Why Mine Software Repositories?

Tim Menzies *IEEE Member*, David Raffo *IEEE Member*, Siri-on Setamanit, Justin DiStefano, Robert M. Chapman

***Abstract*— Data mining results about fault detectors are typically assessed in terms of their predictive accuracy (PDs). While interesting, such results may not convince a project manager that they should reallocate their scarce resources to implementing a new technology on their project.**

This article proposes a methodology for assessing the merits of defect detectors learnt from software repositories of static code measures (Halstead and McCabe). Using process simulation, we find situations where the use of such detectors is useful or useless.

I. Introduction

Good research results are wasted unless there is a compelling business case to use them. Without such a case, a project manager may not be convinced that they should, for example, reallocate scarce resources to implement a new technology on their project. The aim of this article is to offer one example of how to combine research results (on learning defect detectors) with a business case (on why using those detectors is a worthy idea).

In a special issue on mining software repositories, it might be suspected that our conclusions will be biased; i.e. that we would *always* find a good business case for data mining databases of software. We therefore take care to present business cases that endorse mining repositories as well as others that identify situations where the mining of repositories for defect detectors is *not* useful. These business cases include an IV&V case study (where test engineers study someone else's code) and a V&V study (where test engineers study their own code). In summary, in these first two case studies, using repositories to find defect detectors *helps* IV&V but *hurts* V&V

A third scenario is then run that demonstrates the utility of our methodology. This third scenario finds a change to the V&V scenario that reverses the previous negative result. That is, by using the process simulation we identify situations in which application of this new technology would be beneficial. We also find situations in which applying this technology would *not* be beneficial. Moreover, we can set performance benchmarks for vendors of this type of technology, diagnose problems associated with implementation and assess alternative approaches for applying the technology to the benefit of the organization

Tim Menzies is with the Department of Computer Science, Portland State University, P.O. Box 751 Portland, Oregon 97207-0751 tim@menzies.us. Dr. Menzies' web site is http://menzies.us

David Raffo and Siri-on Setamanit are with the School of Business Administration, Portland State University, Portland, Oregon: raffod@pdx.edu; setamanit@comcast.net.

Justin DiStefano and Robert Chapman are with Galaxy Global Corporation, Fairmont West Virginia, justin@lostportal.net and {Robert.M.Chapman@ivv.nasa.gov}.

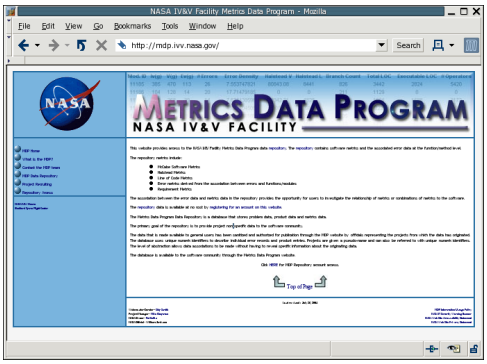

Fig. 1. The MDP data repository.http://mdp.ivv.nasa.gov.

Our study is in three parts. First, we present a related work section That offers some background notes on the technologies used in this paper.

Second, we report stable properties of defect detectors learned from software repositories. This second part will base its conclusions from five public domain data sets from the NASA Metrics Data Program (MDP) (see Figure 1). The MDP dataset is *sanitized* data and so an anonymous name is given to each data set; e.g. the CM1,JM1,KC1,KC2,PC1 data sets shown in Figure 2. This data contains static code measures (Halstead, McCabe, lines of code) and defect rates (which we convert to booleans: $defects \in \{true, false\}$). To be precise, we will actually study 12 MDP datasets but Figure 3 shows that seven of those are sub-divisions of the KC1 dataset. While a larger sample than five (or 12) data sets would be preferred, our corpus is much larger than those seen in many other high-profile studies[1].

The third and final part of our study will use the properties identified in the second part to reconfigure a *process simulation* of the IEEE 12207 software development life cycle. IEEE 12207 is used at many sites, including NASA and the Department of Defense (DoD). The process simulation model will then be used to determine the costs and benefits of using defect detectors mined from software repositories.

II. Related Work

In this paper we propose a process change for software engineers. In short, we would augment their current activity with defect detectors built by *data miners* from static code measures. This new process is assessed via *process simulation*. This section offers some background on process simulation, data mining and the merits of static code measures.

[1]e.g. A recent report on software defect detectors in *IEEE Transactions on Software Engineering* used just 2 data sets [1].

project	file	level	# modules	% with defects	developed language	at	notes
PC1	pc1.arff	1	1107	6.8	C	location 1	Flight software for earth orbiting satellite
JM1	jm1.arff	1	10885	19%	C	location 2	Real-time predictive ground system: Uses simulations to generate predictions
CM1	cm1.arff	1	496	9.7%	C	location 3	A NASA spacecraft instrument
KC1	kc1.arff	3	2107	15.4%	C++	location 4	Storage management for receiving and processing ground data
	kc1_1|2|3|13|16|17|18.arff	4	80..290		C++	location 4	7 divisions of KC1
KC2	kc2.arff	4	523	20%	C++	location 4	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.
Total	15118						

Fig. 2. Data sets used in this study. $File$ denotes a data set file name available in *ARFF* format from `http://scant.org/2/eg/arff`.

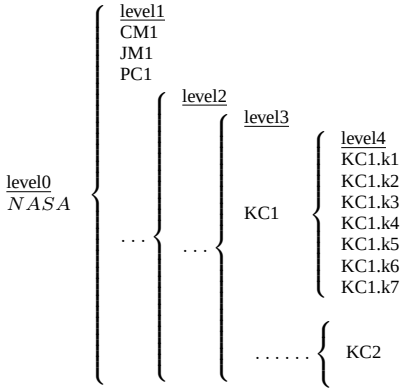

Fig. 3. Data sets used in this study.

