Mining Repositories to Assist in Project Planning and Resource Allocation

Tim Menzies
Department of Computer Science,
Portland State University,
Portland,
Oregon
tim@menzies.us

Justin S. Di Stefano, Chris Cunanan,
Robert (Mike) Chapman,
Integrated Software Metrics Inc.,
Fairmont, West Virginia
justin@lostportal.net, ccunanan@ismwv.com,
Robert.M.Chapman@ivv.nasa.gov

Abstract

Software repositories plus defect logs are useful for learning defect detectors. Such defect detectors could be a useful resource allocation tool for software managers. One way to view our detectors is that they are a V&V tool for V&V; i.e. they can be used to assess if "too much" of the testing budget is going to "too little" of the system. Finding such detectors could become the business case that constructing a local repository is useful.

Three counter arguments to such a proposal are (1) no general conclusions have been reported in any such repository despite years of effort; (2) if such general conclusions existed then there would be no need to build a local repository; (3) no such general conclusions will ever exist, according to many researchers. This article is a reply to these three arguments.

Submitted to the International Workshop on Mining Software Repositories (co-located with ICSE 2004) May 2004; http://msr.uwaterloo.ca.

1 Introduction

To make the most of finite resources, test engineers typically use their own expertise to separate critical from non-critical software components. The critical components are then allocated more of the testing budget than the rest of the system. A concern with this approach is that the wrong parts of the system might get the lions-share of the testing resource.

Defect detectors based on static code measures of components in repositories are a fast way of surveying the supposedly non-mission-critical sections. Such detectors can be a V&V tool for V&V; i.e. they can be used to assess if *too much* of the testing budget is going to *too little* of the system. As shown below, satisfactory detectors can be learnt from simple static code measures based on the Halstead [2] and Mccabes [3] features[1]. Such measures are rapid and simple to collect from source code. Further, the detectors learnt from these measures are easy to use.

Our experience with detect detectors has been very positive. Hence, we argue that organizations should routinely build and maintain repositories of code and defect logs. When we do so, we often hear certain objections to creating such repositories. This paper is our reply to three commonly-heard objections. For space reasons, the discussion here is brief. For full details, see [5, 6].

The first objection concerns a **lack of external validity**. Despite years of research in this area, there has yet to emerge standard static code defect detectors with any demonstrable external validity (i.e. applicable in more than just the domain used to develop them). Worse still, many publications argue that building detectors from static code measures is a very foolish endeavor [1, 7].

To counter the first argument, there has to be some demonstration from somewhere that at least once, another organization benefited from collecting such an endeavor. Paradoxically, making such a demonstration raises a second objection against local repository construction. If detectors are externally valid then organizations don't need *new data*. Rather, they can just *import data* from elsewhere. To refute this **buy not build** objection, it must be shown that detectors built from local data are *better than* detectors built from imported data.

Finally, if the proposal to build a repository survives objections one on two, then a third objection remains. Why is it that we make such an argument *now* when so many others have *previously* argued the opposite for so long? That is, it must be explained the **source of opposition** to static defect detectors.

The rest of this paper addresses these objections using the NASA case study described in the next section. Using that data, we show that external valid detectors can be generated. Next, we show that these detectors can be greatly improved using detectors tuned to a local project. Finally, we identity potential sources of systematic errors that may have resulted in prior negative reports about the merits of static code defect detectors.

2 Case Study Material

Our case study material comes from data freely available to other researchers via the web interface to NASA's Metrics Data Program (MDP) (see Figure 1). MDP contains around two dozen

[1] Elsewhere, we summarize those metrics [4]. Here we just say that Halstead measures reflect the density of the *vocabulary* of a function while Mccabe measures reflect the density of *pathways* between terms in the vocabulary.

