Learning Defect Predictors

Tim Menzies, *Member, IEEE*, Jeremy Greenwald, Art Frank

***Abstract*— The value of using static code measures to learn defect detectors has been widely debated. Contrary to prior pessimism, we show here such defect predictors are demonstrably useful. Such measures can yield predictors with a mean probabilities of detection of 71% and mean false alarms rates of 25%. These predictors have their limitations (non-zero false alarm rates) but would still be useful for (e.g.) prioritizing a resource-bound exploration of code that has yet to be inspect.**

I. INTRODUCTION

There are many forms of software assessment: manual inspections, automatic formal methods, etc. These methods differ in their effectiveness and the effort required to apply them. Typically, the more effective methods are more expensive. Project managers hence skew the assessment resources, with the most effort going to methods that seem 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 are a serious problem with modern software. Leveson remarks that in modern complex systems, unsafe operations often result from an unstudied interaction between components [1]. Lutz and Mikulski [2] 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.

The ideal blind spot sampling policy is rapid and cheap to apply. It should be fully automatic and use information that is easily and quickly collected from a project. Our proposal is to control blind spot sampling using data miners to learn defect predictors from static code measures. Such measures include line counts, or the McCabe [3]/Halstead [4] measures.

The value of these static code measures remains an open issue. Much has been be written about the merits, or otherwise, of assessing code quality using static code measures. Many writers such as Fenton and Ohlsson [5] and Shepperd and Ince [6] are quite scathing in their critique of these measures, arguing that they reveal little (perhaps even nothing) about the quality of their code. Nevertheless, these measures can be automatically and cheaply collected, and hence are widely used in industry. For example, verification and validation (V&V) textbooks advise using static code complexity measures to decide which modules are worthy of manual inspections [7]. Many researchers use static measures to guide software quality predictions; e.g. [3], [4], [8]–[25].

Recently, more public domain data sets have become available from the NASA *Metrics Data Program* (or MDP[1]). Using the MDP data, a *10*10-way* evaluation of three *learners* exploring numerous *subsets* of the attributes in eight data sets, *filtered* in one of two ways (all the terms in *italics* are defined below). The clear conclusion from this experimentation, is that static code measures are indeed useful for predicting quality. Our learned predictors have a mean $pd = 71\%$ (probability of detection) and a mean $pf = 25\%$ (probability of false alarm).

An important feature of this work is that it is reproducible. Many writers have argued for repeatable software engineering results [26], [27]. However, so far, there have been all too few examples of such reproducible experiments. Accordingly, all the data and scripts and learners used in the study are available on-line so others can repeat, refute, or improve on the results shown here.

As far as we know, this is the most extensive, reproducible and (measured in terms of number of data sets), the most diverse study of learning defect predictors yet reported. Prior studies based their conclusions on fewer data sets; e.g. one data set [11], [17]; two data sets [5]). Further, our new results are much better than all our prior results [9], [18], [20], [21], [28].

These results both endorse the use of static code defect detectors for blind-spot sampling *and* explain prior pessimism about such detectors [5], [6]. Much of that prior work debated issues like "McCabes vs Halstead vs lines of code counts". We show below that the choice of learning method is far more important than which subset of the available data is used for learning. Further, it took much searching for this study to find a data mining method with a performance better than random noise. Prior work might have been more pessimistic since that prior work explored less-informative issues such as "McCabe vs lines of code" and did not explore a wide enough range of learning methods.

II. HYPOTHESES

This study comments on eleven hypotheses relating to the merits of defect predictors learned from static measures.

Numerous researchers have claimed the measures extracted from source code can be used to predict which new modules will be defective. Initially, intra-module[2] measures were proposed to predict detects. Halstead and McCabe argued that the complexity of reading a single module [4] or the pathways through a module [3] were predictors for module defects. Tacitly, Halstead and McCabe were arguing:

Dr. Menzies is with the Lane Department of Computer Science, West Virginia University and can be reached at tim@timmenzies.net

Jeremy Greenwald is with Computer Science, Portland State University and can be reached at jegreen@cecs.pdx.edu.

Art Frank is a nice guy.

This research was conducted with funds from the NASA 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.

Manuscript received January 20, 2001; revised November 18, 2002.

[1] http://mdp.ivv.nasa.gov

[2] Here, "module" is a synonym for the smallest compilable unit in a language; e.g. a "C" function or a PASCAL procedures or a JAVA method.

Hypothesis 1: Static code measures predict for defects.

Hypothesis 2: There are better intra-module measures than lines of code for predicting for defects.

Fenton and Pfleeger [26] disagree with Hypothesis 1. They offer examples where *the same* module functionality is achieved via *different* programming language constructs resulting in *different* intra-module measurements. They use this example to argue the uselessness of static code measures.

This study will explore numerous other hypotheses. For example, since Halstead published after McCabe, we can assume that Halstead was arguing:

Hypothesis 3: The Halstead measures are better than the McCabe measures building defect predictors.

Using his base measures, Halstead defined a set of *derived attributes* that, supposedly, describe the complexity of the reading source code. Halstead assumes that:

Hypothesis 4: There is added value in the derived Halstead attributes.

A common use of McCabe's is to use his preferred rule $(v(g) > 10 \vee iv(g) > 4)$ as a defect predictor. This assumes:

Hypothesis 5: Disjunctions of single attribute thresholds suffice for defect prediction.

Other researchers have challenged this assumption, preferring instead to build predictors using decision tree learners [22], [23], [25]. Each branch of a decision tree is a conjunction of tests on attribute thresholds. All the branches form one large disjunction (i.e. this branch OR that branch OR that branch OR...). Proponents of decision trees assume:

Hypothesis 6: Defect predictions are best made by disjunctions of conjunctions of attribute thresholds.

Elsewhere, we have used NäiveBayes classifiers as a method for learning defect detectors [9]. Such classifiers make no use of disjunctions or conjunctions. Rather, they compare the product of probabilities that a new example falls into one class or another. Proponents of NäiveBayes classifiers assume:

Hypothesis 7: Defect predictions are best made by products of probabilities.

Decision tree learning and NäiveBayes classifiers are two different kinds of learners. There any many more. Khoshgoftaar, for example, has explored a very wide range of learning methods such as to Poisson regression [15], to genetic programming to k-means clustering [16], as well as many others [17]. In our own previous work, for a while, we advocated the use of various learners including a minimal set contrast learners [19] and a minimalist discretization policy called ROCKY [18]. Like Khoshgoftaar, our experimentation with new learners was driven by the assumption that:

Hypothesis 8: Learning defect predictors is a hard problem requiring intricate solutions.

