Software Fault Localisation via Probabilistic Modelling

Colin G. Johnson[0000−0002−9236−6581]

School of Computer Science, University of Nottingham,
Jubilee Campus, Nottingham, UK
Colin.Johnson@nottingham.ac.uk

Abstract. Software development is a complex activity requiring intelligent action. This paper explores the use of an AI technique for one step in software development, *viz.* detecting the location of a fault in a program. A measure of program progress is proposed, which uses a Naïve Bayes model to measure how useful the information that has been produced by the program to the task that the program is tackling. Then, deviations in that measure are used to find the location of faults in the code. Experiments are carried out to test the effectiveness of this measure.

Keywords: Software development, bug finding, Naïve Bayes

1 Introduction

Software development is a task requiring substantial intelligence. In recent years, a number of AI based approaches have been taken to software development tasks [8], both in terms of actually writing code, and in the wide variety of tasks that surround this, such as decomposition of tasks [17], fault finding and fixing [15,16], building and improving test suites [3], etc.

This paper is concerned with applying an AI technique to fault localization—that is, is the problem of identifying which component of a faulty system is causing the fault [23]. For a software system, fault localization can be used at a number of scales, from identifying which component of a large multi-component system is causing the fault, through to identifying which line of code is the cause of a fault in a single function.

In this paper, we are concerned with the smallest scale of fault localization, finding the location of a fault in a single unit of faulty code: a function/method consisting of a number lines of code. The problem is as follows. Take the source code for a function f, and a test set consisting of sample inputs f and the expected output for each sample input. A fault in f means that the input-output behaviour of the function on the inputs in the test set do not match with the input-output pairs in the test set; for some inputs, the output from f is different to the output in the test set. The localization problem is to identify the line(s) of code that give rise to this fault.

This problem is commonly tackled using a *spectrum*-based approach. This is where a spectrum matrix, indexed by test set inputs $\times$ lines of code in f is

created, with the entry being *True* if that line of code was executed during that test case, and *False* otherwise. Fault localization is then carried out using one of many *suspiciousness metrics* [21], which assign a numerical value to each line of code, with the idea that the lines most likely to get a high suspiciousness value are those that are the causes of the fault. Typically, suspiciousness metrics are based on the difference between program paths taken when the correct output is obtained and when a wrong output is obtained; lines which occur most frequently in traces that give incorrect outputs are treated as most suspicious.

However, sometimes the cause of the fault is to do with the *values taken* during that execution, not the different paths taken. Indeed, in some applications—signal processing, image and audio transformation, numerical computation, etc.—there many be few conditional statements to facilitate the creation of usefully different traces to calculate suspiciousness metrics from. The focus of this paper is on testing such programs, consisting of a succession of calculations.

To tackle these problems, we introduce two concepts. The first is that of a *rich spectrum*. A conventional program spectrum consists of a 2-tensor of size $(\mathit{numberOfTestCases}, \mathit{numberOfLines})$ with entries in the set $\mathit{True}, \mathit{False})$. The entries indicate, for each test case, whether the program executed that particular line whilst running that test case. A rich spectrum is a 3-tensor, of size $(\mathit{numberOfTestCases}, \mathit{numberOfLines}, \mathit{numberOfVariables})$, with values in a set appropriate to the problem at hand (in the examples in the paper, these will be integer or floating point numbers). The entries in the rich spectrum represent the values computed by the various lines of the code. The aim of this is to capture a richer set of data about the program's execution, to which AI and data mining algorithms can be applied.

The second concept is the idea of probabilistic accumulation of evidence as a program executes. The key concept here is that we can quantify an approximation to the progress of a program towards solving its task. In this paper the model used is a Naïve Bayes model, which is recalculated as each line is executed. A record is kept of the accuracy of each of these models. The key idea is that, if a program is successful at its task, the accuracy of this model will increase line-by-line, because the data encoded in the variables at the end of each line is more informative, therefore the probability of being able to predict the output based on that information will increase. Conversely, if there is a fault, then a piece of irrelevant, distracting information will be created, which will mean that there will be a dip in that probability.

The paper is structured as follows. Section 2 explains previous approaches to the problem, Section 3 explains the new model of measuring program execution, and then Section 4 explains how that model is used to detect faults. Two experiments are described in Section 5, then there is a discussion in Section 7 of the limitations and threats to validity of the model. Finally, Section 8 summarises the ideas in the paper and presents ideas for further developments.

