SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **00**:1–21
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Spectral Based Fault Localization Using Hyperbolic Function

Neelofar*, Lee Naish and Kotagiri Ramamohanarao

Department of Computing and Information Systems, University of Melbourne, Melbourne, Australia

SUMMARY

Debugging is crucial for producing reliable software. One of the effective bug localization techniques is Spectral-Based Fault Localization (SBFL). It tries to locate a buggy statement by applying an evaluation metric to program spectra and ranking program components on the basis of the score it computes. Here, we propose a restricted class of "hyperbolic" metrics, with a small number of numeric parameters. This class of functions is based on past theoretical and empirical results. We show that optimization methods such as genetic programming and simulated annealing can reliably discover effective metrics over a wide range of datasets of program spectra. We evaluate the performance for both real programs and model programs with single bugs, multiple bugs, "deterministic" bugs and non-deterministic bugs and find that the proposed class of metrics performs as well or better than the previous best performing metrics over a broad range of data.

Received . . .

KEY WORDS: spectral debugging, dynamic analysis, fault localization, deterministic bugs, non-deterministic bugs, optimization methods

1. INTRODUCTION

Debugging software is a very important and resource intensive task in software engineering, with 50% to 80% of software development and maintenance costs attributed to bug fixes [1]. Debugging requires fault localization at the initial stage. This is a tedious process, requiring substantial manual work. Therefore, automatic fault localization techniques that can guide developers to a fault location are in high demand. Due to this demand many unique and creative techniques and processes have been proposed and automatic fault localization is an active field of research in academia and industry.

There are many techniques proposed for fault localization and debugging in the literature [2]. Spectral based fault localization (SBFL) techniques [3][4][5][6] have gained much popularity in the last few years due to their simplicity. These methods extract program spectra, that is execution profiles of program components (statements, predicates, functions etc.) and information on whether tests pass or fail. The data is used with "risk evaluation" metrics to rank the program components according to how likely they are to be buggy.

SBFL is not only of research interest but has potentially many practical applications in industry [7][8]. For instance, Tarantula (a tool developed by the Spider Lab at University of California, uses the pass/fail statuses of test cases and the events that occurred during execution of each test case to offer the developer recommendations of what faults may be causing test-case failures [3]), GZoltar (a framework for automating the testing and debugging phases of the software development life-cycle. The framework is provided as a library for developers and researchers and as an Eclipse plug-in that integrates seamlessly with JUnit tests [9]), The Cooperative Bug Isolation (CBI) (collects a

*Correspondence to: Neelofar, Department of Computing and Information Systems, University of Melbourne, Melbourne, Australia

Prepared using speauth.cls *[Version: 2010/05/13 v3.00]*

form of program spectra using low-overhead sampling, making it feasible for end users of programs to help collect information that is later used for debugging [10]) and Ample (provided as an Eclipse plug-in to identify faulty classes in Java programs [11]) etc.

Performance of SBFL methods critically depend on the metrics used. Over one hundred metrics have been developed manually and evaluated for SBFL [12]. More recently, genetic programming has been used to automatically develop metrics [13] [14], resulting in many thousands being evaluated. For programs with a single bug, SBFL is relatively easy and we now have a good theoretical understanding [6][15][16]. The same is true for locating bugs which are "deterministic" (cause test case to fail whenever they are executed) [17]. However, the general case which is studied in this research, where there are multiple bugs that may not be deterministic is much more challenging. Most SBFL research to date has had a strong bias towards single bug programs, partly due to the available datasets used for evaluation, and tackling the general case is an important priority. We propose a new class of "hyperbolic" metrics which have a small number of numeric parameters whose values can be adjusted to vary the behaviour. Depending on the parameter values, hyperbolic metrics can be optimal for single bugs, optimal for deterministic bugs or similar to other metrics known to perform well for some multiple non-deterministic bug benchmarks.

This paper is a substantial extension of our previous work [18] with the following additions:

- In our previous work, we used the Siemens Test Suite (STS) [19] and Model data [6] for our experiments. STS contains seeded bugs and size of programs are very small. In order to further validate our results, we have extended our experiments to larger datasets including Space, Flex, Grep, Sed, and Gzip.
- We use multiple optimization methods for learning parameter values for our hyperbolic class of metrics. Although metrics learned by different methods are similar, learning time with various methods can vary greatly and we show a significant improvement over previous work.
- We introduce a statement pruning technique which decreases the size of training data, resulting in decreasing the learning time considerably with little or no decrease in performance, thus scaling the technique to large programs.
- In order to keep scores computed by a metric within a certain range, scaling or normalization of input values is required. Scaling affects overall performance of hyperbolic metrics. Here we discuss different scaling methods and their performance effects.
- The class of metrics introduced in our previous work did not perform as well as expected on some model programs due to the difficulty of choosing appropriate parameter values[6]. We introduce a new class of hyperbolic metrics that is equivalent to the previous class but it designed to make learning easier, improving its performance.

The rest of the paper is structured as follows. Section 2 provides background information on SBFL (some material is based on [17] without additional citation), including theoretical results which motivate our approach, and gives the definitions of selected metrics from the literature that we use for comparison purpose. Section 3 motivates and defines the class of hyperbolic metrics and Section 4 briefly describes how we use different optimization methods to learn parameter values. Section 6 describes experiments and their results, using spectral data from both model programs and real programs while Section 7 discusses statistical significance of these results. Section 5 describes the statement pruning method we used with machine learning to improve learning time of hyperbolic metrics. Section 9 briefly reviews some additional related work, Section 10 suggests some future directions, and Section 11 concludes.

2. DEFINITIONS AND NOTATIONS

SBFL methods use a set of tests, each classified as failed or passed; this can be represented as a binary vector, where 1 indicates failed and 0 indicates passed. For statement spectra [5, 6, 15, 17, 20, 21, 22] which we use here, we gather data on whether each statement is executed or not for each test. This can be represented as a binary matrix with a row for each statement and a column for each test; 1 means executed and 0 means not executed. For each statement, four a_{ij}

DOI: 10.1002/spe

Table I. Statement spectra with tests $T_1 \ldots T_5$

	T_1	T_2	T_3	T_4	T_5	ef	ep
S_1	1	0	0	1	0	1	1
S_2	1	1	0	1	0	2	1
S_3	1	1	1	0	1	2	2
			⋮				
Res.	1	1	0	0	0	$F=2$	$P=3$

values are ultimately produced, $i \in \{n, e\}$ *and* $j \in \{p, f\}$. They are the number of passed/failed test cases in which the statement was/wasn't executed — $\langle ef, ep, nf, np \rangle$, where the first letter indicates whether the statement was executed (e) or not (n) and the second indicates whether the test passed (p) or failed (f). We use F and P to denote the total number of tests which fail and pass, respectively while E is number of times a statement is executed. Clearly, $nf = F - ef$ and $np = P - ep$ and it is sometimes convenient to use F, P, ef (or nf) and ep (or np) rather than all four $\{n, e\} \times \{p, f\}$ values. For fixed F and P there are just two degrees of freedom. Table I gives an example binary matrix of execution data and binary vector containing the test results. This data allows us to compute F, P and all the $\{n, e\} \times \{p, f\}$ values.

Metrics, which are numeric functions, can be used to rank the statements. Most commonly they are defined in terms of the four $\{n, e\} \times \{p, f\}$ values. Statements with the highest metric values are considered the most likely to be buggy. We would expect buggy statements to generally have relatively high ef values and relatively low ep. Table II gives definitions of the established metrics used here and cites where they were first used for SBFL. More than 150 metrics are evaluated in [23]; our chosen selection is justified in section 6.1.2.

We refer three types of bugs in this work – single, deterministic and multiple. Single bug program is where all the failures are caused by just one bug. Whenever a test case fails, buggy statement should have executed in that test case. Sometimes bugs are in preprocessor directives (#define) which are called at multiple places in the program. Such programs wouldn't be single bug according to the single bug definition in [6], which we follow here. Deterministic bugs are defined as "Bugs which cause test case to fail whenever they are executed" in [17]. There are some other aspects of determinism in literature like non-deterministic behaviour of programs and intermittent bugs as discussed in [16], however we use the definition of deterministic bugs in [17] in this work. Multiple bug situation is when there are more than one bugs causing program to fail and failure of different test cases can be due to different bugs.

Programmers searching for a bug are expected to examine statements, starting from the highest-ranked statement, until a buggy statement is found. In reality, programmers may well modify the ranking due to other considerations, and checking correctness generally cannot be done by a single statement at a time. Evaluation of different ranking methods generally ignores such refinement and just depends on where the bug(s) appear in the ranking. Here we use two common measures for the evaluation of our technique: "Rank Percentage" and "Successful Diagnosis". *Rank percentage* is the rank of the top-ranked bug, expressed as a percentage of the program size. Results are averaged over all buggy programs in the benchmark set. *Successful Diagnosis* is the number of bugs which are detected by analyzing top n% of code [27][28]. We have evaluated our technique at 1%, 2%, 5%, 10%, 20% and 50% of code. Statements which are not executed in any test are ignored.