A. Process Simulation

In order to make a business case for these learned defect detectors, we need to assess their impact on the performance of the project. There are many approaches that can be used to assess the business implications of process changes [2]–[9]. One key question with all the methods is "Where do we get the numbers to assess the impact of the process change for the business case analysis?" Obtaining the numbers for the business case analysis can be non-trivial. Data must be collected from the project or provided using a collection of industry benchmarks or expert opinion obtained through surveys or interviews. Models must then be developed to assess the impact of the process changes usually along multiple dimensions of performance (such as cost, quality and schedule). These models must be developed to predict process level performance and the results must also be provided at the project level. Most cost estimating tools including COCOMO and SLIM do not explicitly include the details of the development process. They only provide project-level outputs and are therefore not well suited for this purpose.

Process simulation is commonly used in many industries including manufacturing and service operations to address these kinds of issues. In recent years, process simulation has been applied to software development processes[2]. The key advantage to process simulation is that these models can capture the details associated with the development process and provide a systematic approach for incorporating metrics data and creating the necessary process level predictions along multiple measures of performance [6], [10], [11]. Simulation modeling tools (e.g. Arena, Extend, Stella, etc.) simplify conducting sensitivity (or "what if") analyses. As a result, process simulation models can explicitly capture localized changes to the process made by implementing a new tool, technology or method and then predict the overall project-level impacts.

In this paper, we employ process simulation to assess the impact of applying learned fault detectors under three possible operational scenarios. The specific process simulation model that will be used for this study is a model of the IEEE 12207 systems development process [12]. This process is representative of the process used on large-scale NASA and US Department of Defense (DoD) projects. The model contains industry standard benchmark data from [13] for large-scale systems development. Moreover, the model has been tuned using a data set from 8 NASA projects over 100 KSLOC in size. Predictions made with the model provide similar accuracy to those obtained using COCOMO I (i.e. predictions were within 30% of actual values, more than 70% of the time).

Figure 4 shows a top-level view of the software development model used for this study. As can be seen in that figure, the main life cycle phases of the IEEE 12207 process are:

- Process implementation
- System and software requirements analysis
- Software architecture and, detailed design
- Software coding and unit testing
- Software and system integration planning
- Integration and qualification testing
- Integration and acceptance support

Figure 4 shows that we have augmented IEEE 12207 with an additional IV&V layer that models the actions of external consultants auditing software artifacts. In the sequel, our conclusions will be based on a comparison of different simulations of this 12207+IV&V process model: a baseline AS-IS simulation; and a TO-BE simulation where the IV&V and V&V work is informed by data mining.

[2]See the proceedings of the *ProSim International Workshops*, at `http://www.prosim.pdx.edu/` and special issues of the *Journal of Systems and Software* (Vol 46, No 2/3, Vol. 47, No. 9 and Vol, 59, No. 3), and the international journal of *Software Process: Improvement and Practice*, Vol 5, No. 2/3, Vol. 7, No. 3/4 and Vol 9, No. 2) on this topic.

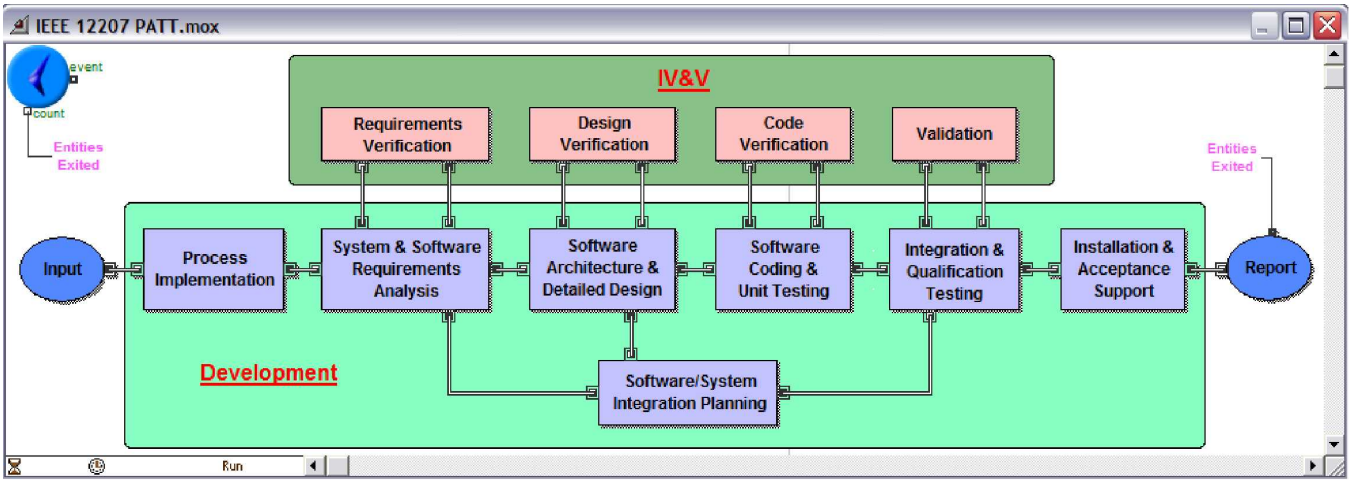

Fig. 4. IEEE 12207 process simulation model with an additional IV&V Layer.

B. Data Mining

Data mining is a summarization technique that reduces large sets of examples to a small understandable pattern using a range of techniques taken from statistics and artificial intelligence [14]–[16]. We use such data miners here to learn defect detectors from defect logs. For the purposes of comparison, this study will employ two data mining methods: entropy - based decision tree learning and a Naive Bayes classifier. For the purposes of repeatability, we will use the public-domain implementations of those algorithms available in the WEKA tool [14].