2 Background

A number of approaches to software fault localisation have been investigated in the literature (see e.g. [22] for a survey). Some of these are driven by human programmer's understanding of the program—e.g. printing out variable values and looking for patterns in light of the fault, comparing program activities with asserted properties, inserting breakpoints to review machine state at certain points, or examining the run time or execution frequencies of parts of the program and comparing it to an expected profile.

There are a number of approaches to automating or semi-automating the fault location detection process. One of these is based around *slicing* the program, that is, finding which subset of the code precede the point where the fault has been detected. A related idea is that of *spectral* testing [11], where an array is constructed to map out which statements are executed when there is a fault, and when there isn't. Then, a *suspiciousness statistic* is calculated, which predicts which line is most likely to be the cause of the fault—the basic idea being that lines that are often executed when a fault occurs, but not when it doesn't, are more likely to be the cause of the fault. We will take this idea further in this paper, both extending the idea of the spectrum, and using a different kind of statistical model to detect faults.

Another approach concentrates on detecting or constructing test examples that cause faults, so that the programmer can look for patterns in these test examples. For example, delta debugging [6] focuses on trimming down failure-causing inputs until some minimal failure-causing cases are found.

A number of approaches to the localisation problem have been grounded in AI and machine learning methods. For example [24] have used neural networks to discover the association between test case failures and code coverage, whilst another paper [4] has used decision trees to group together test cases that cause similar kinds of faults. Machine learning has also been used to improve existing techniques—for example, the spectral based approach to fault localisation has been improved by using genetic programming to discover suspiciousness formulae that are more effective than the traditional formulae [10] written by people [25]

There are a number of problems that contemporary fault localization methods struggle to address. Some of these are concerned with multiple faults in the same program—a fault can be obstructed by a later fault, and sometimes a later fault can mask or even undo earlier one. In this paper we address this problem by constructing a set of models that take a certain number of lines of the code and build a probabilistic model for the remainder of the computation, thus meaning that the rest of the code after the initial fault is not executed. Secondly, most approaches rely on differences between different execution paths in code. This means that localisation cannot be done in blocks of code without any conditional branching. By contrast, in this paper, we focus on the information calculated by each step in the code, meaning that faults can be localised within a non-branching piece of code.

3 Modelling Program Execution with Naïve Bayes Models

Programs are written to carry out a task. This task can be described in a number of ways: by a formal specification, by a set of input-output examples, and/or by a natural language description. As a program executes, it makes progress on the task. A key idea of this approach is that we can quantify this progress. For a correct program without any computations that are irrelevant to the task, we expect each line to calculate some information that is relevant to the task. After each line of the program has been executed, the computer's memory contains more information relevant to the task then it did before.

For the purposes of this paper we will focus on imperative programs that have no loops or conditionals (apart from single-line conditions and folds). There is no fundamental reason why these ideas cannot be developed for such programs, or for other programming paradigms, but for a first attempt at these approaches, this provides a starting point. Furthermore, it emphasises an aspect of code that has been neglected by previous approaches to fault localisation. In many previous approaches, such as the spectral approaches discussed above, the branching execution of the program is key to discovering the fault, and the location of faults in parts of code without branches cannot be found.

3.1 Progress Modelling

This section explains how we quantify the progress of a program towards its task. Before the program is executed, no memory is allocated to the program. Therefore, there is no information to tell us whether the program has made any progress towards the solution. As each statement is executed, the state of the computer's memory contains more information that is relevant to the task. To measure this progress, we execute the program on a large number of test cases, accumulating the variable states after each statement has been executed.

There is then a gap between these variable values for the partially-executed program and the test case outputs—what Androutsopoulos et al. [1] call a "ghost" program. We then build a probabilistic model of the relationship between these variable values and the set of outputs in the test cases. The measure of progress is the accuracy of that model on a reserved test set of examples of variable values and test case outputs. Repeating this process for each line of the program in turn gives a vector of numbers, each corresponding to a line in the code. This process is illustrated in Figure 1.

