Applying Classification Techniques to Remotely-Collected Program Execution Data

Murali Haran,[†] Alan Karr,[♯] Alessandro Orso,[‡] Adam Porter,[♭] and Ashish Sanil [♯]

[†] Dept. of Statistics, Penn State University, University Park, PA

[♯] National Institute of Statistical Sciences, Triangle Park, NC

[‡] College of Computing, Georgia Inst. of Technology, Atlanta, GA

[♭] Dept. of Computer Science, University of Maryland, College Park, MD

ABSTRACT

There is an increasing interest in techniques that support measurement and analysis of fielded software systems. One of the main goals of these techniques is to better understand how software actually behaves in the field. In particular, many of these techniques require a way to distinguish, in the field, failing from passing executions. So far, researchers and practitioners have only partially addressed this problem: they have simply assumed that program failure status is either obvious (i.e., the program crashes) or provided by an external source (e.g., the users). In this paper, we propose a technique for automatically classifying execution data, collected in the field, as coming from either passing or failing program runs. (Failing program runs are executions that terminate with a failure, such as a wrong outcome.) We use statistical learning algorithms to build the classification models. Our approach builds the models by analyzing executions performed in a controlled environment (e.g., test cases run in-house) and then uses the models to predict whether execution data produced by a fielded instance were generated by a passing or failing program execution. We also present results from an initial feasibility study, based on multiple versions of a software subject, in which we investigate several issues vital to the applicability of the technique. Finally, we present some lessons learned regarding the interplay between the reliability of classification models and the amount and type of data collected.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging—*Monitors*; G.3 [Mathematics of Computing]: Probability and Statistics;

General Terms: Reliability, Experimentation

Keywords: Machine learning, classification, software behavior

ESEC-FSE'05, September 5–9, 2005, Lisbon, Portugal.

1. INTRODUCTION

Several research efforts are focusing on tools and techniques to support the remote analysis and measurement of software systems (RAMSS) [1, 2, 9, 11, 13, 14, 15, 16, 18, 19, 20, 22]. In general, these approaches instrument numerous instances of a software system, each in possibly different ways, and distribute the instrumented instances to a large number of remote users. As the instances run, they collect execution data and send them to one or more collection sites. The data are then analyzed to better understand the system's in-the-field behavior.

Recently, several RAMSS techniques have been developed. Each of these techniques collects different kinds of data and uses different analysis techniques to better understand specific system properties. One common characteristic of many of these techniques is a need to distinguish failing executions from passing executions in the field. For example, remote analyses that use information from the field to direct debugging effort need to know whether that information comes from a successful or failing execution [13, 14, 17]. For another example, applications that leverage users' platforms to validate deployed software in varied environments need an oracle that can tag the execution data that are coming from failing executions [9, 11, 18, 19, 22].

More generally, modern software systems are routinely released with known problems (e.g., for time-to-market considerations). Developers of such systems use early user experiences to help evaluate, prioritize, and isolate failures so they can later be removed. Being able to automatically identify failing executions in deployed software would allow for selectively collecting execution data only for such executions. Developers would then be able to perform tasks such as measuring how often specific failures occur, determining how severe they are, and gathering detailed information about likely causes of failure. Automatic identification of failures could also be used in-house, in place of expensive test oracles, for newly generated test cases once enough test cases have been (manually) classified.

In most cases, existing techniques address the problem of classifying executions by either focusing on program crashes or simply assuming that information about program failures is provided by some external source. In practice, however, many failures do not result in a system crash, but rather in a wrong outcome. This issue is especially problematic in the case of languages such as Java, in which fatal errors tend to result in exceptions that may be caught and discarded by the program or may simply be indistinguishable from

exceptions caused by erroneous user behaviors (e.g., a user trying to open a non-existent file).

This paper proposes and begins to study one new technique for automatically classifying execution data, collected from applications in the field, as coming from either passing or failing program runs. The technique uses a statistical learning algorithm, called random forests, to model and predict the outcome of an execution based on the corresponding execution data. More specifically, the technique builds a model by analyzing execution data collected in a controlled environment (e.g., by executing a large set of test cases in-house). It then uses this information to "lightly" instrument numerous instances of the software (i.e., captures only the small subset of predictors referenced by the model). These lightly instrumented instances are then distributed to users who run them in the field. As the instances run, the execution data is fed to the previously built model to predict whether the run is likely to be a passing or failing execution.

To help us achieve this high-level vision, we have defined and developed a preliminary version of the technique and performed a three-part feasibility study. In the study, presented in this paper, we apply the initial version of the technique to multiple versions of a medium-size software subject and study the technique's performance. Our goal is to better understand several issues crucial to the technique's success and, thereby, improve the technique's ultimate implementation.

The first study aims to determine whether it is possible to reliably classify program executions based on readily-available execution data. The following study explores the interplay between the type of execution data collected and the accuracy of the resulting classification models. The third study examines whether it is possible to further reduce the end users' data collection overhead, without overly compromising the accuracy of the classification.

We also examine some methodological and statistical issues that have not been well covered in the existing literature. In particular, we describe some useful heuristics for separating genuine relationships between predictors and the binary response (*pass/fail*) from spurious predictor-response relationships. This is particularly important in software engineering situations, in which it is easy to define vast numbers of predictors, but costly to actually collect a large enough number of data points.

The main contributions of this paper are:

- A vision for and a preliminary definition of a new technique for classifying program executions that (1) distinguishes passing and failing executions with low misclassification rates and (2) requires execution data that can be collected using lightweight instrumentation.
- An empirical evaluation of several key issues underlying this (and similar) techniques.
- A discussion of the issues related to the use of machine-learning approaches for the classification of program executions.