Elsewhere, we have conducted limited experiments suggesting that a *logarithmic filter* on all numeric values might improve predictor performance [19]. Figure 1, on the left-hand-side, shows some measures collected from the MDP *cm1* data set. These values form a exponential distribution with many small values and a few much larger values. The right-hand-side of that figure shows the same values, after being filtered through a logarithmic function. Note how the values

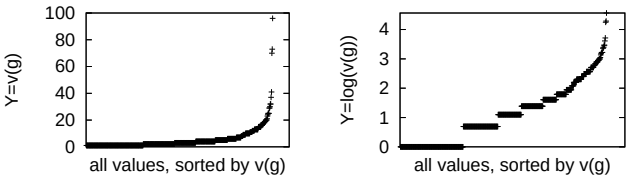

Fig. 1. A McCabe's metric from $cm1$: raw values on left, log-filtered on right.

		sub-system		
system	language	data set	# instances	% defective
spacecraft instrument	C	cm1-05	506	9
storage management for ground data	Java	kc3	459	9
		kc4	126	49
Db	C	mw1	404	7
Flight software for earth orbiting satellite	C	pc1-05	1,108	6
		pc2	5,590	0.4
		pc3	1,564	10
		pc4	1,458	12

Fig. 2. Data sets used in this study. The datasets cm1-05 and pc1-05 updates data sets $cm1$ and $pc1$ processed previously by the author; e.g. in [20].

are now more evenly spread across the y-range, making it easier to reason about them. Figure 1 suggest that:

Hypothesis 9: Log-filtering of all numerics can improve predictor performance.

All the MDP datasets contain miscellaneous attributes describing data that was easily available at the time of collection. Several of these miscellaneous variables have no proponents in the literature, yet are included under the assumption that:

Hypothesis 10: The more information, the better the learned predictors.

The MDP data sets only refer to simple intra-module measures. Much newer research has focused on *what else* can be gleamed from source code. For example, the PolySpace Verifier [10] checks for a variety of common program errors such as out of bounds array index and uninitialized variables. Other tools don't just look at the current version of the source code, but ask what is the *rate of change* in that source code. Microsoft has a static code analysis tool that predicts defects via *churn*; i.e. the percent change in the code seen by a version control system [11]. Similarly, Hall, Nikora, and Munson [12], [13] use uses churn to characterize problematic software evolution paths. Beyond mere static code measures are other automatic tools which, after decades of research, are becoming practical for industrial applications such as runtime monitoring [33], and model checking [34], and automatic theorem proving [36]. Other promising avenues include Fenton and Neil's work on Bayesian modeling where data collection and analysis is guided by an underlying causal model of factors that influence software production [37]. These tools assume:

Hypothesis 11: There are better ways than intra-module measures to find module defects.

III. Data

The above hypotheses were explored using data from the NASA Metrics Data Program (MDP). Ten such data sets are

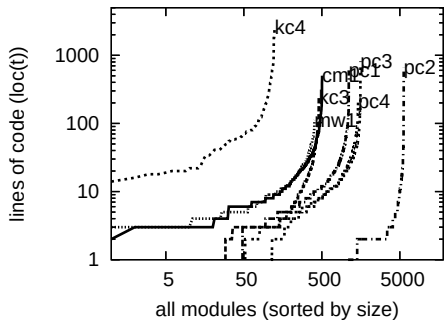

Fig. 3. Log-log plot of module sizes in the Figure 2 data.

available in that repository. Two of those data sets have a different format to the rest and were not used in this study. This left the eight, shown in Figure 2. Each instance of each data sets describes certain aspects of a single module, plus the number of defects known for that module. This data comes from eight sub-systems taken from four systems. These systems were developed in different geographical locations across North America. Within a system, the sub-systems shared some a common code base but did not pass personnel or code between sub-systems. Figure 3 shows the module sizes of our data; e.g. there are 126 modules in the $kc4$ data set, most of them are under 100 lines of code, but a few of them are over 1000 lines of code long.

Each data set was pre-processed removing the module i.d. column (which is different for each row). Also, the $error\_count$ column was converted into a boolean called *defective?* as follows: $defective? = (error\_count \geq 1)$. Finally, the $error\_density$ column was removed (since it can be derived from line counts and $error\_count$).

The pre-processed data sets had 38 attributes, plus one target class attribute attribute (*defective?*)[3]. These attributes fall into the subsets shown in Figure 4. Most of the subsets are well defined, with the exception of the *misc* group (*misc* is included under Hypothesis 10). As to the other attributes, Halstead argued that modules that are hard to read are more likely to be fault prone [4]. Halstead estimates reading complexity by counting the number of operators and operands in a module: see the h measures of Figure 4. These three raw h Halstead measures were then to compute the H: the eight derived Halstead measures using the equations shown in Figure 4. In between the raw and derived Halstead measures are certain intermediaries (which don't appear in the MDP data sets);

- $\mu = \mu_1 + \mu_2$;
- minimum operator count: $\mu_1^* = 2$;
- μ_2^* is the minimum operand count and equals the number of module parameters.

Unlike Halstead, McCabe argued that the complexity of pathways *between* module symbols are more insightful than just a count of the symbols [3]. The first three lines of Figure 4 shows McCabe three main measures for this pathway complexity. These are defined as follows. A module is said

[3]The pre-processing script is available from `http://promise.unbox.org/CategoryMDP43`.

most	m = Mccabe		$v(g)$	cyclomatic_complexity
			$iv(G)$	design_complexity
			$ev(G)$	essential_complexity
	locs	loc		loc_total (one line = one count
		loc(other)		loc_blank
				loc_code_and_comment
				loc_comments
				loc_executable
				number_of_lines (opening to closing brackets)
	Halstead	h	N_1	num_operators
			N_2	num_operands
			μ_1	num_unique_operators
			μ_2	num_unique_operands
		H	N	length: $N = N_1 + N_2$
			V	volume: $V = N * log_2 \mu$
			L	level: $L = V^*/V$ where $V^* = (2 + \mu_2{}^*) log_2 (2 + \mu_2{}^*)$
			D	difficulty: $D = 1/L$
			I	content: $I = \hat{L} * V$ where $\hat{L} = \frac{2}{\mu_1} * \frac{\mu_2}{N_2}$
			E	effort: $E = V/\hat{L}$
			B	error_est
			T	prog_time: $T = E/18$ seconds
	misc = Miscellaneous			branch_count
				call_pairs
				condition_count
				decision_count
				decision_density
				design_density
				edge_count
				global_data_complexity
				global_data_density
				maintenance_severity
				modified_condition_count
				multiple_condition_count
				node_count
				normalized_cyclomatic_complexity
				parameter_count
				pathological_complexity
				percent_comments

Fig. 4. Measures used in this study.

to have a *flowgraph*; i.e. a directed graph where each node corresponds to a program statement, and each arc indicates the flow of control from one statement to another. The *cyclomatic complexity* of a module is $v(G) = e - n + 2$ where G is a program's flowgraph, e is the number of arcs in the flowgraph, and n is the number of nodes in the flowgraph [38]. The *essential complexity*, $(ev(G))$ or a module is the extent to which a flowgraph can be "reduced" by decomposing all the subflowgraphs of G that are *D-structured primes* (also sometimes referred to as "proper one-entry one-exit subflowgraphs" [38]). $ev(G) = v(G) - m$ where m is the number of subflowgraphs of G that are D-structured primes [38]. Finally, the *design complexity* $(iv(G))$ of a module is the cyclomatic complexity of a module's reduced flowgraph.

