A Framework for Quality Assurance 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 and debug such ML software, because there is no reliable test oracle. We describe a framework and collection of tools aimed to assist with this problem. We present our findings from using the testing framework with three implementations of an ML ranking algorithm (all of which had bugs).

1. Introduction

We investigate the problem of making machine learning (ML) applications dependable, focusing on software quality assurance. Conventional software engineering processes and tools do not always neatly apply: in particular, it is challenging to detect subtle errors, faults, defects or anomalies (henceforth "bugs") in those ML applications where there is no reliable test "oracle". The general class of software systems with no reliable test oracle available is sometimes known as "non-testable programs" [1].

We are specifically concerned with ML applications addressing *ranking* problems, as opposed to the perhaps better-known *classification* problems. When such applications are applied to real-world data (or, for that matter, to "fake" data), there is typically no easy way to determine whether or not the program's output is "correct" for the input. In general, there are two phases to "supervised" machine learning – the first where a training data set with known positive or negative labels is analyzed, and the second where the results of that analysis (the "model") are applied to another data set where the labels are unknown; the output of the latter is a ranking, where when the labels become known, it is intended that those with a positive label should appear as close to the top of the ranking as possible given the information known when ranked. (More accurately, labels are non-negative numeric values, and ideally the highest valued labels are at or near the top of the ranking, with the lowest valued labels at or near the bottom.) Formal proofs of an ML ranking algorithm's optimal accuracy do not guarantee that an application implements or uses the algorithm appropriately, and thus software testing is needed.

In this paper, we describe a framework supporting testing and debugging of supervised ML applications that implement ranking algorithms. The current version of the framework consists of a collection of modules targeted to several ML implementations of interest, including a test data set generator; tools to compare the output models and rankings; several trace options inserted into the ML implementations; and utilities to help analyze the traces to aid in debugging.

We present our findings to date from a case study concerning the Martingale Boosting algorithm, which was developed by Long and Servedio [2] initially as a classification algorithm and then adapted by Long and others [3] into a ranking algorithm. "MartiRank" was a nice initial target for our framework since the algorithm is relatively simple and there were already three distinct, actively maintained implementations developed by different groups of programmers.

2. Background

2.1. Machine learning applications

Previous and ongoing work at the Center for Computational Learning Systems (CCLS) has focused on the development of ML applications like the system illustrated in Figure 1 [3]. The goal of that system, commissioned by Consolidated Edison Company of

New York, is to rank the electrical distribution feeders most susceptible to impending failure with sufficient accuracy so that timely preventive maintenance can be taken on the right feeders at the right time. The prospective users would like to reduce feeder failure rates in the most cost effective manner possible. Scheduled maintenance avoids risk, as work is done when loads are low, so the feeders to which load is shifted continue to operate well within their limits. Targeting preventive maintenance to the most at-risk feeders (those at or near the top of the ranking) offers huge potential benefits. In addition, being able to predict incipient failures in close to real-time can enable crews and operators to take short-term preventative actions (e.g., shifting load to other, less loaded feeders). However, the ML application must be quite dependable for an organization to trust its results sufficiently to thusly deploy expensive resources.

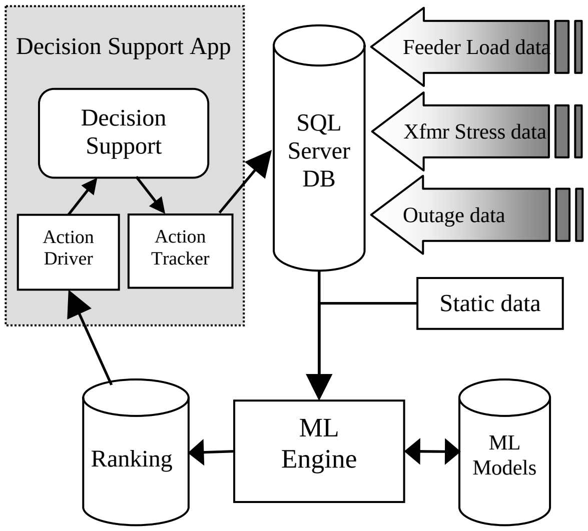

Figure 1. Incoming dynamic data is stored in the main database. The ML Engine combines this with static data to generate and update models, and then uses these models to create rankings, which can be displayed via the decision support app. Any actions taken as a result are tracked and stored in the database.