In the rest of the paper, we first provide background information on classification techniques (Section 2). We then introduce our approach and illustrate how we evaluated and refined it (Sections 3 and 4). Finally, we discuss in detail the results of our empirical evaluation (Section 5), provide some conclusions, and sketch out some future-work directions 6.

2. CLASSIFYING PROGRAM EXECUTIONS: BACKGROUND

Classifying program executions means using readily-available execution data to model and predict (more difficult to determine) program behaviors. Software engineers have proposed classification to address different software engineering goals, and based on different learning techniques.

2.1 Machine Learning

In very general terms, machine learning is concerned with the discovery of patterns, information and knowledge from data. In practice, this often means taking a set of data objects, each described by a set of measurable features, and relating the features' values to (known or inferred) higher-level characterizations. In this paper, we focus on a type of learning called *supervised learning*. In supervised learning, the learning algorithm builds models by analyzing feature data in which each data object has a label giving its correct high-level characterization (e.g., $\{x \mid 1 \leq x \leq 100\}$). Supervised learning tries to concisely model how the feature values relate to these labels. In particular, when the behavior is a non-numeric categorical value, such as "pass" or "fail," the supervised learning problem is known as *classification*. Because in this paper we are concerned with distinguishing passing from failing executions, we will be using classification techniques.

There is a vast array of established classification techniques, ranging from classical statistical methods, such as linear and logistic regression, to neural network and tree-based techniques (e.g., [8, 10]), to the more recent Support Vector Machines [21]. In our approach, we use a recently developed technique called *random forests* [3]. Section 3 explains the reasons that led us to chose this specific classification technique.

2.2 Classification in Software Engineering

Several software engineering researchers have used machine learning techniques to model and predict program execution behaviors. We discuss some recent work in this area.

Podgursky and colleagues [6, 7, 9, 12, 20] present a set of techniques for clustering program executions. The goal of their work is to support automated fault detection and failure classification. This work, like ours, considers different execution data and selects the ones with most predictive power. Unlike our approach, however, their work assumes that a program's failure status is either obvious (e.g., a crash) or is provided by an external source (e.g., the user).

Bowring and colleagues [2] classify program executions using a technique based on Markov models. Their model considers only one specific feature of program executions: program branches. Our work considers a large set of features and assesses their usefulness in predicting program behaviors. Also, the models used by Bowring and colleagues require complete branch-profiling information, whereas we found that our approach can perform well with only minimal information (see Section 5).

Brun and Ernst [5] use two machine learning approaches to identify types of invariants that are likely to be good fault indicators. Their approach is very specific to the problem that they are addressing and is not immediately applicable to the classification of program executions. Moreover,

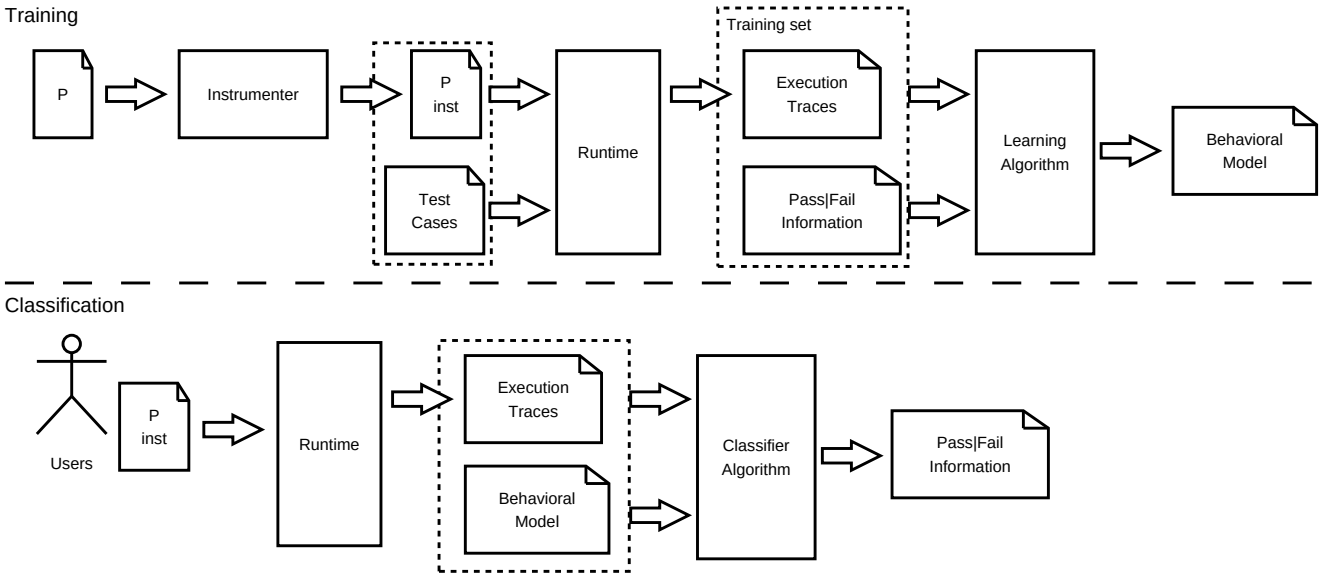

Figure 1: Overview of the technique.

their approach is also strictly dependent on the language considered—their results do not generalize from one set of subjects in C to a set of subjects in Java, or vice versa.