Finally, in order to test the value of logarithmic transforms (Hypothesis 9), all the data was passed through two filters:

1) *none*; i.e. no change;
2) *logNums*: i.e. all numerics were replaced with their logarithm. To avoid numerical errors with $ln(0)$, all numbers under 0.000001 were replaced with $ln(0.000001)$.

IV. LEARNERS

The data was passed to three learners from the WEKA data mining toolkit [40]: OneR, J48, and NäiveBayes[4]. OneR was chosen to test Hypothesis 6 (defect predictions can be based on single attribute thresholds). This algorithm can only generate single attribute rules. OneR builds classification rules using one or more values from a single attribute attribute. OneR does not support even simple logical functions such as $conjunction$ [41]. Hence, if OneR performs as well as the other learners, then that would support Hypothesis 6.

Another reason for the use of OneR in this study was to test Hypothesis 8 (that learning defect predictors ins a hard task). In his text *Experimental Methods for Artificial Intelligence*, Cohen advises comparing the performance of a supposedly more sophisticated approach against a seemingly stupider "straw man" method [42, p81]. After that comparison, the merits of the more sophisticated methods would be doubted if its performance was close to that of the "straw man". Holte's OneR algorithm is the standard "straw man" data miner and has been shown to perform remarkable well compared to standard data miners on many commonly used data sets [41].

Hypothesis 6 argued that the combination of disjunctions and conjunctions in a decision tree is the best way to learn a defect predictor. To test this hypothesis, this study includes experiments with the J48 learner. J48 is a JAVA implementation of Quinlan's C4.5 (version 8) algorithm. C4.5/J48 summarizes the training data in the form of a decision tree [43]. The algorithm recursively *splits* the training data according to tests on attribute values in order to separate the classes (although attribute tests are chosen one at a time in a greedy manner, they are dependent on results of previous tests). C4.5/J48 uses information theory to assess candidate splits: the *best split* is the one that *most simplifies* the target concept. Concept simplicity is measured using information theory as follows. The class distribution C_0 contains classes $c(1), c(2), ...$ occurring with frequencies $n(1), n(2), ...$ The number of bits required to encode C_0 is $H(C_0)$ defined as follows:

$$\begin{array}{rcl} N & = & \sum_{c \in C} n(c) \\ p(c) & = & n(c)/N \\ H(C) & = & -\sum_{c \in C} p(c) log_2 p(c) \end{array} \quad (1)$$

The splits divide C_0 (before the split) into C_1 and C_2 (after the split) and the best splits lead to the simplest concepts; i.e. maximize $H(C_0) - H(C_1) + H(C_2)$.

Hypothesis 7 argued that the product of probabilities computed by a NäiveBayes classifier is the best way to learn a defect predictor. To test this hypothesis, this study includes experiments with a NäiveBayes classifier. Such classifiers employs a simplified version of Bayes formula to decide which class a test instance belongs to. The posterior probability of each class is calculated, given the feature values present in the instance; the instance is assigned to the class with the highest probability. This is a straightforward processing and involves simply estimating the probability of attribute values within each class from the training instances. Simple frequency counts are used to estimate the probability of discrete attribute values. For numeric attributes it is common practice to use the probability density function for a normal distribution [40]:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where $\{\mu, \sigma\}$ are the attribute $\{$mean,standard deviation$\}$[5].

The above learning technology can be used to generate theories from data *or* to assess the value of different portions of the data. Various *feature subset selection* (FSS) algorithms [44] find what attributes can be deleted, without damaging the performance of the learned theory. The simplest and fastest FSS method is to rank attributes from the most informative to least informative. After discretizing numeric data, then if A is a set of attributes, the number of bits required to encode a class after observing an attribute is:

$$H(C|A) = -\sum_{a \in A} p(a) \sum_{c \in C} p(c|a) log_2(p(c|a)$$

The highest ranked attribute A_i is the one with the largest *information gain*; i.e the one that most reduces the encoding required for the training data *after* using that attribute; i.e.

$$InfoGain(A_i) = H(C) - H(C|A_i) \quad (2)$$

where $H(C)$ comes from Equation 1. In *iterative InfoGain FSS*, theories are learned using the $i = 1, 2..., N$-th top-ranked attributes. FSS terminates when $i + 1$ attributes perform no better than i. In *exhaustive InfoGain FSS*, the attributes are first ranked using iterative FSS. Next, theories are built using all subsets of the top j ranked attributes. For both iterative and exhaustive FSS, the process is repeated 10 times using 90% of the training data (randomly selected). Iterative FSS takes time linear on the number of attributes N while exhaustive FSS takes time 2^j (so it is only practical for small $j \leq N$).

V. EXPERIMENTAL DESIGN

This studies used a *(M=10)*(N=10)-way cross-evaluation iterative feature subset selection* study shown in Figure 5. In such studies the training set is divided into N buckets. For each bucket in a 10-way cross-evaluation, a theory is learned on 90% of the data in the $N - 1$ buckets then tested on 10% of the data from the remaining bucket.

It is considered good practice to repeat an N-way study M times, randomizing the order each time. Many algorithms exhibit *order effects* where certain orderings dramatically improve or degrade performance (e.g. insertion sort runs slowest if the inputs are already sorted in reverse order). Randomizing the order of the inputs defends against order effects.

Such M*N-way studies orchestrate multiple experiments where a learned theory is tested against data not seen during training. In all, a 10*10-way study generates 100 training sets $t_1, t_2, .., t_{100}$ and 100 disjoint test sets $t_1, t_2, .., T_{100}$ (i.e. $t_i \cup T_i = \emptyset$). In the data mining literature, this is the preferred evaluation method when the goal is to produce theories intended to predicting future as-yet-unseen events [40].

[4]Available from `http://www.cs.waikato.ac.nz/~ml/weka/index_downloading.html`.

[5]To be precise, the probability of a continuous attribute being a particular continuous value x is zero, but the probability that it lies within a small region, say $x \pm \epsilon/2$, is $\epsilon \times f(x)$. Since ϵ is a constant that weighs across all possibilities, it cancels out and needs not be computed.

```
M         = 10
N         = 10
All       = 38  # all the attributes
DATAS     = (cm1 kc3 kc4 mw1 pc1 pc2 pc3 pc4)
FILTERS   = (none logNums)
LEARNERS  = (oneR j48 nb)

for data in DATAS
  for filter in FILTERS
      data' = filter(data)
      rank data' attributes via InfoGain # Equation 2
       for i = 1,2,3,.. ,All
          some = data's i-th highest ranked attributes
          repeat M times
              randomized order from "some"
              generate N bins from "some"
             for i in 1 to N
                tests  = bin[i]
                trainingData = some - tests
               for learner in LEARNERS
                  METHOD = (filter attributes learner)
                  theory = learner(trainingData)
                  RESULT[METHOD] = use theory on tests
```

Fig. 5. This study.