Other ML algorithms have also been investigated, such as Support Vector Machines (SVMs) [4] and linear regression, as the basis for the ML Engine of the example system and other analogous applications. However, much of the CCLS research has focused on MartiRank because, in addition to producing good results, the models it generates are relatively easy to understand and sometimes "actionable". That is, it is clear which attributes from the input data most contributed to the model and thus the output ranking. In some cases the values of those attributes might then be closely monitored and/or externally adjusted.

This example ML application is presented elsewhere [3]. The purpose of this paper is to present the framework we developed for testing and debugging such applications, with the goal of making them more dependable. The framework is written in Python on Linux. Our initial results reported here focus on the MartiRank implementations.

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 those rules to rank the training data set, rather than a different data set).

2.2. MartiRank algorithm

The algorithm is shown in Figure 2 [3]. The pseudo-code presents it as applied to feeder failures, where the label indicates the number of failures (zero meaning the feeder never failed); however, the algorithm could be applied to any attribute-value data set labeled with non-negative values. In each round of MartiRank, 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 failures). For each sub-list, MartiRank sorts that segment by each attribute, ascending and descending, and chooses the attribute that gives the best "quality". For quality comparisons, the implementations all use a slight variant, adapted to ranking rather than classification, of the Area Under the receiver operating characteristic Curve (AUC) [5]. The AUC is a conventional quality metric employed in the ML community: 1.0 is the best possible, 0.0 is the worst possible, and 0.5 is random.

In each round, the definition of each segment thus has three facets: the percentage of the examples from the original data set that are in the segment, the attribute on which to sort them, and the direction (ascending or descending) of the sort. In the model that is generated, the N^{th} round appears on the N^{th} line of a plain-text file, with the segments separated by semicolons and the segment attributes separated by commas. For instance:

```
0.4000,32,a;0.6500,12,d;1.0000,nop
```

might appear on the third line of the model file, representing the third round. This means that the first segment contains 40% of the examples in the data set and sorts them on attribute 32, ascending. The second segment contains the next 25% (65 minus 40) and sorts them on attribute 12, descending. The last segment contains the rest of the attributes and does a "NOP" (no-op), i.e., does not sort them again because the order resulting from the previous round had the best quality compared to re-sorting on any attribute.

This model could then be re-applied to the training data (called "validation" in ML terminology) or applied to another, previously-unseen set of data (called the "testing data"). In either case, the output is a ranking of the data set examples and the overall quality of the entire ranked list can be calculated.

```
inputs: list $L$ of attribute-value descriptions of feeders
with associated nr. of failures; nr of boosting rounds $T$
output: marti-model $M$

1. let M be the empty model
2. for each round t=$1,..,T$ do:

     - partition $L$ into t sub-lists $L_1$, .., $L_t$ s.t. each $L_j$ has
       same nr. of failures; let $th_2$, .., $th_t$ be the location
       of the splits in terms of the normalized fraction
         of feeders that fall above the split.

     - for each sub-list $i=1,..,$t do:

           i.  compute quality of $L_i$ sort
          ii.  for each attribute $A$ do:
               1. sort $L_i$ according to $A$ in ascending
                  order, compute quality of resulting sort
               2. sort $L_i$ according to $A$ in descending
                  order, compute quality of resulting sort

     - if there exists attribute A and polarity P that
       improves $L_i$'s sort, then:
           i.  if $i > 1$, add $th_i$ to M at level t, position i
          ii.  add A to M at level t, position i.
         iii.  sort $L_i$ according to (A,P)
     - else:
           i.  if $i > 1$, add $th_i$ to M at level t, position i
          ii.  add "NOP" to M at level t, position i.
3.         output M
```

Figure 2: MartiRank Algorithm.

2.3. MartiRank implementations

The first of the three implementations was written in Perl, hereafter referred to as PerlMarti, as a straightforward implementation of the algorithm that included no optimizations. However, when applied to large data sets, e.g., thousands of examples with hundreds of attributes, PerlMarti is rather slow.

A C version, hereafter CMarti, was written to improve performance (speed). CMarti also introduced some experimental options to try to improve quality.

Another implementation also written in C, called FastCMarti, was designed to minimize the costly overhead of repeatedly sorting the attribute values. It sorted the full data set on each attribute at the beginning of an execution, before the first round, and remembered the results; it also used a faster sorting algorithm than CMarti (hence the name FastCMarti). This implementation also introduced some different experimental options from those in CMarti.