3. THE HYPERBOLIC METRIC CLASS

This section motivates and defines the class of "hyperbolic" metrics we propose. Rather than a fixed formula, we use a parametrized hyperbolic metric (equation 1). The general idea is to be able to choose parameter values so the resulting formula performs well for the chosen dataset. Later we

DOI: 10.1002/spe

O^p [6]	$ef - \frac{ep}{ep+np+1}$
O^d [17]	$ep - \frac{ef}{ef+nf+1}$
Zoltar [24]	$\frac{ef}{ef+nf+ep+\frac{10000nf\times ep}{ef}}$
Kulczynski1 [22]	$\frac{aef}{anf+aep}$
Kulczynski2 [22]	$\frac{1}{2}\left(\frac{ef}{ef+nf}+\frac{ef}{ef+ep}\right)$
Ochiai [5]	$\frac{ef}{\sqrt{(ef+nf)(ef+ep)}}$
Ample [25]	$\lvert\frac{ef}{ef\times ep}-\frac{ep}{ep\times np}\rvert$
Wong1 [21]	ef
Wong2 [21]	$ef - ep$
Tarantula [3]	$\frac{\frac{ef}{ef+nf}}{\frac{ef}{ef+nf}+\frac{ep}{ep+np}}$
Hyperbolic [18]	$\frac{1}{K_1+\frac{nf}{nf+ef}}+\frac{K_3}{K_2+\frac{ep}{ep+ef}}$
GP13 [14]	$ef(1+\frac{1}{2ep+ef})$
Information Gain [26]	$(-F\log F - P\log P)$ $-(ef+ep)\times(-\frac{ef}{ef+ep}\log\frac{ef}{ef+ep}-\frac{ep}{ef+ep}\log\frac{ep}{ef+ep})$ $-(nf+np)\times(-\frac{nf}{nf+np}\log\frac{nf}{nf+np}-\frac{np}{nf+np}\log\frac{np}{nf+np})$

Table II. Formulas for several risk evaluation metrics

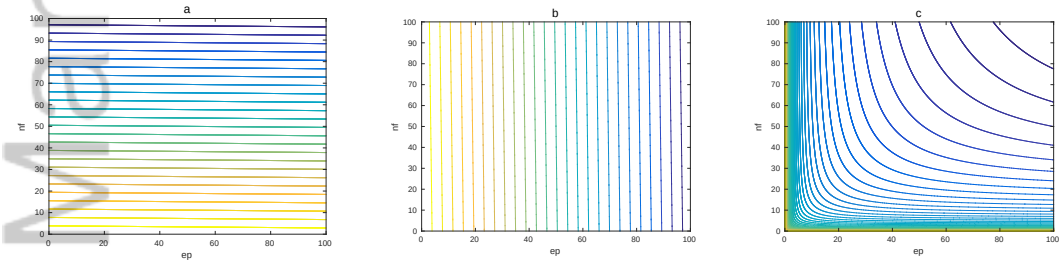

Figure 1. Contour plots for O^p (a), O^d(b) and (Hyperbola) $\frac{1}{nf}+\frac{1}{ep}$ (c).

describe how we use optimization methods to choose the parameter values. Recent research uses genetic programming to construct metrics [14]. However, we failed to reproduce the results reliably in our experiments and found that it is significantly harder to generate metrics which perform well in general. However, using machine learning to find a small number of numeric parameters rather than a complete formula makes the learning task much simpler, efficient and reliable.

3.1. Motivation for Hyperbolic Metrics

Most evaluation of SBFL has a strong bias to programs with a single bug. This problem is now reasonably well understood [16] [6] [17]. O^p was proposed in [6] and proven optimal with respect to a single bug "model" program. This optimality result was strengthened in [15] to arbitrary single bug programs and test sets by restricting attention to "strictly rational" metrics, which are those whose value strictly increases in ef when ep is fixed and strictly decreases in ep (or increases in np) when ef is fixed. Metrics such as O^p which rank statements primarily on their ef value and use np to break ties when the ef values are equal are single bug optimal. In [17] it was shown that metrics such as O^d which rank statement primarily on np and break ties using ef are optimal for programs with only deterministic bugs (which cause failure whenever they are executed).

Between these two extreme cases we may have multiple bugs, some of which are not deterministic, which is the case in large software systems. There is a little theoretical understanding

DOI: 10.1002/spe

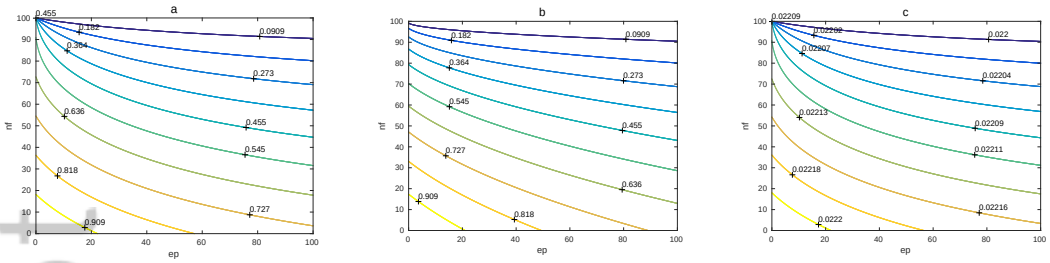

Figure 2. Contour plots for Kulczynski2 (a), Ochiai (b) and hyperbolic metrics(c).

of this more general case and constructing metrics which perform well is a challenging problem. Any given bug is not always executed in all failed tests and not all tests in which it is executed will fail. Thus, no strong relationship between a particular fault and a failure can be established. Metrics which are derived for or perform well for the single bug (or deterministic bug) case often do not perform well in the general case. Although we can not find any metric in literature which is considered as optimal for multi-bug datasets, there are two metrics (Ochiai and Kulczynski2) which are shown to perform better than other metrics on few multi-bug datasets in [22].

In [17] it was noted that plotting the contours of a metric over a rectangular domain ($F \times P$) gives useful insights. There is a one to one relationship between contours and the rankings produced by a metric. If we plot metric contours using ep and nf as the axes then strictly rational metrics have contours with finite strictly negative gradients, with the contours close to $(0, 0)$ corresponding to the highest metric values. If all gradients are sufficiently close to zero the metric is single bug optminal, like O^p, whereas if all gradients are sufficiently large negative numbers the metric is deterministic bug optminal, like O^d — see Figure 1a and Figure 1b which are the contour plots of O^p and O^d respectively with ep as x-axis and nf as Y-axis. Figure 2a and Figure 2b show the similar contours for Ochiai and Kulczynski2. Unlike O^p and O^d, these contours are curvy.

We believe that any metric having contours a combination of all three of O^p, O^d and Ochiai/Kulczynski2 would perform well for all the above mentioned cases. Figure 1(c) shows contours which are hyperbolas ($\frac{1}{nf} + \frac{1}{ep}$) with negative gradients. The contours at the bottom right of this plot are like those of O^p whereas those at the top left are like those of O^d. Thus by selecting part of the domain, hyperbolas can adopt to either of the extremes. Between these extremes, the gradients of the hyperbolic contours increase (get closer to zero from negative value) as ep increases. At the bottom left there is a very sharp increase and as we proceed up and right the increase is more gradual. Ochiai and Kulczynski2 also have similar trend for selected parts of the domain — see Figure 2a, Figure 2b and Figure 2c. Thus the motivation for using metrics with hyperbolic contours is they can be optimal in the two extreme cases we have theoretical results for, and similar to the best metrics we know of in other cases. The hyperbolic formula shown in Figure 1(c) is a specific instance of hyperbolic metrics shown in equation 1. The parameters in Equation 1 allows adjustments according to the data (the typical number of bugs and how deterministic they are). Details are given in next section.

3.2. Hyperbolic Metrics

We now define the parametrized hyperbolic function. There are four adjustments that we make to the simple formula of $\frac{1}{nf} + \frac{1}{ep}$ of Figure 1(c) to have desired properties. The first is to scale both nf and ep values to the range 0–1. The aim of this is to help limit the range of values for the parameters introduced in the next steps. Different forms of scaling which we used in our experiments are discussed in section 3.3. The second adjustment is to add a parameter K_1 to the scaled nf value. A positive K_1 makes contours steeper as in Figure 1(a) and a large enough value results in a deterministic bug optimal metric if other things remain unchanged. The third adjustment is to add a parameter K_2 to the scaled ep value. A positive K_2 makes contours flatter as in Figure 1(b) and a large enough value results in a single bug optimal metric if other things remain unchanged (that

Prepared using speauth.cls
DOI: 10.1002/spe

Figure 3. Different scaling examples for hyperbolic metrics. (a) $nf \rightarrow \frac{nf}{F}$, $ep \rightarrow \frac{ep}{P}$ (b) $nf \rightarrow \frac{nf}{T}$, $ep \rightarrow \frac{ep}{T}$ (c) $nf \rightarrow \frac{nf}{F}$, $ep \rightarrow \frac{ep}{E}$

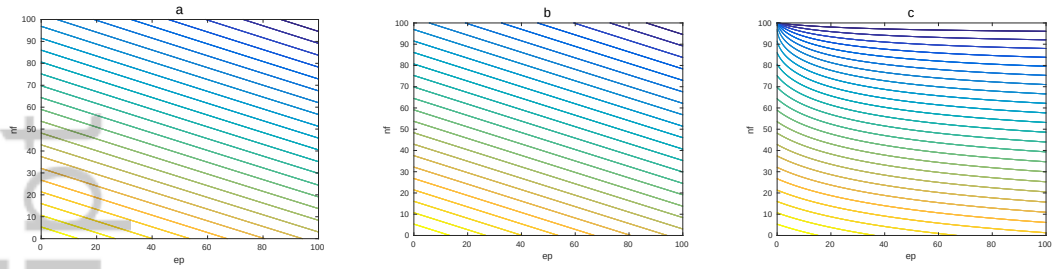

Table III. Average Rank Percentages of various scaling forms of Hyperbolic Metrics on different datasets

	$\frac{1}{K_1+\frac{nf}{F}}+\frac{K_3}{K_2+\frac{ep}{P}}$	$\frac{1}{K_1+\frac{nf}{T}}+\frac{K_3}{K_2+\frac{ep}{T}}$	$\frac{1}{K_1+\frac{nf}{F}}+\frac{K_3}{K_2+\frac{ep}{E}}$
Siemens Test Suite	20.23	23.83	20.32
Space	5.48	9.74	2.50
Flex, Gzip, Grep, Sed	6.37	6.05	5.98

is, K_1 is relatively small). If K_1 and K_2 are small, contour gradients change abruptly whereas if they are both large the gradients change slowly. Lastly, we multiply the ep term by parameter K_3. This allows us to effectively stretch or compress the contours in a horizontal direction. With large identical K_1 and K_2 values the contours are close to straight lines, but their gradient can be adjusted using K_3. In short, the first adjustment is for scaling, while the second, third and fourth are for making contours flatter or steeper like single bug optimal or deterministic bug optimal respectively. The overall formula for our class of hyperbolic metrics is as follows:

$$\frac{1}{K_1 + \frac{nf}{F}} + \frac{K_3}{K_2 + \frac{ep}{E}} \tag{1}$$

3.3. Scaling

As stated above, scaling is required to limit the range of possible values for input to the hyperbolic metrics. There are several ways scaling can be done. In our experiments we used hyperbolic metrics with three forms of scaled versions. The first is to divide nf and ep by F and P respectively—Figure 3a. This results in metrics for which the number of passed and failed tests can be scaled separately without affecting the ranking, called "general scalable" in [12]. The second scaling method is to divide both nf and ep by total number of test cases $T = F + P$—Figure 3b; this is also general scalable. The third scaling method that we used in our experiments is to divide nf by F and ep by E (number of times a statement is executed i.e $ef + ep$) —Figure 3c. This form of scaling produces "uniform scalable" metrics, [12] for which the scaling of passed and failed tests must be the same in order to ensure the same ranking. The class still has the desired properties (a combination of single bug optimal, deterministic bug optimal and Ochiai or Kulczynski2 like contours). We experimented with all the three scaling methods on different datasets and found that the third (only uniform scalable) form of hyperbolic metrics perform consistently better than other forms as shown in Table III. Although difference between $\frac{ep}{E}$ and $\frac{ep}{P}$ forms is not very obvious in the Siemens Test Suite, for other datasets this difference is quite noticeable. Thus our choice of scaling method is based on performance for datasets and inspiration from other metrics like Kulczynski2 and Ochiai which uses same scaling rather than theoretical analysis.

