors, faults, defects or anomalies (henceforth "bugs") 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 do not guarantee that an application implements or uses the algorithm correctly, and thus software testing is needed. Our testing, then, 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.

In this paper, we describe our approach to testing ML applications, in particular those that implement ranking algorithms (a requirement of the real-world problem domain). 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.

Our approach for creating test cases consists of three facets: analyzing the problem domain and the corresponding real-world data sets; analyzing the algorithm as it is defined; and analyzing the implementation's runtime options. While this approach is conventional, not novel, a number of issues arise when applying it to determining equivalence classes and generating data sets for testing ML ranking code.

We present our findings to date from two case studies: our first concerns the Martingale Boosting algorithm, which was developed by Long and Servedio [3] initially as a classification algorithm and then adapted by Long and others into a ranking algorithm called MartiRank [4]; we then generalized the approach and applied it to an implementation of Support Vector Machines (SVM) [5] called SVM-Light [6].

2. Background

We are concerned with the development of an ML application commissioned by a company for potential future experimental use in predicting impending 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. The prototype application uses both the MartiRank and SVM algorithms to produce rankings; the dependability of the implementations has real-world implications, rather than just academic interest. We do not discuss the full application further in this paper; see [4] for details.

2.1. Machine learning fundamentals

In general, there are two phases to supervised machine learning. The first phase (called the *learning* phase) analyzes a set of *training* data, which consists of a number of *examples*, each of which has a number of *attribute* values and one *label*. 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, 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.

One complication in this effort 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" (i.e., the rules generated during training on a set of examples) and "validation" (measuring the accuracy achieved when using the model to rank the training data set with labels removed, rather than a new data set).

2.2. MartiRank and SVM

MartiRank [4] was specifically designed as a ranking algorithm with the device failure application in mind. In the learning 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 N^{th} round, each containing $1/N^{th}$ of the total number of device failures. 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) [7] 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 the sorting rules from the model to the testing data set to produce the ranking (the final sorted order).

SVM [5] 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 K dimensions (since it has K attributes), and attempts to segregate the examples with a hyperplane of K-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 then recording its distance from the hyperplane. The examples are then ranked according to this distance.

2.3. Related work

Although there has been much work that applies machine learning techniques to software engineering in general and software testing in particular (e.g., [8]), there seems to be very little published work in the reverse sense: applying software testing techniques to ML software. There has been much research into the creation of test suites for regression testing [9] and generation of test data sets [10, 11], but not applied to ML code. Repositories of "reusable" data sets have been collected (e.g., the UCI Machine Learning Repository [12]) for the purpose of comparing result quality, but not for the software engineering sense of testing. Orange [13] and Weka [14] are two of several frameworks that aid ML developers, but the testing functionality they provide is focused on comparing the quality of the results, not evaluating the "correctness" of the implementations.

3. Software Testing Approach

3.1. Analyzing the problem domain

The first part of our approach 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, and what sort of precision is expected when dealing with floating point numbers.

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).

The attribute values were either numerical or categorical. Many non-categorical attributes had repeated values and many values were missing, raising the issues of breaking "ties" during sorting and handling unknowns. We do not discuss categorical attributes further (because we found no relevant bugs).

3.2. Analyzing the algorithm as defined

The second element to our approach to creating test cases 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 bugs 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 "perfect" ranking. 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.

For instance, we know that SVM 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 X, Y or Z" or "anything with the attribute between A and B" or "anything with the attribute above M". We could also create data sets that are predictable but have noise in them to try to confuse the algorithm.

Generating 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 a perfect ranking.

3.3. Analyzing the runtime options

The last part of our approach to generating test cases for ML algorithms is to look at their 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. So we created test cases that randomly permuted such a data set; regardless of the input order, we should see the same final ranking each time.

SVM-Light has numerous runtime options that deal with optimization parameters and variables used by the different kernels for generating the hyperplane(s). To date we have only performed software testing with the default options, although we did test with three of the different kernels: linear, polynomial, and radial basis.

4. Findings

To facilitate our testing, we created a set of utilities targeted at the ML algorithms we investigated. The utilities currently include: a data set generator; tools to compare a pair of output models and rankings; several trace options inserted into the ML implementations; and tools to help analyze the intermediate results indicated by the traces.

Using our testing approach, 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 parameterize the test case selection criteria applied by our data generator tool to automate creation of appropriate input data sets.

We first applied our approach to creating the selective test cases for execution by MartiRank. We then generalized the approach and applied it to SVM-Light. Here we describe our most important findings.

4.1. Testing MartiRank

The MartiRank implementation did not have any difficulty handling large numbers of examples, but for larger than expected numbers of attributes it reproducibly failed (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. We refactored the code and the failures disappeared.

Our approach to creating test cases based on analysis of the pseudocode led us to notice that the MartiRank algorithm does not explicitly address how to handle negative labels. Because the particular implementation we were testing was designed specifically to predict device failures, which would never have a negative number as a label, this was not considered during development. However, the implementation did not complain about negative labels but produced obviously incorrect results when a negative label existed. In principle a general-purpose ranking algorithm should allow for negative labels (-1 vs. +1 is sometimes used in other applications).

Also, by inspecting the 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 would be maintained in the case of ties. We constructed data sets such that, if a stable sort were used, a perfect ranking would be achieved because examples 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 perfect 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 we substituted another, "stable" sorting routine.

4.2. Regression testing

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. Development of the MartiRank implementation is ongoing, and our data sets have been 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 for 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 were once again being generated.

4.3. Testing multiple implementations

Davis and Weyuker suggest a "pseudo-oracle" as the solution to testing non-testable programs, i.e. constructing a second implementation and comparing the results of the two implementations on the same inputs [2]. Should multiple implementations of an algorithm happen to exist, our approach could be used to create test cases for such comparison testing. If they are *not* producing the same results, then presumably one or both implementations has a bug.