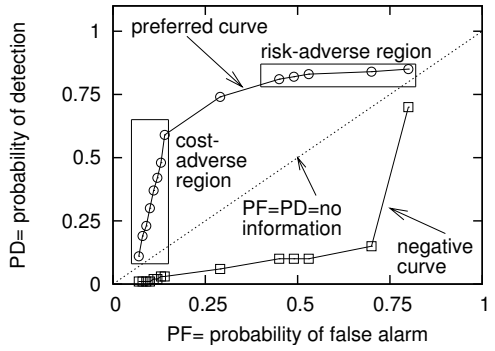

Fig. 6. Regions of a typical ROC curve.

The 10*10-way study was wrapped inside scripts that explored different subsets of the attributes in the order suggested by *InfoGain* (Equation 2). In the inner most-loop of the study, some *method* was applied to some data set. As shown in the third last line of Figure 5, these *methods* were some combination of *filter, attributes, learner*.

VI. Performance Measures

The performance of the learners on the MDP data was assessed using *ROC curves*. Formally, a defect *detector* hunts for a *signal* that a software module is defect prone. Signal detection theory [45] offers *receiver operator characteristic* (ROC) curves as an analysis method for assessing different detectors. A typical ROC curve is shown in Figure 6. The y-axis shows probability of detection (pd) and the x-axis shows probability of false alarms (pf). By definition, the ROC curve must pass through the points $pf = pd = 0$ and $pf = pd = 1$ (a detector that never triggers never makes false alarms; a detector that always triggers always generates false alarms). Three interesting trajectories connect these points:

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) 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*.
3) The point ($pf = 0$,$pf = 1$) is the ideal position (a.k.a. "sweet spot") on a ROC curve. This is where we recognize all errors and never make mistakes. *Preferred curves* bend up towards this ideal point.

		module found in defect logs?	
		no	yes
signal detected?	no (i.e. $v(g) < 10$)	A = 395	B = 67
	yes (i.e. $v(g) \geq 10$)	C = 19	D = 39

$$pd = \quad Prop.detected = \quad 37\%$$
$$pf = \quad Prob.falseAlarm = \quad 5\%$$
$$notPf = \quad 1 - pf = \quad 95\%$$
$$bal = \quad Balance = \quad 45\%$$
$$Acc = \quad accuracy = \quad 83\%$$

Fig. 7. A ROC sheet assessing the detector $v(g) \geq 10$. Each cell {A,B,C,D} shows the number of modules that fall into each cell of this ROC sheet.

In the ideal case, a detector has a high probability of detecting a genuine fault (pd) and a very low probability of false alarm (pd). This ideal case is very rare. The only way to achieve high probabilities of detection is to trigger the detector more often. This, in turn, incurs the cost of more false alarms.

Pf and pd can be calculated using the ROC sheet of Figure 7. 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 7 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).

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 this detection is the ratio of detected signals, true positives, to all signals:

$$\text{probability detection} = pd = recall = D/(B + D) \quad (3)$$

(Note that pd is also called $recall$.) In another case, the probability of a false alarm is the ratio of detections when no signal was present to all non-signals:

$$\text{probability false alarm} = pf = C/(A + C) \quad (4)$$

For convenience, we say that $notPf$ is the complement of pf:

$$\text{notPf} = 1 - C/(A + C) \quad (5)$$

Figure 7 also lets us define the *accuracy*, or acc, of a detector as the percentage of true negatives and true positives:

$$\text{accuracy} = acc = (A + D)/(A + B + C + D) \quad (6)$$

If reported as percentages, these measures have the range:

$$0 \leq \ acc\%, \ pd\%, \ , notPf\% \leq 100$$

Ideally, we seek detectors that maximize *acc%, pd%, notPf%*.

Note that maximizing any one of these does not imply high values for the others. For example Figure 7 shows an example with a high accuracy (83%) but a low probability of detection (37%). Accuracy is a good measure of a learner's performance when the classes occur with similar frequencies. The data sets used in this study, however, have very uneven class distributions (see Figure 2). Therefore this paper will assess its learned theories using $bal, pd, notPf$ and not acc.

In practice, engineers *balance* between pf and pd. To operationalize this notion of *balance*, we define bal to be the Euclidean distance from the sweet spot $pf = 0, pd = 1$ to a pair of $< pf, pd >$. For convenience, we (a) normalize bal by the maximum possible distance across the ROC square ($\sqrt{2}$); (b) subtract this from 1; and (c) express it as a percentage; i.e.

$$balance = bal = 1 - \frac{\sqrt{\left(\sqrt{0 - pf}\right)^2 + \left(\sqrt{1 - pd}\right)^2}}{\sqrt{2}} \quad (7)$$

Hence, better and *higher* balances fall *closer* to the desired sweet spot of $pf = 0, pd = 1$.

VII. Quartile Charts of Performance Deltas

Recall from Figure 5 that a *method* is some combination of $filter, attributes, learner$. This experiment generated nearly 800,000 *performance deltas* comparing $pd, notPf, bal$ values from different *methods* applied to the same $test$ data.

The performance deltas were computed using simple subtraction, defined as follows. A *positive performance delta* for method X means that method X has out-performed some other method in *one* comparison. Using performance deltas, we say that the best method is the one that generates the largest performance deltas over *all* comparisons.

The performance deltas for each method were sorted and displayed as *quartile charts*. To generate these charts, the performance deltas for some $method$ were sorted to find the lowest and highest quartile as well as the median value; e.g.

$$\overbrace{\underbrace{-59}_{min}, -19, -19}^{lowest\ quartile}, -16, -14, -10, \underbrace{-10}_{median}, 5, 14, 39, \overbrace{42, 62, \underbrace{69}_{max}}^{highest\ quartile}$$

In a quartile chart, the upper and lower quartiles are marked with black lines; the median is marked with a black dot; and vertical bars are added to mark (i) the zero point and (ii) the minimum possible value and (iii) the maximum possible value (in our case, -100% and 100%). The above numbers would therefore be drawn as follows:

$$-100\% \mid \quad — \quad \bullet \mid \quad — \mid 100\%$$

We prefer quartile charts of performance deltas to other summarization methods for M*N studies. Firstly, they offer a very succinct summary of a large number of experiments. For example, Figure 8 display 200,000 performance deltas in $\frac{1}{4}$ of a page. Secondly, they are a *non-parametric* display; i.e. they make no assumptions about the underling distribution. Standard practice in data mining is to compare the mean performance of different methods using t-test [40]. T-tests are a *parametric method* that assume that the underling population