DOI: 10.1002/spe

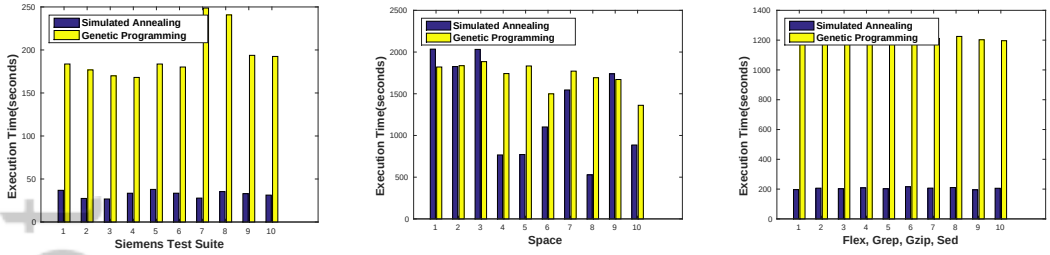

Figure 4. Comparison of Execution Time of Genetic Programming and Simulated Annealing on Different Datasets.

4. FINDING PARAMETER VALUES USING OPTIMIZATION METHODS

We use different optimization methods such as Genetic Programming and Simulated Annealing to find parameter values for hyperbolic metrics. The details are given in sections below.

4.1. Finding Parameter Values Using Genetic Programming

Yoo el al. propose to use genetic programming to generate metrics directly from spectral data [14]. In our efforts to reproduce the results, we found that the results are not consistently good, especially in the case of multiple bugs. The search space is huge and the choice of GP operators often makes it hard to gradually improve the metrics. For example, subtracting a good metric from another good metric can often results in a very bad metric. Even our attempts to constrain the search to more sensible metrics, such as those which are strictly rational [15], were not particularly successful. In contrast, learning three numeric parameter values is a much simpler task, and a small change in parameter values generally results in a relatively small change in metric performance — the objective function is more "smooth" in some sense. We used the genetic programming software to learn the hyperbolic metric parameter values K_1, K_2 and K_3 from training data and applied the resulting metric to localize bugs.

4.2. Parameter Values using Simulated Annealing

Genetic Programming is a complex technique, starting with a huge population and selecting the best solution from different generations. It is very costly in terms of time for a complex data like program spectra. As our goal is not to find different metrics from spectral data, but finding the best parameter values, comparatively simpler optimization methods could perform equally well. We computed K_1, K_2 and K_3 values using simulated annealing and found that results are as good as with genetic programming but requiring substantial less computing time.

As stated earlier, although Genetic Programming and Simulate Annealing learn metric parameters equally effectively in our experiments, Simulated Annealing computes results much faster than Genetic Programming. A comparison of execution time of both of these methods on different datasets is given in Figure 4. Except for a few runs in the Space experiments, simulated annealing took much less time to learn parameter values as compared to genetic programming. The difference is huge (about six times less) for Flex, Grep, Gzip, Sed and most runs of Space. We further reduce the training time taken by simulated annealing to learn parameter values by introducing a statement pruning technique. Details are provided in section 5.

4.3. Time Complexity

Our technique consists of algorithm learning using training data and actual fault localization using trained algorithm. Time complexity of the training depends on the optimization algorithm used. We have used simulated annealing and genetic programming in our experiments. Testing/actual fault localization consists of two phases. First is to compute the score of each statement using hyperbolic metrics. We are assuming that test data have the execution frequency of the statements

Prepared using speauth.cls
DOI: 10.1002/spe

i.e ep, ef, np, nf values. Let's suppose there are M statements in the program, the cost of giving score to every statement is $O(M)$. These statements are then sorted in ascending order $O(M.logM)$. Thus overall complexity of the fault localization phase is $O(M + MlogM)$.

5. SIMULATED ANNEALING WITH STATEMENT PRUNING

There are many fault localization techniques available in literature using machine learning methods like clustering [29] [30], decision trees [31], frequent pattern mining [32], support vector machine [33], and genetic programming [14][13]. Compared to using fault localization metrics directly to localize faults, these approaches are time intensive and require training of optimization method using training data. For smaller datasets like the Siemens Test Suite, this time is reasonable depending on the optimization technique used, however, for larger datasets like Space, Flex, Grep, Gzip or Sed having more than 10,000 statements on average, training is much more time consuming. Although in all the studies cited above, comparison in terms of improvement in the performance is given, they do not discuss extra time required in training of their algorithms. We use Simulated Annealing and Genetic Programming to find hyperbolic metrics parameter values, and acknowledge that training phase takes much more time than actual fault localization.

Fault localization metrics rank statements on the basis of higher ef and lower ep. It constitutes the bottom left area in Figure 2a, Figure 2b and Figure 2c. These statements correspond to the top of the ranking and are thus the most important for performance of a metric. On the other hand, statements having very low ef or very high ep are ranked at bottom and thus play a minimum role in localizing faults. There are huge number of such statements and contribute a substantial amount of time in training. By removing these statements from training data, training can be done more efficiently. We propose a *Statement Pruning* technique that removes a particular percentage of low ranked statements having minimum or no impact on fault localization during learning phase, thus decreasing the size of training data. Our technique is explained in algorithm 1, where low ranked statements are pruned in different iterations of simulated annealing. The algorithm is standard simulated annealing with an addition of *Statement Pruning* method called at line 14. Variables are initialized from line 1 to 6 and explained in comments. No pruning is performed in initial phases of learning to allow *initial burn-in* of the algorithm. Once algorithm achieves stability in terms of finding better values for parameters that lowers the rank percentage, bottom ranked statements are pruned (Line 19). The pruning is performed in lines 20 – 24. It removes bottom n% statements and replaces current training data with pruned data. The process continues until pruning percentage is less than maximum pruning percentage allowed. Thus only in first few iterations of simulated annealing, algorithm would use whole dataset for training. Later on, every time pruning is performed, size of training data decreases for further iterations.

Table IV shows the affect of statements pruning on execution time and performance of hyperbolic metrics on various datasets. The first column of the table shows the percentage of code removed by pruning, while the first row shows different datasets on which the pruning technique is applied. The "Performance" column under each dataset shows the percentage decrease in performance in terms of rank percentage while "Exec-Time" shows percentage decrease in execution time.

Execution time nearly always decreases with a decrease in number of statements except in case of 20% code reduction for Flex/Grep/Gzip/Sed data. The reason why pruning statements can increase in execution time is explained next. The pruning method is called many times in the algorithm until the pruning size reaches the maximum allowed pruning percentage, as shown at line 13 of algorithm 1. In each pruning round, all the statements are sorted again on the basis of their scores, the bottom n% statements are pruned and the new pruned data replaces old data. These operations take time, particularly sorting statements when there are a large number of them such as in Flex, with more than 10,000 statements. This additional work can outweigh the time saved elsewhere. However, with a larger percentage of statements pruned, the additional time required for sorting is much lower compared to the time saved in learning and the total time is decreased by almost half or more.

DOI: 10.1002/spe

One may expect performance to either decrease or remain constant with statement pruning but we can see exceptions in table IV due to degree of randomness in learning. For space and the Siemens Test Suite, execution time decreases more than 7.5% and 4.5% respectively with only 1.2% and 0.2% decrease in performance. By pruning almost 70% of the bottom ranked statements, execution time decreases by half on average, with a reduction of only 1.1%, 3.6% and 0.7% performance for Flex/Grep/Gzip/Sed, Space and the Siemens Test Suite respectively. Thus, with a very little compromise with the performance, our pruning method decreases execution time noticeably.

Algorithm 1 Finding Hyperbolic Metrics Using Simulated Annealing with Statement Pruning

Input: Training Data
Output: Best Parameter Values for Hyperbolic Metrics(k1, k2, k3)

procedure Anneal($training\_data$)
  $initial\_data\_size \leftarrow init\_size$ ▷ number of statements in training data
  $initial\_temperature \leftarrow init\_t$ ▷ starting annealing temperature
  $minimum\_temperature \leftarrow min\_t$ ▷ minimum annealing temperature
  $maximum\_pruning\_allowed \leftarrow max\_pruning$ ▷ %age of statements allowed to be pruned
  $initial\_burnin\_iterations \leftarrow init\_burnin$ ▷ minimum number of annealing iterations only after which pruning is allowed
  while $initial\_temperature \geq minimum\_temperature$ **do**
    $compute\ k1,\ k2,\ k3\ values\ using\ optimisation\ method$
    if $temperature\_iteration > initial\_burnin\_iterations$ **then**
      $current\_data\_size \leftarrow count\ total\ number\ of\ statements\ in\ training\_data$
      $statements\_pruned \leftarrow initial\_data\_size - current\_data\_size$
      $pruning\_percentage\_applied \leftarrow 100 - (\frac{statements\_pruned}{initial\_data\_size} \times 100)$
      if $pruning\_percentage\_applied < pruning\_percentage\_allowed$ **then**
        pruneBottomStmts(training_data) ▷ calling of statement pruning method
      end if
    end if
  end while
  $decrease\ initial\_temperature\ by\ a\ fixed\ value$
end procedure

▷ Prune bottom ranked statements if pruning percentage is less than maximum pruning percentage allowed
procedure pruneBottomStmts($statements\_list$)
  $Get\ list\ of\ statements\ and\ sort\ in\ descending\ order\ of\ their\ scores$
  $Remove\ bottom\ n\%\ of\ statements$
  $Replace\ training\ data\ with\ pruned\ data$
end procedure

Table IV. Decrease in Execution Time of Hyperbolic Learning with Statements Pruning

	Flex, Grep, Gzip, Sed		Space		STS	
%age code removed	Performance	Exec-Time	Performance	Exec-Time	Performance	Exec-Time
20	0.11	-7.82	-1.26	7.69	-0.26	4.76
43	-0.15	22.18	1.07	34.25	-0.13	24.26
55	-0.87	39.17	0.18	47.01	-1.73	37.50
70	-1.11	52.73	-3.66	62.06	-0.78	48.99

6. EXPERIMENTAL SETUP AND RESULTS

6.1. Experimental Setup

DOI: 10.1002/spe

6.1.1. Dataset To evaluate our technique, we used data from model programs as described in [6] and real data from Software-artifact Infrastructure Repositories (SIR)[19].