2.4. Data sets

The MartiRank algorithm is based on sorting, with the implicit assumption that the sorted values are numerical. 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 than something made by "General Electric" just because of their alphabetical ordering. Thus the implementations expect that all input data will be numerical.

Though much of the real-world data of interest (from the system of Figure 1) indeed consists of numerical values – including floating point decimals, dates and integers – some of the data is instead categorical. Categorical data refers to attributes in which there are K different distinct values (typically alphanumeric as in the manufacturer example), 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.

Some attributes in the real-world data sets need to be removed or ignored, for instance, because the values consist of free-text comments. Generally, these cannot be converted to values that can be meaningfully sorted.

2.5. Related work

Although there has been much work that applies machine learning techniques to software engineering and software testing [6, 7], there seems to be very little work in the reverse sense: applying software testing techniques to machine learning software, particularly those ML applications that have no reliable test oracle. Our framework builds upon Davis and Weyuker's [8] approach to testing with a "pseudo-oracle" (comparing against another implementation of the specification), but most aspects of our framework are still useful even when there is just one implementation.

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" ML data sets have been collected (e.g., the UCI Machine Learning Repository [12]) for the purpose of comparing result quality, but not for testing in the software engineering sense.

Orange [13] and Weka [14] are two of the several frameworks that aid in developing ML applications, but the testing functionality they provide is again focused on comparing the quality of the results, not the "correctness" or dependability of the implementations.

3. Testing Approach

3.1. Optimization options

CMarti and FastCMarti provide runtime options that turn on/off "optimizations" intended to improve result quality. These generally involve randomization (probabilistic decisions), yet it is challenging to evaluate test results when the outputs are not deterministic. Therefore, these options were disabled for all testing thus far: Our goal in comparing these implementations was not to get *better* results but to get *consistent* results.

We initially believed that PerlMarti was a potential "gold standard" because it was truest to the algorithm as well as originally coded by the algorithm's inventor, but as we shall see we found bugs in it, too. However, the fact that we had three implementations of MartiRank coded by different programmers helped immensely: we could generally assume that – with all options turned off – if two implementations agreed and the third did not, the third one was probably "wrong" (or, at least, we would know that something was amiss in at least one of them).

3.2. Types of testing

We focused on two types of testing: comparison testing to see if all three implementations produced the same results, and regression testing to compare new revisions of a given implementation to previous ones (after bug fixes, refactorings, and enhancements to the optimization options).

The data sets for some test cases were manually constructed, e.g., so that a hand-simulation of the MartiRank algorithm produced a "perfect" ranking, with all the positive examples (feeder failures) at the top and all the negative examples (non-failures) at the bottom. These data sets were very small, e.g., 10 examples each with 3 attributes.

We also needed large data sets, to exercise a reasonable number of MartiRank rounds (the implementation default is 10) with still sufficiently many examples in each segment in the later rounds. We tested with some (large) real-world data sets, which generally have many categorical attributes, many repeating numerical values, and many missing values. However, in order to have more control over the test cases, e.g., to focus on boundary conditions from the identified equivalence classes, most of our large data sets were automatically generated with F failures (positive-labeled examples), N numerical attributes and K categorical attributes. F is any percentage between 0 and 100. The N numerical attributes were specified as including or not including any repeating values, with 0 to 100 percent missing values; the sets of values for each attribute were independent. For each of the K categorical attributes, the number of distinct values and the percent per category and missing were specified.

3.3. Models versus rankings

Our evaluation of test outputs focused primarily on the models, as it is virtually always the case that if two versions produce two different models, then the rankings will also be different: if different models do produce the same rankings, that is likely by chance (i.e., an effect of the data set itself and not the model) and does not mean that the versions were producing "consistent" results. However, even when two implementations or revisions generate the same model, we cannot assume that the rankings will be the same: CMarti and PerlMarti generate rankings via programs that are separate from the code used to generate the models, so it is possible that differences could exist.

FastCMarti does not follow the typical supervised ML convention in which a training data set is used to generate a model and then that model is given a separate "testing" data set with unknown labels to rank. Instead, the two data sets are joined together and each example marked accordingly. FastCMarti runs on the combined data set, but only the training data are used to create the model. The testing data are sorted and segmented along with the training data, and the final ranking of the testing data is the output – the model itself is merely a side effect that we needed to extract in order to compare across versions.