Figure 1. The MDP repository: `http://mdp.ivv.nasa.gov`.

		module found in defect tracking log?	
		no	yes
signal detected?	no; i.e. $v(g) < 10$	A = 395 $LOC_A = 6816$	B = 67 $LOC_B = 3182$
	yes i.e. $v(g) \geq 10$	C = 19 $LOC_C = 1816$	D = 39 $LOC_D = 7443$

$$\begin{array}{rrr} Acc = & accuracy = & 83\% \\ PF = & Prob.falseAlarm = & 5\% \\ PD = & Prop.detected = & 37\% \\ prec = & Precision = & 67\% \\ E = & effort = & 48\% \end{array}$$

Figure 2. A ROC sheet assessing the detector $v(g) \geq 10$. Each cell {A,B,C,D} shows the number of modules, and the lines of code associated with those modules, that fall into each cell of this ROC sheet.

static code measures for thousands of modules based on the Halstead and McCabe measures. The data also include defect counts seen in up to eight years of project data.

From that data, various *data mining* [9] techniques have been applied to automatically build detectors. The output of these learners were compare to detectors generated by a DELPHI approach; i.e. asking experienced test engineers what thresholds they use to identify problematic code. These DELPHI predictors return "true" or "false" if some code measure passes some value.

The LSR and M5 data miners build predictors for the number of defects expected in new modules [9]. LSR uses linear standard regression to fit a single multi-dimensional linear model to the continuous detect data. For example, LSR generates equation such as Equation 1 below:

$$\begin{array}{rcl} defects_1 & = & 0.231 + (0.00344 * N) + (8.88e-4 * V) \\ & & -(0.185 * L) - (0.0343 * D) - (0.00541 * I) \\ & & +(1.68e-5 * E) + (0.711 * B) \\ & & -(4.7e-4 * T) \\ c_1 & = & -0.3616 \end{array} \quad (1)$$

Here, $\langle N, V, L, D, I, E, B, T\rangle$ are the *derived Halstead metrics* discussed in [4] and c_1 is the *correlation* of $defects_1$ to the actual error per module count. Correlation is discussed further below.

While LSR generates one equation, the M5 data miner can generate systems of equations. M5 is an extension of LSR that divides the data into a small number of regions and fits one linear model to each region.

Two other data miners used in this study were the J48 [9] ROCKY [4]. J48 is a standard decision tree learner and ROCKY is a home-brew learner than used a Gaussian approximation to propose interesting divisions of numeric data. ROCKY generates detector for each number attribute a of the form $a \geq N$ where $\geq N$ covers $\alpha\%$ of the Gaussian area. N is set such that:

$$\alpha \in \{0.05, 0.1, 0.15, \ldots, 0.95\} \quad (2)$$

ROCKY is a very simple learner that was run on subsets of the available data; i.e. just on the Mccabes data; just on the Halstead data; or just on simple lines of code (LOC) counts per module.

ROCKY and J48 process *discrete* defect classes. To generate discrete defect data, we took numeric defect counts per module and declared predicted defects "true" if $\#defects \geq 1$. In order to compare M5 and LSR to ROCKY and J48, the M5 and LSR output was converted to discrete booleans as follows. If M5 or LSR's predictions passes some threshold T, then predicted defects was set to "true". Our experiments repeated that test for:

$$T \in \{0.3, 0.6, \ldots, 3\} \quad (3)$$

The predictors generated by these methods were assessed via several assessment metrics. The *accuracy*, or Acc, of a detector as the number of true negatives and true positives seen over all events. In terms of the cells $\langle A, B, C, D\rangle$ shown in Figure 2, accuracy is $Acc = \frac{A+D}{A+B+C+D}$.

Apart from accuracy, several other measures are of interest. The *probability of detection*, or "PD", is the ratio of detected signals, true positives, to all signals; i.e. $PD = \frac{D}{B+D}$ (PD is also called the $recall$ of a detector). Also, the *probability of a false alarm*, or "PF", is the ratio of detections when no signal was present to all non-signals: i.e. $PF = \frac{C}{A+C}$. Further, the $precision$ of a detector comments on its correctness when it is triggered; i.e. $prec = \frac{D}{C+D}$.

Another statistic of interest is the $effort$ associated with a detector. If the detector is triggered, then some further assessment procedure must be called. For the particular static code defect detectors discussed in this paper, we will assume that this effort is proportional to the lines of code in the modules. Under that assumption, the $effort$ for a detector is what percentage of the lines of code in a system are selected by a detector; i.e. $effort = E = \frac{LOC_C + LOC_D}{LOC_A + LOC_B + LOC_C + LOC_D}$.

