Evaluating and Tuning Predictors for Execution Classification

Charles Song
Department of Computer Science
University of Maryland
College Park, Maryland USA
csfalcon@cs.umd.edu

Abstract

Automated classification of executions is essential to a successful Remote Analysis and Measurement of Software System (RAMSS). However, most automated classification techniques require large amount of data to be collected, which cause the RAMSS to have a large performance overhead. A technique called Association Tree addresses this overhead issue, we extend on this work by constructing and evaluating predictors to improve the performance of Association Tree. Our results show a few promising predictors and a technique that produces predictors that can classify more accurately with less collected data.

1 Introduction

Software systems are rarely deployed in the same environments they are developed and tested in. The actual deployment environments can differ greatly from the expected environments, thus the behavior of the software systems can differ greatly from the expected behavior. Recently, there has been increasing interest in Remote Analysis and Measurement of Software System (RAMSS) [4], [5], [8], [9], [11]. Generally, RAMSS instrument numerous instances of the software system and distribute the instrumented instance to remote users. As the instances run, they collect execution data and send them back to the developers for analysis. These in-field execution data can provide better understanding of the behavior of the software system in the deployment environment.

RAMSS typically collect different kinds of data and use different analysis to enable the developers to monitor and model software behavior in the field. However, to effectively use these in-field data, automated classification of program executions is essential to most of the RAMSS techniques. For example, when a developer is trying to ping point a bug using information from the remote environment, knowing whether the executions are successful or failed is essential. Machine learning techniques can be used to model software behavior that corresponds to different execution outcomes; execution outcomes such as successful, crashed or not responding.

Most existing techniques suffer from one major problem; they impose significant overheads on the software systems. These overheads include system slowdown due to code instrumentation and bandwidth occupation due to data collection. In Haran et als work [10], their novel classification technique Association Tree greatly reduce the amount of execution data needed to be collected in the remote environment while maintaining the execution classification accuracy.

This paper extends Haran et als work in two ways. First, we apply their techniques on a Graphical User Interface (GUI) application. Then, we improve their techniques by evaluating and tuning different execution classification predictors. In Section 2, we describe how Association Trees are constructed. Section 3 describes the software application and data we used in our experiments. In Section 4, we present experiments we conducted to evaluate and tune Association Tree performance. In Section 5, we discuss future work to further improve our approaches. And Section 6 presents our conclusions.

2 Background

In Haran et al, they found that from a large (thousands) pool of potential predictors only a very small fraction (less then one percent) were important for the classification. This finding implies only small fraction of the software system would need to be instrumented, thus reduce both runtime overhead and data collection overhead. However, the data set collected with such small amount of instrumentation would greatly reduce the performance of tree-based classification techniques. To solve this problem the Association Tree classification was developed.

The training set for the Association Tree is still a set of labeled executions. Each data vector has one column for

each potential predictor. If a predictor is collected during a given run, its value is recorded in the vector. Otherwise, a special value (NA) is recorded. The output of the Association Tree is a model that predicts the outcome of an execution based on the predictor vector. The models will predict pass, fail and unknown, where unknown means the model is unable to make a prediction due to lack of data.

The Association Tree is built in three stages:

1. transform the predictors into items that are present or absent.
2. find association rules from these items.
3. construct a classification model based on these rules.

In the first stage, the algorithm first screens each predictor and checks that the Spearman rank correlation between the predictor and the observed outcomes exceeds a minimum threshold (Spearmans rank correlation is similar to the traditional Pearsons correlation). The goal of this step is to discard predictors whose values are not correlated with outcomes and, thus, are unlikely to be relevant for the classification. Next, the algorithm splits the distribution of each remaining predictor into two parts. Ideally, after the split, all executions with one outcome would have values above the split, while all runs with the other outcome would have values below it. Neither item is present if the corresponding predictor was not being measured during the run.

After the algorithm completes the first stage, the original set of training data has been transformed into a set of observations, one per run, where each observation is the set of items considered present for that run. The goal of the second stage of the algorithm is to determine which groups of frequently occurring items are strongly associated with each outcome. To achieve this, the well-known a priori data-mining algorithm is used. The a priori algorithm is used to efficiently find which sets of items are correlated with successful executions and failure executions. These sets of items, together with their correlations with outcomes are called association rules.

