A Dynamic Approach to Statistical Debugging: Building Program Specific Models with Neural Networks

Matthew Wood
Georgia Institute of Technology
woodzy@cc.gatech.edu

Abstract

Computer software is constantly increasing in complexity; this requires more developer time, effort, and knowledge in order to correct bugs inevitably occurring in software production. Eventually, increases in complexity and size will make manually correcting programmatic errors impractical. Thus, there is a need for automated software-debugging tools that can reduce the time and effort required by the developer. The performance of previously developed debugging techniques can be greatly improved by combining them with machine-learning. Our research focuses on the application of neural networks within the domain of statistical debugging. Specifically, we develop methods to mine statistical debugging data that can then be used to train neural networks; these generated multi-layered neural networks can then be used to identify suspicious programmatic entities. Our developed networks are generated on a per program basis in order to leverage specific programmatic properties. In our empirical evaluation we compare our proposed approach with a state-of-the-art automated debugging technique. The results of the evaluation indicate that, for the cases considered, our approach is more effective than the considered technique.

Contents

1. INTRODUCTION

The more complex software becomes, the more arduous debugging tasks will become as well. This necessitates a strong need for the development of practical automated software debugging tools; thus for the past decade software engineers have focused on developing practical automated debugging tools. Current automated debugging techniques develop a profile for a program's execution either through static inspection or dynamic instrumentation [Reps et al. 1995; Renieris and Reiss 2003]. These profiles are then subjected to mathematical and heuristic based algorithms in hope of helping a developer identify the programmatic instruction(s) causing a fault or bug [Liu et al. 2005; Liblit et al. 2005; Renieris and Reiss 2003; Bowring et al. 2003; Cleve and Zeller 2005]. These different dynamic and static profiling techniques have been developed and are utilized by debugging algorithms in order to accomplish this goal [Liblit et al. 2005; Zheng et al. 2006; Cleve and Zeller 2005; Brat et al. 2000]. One recently developed profiling technique is statistical debugging, and over the past few years has seen significant and promising development [Liu et al. 2005; Liblit et al. 2005; Liblit et al. 2003].

Recently published automated statistical debugging techniques focus on collecting dynamic execution statistics, and performing statistical analysis on the collected information [Liblit et al. 2005; Liu et al. 2005]. These techniques automatically instrument a program's source code to collect control-flow, data-flow, and program state information at runtime. A statistical analysis identifies a set of suspicious program attributes that may be the root cause of the failing executions. The identification of faults generally relies upon a predetermined model driven by specific statistical data. Thus a predetermined model clearly limits the amount of statistical data incorporated. Incorporating more complex statistical input into a predetermined model is more difficult to justify formally and reduces a model's generality.

Our research utilizes a machine learning algorithm to correlate multiple programmatic statistics into one program-specific model. This program-specific model is developed through training and produces more accurate fault localization than predetermined models. Our models also highlight the statistical data most representative of a fault's symptoms. The machine learning model we use is a multi-layered neural network. This model is then used to classify programmatic faults via statistical data. A multi-layered neural network is capable of representing non-differentiable mathematical models and thus is preferable to other machine learning techniques [Mhaskar and Micchelli 1992]. The flexability provided by neural networks allows highly dynamic and complex mathematical models to be developed from per program statistics and thus is not limited to a specific statistical interpretation, a predetermined model.

2. BACKGROUND

Crucial to the development of debugging is an understanding of the types of bugs that a developer must handle, hence an automated debugging technique as well. Subsequently, a definition of automated debugging is followed by introducing different debugging techniques including the methodology our research builds upon, namely statistical debugging. Statistical debugging, after a cursory introduction, is then further developed; we introduce a basic methodology and highlight an overview of the algorithms involved within. After introducing some automated debugging techniques, we then introduce perceptrons and multi-layered neural networks. Finally, in order to provide a more practical grounding for our debugging techniques, an argument for the scalability of statistical debugging is presented, as well as, how statistical debugging techniques capitalize on this methods inherent properties.

2.1 Bug Classification

Initially we must begin by introducing the two classes of bugs that exist: deterministic and non-deterministic. Deterministic bugs are faults that manifest immediately with the execution of a specific path. Non-deterministic bugs manifest in a multitude of ways (mainly memory corruption, race conditions, buffer overflows, and null pointers) and are not necessarily reproducible given a specific execution path. Non-deterministic bugs naturally also depend on the environment within which they are exhibited.