$Correlation$ is a statistic representing how closely two variables co-vary. Let a_i and p_i denote some actual and predicted values respectively. Let n and $\overline{x}$ denote the number of observations and the mean of the n observations, respectively. Then:

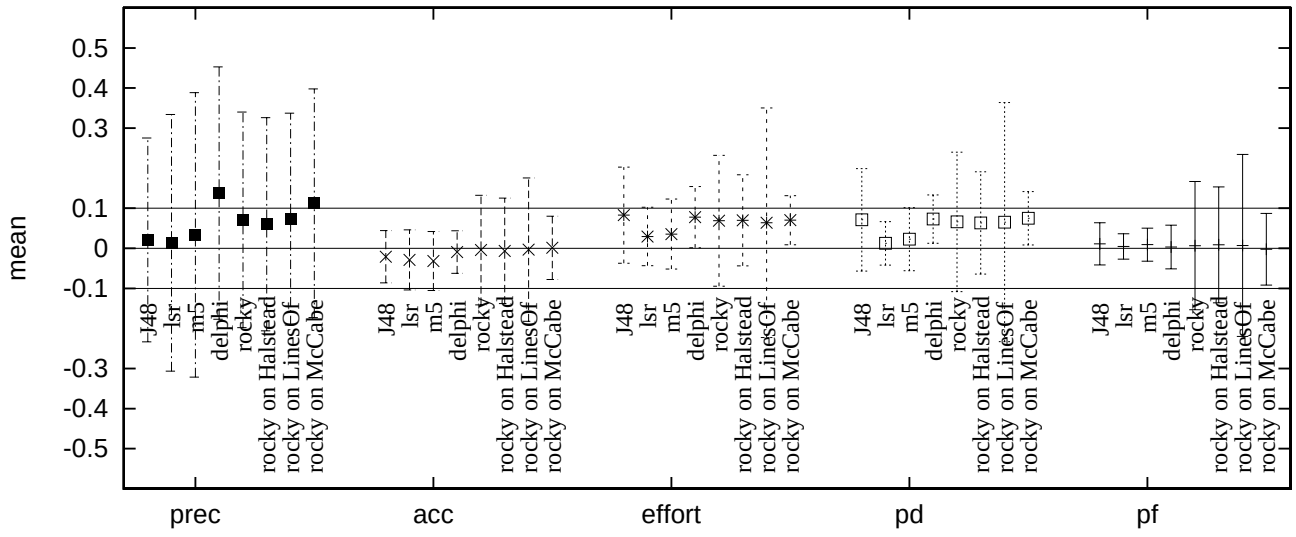

Figure 3. Between project stabilities in defect detectors. Mean μ and standard deviation σ of changes in defect detector statistics. Dots denote mean (μ) values. Whiskers extend from $\mu + \sigma$ to $\mu - \sigma$.

$$S_{PA} = \frac{\sum_i (p_i - \overline{p})(a_i - \overline{a})}{n-1};\ S_p = \frac{\sum_i (p_i - \overline{p})^2}{n-1};\ S_a = \frac{\sum_i (a_i - \overline{a})^2}{n-1}$$
$$correlation = c = \frac{S_{PA}}{\sqrt{S_p S_a}}$$

Correlation varies from -1 (perfect negative correlation) through 0 (no correlation) to +1 (perfect positive correlation). For example, the following equation, found via LSR using just lines of code LOC counts, has a very different correlation c to Equation 1 shown above:

$$\begin{aligned} defects_2 &= 0.0164 + 0.0114 * LOC \quad (4) \\ c_2 &= 0.65 \end{aligned}$$

3 Lack of External Validity?

To test the external validity of our detectors, we took five NASA applications, then learnt detectors from each of them using $\langle DELHI, LSR, M5, J48, ROCKY \rangle$. Because of Equation 2 and Equation 3, this resulted in hundreds of detectors. All these detectors were then applied to the other four applications.

As predicted by the **lack of external validity** objection, the detectors behaved differently when applied to the different applications. Figure 3 shows the mean and standard deviation of the *differences* in the values when *the same* detector was applied to *different* applications. For some learners and some assessment metrics, the observed standard deviations were quite large. For example, precision varied wildly and the variance in detectors built from module $linesofcode$ was always large.