This program progress measure should increase monotonically where a program is correct (and doesn't contain any irrelevant or redundant lines), because each execution step will provide some useful information towards solving the test cases, thus reducing the complexity of model needed to bridge the gap to the target output. Contrastingly, many kinds of error will break this monotonicity: by executing an error-causing step, either an irrelevant or distracting piece of information will be created, or a piece of information needed to solve the test

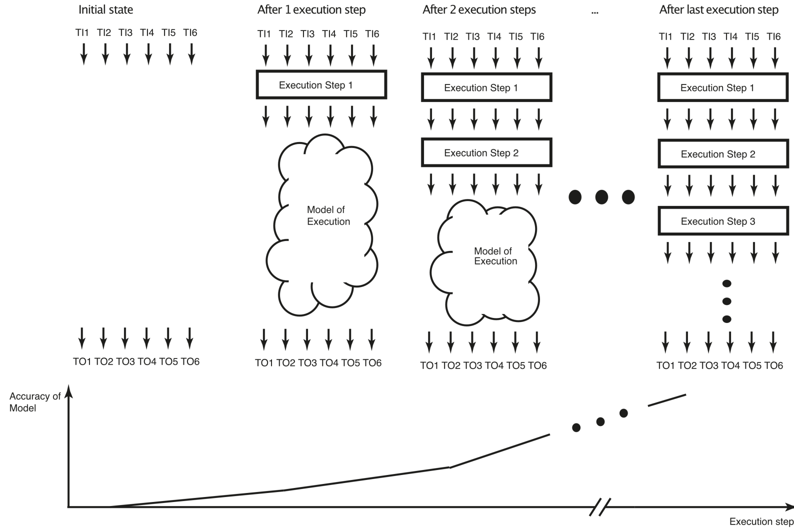


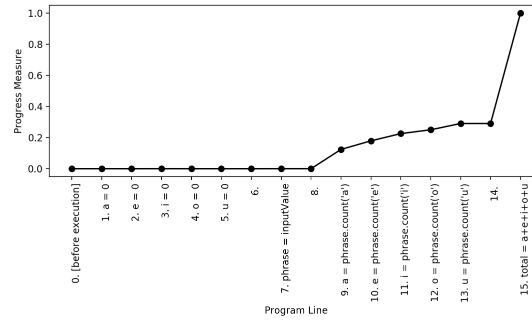
Fig. 1. Measuring the progress of a program towards its task, by building a model of the remaining computation at the end of each line in the program.

cases will be deleted, thus making the learned model *more* complex. By looking for these "spikes" in the model complexity/execution time graph, suspicious statements in the code can be identified for future examination by the programmer. This is illustrated by the example in Figure 2, which shows the difference in progress curves (calculated by the process described below) for a correct and faulty program.

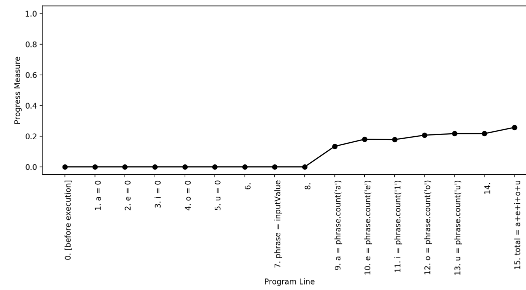
Importantly, this process can also be run on *incomplete* code. The system will only need to execute the actual code up to the current step of interest, the learned model substituting for the remainder of the processing. This is valuable because it can be used to test code which is actually incomplete, or where the latter part of the code is uncompilable. This also means that bug localisation can begin before a particular unit is finished. Furthermore, the common problem of masking [5], where bugs later in a program make it difficult to find the location of a bug, is not an issue for this approach because the portion of the code that would cause the masking is not executed, but modelled by the machine learning model.

3.2 Building the Rich Spectrum

We implement this in the following way. The inputs to the process are a piece of program code (the final line of which calculates a single integer value that is the output from the code), and a set of test cases that are in the form of input-output pairs.



(a) Progress for a program without fault



(b) Progress for a program with a typo (line 11)

Fig. 2. Progress measures for a program without a fault and with a small typographical error.

Firstly, we scan the program for variable names, and make a list of these. We remove any variable names that represent libraries etc. For the purposes of this paper, these variables will all be integers, but extensions to other data types are readily possible. We then create a rank 3 tensor $S(t, l, v)$, indexed by the test cases, the lines of code in the program, and the variable names discovered in that initial pass. Let us call S the *execution trace spectrum* of the program with regard to that test case set.

We then fill values into S in the following way. For each test case, we run the following process. Create a sequence of program texts, the first one with the first line of the program, the second one with the first two lines of the program, and so on until the final sequence contains the whole program. For each of the programs in this sequence, we run them on the current test case, and calculate any variable value that has changed due to the execution of that line.

