How Good is Your Blind Spot Sampling Policy?

Tim Menzies[‡], Justin S. Di Stefano[†]
[‡]Lane Department of Computer Science & Electrical Engineering, West Virginia University
[†]Galaxy Global Corporation, Fairmont, West Virginia, and WVU CSEE
tim@menzies.us, justin@lostportal.net

Abstract

Assessing software costs money and better assessment costs exponentially more money. Given finite budgets, assessment resources are typically skewed towards areas that are believed to be mission critical. This leaves blind spots*: portions of the system that may contain defects which may be missed. Therefore, in addition to rigorously assessing mission critical areas, a parallel activity should sample the blind spots. This paper assesses defect detectors based on static code measures as a blind spot sampling method. In contrast to previous results, we find that such defect detectors yield results that are stable across many applications. Further, these detectors are inexpensive to use and can be tuned to the specifics of the current business situations.*

1 Introduction

High assurance software requires extensive and expensive assessment. There are many forms of software assessment, ranging from manual inspections to automatic formal methods. These assessment methods differ in their effectiveness and the effort required to apply them. Typically, the more effective methods are more expensive. Hence, project managers often skew the assessment resources and apply more effort where that extra effort might be most useful.

If most of the assessment effort explores project artifacts *A,B,C,D*, then that leaves a *blind spot* in *E,F,G,H,I,...*. Blind spots can compromise high assurance software. Leveson remarks that in modern complex systems, unsafe operations often result from an unstudied interaction between components [4]. Lutz and Mikulski [6] found one such interaction in NASA deep-space satellites: mission critical anomalies of *flight software* can result from errors in *ground software* that fails to correctly collect data from the flight systems. This paper is about how to assess methods for sampling blind spots.

An alternative to our proposal is to remove the blind spots by better assessing the entire system. This is impractical. Blind spots are unavoidable and result from fundamental properties of software assessment and the economics of software development. Software assessment budgets are finite while assessment effectiveness increases exponentially with assessment effort. For example:

Black box probing: A *linear* increase in the confidence C that we have found all defects can take *exponentially* more effort. For example, for one-in-a-thousand detects, moving C from 90% to 94% to 98% takes 2301, 2812, and 3910 black box probes (respectively)[1].

Automatic formal methods: The infamous state space explosion problem imposes strict limits on how much a system can be explored via automatic formal methods [9].

Other methods: Lowry et.al. [5] and Menzies and Cukic [8] offer numerous other examples where assessment effectiveness is exponential on effort.

Exponential costs quickly exhaust finite resources. Hence, the blind spots can't be removed, and must be managed. Our proposal is to mix assessment methods. Standard practice is to apply the best available assessment methods on the sections of the program that the best available domain knowledge declares is most critical. We endorse this approach. Clearly, the most critical sections require the best known assessment methods. However, this focus on certain sections can blind us to defects in other areas. Therefore, standard practice should be augmented with a *lightweight*

[0]Submitted to the 8th IEEE International Symposium on High Assurance Systems Engineering March 25-26, 2004, Tampa, Florida http://hasrc.csee.wvu.edu/hase04. WP ref: 03/hase/hase November 5, 2003. Available on-line at http://menzies.us/pdf/03sample.pdf.

[1]A randomly selected input to a program will find a fault with probability x. After N random black-box tests, the chances of the inputs not revealing any fault is $(1-x)^N$. Hence, the chances C of seeing the fault is $1-(1-x)^N$ which can be rearranged to $N(C,x) = \frac{log(1-C)}{log(1-x)}$. For example, $N(0.90, 10^{-3}) = 2301$ [15].

project	# modules	% with defects	language	developed at	notes
CM1	496	9.7%	C	location 1	A NASA spacecraft instrument
JM1	10885	19%	C	location 2	Real-time predictive ground system: Uses simulations to generate predictions
KC1	2107	15.4%	C++	location 3	Storage management for receiving and processing ground data
KC2	523	20%	C++	location 3	Science data processing; another part of the same project as KC1; different personnel than KC1. Shared some third-party software libraries with KC1, but no other software overlap.
PC1	1107	6.8	C	location 4	Flight software for earth orbiting satellite
Total	15118				

Figure 1. Data sets used in this study

sampling policy to explore the rest of the system. This sampling policy will always be incomplete. Nevertheless, it is the only option when resources do not permit a complete assessment of the whole system.

For high assurance systems, the sampling policy must be carefully audited. For safety-critical or mission-critical software, it is best to use a sampling policy with known properties. For example, if historical logs tell us the probability that sampling policies S_1, S_2 require effort E_1, E_2, have probability of detecting detects PD_1, PD_2, and have probability of false alarms PF_1, PF_2, then software managers can make an informed choice about how to mix and match S_1 and S_2.