Not only are bugs deterministic or non-deterministic, bugs are either fatal or non-fatal. Fatal executions will manifest as a crash usually based on some operating system violation; while non-fatal executions will manifest as incorrect output or glitches that are generally ignored or disregarded. Finding fatal deterministic bugs can be relatively easy with conventional debugging methods, stack traces and register dumps, used by an informed developer. Detecting non-fatal deterministic bugs or non-deterministic bugs, on the other hand, is far more difficult. Conventional debugging methods generally do not provide enough information associated with the fault to correctly identify it. Automated debugging techniques aim to help a developer identify where these faults exist or at least help direct an informed developer in the correct direction [Renieris and Reiss 2003; Liblit et al. 2005; Zheng et al. 2006; Liblit et al. 2003; Liu et al. 2005; Liu et al. 2006].

2.2 Automated Debugging

There are multiple types of automated debugging techniques that are used by programmers and can be divided into two categories: static and dynamic. Static techniques focus on the analysis of source code and coding heuristics to locate program defects with or without the availability of a program model/specification [Brat et al. 2000]. Dynamic techniques generally do not require any information about the program model or semantics but simply an oracle, or informed developer, that will label individual program executions as correct or incorrect [Reps et al. 1995; Liblit et al. 2003]. Statistical debugging is one such dynamic method that has recently captivated the interest of many researchers within the software engineering field [Liblit et al. 2005; Liu et al. 2005; Zheng et al. 2006; Zheng et al. 2003; Liblit et al. 2003].

2.3 Statistical Debugging

Statistical debugging focuses on instrumentation of predicates, an assertion about program variables that is either true or false, within a program and records these statistics in some form of a report at the end of execution. More specifically, a statistical debugger will first instrument a program's source code by identifying some predicates and inserting statistical checks to collect instrumentation data during execution. The new executable is then run with various input parameters; statistical data from a programs execution will then be collected. The statistics collected from each instrumentation site are then formulated into a report containing specific execution data and parameters; this is referred to as a test case. Finally, this test case is labeled as passing or failing by an oracle to determine whether the output was correct or not. Such an oracle is part of a standard test-suite in production test environments; generally it is a developer, a tester or a set of files that contain corresponding input and output pairs [Renieris and Reiss 2003; Liblit et al. 2003]. Statistical debugging techniques take this set of test cases and apply an algorithm determining which predicates are responsible for the programs failure. These algorithms are generally based on some comparison algorithm or metric dependent on correct and incorrect test cases [Liblit et al. 2005; Liu et al. 2005; Zheng et al. 2006; Liblit et al. 2003]. We defer specific development of these algorithms until Section 3. Generally for such an algorithm to correctly identify fault locations, many test cases will be required [Liblit et al. 2005]. Not only do there need to be a sufficient number of test cases, but the instrumentation coverage must be sufficient as well. Generally multiple instrumentation sites will be needed for every line of code [Liblit et al. 2005; Zheng et al. 2003; Liblit et al. 2003]. Collecting statistics at every instrumentation site is a time consuming process and is likely to reduce the attractiveness of this technique; but not every execution needs to observe every instrumentation site for this method to be effective[Bowring et al. 2003; Orso et al. 2003; Liblit et al. 2003].

2.4 Machine Learning

Suspending our discussion on debugging techniques, we introduce neural networks that when coupled with statistical debugging can develop complex interpretive mathematical models. Most other researchers have developed models by hand, while neural networks can produce novel organizations of the basic statistics collected via predicate instrumentation on a per program basis [Liblit et al. 2005; Liblit et al. 2003; Liu et al. 2005; Zheng et al. 2006]. We introduce the following basic machine learning techniques: a single-layered neural network and a multi-layered neural network.

2.4.1 *The Single-layered Neural Network.* Also referred to as a perceptron, the single-layered neural network is a linear classifier; meaning that it can only separate two classes via a $(N-1)$-th dimensional plane in N-space. A perceptron is composed of inputs, network weights, an activation function and an output. For each input a_i, where $1 \leq i \leq n$, the perceptron is correlated with a network weight w_i. The input sum, in, is calculated as follows:

$$in = \sum_{1 \leq i \leq n} (a_i * w_i)$$

This sum is then processed by an activation function, g, to determine whether this perceptron, which can also be viewed as a neuron, will fire. Generally this activation function is either a sigmoid function or a step function. An example step function might be:

$$g(in) = \begin{cases} 1 & in \geq 0 \\ 0 & in < 0 \end{cases}$$

While a sigmoid function is non-linear and is generally similar to:

$$g(in) = \frac{1}{1 - e^{-x}}$$

These activation functions $g(in)$ represent the classification, either a 1 or 0, of an input set $A = \bigcup(a_i)$. There are generally many input sequences $A \in \Phi$ on which the perceptron is then trained. The perceptron seeks to classify a maximal number of input sets within Φ as correctly as possible, given we know the correct classification of each specific input $a_i \in A$. The learning phase of this model is affected by changing the weights in a manner that increases the number of correctly classified inputs. This is generally done by adding a learning rate scaled by the activation function $g(in)$ to the weight vector W.