Liblit and colleagues [13, 14] use statistical techniques to sample execution data collected from real users and use the collected data to perform fault localization. Although related to our work, their approach is targeted to supporting debugging. Like Podgurski and colleagues' work, their approach requires failure status to be provided by an external source and may thus benefit from the technique presented in this paper.

3. OUR APPROACH

When classifying executions, developers must consider three aspects of the problem: (1) what specific *behavior* they want to classify, (2) on what *execution data* the classification will be based, and (3) what *learning technique* will be used to create the classification model. Our goal, in this work, is to distinguish passing from failing executions. Therefore, the behavior that we are interested in classifying is functional correctness. In terms of execution data, instead of selecting a specific kind of data a priori, we considered different control- and value-related types of execution information and assessed their predictive power (see Section 5). Finally, the machine learning technique that we use is random forests, explained below.

Our technique, like most techniques based on machine learning, has two phases, training and classification, as illustrated in Figure 1. In the training phase, we instrument the program to collect execution data at runtime. Then, we run the program in-house against a set of test cases. Based on its outcome, we label each test case as *pass* or *fail*. (Without loss of generality, we assume that we have an oracle for the in-house test cases.) The set of labeled execution data is then fed to the learning algorithm, which analyzes it and produces a classification model of program executions.

In the classification phase, we lightly instrument the code (capturing only the data needed by the models), which will later be run in the field by actual users. As the code runs, appropriate execution data are collected. At this point, execution data can be fed to the previously built classification model to predict whether the executions pass or fail.

The rest of this section presents tree-based classifiers in general and then discusses the specific kind of tree classifier that we use in our technique: random forests.

3.1 Tree-Based Classifiers

One popular classification method is called tree-based classification. There are several widely-used implementations of this approach, such as CART [4] and ID4 (see `http://www.rulequest.com`). Tree-based classifiers are essentially algorithms that partition the predictor-space into (hyper) rectangular regions, each of which is assigned a predicted outcome value. To classify new observations (i.e., to a take a vector of predictor variables and predict the outcome), tree classifiers (1) identify the rectangular region to which the considered observation belongs and (2) predict the outcome as the outcome value associated with that particular region.

For example, Figure 2 shows a hypothetical tree classifier model that predicts the *pass/fail* outcome based on the value of the (suitably scaled) program running time and input size. The decision rules prescribed by the tree can be inferred from the figure. For instance, an execution with $Size \geq 8.5$ is predicted as *pass*, while if $(8.5 < Size \geq 14.5)$ AND $(Time < 55)$, the execution is predicted as *fail*. One important advantage of using fitted classification trees over other classification approaches is that they provide a prediction model that is easy to interpret. However, because they are constructed using a greedy procedure, the fitted model can be quite unstable, that is, fitted classification trees can be very sensitive to minor changes in the training data [3].

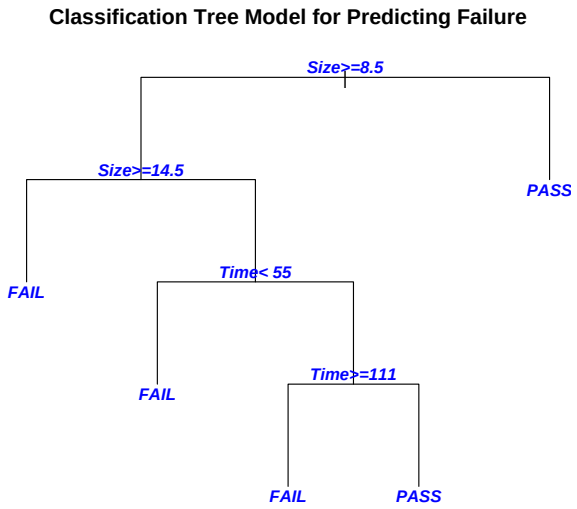

Figure 2: Example of tree classifier.

3.2 Random Forests Classifiers

As mentioned earlier, software systems have numerous features that one can envision measuring. Finding enough program runs from which to collect these measures and collecting the actual data, however, can be expensive. This problem leads to situations in which there may be a large number of predictors per program run, but relatively few runs. This scarcity of data, together with the use of greedy model-construction algorithms, often leads to spurious and unreliable classification models.

To address these problems, we chose to use a generalization of tree-based classification, called random forests, as our classification technique. Random forests is an ensemble learning method which builds a robust tree-based classifier by integrating hundreds of different tree classifiers via a voting scheme. Intuitively, this approach maintains the power, flexibility and interpretability of tree-based classifiers, while greatly improving the robustness of the resulting model.

Consider the case in which we have M features and a training set with N elements. Each tree classifier is *grown* (i.e., incrementally created) as follows:

1. Sample N cases at random with replacement (bootstrap sample), from the original data. This sample will be the training set for growing the tree.
2. A number $m << M$ is specified such that, at each tree node, m variables are selected at random out of the M and the best split on these m is used to split the node.[1]

[1]A split is simply a division of the samples at the node into subsamples. The division is done using simple rules based on the m selected variables.

The forest consists of a large set of trees (500 in our feasibility studies), each grown as described above. For prediction, new input is fed to each tree in the forest, each of which returns a predicted classification label. The most frequently selected label is returned as the predicted label for the new input. In the case of a tie (i.e., the same number of trees classify the outcome as *pass* and *fail*), one of the two outcomes is arbitrarily chosen.

Random forests have many advantages [3]. One is that they efficiently handle large numbers of variables. Another is that the ensemble models are quite robust to outliers and noise. Finally, the random forests algorithms produce error and variable-importance estimates as a byproduct. We use the error estimates to study the accuracy of our classifiers and use the variable importance estimates to determine which predictors must be captured (or can be safely ignored) in the field in order to classify executions.