In *decision tree* learning, the whole training set is *split* 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. Hence, a *good split* decreases the percentage of different classifications in a subset. Such a *good split* ensures that smaller subtrees will be generated since less further splitting is required to sort out the subsets. Various schemes have been described in the literature for finding good splits. For example, the CART [17] decision tree learner uses the GINA index while the C4.5 [18] decision tree algorithm uses an information theoretic measure (entropy) to find its splits. J48 is version eight of C4.5, ported to JAVA.

A *Naive Bayes classifier* tunes past knowledge to new evidence using Bayes' Theorem:

$$P(H \mid E) = \frac{P(H)}{P(E)} \prod_i P(E_i \mid H)$$

That is, given fragments of evidence E_i and a prior probability for a class $P(H)$, a posterior probability $P(H \mid E)$ is calculated for the hypothesis given the evidence. The Bayes classifier returns the class with highest probability. Many studies (e.g. [19], [20]) have reported that, in many domains, this simple Bayes classification scheme exhibits excellent performance compared to other learners. This is a surprising result. Such classifiers are often called *naïve* [14], since they assume that the frequencies of different attributes are independent. In practice [21], the absolute values of the classification probabilities computed by Bayes classifiers are often inaccurate. However, the relative ranking of classification probabilities is adequate for the purposes of classification.

Bayes classifiers can be extended to numeric attributes using *kernel estimation* methods. For this study, we use John and Langley's Gaussian summation kernel estimation method implemented in the WEKA's Bayes classifier [14]. Other, more sophisticated methods are well-established [22], [23], but several studies report that even simple methods suffice for adapting Bayes classifiers to numeric variables [20], [24].

C. Static Code Measures

For this study, our data miners learn from static code measures defined by McCabe [25] and Halstead [26]. McCabe (and Halstead) are "module"-based metrics where a module is the smallest unit of functionality. In C or Smalltalk, "modules" would be called "function" or "method" respectively. For a brief tutorial on the Halstead and McCabe measures, see [27].

We study these static code measures since they are:

- *Useful*: see the business case made in this article;
- *Easy to use*: static code measures (e.g. lines of code, the McCabe/Halstead measures) can be automatically and cheaply collected;
- And *widely used*. Many researchers use static measures to guide software quality predictions; e.g. [27]–[34]. Verification and validation (V&V) textbooks (e.g. [35]) advise using static code complexity measures to decide which modules are worthy of manual inspections. Further, we know of several large government software contractors that won't review software modules *unless* tools like McCabe predict that they are fault prone. Hence, defect detectors have a major economic impact when they may force programmers to rewrite code.

Static code measures are hardly a complete characterization of the internals of a function. Fenton offers an insightful example where *the same* functionality is achieved using *different* programming language constructs resulting in *different* static

dataset	kind	detector
AN1	Halstead	$unique\ operands >= 8.14$
CM1	Halstead	$I \geq 167.94$
JM1	Halstead	$unique\ operands >= 60.48$
KC1	Halstead	$V \geq 1106.55$
KC2	McCabe	$ev(g) >= 4.99$

Fig. 5. Best single measures found in five different defect logs. From [39].

measurements for that module [36]. Fenton uses this example to argue the uselessness of static code measures.

An *alternative interpretation* of Fenton's example is that static measures can never be a definite and certain indicator of the presence of a fault. Rather, defect detectors based on static measures are best viewed as probabilistic statements that the frequency of faults tends to increase in code modules that trigger the detector. By definition, such probabilistic statements will are not categorical claims with some a non-zero false alarm rate. We show below that false alarm rate can be under 10% while still achieving a high probability of detection. That is, on average, these static measures contain some useful indications of the structure of a system.

Shepperd & Ince and Fenton & Pfleeger might reject the *alternative interpretation*. They present empirical evidence that (e.g.) the McCabe static measures offer nothing more than uninformative measures such as lines of code. Fenton & Pfleeger note that the main McCabe's measure (cyclomatic complexity, or $v(g)$) is highly $correlated$ with lines of code [36]. Also, Shepperd & Ince remarks that "for a large class of software it (cyclomatic complexity) is no more than a proxy for, and in many cases outperformed by, lines of code" [37].

Our own experiments with *feature subset selection* (FSS) suggest that static measures can be more informative than suggested by Shepperd & Ince and Fenton & Pfleeger. FSS is the preferred method in the data mining community of find good subsets of the available measures. Hall and Holmes offer a good tutorial and experimental evaluation of seven different FSS methods [19]. For example, the WRAPPER methods of Kohavi and John [38] performs a *hill-climbing search* through the space of possible measures. At each step in the climb, some data mining algorithm is used as an *oracle* to compare the performance of a smaller to a larger set of features.

With Nikora and Ammar [39], we have applied seven different FSS methods like WRAPPER on defect logs comprising lines of code, McCabe and Halstead measures. Two different data miners were used as oracles: the J48 decision tree learner and a NaiveBayes classifier with kernel estimation. Figure 5 shows the detectors learnt using the most influential static measure found by any of the seven FSS methods using either of the two oracles. Note that in no case was lines of code the most influential measure.

Why are we so optimistic about static measures and Shepperd & Ince and Fenton & Pfleeger are so pessimistic? One possibility is that we use a different set of *assessment criteria* for detectors. A commonly used assessment criteria is *correlation* (defined in Figure 6) and, as the following example shows, correlation may be uninformative on the merits of a measure for detecting a defect.