There are conveniently multiple implementations of the MartiRank algorithm: the original written in Perl and then a faster version written in C (most of the above discussion is with respect to the C implementation, except the bug mentioned for regression testing was in the Perl version). Using one as a "pseudo-oracle" for the other, we noticed a difference in the rankings they produced during testing with the equivalence class of missing attribute values. Using traces to see how the examples were being ordered during each sorting round, we noticed that the presence of missing values was causing the known values to be sorted incorrectly by the Perl version. This was due to using a Perl starship comparison operator that assumed transitivity among comparisons even when one of the values in the comparisons was missing, which is incorrect.

4.4. Generalization to SVM-Light

After completing the testing of MartiRank, we then generalized the approach and applied it to SVM-Light.

We did not uncover any issues with respect to most of the test cases involving unexpected values (such as negative labels or missing attributes) or repeating

attribute values. However, with the linear and polynomial kernels, permuting the training data caused SVM-Light 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 or missing values would be broken depending on the input order. We confirmed that these models were not "equivalent" by using the same testing data with each pair of such different models, and indeed obtained two different rankings. The practical implication is that the order in which the training data happens to be assembled can have an effect on the final ranking. This did not happen for the radial basis kernel in any of our tests to date.

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 [15]. 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.

Our other key findings came from those test cases involving "predictable" rankings. We created a small data set by hand that should yield a perfect ranking in SVM: 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 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 row's first attribute incremented by 1. The linear and radial basis kernels found the perfect ranking but the polynomial kernel did not. We assumed that this was because of the noise, so we removed the noise and it indeed found the perfect ranking. This was the only case in our testing in which noise in the data set caused SVM-Light to fail to find the perfect ranking.

In other test cases with predictable rankings, we noticed that different kernels exhibited different behaviors with respect to how they performed on different types of data sets. For example, the linear and polynomial kernels could find the perfect rankings when a particular attribute had a *range* of values that correlated to a label of 1, but the radial basis kernel only found the perfect rankings when an attribute had a *single* value that correlated. 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.

Finally, although there are multiple implementations of the SVM algorithm, our testing did not include comparison testing (using one as a "pseudo-oracle" for another). We leave this as future work.

5. Discussion

Our approach was successful in that it helped us discover bugs in the implementations and discrepancies from the stated algorithms. By inspecting the algorithms, we could create predictable data sets that should yield perfect rankings and indicate whether the algorithm was implemented correctly; we could also see where the imprecisions were, especially in the case of MartiRank with respect to sorting missing or repeated values. Lastly, by considering the runtime options, we conceived test cases that permuted the input data, revealing an inconsistency in SVM-Light.

Possibly the most important thing we discovered is that what is "predictable" for one algorithm will not necessarily lead to a perfect ranking in another. For instance, in cases when the examples with a 1 label have a particular attribute whose value is in the middle of a range, it is hard for MartiRank to get that example towards the top of the ranking, though this is possible for most SVM kernels.

Also, as noted, although MartiRank is based on sorting, it does not specify whether the sorting of attributes should use a "stable" sort, so we found problems with how repeated or missing attribute values were handled. We also noticed that the algorithm states that each partition should have the same number of failures, but it does not address how many *non-failures* should appear in each partition, i.e. whether the dividing point is above or below those non-failures, or how the failures should be split amongst partitions when it is impossible to do so evenly.

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 the 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 unstable sorting problem.

Although our testing to date has focused only on two ML algorithms, by developing the testing approach for MartiRank and then applying it to SVM-Light, we have shown that our approach and even specific test

cases can be generalized to other ML ranking algorithms, which are likely to require many of the same equivalence classes discussed here. The general approach also seems appropriate to software testing of the implementations of supervised ML classification algorithms. The primary difference is that classification algorithms seek to categorize each single example, not rank-order a group of them, but investigating the problem domain and considering the algorithm as defined as well as the code's runtime options (if any) should still apply.

6. Limitations and Future Work

We have not yet addressed the issue of test suite adequacy, e.g. to extend our data generation tool to automatically generate sets of test cases that collectively cover all statements, branches or paths. Further, mutation analysis could be used for evaluating and improving the effectiveness of a given test suite. We leave these as future directions.

Other future work could include the investigation of automatically generating data sets that exhibit the same correlations among attributes and between attributes and labels as do real-world data, such as in [16]. Additionally, since some ML algorithms are intentionally non-deterministic and necessarily rely on randomization, more detailed trace analysis techniques should be investigated towards determining software implementation correctness.

7. Acknowledgements

Numerous people contributed to this effort. We would particularly like to thank David Waltz, Wei Chu, John Gallagher, Philip Gross, Bert Huang, 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.

8. References

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

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

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

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

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

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

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

[8] T.J. Cheatham, J.P. Yoo and N.J. Wahl, "Software testing: a machine learning experiment", *Proceedings of the 1995 ACM 23rd Annual Conference on Computer Science*, Nashville TN, 1995, pp. 135-141.

[9] G. Rothermel *et al*., "On Test Suite Composition and Cost-Effective Regression Testing", *ACM Transactions on Software Engineering and Methodology, vol.13, no.3*, July 2004, pp 277-331.

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

[11] C.C. Michael, G. McGraw and M.A. Schatz, "Generating Software Test Data by Evolution", *IEEE Transactions on Software Engineering, vol.27 no.12*, December 2001, pp.1085-1110.

[12] D.J. Newman, S. Hettich, C.L. Blake and C.J. Merz, UCI *Repository of machine learning databases*, University of California, Department of Information and Computer Science, Irvine CA, 1998.

[13] 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.

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

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

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