Here, we present a case study that details the effectiveness of creating sampling policies based on static code measures. Defect detectors based on static code measures have a bad reputation in the literature; e.g. by Shepperd & Ince [14] and Fenton&Pleeger [2]. For example, with Nikora we have criticized needlessly complicated analyses of such static measures [7]. Nevertheless, it is now our contention that the reason previous studies found static code detectors uninformative was due to small sample sizes and a limited number of statistics from which to assess them. With our larger sample size and greater number of statistics, we can clearly and precisely demonstrate the advantages and disadvantages of static code defect detectors.

For example, Figure 2 shows the kind of analysis enabled by this paper. The solid lines were generated by repeating the following $worst$ sampling procedure, thirty times. $X\%$ of the modules (a.k.a. "C" functions of "C++" methods) in each of the datasets of Figure 1 where selected at random for $X \in \{0.2, 0.4, 0.6, 0.8\}$. Using historical logs of known defects, it was possible to compute for each X value, the probability of detection and false alarm, using the methods described in §3. Note that as more of the modules were selected (i.e. increasing X), the probability of detecting errors increased. However, using this *worst* sampling policy, as the detection probability increased, so did the the probability of making a mistake (see the bottom plot of Figure 2).

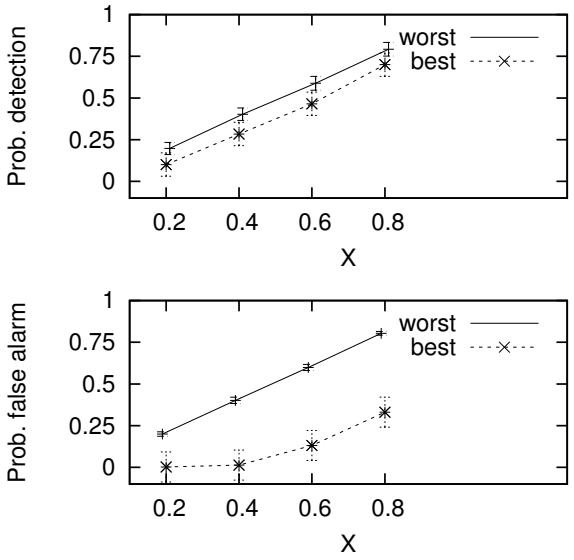

Figure 2. Comparing our results: whiskers denote $\pm 1\sigma$; i.e. plus or minus one standard deviation. The detectors that generate the *best* curve as shown in Figure 14.

The dashed lines of Figure 2 show the $best$ sampling policy found by this paper. After trying many different machine learners (linear regression, the M5' model tree learner [12]; the J48 decision tree learner [13]; and our own home-grown learner [10]) on several subsets of the available data from Figure 1 (just the McCabe's metrics; just the Halstead metrics; just the LinesOf metrics), we found detectors with very low false alarm rates. As shown in Figure 2, these detectors came at the cost of slightly reduced probability of detection. However, the very low false alarm rates of the *best* curves enable effective blind spot sampling. If most of the software assessment resources are being allocated to artifacts A, B, C, D, and the detectors of *best* curves of Fig-

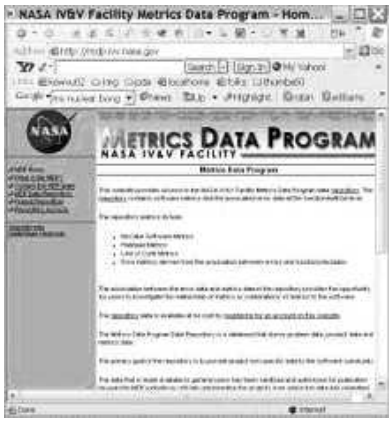

Figure 3. The MDP data repository.

		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 $LOC_D = 7443$

$$Acc = \quad accuracy = \quad 83\%$$
$$PF = \quad Prob.falseAlarm = \quad 5\%$$
$$PD = \quad Prop.detected = \quad 37\%$$
$$prec = \quad Precision = \quad 67\%$$
$$E = \quad effort = \quad 48\%$$

Figure 4. 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.

ure 2 are triggered by artifacts $E, F, G, H, I, ...$, then it is clear that the assessment resources should be reallocated.

The *best* curves in that figure were generated by learning detectors from one data set, then applying it to all the other data sets of Figure 1. The whiskers of Figure 2 show that the standard deviation of the *best* curve is very small. That is, our results are stable across multiple data sets. To the best of our knowledge, this is the first time that *any* defect detectors have been shown to be general to multiple projects.

Results like Figure 2 mean we offer a very detailed reply to the question "how good is a blind spot sampling policy based on static code detectors?". Our detectors are useful (low false alarm rates) and stable across multiple projects. We would encourage proponents of other methods to be so forthcoming.

The rest of this paper is structured is as follows. First, we explain our data sources and show how other researchers can access the same data. Next, we define the statistics we will collect on our defect detectors. Then we will discuss some general properties of detectors, followed by our methods and results. Finally, we will discuss our results, and finish with a conclusion on what this study says about blind spot sampling policies, and what direction future work could possibly take.