Elsewhere [27], we have applied linear regression to logs of defects in modules described only in terms of lines of code (LOC) or the Halstead (HAL) measures. This generated these equations:

Let a_i and p_i denote some actual and predicted values respectively. Let n and $\overline{x}$ denote the number of observations and the mean of the n observations, respectively. Correlation c is then calculated as follows:

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

Fig. 6. Correlation

PD:

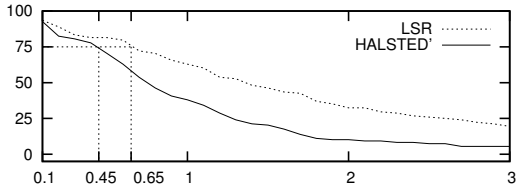

Fig. 7. Probability of detection seen using $defects_i \geq T$ where $defects_i$ is one of Equation 1 (the "LSR" curves) or Equation 2 (the "HALSTEAD" curves) and T controls when the detector triggering.

$$\begin{aligned} defects(LOC) &= 0.0164 + 0.0114 * LOC && (1)\\ c(LOC) &= 0.65 \\ defects(HAL) &= 0.231 + (0.00344 * N) + (8.88e-4 * V) && (2)\\ &\quad -(0.185 * L) - (0.0343 * D) - (0.00541 * I) \\ &\quad +(1.68e-5 * E) + (0.711 * B) \\ &\quad -(4.7e-4 * T) \\ c(HAL) &= -0.36 \end{aligned}$$

At first glance, the equations seem to endorse the thesis that seemingly sophisticated static measures are even more useless than lines of code. The simple LOC-based predictor (Equation 1) correlates at $c = 0.65$ to defects. This is much stronger that the correleation of $c = -0.36$ seen in the the Halstead-based detector (Equation 2). However, further study shows that the correlation of these equations is *irrelevant* to their merits as a defect detector. Equation 1 and Equation 2 can be converted to a detector by combining them with some threshold value T. If a module generates numbers from these equations that exceeds the threshold, then a detector is *triggered*. For all the modules, the probability of detection (PD) is the percentage of modules with known defects that lead to a trigger. Figure 7 shows how PD changes as the threshold changes 0.1 to 3 for the KC2 dataset from Figure 2. Note that a $PD = 75\%$ can be reached using either method by setting $T \geq 0.65$ or $T \geq 0.45$. That is, *either* equation can reach some desired level of detection *regardless of their correlations* just by using different threshold values.

Our conclusion from this example is that *candidate measures for defect detectors need to be assessed on more than just correlation*. The next section offers several such *specific criteria* and *meta-criteria*.

III. Instead of Correlation

A. Specific Criteria

Particular measures can be assessed by combining them to form detectors and then assess each *specific* detector using:

- PD: the probability of defecting faults (a.k.a. recall);
- Accuracy;
- The effort associated with the detector triggering;
- The probability of false alarm (PF).

To formally define the specific criteria, we say that a defect detector hunts for a *signal* that a software module is defect prone. Statistics on a detector can be kept in a 2-by-2 matrix of Figure 8. If a detector registers a signal, sometimes the signal is actually present (cell D) and sometimes it is absent (cell C). Alternatively, the detector may be silent when the signal is absent (cell A) or present (cell B).

If the detector registers a signal, there are two 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:

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

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

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

Another criteria of interest is the accuracy, or "Acc", of a detector which is the number of true negatives and true positives seen over all events:

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

For example, the numbers in the ABCD cells of Figure 8 show the number of KC2 modules that a Naive Bayes classifier allocated to each cell. Based on those numbers:

$$\begin{array}{rcccccc} PD & = & \frac{D}{B+D} & = & \frac{48}{60+48} & = & 44\% \\ PF & = & \frac{C}{A+C} & = & \frac{23}{392+23} & = & 5\% \\ Acc & = & \frac{A+C}{A+B+C+D} & = & \frac{392+48}{392+60+23+48} & = & 84\% \end{array}$$

Yet another criteria of interest is the work required *after* a detector is triggered. Based on cost-model developed by one contractor at NASA's IV&V facility [40], and a model from Forrest Shull (personnel communication), our analysis will assume that inspection $Effort$ is linearly proportional to lines of code. Under that assumption, the inspection $Effort$ for a detector is proportional to the percentage of the lines of code in a system are selected by a detector. If the lines of code

		signal present? no	signal present? yes
signal detected?	no	A= true negative 392	B 60
	yes	C 23	D= true positive 48

Fig. 8. Statistics for applying the specific criteria.

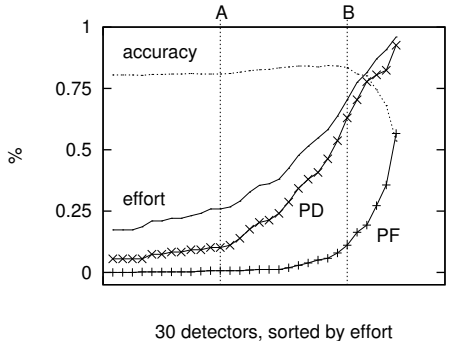

Fig. 9. At each x-axis value, x describes the $pf, pd, effort, accuracy$ of one detector. From [27].

in the modules falling into each cell of Figure 8 is LOC_A, LOC_B, LOC_C, and LOC_D, then:

$$Inspection\ Effort = \frac{LOC_C + LOC_D}{LOC_A + LOC_B + LOC_C + LOC_D} \quad (3)$$

Often it is the larger modules trigger the detector and fall into cells C and D. Hence, while only $\frac{23+48}{392+60+23+48} = 14\%$ of the modules fall into cells CD of Figure 8, these represent an $Effort$ of 56% of the code. Such a result lends support that lines of code is a good predictor for defects. However, as seen above in Figure 5 other static measures can be better.