We call the output of this the rich spectrum because it is calculated in the same way as a program spectrum in traditional spectral testing [21], but each line contains a richer set of information. A related idea is the execution trace

Algorithm 1 Construct the Rich Spectrum

```
 1: procedure ConstructExecutionTraceSpectrum(P, T)
                                ▷ P is the program text, T the set of test-case pairs
 2:     let V be the set of variable names in P
 3:     let n_t ← |T|
 4:     let n_v ← |V|
 5:     let n_p ← number of lines in P
 6:     create 3-d array S with dimensions (n_t, n_p, n_v)
 7:     for t ∈ T do
 8:         for ℓ ∈ [1, n_p] do
 9:             let v_n ← NULL
10:             let P_ℓ ← first ℓ lines in P
11:             execute P_ℓ with input from T
12:             if the last line of P_ℓ sets a variable value then
13:                 let v_n be the variable name that is set in the last line of P_ℓ
14:                 let v_c be the variable value that is set in the last line of P_ℓ
15:                 let S(t, ℓ, v_n) ← v_c
16:             end if
17:             for v ∈ V do (except when ℓ == 0)
18:                 if v ≠ v_n then
19:                     let S(t, ℓ, v) ← S(t, ℓ − 1, v_n)
20:                 end if
21:             end for
22:         end for
23:     end for
24:     return S
25: end procedure
```

spectrum by Harrold et al. [9], but that is focused on tracking the text of the lines of code visited rather than the values computed by those lines.

We now use this to calculate a vector of numbers, the *progress measure*, with one number in the interval $[0.0, 1.0]$ for each line in the code. As discussed above (and illustrated in Figure 1), the key idea is to take the rich spectrum as computed above, and for each line to take the variable values for each test case, and the test case outputs, and build a probabilistic model of the dependencies between them. In this case, the model is the Naïve Bayes classification model; for each line in the code, a Naïve Bayes model is produced that takes the current variable values, and tries to predict the output. The progress measure associated with each line is the accuracy of its Naïve Bayes model on a reserved test set. Pseudocode for this is given in Algorithm 2.

4 Applying this Model to Fault Localisation

We now have a notion of identifying a *progress model* with a piece of code and associated test case set. For a correct program, we would expect this progress model to output a monotonically increasing vector of values of the progress metric—

Algorithm 2 Construct the Progress Vector

```
1: procedure ConstructProgressVector(P, T)
                    ▷ P is the program text, T is the set of test cases
2:     let S ← ConstructExecutionTraceSpectrum(P, T)
3:     let x_train ← []                                    ▷ empty list
4:     let y_train ← []
5:     let x_test ← []
6:     let y_test ← []
7:     let V ← []
8:     for t ∈ T do
9:         let t_0 ← output value of t
10:        if random() > 0.1 then
11:            let X_train ← S(t, ℓ, ∗)      ▷ slice of the spectrum for line ℓ
12:            let y_train ← output value of t
13:        else
14:            let X_test ← S(t, ℓ, ∗)
15:            let y_test ← output value of t
16:        end if
17:        let c ← run Naïve Bayes classifier on X_train, y_train
18:        append accuracy(c) to V
19:    end for
20:    for ℓ ∈ [2, |P|] do
21:        if P[ℓ] is a comment, a blank line, or a library import then
22:            let V[ℓ] ← V[ℓ − 1]
       ▷ if the line makes no substantive computation, copy value from previous line
23:        end if
24:    end for
25:    return V
26: end procedure
```

each line executed produces task-relevant information, and so the accuracy of the mode will increase because it has more task-relevant information.

In this section of the paper, we look at how this can then be used to find the location of faults. The kind of faults we are interested in here are the minor typographic faults that are commonly made by programmers—typing the wrong variable name, typing the wrong operator value, mixing up 0 and o or 1 and i or l, or copying-and-pasting and then failing to make the required changes. These are the kinds of faults that the competent programmer hypothesis [7] predicts will be common—most of the time, a competent programmer will not make egregious errors of logic or language syntax, but minor typographical errors and "brainos" will remain regardless of high-level competence.

Our approach looks for two kinds of departures from monotonicity in the progress vectors. The first approach finds the earliest (large) downtick in the progress vector. The second approach finds the largest downtick. In the results, we present both of these.