4. EVALUATING AND REFINING THE APPROACH

One of the goals of this work is to evaluate the initial definition of our technique and, at the same time, improve our understanding of the issues underlying the proposed approach to further refine it. As discussed above, there are many ways in which our technique could be instantiated (e.g., by considering different types of execution data). Instead of simply picking a possible instance of the technique and studying its performance, we used an empirical approach to evaluate the different aspects of the technique. To this end, we designed and conducted a multi-part empirical study that explored three main research questions:

1. RQ1: Can we reliably classify program outcomes using execution data?
2. RQ2: If so, what kinds of execution data should we collect?
3. RQ3: How can we reduce the runtime data collection overhead while still producing accurate and reliable classifications?

To address these questions, we performed an exploratory study in which we used our technique, based on random forests, to classify program executions for several versions of a medium-sized real program. Based on this study, we addressed RQ1 by measuring classification error rates; then, we addressed RQ2 by examining the relationship between classification error rates and the type of execution data used in building classification models; finally, we addressed RQ3 by examining the effect of predictor screening (i.e., collecting only a subset of the predictors) on classification error rates.

In the following sections, we describe the design and methodology of our exploratory study in detail.

4.1 Subject

As a subject program for our studies, we used JABA (Java Architecture for Bytecode Analysis),[2] a framework for analyzing Java programs. JABA consists of about 60,000 lines of code, 400 classes, and 3,000 methods. JABA takes Java bytecodes as input and performs complex control-flow and data-flow analyses on them. For instance, JABA performs

[2]http://www.cc.gatech.edu/aristotle/Tools/jaba.html

stack simulation (Java is stack based) to track the types of expressions manipulated by the program, computes definitions and uses of variables and their relations, and analyzes the interprocedural flow of exceptions.

We selected JABA as a subject because it is a good representative of real, complex software that can contain subtle faults. In particular, for the study, we considered 19 real faults taken from JABA's CVS repository. We selected the latest version of JABA as our golden copy of the software and generated 19 different versions by inserting one error into the golden copy. In this way, we can use the golden copy as an oracle. (Actually, we also created 10 versions that contained more than one fault. However, because we did not use these additional versions in our first three studies, we do not discuss them here and present them in Section 5.3.2 instead.)

4.2 Pass/Fail and Execution Data

To build a training set for the versions of JABA considered, we used a set of executions consisting of all the test cases in JABA's regression test suite. These test cases were created and used over the last several years of the system's evolution. Because JABA is an analysis library, each test case consists of a driver that uses JABA to perform one or more analyses on an input program. There are 7 such drivers and 101 input programs—divided into real programs (provided by users) and ad-hoc programs (developed to exercise a specific feature). Thus, overall, there are 707 test cases (101 times 7). The entire test suite achieves about 65% statement coverage.

For each of the versions, we ran this complete regression test suite and collected (1) information about passing and failing test cases, and (2) various types of execution data. Because performance was not an issue in this case, we were able to collect all types of execution data at once. In particular, we collected statement counts, branch counts, call-edge counts, throw and catch counts, method counts, and various kinds of value spectra (see Section 5).

Considering all versions, we ran over 13,000 test cases. The outcome of each version v and test case t was stored in binary form: "1" if the execution of t on v terminated and produced the correct output; "0" otherwise. As mentioned above, we used the golden version of JABA as an oracle for the faulty versions. Because the test drivers output, at the end of each test-case execution, an XML version of the graphs they build, we were able to identify failures of t for v by simply comparing the golden output and the output produced by t when run on v. In addition, we labeled each failing execution, as either a *fatal failure*, for executions that terminated because of an uncaught exception (analogous to a system crash for a C program), or a *non-fatal failure*, for executions that terminated normally but produced the wrong output.

Table 1 summarizes the distribution of failures and failure types across the different versions. Each row in the table shows the version number, the total number of failures (both in absolute terms and in percentage, in parentheses), the number of non-fatal failures, and the number of fatal failures. (An asterisk in the first column indicates a failure rate greater than 8%.) For example, version 16 fails for 30 of the 707 test cases (i.e., 4.2% of the executions). Of these failures, 22 are non-fatal, whereas 8 are fatal.

Table 1: Errors associated with each version (*=failure rate greater than 8%)

Ver	Total failures	Non-fatal failures	Fatal failures
1	0 (0%)	0	0
2	12 (1.7%)	12	0
3	0 (0%)	0	0
*4	372 (52.6%)	372	0
*5	138 (19.5%)	138	0
6	0 (0%)	0	0
* 7	144 (20.4%)	144	0
8	0 (0%)	0	0
*9	86 (12.2%)	86	0
10	0 (0%)	0	0
* 11	69 (9.8%)	57	12
12	4 (0.6%)	4	0
13	38 (5.4%)	38	0
14	14 (2%)	4	10
15	0 (0%)	0	0
16	30 (4.2%)	22	8
*17	105 (14.9%)	105	0
18	0 (0%)	0	0
19	21 (3%)	14	7

4.3 Applying the Technique

For each program version and type of execution data collected, we fit a random forest of 500 classification trees using only predictors of that type. We then obtain the most important predictors by using the variable importance measures provided automatically by the random forest algorithm, and find the smallest subset of the predictors that achieves the minimal error rate.