Model/synthetic data have many advantages in using for spectral fault localization study: we can control the number of passed/failed test cases, change the bug consistency and thus generate deterministic bug data for which no real benchmark dataset is available, control number of bugs and generate as much as needed. Furthermore, [6] shows quite a good fit between model data and the Siemens Test Suite as far as various metrics' performance is concerned.

SIR is a repository of software-related artifacts meant to support experimentation with program analysis and software testing techniques. Benchmark datasets used in our experiments are the Siemens Test Suite, Space, Flex, Gzip, Grep and Sed. The Siemens Test Suite contains 7 programs in total: Printtokens, Printtokens2, Tcas, Replace, Totinfo, Schedule and Schedule2. For Printtokens2 and Schedule2, there are very few versions available in repository and most contain bugs introduced either by deletion of a statement or commenting a statement. SBFL uses execution profiles of statements to localize bugs. As deleted or commented statements are not executed, there is no associated spectra. One way to evaluate a SBFL method for such programs is to replace the statement by an empty/non-operational statement, but due to very limited number of versions, we ignored these programs instead of editing and using them.

The Siemens Test Suite contains small programs having 563, 411, 173, 406 and 563 lines of code in Replace, Schedule, Tcas, Totinfo and Printtokens, respectively. While Space, Flex, Grep, Gzip and Sed are larger programs having 9126, 11784, 10929, 6357 and 8059 statements, respectively. SIR contains single bug versions for all the programs. In order to generate two or multiple bug versions, we combined two or more single bug versions and tested the resulting program against the provided test suite to generate spectra. We generated 545 two bug versions of Siemens Test Suite programs, 248 two bug versions of space, 92 single bug version of Flex, Grep, Gzip and Sed and 93 single bug versions of Siemens Test Suite programs. Any version having a buggy statement that is not executed by any failed test case is ignored. We use ten-fold cross validation for training and testing purposes.

6.1.2. Fault Localization Metrics Used In our experiments here we include optimal metrics for the two extreme cases, O^p and O^d. Zoltar is close to optimal for single bug programs and also performs well for multiple bug programs [22]. $GP13$ produces the same ranking for single bug programs as O^p [14]. Kulczynski2 and Ochiai perform extremely well for the multiple-bug datasets of [22] which we use here. A version of Ochiai is proved to be the best based on benchmarks from the Software-artefact Infrastructure Repository(SIR) [23]. Information Gain is shown to outperform Ochiai for some datasets [26]. Tarantula was the first metric used for SBFL and though it performs relatively poorly for many datasets, a small adjustment makes it optimal for deterministic bugs [17]. Some other known metrics like Wong1, Wong2 and Ample are also included to study their effectiveness in bug localization.

6.1.3. Genetic Programming and Simulated Annealing Configuration For genetic programming experiments, we used JGAP — an open source Java based framework. There are many ways in which the learning can be adjusted. For the results presented here, we use a population size of 1000 and stop after 100 generations. GP is configured with a mutation operator with the rate of 0.9.

The three parameters of the hyperbolic metrics are used as terminal symbols and we fix the range of K_1 and K_2 to be 0–100 and K_3 to be 0–2. The GP operators increase and decrease the parameters within these ranges. We found that for different runs on the same data there is quite a wide variation of parameter values found, though the ratio of K_1 and K_2 remains stable.

For Simulated Annealing results presented in this paper, we use the temperature cooling range from 1.0 to 0.00001, with a decrement rate of α. Starting initially with a large α, we decrease its value with each temperature iteration to search more extensively around the best solution found so far in the current iteration. Like genetic programming, we fixed the range of K_1 and K_2 to be 0–100 and K_3 to be 0–2.

Prepared using speauth.cls
DOI: 10.1002/spe

Table V. Average Rank Percentages for Models With Different Bug Consistencies

# of bugs	Con*	O^p	O^d	Ochiai	Zoltar	Kulc2	Ample	Wong1	Wong2	Tarantula	Hyp**
1	20%	**26.130**	48.854	26.641	**26.130**	26.204	29.736	26.468	35.361	30.220	**26.130**
2	20%	28.968	34.933	29.000	28.968	28.968	33.373	31.326	32.353	30.129	**28.963**
2	40%	28.483	30.357	28.119	28.433	28.363	31.003	31.609	29.464	28.307	**27.981**
2	60%	28.062	27.040	27.080	27.837	27.601	28.714	31.982	27.365	26.769	**26.510**
2	80%	27.913	25.573	26.011	27.011	26.500	26.843	32.434	26.417	25.822	**25.568**
2	100%	29.530	**26.736**	26.746	26.745	26.746	26.954	33.003	26.738	30.500	**26.736**

*Bug Consistency
**Hyperbolic Metrics

6.2. *Experimental Results*

6.2.1. Model program experiments The model-based approach is explained in detail in [6]. An advantage of using models is that large synthetic datasets with precisely controlled parameters can be generated. Here we use six different model programs, each with four statements, where execution of correct statements is statistically independent of test case failure but execution of buggy statements is correlated to varying degrees. In the first model, only the first statement is a bug (so O^p is optimal) and execution of the bug leads to test case failure 20% of the time. In models 2–6 the first two statements are buggy, each of which cause failure in 20%, 40%, 60%, 80% and 100% of cases where they are executed, respectively (thus the last model has only deterministic bugs and O^d is optimal). The first two statements are modelled using ten execution paths, five of which execute the statement and a number of those lead to failure, dependent on the model. The other two statements are each modelled using just two execution paths, one of which executes the statement. The models are designed so the relative discrimination of ef versus ep drops as we go from model 1–6, and this affects what metric is best to use for each of these models. For our experiments we used 10,000,000 test sets, each with 15 passed and 5 failed tests. Learning was done on 100,000 instances.

Table V shows the comparison of average rank percentage for learned hyperbolic metrics and previously established metrics on model data described above. The class of hyperbolic metrics introduced in our previous work [18] did not perform as well as the best existing metrics for bug consistencies of 20% and 80%. For the results in Table V, we introduce a new form of hyperbolic metrics which is equivalent to the previous class, but modified to make learning easier (equation 2). In this new form there is a single parameter K_1, with range (0–1), that can control the overall gradient of the contours, from single bug optimal (almost horizontal contours) to deterministic bug optimal (almost vertical contours).

$$\frac{1}{K_1 \times K_2 + \frac{nf}{F}} + \frac{K_3}{(1-K_1) \times K_2 + \frac{ep}{E}} \tag{2}$$

The best results for each bug consistency are shown in bold. As expected, O^p and O^d perform best for the first and last models, respectively — these are known to be optimal. The hyperbolic metrics match this optimal performance. They also do better than all other metrics evaluated for the models with 20%, 40%, 60% and 80% bug consistencies. The previous less than best performance for 20% and 80% cases was due to the difficulty of learning an optimal combination of the three parameter values. We believe learning is made easier by having a single parameter that has the main influence on performance, with the other two parameters allowing refinements.

6.2.2. Single Bug Experiments We use single bug data from Siemens Test Suite, Space, Flex, Grep, Gzip and Sed. Tables VI, VII and VIII show comparison of average rank percentages of various metrics on single bug data using Genetic Programming and Simulated Annealing.

O^p has been theoretically and empirically proven to be the best metric for single bug data [6] while [14] shows that $GP13$ is equivalent to O^p for such programs. As shown in Table VI, these two metrics have lowest average rank percentage for single bug Siemens Test Suite while hyperbolic metrics perform almost equivalent to them on average.

DOI: 10.1002/spe

For Space, we have only eighteen single bug versions available which are not quite enough for learning. However, using these limited number of versions, Table VIII shows that hyperbolic performs almost as good as O^p and $GP13$. We believe that using more training data, the slight difference of rank percentage between single bug best performing metrics and hyperbolic will vanish.

Unlike STS and Space, Table VII shows that hyperbolic performs better than O^p for data containing single bug versions of Flex, Grep, Gzip and Sed. The reason behind less than best performance of O^p for this dataset is explained next.

We have generated data for Siemens Test Suite by taking program versions and test cases from SIR and made sure that only those program versions are included which satisfy "Single Bug" definition according to [6]. For example, if a bug is introduced by #define in any program version, it will not be considered single bug if this definition is used multiple times in the program. However, most of the versions of Flex, Grep, Gzip and Sed in SIR do not satisfy above mentioned single bug definition and thus performance of O^p is slightly lower than hyperbolic.

Tables IX, XI and X show performance of various metrics in terms of successful diagnosis using 1%, 2%, 5%, 10%, 20% and 50% of code. Results show that hyperbolic performs as either the best or one of the best metric in most of the cases. By analyzing only 10% of the source code, hyperbolic along with O^p and $GP13$ diagnose 54.29% of bugs in STS. For space, this percentage is slightly lower than O^p and $GP13$ due to limited number of data for learning. For Flex, Grep, Gzip and Sed hyperbolic diagnoses 68.78% of bugs which is better than O^p and $GP13$ for the reason explained earlier.

In [16] it is shown that Barinel – a combination of spectral based and model based debugging – is better than Tarantula and Ochiai for single bug STS. The comparison is made based on successful diagnosis. By analyzing 10% of source code, Barinel detects 8% more bugs than Ochiai while 12% more than Tarantula on single bug STS. Although we haven't directly compared the performance of our technique with Barinel, Table IX shows that hyperbolic detects 8% and 14% more bugs than Ochiai and Tarantula respectively on single bug STS by analyzing only 10% of source code. We thus believe that hyperbolic metrics is as effective in single bug fault localization as Barinel.