Algorithm 3 Find the Fault Location

```
1: procedure FindFaultLocation(P, T, m, n_r)
     ▷ P is the program text, T is the set of test cases, m is the method used, n_r the number of runs
2:     let F ← [0, 0, ..., 0] of length |P|
3:     for r ∈ [1, n_r] do
4:         let V ← ConstructProgressVector(P, T)
5:         append 0.0 to start of V
6:         if thenm == "first substantive downtick"
7:             for dov ∈ V[2 : |V|]
8:                 if thenV[v] − V[v − 1] < −0.025
                                     ▷ is decreasing, and not by a trivial amount
9:                     let F[v] ← F[v] + 1
10:                    let a ← True
11:                    break
12:                end if
13:            end for
14:        end if
15:        if thenm == "largest downtick"
16:            let d ← inf
17:            let arg_d ← NULL
18:            for dov ∈ V[2 : |V|]
19:                if thenV[v] − V[v − 1] < d
                                     ▷ is decreasing by more than previous lines
20:                    break
21:                end if
22:            end for
23:            let F[arg_d] ← F[arg_d] + 1
24:        end if
25:    end for
26:    return a, argmax(F)
               ▷ return both whether a solution has been found, and the solution
27: end procedure
```

5 Experiments

Two experiments have been carried out to test the above ideas and algorithms. The first of these examines whether the program progress model accurately captures the progress of the example programs, and the second measures whether that same measure can be used to detect the location of faults.

Two python programs will be used in both experiments. These are given in Figure 3. These are designed to represent the kinds of simple data-transformation functions that are often written as units of a larger piece of code. The first counts the vowels in a piece of text, the second adds a vector of eight numbers together. For each program, 50000 random test cases are generated.

```
a = 0
e = 0
i = 0
o = 0
u = 0

phrase = inputValue

a = phrase.count('a')
e = phrase.count('e')
i = phrase.count('i')
o = phrase.count('o')
u = phrase.count('u')

total = a+e+i+o+u
```

```
a0 = inputValue[0]+inputValue[1]
a1 = inputValue[2]+inputValue[3]
a2 = inputValue[4]+inputValue[5]
a3 = inputValue[6]+inputValue[7]

b0 = a0+a1
b1 = a2+a3

c0 = b0+b1
```

Fig. 3. The programs used in the experiments: `VowelCounter.py` and `AddingNumbers.py`.

5.1 Experiment 1: Does the Program Progress Measure actually Measure Progress?

In this experiment we run the *ConstructProgressVector* algorithm (described above as Algorithm 2 on the four sample programs. We then measure whether the progress measure is monotonically increasing. Because there are occasionally minor fluctuations in the measure because of the Naïve Bayes process detecting random coincidences in otherwise uncorrelated data, we discount small downturns, where the downward difference between successive entries is less than 0.01. So, we refer to this behaviour as "near monotonic". The results for this experiment are presented in Table 1; the results show that the progress measure is capturing progress in the program.

Table 1. Measure of whether the progress measure is near monotonic (for 100 runs of the algorithm per program)

Program Name	Number of Near-monotonic Vectors
VowelCounter.py	100/100
AddingNumbers.py	100/100

6 Experiment 2: Can the Program Progress Measure Detect Faults?

In this experiment we run the *FindFaultLocation* algorithm (described above as Algorithm 3) on several erroneous versions of the programs from Figure 3. The

details of the errors introduced, and the results of the experiments, are detailed in Table 2. Overall, the results are positive; in most cases, the most common line identified as erroneous was the genuine erroneous line, and when it is not, it is in all but one case the second most chosen line.

Table 2. Error detection experiment (for 100 runs per error). The rank of the erroneous line (ranked by how many times out of 100 that line was chosen as the error) is also indicated.