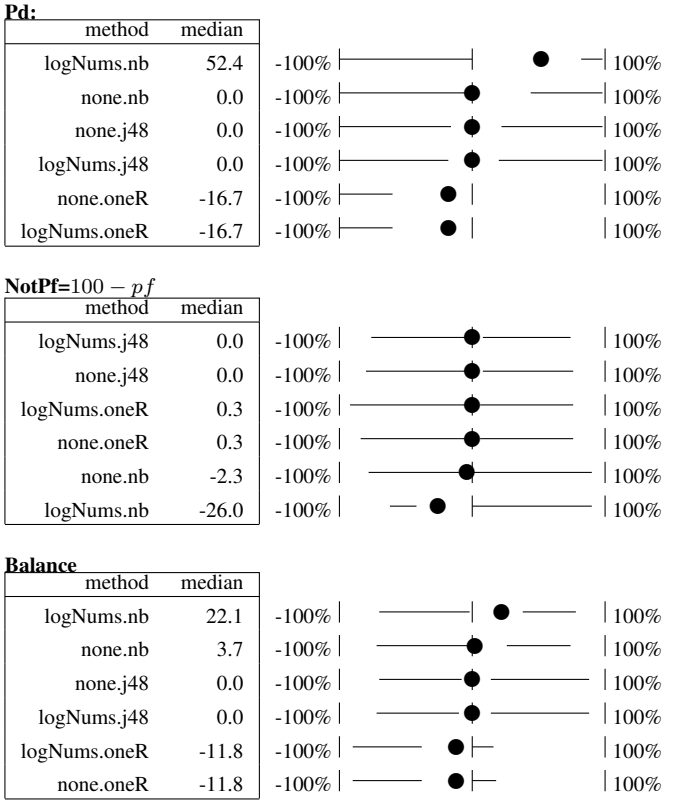

Fig. 8. Performance deltas for $pd, notPf, bal$ using all 38 attributes.

.

distribution is a Gaussian. Recent results suggest that there are many statistical issues left to explore regarding how to best to apply those t-tests for summarizing M*N-way studies [46]. Such t-tests assume Gaussian distributions and some of our results are clearly non-Gaussian:

- The NäiveBayes performance delta pd results (using $logNums$) of Figure 8 exhibits an extreme skewness (a median point at 52.4 with a quarter of the performance deltas pushed up towards the maximum figure of 100%).
- All the OneR performance delta pd results of Figure 8 are highly skewed. OneR's pd performance delta was *never* higher than 16.7 and over half the performance deltas for that method had that value (hence, the missing upper arms in the OneR results of Figure 8).

For the sake of completeness, we applied t-tests when sorting quartile charts: one quartile chart appears above its neighbor if it was statistically different (at the 95% confidence level) and has a larger mean. However, given the skews we are seeing in the data, we base our conclusions on *stand-out* effects seen in the non-parametric quartile diagrams. A *stand-out* effect is a large and positive median with a highest quartile bunched up towards the maximum figure. The pd results for NäiveBayes (with $logNums$) are an example of such a stand-out effect.. On the other hand, OneR's $notPf$ results are a *negative stand-out*: those performance deltas tend to bunch down towards -100%; i.e. on the $notPf$ measure, OneR usually performs much worse than anything else.

Balance:

#attributes	median	
3	10.0	-100% | ——●—— | 100%
38	8.0	-100% | ——●—— | 100%
2	0.0	-100% | ——●—— | 100%
1	-30.7	-100% | —●— | 100%

Fig. 9. On balance performance deltas of NaiveBayes (with logNums) using just the best 1,2,3 attributes, or all 38 attributes.

VIII. Results

NäiveBayes with a log-transform has both a positive stand-out result for pd and a negative stand-out result for $notPf$. This result, of winning on pd but losing on pf, is to be expected. Figure 6 showed that the cost of high pds are higher pfs. The other learning methods cannot emulate the high pds of NäiveBayes (with log-transforms) since they take less chances (hence, have lower false alarm rates).

The *balance* results of Figure 8 combines the pd and pf results, using Equation 7. On balance, with 38 attributes:

- OneR loses more often than it wins: observe that OneR has a negative median $balance$.
- The best method, on balance, is clearly NäiveBayes with log-transforms since it has a minority of negative balance performance deltas (only 25%); and it beats other methods by 22.1% (or more) half the time.

A review of the J48 and OneR quartile charts in the Figure 8 shows that J48 out-performs OneR in terms of pd and $notPf$ and bal. That is, for these data sets, theories comprising elaborate sets of conjunctions and disjunctions (i.e. J48's decision trees) perform better than than theories comprising disjunctions of single attributes (i.e. OneR's simple rules).

Since, on balance *bayes.log* performs best, the rest of this article only presents the FSS results for that method. Initial experiments with iterative InfoGain FSS showed that all but one of the data sets (PC2) could be reduced from 38 to three attributes without degrading the on-balance performance. However, iterative FSS selected seven attributes for PC2. Therefore, for that data set only, exhaustive FSS was performed on 2^7 subsets. This procedure yielded two attributes that worked as well as all 38.

These InfoGain results were then compared to various other FSS methods: CFS [47]; Relief [48], [49]; and CBS [50]. Measured in terms of pd or $nofPf$ or $balance$, or number of selected attributes, there was no apparent advantage is using these other FSS methods instead of InfoGain

Figure 9 shows the InfoGain results for NäiveBayes with logNums. On balance, using two or three attributes worked as well as using 38 attributes. However, using only one attribute resulted in inferior performance.

In retrospect, this large reduction in the attribute space (from 38 attributes to just to two or three) should have been expected. Figure 4 shows that many of the static code attributes are correlated; e.g. the derived Halstead measures are computed from the raw Halstead measures. It is hardly surprising that such a highly correlated set of attributes can be reduced to a smaller set without lose of overall performance.

Figure 10 shows more details of the performance of

		%			selected attributes	selection
data	N	pd	pf	acc	(seeFigure 11)	method
cm1	100	71	27	73	5, 35, 36	iterative FSS
kc3	100	69	28	72	16, 24, 26	iterative FSS
kc4	100	79	32	73	3, 13, 31	iterative FSS
mw1	100	52	15	82	23, 31, 35	iterative FSS
pc1	100	48	17	81	3, 35, 37	iterative FSS
pc2	100	72	14	86	5, 39	exhaustive FSS
pc3	100	80	35	67	1, 20, 37	iterative FSS
pc4	100	98	29	74	1, 4, 39	iterative FSS
all	800	71	25	76		

Fig. 10. Best defect predictors learned in this study. Mean results from a NäiveBayes classifier after a 90%:10% split into training:test. Prior to learning, all numerics where replaced with logarithms. InfoGain was then used to select the best two or three attributes shown in the right-hand column (and if "three" performed as well as "two", then this table shows the results using "two").