In Summary, hyperbolic metrics perform as good as best performing metrics in single bug domain. Last column of these tables (p-*value*) is explained in detail in Section 7.

6.2.3. MultiBug Experiments Tables XII, XIII and XIV show comparison of average rank percentages of various metrics on multi bug data from Siemens Test Suite, Space, Grep, Flex, Gzip and Sed. The comparison includes hyperbolic metrics learnt by genetic programming as well as simulated annealing. Hyperbolic metrics have lowest average rank percentage for the STS and a combination of Flex, Grep, Gzip and Sed data. Although Ample is the best performing metric for Space, Average Rank Percentage of hyperbolic is quite close to it (5.69 vs. 5.99). Note that [22] evaluated performance of over 80 metrics for two and three bug versions of the Siemens test suite and Unix benchmarks and showed that Kulczynski2 and Ochiai are the best performing metrics. It can be noted that hyperbolic outperforms these two metrics in many cross validation runs as well as on average, which is an impressive achievement. On datasets where hyperbolic is not the best performing metric, p-*value* shows that the difference between the rank percentage of hyperbolic and best performing metric is not statistically significant that means hyperbolic is neither better nor worse in performance compared to these metrics.

Tables XV, XVI and XVII show percentage of successful bug diagnosis of various metrics by analyzing 1%, 2%, 5%, 10%, 20% and 50% code. It can be seen that hyperbolic metrics detect more than 45% of bugs by analyzing less than 10% of source code. Although it does not perform the best for all the percentages of code analyzed, it out performs all other metrics in most of the cases. In other cases, its performance is quite close to the best performing metrics.

DOI: 10.1002/spe

Table VI. Comparison of Average Rank Percentages of various metrics for single bug Siemens Test Suite

Run #	O^d	Tarantula	Hyp_sa	Hyp_gp	O^p	Ochiai	Zoltar	Kul2	Kul1	Wong1	Wong2	Ample	GP13	Information Gain
1	47.62	25.21	**17.39**	**17.38**	**17.39**	21.93	**17.39**	17.58	25.15	27.98	47.30	40.46	**17.39**	36.98
2	37.87	17.61	**11.85**	**11.85**	**11.85**	13.68	**11.85**	**11.85**	16.20	30.30	37.44	17.77	**11.85**	38.10
3	35.55	23.45	**15.52**	**15.52**	**15.52**	20.00	15.73	15.73	23.02	33.12	34.99	21.96	**15.52**	38.36
4	41.14	14.28	9.09	**8.67**	**8.67**	10.67	**8.67**	9.09	12.61	28.92	38.67	12.46	**8.67**	34.50
5	50.40	28.39	**18.37**	**18.37**	**18.37**	24.29	18.58	18.98	27.15	27.71	49.54	36.89	**18.37**	35.54
6	43.92	23.27	**15.36**	**15.37**	**15.36**	19.47	**15.36**	15.55	22.64	32.21	43.92	21.72	**15.36**	35.79
7	31.36	12.49	**8.66**	**8.66**	**8.66**	10.85	**8.66**	**8.66**	12.27	31.31	30.75	16.98	**8.66**	38.55
8	39.13	31.23	**25.36**	**25.35**	**25.36**	27.87	**25.36**	**25.36**	30.93	38.19	38.52	25.65	**25.36**	47.44
9	44.36	26.93	**17.41**	**17.40**	**17.41**	21.55	**17.41**	19.06	26.26	31.97	42.02	28.37	**17.41**	36.88
10	56.57	19.65	**15.00**	**14.99**	**15.00**	17.80	**15.00**	15.43	18.51	24.25	55.65	39.61	**15.00**	35.17
Average	42.79	22.25	15.40	**15.36**	**15.36**	18.81	15.39	15.73	21.47	30.59	41.88	26.19	**15.36**	37.73
p-value	0.002	0.002	–	–	1.000	0.002	1.000	0.031	0.002	0.002	0.002	0.002	1.000	0.002
Confidence Interval	10.10 – 44.68	2.53 – 11.18	–	–	–	1.26 – 5.56	–	-0.04 – 0.70	-0.82 – 12.97	5.60 – 24.79	9.77 – 43.19	3.98 – 17.60	–	8.24 – 36.43

Table VII. Comparison of Average Rank Percentages of various metrics for single bug Flex, Grep, Gzip, Sed

Run #	O^d	Tarantula	Hyp_sa	Hyp_gp	O^p	Ochiai	Zoltar	Kul2	Kul1	Wong1	Wong2	Ample	GP13	Information Gain
1	17.41	11.59	**9.85**	9.97	21.07	15.10	12.78	14.26	15.37	21.15	17.52	17.58	21.07	41.95
2	42.14	24.13	20.54	**18.23**	24.21	24.88	24.38	24.97	25.36	24.92	30.34	39.63	24.21	41.24
3	54.84	7.39	**7.36**	7.40	12.99	25.47	25.47	25.47	25.47	31.53	20.58	45.28	31.11	40.40
4	38.23	13.66	11.22	**11.21**	19.62	20.21	19.56	19.56	22.05	36.04	21.58	39.40	28.02	40.24
5	44.05	17.16	16.64	16.64	21.95	16.56	16.58	**16.63**	16.73	19.37	43.39	33.75	21.95	43.76
6	39.77	1.68	**1.49**	**1.49**	14.36	24.41	24.41	24.41	24.41	44.78	4.22	36.95	38.45	32.57
7	36.97	26.11	20.54	20.33	25.69	**20.15**	22.58	19.90	20.54	29.28	31.64	20.76	25.90	45.94
8	46.65	15.82	**11.58**	**11.58**	17.06	24.56	24.23	24.31	24.95	34.97	27.40	37.59	30.36	34.13
9	22.73	**8.96**	9.86	9.22	34.49	16.01	16.42	16.20	16.35	38.38	15.98	14.95	34.49	36.27
10	62.63	25.33	17.71	**16.61**	23.98	37.05	31.26	35.05	38.42	36.71	33.83	48.93	37.25	36.61
Average	40.54	15.18	12.68	**12.27**	21.54	22.44	21.77	22.07	22.97	31.71	24.65	33.48	29.28	39.31
p-value	0.002	0.014	–	–	0.002	0.010	0.004	0.010	0.002	0.002	0.002	0.002	0.002	0.002
Confidence Interval	10.28 – 45.45	0.51 – 4.50	–	–	3.27 – 14.46	2.36 – 17.17	2.92 – 15.25	2.27 – 16.52	3.79 – 16.78	7.02 – 31.05	4.41 – 19.52	7.67 – 33.93	6.12 – 27.08	9.82 – 43.44

Table VIII. Comparison of Average Rank Percentages of various metrics for single bug Space

Run #	O^d	Tarantula	Hyp_sa	Hyp_gp	O^p	Ochiai	Zoltar	Kul2	Kul1	Wong1	Wong2	Ample	GP13	Information Gain
1	69.20	8.44	**1.11**	**1.11**	**1.11**	1.69	1.37	1.41	3.99	12.81	62.05	21.69	**1.11**	35.36
2	10.23	**2.60**	**2.60**	**2.60**	**2.60**	**2.60**	**2.60**	**2.60**	**2.60**	25.69	**2.60**	3.27	**2.60**	41.69
3	14.08	4.26	2.56	2.55	**2.54**	2.79	**2.54**	2.57	3.48	14.91	12.08	11.44	**2.54**	42.89
4	31.35	8.50	1.83	1.83	**1.37**	2.09	1.57	2.01	2.15	20.78	5.31	2.29	**1.37**	32.94
5	34.21	3.95	**1.38**	1.66	**1.38**	1.66	**1.38**	1.66	1.66	18.63	3.29	1.66	**1.38**	30.45
6	30.46	2.39	**0.81**	**0.81**	**0.81**	0.89	**0.81**	0.89	0.90	18.59	5.19	0.89	**0.81**	31.27
7	47.27	11.43	4.12	4.12	**3.49**	5.63	4.07	5.41	5.83	18.44	15.31	6.23	**3.49**	27.59
8	36.61	12.17	7.38	7.34	**7.32**	8.07	**7.32**	7.42	10.13	15.93	35.94	34.03	**7.32**	40.37
9	24.45	**1.56**	**1.56**	**1.56**	**1.56**	**1.56**	**1.56**	**1.56**	**1.56**	14.68	**1.56**	3.58	**1.56**	47.18
10	55.46	5.02	**1.17**	**1.17**	**1.17**	1.39	1.26	1.27	2.49	15.64	51.26	12.96	**1.17**	39.16
Average	35.33	6.03	2.45	2.47	**2.33**	2.84	2.45	2.68	3.48	17.61	19.46	9.80	**2.33**	36.89
p-value	0.001	0.004	–	–	0.063	0.004	0.813	0.004	0.004	0.001	0.004	0.001	0.063	0.001
Confidence Interval	13.41 – 52.35	1.15 – 6.01	–	–	-0.24 – 0.01	0.12 – 0.65	-0.03 – 0.02	0.07– 0.39	0.33 –1.73	6.18 –24.13	5.47 –28.54	3.00 –11.70	-0.24 –0.01	14.05– 54.83

DOI: 10.1002/spe

Table IX. Percentage of Successful Diagnosis for Single Bug Siemens Test Suite

Code Examined	1 %	2 %	5 %	10 %	20 %	50 %
Ample	7.68	**12.86**	34.82	40.00	56.96	77.68
DOP	1.25	3.93	7.68	11.43	26.07	65.89
GP13	**9.11**	19.46	**47.86**	**54.29**	**72.68**	93.21
Hyperbolic	**9.11**	19.46	**47.86**	**54.29**	**72.68**	93.21
Info_Gain	0.00	0.00	0.00	0.00	7.86	75.18
Kul1	6.43	10.36	33.75	41.43	55.71	85.36
Kul2	7.68	16.79	46.61	51.61	**72.68**	93.21
Ochiai	7.68	**12.86**	37.50	46.61	61.07	85.36
Optimal	**9.11**	19.46	**47.86**	**54.29**	**72.68**	93.21
Tarantula	6.43	10.36	33.75	40.00	54.46	84.11
Wong1	0.00	0.00	0.00	5.36	16.96	**98.57**
Wong2	5.18	5.18	8.93	11.43	27.32	68.57
Zoltar	**9.11**	19.46	46.61	**54.29**	**72.68**	93.21