However, in stark contrast to the **lack of external validity** objection, the differences were mostly very small. For example, with the exception of precision, most of the differences in the means were ≤ 0.1; and some learners consistently generated detectors with a very small variances (e.g. LSR,J48,DELPHI). To place Figure 3 in perspective $\langle pd, pf, acc, effort, prec$ all vary from zero to one so the *difference* between two (e.g.) pd values can vary from -1 to +1.

4 Buy, not Build?

The results Figure 3 come from a very varied set of applications. While all the studied applications used C or C++, they were built at four different locations around the country by five different teams for five very different application areas (ground station telemetry processing, flight software for earth orbiters, simulation tools for making predictions about hardware behavior, etc).

If defect detectors are so stable across domains, then the **buy not build** objection states that we need not build our own local repository. Instead, we need only reuse detectors learnt elsewhere.

Figure 4 is our reply to this objection. That figure shows results from learning detectors at *different* times in the life cycle of *the same* application. Defect logs were extracted at 6,12,18,24, and 48 months into the development of one of our applications. Defect detectors learnt at $time < X$ were applied to source code developed at $time \geq X$. Figure 4 shows the mean and standard deviations of the *differences* in $\langle pd, pdf, effort, acc, prec \rangle$ seen when *the same* detector was applied at *different times* to the *the same* application. Compared to Figure 3, the mean differences and standard deviations are greatly reduced.

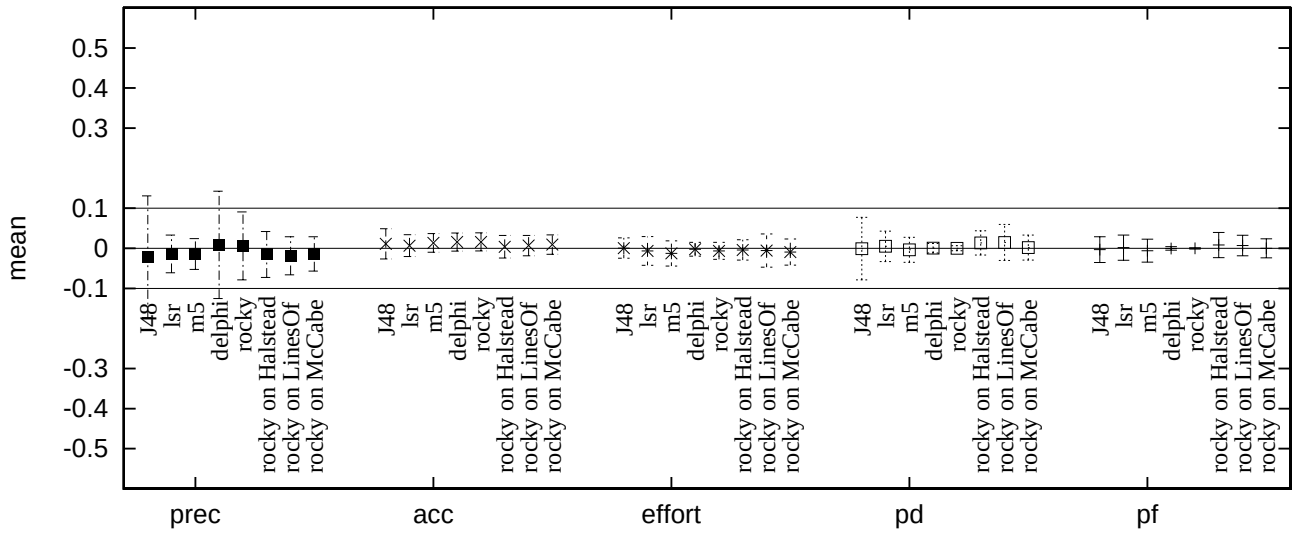

Figure 4. Within-project stabilities in defect detectors (same format as Figure 3.

5 Source of Opposition?

Figure 3 showed that defect detectors from other applications are *stable*; i.e. provide nearly the same results when applied to the current applications. Further, Figure 4 shows that detect detectors learnt from an historical record of the current application are *stabler*. This report is hence very positive on the merits of using repositories to build static code defect detectors.