For the study, we excluded versions with error rates below an arbitrarily chosen cutoff of 8%, since guessing 'Pass' all the time would produce reasonable (8% or better) error rates for such versions, and it is difficult to assess the prediction accuracy when there are very few failures. (Although there are ways to handle low error rates, such as boosting, we decided to eliminate those cases from our initial investigation to have more control on the study.) Six versions made the 8% cutoff. Of these, versions v4, v5, v7, and v9 produced no fatal failures, while v11 produced 12 fatal failures.

4.4 Evaluating Classifier Performance

For each resulting classification model we computed an error estimate, called the *Out Of Bag* (*OOB*) errors estimate. This quantity is computed as follows. The random forest algorithm constructs each tree using a different bootstrap sample from the original data. When selecting the bootstrap sample for the k^{th} tree, only two-thirds of the elements in the training set are considered (i.e., these elements are in the bag). After building the k^{th} tree, the one-third of the training set that was not used to build the tree (i.e., the OOB elements) is fed to the tree and classified. Given the classification for an element n obtained as just described, let j be the class that got most of the votes every time n was OOB. The OOB error is simply the proportion of times that j is not equal to the actual class of n (i.e., the proportion of times n is misclassified) averaged over all elements. This evaluation method has proven to be unbiased

in many studies. More detailed information and an extensive discussion of this issue is provided in Breiman's original paper [3] and web site (`http://stat-www.berkeley.edu/users/breiman/RandomForests`).

5. EMPIRICAL STUDIES

In this section, we describe the studies in which we applied our technique to the 19 single-fault program versions to investigate the three research questions listed above.

5.1 Study 1: Research Question 1

The goal of this first study is to assess whether execution data can be used at all to predict the outcome of program runs. To do this, we selected one obvious type of execution data, statement counts (i.e., the number of times each basic block is executed for a given program run), and used it within our technique. We chose statement counts because they are a simple measure and capture diverse information that is likely to be related to various program failures. For each JABA version there are approximately 20,000 statement counts (one for each basic block in the program).

Following the methodology described in Section 4.3, we built a classification model of program behavior for each version of the subject program. We then evaluated those models by computing OOB error estimates. We found that statement counts were nearly perfect predictors for this data set. In fact, almost every model had OOB error rates near zero. This result suggests that at least this one kind of execution data might be useful in predicting program execution outcomes.

Although statement counts were good predictors, capturing this data at user sites is expensive. For our subjects, instrumentation overhead accounted for an increase around 15% in the total execution time. While this might be acceptable in some cases, it is still a considerable slowdown that may not be practical for some applications. Moreover, the amount of information collected, one integer per basic block, can add considerable memory and bandwidth overhead for large programs and large numbers of executions.

5.2 Study 2: Research Question 2

In Study 2, we investigate whether other kinds of (more compact and easy to collect) execution data can also be used to reliably estimate execution outcomes.

Using statement counts as a starting point, we investigated whether other data might yield similar prediction accuracy, but at a lower runtime cost. Note that because statement counts contained almost perfect predictors, we did not consider richer execution data, such as data values or paths. Instead, we considered three additional kinds of data that require the collection of a smaller amount of information: throw counts, catch counts, and method counts.

5.2.1 Throw Counts and Catch Counts

Throw counts measure the number of times each throw statement is executed in a given run. Analogously, catch counts measure the number of times each catch block is executed. Each version of JABA has approximately 850 throw counts and 290 catch counts, but most of them are always zero (i.e., the corresponding throw and catch statements are never exercised). This is a typical situation for exception handling code, which is supposed to be invoked only in exceptional and often rare situations.

As with statement counts, we built and evaluated classification models using throw counts as predictors. We found that throw counts are excellent predictors for only one version (v17), with error rates well below 2%, but are very poor predictors for all other versions. Further examination of the fault in v17 provided a straightforward explanation of this result. Fault #17 causes a spurious exception to be thrown almost every time that the fault is executed and causes a failure. Therefore, that specific exception is an almost perfect predictor for this specific kind of failure. All the other throw counts refer to exceptions that are used as shortcuts to rollback some operations when JABA analyzes certain specific program constructs. In other words, those exceptions are (improperly) used to control the flow of execution and are suitably handled by `catch` blocks in the code, so they are typically not an indicator of a failure. Note that throw counts did not perform well for v11, although that version terminates with an uncaught exception in 12 cases, because the uncaught exception is a runtime exception (i.e., an exception that is not explicitly thrown in the code). Therefore, there is not a throw statement (and count) related to that exception.

The results that we obtained using catch count predictors were practically identical to those obtained using throw counts. Overall it appears that, for the data considered, throw and catch counts do not, by themselves, provide wide predictive ability for different failures. Although this may be an artifact of the specific subject considered, we believe that the results will generalize to other subjects. Intuitively, we expect throw (and catch) counts to be very good predictors for some specific failures (e.g., in the trivial case of executions that terminate with fatal failures related to explicitly-thrown exceptions). However, we do not expect them to predict reliably other kinds of (more subtle) failures and to work well in general, which is confirmed by what we have found in our study.

5.2.2 Method Counts

Method counts measure the number of times each method has been executed in a given run. For each version of JABA considered, there are approximately 3,000 method counts (one for each method in the program).

The models built using method counts performed extremely well for all program versions. Like with statement counts, method counts led to models with OOB error rates near zero. Interestingly, these results are obtained from models that use only between two and seven method count predictors (for each program version). Therefore, method counts demonstrated to be as good predictors as statement counts, but with the advantage of being less expensive to collect.