ID	frequency in Figure 10	what	type
1	2	loc_blanks	locs
3	2	call_pairs	misc
4	1	loc_code_and_command	locs
5	2	loc_comments	locs
13	1	edge_count	misc
16	1	loc_executable	locs
20	1	I	H (derived Halstead)
23	1	B	H (derived Halstead)
24	1	L	H (derived Halstead)
26	1	T	H (derived Halstead)
31	2	node_count	misc
35	3	μ_2	h (raw Halstead)
36	1	μ_1	h (raw Halstead)
37	2	number_of_lines	locs
39	2	percent_comments	misc

Fig. 11. Attributes used in Figure 10, sorted into the groups of Figure 4.

NäiveBayes (with logNums) on the different data sets using the reduced attribute sets. On average, the methods proposed here achieve mean $accuracy = 76\%$, mean $pd = 71\%$, and mean $pf = 25\%$. The Figure 10 results are actually better than they first appear:

- Recall from Figure 2 that the number of defective modules may be very small: the most extreme example of this is PC2 with only 0.4% defective modules. It is somewhat of an achievement that, for PC2, our methods yielded $\{pd = 72\%, pf = 14\%\}$ for such a tiny target.
- The best we have achieved in the past with cross-validation was a mean $pd \approx 50\%$ and mean $pf \approx 30..40\%$ [18], [21]. The results of Figure 9 results have higher pds and lower pfs.

One interesting aspect of Figure 10 is that different data sets selected very different "best" attributes (see the *selected attribute* column). This aspect can be explained by Figure 12 which shows the InfoGain of all the attributes in an MDP data set. As might be expected in datasets with many correlated attributes, the highest ranked attributes (those on the left-hand-side) offer very similar information. That is, there are no clear winners so minor changes in the training sample (e.g. the 90% sub-sampling used in FSS or a cross-validation study) can result in the selection of very different "best" attributes.

The pattern of InfoGain values of Figure 12 (where there are many alternative"best" attributes) repeats in all the MDP data sets. This pattern explains an early observation of Shepperd

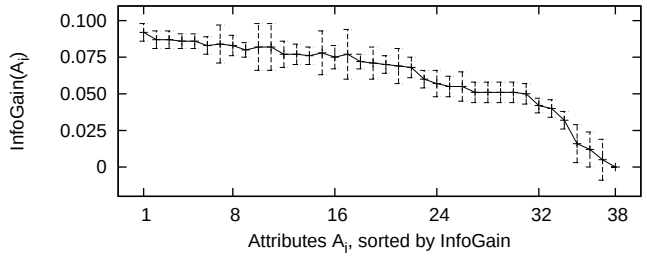

Fig. 12. InfoGain for KC3 attributes. Calculated from Equation 2. Lines show means and t-bars show standard deviations after 10 trials on 90% of the training data (randomly selected).

and Ince who found 18 publications where an equal number of studies report that the McCabe cyclomatic complexity is the same; is better; or is worse than lines of code in predicting defects [6]. Such conclusion instabilities is to expected in data sets with numerous alternative "best" attributes. Figure 12 motivates the following conclusion: don't seek "best" subsets of static code measures. Rather, seek instead for learning methods that can combine multiple partial defect indicators (e.g. the statistical methods of NäiveBayes).

IX. Review of Hypotheses

Within the context of the above results, we can now comment on our hypotheses. In the following, ✓ and ✗ denote supported and unsupported hypotheses (respectively).

Hypothesis 1 (✓): *static code measures predict for defects*. This study used static code measures to generate predictors with average results of $accuracy = 76\%$, $pd = 71\%$ and $pf = 25\%$ (see Figure 10). Such detectors are useful, at least for blind-spot sampling.

Hypothesis 2 (✓): *There are better measures than lines of code*. Clearly, lines of codes by themselves are often inadequate. Most of the best attribute subsets shown in Figure 10 augmented the $locs$ measures with non-loc measures.

Hypothesis 3 (✓): *Halstead measures are better than McCabe measures*. In this sample here, none of the best attributes found in the best feature subsets of Figure 12 contained the classic McCabe measures: $v(g), iv(g), ev(g)$. However, this conclusion should be treated with some caution (see the remarks below relating to Hypothesis 10).

Hypothesis 4 (✓): *There is value added in the derived Halstead measures*. This hypothesis is supported since Figure 11 includes both raw and derived Halstead measures.

Hypothesis 5 (✗): *Single attribute disjunctions are enough.* If single attribute disjunctions such as $v(g) > 10 \vee iv(g) > 4$ where the best defect detectors, then two results would be predicted. Firstly, the single attribute tests of OneR would perform as well as the multiple tests of J48. Secondly, the FSS methods would select attribute sets of size one. Neither of these results were seen in Figure 8 and Figure 9.

Hypothesis 6 (✗): *Disjunctions+conjunctions are enough.*

Hypothesis 7 (✓): *Products of probabilities are enough.* NäiveBayes with logNums' superior performance over J48 in Figure 8 endorses Hypothesis 7, but not Hypothesis 6. We offer two explanations why NäiveBayes with logNums outperforms our prior work:

- Recalling Figure 1, it is possible that code defects are actually associated in some log-normal way to static code measures. Of all the methods studied here, only NäiveBayes with $logNums$ was able to directly exploit this association.
- Recalling Figure 12, many of the static code measures have similar information content. Perhaps defect detection is best implemented as some kind of thresholding systems; i.e. by summing the signal from several partial indicators. Of all the learners used in this study, only the statistical approach of NäiveBayes can sum information from multiple attributes.

Hypothesis 8 (✗): *Learning defect predictors requires intricate solutions.* Most of our results come from very simple data mining technology: iterative InfoGain FSS over a basic NäiveBayes classifier that assumed a single normal distribution for its internal numerics. The NäiveBayes classifier used in this study was capable of more elaborate kernel estimation, but we disabled that feature. In the space of possible data mining algorithms, iterative FSS over NäiveBayes is a very simple method indeed. Hence, for the most part, we can't support Hypothesis 8. The one exception to this conclusion comes from the PC2 results. Recall that PC2 had the lowest defect rate (0.4%) and exhaustive FSS was required to find the select the best attributes for this data set. It may be that for the special case of very small target concepts, more intricate solutions are required.

Hypothesis 9 (✓): *Log-filtering of all numerics improves predictor performance.* In Figure 8, log-filtering is clearly associated with the stand-out results. With regard to preprocessing the numerics, we might be able to do even better than our current results. Dougherty et.al. [51] report spectacular improvements in the performance of NäiveBayes classifiers via the use of better numeric pre-processing that just simple log-filtering. In the near future, we will try their methods on this domain.

Hypothesis 10 (✓): *The more information, the better the learned predictors.* Figure 10 shows the best attribute subsets for different data sets can be different indeed. In the eight data sets explored here, no particular attribute appeared in the best attribute subsets more than three times. These best attributes came from all measurement types and not just from (e.g.) the Halstead set.