Other researchers are not as positive. This section reviews some of those critiques and offers several source of systematic errors that may explain prior negative results in this area.

There are many reasons to doubt the merits of static code measures such as the Halstead/Mccabe metrics. Such metrics collected from a single module know neither (a)how often that module will be called nor (b)the severity of the problem resulting from the module failing nor (c)the connections *from* this module *to* other modules. Also, static code measures are hardly a complete characterization of the internals of a function. Fenton offers an insightful example where *the same* functionality is achieved using *different* programming language constructs resulting in *different* Mccabe measurements for that module [1]. Worse still, certain empirical evidence suggests that the Mccabe metrics might be no more informative than more simplistic measures. For example, Fenton & Pleeger note that cyclomatic complexity is highly correlated with lines of code [1]. Sheppard & Ince remarks that "for a large class of software it is no more than a proxy for, and in many cases outperformed by, lines of code" [7].

In reply to this pessimism, we take care to distinguish between *primary* and *secondary* detect detectors. We endorse standard practice in which test engineers *primarily* use their domain knowledge and the available documentation to identify the modules that require most of their attention. Our detectors are only *secondary* tools to quickly survey the parts of the system that were ruled out by the primary methods. If our secondary detectors trigger, then we would suggest that test engineers divert a little of the resources allocated by the primary method to check some other regions.

Primary detectors need a high probability of detection (pd). For all the reasons listed by Fenton & Pleeger and Sheppard & Ince, it is clear that defect detectors learnt from Halstead/Mccabe-style static code measures may not yield high pds. However, static code defect detectors are satisfactory secondary detectors. An important property of a secondary detector is a low probability of false alarm pf. Such low false alarms are required to ensure test engineers are not inappropriate distracted from their inspections of modules selected by the primary detectors. The bottom plot of

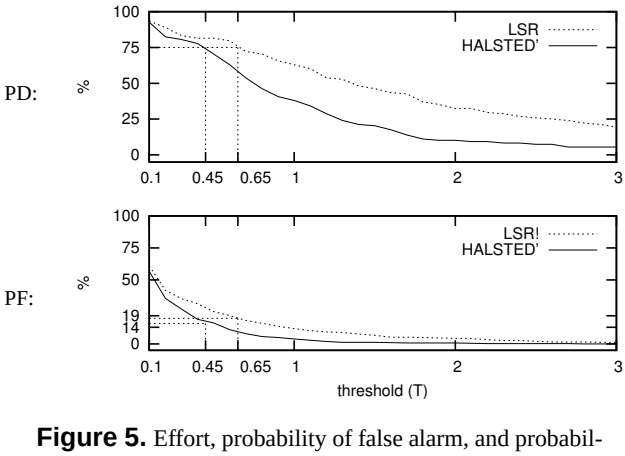

Figure 5. Effort, probability of false alarm, and probability of detection seen using $defects_i \geq T$ where $defects_i$ is one of Equation 1 (the "HALSTEAD" curves) or Equation 4 (the "LSR" curves) and T controls when the detector triggering (see the discussion around Equation 3).

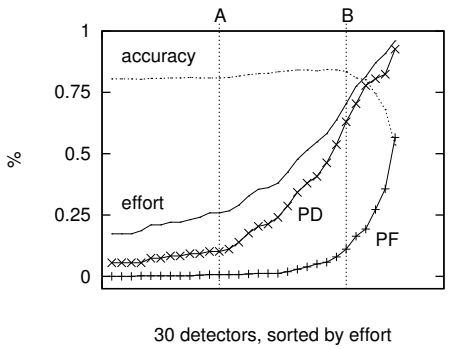

Figure 6. Each x-axis point x describes the $pf, pd, effort, accuracy$ of one detector.

Figure 5 shows the range of pf as a function of a detection threshold constant T. Note that by selecting T appropriately, detectors can be created with a pf that are so low that, if they are triggered by module X, then it becomes nearly certain that there is a defect in module X.