4. Testing Framework

4.1. Generating data sets

We 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 covered those classes, focusing on boundaries. The parameters include the number of examples, the number of attributes, and the names of the output test data set files (which were produced in different formats for the different implementations).

The data generation tool can be run with a flag that ensures that no 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 this is not guaranteed.

The utility is also given the percentage of failures to include in the data set. For all test cases discussed in this paper, each example could only have a label of 1 (indicating a failure) or 0 (non-failure). Similarly, a parameter specifies the percentage of missing values. Note that the label value is *never* missing.

Lastly, 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 percent of unknown values. The utility also allows for having multiple categorical attributes, or for having none at all.

4.3. Comparing models

We created a utility that compares the models and reports on the differences in each round: 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.

4.4. Comparing rankings

As explained above, we cannot simply assume that the same models will produce the same rankings for different implementations or revisions. This utility reports some basic metrics, such as the quality (AUC) for each ranking, the number of differences between the rankings (elements ranked differently), the Manhattan distance (sum of the absolute values of the differences in the rankings), and the Euclidean distance (in N-dimensional space). Another metric given is the normalized Spearman Footrule Distance, which attempts to explain how similar the rankings are (1 means that they are exactly the same, 0 means they are completely in the opposite order) [15]. Some of these metrics have mostly been useful when testing the "optimization" options, outside the scope of this paper.

4.5. 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. We added runtime options to each implementation to report significant intermittent values that arise during the algorithm's execution, specifically 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 is extremely useful in debugging differences in the models and rankings, as it allows 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).

5. Findings

5.1. Testing with real-world data

We first ran tests with some real-world data on all three implementations. Those data sets contained categorical data and both missing and repeating values. Our hope was that, with all "optimizations" disabled, the three implementations would output identical models and rankings.

Not only did PerlMarti and FastCMarti produce different models, but CMarti reproducibly gave seg faults. Using the tracing utilities for the CMarti case, we found that some code that was only required for one of the optimization 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 seg faults disappeared. However, the model then created by CMarti was still different from those created by either of the other two.

These tests demonstrated the need for "fake" (controlled) data sets, to explore the equivalence classes of non-repeating vs. repeating values, none-missing vs. missing values, and non-categorical vs. categorical attributes (which are necessarily repeating).

5.2. Simple comparison testing

We hand-crafted data sets (i.e., we did not yet use the framework to generate data sets) to see whether the implementations would give the same models in cases where a "perfect" ranking was possible. That is, we constructed data sets so that a manually-simulated sequence of sorting the segments (i.e., model) led to a ranking in which all of the failures were at the top and all the non-failures were at the bottom. It was agreed by the CCLS machine learning researchers that any implementation of MartiRank should be able to find such a "correct" model. And they generally did.

In one of the "perfect" ranking tests, however, the implementations produced different results because the data set was already ordered as if sorted on the attribute that MartiRank would choose in the first round. In the reported models, CMarti sorted anyway, but PerlMarti and FastCMarti did NOPs because leaving the data as-is would yield the same quality (AUC).

After consulting with the CCLS ML researchers, we "fixed" PerlMarti and FastCMarti so that they would *always* choose an attribute to sort on in the first round, i.e., never select NOP in the first round. The rationale was that one could not expect that the initial ordering of a real-world data set would happen to produce the best ranking in the first round, and any case in which the data are already ordered in a way that yields the "best" quality is likely just a matter of luck – so sorting is always preferable to not sorting. However, the MartiRank algorithm as defined in Figure 2 does not treat the first round specially, so the implementations now thus deviate from the algorithm.

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 implementations 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 (rightmost), as specified in MartiRank.

This led to the interesting discovery that FastCMarti was doing the segmentation (sub-list splits) differently from PerlMarti and CMarti. By using the framework's model analysis tool, we found that even when FastCMarti was choosing the same attribute to sort on as the other implementations, in the subsequent round the percentage of the data set in each segment could sometimes be different.