More generally, as these results suggest, there are several kinds of execution data that may be useful for classifying execution outcomes. In fact, in our preliminary investigations, we also considered several other kinds of data. For example, we considered branch counts and call-edge counts. (Branch counts are the number of times each branch (i.e., method entries and outcomes of decision statements) is executed. Call-edge counts are the number of times each call edge in the program is executed, where a call edge is an edge between a call statement and the entry to the called

method.) Both branch counts and call-edge counts were as good predictors as statement or method counts.

Note that the execution data that we considered are not mutually independent. For example, method counts can be computed from call-edge counts and throw counts are a subset of statement counts. It is also worth noting that we initially considered value-based execution data and captured data about the values of specific program variables at various program points. However, we later discarded these data from our analysis because the compact and easy to gather count data defined above yielded almost perfect predictors. In particular, because method counts are excellent and fairly inexpensive to collect, we decided to consider only method counts for the rest of our investigation. (There is an additional reason to use method counts, which is related to the statistical validity of the results, as explained in Section 5.3.1.)

5.3 Study 3: Research Question 3

The results of Study 2 show that our approach was able to build good predictors consisting of only a small number of method counts (between two and seven). This suggests that, at worst, we need to instrument around 130 of the 3,000 methods (assuming 7 different methods over 19 faulty versions). We use this result as the starting point for investigating our third research question.

One possible explanation for this result is that only these few counts contain the relevant "failure signal." If this is the case, then choosing exactly the right predictors is crucially important. Another possibility is that the signal for program failures is spread throughout the program, and that multiple counts carry essentially the same information (i.e., they form, in effect, an equivalence class). In this case, many different predictors may work equally well, making it less important to find exactly the right predictors. Moreover, if many different predictor subsets are essentially interchangeable, then lightweight instrumentation techniques are more likely to be widely applicable to other remote measurement and analysis applications.

To investigate this issue, we first performed a principal components analysis (PCA) of several sets of randomly chosen method counts. PCA is a statistical technique that identifies sets of correlated variables that together explain significant variance in the outcome variable. These variables are called the principal components. PCA then transforms the components into new composite variables that are uncorrelated with each other. The technique is often used to reduce the dimensionality of a data set and to test for the presence of important, but hidden, variables.

Figure 3 presents the results for JABA Version 4 (the results for the other versions are analogous). In the diagram, the horizontal axis represents the 10 most substantial principal components and the vertical axis represents the variance they explain. The figure shows that there are only three to (at most) five significant principal components in the data, which suggests that many of the method counts are essentially measuring the same thing.

To further investigate this issue, we randomly sampled a small percentage of the method counts and then investigated the predictive power of this small subset. More precisely, we (1) randomly selected 1% (about 30) and 10% (about 300) of the method counts, (2) built a model based only on these counts, and (3) validated the model as described in Section 4.4. We repeated this experiment 100 times, selecting different 1% and 10% subsets of method counts every time. This approach is an effective way to discover if there are many sets of common predictors that are equally significant. Without random sampling, it is easy to be misled into identifying just a few important predictors, when in fact there are other predictors which have comparable predictive capabilities. Also, random sampling provides a way of assessing how easy (or difficult) it may be to find good predictors. For instance, if 1% subsamples return a good subset of predictors 90% of the time, the number of equally good predictors is very high. On the other hand, if 10% subsets contain good predictors only 5% of the time, we would conclude that good predictors are not as easily obtained.

We found that the randomly selected 1% and 10% of method counts invariably contained a set of excellent predictors over 80% and 90% of the time, respectively. This result suggests that many different predictor subsets are equally capable of predicting passing and failing executions. This result is interesting because most previous research has assumed that one should capture as much data as possible at the user site, possibly winnowing it during later post-processing. Although ours is still a preliminary result, it suggests that large amounts of execution data can be safely ignored, without hurting prediction accuracy and greatly reducing runtime overhead on user resources. (We actually measured the overhead imposed by the collection of these data, and verified that it is negligible in most cases.) Moreover, we could further lower the overhead by combining our approach with other sampling techniques, such as counter sampling [13].

5.3.1 *Multiplicity Issues*

When the number of predictors is much larger than the number of data points (test cases, in our case), it is possible to find good predictors purely by chance. When this phenomenon happens, predictors that work well on training data do not have a real relationship to the response, and therefore perform poorly on new data. If the predictors are heavily correlated, it becomes even more difficult to decide which predictors are the best and most useful for lightweight instrumentation. Inclusion of too many predictors may also have the effect of obscuring genuinely important relationships between predictors and response. This issue is quite important because multiplicity issues can essentially mislead statistical analysis and classification. Unfortunately, this issue has been overlooked by many authors in this area.

Our first step to deal with multiplicity issues was to reduce the number of potential predictors by considering method counts, the execution data with the lowest number of entities. Further, we conducted a simulation study to understand how having too many predictors may result in some predictors that have strong predictive powers purely by chance. The simulation was conducted as follows: we selected a version of the subject and treated the method counts for that version as a matrix (i.e., we arranged the counts column by column, with each row representing the counts for a particular test case). To create a randomly sampled data set, we then fixed the row totals (counts associated with each test case), and randomly permuted the values within each row. In other words, we shuffled the counts among methods, so as to obtain a set of counts that does not relate to any actual execution.