Clearly, the best attributes to use for defect detection vary from data set to data set. Hence, rather than advocating a particular subset of possible measures as being the *best measures*, these experiments suggest that defect detectors should be built using *all available* attributes, followed by FSS to find the best subset for a particular domain.

Hypothesis 11 (✓): *There are better ways than intra-module measures to find module defects.* The case was made above that our current results are surprisingly good, particularly since some of our datasets a very small number of defective modules. Nevertheless, there is much room for improvements. For example, higher pds and lower pfs would be preferred.

X. Future Work

There is much current work on building better static code measures (e.g. [10]–[13], [33], [34], [36], [37]) that could yield better pd/pf results. To date, the results with these alternate methods are promising, but hardly conclusive. For example, Nagappan and Ball [11] report accuracies of 88.8% using measures based on *churn* (rate of change in code base) but that result is not directly comparable with the Figure 10 results. Unlike our 10*10-way study, their study did not separate a test set from a train set. That is, our results come from estimates of errors on novel test cases while their results were *self-tests*; i.e. came from testing their theory on their training data. Self-tests over-estimate performance on data not used in training [40]. That is, our Figure 10 results might indeed be as good as churn-based results.

In order to truly compare methods, comparative studies on the same data must be performed. Such comparative studies discussing are only just appearing (e.g. [52]) and it is too soon to generalize their conclusions. Perhaps, in the future, there will exist data sets with attributes created from intra-module measurements as well as measures from a growing number of new tools At that time, the experimental procedures defined in this paper would be useful for assessing the relative merits of these different tools.

XI. Discussion

Our proposal is to use static code measures to control blind spot sampling. An alternative to this proposal is to remove the blind spots all together 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)[6].
- *Automatic formal methods:* The state space explosion problem imposes strict limits on how much a system can be explored via automatic formal methods [30].
- *Other methods:* Lowry et.al. [31] and Menzies and Cukic [32] offer numerous other examples where assessment effectiveness is exponential on effort. For example, the more complicated the tool, the better the user needs to be. This is particularly true for tools that require temporal logic constraints such as runtime monitoring [33] and model checking [34]. However, it is also true for the more complex static analysis tools such as Polyspace [35].

Exponential costs quickly exhaust finite resources. Hence, the blind spots can't be removed, and must be managed.

[6]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, $N(0.90, 10^{-3}) = 2301$ [29].

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 sampling policy* to prioritize the exploration the rest of the system. Here, we have argued for a sampling policy based static code measures assessed via InfoGain FSS and summarized via NäiveBayes with logNums. 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. Further, such a policy results in detectors which can trigger on possibly error-prone modules. For the sample explored here, those triggers had non-trivial levels of performance ($mean\ pd = 71\%, mean\ pf = 25\%$).

Since we are optimistic about using static code measures, we need to explain prior pessimism about such measures (e.g. [5], [6]):

- Prior work would not have found good detectors if that work had focused on attribute subsets, rather than the learning methods. Figure 10 showed that the best attribute subsets for defects detectors can change dramatically from data set to data set. Hence, conclusions regarding the *best dataset* are very brittle; i.e. may not still apply when we change data sets.
- Prior work would not have found good detectors if that work had not explored a large space of learning methods. It took much searching for this study to find a data mining method with a performance better than random noise. Figure 8 shows that, of the six methods explored here, only *one* (NäiveBayes with logNums) had a median performance that was both large and positive.

In summary we endorse the use of static code measures from predicting detects with the following caveat. Those predictors should be treated as probabilistic, not categorical, indicators. While our best methods have a non-zero false alarm, they also have a usefully high probability of detection (over $\frac{2}{3}$rds). Just as long as users treat these predictors as *indicators* and not definite *oracles*, then the predictors learned here would be pragmatically useful for (e.g.) focusing limited V&V budgets on portions of the code base that are predicted to be problematic.

XII. External Validity

Like any empirical data mining paper, our conclusions are biased according to what data was used to generate them. Issues of *sampling bias* threaten any data mining experiment; i.e. what matter *there* may not be true *here*. For example, the sample used here comes from NASA and NASA works in a particularly unique market niche.

Nevertheless, we argue that results from NASA are relevant to the general software engineering industry. NASA makes extensive use of contractors who are contractually obliged (ISO-9O01) to demonstrate their understanding and usage

of current industrial best practices. These contractors service many other industries; e.g. Rockwell-Collins builds systems for many government and commercial organizations. For these reasons, other noted researchers such as Basili, Zelbowitz, et.al. [39] have argued that conclusions from NASA data are relevant to the general software engineering industry.

All inductive generalization suffers from a sampling bias. The best we can do is define our methods and publicize our data such that other researchers can try to repeat our results and, perhaps, point out a previously unrealized bias in our analysis. Hopefully, other researchers will emulate our methods in order to repeat or refute or improve our results. We would encourage such researchers to offer not just their conclusions, but the data used to generate those conclusions. The MDP is a repository for NASA data sets and the PROMISE repository[7] is a place to store and discuss software engineering data sets from other organizations.

REFERENCES

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

[2] 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.

[3] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, Dec. 1976.

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

[5] N. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software Engineering*, pp. 797–814, August 2000.

[6] M. Shepped and D. Ince, "A critique of three metrics," *The Journal of Systems and Software*, vol. 26, no. 3, pp. 197–210, September 1994.

[7] S. Rakitin, *Software Verification and Validation for Practitioners and Managers, Second Edition*. Artech House, 2001.

[8] M. Chapman and D. Solomon, "The relationship of cyclomatic complexity, essential complexity and error rates," 2002, proceedings of the NASA Software Assurance Symposium, Coolfont Resort and Conference Center in Berkley Springs, West Virginia. Available from http://www.ivv.nasa.gov/business/research/osmasas/conclusion2002/Mike_C%hapman_The_Relationship_of_Cyclomatic_Complexity_Essential_Complexity_and_Erro%r_Rates.ppt.

[9] T. Menzies, J. DiStefano, A. Orrego, and R. Chapman, "Assessing predictors of software defects," in *Proceedings, workshop on Predictive Software Models, Chicago*, 2004.

[10] "Polyspace verifier®," 2005. [Online]. Available: \url{http://www.di.ens.fr/~cousot/projects/DAEDALUS/synthetic_summary/POLYSPACE/polyspace-daedalus.htm}

[11] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *ICSE 2005, St. Louis*, 2005.

[12] G. Hall and J. Munson, "Software evolution: code delta and code churn," *Journal of Systems and Software*, pp. 111 – 118, 2000.

[13] A. Nikora and J. Munson, "Developing fault predictors for evolving software systems," in *Ninth International Software Metrics Symposium (METRICS'03)*, 2003.

[14] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *ICSE*, 2005, pp. 580–586. [Online]. Available: http://doi.acm.org/10.1145/1062558

[15] T. Khoshgoftaar, "An application of zero-inflated poisson regression for software fault prediction," in *Proceedings of the 12th International Symposium on Software Reliability Engineering, Hong Kong*, Nov 2001, pp. 66–73.