Table X. Percentage of Successful Diagnosis for Single Bug Flex, Grep, Gzip, Sed

Code Examined	1 %	2 %	5 %	10 %	20 %	50 %
Ample	30.95	37.83	44.97	50.53	53.97	70.63
O^d	30.95	34.13	37.83	39.42	48.41	66.93
GP13	21.96	30.69	34.13	35.98	39.68	74.34
Hyperbolic	**47.62**	**56.35**	**61.90**	**68.78**	68.78	94.71
Info_Gain	0.00	0.00	0.00	1.85	5.56	69.05
Kul1	38.36	47.09	54.23	56.08	56.08	76.19
Kul2	40.21	48.94	54.23	56.08	56.08	78.04
Ochiai	40.21	47.09	54.23	56.08	56.08	78.04
O^p	33.07	41.80	45.24	47.09	50.79	87.30
Tarantula	43.92	50.79	58.20	66.93	**70.63**	**89.15**
Wong1	0.00	1.85	7.14	15.61	32.28	78.04
Wong2	45.77	48.94	57.94	59.79	65.08	79.89
Zoltar	38.36	47.09	54.23	56.08	57.94	77.78

Table XI. Percentage of Successful Diagnosis for Single Bug Space

Code Examined	1 %	2 %	5 %	10 %	20 %	50 %
Ample	49.46	49.46	75.00	80.00	90.00	95.00
DOP	10.00	10.00	20.00	25.00	34.11	69.46
GP13	**54.46**	**74.46**	**90.00**	**95.00**	**100.00**	**100.00**
Hyperbolic	**54.46**	**74.46**	**90.00**	90.00	**100.00**	**100.00**
Info_Gain	0.00	0.00	0.00	0.00	0.00	**100.00**
Kul1	49.46	64.46	85.00	90.00	95.00	**100.00**
Kul2	49.46	**74.46**	**90.00**	90.00	**100.00**	**100.00**
Ochiai	49.46	69.46	**90.00**	90.00	95.00	**100.00**
Optimal	**54.46**	**74.46**	**90.00**	**95.00**	**100.00**	**100.00**
Tarantula	29.11	54.46	64.46	80.00	90.00	**100.00**
Wong1	0.00	0.00	0.00	20.00	60.89	**100.00**
Wong2	34.29	44.29	59.82	59.82	70.00	85.00
Zoltar	49.46	**74.46**	**90.00**	90.00	**100.00**	**100.00**

DOI: 10.1002/spe

Table XII. Ten Fold New –Comparison of Average Rank Percentages of various metrics for multi bug STS

Run #	O^d	Tarantula	Hyp_sa	Hyp_gp	O^p	Ochiai	Zoltar	Kul2	Kul1	Wong1	Wong2	Ample	GP13	Information Gain
1	42.88	24.87	23.40	**23.08**	24.10	24.04	27.94	23.51	25.07	31.21	37.99	30.05	25.90	35.66
2	39.33	21.01	**19.46**	19.47	25.06	22.42	25.83	21.98	22.77	35.11	34.23	29.17	28.75	34.87
3	34.76	18.42	**12.94**	12.97	22.11	20.16	23.18	18.90	21.38	34.61	28.06	27.39	25.81	35.40
4	32.99	19.92	**14.84**	**14.84**	23.87	16.04	22.45	18.02	19.06	34.39	30.90	25.08	23.87	38.46
5	37.37	19.72	20.47	**20.04**	25.90	23.97	30.24	28.33	24.23	39.21	28.35	28.25	31.69	37.81
6	31.52	23.33	**21.01**	21.09	25.63	21.57	21.82	20.65	21.85	34.62	30.72	22.81	25.63	39.26
7	35.94	25.25	23.83	23.73	32.63	**22.89**	30.10	23.24	23.79	36.27	34.79	33.02	32.63	40.07
8	35.07	22.12	19.04	**19.00**	25.15	18.86	26.39	23.36	21.51	35.44	32.38	22.35	26.49	35.49
9	32.44	20.14	**14.73**	14.76	26.00	15.44	23.03	15.21	18.52	34.61	30.80	21.11	26.47	37.54
10	37.00	21.73	17.44	**17.42**	26.29	19.69	28.20	19.85	21.27	34.55	34.14	27.31	28.04	39.03
Average	35.93	21.65	18.72	**18.64**	25.67	20.51	25.92	21.30	21.95	35.00	32.24	26.66	27.53	37.36
p-value	0.002	0.004	–	–	0.002	0.027	0.002	0.027	0.004	0.002	0.002	0.002	0.002	0.002
Confidence Interval	6.35 – 28.08	0.94 – 4.93	–	–	2.57 – 11.35	0.20 – 3.39	2.66 – 11.75	-1.64 – 6.82	-1.62 – 8.09	6.01 – 26.56	4.99 – 22.06	2.93 – 12.95	3.25 – 14.37	6.88 – 30.41

Table XIII. Ten Fold New –Comparison of Average Rank Percentages of various metrics for multi bug Space

Run #	O^d	Tarantula	Hyp_sa	Hyp_gp	O^p	Ochiai	Zoltar	Kul2	Kul1	Wong1	Wong2	Ample	GP13	Information Gain
1	34.65	15.21	9.87	9.62	18.45	12.86	20.09	13.23	13.00	29.94	14.94	**5.26**	21.89	40.93
2	25.32	12.09	10.89	10.75	21.94	11.29	20.21	11.35	11.58	27.79	17.50	**5.92**	21.19	42.76
3	18.73	8.81	6.83	**6.77**	20.36	10.29	17.45	9.91	10.75	30.60	9.99	8.17	22.77	39.84
4	18.29	5.51	**1.90**	1.95	9.01	3.41	8.10	3.39	3.43	23.25	1.45	2.42	11.01	36.71
5	21.51	7.38	3.66	3.82	11.31	5.07	11.01	5.03	5.10	24.06	**3.03**	3.95	13.31	40.45
6	29.05	10.35	3.73	**3.58**	11.69	7.65	13.92	7.72	7.89	26.69	8.60	4.50	16.34	38.49
7	26.60	8.19	5.38	5.66	14.42	5.06	11.49	**5.05**	5.31	24.86	7.90	6.99	15.07	36.18
8	29.05	11.74	8.90	8.79	20.30	10.96	19.34	10.98	11.19	29.09	14.17	**5.35**	21.47	39.17
9	23.48	7.85	5.63	**5.47**	10.36	6.36	11.21	6.29	6.40	23.42	8.65	7.86	11.31	38.68
10	32.31	12.02	**3.07**	3.59	8.84	9.02	13.02	8.87	9.56	25.12	4.65	6.53	15.09	38.27
Average	25.90	9.92	5.99	6.00	14.67	8.20	14.59	8.18	8.42	26.48	9.09	**5.69**	16.94	39.15
p-value	0.002	0.002	–	–	0.002	0.004	0.002	0.002	0.002	0.002	0.010	1.000	0.002	0.002
Confidence Interval	7.34 – 32.48	1.45 – 6.41	–	–	3.20 – 14.16	0.71 – 3.71	3.17 – 14.03	0.81 – 3.58	0.90 – 3.97	7.56 – 33.43	0.75 – 5.46	–	4.04 – 17.87	12.23 – 54.09

Table XIV. Ten Fold New –Comparison of Average Rank Percentages of various metrics for multi bug Flex, Grep, Sed, Gzip

	O^d	Tarantula	Hyp_sa	Hyp_gp	O^p	Ochiai	Zoltar	Kul2	Kul1	Wong1	Wong2	Ample	GP13	Information Gain
1	20.30	15.52	8.46	8.55	17.66	**7.04**	9.22	**7.04**	6.96	19.98	7.14	12.26	17.66	28.38
2	20.57	14.73	5.04	4.91	17.95	4.91	5.45	4.91	4.91	25.63	4.93	**3.88**	17.95	30.66
3	27.61	16.23	**8.43**	8.59	24.48	11.92	13.05	11.30	12.30	25.58	12.15	12.08	24.48	32.84
4	24.46	12.17	**11.41**	11.48	29.84	16.05	12.96	16.08	16.14	34.80	19.54	15.39	29.84	34.02
5	8.88	8.93	5.91	5.99	26.87	12.82	10.30	12.20	13.36	27.00	**5.82**	11.48	26.87	27.69
6	25.11	18.82	10.18	**10.11**	24.29	13.60	10.62	12.78	15.24	25.32	15.86	16.55	24.29	26.53
7	16.56	16.61	18.31	18.32	29.93	22.82	23.99	22.24	23.24	30.22	18.48	**12.67**	29.93	33.06
8	33.07	22.00	9.20	**9.01**	25.29	14.77	15.53	13.51	16.35	27.56	22.63	18.29	25.29	28.38
9	11.22	9.32	3.91	4.24	12.23	2.75	**2.72**	2.75	2.75	18.27	2.77	5.07	12.23	26.82
10	26.45	15.40	6.40	6.30	17.71	5.94	6.33	**5.53**	7.19	22.76	17.55	13.89	17.71	31.69
Average	21.42	14.97	**8.72**	8.75	22.62	11.26	11.02	10.83	11.84	25.71	12.69	12.16	22.62	30.01
p-value	0.004	0.006	–	–	0.002	0.084	0.027	0.083	0.048	0.002	0.131	0.064	0.002	0.002
Confidence Interval	4.08 – 21.31	1.81 – 10.69	–	–	5.13 – 22.68	-0.34 – 5.42	0.25 – 4.33	0.78 – 3.44	1.15 – 5.09	6.27 – 27.71	-1.18 – 9.10	-0.21 – 7.08	5.13 – 22.68	7.85 – 34.71