2 Public Domain Defect Data

An important feature of this study is that it is repeatable and hence refutable. The data sets used in this study are shown in Figure 1 and are freely available to other researchers via the web interface to NASA's Metrics Data Program (MDP) at `http://mdp.ivv.nasa.gov` (see Figure 3).

The MDP is funded by NASA's Software Independent Verification & Validation (IV&V) facility at Fairmont, West Virginia. The IV&V Facility is responsible for verifying that software developed or acquired to support NASA missions complies with the stated requirements. As the sole entity with the responsibility for IV&V of all NASA mission software, the IV&V Facility is in a unique position to create and maintain a master repository of software metrics such as the MDP.

Once NASA projects agree to distribution, then the sanitized data is made available to NASA, industry, and academia to support software development and research by other organizations. This is consistent with the IV&V Facilities research vision of "See more, learn more, tell more."

3 Definitions

This paper reviews detectors based on their *accuracy*, *probability of detection*, *probability of false alarm*, *precision*, and *effort*. For an example of all statistics, see Figure 4. Consider a detector which, when presented with some signal, either triggers or is silent. If some oracle knows whether or not the signal is actually present, then Figure 4 shows four interesting situations. The detector may be silent when the signal is absent (cell A) or present (cell B). Alternatively, if the detector registers a signal, sometimes the signal is actually absent (cell C) and sometimes it is present (cell D).

Figure 4 lets us define the *accuracy*, or Acc, of a detector as the number of true negatives and true positives seen over all events:

$$\text{accuracy} = Acc = \frac{A + D}{A + B + C + D}$$

If the detector registers a signal, there are three cases of interest. In one case, the detector has correctly recognized

the signal. This *probability of detection*, or "PD", is the ratio of detected signals, true positives, to all signals. PD is also called the $recall$ of a detector:

$$\text{probability detection} = PD = recall = \frac{D}{B+D}$$

In another case, the *probability of a false alarm*, or "PF", is the ratio of detections when no signal was present to all non-signals:

$$\text{probability false alarm} = PF = \frac{C}{A+C}$$

Lastly, the $precision$ of a detector comments on its correctness when it is triggered:

$$\text{precision} = prec = \frac{D}{C+D}$$

$PD, PF, Acc, Prec$ are also defined for numeric detectors. Give a numerical prediction, N, then this can be converted to a categorical detector by adding a threshold value X. Once expressed in this form, Figure 4 can be completed and $PD, PF, Acc, Prec$ calculated for this detector. For example, many of the case studies in this paper use the following threshold values:

$$X \in \{0.33, 0.43, \ldots 3\} \wedge if\ N \begin{cases} \geq X\ then\ trigger \\ < X\ then\ silent \end{cases} \quad (1)$$

Another statistic of interest is the $effort$ associated with a detector. Our model is that these detectors are alerts instructing us to place more effort in some part of the software. That is, 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 (this assumption is easily changed). Under that assumption, the $effort$ for a detector is what percentage of the lines of code in a system are selected by a detector.

$$effort = E = \frac{LOC_C + LOC_D}{LOC_A + LOC_B + LOC_C + LOC_D}$$

4 General Detector Properties

This section describes some general properties of the detectors that were generated during this research.

4.1 ROC Curves

Formally, a defect *detector* hunts for a *signal* that a software module is defect prone. Signal detection theory [3] offers *receiver operator characteristic* (ROC) curves as an analysis method for assessing different detectors. ROC curves are widely used in various fields including assessing different clinical computing systems [1] and assessing different machine learning methods [11]. The central intuition of ROC curves is that different detectors can be assessed via how often they correctly or incorrectly respond to the presence or absence of a signal.

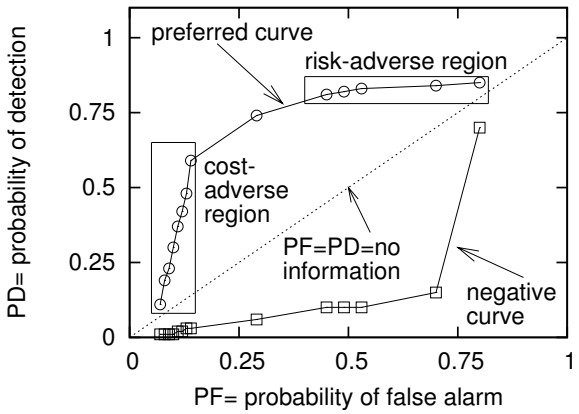

Figure 5. Regions of a typical ROC curve.

A typical ROC curve is shown in Figure 5. By definition, the ROC curve must pass through the points PF=0,PD=0 and PF=1,PD=1 (a detector that never triggers never makes false alarms; a detector that always triggers always generates false alarms). Between these two points, the curve can take three interesting trajectories:

1. A straight line from (0,0) to (1,1) is of little interest since it offers *no information*: i.e. the probability of a detector firing is the same as it being silent.
2. The point (PF=0,PD=1) is the ideal position on a ROC curve. This is where we recognize all errors and never make mistakes. In practice, this point is never reached but some *preferred curves* bend up towards this ideal point.
3. Another trajectory is the *negative curve* that bends away from the ideal point. Our experience has been that these are detectors which, if their tests were negated, would transpose into a *preferred curve*.