Ideally, a detector has a high probability of detection, a low effort, and a low false alarm rate. In practice, this is hard to achieve. The general pattern of Figure 9 has been observed in hundreds of defect detectors generated from various subsets of the available static measures from the Figure 2 data sets using a wide variety of data miners including decision tree learners, model tree learners, linear regression and a home brew learner called "ROCKY" [27], [30]. Each x-value of that figure describes one detector. The y-values on that figure offer four values for each detector: effort, PD, PF, and accuracy. The detectors are sorted on effort. The salient features of Figure 9 are as follows:

- Not surprisingly, to detect *more* faults, our detectors must trigger on *more* modules. Hence, $Effort$ hovers above PD.
- There exists a large number of detectors with very low false alarm rates; i.e. $PF \leq 10\%$.
- Very high probabilities of detection usually means triggering many modules which, in turn, increases *both* the false alarm rate and the effort. Hence, high PDs come at the costs of high PFs and $effort$.
- Accuracy can be as uninformative as correlation for predicting detection, false alarms, and effort. Consider the detectors marked A and B on Figure 9. These two detectors have nearly the same accuracy, yet with $effort$s, PDs, and PFs that can vary by a factor as large as 4.

B. Meta-Criteria and Sequence Studies

Apart from the criteria used to assess a specific detector, it is also insightful to consider certain *meta-criteria*. If the goal

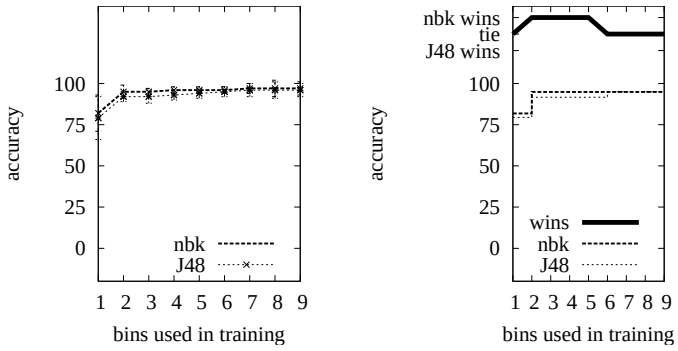

Fig. 10. Learning curves (left); summarized (right).

is to generate detectors that have some useful future validity, then an important meta-criteria is how well the detector performs on data *not* used to generate it. Failing to do so can result in an excessive over-estimate of the learned model-for example, Srinivasan and Fisher report an 0.82 correlation between the predictions generated by their learned decision tree and the actual software development effort seen in their training set [34]. However, when that data was applied to data from another project, that correlation fell to under 0.25. The conclusion from their work is that a learned model that works fine in one domain may not apply to another.

In order to apply this meta-criteria, a commonly-used meta-criteria assessment is a M*N-way cross validation study where, the data is sorted into a random order M times [14]. For each order, the data is divided into N bins and the theory learned on N-1 bins is tested on the remaining *hold-out* bin. This procedure allows a prediction on how well the learner will perform on new data.

Any empirical conclusion, such as ours, should come with a *self-test* that can quickly and clearly recognizes when an old conclusion does not apply to a specific new situation. To implement that self-test, we modify the standard N*M-way procedure to find the lower-limit on how many examples are required to learn a detector. Our modified procedure is called a *sequence study* and is defined by the tuple $<$*L,M,N,X*$>$. In summary, a sequence study tests how well we can learn a theory using *less and less data*:

- M times the data ordering is randomized.
- The first L examples are divided into N bins. The class distribution within the N bins is selected to be similar to the class distribution in the original data set.
- $\frac{X}{N} * L$ of the data for $X \in \{1, 2, \ldots N-1\}$ is used for training and...
- The remaining $\frac{N-X}{N} * L$ of the data is used for testing.

The "L" in the sequence study is very important since this it the upper limit on the number of examples processed M*N*X times. Such upper limits are required when processing very large data sets (e.g. the JM1 data set in Figure 2 which has 10885 modules).

Figure 10 shows the results from a sequence study where $<$*L=150,M=N=10,X=1...9*$>$ on the IRIS data set from the UCI machine learning data repository [41]. The plot on the left shows the mean classifier accuracy improving as more and more of the data is used to train J48 and a Naive Bayes classifier with kernel estimation.

The error bars in Figure 10 show ± 1 standard deviations for the accuracies seen in the M repeats. Those standard deviation values can be used in t-tests to generate the right-hand-side *sequence summary plot* of Figure 10. In a sequence summary plot, the display of the mean accuracy of a classifier only changes if it is statistically difference (at the 95% level) to the last seen change; otherwise the displayed mean value is the same as the value seen at the last change. The summary plot shows that there was little significant improvement in classification accuracy of Naive Bayes or J4.8 after using 20% of the data. Further, there was no significant improvement in either method after using 60% of the data.

On top of the y-axis of the sequence summary plot are three marks: *nbk wins*; *tie*; and *J48 wins*. The plot to the right of these marks are a comparison of the performance of the two learners at each X value. One learner *wins* over the other if it is *both* statistically different (using a t-test at the 95% level) *and* the mean performance of one learner is larger than the the other. In the case of this study with the IRIS data set, Naive Bayes won over J48 four times; J48 never won; and both learners tied four times.

In an sequence study, a learner is said to have *plateaued* when the curves in the summary plot stop changing; i.e. there are no significant changes to the performance of the learner. In Figure 10, significant changes to the learners can be seen twice: a large change after training on 20% of the data and a very small change (in J48 only) after seeing 60% of the data.

A *phantom change* is when the mean of one learner at x_2 is significantly different to a proceeding x_1 change *but* the win-tie plots shows that the change is not significantly different to the other learner. In Figure 10 J48's second change at 60% is *not* a phantom since the win-tie report also changes at 60%.

The start of this plateau (in Figure 10 after 60%*150=80 examples) defines the point at which further data is superfluous. At this point, we can report the N number of examples needed to learn a detector. If this *plateau point* is small then it would also be possible to quickly determine how data is required before we can assess if the detector will be adequate.