There are several weaknesses to the single layered neural network. Most importantly, it is unable to represent non-linear classifications, thus it is only capable of dividing on N-space into two regions. While the classification between a bug or not a bug is two regions in space, generally collected statistics are not separable through such a linear function. Thus a more expressive model is needed: one that is capable of more complex classification regions.

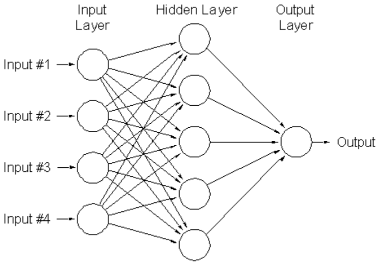

Fig. 1. Multi-Layered Neural Network Example Organization

2.4.2 *The Multi-layered Neural Network.* A multi-layered neural network is based upon the combination of many perceptrons/neurons (see Figure 1). This combination allows for a multi-layered neural network to model any function differentiable or not. Unfortunately the training and representation of a multi-layered is far more complex. The constructs are similar to the perceptron except that instead of a single weight vector w, there will be many simultaneous weight vectors W operating in parallel. More formally we have an input set A that is mapped to an arbitrary number of neurons in the input layer. These neurons each have their own activation function which can be a gaussian function, sigmoid function or linear function. In order to represent non-linear functions, a non-linear activation function must be used. After the neurons activation is computed, the activation output is mapped into another layer of neurons as input. This process can be repeated recursively until the output layer is reached. The output layer generally has a predetermined number of neurons, and thus this specific number of outputs has each of the previous layers activation outputs mapped to every output neuron's inputs.

2.5 Scalability

In order to scale statistical debugging, or any debugging technique, to larger program domains, one must consider the potential performance issues that complicate the collection of full statistics during a program execution. A current approach proposed for statistical debugging distributes executables with pre-compiled statistical collection mechanisms across the user-base [Bowring et al. 2003; Liblit et al. 2003]. This approach appears to be very promising because the more complex software becomes, the more unlikely it is that a group of engineers will be able to test all conceivable failure conditions. Thus due to economic and time limitations a user is generally subjected to errors and faults that were not tested or realized by the organization's engineers. Capturing information from user executions of an organization's software is hence an excellent and necessary means for collecting information on failures in the software [Bowring et al. 2003; Orso et al. 2003; Liblit et al. 2003]. There are many organizations that already capture limited execution information from users; the largest of these organizations include Microsoft, GNOME, Mozilla, Netscape, KDE and Open Office [Liu et al. 2006]. The execution reporting mechanisms used by these large organizations only report stack traces and register contents; furthermore reports are only generated in the event of a crash. As discussed previously, not all programmatic bugs are manifested by a crash. Many bugs exist due to logic errors or incorrect outputs that do not necessarily crash a program, but can easily be recognized as incorrect by a developer. These types of bugs are subsequently much more difficult to triage. With a distributed statistical debugging system in place an organization could effectively diagnose and fix identified bugs more efficiently. A further goal of automated debugging is to identify new, complex bugs without the requirement for developers debugging manually, or even the knowledge of where a bug exists [Liblit et al. 2005].

3. RELATED RESEARCH

The discussion thus far has focused on a high level description of statistical debugging and not specifically discussing the details of the algorithms used to rank predicates. Given this high level of understanding, we shift the discussion to the first algorithms used to interpret statistical data. Finally, an in-depth look at three of the current algorithms will be taken and discuss their strengths and weaknesses.

3.1 Early Statistical Debugging Techniques

In order to provide background knowledge of the current techniques, a quick review of early statistical debugging techniques is necessary. The *Union* and the *Intersection* functions of related predicates was first proposed as a comparison metric by Renieris and Reiss [2003]. More specifically their methodology assumed that there exists a set of passing executions, $P = \cup_{1 \leq i \leq n} p$, and one failing execution, f. We then define $S(x)$ to be the set of program predicates for program executions x. We can now define *Union* to be a fault localization report:

$$Union(P, f) = S(f) - \left[\bigcup_{p \in P} S(p) \right]$$

We can also subsequently now define *Intersection*:

$$Intersection(P, f) = \left[\bigcup_{p \in P} S(p) \right] - S(f)$$

Renieris and Reiss [2003] compared their nearest neighbor algorithm to the union and intersection methods. The nearest neighbor algorithm works by comparing f with the entire set P and locating the most similar, nearest neighbor execution, p_k; then a fault localization report is generated as follows:

$$NN(p_k, f) = S(f) - S(p_k)$$

A further mathematical treatment and discussion of all three algorithms exists in [Renieris and Reiss 2003]. These algorithms are relatively simple to implement and understand, as well as being relatively good predictors of fault locales. This intuition provided the original impetus for further academic research into program execution for automated debugging, which then lead to looking at purely statistical data.