In the ideal case, a detector has a high probability of detecting a genuine fault (PD) and a very low probability of false alarm (PF). This ideal case is very rare. In practice, engineers must trade-off between PF and PD.

4.2 Risk vs Cost-adverse Projects

One advantage of ROC curves is that they let us characterize two important and different types of software

projects. Defect detectors that fall into the *cost-adverse region* shown in Figure 5 have low probabilities of false alarms. Such defect detectors are best when the budget available for blind spot assessment is limited and the extra effort associated with chasing false alarms is unacceptable.

On the other hand, detectors that fall into the *risk-adverse region* of Figure 5 have high probabilities of detecting a signal. However, due to the usual relationship between PD and PF, this high probability comes at the cost of a high false alarm rate. Hence, detectors that fall into this region are best for safety-critical systems where the cost of chasing false alarms is out-weighed by the cost of system failure.

5 Methods

Using the datasets of Figure 1, we have built hundreds of detectors and computed their $PD, PF, Acc, Effort$ and $Prec$. This section describes our detector generation methods. In the next section, we review the repeated patterns of $PD, PF, Acc, Effort$ and $Prec$ seen in numerous NASA projects.

All our detectors are learnt from examples of the form:

$$LOC, M_1, M_2, M_3, H_1, \ldots, H_8, HD_1, \ldots, HD_8, LO_1, \ldots, LO_4 \longrightarrow \#defects \quad (2)$$

where LOC is lines of code per module (including blank lines and comments); M_i are the McCabe metrics (cyclomatic complexity, essential complexity, design complexity); H_i are the basic Halstead metrics (including unique and total number of operators and operands); HD_i are the Halstead metrics derived from H_i (including an estimate of required programming time); and LO_i are the LinesOf Metrics (Lines Of Code, Comment, Code And Comment, and Blank). Our MDP source offers $\#defects$ as a integer ≥ 0.

5.1 Delphi Detectors

The *traditional* method of generating detectors is to use the detectors recommended by McCabe, or to query experienced test engineers for their preferred detector(s). This type of query results in detectors like these:

$$v(g) \geq 10 \quad (3)$$

$$iv(g) \geq 4 \quad (4)$$

$$iv(g) \geq 4 \vee v(g) \geq 10 \quad (5)$$

5.2 LSR Detectors

Another method to generate a detector is linear standard regression or LSR. LSR is a standard statistical method that fits a straight line to a set of points. The line offers a set of predicted values. If the points are scattered, then a single regression line can't pass through each point. The distance from these predicted values to the actual values is a measure of the error associated with that line. Linear regression packages search for a line that minimizes that error and maximizes the correlation between predicted and actual values. LSR generates equation such as Equation 6 below (based on the derived Halstead metrics and $\#defects$):

$$\begin{aligned} defects_2 \quad = \quad & 0.231 + (0.00344 * N) + (8.88e-4 * V) \\ & -(0.185 * L) - (0.0343 * D) - (0.00541 * I) \\ & +(1.68e-5 * E) + (0.711 * B) \\ & -(4.7e-4 * T) \end{aligned} \quad (6)$$

5.3 Model Tree Detectors

A drawback with linear regression is that the *same* line is fitted through all points. That is, linear regression assumes that all the data comes from a single simple linear distribution. Where this is not true, it may be better to divide the space into different regions and then make a different decision about each region. There are various techniques for doing so, but the one we will use in this study is *model trees*.

A model tree is a decision tree with different linear regression equations at each leaf. Model trees are used like linear regression to generate defect detectors: if the prediction is p_i, then the detector is triggered when $p_i \geq X$. Using this method, the following detectors were generated by the $M5'$ model tree learner [12, 16] from the KC2 data set, using nearly all available features (basic and derived Halstead, McCabe's, but not LOC):

$$\left(\begin{array}{lll} if & N_2 \leq 49.5 & \\ then & \left\{ \begin{array}{ll} if & I <= 24.7 \\ then & 0.0375 \\ else & 0.284 \end{array} \right. & \\ else & \left\{ \begin{array}{ll} if & N_2 \leq 142 \\ then & 1.06 \\ else & \left\{ \begin{array}{l} 0.663- \\ 0.164 * ev(g)+ \\ 0.0128 * T \end{array} \right. \end{array} \right. & \end{array} \right) \geq X \quad (7)$$

5.4 Decision Tree Detectors

Model trees and LSR learn predictors for numeric classes. Decision tree learners like J4.8 from the WEKA

toolkit generate predictions for discrete classes [16]. Decision tree learners *split* the whole example set into subsets based on some attribute value test. The process then repeats recursively on the subsets. Each splitter value becomes the root of a sub-tree. Splitting stops when either a subset gets so small that further splitting is superfluous, or a subset contains examples with only one classification.