It appeared (and we confirmed using the tracing analysis tool) that the difference was that FastCMarti 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, CMarti and PerlMarti took only enough failures to fill the segment and stopped there. For example, if the sequence of labels were:

1 1 0 0 1 0 0 1 0 0

and we were in the second round (two segments, each having ½ of the failures), then CMarti and PerlMarti would create segments like this:

1 1 | 0 0 1 0 0 1 0 0

but FastCMarti would create segments like this:

1 1 0 0 | 1 0 0 1 0 0

Both are "correct" because the algorithm merely says that, in the Nth round, each segment should contain 1/Nth 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 is the first instance we found in which the MartiRank algorithm did not address an implementation-specific issue, which does not matter with respect to formal proofs, but does matter with respect to consistent testing.

Once these issues were addressed, we repeated all the small test cases as well as with larger generated data sets, both for regression testing purposes (to ensure that the fixes did not introduce any new bugs) and for comparison testing (to ensure that all three implementations produced the same models).

5.3. Comparison testing with repeating values

The next tests we performed with repeating values, that is, the same value could appear for a given attribute for different examples (in the real-world data sets, voltage level and activation date attributes involve many repeating values). We again started with small hand-crafted data sets that allowed us to judge the behavior by inspection. In one test, PerlMarti and CMarti found a "perfect" ranking after two rounds, but FastCMarti did not find one at all. In another test, PerlMarti/CMarti vs. FastCMarti showed different segmentations in a particular round.

Then by using larger, automatically generated data sets, we confirmed our intuition that the CMarti and PerlMarti sorting routines were "stable" (i.e., they maintain the relative order of the examples from the previous round when the values are the same), whereas FastCMarti 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"). Again, the algorithm did not address a specific implementation issue – which sorting approach to use – and different implementation decisions led to different results.

After replacing FastCMarti's sorting routine with a stable sort, we noticed that – again in an effort to be "fast" – the resulting list from the descending sort was simply the reverse of the list from the ascending sort, which does not retain the stability. For instance, if the stable ascending sort returned examples in this order:

1 2 A B 5 6

where A and B have the same values, then the stable descending sort should be:

6 5 A B 2 1

But FastCMarti was simply taking the reverse of the ascending list to produce:

6 5 B A 2 1

This code was "fixed". This modification necessarily had an adverse effect on runtime, but provided the consistency we sought.

5.4. Comparison testing of rankings

Previously we had only compared the models. Now for the cases where the models were the same, we wanted to check whether the rankings were also identical. For CMarti and PerlMarti, ranking generation involved a separate program that we had not yet tested.

We used the testing framework to create new large data sets with repeating values and used the analysis tool to analyze the rankings (at this point, all three implementations were producing the same models). CMarti and PerlMarti agreed on the rankings, but FastCMarti did not. The framework allowed us to determine *how* different, based on the various metrics such as normalized Spearman Footrule Distance and AUCs, as well as to determine *why* they were different, using the trace analysis tool.

Using the tracing utility to see how the examples were being ordered during each sorting round, we found that the "stability" in FastCMarti 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, CMarti and PerlMarti would leave those examples in their relative order as they stood at the end of the previous round, but FastCMarti would leave them in the relative order as they stood in the original data set. FastCMarti was designed this way to make it faster, i.e., by "remembering" the sort order for each attribute at the very beginning of the execution, and not having to re-sort in each round.

For instance, a data set with entries A and B such that A appears in the set before B would look like:

....A....B....

If in the first round MartiRank sorts on some attribute such that B gets placed in front of A, the ordering would then look like:

....B....A....

In the second round, if the examples are in the same segment and MartiRank sorts on some attribute that has the same value for those two examples, PerlMarti and CMarti would then end up like this:

......BA......

because B was before A at the end of round 1. However, FastCMarti would do this:

......AB......

because A was before B in the *original* data set.

Since this was not explicitly addressed in the MartiRank algorithm, we contacted Long and Servedio, who agreed 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. Fixing this problem will require rethinking the entire approach to "fastness", which has not yet occurred; thus all further comparison testing omitted FastCMarti.

5.5. Comparison testing with sparse data sets

Once PerlMarti and CMarti were producing the same models for the cases with repeating values, we began to test data sets that had missing values. We used the framework to create large, randomly-generated (but non-repeating) data sets with percent of missing values as a parameter (0.5%, 1%, 5%, 10%, 20%, and 50%).

In these tests, both implementations were initially generating different models, 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 CCLS 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 representing 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 of the list) were considered, but this was deemed to be most in the MartiRank spirit (as in the ordering reuse case above).