Zheng et al. [2003] developed a method based on regularized logistic regression that attempts to select a set of predicates that best predict the outcome of every run [Liblit et al. 2005]. This is done through the designing of a regression utility function which is then subjected to an optimization technique (gradient ascent) based on the collected statistics containing correct and incorrect executions [Zheng et al. 2003]. This method is effective at identifying bugs in smaller programs, but as the program space is expanded so does the redundancy and misclassification [Liu et al. 2006; Liblit et al. 2005].

3.2 Cause Isolation

Liblit et al. [2005] continue to build upon previous work in the field of statistical debugging by further developing a statistical model for representing collected predicate data. Their work presents a new scheme, called cause isolation, that isolates multiple bug predicates, or predictors, based on the statistical relevance to the program's failure. In order to isolate multiple bug predicates effectively, Liblit et al. [2005] also present a statistical test to prune super-bug predictors and sub-bug predictors. Such predictors were realized to cause redundancy and introduce information irrelevant to the faults; this was a limitation of previous ranking algorithms. Super-bug predictors are predicates that predict an execution's failure due to numerous bugs. On the other hand, sub-bug predictors represent predicates that are a subset of failing conditions of a specific bug, the aftermath of a bug [Liblit et al. 2005]. With this pruned set of predicates the following formula are then applied: the harmonic mean between the predicates relevance, as defined by Liblit et al. [2005], and the ratio of the observations of the predicate to the total number of test cases. This fault localization formula provides two major accomplishments. First, the ability to remove super-bug predictors and sub-bug predictors algorithmically; as well as ranking predicates without the need for a detailed program specification. This allows statistical debugging using cause isolation to simply require the knowledge of passing or failing executions. Without the restriction for program specifications, statistical debugging is not limited to specific bugs that are known to exist; thus any unknown deterministic or non-deterministic fault can be revealed simply by classifying program execution's passing or failing.

3.3 Hypothesis Testing

Liu et al. [2005] seek to advance the dynamic analysis techniques of Liblit et al. [2005] by reinterpreting the predicate statistics to locate predicates with significant difference between a normalized distribution of passing and failing executions. Because the ranking and pruning methodology of Liblit et al. [2005] only relies on whether a predicate was either observed or not in each test case, this methodology fails to cope well with predicates observed only once in all test cases. Liu et al. [2005] therefore focus on developing a new approach that can utilize information about the number of times an observation of a specific predicate occurs [Liu et al. 2006]. Instead of identifying predicates strongly related to incorrect executions, as Liblit et al. [2005] suggests, it makes statistical sense to develop models for both correct and incorrect executions per predicate [Liu et al. 2005; Liu et al. 2006]. Measuring the difference in these models will reveal the predicates relevance to a failure [Liu et al. 2006]. Because it is a fallacy to assume predicates necessarily follow any regular statistical distribution, a test statistic having the properties of a normal distribution under the null hypothesis must be developed [Liu et al. 2005]. Specifically their null hypothesis states that the ratio of predicates observed to be true to the total number observations per test case should be the same in correct and incorrect test cases. It is intuitive that these ratios should differ for predicates which are strongly correlated to a failing condition or predicate. This test case data is then combined into a normalized distribution (based on the Central Limit Theorem) and the difference between the passing and failing models is computed revealing the predicates fault relevance, or suspiciousness [Liu et al. 2005].

3.4 Identification of Multiple-Bugs

Both of [Liu et al. 2006] and Liblit et al. [2005] focus on the identification of the predicate which delivers the highest relevance to a single fault, but neither algorithm has the ability to identify distinct bugs and deliver a report containing predicates most relevant to these uniquely identified bugs [Zheng et al. 2006]. It is difficult to separate one bug from others in a suite of test cases since each test case may exhibit somewhat predictive predicates for an arbitrary number of bugs. It is similarly as hard to simultaneously recognize a way to select predicate features that help identify or categorize the bugs into distinct sets. Zheng et al. [2006] present an intuition: Predicates should group by the runs that they predict; runs should group by the predicates that predict them [Zheng et al. 2006]. This intuition effectively implies an algorithm to single out unrelated predicates and associates predicates that are related. Thus Zheng et al. [2006] are able to categorize and classify bugs based on no prior knowledge except passing or failing test cases. Zheng et al. [2006]'s algorithms utilize the predicates to essentially perform a collective voting step within a bi-clustering algorithm; this further associates related predicates. The complete algorithm is beyond the scope of this thesis but is fully treated by Zheng et al. [2006].