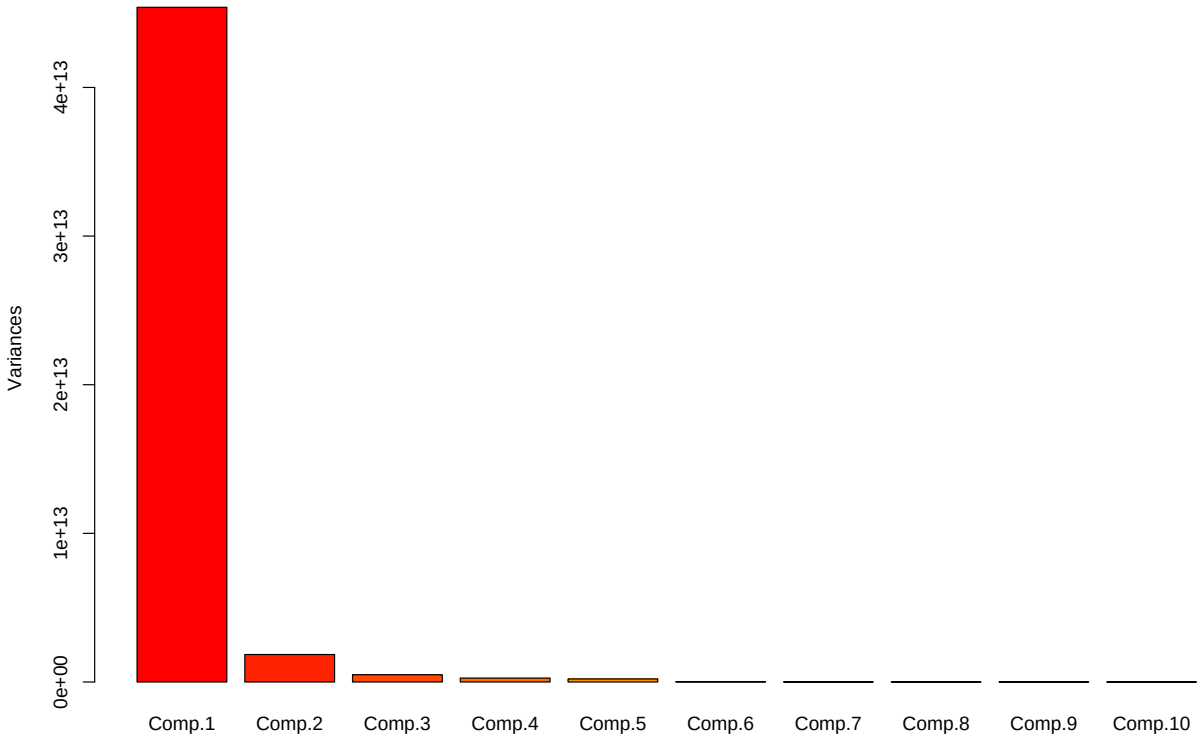

Figure 3: Principal Components Analysis.

We repeated this process for each row in the data set to produce a new randomly sampled data set. With the data set so created, we then randomly sampled 10% or 1% subsets of the column predictors. Our simulations of 100 iterations each showed that a random 10% draw never (for all practical purposes) produced a set of predictors able to classify executions as *pass/fail* in a satisfactory manner. We therefore concluded that the probability of obtaining, purely by chance, good predictors from a 1% random subset of the predictors 80% of the time (which is what we observed in our study on the real data) is very slim. We can thus conclude that the results we obtained are due to an actual relation between the occurrence of a failure and the value of the method counts.

5.3.2 *Generality Issues*

All the results presented so far are related to the prediction of the outcomes within single versions. That is, each model was trained and evaluated on the same version of the subject program. Although a classification approach that works on single versions is useful, an approach that can build models that work across versions is much more powerful and useful. Intuitively, we can think of a model that works across versions (i.e., a model that can identify failures due to different faults) as a model that, to some extent, encodes the concept of "correct behavior" of the application. Conversely, a model that works on a single version and provides poor results on other versions, is more likely to encode only the "wrong behavior" related to that specific fault.

Since one of our interests is in using the same models across versions, we also studied whether there were predictors that worked consistently well across all versions. We were able to find a common set of 11 excellent predictors for all 12 versions that we have studied. Classification using these predictors resulted in error rates below 7% for all versions of the data. Moreover, the models that achieved these results never included more than 5 of those 11 predictors.

Another threat to our results, in terms of generality, is the fact that we only considered versions with a single fault (like most of the existing literature). Therefore, we performed a preliminary study in which we used our technique on 10 additional versions of JABA, each one containing two or more errors. Also for this second set of versions, we gathered *pass/fail* information and execution data using JABA's regression test suite. Table 2 shows these additional versions in a format analogous to the one used in Table 1 (this versions are numbered starting from 20 to have unique IDs). In this case, we also show the list of faults inserted in each version (column "Faults included"). For example, version 29 contains 6 faults: 5, 9, 11, 12, 13, and 19. In this case, six of the 10 versions made the 8% cutoff, and two of them (v21 and v22) produced no fatal failures.

In the study, we selected predictors that worked well for single-error versions and used them for predicting versions

Table 2: Versions of Jaba with multiple errors (*=failure rate greater than 8%)

Ver	Faults included	Total failures	Non-fatal failures	Fatal failures
20	1 3	0 (0%)	0	0
* 21	2 17	113 (16%)	113	0
* 22	9 12	86 (12.2%)	86	0
23	4 7 13	0 (0%)	0	0
24	13 12 19	7 (1%)	0	7
* 25	7 16 13 17	113 (16%)	105	8
26	7 19 5 11	19 (2.7%)	0	19
* 27	2 5 19 9 12	102 (14.4%)	95	7
* 28	13 17 9 16 12	180 (25.5%)	172	8
* 29	13 19 9 12 5 11	101 (14.3%)	83	18

with multiple errors. We found that, although there are instances in which these predictors did not perform well and produced error rates around 18%, the predictors worked well most of the time, with error rates below 2%. In future work, we will further investigate the use of our technique in the presence of multiple errors to better understand its performance in those cases and possibly adapt it.