[16] W. Tang and T. M. Khoshgoftaar, "Noise identification with the k-means algorithm," in *ICTAI*, 2004, pp. 373–378. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ICTAI.2004.93

[17] T. M. Khoshgoftaar and N. Seliya, "Fault prediction modeling for software quality estimation: Comparing commonly used techniques," *Empirical Software Engineering*, vol. 8, no. 3, pp. 255–283, 2003. [Online]. Available: http://dx.doi.org/10.1023/A:1024424811345

[7] http://promise.unbox.org

[18] 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.

[19] T. Menzies, J. S. DiStefeno, M. Chapman, and K. Mcgill, "Metrics that matter," in *27th NASA SEL workshop on Software Engineering*, 2002, available from http://menzies.us/pdf/02metrics.pdf.

[20] T. Menzies, J. D. Stefano, and M. Chapman, "Learning early lifecycle IV&V quality indicators," in *IEEE Metrics '03*, 2003, available from http://menzies.us/pdf/03early.pdf.

[21] 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.

[22] A. Porter and R. Selby, "Empirically guided software development using metric-based classification trees," *IEEE Software*, pp. 46–54, March 1990.

[23] J. Tian and M. Zelkowitz, "Complexity measure evaluation and selection," *IEEE Transaction on Software Engineering*, vol. 21, no. 8, pp. 641–649, Aug. 1995.

[24] T. Khoshgoftaar and E. Allen, "Model software quality with classification trees," in *Recent Advances in Reliability and Quality Engineering*, H. Pham, Ed. World Scientific, 2001, pp. 247–270.

[25] K. Srinivasan and D. Fisher, "Machine learning approaches to estimating software development effort," *IEEE Trans. Soft. Eng.*, pp. 126–137, February 1995.

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

[27] T. Menzies and P. Haynes, "The Methodologies of Methodologies; or, Evaluating Current Methodologies: Why and How," in *Tools Pacific '94*. Prentice-Hall, 1994, pp. 83–92, available from http://menzies.us/pdf/tools94.pdf.

[28] T. Menzies, "21^{st} century ai: proud, not smug," *IEEE Intelligent Systems*, 2003, available from http://menzies.us/pdf/03aipride.pdf.

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

[30] 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.

[31] 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*, 1998, pp. 322–331.

[32] T. Menzies and B. Cukic, "How many tests are enough?" in *Handbook of Software Engineering and Knowledge Engineering, Volume II*, S. Chang, Ed., 2002, available from http://menzies.us/pdf/00ntests.pdf.

[33] K. Havelund, "Using runtime analysis to guide model checking of java programs," in *SPIN Model Checking and Software Verification*. Springer-Verlag, 2000, pp. 245–264, available from http://citeseer.nj.nec.com/havelund00using.html.

[34] E. Clarke, Orna Grumberg, and Doron A. Peled, *Model Checking*. Cambridge, MA: MIT Press, 1999.

[35] W. Visser, "Personel communication: Comments on different tools," 2005.

[36] S. Owre, J.M. Rushby, N. Shankar, and M.K. Srivas, "A tutorial on using PVS for hardware verification," in *Proc. 2nd International Conference on Theorem Provers in Circuit Design (TPCD94)*, T. Kropf and R. Kumar, Eds., vol. 901. Bad Herrenalb, Germany: Springer-Verlag, 1994, pp. 258–279. [Online]. Available: citeseer.nj.nec.com/owre95tutorial.html

[37] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 675–689, 1999, available from http://citeseer.nj.nec.com/fenton99critique.html.

[38] N. E. Fenton and S. Pfleeger, *Software Metrics: A Rigorous & Practical Approach (second edition)*. International Thompson Press, 1995.

[39] V. Basili, F. McGarry, R. Pajerski, and M. Zelkowitz, "Lessons learned from 25 years of process improvement: The rise and fall of the nasa software engineering laboratory," in *Proceedings of the 24th International Conference on Software Engineering (ICSE) 2002, Orlando, Florida*, 2002, available from http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/83.88.pdf.

[40] I. H. Witten and E. Frank, *Data mining. 2nd edition*. Los Altos, US: Morgan Kaufmann, 2005.

[41] R. Holte, "Very simple classification rules perform well on most commonly used datasets," *Machine Learning*, vol. 11, p. 63, 1993.

[42] P. Cohen, *Empirical Methods for Artificial Intelligence*. MIT Press, 1995.

[43] R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufman, 1992, iSBN: 1558602380.
[44] M. Hall and G. Holmes, "Benchmarking attribute selection techniques for discrete class data mining," *IEEE Transactions On Knowledge And Data Engineering*, vol. 15, no. 6, pp. 1437– 1447, 2003.
[45] D. Heeger, "Signal detection theory," 1998, available from http://white.stanford.edu/~heeger/sdt/sdt.html.
[46] R. Bouckaert, "Choosing between two learning algorithms based on calibrated tests," in *ICML'03*, 2003, available from http://www.cs.pdx.edu/~timm/dm/10x10way.
[47] M. A. Hall., "Correlation-based feature selection for machine learning," Ph.D. dissertation, Department of Computer Science, University of Waikato, Hamilton, New Zealand, 1998.
[48] K. Kira and L. Rendell, "A practical approach to feature selection," in *The Ninth International Conference on Machine Learning*. Morgan Kaufmann, 1992, pp. pp. 249–256.
[49] I. Kononenko, "Estimating attributes: Analysis and extensions of relief," in *The Seventh European Conference on Machine Learning*. Springer-Verlag, 1994, pp. pp. 171–182.
[50] H. Almuallim and T. Dietterich, "Learning with many irrelevant features," in *The Ninth National Conference on Artificial Intelligence*. AAAI Press, 1991, pp. pp. 547–552.
[51] J. Dougherty, R. Kohavi, and M. Sahami, "Supervised and unsupervised discretization of continuous features," in *International Conference on Machine Learning*, 1995, pp. 194–202.
[52] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, R. Washington, and W. Visser, "Experimental evaluation of verification and validation tools on martian rover software," *Formal Methods in Systems Design Journal*, September 2005.

Tim Menzies Tim Menzies is an associate professor at the University of West Virginia and works with NASA on software quality. His recent research concerns modeling and learning, with a particular focus on lightweight modeling methods. He received his PhD in AI and knowledge engineering from the University of New South Wales. Contact him at tim@menzies.us.

Art Frank arf@cs.pdx.edu

Jeremy Greenwald Jeremy Greenwald is a student at Portland State University. He received his BS in Physics and Astronomy from the University of Pittsburgh in 2001. He is currently working towards a Master's degree in Computer Science with a planned graduation date of June, 2006. He works on applying data mining techniques to the field of software engineering. He is also interning at a software company in Beaverton, Oregon