Empirically, detectors with low pfs also have low pds. For example, in Figure 5, a detector with a $pd = 75\%$ has a pf of around 20% and higher pfs have higher pds. This is to be expected since, as discussed above, static code defect detectors are ignorant of many features of an application. Hence, it is important that secondary detectors are paired with primary detectors since the latter has the greatest chance of finding bugs in the inspected regions. Similarly, primary detectors should be paired with low pf secondary detectors. While we *hope* test engineers are the most effective defect detectors, the available empirical evidence is, at best, anecdotal [8]. As shown here, much is known about the $\langle pf, pd, accuracy, effort, precision\rangle$ of static code measures. Further, they are cheap to build and easy to run over large code libraries. Hence, they are a useful way to check that aren't test engineers look in the wrong place.

As to Fenton & Pleeger's and Sheppard & Ince's comments about the merits of lines of code vs more complex measures such as Halstead/Mccabe, we saw above that lines of code can generate detectors with large variances across different applications (recall the between-application results of Figure 3). Also, we take issue with their use of correlation to assess detectors. Recall that Figure 5 results come from two equations with very different correlations to number of defects: -0.3616 and 0.65 for Equation 1 and Equation 4 and (respectively). Either equation can reach some desired level of detection, *regardless of their correlations*, merely by selecting the appropriate threshold value. For example, a $pd = 75\%$ can be reached using either method by setting $T \geq 0.65$ or $T \geq 0.45$.

More generally, we have found several commonly use assessment metrics to be uninformative about defect detectors. The problematic assessment measures are correlation, precision, and accuracy. Figure 3 showed that precision can vary wildly while other measures are more stable. Figure 5 showed that correlation can be insensitive to other measures like pd. Figure 6 shows a problem with accuracy. In that figure, hundreds of our detectors are shown sorted on increasing effort. Consider the detectors marked A and B on Figure 6. These two detectors have nearly the same accuracy, yet with $effort$s, PDs, and PFs that vary by factors as high as 4. That is, accuracy can be uninformative regarding issues of pf, pd, and $effort$.

In summary we are positive about static code defect detectors and others are not for several reasons. Firstly, we as negative as others about the merits of static code measures as a *primary* defect detection method. However, we are very positive about using static code defect detectors with low pfs as *secondary detectors* which can *augment* some other detection method. Secondly, we can demonstrate stable pf results across multiple applications. That is, if our secondary detectors trigger then it is highly unlikely that they are incorrectly reporting a detect. Thirdly, prior criticisms may be passed on problematic assessment measures such as correlation. We recommend using pf, pd, and $effort$ to assess detectors.

A drawback to this analysis is the sample size. While our work is based on a larger sample that some other publications in this area, more data is always better. We plan to frequently re-sample NASA's metrics data repositories to check our conclusions. This ability to revisit and revise old conclusions about software engineering is an important benefit of public domain code+defect repositories such as NASA's MDP program.

References

[1] N. E. Fenton and S. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. International Thompson Press, 1997.

[2] M. Halstead. *Elements of Software Science*. Elsevier, 1977.

[3] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

[4] T. Menzies, J. D. Stefano, K. Ammar, K. McGill, P. Callis, R. Chapman, and D. J. When can we test less? In *IEEE Metrics'03*, 2003. Available from `http://menzies.us/pdf/03metrics.pdf`.

[5] T. Menzies and J. S. D. Stefano. How good is your blind spot sampling policy? In *2004 IEEE Conference on High Assurance Software Engineering*, 2003. Available from `http://menzies.us/pdf/03blind.pdf`.

[6] T. Menzies, J. S. D. Stefano, C. Cunanan, and R. M. Chapman. The business case for defect logging. In *IEEE Transactions Software Engineering (submitted)*, 2004.

[7] M. Shepped and D. Ince. A critique of three metrics. *The Journal of Systems and Software*, 26(3):197–210, September 1994.

[8] F. Shull, V. B. ad B. Boehm, A. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz. What we have learned about fighting defects. In *Proceedings of 8th International Software Metrics Symposium, Ottawa, Canada*, pages 249–258, 2002. Available from `http://fc-md.umd.edu/fcmd/Papers/shull_defects.ps`.

[9] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.

Acknowledgements: The work was sponsored by the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program led by the NASA IV&V Facility under NASA contract NCC2-0979 and NCC5-685.