5.5 Rocky Detectors

J4.8 and model tree learning are state-of-the-art machine learning techniques. A much simpler technique is our *ROCKY* learner that exhaustively explores all singleton rules of the form

$$feature \geq threshold$$

Here, *feature* is every numeric feature present in a dataset and *threshold* is found as follows: Every numeric feature is assumed to come from a Gaussian distribution. $Threshold$s are then selected corresponding to equal areas under that distribution. For example, in one of the datasets we examine, $v(g)$ had a mean of $\mu = 4.9$ and a standard deviation of $\sigma = 11$. If this Gaussian is converted to a unit Gaussian (by subtracting the mean and dividing by the standard deviation), then standard Z-tables could be used to calculate a $v(g)$ $threshold$ value of 7.65 for an area of 0.6. *ROCKY* generates one detector

$$X \geq X.threshold(area) \tag{8}$$

for the range

$$\begin{aligned} X \in &\quad \{LOC, M_i, H_i, HD_i, LO_i\} \\ area \in &\quad \{0.05, 0.1, 0.15, \ldots 0.9, 0.95\} \end{aligned}$$

That is, the detector shown in equation 8 is really hundreds of different detectors.

6 Results

To generate our detectors, we applied the above methods to all the datasets in Figure 1; i.e. CM1, KC1, KC2, JM1, PC1. For each dataset, we applied LSR to LOC, to subsets of the attributes containing just the McCabe metrics, just the basic Halstead, or just the derived Halstead. Also, for all datasets, $M5'$, $ROCKY$ and $J4.8$ were applied to all available attributes (for $J4.8$, the numeric $\#defects$ was changed into a boolean for defects present/absent). For the $M5'$ and LSR detectors that generate numeric predictions, we collected statistics of 30 different variants with the threshold set to $X \in \{0.3, 0.6, \ldots 3\}$.

For each dataset, a diagram like Figure 6 was generated. Each x-axis of Figure 6 shows statistics from one detector. In that figure, all the detectors are sorted by $effort$.

The general shape of Figure 6 was seen in all our five datasets:

1. PD rises with $effort$ and rarely rises above it.
2. High PDs are associated with high PFs.
3. $PD, PF, effort$ can change significantly while $accuracy$ remains essentially stable.

The repeated nature of Figure 6 in different datasets prompted us to check the stability of detectors between datasets. For each data set, detectors were generated using the methods described in the previous section. The detectors learnt from dataset I were then applied to dataset $J (J \neq I)$. The differences Δ in the $PF, PD, Acc, prec$ and $effort$ between the training set I and the test set J were collected. Figure 7 shows the mean and standard deviation of all the Δ values generated from any training dataset I applied to any other test dataset J.

Referring to Figure 7, you can see that many of the Δ values for the various metrics are relatively stable. Some of the detector generation methods are more stable than others, but for almost every metric shown *some* detector generation method is stable. The sole exception to this is the $prec$ metric, which never demonstrates any type of stability whatsoever.

7 Discussion

In this paper, we detailed several different methods for generating defect detectors. These included LSR, Model Trees, Decision Trees, DELPHI and Rocky. Below is a short summary of the results of each of these methods in detail, along with some explanatory text on why we either recommend for or against these methods.

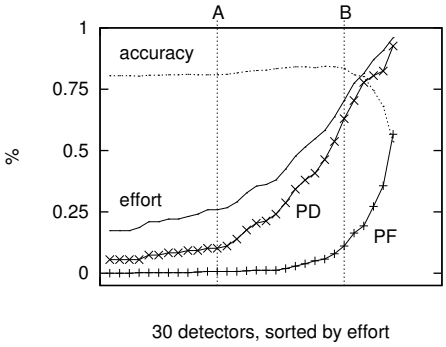

Figure 6. Properties of detectors of the form $defects_i > X_i$. Each x-axis point x describes the $PF, PD, effort, accuracy$ of one detector.