Program	Change Made	Method	Error Found	Number of Correct Pred's	Rank of Best Pred.
VowelCounter.py	`i = phrase.count('i')` → `i = phrase.count('1')`	first	yes	39/100	1/15
VowelCounter.py	`i = phrase.count('i')` → `i = phrase.count('1')`	largest	yes	44/100	1/15
VowelCounter.py	`i = phrase.count('i')` → `i = phrase.count('e')`	first	yes	65/100	1/15
VowelCounter.py	`i = phrase.count('i')` → `i = phrase.count('e')`	largest	yes	93/100	1/15
VowelCounter.py	`i = phrase.count('i')` →`e = phrase.count('e')`	first	yes	9/100	1/15
VowelCounter.py	`i = phrase.count('i')` → `e = phrase.count('e')`	largest	no		2/15
VowelCounter.py	`total = a+e+i+o+u` → `total = a-e+i+o+u`	first	yes	42/100	1/15
VowelCounter.py	`total = a+e+i+o+u` → `total = a-e+i+o+u`	largest	yes	87/100	1/15
AddingNumbers.py	`b1 = a2+a3` → `b1 = a2+a1`	first	yes	94/100	1/9
AddingNumbers.py	`b1 = a2+a3` → `b1 = a2+a1`	largest	no		2/9
AddingNumbers.py	`b1 = a2+a3` → `b1 = a2-a3`	first	no		2/9
AddingNumbers.py	`b1 = a2+a3` → `b1 = a2-a3`	largest	no		2/9
AddingNumbers.py	`a2 = inputValue[4]+inputValue[5]` → `a2 = inputValue[2]+inputValue[5]`	first	no		2/9
AddingNumbers.py	`a2 = inputValue[4]+inputValue[5]` → `a2 = inputValue[2]+inputValue[5]`	largest	no		4/9

7 Limitations and Threats to Validity

One limitation of the current work is that the programs are fairly small and simple, and that they lack substantial use of loops and conditionals. However, there is no principled reason why these ideas cannot be applied to these more complex programs, and techniques such as that of Silva et al. [20] show how

the execution traces can be aligned between programs of different lengths of execution.

One threat to the wider applicability of these techniques is their reliance on a large number of test cases. For example, in the above experiments, 50000 test cases were used for each program. Initially, this would seem to render the approach useless (the so-called *test oracle* problem [2])—you need a program to correctly carry out the tasks in order to debug the program that is doing the task. In some cases, this might not be a problem, for example there might be a program in another language to generate the test problems, or some kind of dataset or physical phenomenon to generate the examples. But, it is more likely that an alternative approach to generating the test cases will be needed—for some problems, it is possible to calculate a set of inputs back from a target output, and in others it is is possible to use generate-and-test methods or machine learning driven by an oracle that can say whether a particular example is a valid [18].

One limitation of the current approach is that each test case set has a small number of possible outputs, which renders possible the use of a classification algorithm as our model for program progress. For more complex cases, the output might be continuous, multi-dimensional, or be of a complex data type. In such cases, other modelling approaches will be needed, for example replacing the classification model with a regression model.

8 Conclusions and Future Work

The probabilistic model used in this work is somewhat simplistic. By using the naïve Bayes model, we are assuming that all values calculated are equally and independently contributing towards the prediction of the output. This provides a first approximation for this, but in future it would be interesting to explore how the program progress measures proposed in this paper could be combined with more sophisticated models of program execution, such as probabilistic graphical models (as explored by Yu et al. [26]).

This paper has focused on providing candidate bug locations to human programmers. However, this work could also, in the future, support the growing research effort into automated bug *fixing* [15,16]. These automated bug-fixing systems rely heavily on automated methods for finding the location of bugs as their starting point, so methods such as the ones on this paper can feed directly into this important new direction in automated software development.

A related idea would be to use these progress measures to achieve program synthesis from scratch. Rather than beginning the program synthesis process from random code as is done in methods such as genetic programming [19], instead the system would synthesise a number of putative first lines, then use the program progress measure to ascertain which of those lines was making the strongest contribution to the problem as defined by the test cases. This line would then be fixed, and the system would move onto the next line, and so on with the aim of finding a monotonically increasing sequence of lines with the last one computing the output.

There are more general implications here for building AI systems. Usually, we build a machine learning model such as a classifier so that we can apply it to new data, and use the results of the classification. Here, we are using the success of the model as a proxy for some more complex task. This echoes the arguments recently made by Krawiec, Swan and O'Reilly [12,13,14], who use the complexity of a machine-learned model as a proxy for the difficulty of computing a given task.