4. PROPOSED APPROACH

Our approach focuses on combining per predicate statistics collected during execution with machine learning techniques. The goal of our approach is to develop a program specific ranking model by leveraging previously identified bugs and their associated predicates to help identify bugs and their predictive predicates. This dynamic model does not exhibit the limitations of previous models. These limitations are bypassed by using a broad per predicate feature set coupled with dynamic per program training. In the following subsections we formally develop the statistical data used to train a neural model on a per predicate basis.

4.1 Definitions

When each program P is executed with a given input and expected output, we refer to this as a test case. A program P is associated with n test cases. We denote each test case as T_i, where $1 \leq i \leq n$. Each T_i has an associated input that will produce output that will either match the expected output or not. We thus refer to a test case which causes a program P to produce incorrect output as a failing execution. Conversely, we refer to a test case which produces correct, or expected, output as a passing execution. Now we construct the following two classification functions.

$$Output_F(T_i) = \begin{cases} T_i & \text{iff } T_i \text{ is a failing execution} \\ \emptyset & \text{otherwise} \end{cases} \tag{1}$$

$$Output_P(T_i) = \begin{cases} T_i & \text{iff } T_i \text{ is a passing execution} \\ \emptyset & \text{otherwise} \end{cases} \tag{2}$$

Using functions 1 and 2, we can now define the following sets.

$$T_F = \bigcup_{1 \leq i \leq n} Output_F(T_i)$$

$$T_P = \bigcup_{1 \leq i \leq n} Output_P(T_i)$$

Each instrumentation point in which we collect program execution statistics is called a predicate. A program has a fixed number of predicates which are executed on both passing and failing executions. A predicate, p, consists of two counters: the number of times this predicate was observed to be true, and the number of times the predicate was observed to be false. Because predicates are staticly located across executions we refer to them simply as p, but utilize the following functions to represent the data collected by each predicate p.

Definition 1. $Executions_{T_i}(p, true) \equiv$ *the number of times p was observed to be true in* T_i

Definition 2. $Executions_{T_i}(p, false) \equiv$ *the number of times p was observed to be* $false$ *in* T_i

Both of these functions help identify the number of times a predicate was executed. The summation of both is effectively the total number of times p was executed on a given test case T_i. Using definitions 1 and 2 we can now define the following useful set of functions:

$$Observed_{T_i,true}(p) = \begin{cases} 1 & \text{iff } Executions_{T_i}(p, true) \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

$$Observed_{T_i,false}(p) = \begin{cases} 1 & \text{iff } Executions_{T_i}(p, false) \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

4.2 Machine Learning Features

With these interpretive functions defined, a formal definition of the features used in our machine learning algorithms can be developed. Utilizing the functions and definitions presented previously, we will in the following section develop six features. Each feature is applicable to every predicate and will be utilized within the neural network to identify, classify or rank the predicates likely to be responsible for a bug. Our features explore the probability that a predicate will be observed within the passing and failing test cases, the probability that when a predicate is observed if its value is true, and how predicate execution counts differ during passing and failing executions.

4.2.1 *Probability of Observation.* We first seek a feature that will help identify how likely a predicate is to be executed over all passing or failing test cases. A predicate is reached during execution of test case T_i if the predicate p is *Observed*. The summation of observations over a set of test cases leads to the total number of test cases in which the predicate p was executed, and thus relevant to program flow. If a predicate p is correlated with a failing set of test cases, we expect that the probability predicate p will be observed on any given passing test case will be different from the probability on any given failing test case. More formally we define the following two functions:

$$P_{T_F}(p) = \frac{\sum_{T_i \in T_F} (Observed_{T_i,false}(p) \parallel Observed_{T_i,true}(p))}{|T_F|} \tag{5}$$

$$P_{T_P}(p) = \frac{\sum_{T_i \in T_P} (Observed_{T_i,false}(p) \parallel Observed_{T_i,true}(p))}{|T_P|} \tag{6}$$

These functions define the correlation of a predicate p to the failing or passing executions. But it is not enough to simply focus on this correlation, some points in a program are always executed (initialization routines) and thus require an additional interpretation.