IV. Results for Learning Defect Detectors

Figure 11 shows the results for a $<$*L=500,M=N=10, X=1...9*$>$. sequence study on the Figure 2 data using a NaiveBayes (with kernel estimation) classifier to an entropy-based decision tree learner (J48) [14]. In all, learning was conducted 2700 times. These calls divide into 135 calls where the two learners were executed on the same data within a 10-way cross-validation experiment.

For space reasons, only the summary graphs for PD and PF are shown. KC2's PD exhibits a phantom drop at $x = 50\%$. That drop is a phantom since win-tie plot shows the change is indistinguishable from the J48 plot.

We saw above in Figure 9 that accuracy can be uninformative for predicting false alarms (PF) and prediction (PD). The same effect was seen in this experiment. For all our runs, the accuracies hover between 75% and 85%. However, as seen in

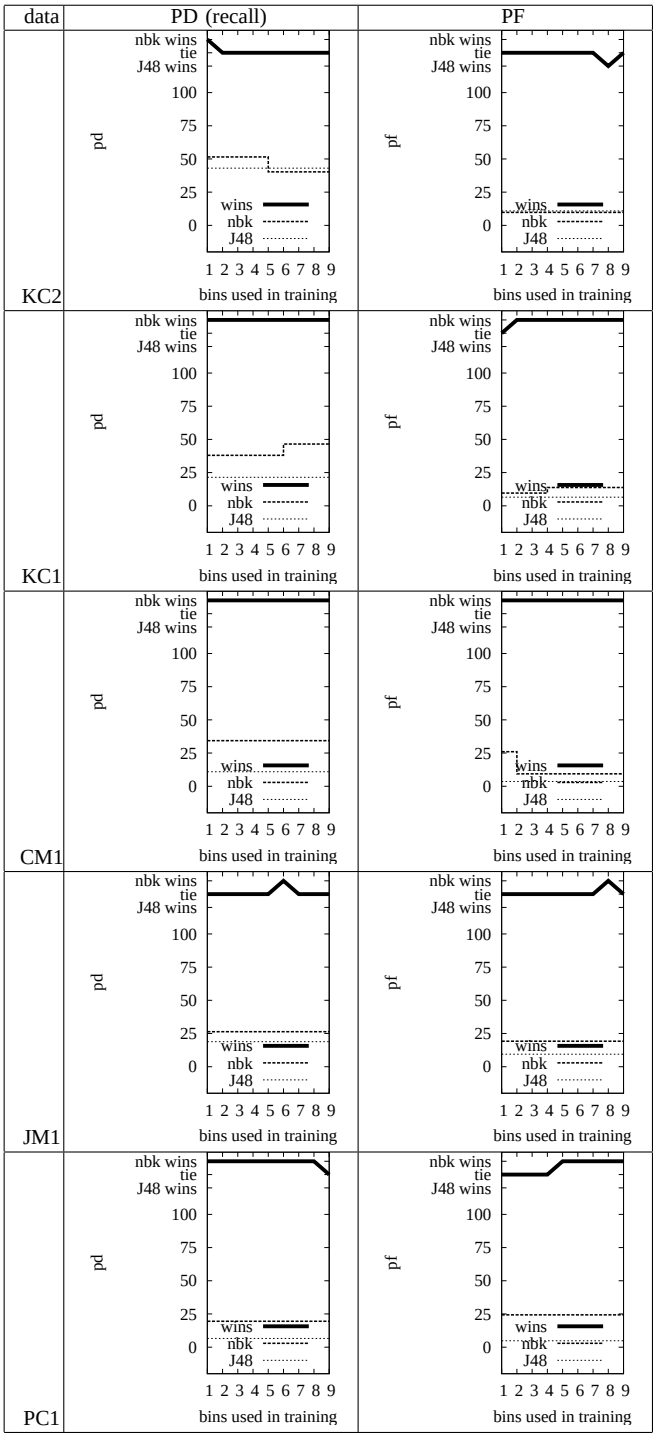

Fig. 11. PD and PF results from a sequence study for Figure 3 data.

Figure 11, these small changes in accuracy are associated with large chances to PF and PD:

- PD change by a factor of five from 10% (PC1's J48 results) to 50% (KC1 and KC2's PD).
- PF change by a factor of four from 5% (in PC1's J48 results) to 20% (in JM1's nbk results).

Based on the Figure 9 and Figure 11 results, we hence *advise against using accuracy to assess defect detectors*.

Figure 11 also lets us assess the merits of different learning methods. At the 95% level, Naive Bayes won 51 of those 90 cross-validations; J48 won once (see the KC2 PF curve at $bins = 8$) , and in the remaining 90-51-1=48 times, the two methods tied. This means that in $\frac{90-1=89}{90} \approx 99\%$ of the cross-validation experiments, measured in terms of PD and PF, Naive Bayes generated equivalent or better defect detectors than J48. Hence, *Naive Bayes is better than J48 for learning defect detectors*.

An interesting feature of Figure 11 are the *early plateaus* seen in all the runs. KC1's PD plateaus after $500 * \frac{6}{10} = 300$ modules. The other PD plots plateaus much earlier: i.e. after only $500 * \frac{1}{10} = 50$ modules. Early plateau lets us implement the *self-test* described in the introduction: i.e. a quick and clear recognition when a conclusion is failing. Such early plateaus means that *after sampling 50 modules it is be possible to determine detector effectiveness for a particular data set.*

At the plateau point, the height of the plateau seems to be determined by how certain features of the data set. Figure 11 is sorted in according to maximum reached PD and this order corresponds to how specialized was the code. KC2 and KC1 are shown on top and these have the highest PDs and precisions. JM1 and PC1 are shown at bottom and these have the lowest PDs. Note that this sort order corresponds to the system/sub-system break up shown in Figure 3. That is, defect detectors learnt from data coming from below the sub-system level (e.g. KC1 and KC2) have a higher probabilities of finding faults than defect detectors learnt for an entire system. Note also the false alarm rate is largest at the system level (JM1, PC1 $PF \leq 25\%$) and smallest below the sub-system level (KC1, KC2 $PF \leq 10\%$). Hence, we recommend *learning defect detectors from data sets divided below the sub-system level*. We call this improvement in PD via learning from data specialized below the sub-system level the *stratification effect* and will explore it further below.