5.4 Discussion

In our studies, we used a real program and a set of real faults and carefully considered various statistical issues. Nevertheless, as with all empirical studies, these results may not generalize because they are based on one type of program and on one set of test cases and failures. We must perform further experiments with more subjects and faults to increase the external validity of our findings.

Keeping these caveats in mind, we draw the following initial main conclusions. These studies suggest that, for the subject and executions considered, we can (1) reliably classify program outcomes using execution data, (2) do it using different kinds of execution data, and (3) do it both reliably and at a low cost. These results represent a first important step towards our overall goal of defining a technique that can be reliably used as an oracle for field executions.

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented and studied a technique for remote analysis and monitoring of software systems whose goal is to automatically classify program execution data as coming from passing or failing program executions. Such a technique can support various kinds of analysis of deployed software that would be otherwise impractical or impossible. For instance, the technique would allow developers to measure how often specific failures occur and to gather detailed information about likely causes of failure.

We studied three fundamental questions about the technique. The results of our studies show that, for the cases considered, we can reliably and efficiently classify program outcomes using execution data. Moreover, our technique was able to generate models that can classify execution outcomes across versions. We consider this work a first, important step towards our overall goal: the development of a classification technique that can be reliably used on deployed software to provide an automated oracle for field executions.

We will continue our investigations in several directions. The first direction will be to further study our technique in the presence of multiple faults. Although studying single faults is important for an initial assessment of the approach, realistic situations are likely to involve multiple faults in most of the cases. The results of the preliminary study described in the paper will drive our future work in this direction.

The second direction for future work is the investigation of the relation between predictors and faults. We will investigate whether the entities with the strongest predictive power can be used to localize the source of the problem (i.e., the fault or faults) that caused the failure. Also in this case, we have performed an initial investigation and assessed that, for most of the cases considered, a direct relation between predictors and faults does not seem to exist. In fact, most faults seem to cause failures whose effects are spread throughout the code, which justifies the high number of excellent, alternative predictors identified by our technique. We will further investigate this issue by looking at various kinds of relations, such as data dependences, between the faulty statement(s) and the entities that are good predictors (e.g., the methods or statements whose counts result in a perfect classification).

Finally, we will perform an experiment involving real users to validate our technique in real settings. We will release the versions of JABA that we have studied to a set of students in a program analysis class. We will modify such versions by adding assertions that will be triggered when a known failure manifests itself. (Otherwise, failures that result in an incorrect outcome but no fatal exception may go undetected.) Additionally, we will ask the students to tag the executions to account for possibly unknown faults and be able to perform a more controlled experiment. This study will let us assess whether our models maintain the same predictive power that we assessed in our evaluation when they are applied to real users' executions.

Acknowledgments

This work was supported in part by National Science Foundation awards CCF-0205118 to the National Institute of Statistical Sciences (NISS), CCR-0098158 and CCR-0205265 to University of Maryland, and CCR-0205422, CCR-0306372, and CCR-0209322 to Georgia Tech. We used the `R` statistical computing software and the `randomForest` library, available at `http://cran.r-project.org/` to perform all statistical analyses. Jim Jones prepared and provided the 19 faulty program versions. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

7. REFERENCES

[1] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2002)*, pages 2–8, Charleston, SC, USA, november 2002.

[2] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 195–205, July 2004.

[3] L. Breiman. Random Forests. *Machine Learning*, 45(1):5–32, Oct. 2001.

[4] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[5] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 480–490, May 2004.

[6] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: the distribution of program failures in a profile space. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of Software Engineering*, pages 246–255, September 2001.

[7] W. Dickinson, D. Leon, and A. Podgursky. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, pages 339–348, May 2001.

[8] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley, second edition, 2000.

[9] P. Francis, D. Leon, M. Minch, and A. Podgurski. Tree-based methods for classifying software failures. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 451–462, November 2004.

[10] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, 2001.

[11] D. M. Hilbert and D. F. Redmiles. Extracting usability information from user interface events. *ACM Computing Surveys*, 32(4):384–421, Dec 2000.

[12] D. Leon, A. Podgurski, and L. J. White. Multivariate visualization in observation-based testing. In *Proceedings of the 22nd international conference on Software engineering (ICSE 2000)*, pages 116–125, May 2000.

[13] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI 2003)*, pages 141–154, June 2003.

[14] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI 2005)*, June 2005.

[15] Microsoft online crash analysis, 2004. `http://oca.microsoft.com`.

[16] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the 9th European Software Engineering Conference and 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*, pages 128–137, Helsinki, Finland, september 2003.

[17] A. Orso, J. A. Jones, and M. J. Harrold. Visualization of program-execution data for deployed software. In *Proceedings of the ACM symposium on Software Visualization (SOFTVIS 2003)*, pages 67–76, San Diego, CA, USA, june 2003.

[18] A. Orso and B. Kennedy. Selective Capture and Replay of Program Executions, may 2005. `http://www.csd.uwo.ca/woda2005/proceedings.html`.

[19] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proceedings of the 21st International Conference on Softw. Eng., 1999*, pages 277–284, May 1999.

[20] A. Podgurski, D. Leon, P. Francis, W. Masri, M. M. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 465–474, May 2003.

[21] J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.

[22] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 45–54, 2004.