Source code for the experiments can be found at http://www.colinjohnson.me.uk/researchSoftware.php

References

1. Androutsopoulos, K., Clark, D., Dan, H., Hierons, R., Harman, M.: An analysis of the relationship between conditional entropy and failed error propagation in software testing. In: Proceedings of the 36th International Conference on Software Engineering (ICSE) (2014)
2. Barr, E., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. IEEE Transactions on Software Engineering **41**(5), 507–525 (May 2015)
3. Briand, L.C., Labiche, Y., Bawar, Z.: Using machine learning to refine black-box test specifications and test suites. In: 2008 The Eighth International Conference on Quality Software. pp. 135–144 (2008)
4. Briand, L.C., Labiche, Y., Liu, X.: Using machine learning to support debugging with tarantula. In: The 18th IEEE International Symposium on Software Reliability (ISSRE '07). pp. 137–146 (2007)
5. Clark, D., Hierons, R.M.: Squeeziness: An information theoretic measure for avoiding fault masking. Information Processing Letters **112**(8–9), 335–340 (2012)
6. Cleve, H., Zeller, A.: Locating causes of program failures. In: Proceedings of the 27th International Conference on Software Engineering. pp. 342––351. Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1062455.1062522
7. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practicing programmer. IEEE Computer **11**(4), 34–41 (1978)
8. Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys **45**(1) (Dec 2012). https://doi.org/10.1145/2379776.2379787
9. Harrold, M.J., Rothermel, G., Sayre, K., Wu, R., Yi, L.: An empirical investigation of the relationship between spectra differences and regression faults. Software Testing, Verification and Reliability **10**(3), 171–194 (2000). https://doi.org/10.1002/1099-1689(200009)10:3¡171::AID-STVR209¿3.0.CO;2-J
10. Jones, J., Harrold, M.: Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. pp. 273—282 (2005)
11. Keller, F., Grunske, L., Heiden, S., Filieri, A., van Hoorn, A., Lo, D.: A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS). pp. 114–125 (2017)

12. Krawiec, K., O'Reilly, U.M.: Behavioral programming: A broader and more detailed take on semantic GP. In: Proceeding of the sixteenth annual conference on Genetic and evolutionary computation conference. ACM, New York, NY, USA (2014)
13. Krawiec, K., Swan, J.: Pattern-guided genetic programming. In: Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation. pp. 949–956. GECCO '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2463372.2463496, http://doi.acm.org/10.1145/2463372.2463496
14. Krawiec, K., Swan, J., O'Reilly, U.M.: Behavioral program synthesis: Insights and prospects. In: Genetic Programming Theory and Practice XIII. Springer (2015)
15. Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) International Conference on Software Engineering. pp. 3–13. IEEE (2012)
16. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: A generic method for automatic software repair. IEEE Transactions on Software Engineering **38**, 54–72 (2012)
17. Lutz, R.: Evolving good hierarchical decompositions of complex systems. Journal of Systems Architecture **47**(7), 613–634 (2001). https://doi.org/https://doi.org/10.1016/S1383-7621(01)00019-4, evolutionary computing
18. McMinn, P.: Search-based software test data generation: A survey. Software Testing, Verification and Reliability **14**(2), 105–156 (2004). https://doi.org/10.1002/stvr.294
19. Poli, R., Langdon, W.B., McPhee, N.F.: A Field Guide to Genetic Programming. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk` (2008), http://www.gp-field-guide.org.uk, (With contributions by J. R. Koza)
20. Silva, L., Paixão, K., de Amo, S., de Almeida Maia, M.: Software evolution aided by execution trace alignment. In: 2010 Brazilian Symposium on Software Engineering. pp. 158–167 (2010)
21. de Souza, H.A., Chaim, M.L., Kon, F.: Spectrum-based software fault localization: A survey of techniques, advances, and challenges (2016)
22. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Transactions on Software Engineering **42**(8), 707–740 (2016)
23. Wong, W.E., Debroy, V.: A survey of software fault localization (2009), university of Texas at Dallas, Department of Computer Science, Technical Report UTDCS-45-09
24. Wong, W.E., Qi, Y.: Bp neural network effecitve fault localization. International Journal of Software Engineering and Knowledge Engineering **19**(04), 573–597 (2009). https://doi.org/10.1142/S021819400900426X
25. Xie, X., Kuo, F.C., Chen, T.Y., Yoo, S., Harman, M.: Provably optimal and human-competitive results in SBSE for spectrum based fault localisation. In: Ruhe, G., Zhang, Y. (eds.) Search Based Software Engineering, Lecture Notes in Computer Science, vol. 8084, pp. 224–238. Springer Berlin Heidelberg (2013)
26. Yu, X., Liu, J., Yang, Z., Liu, X.: The bayesian network based program dependence graph and its application to fault localization. Journal of Systems and Software **134**, 44–53 (2017). https://doi.org/https://doi.org/10.1016/j.jss.2017.08.025