DOI: 10.1002/spe

Table XV. Percentage of Successful Diagnosis for multi Bug Siemens Test Suite

Code Examined	1 %	2 %	5 %	10 %	20 %	50 %
Ample	3.96	8.68	33.40	39.81	57.17	78.30
O^d	1.13	6.04	13.02	20.00	37.74	72.26
GP13	8.49	11.70	23.21	34.15	53.21	79.06
Hyperbolic	9.81	**13.96**	**39.06**	**46.98**	**68.49**	**89.06**
Info_Gain	0.00	0.00	2.08	5.47	21.89	83.40
Kul1	6.42	9.81	36.79	43.58	57.17	84.53
Kul2	7.36	10.38	34.34	41.51	65.09	87.55
Ochiai	5.85	10.57	37.92	46.23	63.77	84.53
O^p	**10.38**	13.58	25.09	36.04	55.09	81.32
Tarantula	9.81	13.02	38.49	45.47	55.66	83.02
Wong1	0.75	0.75	3.58	5.85	24.15	81.13
Wong2	9.06	9.43	20.19	22.83	43.40	78.30
Zoltar	9.25	12.64	24.72	36.60	56.23	82.45

Table XVI. Percentage of Successful Diagnosis for multi bug Space

Code Examined	1 %	2 %	5 %	10 %	20 %	50 %
Ample	**51.98**	55.22	75.43	83.05	**93.53**	**99.17**
O^d	16.13	18.57	34.68	41.12	51.55	81.45
GP13	36.67	39.53	49.25	55.73	65.40	88.32
Hyperbolic	48.37	59.27	**81.87**	**85.88**	91.57	96.00
Info_Gain	0.00	0.00	0.00	0.00	2.82	77.47
Kul1	51.15	62.45	76.60	80.23	87.12	88.32
Kul2	49.55	**63.28**	79.03	79.43	87.52	88.32
Ochiai	50.75	62.45	79.43	80.23	87.12	88.32
O^p	41.52	44.38	54.10	60.58	70.25	95.60
Tarantula	29.02	44.75	66.93	70.97	83.05	89.12
Wong1	0.00	0.00	0.00	12.52	39.97	88.32
Wong2	47.13	59.63	74.60	75.40	84.68	93.20
Zoltar	40.30	45.17	56.08	56.08	69.82	88.32

Table XVII. Percentage of Successful Diagnosis for multi bug Flex, Grep, Gzip, Sed

Code Examined	1 %	2 %	5 %	10 %	20 %	50 %
Ample	37.00	39.00	50.33	70.67	72.67	93.78
O^d	35.89	36.89	47.00	3.11	62.44	83.67
GP13	11.44	13.44	18.56	69.67	46.22	92.00
Hyperbolic	41.22	41.22	54.33	71.67	**80.78**	**100.00**
Info_Gain	0.00	0.00	0.00	3.00	16.22	93.89
Kul1	**45.33**	48.33	**57.44**	62.44	70.67	93.00
Kul2	**45.33**	48.33	56.44	40.00	73.78	94.00
Ochiai	**45.33**	48.33	**57.44**	49.22	73.78	94.00
O^p	11.44	13.44	18.56	70.67	46.22	92.00
Tarantula	35.89	36.89	51.00	71.67	68.44	89.89
Wong1	2.11	2.11	2.11	40.00	58.33	92.00
Wong2	**45.33**	**49.33**	**57.44**	**72.67**	73.67	92.78
Zoltar	39.22	41.22	52.33	68.67	70.78	97.00

7. STATISTICAL SIGNIFICANCE

In order to find that the difference in the average rank percentage computed by hyperbolic metrics are significant as compared to other metrics, we use Wilcoxon signed rank test at 5% significance

Prepared using speauth.cls
DOI: 10.1002/spe

level [24]. We establish an alternative hypothesis; the bug localization performance using hyperbolic metrics is better than other metrics. The null hypothesis is that there is no difference between the rank percentages computed by the hyperbolic and other metrics.

The "p-value" column in tables VI, VII, VIII , XII, XIII and XIV show the p-value computed by Wilcoxon signed rank test on single and multi-bug data sets of the STS, Space, Flex, Grep, Gzip and Sed. p-value of 0.05 is a normal standard for statistical significance. It can be seen from above mentioned tables that p-value of the most of the metrics is lower than 0.05 showing that the decrease in average rank percentage by hyperbolic metrics is statistically significant. Average Rank Percentage of hyperbolic metrics (column "Average") is lower than all other metrics except for single bug space and multi bug space. However, even for these two datasets, p-value of best performing metrics is greater than 0.05 showing that performance of hyperbolic metrics is neither inferior nor superior to these metrics. In other words, hyperbolic is as good in performance as these metrics.

There are some situations where average rank percentage of hyperbolic metrics is lower than other metrics but the difference is not statistically significant. For example, p-value of Kulczynski2 and Ochiai in Table XIV is 0.083 and 0.084 respectively showing that the difference in performance of these metrics is not significant. However, Hyperbolic metrics still have an edge over other metrics due their adapting nature i.e. they adjust themselves for single bug, multiple bug and deterministic bug datasets and performs as good as or better than the best performing metrics for these datasets.

In order to find the range of improvement obtained by hyperbolic metrics, we have shown 95% confidence intervals for all the instances where p-value is less than 1. Both for single and multi bug datasets, highest improvement range is achieved over information gain for Space.

To conclude, Hyperbolic metrics perform better or atleast as good as the known best performing metrics for single bug, multiple bug and deterministic bug datasets. p-values show that improvement is statistically significant while confidence intervals show that range of improvement is quite high as compared to the most of the metrics.

8. THREATS TO VALIDITY

The accuracy of a fault localization metric to localize fault is influenced by the granularity level under consideration (e.g. statement, basic block, predicate or method). We have used statement as basic block in this study, however results may vary using method, predicate or other levels of granularity.

We have chosen a wide spectrum of metrics to compare with our proposed class of metrics; best performing metric on single bug benchmarks (O^p), deterministic bug benchmarks (O^d), multiple bug benchmarks (Ochiai and Kulczynski2) and many others. Although we have not reported all SBFL functions (over 150 of them) we believe our results will hold good for any SBFL function as long as the statistical properties of training data and test data are very similar, as performance of hyperbolic metrics depends on effective learning. Furthermore, in order to compare hyperbolic metrics with other metrics, we have coded all these metrics. Although another check is performed to make sure that there is no bug in the code, human error is still possible.

We have used a combination of small and large sized datasets for evaluation of our proposed metrics. The Siemens Test Suite contains small programs having manually induced bugs, Flex, Grep and Gzip are larger programs having manually induced bugs, Sed is a large program having a combination of real and synthetic bugs while Space is a large program having real bugs. Experimental results on all of these programs show that hyperbolic metrics performs better or atleast equal to best performing metrics for different types of bugs. However, to further consolidate the claims, experiments on other datasets would be beneficial and we are endeavouring to collect such datasets.

Prepared using speauth.cls
DOI: 10.1002/spe

9. OTHER RELATED WORK

Landsberg et al. [23] evaluate 157 different similarity metrics from the literature for SBFL and show how many are equivalent for ranking purposes. The Pattern-Similarity metric, and a simpler equivalent metric, PattSim2 = $-nf \times ep$, have hyperbolic contours identical to those in Figure 1(c). Furthermore, in all these metrics small "prior constants" are added to the spectral values to avoid division by zero etc. These are not needed for PattSim2 since there is no division, but the resulting metric is of the form $-(nf + K_1)(ep + K_2)$. These two constants effectively translate the contours vertically and horizontally in the same way as K_1 and K_2 in our hyperbolic metrics, and make the metric strictly rational. By manually selecting the two constant values, this metric was found to perform better than all others from the literature for a benchmark set which included the Siemens programs plus several significantly larger programs with multiple bugs. Our work differs in that we have three parameters rather than two, we scale the spectral values to the range 0–1 and we find the constants automatically, using machine learning techniques. Landsberg et al. also experimented with adapting metrics so they are single bug optimal, in the same way as proposed in [15]. The single bug optimal version of Ochiai performed the best of all metrics considered. Lucia et al. [34]use data fusion method to fuse various fault localization metrics which are more effective in wide spectrum of data. Unlike the work presented in this paper, their technique is bug specific in which the set of techniques to be fused are adaptively selected for each buggy program based on its spectra. Furthermore, it doesn't require any training data.

Most of the formulas or metrics used in Spectral based fault localization are not designed specifically for debugging. Jaccard for example used for biological classification for the first time and Ochiai used in marine zoology [35]. Tarantula was the first metric designed for spectral debugging [20]. Few other widely used metrics are Ample [25], Zoltar [24], Wong [21] etc.

Naish et al. propose optimal metrics for single bug [6] and deterministic bug programs [17] which are empirically proved to be the best metrics in these areas. They, however, do not perform equally well on multiple bug data.

There are few techniques available in literature for multiple bug problem which are combination of spectral based with machine learning or model based approaches. Some of these are given below.

James et al. present a clustering approach for debugging in parallel in presence of multiple bugs. Using fault localization information from program execution and behaviour models, they develop a technique that automatically partitions the failing test cases into clusters that target different faults. These clusters are called fault focusing clusters. Each fault focusing cluster is then combined with all the passing test cases to get a specialized test suite that targets a single fault. These specialized test suits can then be assigned to different developers who can work in parallel for debugging and localizing bugs [29].

Abreu el al. present a multi fault localization technique called BARINEL by combining spectral based fault localization and model based reasoning. Model based approaches are more accurate as compared to spectral fault localization but due to their computational complexity they are very expensive for large applications. BARINEL, however, uses effective candidate selection process that reduces it's complexity and make it better candidate for large programs as well [16].

Wong et al. propose a crosstab-based statistical fault localization technique(CBT). The technique uses statement based coverage information. A comparison has been made between CBT and Tarantula and results prove CBT better then tarantula. The technique is claimed to be effectively applicable for multiple bug programs [36].

Slicing and Dicing are considered as one of the oldest techniques in debugging and fault localization. Slicing refers to the piece of program code that affects the value of any variable while dicing is the part of program, which appears in one slice but not in another. These approaches narrow down the program part, which is more likely to be buggy so that developer can concentrate on a small part of the code for fault localization [22][37].

Mutant based fault localization (MUSE) identifies the fault or bug by utilizing the information obtained by mutating the faulty and correct statement [38]. The technique uses the intuition that if a faulty statement is mutated, it will cause more tests to pass than average and if a correct statement

DOI: 10.1002/spe

is mutated, it will cause more tests to fail then average. These intuitions are the basis of MUSE [39][40][41].