The final stage of the algorithm performs outcome prediction using the association rules produced in the previous stage. Given a new run, the algorithm finds the rules that apply to it. If all applicable rules give the same outcome, it returns that outcome as the prediction. If there is disagreement or no applicable rules exist, the algorithm returns a prediction of unknown.

3 Experimental Subject and Data

We used the same subject software application for all of our experiments to create a consistent environment to understand our proposed approaches. The executions of the subject software application are labels pass or fail. In this section we introduce our experimental subject and data.

3.1 Experimental Subject

The software application we chose for our experiments was CrosswordSage-0.3.5 [1], an open source Java GUI crossword puzzle creation program. CrosswordSage-0.3.5 contains 1664 lines of executable codes, 34 classes and 244 methods. This version of CrosswordSage contains 4 real faults detected during our data collection phase. We picked this software application because it contains roughly same amount of GUI and logic code. In contrast, JABA (Java Architecture for Bytecode Analysis) used in Haran et al contained no GUI code.

3.2 Execution Data and Labels

To generate a training set for the CrosswordSage classification, we used two Java instrumentation tools:

- instr [2]. a source-to-source translation tool that instrument a Java program to collect the number of times each line of code was executed in a run.
- JRat [3]. a Java byte code modification tool that instruments a Java program to collect runtime information at method level.

After the instrumentation with these two tools, we are able to collect three types of runtime information: method entry count, average method runtime and source line execution count.

Since CrosswordSage is a GUI application, we used GUITAR [12], a Java GUI application test tool, to generate GUI test cases and drive application testing. GUITAR inspects the Java GUI application and creates an Event Flow Graph (EFG) of the GUI. From the EFG, test cases composed of Java Swing events are generated to simulate user mouse clicks and keystrokes. We were able to generate 2352 valid GUI test cases for CrosswordSage with GUITAR. With these test cases we achieved 50 percent statement coverage.

We labeled the execution outcomes according to whether any of the four known faults appeared in the execution of a test case. When any of the four faults appear, a Java Exception would be thrown; unlike non-GUI applications, a Java Exception does not constitute halting of the application. Out of 2352 test cases, 1376 were labeled pass and 976 were labeled fail.

4 Experiments and Results

To assess the performance of the Association Tree, we examine two performance measures:

Table 1. Table showing the performance of Association Tree using different simple predictors. Method entry count and average method runtime offered acceptable performance. Source line execution count experienced higher error rate due to large number of potential predictors.

PREDICTOR	%CLASSIFIED	%ERROR	#PREDICTORS
METHOD	68.73	3.03	111
RUNTIME	65.18	4.25	111
SOURCE	65.89	16.49	1664

- Classification rate. Percentage of executions for which the constructed model predicts either pass or fail. If the model is able to classify an execution due to lack of data then the classification rate would be lowered.
- Classification error rate. Percentage of runs whose outcomes were incorrectly predicted.

In Haran et al, the Association Tree was able to classify multiple versions of JABA with high classification rate and accuracy. On average, classification rate of JABA was 63% and Classification error rate was 2%. The predictor type in these classifications was method entry count.

In our experiments, we first examine the performance of the Association Tree on GUI applications by using simple predictors; method entry count, average method runtime and source line execution count. The method entry count predictor would serve as a direct comparison to the results of Haran et al. Next we seek to reduce the total number of potential predictors by observing unique profiles of the simple predictors. And finally, we construct more complex predictors by combining simple predictors with other software application information.

4.1 Simple Predictors

The method entry count is the same predictor used to test Association Tree in Haran et al. The results from this classification test is a good indicator how well Association Tree classifies GUI application executions. In Table 1, method entry count achieved a classification rate of 68.73% and classification error rate of 3.03%. This result shows that Association Tree can successfully classify GUI application as well.

The average method runtime predictor achieved similar results as method entry count; 65.18% classification rate and 4.24% classification error rate.

The source line execution count results are worse than the other two predictor's results, especially in classification error rate. We believe this high error rate caused by the much larger number of potential predictors; this suggests that source line level information make poor predictors on its own.