Using the tracing outputs from the implementations and analyzing them with the framework tools, we noticed that CMarti – even with all optimizations turned off – was still performing randomizations in the case of missing values. In particular, it kept the missing values in the same relative order but placed them randomly throughout the list. So we "fixed" the code so by default all missing values would stay in their original locations in the same relative order.

We also used the framework tools to find that PerlMarti was not only behaving incorrectly with respect to the placement and order of missing values, but also that the missing values were causing the *known* values to be sorted incorrectly. 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. This was "fixed" to also leave missing values in their positions and sort known values "around" them.

5.6. Comparison testing with categorical data

Because categorical data provides a combination of necessarily repeating (all 0s or 1s) and sometimes missing values, we created test data sets with categorical attributes to see what would happen when all of these different criteria came together. Though CMarti and PerlMarti produced the same models and rankings in most of these test cases, in one particular test, the models were different. After seeing them agree for so many other test cases, our intuition was that something might be different in the calculation of the AUC – which had recently been refactored in the Perl implementation. We used the tracing utility to discover that PerlMarti indeed had a bug introduced (during refactoring) by the incorrect use of a global variable in the calculation of the AUC. After fixing the bug, we ran regression tests and the CMarti and PerlMarti models were the same in all cases.

5.7. Testing with real-world data revisited

We reconsidered the real-world data from the original comparison tests, now running tests only for CMarti and PerlMarti. These two implementations produced the same models and the same rankings. They now have the same behavior in all our current test cases with missing values, repeating values, and categorical data – although of course we cannot rule out further bugs (and indeed we have found other bugs in all three implementations besides those discussed here).

6. Evaluation of the framework

6.1. Usefulness in our testing

The testing framework facilitated our work by aiding us in the creation, execution and analysis of the test cases. The ability to control the properties of data sets 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 to culling real-world data.

The model comparison tool provided many advantages over "diff" because it enumerates the differences clearly and is aware of the various facets of a MartiRank round/segment (number of examples, sort attribute, and direction). The ranking comparison tool was admittedly not much more useful than "diff" for the testing presented here, but in our preliminary work not reported here, the comparison metrics have already been very important aids in judging whether the CMarti and FastCMarti "optimization" options are, in fact, improving result quality.

Finally, the trace analysis tool was tremendously useful in determining where differences in models and ranking order were coming from. It provided great insights into the internals of the implementations, aiding us in narrowing down other flaws as well. Most significantly, it was the trace analyses that enabled us to convince the implementations' developers that there were, in fact, bugs in their code – since, as noted in the Introduction, there is no easy way to inspect the output to determine if it is indeed "correct" for the input.

6.2. Applicability to other ML algorithms

Our testing process was greatly aided by the fact that we had three implementations of the same algorithm, effectively acting as "pseudo-oracles" for each other [8], but the framework is still useful even when there is just one implementation of an ML ranking algorithm. The framework is also applicable to supervised ML classification algorithms. For instance, a classification simply deciding, say, failure-prone vs. not-failure-prone would also fall into Davis and Weyuker's class of "*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"* [8].

When there is only one implementation, the framework would support regression testing across revisions of that implementation to ensure that no bugs have inadvertently been introduced: equivalence class and boundary condition data sets can be generated, tests can be run, outputs can be compared between previous and latest revisions, and if need be execution traces can be analyzed.

To re-target our *conceptual* framework to other ML algorithms beyond MartiRank, some code changes would certainly be necessary. The data generation tool might or might not work "as is": this depends on the input data format required by the implementation. The data generator already supports plug-replaceable modules for creating data set files in whatever format is needed. Two such modules are currently implemented, one for PerlMarti and CMarti (csv files) and the other for FastCMarti (a "sparse" attribute-value pair representation that enables more compact representation of data sets with a high proportion of missing values). All the ML algorithms of interest work with example data points, each consisting of a series of attributes and one non-negative numeric label.

The model comparison/analysis tool is specific to MartiRank's model format – which had to be explicitly extracted by modifying FastCMarti since there the model was originally employed only internally, not output. But the concept of "model" is inherent to any ML ranking or classification algorithm intended to be trained on one data set and applied to another, even when semi-merged as in FastCMarti.