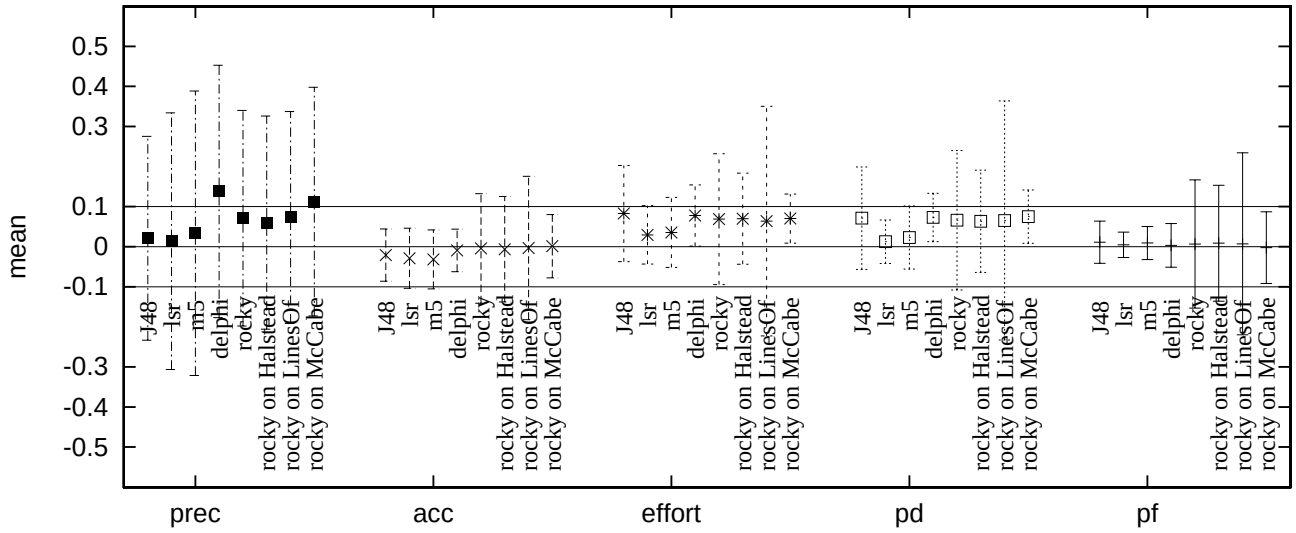

Figure 7. Mean μ and standard deviation σ of changes in defect detector statistics. Dots denote mean (μ) values. Whiskers extend from $\mu + \sigma$ to $\mu - \sigma$.

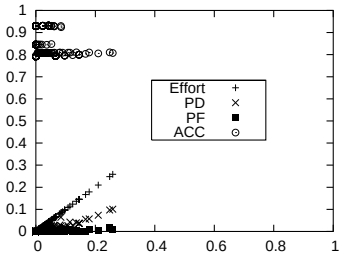

Figure 8. LSR on CM1 Data

7.1 LSR and Model Trees

For the majority of our data sets (3 out of 5), the curves for LSR and Model Tree learners resemble those of Figure 8 . As seen in this graph, these methods tend to cut out at higher effort values, which severly limits the type of detectors which can be generated. *Therefore, we recommend* **against** *using LSR or Model Trees as a basis for locating and choosing detectors.*

7.2 Delphi

Our first positive results are for our Delphi predictors. If you will recall from earlier, Delphi are merely expert(s) opinions on what *should* make a good defect detector. The results bear out the experts, as the Delphi curves all resemble Figure 9, which provides a good variety of defect detector choices[2]. *Therefore, we recommend* **for** *using Delphi as a basis for locating and choosing detectors.*

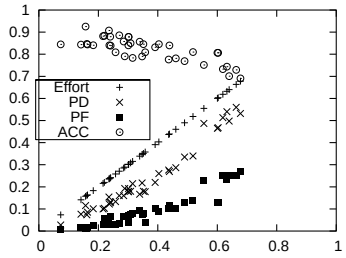

Figure 9. Delphi on KC2 Data

7.3 Decision Trees

For all data sets evaluated in this study, the J4.8 (Decision Tree) learner produced graphs similiar to Figure 10. Decision Tree learners, because of their discrete nature, generate only one point per data set. This *severely* limits their applicability to other data sets, and limits management options. *Therefore, we recommend* **against** *using J4.8 as a basis for locating and choosing detectors.*

[2]The Delphi curves always cut out at an effort of approximately 0.7, but this is not normally a problem, as the extremley high effort detectors are almost always associated with large PF values)

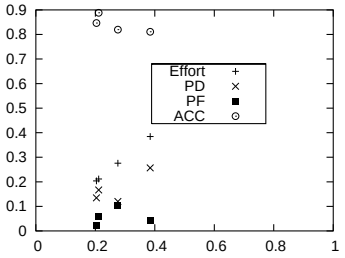

Figure 10. J48 Decision Tree Learner on JM1 Data

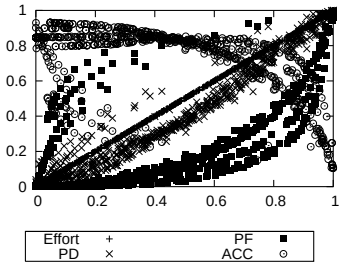

Figure 11. Rocky on Halstead metrics from JM1 Data

7.4 Rocky on Halstead Metrics

Recall that the Rocky learner generates twenty detectors for every attribute found in the dataset. Therefore, it is possible to break up the Rocky results into specific categories based on the attributes, specifically Halstead, McCabe and LinesOf. In the case of Rocky on the Halstead metrics, we get good ranges and a fairly standard curve. However, referring to Figure 11, notice that the PF values are higher than in other results, especially in the low effort ranges. Since this effect is not repeated when Rocky is run on the other metrics, this is obviously not the most successfull use of the Rocky learner. *Therefore, we recommend* **against** *using Rocky on Halstead metrics as a basis for locating and choosing detectors.*