4.2.2 *Probability Observation is True.* Because a failing execution may be associated with a predicate which is always observed to be false (an error condition), we need another set of formulae which will isolate this set of predicates. To isolate true and false predicate outcomes, we can simply restate the previous equations, but force that our observations of predicate executions be true. Formally we define:

$$P_{T_P}(p \mid p \text{ is true}) = \frac{\sum_{T_i \in T_P} Observed_{T_i,true}(p)}{|T_P|} \tag{7}$$

$$P_{T_F}(p \,|\, p \text{ is true}) = \frac{\sum_{T_i \in T_F} Observed_{T_i,true}(p)}{|T_F|} \tag{8}$$

While the previously stated formulas are able to identify major changes in program execution flow, we are still disregarding some execution data.

4.2.3 *Failing and Passing Bias.* In the previous features, we normalize the execution count of predicate p on each test case T_i to either one (observed) or zero (unobserved). Clearly information is lost with this generalization. In the following formulation we hope to capture the number of times a predicate was executed and then, in a non-destructive manner, normalize the values so that they may be compared. In order to normalize an arbitrary range of values, we incorporate data between passing and failing executions at the predicate level. The resulting formula can then be used to normalize the arbitrary range of execution statistics. Our goal with this statistic is to develop a normalized measure for comparing passing and failing executions that is test case independent. Liu et al. [2005] uses a variation of this statistic to generate a normal distribution for passing and failing executions. Thus we know this normalization contains predictive information and will be a useful feature. In our grammar we restate Liu et al. [2005]'s per predicate statistic as follows:

$$\varphi_{T_i}(p) = \frac{Executions_{T_i}(p, true)}{Executions_{T_i}(p, true) + Executions_{T_i}(p, false)} \tag{9}$$

And from the above equation, we can develop the following two features:

$$FailBias(p) = \frac{\sum_{T_i \in T_F} \varphi_{T_i}(p)}{|T_F|} \tag{10}$$

$$PassBias(p) = \frac{\sum_{T_i \in T_P} \varphi_{T_i}(p)}{|T_P|} \tag{11}$$

5. EXPERIMENT EVALUATION

With the complete development of our feature set we now need an effective way to test our approach. In order to test our approach, a corpus of data is required, as well as a methodology for collecting execution statistics that are relied upon for making our predictions. We also must develop a way to evaluate our approach. Since our approach requires historical execution information, the previously used methods of evaluating statistical debugging techniques do not apply. We therefore need to develop a method in which we will be able to compare previous methods in fair settings.

5.1 Test Suite Development

The Software-artifact Infrastructure Repository, SIR, hosted at the University of Nebraska at Lincoln provides a host of test suites containing many different versions of programs containing instrumented or identified bugs [Do et al. 2005]. Two different programs from this repository were used: GZIP and TCAS. TCAS is a small program of about three-hundred lines of code used as an aircraft collision avoidance system and is a part of the Seimen's test suite. GZIP is a standard Unix command-line utility used to compress files and is a medium-sized program of about seven-thousand lines of code. SIR provides multiple versions (with many bugs) of each program as well as a complete test suite for both. Thus each version has a specific test suite, although some may overlap, and contain a set number of test cases.

5.2 Statistic Collection

Statistic collection is done during execution of a test case from SIR. In order to collect statistics from an execution we must first analyze the programs source code and identify all properties we wish to monitor. The program properties monitored include function calls, branches, and local/global scalar pairs. Function calls, branches, and local/global scalar pairs represent the predicates that will be used as predictive statistics in our model. The return value of a function is sampled after each function call; this leads to three predicates: was the return value greater than zero, less than zero, or equal to zero. A similar approach is taken to local/global scalar pairs; whenever a variable assignment occurs, we monitor the left and right side of the assignment and report three predicates: was the left side greater than the right side, less than the right side, equal to the right side. With branches we focus on the direction of program flow, thus we record the number of times a branch was taken or not taken during program execution. These predicates are similar to those collected by Liblit et al. [2005] and Liu et al. [2005] and used in their predicate ranking statistical models.

In order to sample these predicates we must modify the base source code. A source to source transformation is done, making the necessary source code changes to identified and sample predicate locations in order to collect statistical data. Each instrumentation point will keep track of the number of times the associated predicate is executed. After execution these collected statistics are output to a file in a standard format. In order to collect information on all test cases (many executions) these files are stored and mined later for statistical data according to the predicate feature set previously developed.

5.3 Neural Network Training

With the statistics collected and feature sets for all predicates developed, we are now able to train the neural network with program specific statistics. The neural network we will be developing is a feed forward fully connected neural network. Feed forward implies there are no backwards connections, flow goes continuously from the input layer to the output layer; this type of model is well studied and the training methods are mathematically sound [Mhaskar and Micchelli 1992]. Fully connected implies that all inputs will be mapped to all first layer neurons, furthermore all first layer neurons will be connected to all second layer neurons and so forth, until the output layer is reached. A neural network with the properties of a fully connected feed forward network can be trained by an algorithm known as backpropagation.