Table 2. Table showing the performance of Association Tree using unique profile predictors. Comparing to the source line execution count predictor, the classification rate improved and classification error rate decreased drastically. Also, the potential number of predictored dropped from 1664 to 137.

PREDICTOR	%CLASSIFIED	%ERROR	#PREDICTORS
PROFILE	88.76	2.98	137

4.2 Unique Profile Predictors

We observed that many of the predictors shared the same profile across all test case executions. Figure 1 is a table of 5 test cases with 7 potential predictors. Predictor 1, 5 and 7 were executed same number of times in all test cases, thus these three predictors have the same profile. If we can group predictors with the same profiles together to form a single predictor, then we can reduce the total number of predictors in this example from 7 to 4. With this unique profile reduction technique, we can reduce the amount of instrumentation by only enable one predictor from each group of predictors.

We used this technique to reduce the source line execution count data. The total number of potential predictors reduced from 1664 to 137, a 92% reduction. The most common profile is all zeros across all test cases, meaning all these predictors are lines of code never executed by any of the test cases; nearly half of the predictors fall into this group.

With this unique profile predictor, Association Tree achieved 88.76% classification rate and 2.98% classification error rate (Table 2). The Association Tree performed much better then the simple predictors were used. The improvement is amazing considering it was derived from the source line execution count data.

Our understanding is that unique profile reduction technique put less emphasis (weight) on commonly occurring patterns and more emphasis on less commonly occuring partterns. For example, users use the GUI to save a crossword puzzle often, therefore the same sequence of events would occur often. Logically, the less commonly occurring event patterns are more likely to reveal undiscovered faults in the application.

TEST	P1	P2	P3	P4	P5	P6	P7
T1	8	1	2	1	8	0	8
T2	0	5	3	5	0	0	0
T3	1	0	1	0	1	3	1
T4	3	1	0	1	3	2	3
T5	0	1	0	1	0	1	0

Figure 1. Example showing unique profile reduction. Predictor 1, 5 and 7 share one unique profile. Predictor 2 and 4 share another unique profile.

Table 3. Table showing the performance of Association Tree using complex predictors. The classification rate dropped for both complex predictors, however, classification error rate improved for both. Both are good candidate for unique profile reduction.

PREDICTOR	%CLASSIFIED	%ERROR	#PREDICTORS
COVERAGE	54.28	2.91	245
PERCENT	59.89	1.40	111

4.3 Complex Predictors

In the next experiment, we construct 2 complex predictors. In theory, the complex predictors contain more knowledge about the application than the simple predictors and thus should perform better when used with the Association Tree.

4.3.1 Coverage Predictor

The first complex predictor, coverage predictor, is constructed with the source line execution count and the source code itself. The coverage predictor is constructed in two stages:

1. Statically analyze the source code, extract out every method from the source code and gather information on starting line of the method, ending line of the method and total number of executable source code (excludes comments, curly braces and variable definitions).

2. Based on source line execution data for each test case, determine which lines were executed at least once and which methods they belong to. Then calculate the percent of each method covered by that test case.

The logic behind using this predictor is to build application specific knowledge into the Association Tree classification process. For example, a method usually gets high coverage but gets a low coverage could be a signal of a fault; the reverse could also be true. As expected, the coverage of methods generally falls on the two extremes, very low coverage and high coverage, with majority being less than one percent covered.

In Table 3, coverage predictor achieved 54.28% classification rate and 2.91% classification error rate. This complex predictor did not perform as well as the first two simple predictors, however, it did outperform the source line execution count predictor which it was derived from. Once again, the results show that source line execution count data can be useful when processed (number of potential predictors reduced).

4.3.2 Runtime Percentage Predictor

The second complex predictor, runtime percentage predictor, is constructed with the method entry count and the average method runtime data.

The runtime percentage predictor is constructed by:

1. estimate the total runtime:

$$totalRuntime = \sum_{i=1}^{n} avgRuntime_i * entryCount_i$$

2. calculate each methods runtime percent:

$$runtimePercent_i = \frac{avgRuntime_i * entryCount_i}{totalRuntime}$$

The logic behind use this predictor is similar to that of coverage predictor; a method usually takes a long time to run but finishes quickly could be a signal of a fault. However, these two complex predictors are different:

- runtime percentage predictor is not weight by the size of the method.