7.5 Rocky on McCabe Metrics

Rocky on McCabe metrics produces a perfect example of a standard set of curves, as shown in Figure 12. From these curves, you can locate a detector for any business or project situation with relative ease. *Therefore, we recommend* **for** *using Rocky on McCabe metrics as a basis for locating and choosing detectors.*

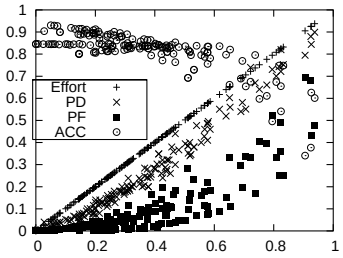

Figure 12. Rocky on McCabe metrics from JM1 Data

7.6 Rocky on LinesOf Metrics

Rocky on LinesOf metrics is another example of a nearly perfect set of curves, as demonstrated in Figure 13. *Therefore, we recommend* **for** *using Rocky on LinesOf metrics as a basis for locating and choosing detectors.*

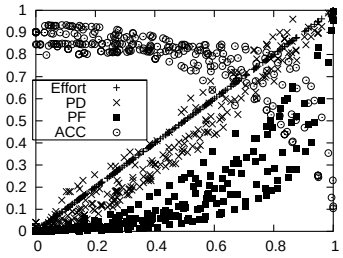

Figure 13. Rocky on LinesOf metrics from JM1 Data

8 External Validity

A unique conclusion of this paper is that, using our data sets, we can demonstrate the *stability* of detector properties across different software projects. For example, we can compare the probability of false alarms PF of our detectors when they are learnt from one software project and applied to another. Figure 7 is a summary of the stability of various learning methods over all our datasets. Using this type of analysis, it is possible to compare different blind spot assessment methods to one and another and conclusively demonstrate a winner. To the best of our knowledge, this is the first time that such a repeatable and refutable claim for the stability of defect detectors has been made in the literature. The detectors for *Rocky on McCabe* were shown in the introduction in Figure 2.

Method	Detector	Effort	PD	PF	ACC	PREC
Rocky On McCabe	$ev(g) \geq 14.508$	0.200	0.101	0.002	0.811	0.917
	$iv(g) \geq 9.231$	0.396	0.284	0.014	0.840	0.838
	$iv(g) \geq 3.655$	0.600	0.465	0.131	0.801	0.395
	$iv(g) \geq 1.640$	0.809	0.700	0.331	0.674	0.280
		$\sigma = 0.06$	$\sigma = 0.07$	$\sigma = 0.09$	$\sigma = 0.08$	$\sigma = 0.28$
Delphi	$v(g) > 20$	0.216	0.1	0.031	0.882	0.263
	$loc > 118$ OR $ev(g) > 7$	0.404	0.26	0.089	0.846	0.245
	$iv(g) > 4$	0.600	0.465	0.131	0.807	0.395
	$v(g) \geq 10$ OR $iv(g) \geq 4$	0.740	0.670	0.173	0.794	0.503
		$\sigma = 0.08$	$\sigma = 0.06$	$\sigma = 0.05$	$\sigma = 0.05$	$\sigma = 0.31$
Rocky On LinesOf	$loc \geq 133.575$	0.202	0.073	0.015	0.809	0.542
	$loc \geq 102.486$	0.403	0.217	0.052	0.807	0.502
	$loc \geq 50.691$	0.595	0.459	0.077	0.826	0.610
	$loc \geq 17.463$	0.800	0.673	0.270	0.721	0.313
		$\sigma = 0.29$	$\sigma = 0.3$	$\sigma = 0.23$	$\sigma = 0.17$	$\sigma = 0.26$

Figure 14. Means and standard deviation σ for various detectors recommeded by this text for $effort \approx \{0.2, 0.4, 0.6, 0.8\}$

9 Conclusion

Blind-spots present a significant hazard to any project, and can lead to devastating and time-consuming errors in later stages. It is our contention that by using the methods presented in this paper, a good defect detector can be found for *any* software project. By using these detectors, the chance of missing critical blind-spot errors can be drastically reduced, leading to better software.

In this paper, we have discussed one specific type of blind spot sampling policy; i.e. using defect detectors generated from static code measures. From the above discussion, we can draw the following general lessons about blind spot samplers:

- In the above analysis, our view of an "ideal detector" depended on the statistic used to assess the detector; e.g. PD, PF, effort, etc. *Therefore, it is important to assess detectors using multiple statistics.*

- The same detector used on different data sets generates different results. *It insightful to study the mean and standard deviation of these differences.* For example, Figure 14 shows the means and standard deviations of some of the detectors seen in this study.

- An ideal sampling policy has a low variation when applied to different sets; that is, given a list of those variations, *a good detector has a mean variation near zero and a small standard deviation of those variations*. The detectors in Figure 14 with a $\sigma < 0.1$ are highly recommeded by this study.