5.3.1 *Backpropagation.* Backpropagation is one method in which a neural network is trained, where training implies the minimization of error in the neural networks output; specifically the mean squared error (MSE) between the output and the expected output vectors. Backpropagation algorithms focus on updating the weights relating to a neurons output. This is a difficult task, especially when hidden layers in neural network are involved. These layers do not have direct inputs or outputs, but have inputs from other neurons, as well as their outputs being fed as inputs to other neurons. Backpropagation focuses on providing a mechanism to aid in the training of hidden neurons whose contribution to the final output is unclear.

In order to train a neural network via backpropagation, we must have a training set T with inputs, I, and expected outputs, E. The goal of training is to minimize the mean squared error of the actual network output, O. In vector notation, the MSE defined as follows: calculating $(O - E)^2$, then computing the square root of the mean of all elements in the vector. Armed with the MSE, we now have an approximate idea of how much training still must occur to the neural network. The MSE, referred to as δ from now on, represents how much the connected neurons need to be updated, thus δ needs to be passed up to parent neurons. But each neuron connected to the output layer is not completely responsible for δ, thus we need to calculate how much each neuron is responsible for δ. Luckily the weighted neuron connection is associated with δ, thus the amount a connection will need to be updated is $w * \delta$. After this calculation we then are able to update the rest of the neural network recursively in this manner.

5.3.2 *Selecting Training Data.* In order to develop a program specific model, we need training data from other versions of a program. Different versions of a program coupled with fault inducing predicates creates a training set. Our two test programs GZIP and TCAS each contain multiple versions with unique faults. It is necessary to select a portion of the versions to train the model, more than one version is preferable to reduce any over-training that might occur. Out of the eleven versions of TCAS, generally a set of five versions, not including our test subject, will be used to train the neural network. We have a total of three versions of GZIP, thus to train a model to work with GZIP, we used the two unassociated versions to train the network. Using these training versions implies that we used the collected predicate data and developed feature sets per predicate of each version execution, then combined the entirety of this data into a network training file. This is then used to train the neural network via backpropagation.

5.4 Method Comparison

In order to compare our method to previous methods, we interpret our results as a predicate ranking scheme. The statistical models in previous research seek to effectively provide a ranking of predicates to the developer. This ranking brings specific lines of code to the developer's attention. This ranking also provides a comparison methodology between our technique and other automated debugging techniques, specifically the techniques developed by Liblit et al. [2005]. Both our method and Liblit et al. [2005] are then executed on the same version of GZIP or TCAS. Our method does require the history of other versions of a program, while the ranking algorithm developed by Liblit et al. [2005] does not require any history. While this seems to be a significant drawback to our methodology, it is not as significant as it seems. Generally in the industrial setting, product releases are large and then are followed by many months of incremental small changes, thus there is a significant amount of programmatic history available. Not only is this

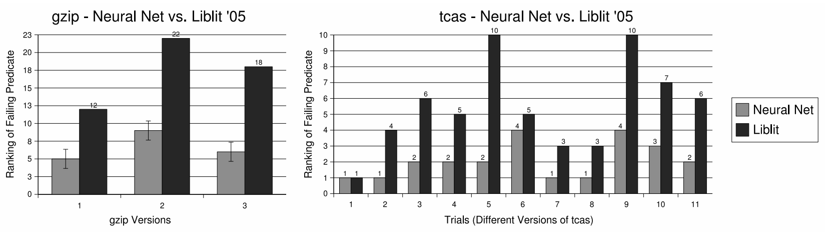

Fig. 2. Two graphs representing the number of lines a developer must analyze in order to identify the program fault.

history available but bugs are constantly being identified which could, under the correct infrastructure, be used to automatically train neural models that could be used to aid the debugging process of later programmatic faults.

5.5 Results

Following our experimental method we find that our method out-performs the ranking method developed by Liblit et al. [2005] (see Figure 2). We believe this is due to the ability of the neural network to be able to leverage specific information about the program domain. An example of such information is of previous versions that were labeled by their as non-failing portions, the neural network is able to learn from these previous identifications and thus will not report these predicates which are unrelated to the failures execution statistics. We believe that neural network training also is able to generate complex comparisons between the feature sets on a per program basis allowing relevant and pertinent features to stand out more prominently than generic mathematical models; nor would generic formulations be able to represent these program specific patterns well. This dynamic approach to model generation is best exemplified by the results obtained for the GZIP program (see Figure 0**??**). Our dynamic debugging model is able to consistently rank the faulty line as highly suspicious out of thousands of other lines of code; compared to the harmonic mean formula developed by Liblit et al. [2005], we rank the faulty line at least as high, and generally two to three times higher. The lines of code ranked above the fault in our ranking model are actually semi-associated with the fault, but do not lead the developer immediately to the line of code containing the fault.