- runtime does not always correlate with line of code
- methods called multiple times carries more weight in this predictor.

Only a small fraction of methods get high percentage (usually higher than 75%) of the total runtime, most method runtimes are insignificant to the total runtime of a test case.

In Table 3, runtime percentage predictor achieved 59.89% classification rate and 1.40% classification error rate. This complex predictor did not outperform the first two simple predictors which it was derived from, however, its classification rate was much lower than the simple predictors.

Both of these complex predictors can be greatly reduced in size with the unique profile reduction technique presented earlier. We hypothesize that performance of these predictors can be improved after unique profile reduction, but we leave those experiments to future work.

5 Future Work

The performance of the Association Tree classification can be improved by using bagging [6] to combine models built from different data set. When one model fails to classify an execution, other models could classify it. This would improve the classification rate. If an execution is classified differently by the models, a voting system can be implemented to select the most fitting outcome. This would lower the classification error rate. Das and Chen [7] has implemented a similar system to classify stock market sentiments from financial message board posts, and shown that this approach can improve classification performance.

Our preliminary results suggest that unique profile reduction technique can improve the performance of Association Tree. We plan to examine the performance impact when this technique is applied to other predictors, especially the complex predictors.

CrosswordSage is a good GUI application to test out our concepts for better predictors. Our next step is to apply these concepts and techniques to a larger application with more known and unknown faults.

We want to study how Association Tree can be used to aid the development and testing of rapidly changing software. For example, those developed with the extreme programming teams. These applications could use the learned models from the previous version to improve learning of the next version. How would Association Tree models change with the application?

6 Conclusion

In this paper, we have presented three experiments conducted on the Association Tree classification technique with a Java GUI application. The first experiment used simple predictors to classify execution outcomes, the results showed that Association Tree can be used on GUI applications and predictors other than method entry count can yield similar accuracy. The second experiment was to test the unique profile reduction technique, we demonstrated that this technique can greatly reduce the amount of instrumentation needed by the application instances and still produce better classification results. The third experiment tested performance impact of more complex predictors, results suggested that simply adding application knowledge does not equate to better classification. But these complex predictors appear to be good candidate for the unique profile reduction technique. To conclude, our results showed the Association Tree can be improved to further reduce the overhead problem face by most RAMSS.

References

[1] Crossword sage: Java puzzle creation utility.

[2] instr: Java test coverage and instrumentation toolkits.

[3] Jrat: the java runtime analysis toolkit.

[4] J. Bowring, A. Orso, and M. Harrold. Monitoring deployed software using software tomography. *ACM SIGPLAN/SIGSOFT Workshop Program Analysis Software Tools and Engineering*, pages 2—8, 2002.

[5] J. Bowring, J. Rehg, and M. Harrold. Active learning for automatic classification of software behavior. *International Symposium on Software Testing and Analysis*, pages 195—205, 2004.

[6] L. Breiman. Bagging predictors. *Machine Learning*, 24:123—140, 1996.

[7] S. R. Das and M. Y. Chen. Yahoo! for amazon: Sentiment extraction from small talk on the web. *8th Asia Pacific Finance Association Annual Conference*, 2001.

[8] S. Elbaum and M. Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Transactions of Software Engineering*, pages 312—327, 2005.

[9] P. Francis, D. Leon, M. Minch, and A. Podgurski. Tree-based methods for classifying software failures. *Proceedings of 15th International Symposium on Software Reliablility Engineering*, pages 451—462, 2004.

[10] M. Haran, A. Karr, M. Last, A. Orso, A. Porter, A. Sanil, and S. Fouche. Techniques for classifying executions of deployed software to support software engineering tasks. *IEEE Transactions on Software Engineering*, pages 287—304, 2007.

[11] D. Hilbert and D. Redmiles. Extracting usability information from user interface events. *ACM Computing Surveys*, pages 384—421, 2000.

[12] A. Memon. An event-flow model of gui-based applications for testing. *Software Testing Verification and Reliability*, pages 137—157, 2007.