State based approaches aim to localize the bugs by observing the changing state of the program and identifying the failure inducing circumstances [42][43][44]. Failure Inducing Circumstances refer to the input to the test cases which causes it to fail e.g. program input (particular URL input fails the web browser), User Interaction (keystroke of the user causing the program to fail) and changes to the program code.

There are many studies found that test reduction and test selection could help in improving the performance of fault localization technique [45][46]. Recently, many machine learning, data mining, static analysis and Information Retrieval Based approaches are proposed in this area which can be used together with spectra-based approaches[29][47][31] [48][49][50] [51][52][53].

10. FUTURE WORK

We plan to analyse different scaling forms to better understand why some forms of scaling perform better than others. Furthermore, we are working to device other set of classes having similar properties and which can perform even better with simple learning. Using other datasets like Defects4J having real bugs would also be helpful in further validating our results. We also plan to develop a practical fault localization tool using the technique we discuss in this study.

11. CONCLUSION

Performance of Spectral Based Fault Localization is strongly influenced by the choice of metric used to estimate the likelihood that a given program component is buggy. We have proposed the class of "hyperbolic" metrics, to efficiently locate single, deterministic and multiple bugs. The class has a small number of numeric parameters; the parameter values can be determined by using training data. We have shown that learned hyperbolic metrics can perform as well or better than previously discovered metrics in a wide range of situations. Using both synthetic and real programs we have demonstrated that these metrics can achieve best performance for programs with a single bug and with deterministic bugs (where bug execution always leads to test case failure) — the two extreme cases we have theoretical results for. In a range of other model programs, with two bugs which cause failure with varying consistency, hyperbolic metrics performed best on average. Using data from small real programs seeded with two bugs and large real programs with real software bugs the learned hyperbolic metrics out-performed the best previously know metrics on average.

REFERENCES

1. Collofello JS, Woodfield SN. Evaluating the effectiveness of reliability-assurance techniques. *Journal of systems and software* 1989; **9**(3):191–195.
2. Wong WE, Gao R, Li Y, Abreu R, Wotawa F. A survey on software fault localization. *IEEE Transactions on Software Engineering* 2016; **42**(8):707–740.
3. Jones JA, Harrold MJ, Stasko J. Visualization of test information to assist fault localization. *Proceedings of the 24th international conference on Software engineering*, ACM, 2002; 467–477.
4. Liblit B. Cooperative Bug Isolation. PhD Thesis, University of California 2004.
5. Abreu R, Zoeteweij P, van Gemund A. An evaluation of similarity coefficients for software fault localization. *PRDC'06* 2006; :39–46.
6. Naish L, Lee HJ, Kotagiri R. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)* August 2011; **20**(3).
7. Kochhar PS, Xia X, Lo D, Li S. Practitioners' expectations on automated fault localization. *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ACM, 2016; 165–176.
8. Xia X, Bao L, Lo D, Li S. automated debugging considered harmful considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, IEEE, 2016; 267–278.
9. Campos J, Riboira A, Perez A, Abreu R. Gzoltar: an eclipse plug-in for testing and debugging. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ACM, 2012; 378–381.

Prepared using speauth.cls
DOI: 10.1002/spe

10. Liblit B. Cooperative bug isolation: Winning thesis of the 2005 acm doctoral dissertation competition, volume 4440 of lecture notes in computer science 2007.
11. Abreu R, Zoeteweij P, van Gemund A. On the Accuracy of Spectrum-based Fault Localization. *TAICPART-Mutation 2007*, IEEE Computer Society: Windsor,UK, 2007; 89–98.
12. Naish L. Similarity to a single set january 2016, doi:10.7287/peerj.preprints.1713v1.
13. Wang S, Lo D, Jiang L, Lau HC, *et al.*. Search-based fault localization. *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, IEEE Computer Society, 2011; 556–559.
14. Yoo S. Evolving human competitive spectra-based fault localisation techniques. *Search Based Software Engineering*. Springer, 2012; 244–258.
15. Naish L, Lee HJ, Kotagiri R. Spectral debugging: How much better can we do? *35th Australasian Computer Science Conference (ACSC 2012), CRPIT Vol. 122*, CRPIT, 2012.
16. Abreu R, Zoeteweij P, Van Gemund AJ. Spectrum-based multiple fault localization. *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, IEEE, 2009; 88–99.
17. Naish L, Lee HJ. Duals in spectral fault localization. *Proceedings of ASWEC 2013*, IEEE Press, 2013.
18. Naish L, Ramamohanarao K, *et al.*. Multiple bug spectral fault localization using genetic programming. *Software Engineering Conference (ASWEC), 2015 24th Australasian*, IEEE, 2015; 11–17.
19. Do H, Elbaum SG, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal* 2005; **10**(4):405–435.
20. Jones J, Harrold M. Empirical evaluation of the Tarantula automatic fault-localization technique. *Proceedings of the 20th ASE* 2005; :273–282.
21. Wong WE, Qi Y, Zhao L, Cai K. Effective Fault Localization using Code Coverage. *Proceedings of the 31st Annual IEEE Computer Software and Applications Conference* 2007; :449–456.
22. Lee HJ. Software Debugging Using Program Spectra. PhD Thesis, University of Melbourne 2011.
23. Landsberg D, Chockler H, Kroening D, Lewis M. Evaluation of measures for statistical fault localisation and an optimising scheme. *Fundamental Approaches to Software Engineering*. Springer, 2015; 115–129.
24. Gonzalez A. Automatic Error Detection Techniques based on Dynamic Invariants. Master's Thesis, Delft University of Technology, The Netherlands 2007.
25. Dallmeier V, Lindig C, Zeller A. Lightweight bug localization with ample. *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, ACM, 2005; 99–104.
26. Lo D, Jiang L, Budi A, *et al.*. Comprehensive evaluation of association measures for fault localization. *Software Maintenance (ICSM), 2010 IEEE International Conference on*, IEEE, 2010; 1–10.
27. Parnin C, Orso A. Are automated debugging techniques actually helping programmers? *Proceedings of the 2011 international symposium on software testing and analysis*, ACM, 2011; 199–209.
28. Pearson S, Campos J, Just R, Fraser G, Abreu R, Ernst MD, Pang D, Keller B. Evaluating and improving fault localization. *Proceedings of the 39th International Conference on Software Engineering*, IEEE Press, 2017; 609–620.
29. Jones J, Bowring J, Harrold M. Debugging in parallel. *Proceedings of the ISSTA* 2007; :16–26.
30. Dickinson W, Leon D, Podgurski A. Finding failures by cluster analysis of execution profiles. *Proceedings of the 23rd international conference on Software engineering*, IEEE Computer Society, 2001; 339–348.
31. Briand LC, Labiche Y, Liu X. Using machine learning to support debugging with tarantula. *Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on*, IEEE, 2007; 137–146.
32. Di Fatta G, Leue S, Stegantova E. Discriminative pattern mining in software fault detection. *Proceedings of the 3rd international workshop on Software quality assurance*, ACM, 2006; 62–69.
33. Jiang L, Su Z. Automatic isolation of cause-effect chains with machine learning. *Technical Report*, Tech. rep., Technical Report CSE-2005-32, University of California, Davis 2005.
34. Lo D, Xia X, *et al.*. Fusion fault localizers. *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, ACM, 2014; 127–138.
35. Ochiai A. Zoogeographic studies on the soleoid fishes found in japan and its neighbouring regions. *Bull. Jpn. Soc. Sci. Fish* 1957; **22**(9):526–530.
36. Wong WE, Debroy V, Xu D. Towards better fault localization: A crosstab-based statistical approach. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* 2012; **42**(3):378–396.
37. Agrawal H, Horgan J, London S, Wong W. Fault localization using execution slices and dataflow tests. *Software Reliability Engineering* 1995; :143–151.
38. Moon S, Kim Y, Kim M, Yoo S. Ask the mutants: Mutating faulty programs for fault localization. *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, IEEE, 2014; 153–162.
39. Papadakis M, Le Traon Y. Using mutants to locate" unknown" faults. *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, IEEE, 2012; 691–700.
40. Debroy V, Wong WE. Combining mutation and fault localization for automated program debugging. *Journal of Systems and Software* 2014; **90**:45–60.
41. Papadakis M, Le Traon Y. Effective fault localization via mutation analysis: a selective mutation approach. *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ACM, 2014; 1293–1300.
42. Zeller A. Isolating cause-effect chains from computer programs. *ACM SIGSOFT Software Engineering Notes* 2002; **27**(6):10.
43. Zeller A, Hildebrandt R. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on* 2002; **28**(2):183–200.
44. Zhang X, Gupta N, Gupta R. Locating faults through automated predicate switching. *Proceedings of the 28th international conference on Software engineering*, ACM, 2006; 272–281.
45. Xia X, Gong L, Le TDB, Lo D, Jiang L, Zhang H. Diversity maximization speedup for localizing faults in single-fault and multi-fault programs. *Automated Software Engineering* 2016; **23**(1):43–75.
46. Xuan J, Monperrus M. Test case purification for improving fault localization. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 2014; 52–63.

Prepared using speauth.cls
DOI: 10.1002/spe

47. Hsu H, Jones J, Orso A. RAPID: Identifying bug signatures to support debugging activities. *23rd IEEE/ACM International Conference on Automated Software Enginering, 2008. ASE 2008*, 2008; 439–442.
48. B Le TD, Lo D, Le Goues C, Grunske L. A learning-to-rank based fault localization approach using likely invariants. *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ACM, 2016; 177–188.
49. Xuan J, Monperrus M. Learning to combine multiple ranking metrics for fault localization. *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, IEEE, 2014; 191–200.
50. Le TDB, Lo D, Thung F. Should i follow this fault localization tools output? *Empirical Software Engineering* 2015; **20**(5):1237–1274.
51. Le TDB, Lo D, Li M. Constrained feature selection for localizing faults. *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, IEEE, 2015; 501–505.
52. Le TDB, Oentaryo RJ, Lo D. Information retrieval and spectrum based bug localization: better together. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ACM, 2015; 579–590.
53. Neelofar N, Naish L, Lee J, Ramamohanarao K. Improving spectral-based fault localization using static analysis. *Software: Practice and Experience* 2017; .

Prepared using speauth.cls
DOI: 10.1002/spe