6. CONCLUSIONS AND FUTURE RESEARCH

Using dynamically collected program statistics is an excellent way to reduce the problem space normally associated with software debugging. Based on our research and previous research, these statistics have strong predictive power that can be leveraged in different ways to aid a developer in locating programmatic errors, whether it is a deterministic or non-deterministic bug. Through using neural networks, we have shown that it is feasible to apply machine learning techniques to this statistical data, and by leveraging program specific model generation, we can identify more effectively the predicates associated with a program fault. The use of larger program data sets, such as GZIP, further strengthens the case for machine learning. Other mathematical models are not well-suited to handling effectively the extremely large number of predicates associated with medium to large programs. Training a machine learning model to handle a large number of predicates was validated as a possibility with experiments with GZIP based purely on historical execution data, even though the control and execution flow of the GZIP versions had changed slightly.

An interesting observation about our approach is that the amount of history required to make an effective training set for the machine learning algorithm is somewhat unexplored. A set of experiments exploring the history to correctly train a neural network, or equivalent algorithm, would have interesting results and would likely provide further insight into how to interpret predicate data. Some techniques have already been experimented with in order to incorporate a developer into the machine learning model training process. Our hope is that such a technique would benefit from the developer giving specific feedback during debugging sessions. In order for this to work two items must be addressed. First, more features need to be developed, in order for the network to make decisions more accurately. Secondly, training a neural network offline is not time constrained, while training a neural network with a developer in the training loop, must be done quickly to avoid perceived lag. This works toward the final goal of implementing these automated debugging techniques into an automated testing suite, where such a system would act like a unit testing framework except using unit tests to help developers automatically identify bugs.

7. ACKNOWLEDGEMENTS

I would like to thank Professor Alex Orso, my mentor throughout the research process, for his invaluable input and direction during the development of this research. I would also like to thank Professor Amanda Gable for her insightful input on organization and formatting of this thesis, and associated presentations. I would also like to thank all those that have taken the time to read and provide feedback on this thesis; making the concepts more tractable to those not within the field. Finally, I would like to thank the official readers of this thesis, Professor Alex Orso and Professor Merrick Furst, for their comments and corrections.

REFERENCES

Bowring, J., Orso, A., and Harrold, M. J. 2003. Monitoring deployed software using software tomography. *SIGSOFT Softw. Eng. Notes 28,* 1, 2–9.

Brat, G., Havelund, K., Park, S., and Visser, W. 2000. Model checking programs.

Cleve, H. and Zeller, A. 2005. Locating causes of program failures. In *ICSE '05: Proceedings of the 27th international conference on Software engineering.* 342–351.

Do, H., Elbaum, S. G., and Rothermel, G. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal 10,* 4, 405–435.

Liblit, B., Aiken, A., Zheng, A. X., and Jordan, M. I. 2003. Bug isolation via remote program sampling. *SIGPLAN Not. 38,* 5, 141–154.

Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I. 2005. Scalable statistical bug isolation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation.* ACM Press, New York, NY, USA, 15–26.

Liu, C., Fei, L., Yan, X., Han, J., and Midkiff, S. P. 2006. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering 32,* 10, 831–848.

Liu, C., Yan, X., Fei, L., Han, J., and Midkiff, S. P. 2005. Sober: statistical model-based bug localization. *SIGSOFT Softw. Eng. Notes 30,* 5, 286–295.

Mhaskar, H. N. and Micchelli, C. A. 1992. Approximation by superposition of sigmoidal and radial basis functions. *Adv. Appl. Math. 13,* 3, 350–373.

Orso, A., Apiwattanapong, T., and Harrold, M. J. 2003. Leveraging field data for impact analysis and regression testing. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering.* ACM Press, New York, NY, USA, 128–137.

Renieris, M. and Reiss, S. 2003. Fault localization with nearest neighbor queries.

Reps, T., Horwitz, S., and Sagiv, M. 1995. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM Press, New York, NY, USA, 49–61.

Zheng, A., Jordan, M., Liblit, B., and Aiken, A. 2003. Statistical debugging of sampled programs.

Zheng, A. X., Jordan, M. I., Liblit, B., Naik, M., and Aiken, A. 2006. Statistical debugging: simultaneous identification of multiple bugs. In *ICML '06: Proceedings of the 23rd international conference on Machine learning.* ACM Press, New York, NY, USA, 1105–1112.