Rankings tend to occur in only one of two basic formats: the ranked list of examples, in order, and the examples in their original input order annotated with their rank. The ranking tool already converts from the latter, the actual output of all three implementations, to the former, to make it more human-readable. We have briefly investigated applying the framework to RankBoost [16], and do not see any significant re-targeting problems.

Furthermore, for comparing rankings to be used in an ML application like the system of Figure 1, where it would likely be feasible to take action on only a small number of elements presumably selected from the top (or possibly the bottom) of a given ranking, the ranking analysis tool already includes some special facilities that look at the top and the bottom. For a parameterized value X, the utility calculates the quality (currently AUC) of only the top and bottom X% of each ranking, and also calculates the "correspondence" between the top and bottom X% of both rankings. Correspondence is simply the number of examples that appear in the top (or bottom) X% of *both* rankings, divided by the number of examples in the top (or bottom) X%. These metrics, along with the other distance metrics described previously, can help decide whether a pair of rankings is "similar" in the ranges that are most important, or if one implementation – with or without optimization options – is notably better than another. This is also useful for comparing *across* algorithms. However, here the framework can only tell us whether the result quality is *better*, but not that the code is *correct*.

The current trace output and analysis is heavily dependent on MartiRank's notion of rounds and segments (sub-lists), and would necessarily have to be re-targeted to the specific algorithm implementation.

7. Future work and next steps

7.1. Improvements to the testing framework

The framework 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 construct a model that produces a "perfect" ranking, i.e., where there is a clearcut "correct" output. But this may be impossible in the general case. 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. Here we could build upon the related work of [17, 18].

7.2. Expansion to complete ML applications

Our research to date has not yet addressed the use of ML algorithm implementations in the context of overall *applications*, e.g., as depicted in Figure 1. 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 framework to encompass the entire system, including for instance the decision support treatment of ranking (or classification) results. The emerging generation of ML applications could take various implementations of MartiRank, SVMs 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. We plan to investigate how to extend the framework to test this kind of software. Our ultimate goal is to make our current and later expanded framework useful outside CCLS, particularly to other ML researchers who rarely cross paths with the software engineering community.

8. Conclusion

We have presented a conceptual framework for testing (and debugging) a particular class of algorithm implementations for which there is no reliable test oracle. Particularly in machine learning applications, where there is often no precise input/output specification, it can be very difficult to determine the "right" answer. But the framework makes it relatively straightforward to conduct regression and comparison testing, especially in the cases where there are multiple implementations of the same algorithm. As we have discussed, though, the framework can still be useful even when there is just one implementation.

9. Acknowledgements

Numerous people at CCLS contributed directly or indirectly to this effort. We would particularly like to thank Wei Chu, Philip Gross and David Waltz for their assistance and encouragement. Murphy and Kaiser are members of the Programming Systems Lab, funded in part by NSF CNS-0426623 and EIA-0202063, NIH 1 U54 CA121852-01A1, and are also affiliated with CCLS. Arias is fully supported by CCLS, with funding in part by Consolidated Edison Company of New York.

10. References

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

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

[3] 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.

[4] Cristianini, N., and J. Shawe-Taylor, *An Introduction to Support Vector Machines and other kernel-based learning methods.* Cambridge University Press, 2000.

[5] 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.

[6] T.J. Cheatham, J.P. Yoo, 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.

[7] Z. Li and Y. Zhou, "PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code". *Proceedings of the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Lisbon, Portugal, Sept 2005, pp. 306-315.

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

[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, 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 [http://www.ics.uci.edu/~mlearn/MLRepository.html], 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] Witten, I.H. and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, *2nd Edition*, Morgan Kaufmann, San Francisco, 2005.

[15] C. Spearman, "Footrule for Measuring Correlation", *British Journal of Psychology* vol.2, pp.89-108, June 1906.

[16] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer, "An efficient boosting algorithm for combining preferences", *Journal of Machine Learning Research vol.4*, Nov. 2003, pp.933–969.

[17] E. Walton, "Data Generation for Machine Learning Techniques", [http://www.cs.bris.ac.uk/Teaching/Resources/COMS30500/ExampleTheses/thesis6.pdf], University of Bristol, 2001.

[18] H. Christiansen and C.M. Dahmke, "A Machine Learning Approach to Test Data Generation: A Case Study in Evaluation of Gene Finders", [http://diggy.ruc.dk:8080/bitstream/1800/1899/2/artikel.pdf], Roskilde University, Roskilde, Denmark.