- *Once stability has been demonstrated, then it is possible to report general conclusions that should hold across many data sets.* Figure 14 shows some of the stable conclusions generated by this study.

- Stability across multiple data sets cannot be assessed without access to multiple data sets. *Therefore it is important to maintain a metrics repository to keep data from multiple projects.*

- Another assessment criteria explored above is the method used to generate the detector; e.g. LSR, Rocky, J4.8, etc. Some of these generators were shown to be inferior to others. That is, when building a library of detectors, *different methods of generating detectors should be explored*.

- Generators of defect detectors work from measurement types within a domain. In the above analysis, our types were LinesOf code measures, Halstead metrics and McCabe metrics. Some of these types turned out to be inferior to other types. Therefore, *defect detector generators should be assessed in combination with measurement types*.

As an example of the last two points, the above study made some specific comemnts about the role of different generator methods and measurement types for converting static code metrics into defect detectors:

- *Rocky On McCabe*: This is the best overall method, being both stable and providing a good range of detectors. If you either already own one of the McCabe

packages, or can afford to purchase one, then we recommend using it and the Rocky learner to tune defect detectors to your projects.

- *Delphi*: Although Delphi detectors tend to cut out at around an effort of 0.7, they are our second pick for defect detector generation. However, it is important to note that most of the Delphi detectors are based on the McCabe package; this limits the Delphi applicability to those companies which can afford the McCabe toolset.

- *Rocky on LinesOf*: Finally, although it is slightly less stable than the first two methods (see Figure 7), Rocky On LinesOf offers a good range and is extremley cheap and easy to collect. For any company or project which either cannot afford the McCabe package or does not wish to purchase it, then this is our recommended method.

An important feature of the above analysis is that it has compared results from different machine learners (linear regression, the M5' model tree learner [12]; the J48 decision tree learner [13]; and our own home-grown learner [10]) on several subsets of the available data from Figure 1 (just the McCabe's metrics; just the Halstead metrics; just the LinesOf metrics). Clearly, these are only (a) some of the ways to generate detectors; and (b some of the metrics from which we can generate detectors. We look forward to future research into other types of generators from other types of metrics. A primary feature of our analysis is that it provides a "proving ground" for new methods; by repeating our stability analysis (as per figure 7), any new assessment methods can be compared and contrasted with previous ones in a clear and consistent manner.

Acknowledgments

This research was conducted at West Virginia University under NASA contract NCC2-0979 and NCC5-685. 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. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government.

References

[1] K. P. Adlassnig and W. Scheithauer. Performance evaluation of medical expert systems using roc curves. *Computers and Biomedical Research*, 22(4):297–313, 1989.

[2] N. E. Fenton and S. Pfleeger. *Software Metrics: A Rigorous & Practical Approach (second edition)*. International Thompson Press, 1997.

[3] D. Heeger. Signal detection theory, 1998. Available from `http://white.stanford.edu/~heeger/sdt/sdt.html`.

[4] N. Leveson. *Safeware System Safety And Computers*. Addison-Wesley, 1995.

[5] M. Lowry, M. Boyd, and D. Kulkarni. Towards a theory for integration of mathematical verification and empirical testing. In *Proceedings, ASE'98: Automated Software Engineering*, pages 322–331, 1998.

[6] R. Lutz and C. Mikulski. Operational anomalies as a cause of safety-critical requirements evolution. *Journal of Systems and Software (to appear)*, 2003. Available from `http://www.cs.iastate.edu/~rlutz/publications/JSS02.ps`.

[7] T. Menzies, K. Ammar, A. Nikora, and J. DiStefano. How simple is software defect detection? In *Submitted to the Emprical Software Engineering Journal*, 2003. Available from `http://menzies.us/pdf/03simpled.pdf`.

[8] T. Menzies and B. Cukic. How many tests are enough? In S. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering, Volume II*, 2002. Available from `http://menzies.us/pdf/00ntests.pdf`.

[9] T. Menzies, J. Powell, and M. E. Houle. Fast formal analysis of requirements via 'topoi diagrams'. In *ICSE 2001*, 2001. Available from `http://menzies.us/pdf/00fastre.pdf`.

[10] 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`.

[11] F. Provost, T. Fawcett, and R. Kohavi. The case against accuracy estimation for comparing induction algorithms. In *Proc. 15th International Conf. on Machine Learning*, pages 445–453. Morgan Kaufmann, San Francisco, CA, 1998. Available from `htpp://citeseer.nj.nec.com/provost98case.html`.

[12] J. R. Quinlan. Learning with Continuous Classes. In *5th Australian Joint Conference on Artificial Intelligence*, pages 343–348, 1992. Available from `http://citeseer.nj.nec.com/quinlan92learning.html`.

[13] R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, 1992. ISBN: 1558602380.

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

[15] J. Voas and K. Miller. Software testability: The new verification. *IEEE Software*, pages 17–28, May 1995. Available from `http://www.cigital.com/papers/download/ieeesoftware95.ps`.

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