In Figure 11, the height of the PD plateaus found by Naive Bayes was 20%, 25%, 30%, 50% and 50% in PC1, JM1, CM1, KC1 and KC2 respectively[3]. These PDs are reached at a much lower cost than what is required for the equivalent manual task. Menzies and Raffo [40] report one manual inspection cost model used by a NASA V&V contractor where 8 to 20 lines of code are inspected per minute. This effort repeats for all four to six members of the review team, which can be as large as four or six. Another cost model comes from Shull (personnel communication) who allows 4 hours to inspect 500 lines of code (2 hours for meeting preparation; 2 hours for the meeting); i.e. 2 LOC/minute- four times slower than the cost model described above. By contrast, our detectors can flag worrying modules in a fraction of that time[4], once from prior inspections has been collected into tables describing modules in terms of Halstead and McCabe metrics. Hence, *defect detectors learnt from static code measures can operate at a fraction of the cost of manual inspections*.

[3]KC2's PD plateau is recorded at 50% since the drop in that plot at bins=5 in Figure 11 is a phantom and, at bins=1 in Figure 11, Naive Bayes's PD of 50% can be distinguished from J48 (i.e. in that first bin, NaiveBayes wins over J4.8).

[4]For example, a ten-way cross-validation on KC2 using Naive Bayes takes 5.3 seconds on our Solaris machines.

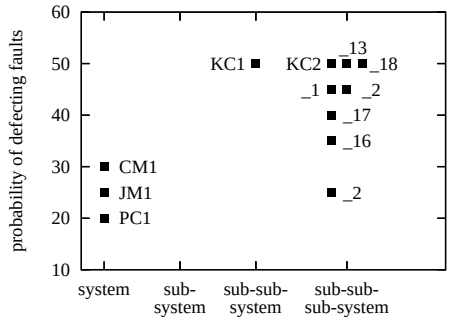

Fig. 12. Naive Bayes PD at plateau in Figure 2 data.

We hasten to add that by "operate" we mean only that oracles that flag faulty modules can be automatically generated and applied very quickly. Once that operation completes, analysts still have to *manually inspect* the flagged modules. The overall economic savings from this automatically-operate then manually-inspect-some modules is discussed in the next section.

In order to assess the generality of the stratification effect, we extracted sub-divisions of the KC1 sub-system. Seven of these sub-sub-systems had defect logs and those contained 101,124,127,193,240,264, and 385 modules. A *<L,M=N=10,X=1...9>* study was conducted on each sub-sub-sub-system with L set to the total number of modules in each data set. The results of that study echo our above conclusions:

- *Naive Bayes is better than J48 for learning defect detectors.* In these seven sub-divisions of KC1, NaiveBayes won 16 of the 10-way cross-validation experiments; J4.8 won 6 times and the two learners tied 32 times.
- *Detectors usually exhibit very early plateau.* In all sub-divisions of KC1, plateau was reached after 50 modules.
- *The PDs can rise to up to 50%.* The observed height of the PD plateaus in the KC1 sub-divisions were 25,35,40,45,45,50,50%.
- *Learning from data divided below the sub-system level improves PD.* The PD plateaus points of the KC1 sub-division, and the PD plateau points generated above, are shown in Figure 12. The x-axis of that plot shows the level of the data set: from "system" on the left-hand-side to "sub-sub-system" on the right-hand-side. The plateaus from sub-divisions of KC1 are marked $\_1, \_2, \ldots, \_18$ etc. With the exceptions of _2 and _16, the general trend in Figure 12 is clear: sub-division improves PD.

As to _2 and _16, the early plateau effect seen in Figure 11 shows that we can quickly check for software like _2 and _16 that generates low PD detectors. Specifically, if after sampling 50 modules collected below the sub-system level, the detectors found by a Naive Bayes classifier do not achieve high PDs, then the conclusions of the next section do not hold.

V. Business Implications of Defect Detectors

Before turning to the business case simulations, we need to state some underlying assumptions. One important issue is the meaning of PD (probability of detection). The above results show PDs learned from defects logs containing issue reports from multiple sources: inspections, software tests, hardware tests, formal method results, etc. As a result of this extensive checking, we assert that the PDs shown above are equal to the percentage of defects *ever* found in the system.

Another kind of PD, would be that seen in data miners executing on just the issue reports seen in the most recent manual inspections. In this case, PD changes to a smaller value (which we denote PD′) since our learners might only find a certain percentage of the defects contained in the inspection logs which, in turn, is some percentage of the total number of defects.

Further research is required to determine the *actual* value of the PD′ learned from just inspection logs. While we await that data, we run two scenarios. In Scenario I, we will assume PD refers to a percentage of the total number of errors; i.e. the IV&V situation where we are learning from rich defect logs. In Scenario II, we will assume PD′ i.e. the V&V situation where we are only learning from the results of inspections. Scenario I's conclusions will endorse using defect detectors while Scenario II's conclusions will be more negative. Hence, we also develop Scenario III where we assess the root cause of the findings in Scenario II to find a remedy to reverse those negative conclusions.

We also make several other assumptions:

- A1: The project is 100,000 lines of code.
- A2: Figure 4 shows the assumed software process: i.e. IEEE 12207+IV&V model.
- A3: Figure 11 shows how many modules our data miners can use to learn defect detectors; i.e. 50 modules.
- A4: Equation 3 and Figure 9 shows the assumed effort associated with our detectors being triggered; i.e. $Effort = PD + 5\% \ldots 10\%$.
- A5: Figure 9 shows the assumed PF with our detectors; $PF \leq 10\%$. There is additional support for this PF assumption: $PF \leq 10\%$ can also been seen in Figure 11 for defect detectors learned from below sub-system data (KC1 and KC2).
- A6: Figure 12 shows the assumed PD associated with the detectors; i.e. $PD = 40 \ldots 50\%$.
- A7: Assumption A6 assumes, in turn, that defect detectors are learnt from data divided below the sub-system level.
- A8: Standard manual inspections find 40% to 60% of the total defects.

In principle, these assumptions can be checked and adjusted as necessary. This is one of the strengths of process simulation. Our current process model has been built and repeatedly checked over the last three years.

As to the other assumptions, A1 and A8 are simply input parameters to our model and can be easily changed prior to a new set of simulations. A3 is based on the early plateau effect and that can be checked using sequence studies using as little as 100 modules. If A3 holds, then checking assumptions A4,A5,A6 is a simple matter: just use the procedure described at the end of the last section.

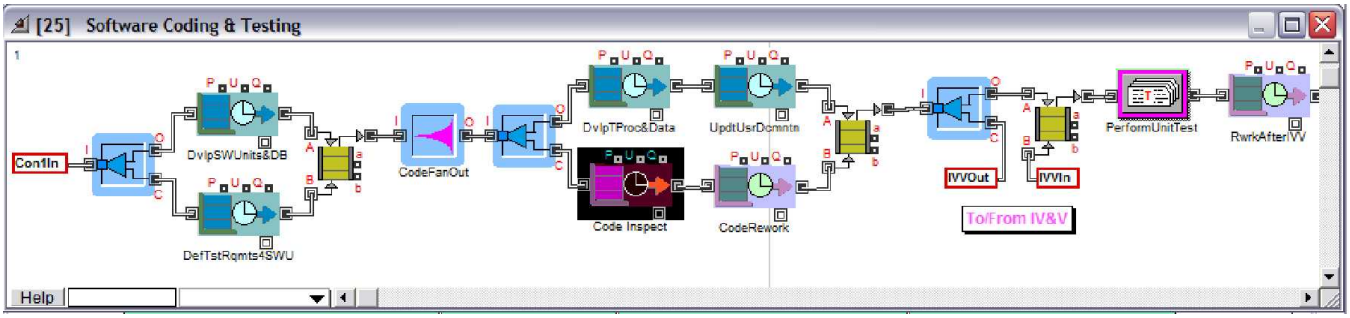

Fig. 13. Coding and unit test process showing Fagan inspections.

	Total Size (KLOC)	Total Effort (PM)	Total Rework Effort (PM)	Total Duration (Month)	Average Duration	Total Defect Corrected	Total Latent Defects	Code Inspection Effort (PM)
Average	99.79	781.41	160.56	32.81	28.51	5,907.08	507.81	9.67
Std Dev	4.00	27.66	6.95	1.43	1.22	257.00	22.00	0.29

Fig. 14. AS-IS Baseline Process Performance predicted by the Simulation Model

A. Baseline Model Results

Baseline performance is predicted in terms of development effort (or cost), effort devoted to rework, IV&V effort, project duration, corrected defects, and escaped (or delivered) defects.

Our baseline of the AS-IS process presumes that full Fagan inspections [42] are done at all development phases- including at the coding phase (see Figure 13). The actual baseline performance for the AS-IS process (without using data miners applied to defect detectors as part of IV&V activities) can be seen in Figure 14.

B. Scenario I: Defect Detectors and IV&V

For the first TO-BE process scenario, we apply the defect detectors as part of an independent verification and validation (IV&V) step after coding and code inspections are complete (see Figure 13). Is this TO-BE scenario, defect detectors are utilized in IV&V work as follows:

- Defect logs and code modules that have completed code inspections and other forms of testing are sent to IV&V.
- Defect detectors are learned on the logs and then applied to 100% of the code. Once the logs are in a format suitable for the learners, this can be done automatically and quickly (a mere matter of seconds). In our simulations we assume that preparing the input logs takes two person days(or 16 hours of effort) for a large 100 KSLOC project.
- The defect detectors identify code modules that are likely to contain defects. Since the code modules will have gone through code inspections and other assessment measures during project level V&V, many of the modules that are identified will already be known to contain defects. These modules will not be looked at again. Instead, the defect detectors will be used to identify modules where no defects were found during their initial code inspections, but whose characteristics indicate that these modules are likely to have defects. From Figure 8, we see that this procedure will select the modules falling into cell C (i.e. modules where the signal is detected, but where defects were not found).
- The modules that trigger the detectors are then re-inspected. The re-inspection rate's upper bound is PF; i.e. 10% (from [A5]).

To assess the impact of learning data miners for defect detectors, then using them in IV&V mode, the next parameter required is an estimate of the percentage of the escaped defects that will be found using the above procedure. At present, more research is necessary to empirically determine this percentage. While we await those results, we can use the process simulation model to identify the minimum percentages required in order to break even (where expenses equal benefits). Moreover, the process simulation model can help assess the risk of applying the defect detectors by assessing the worst-case scenario (i.e. when no additional defects are detected).

The results of these tests are shown in Figure 15 and, for the purposes of this discussion, we focus on the cells marked with a black triangle (◄). These cells show the difference between the baseline data of Figure 14 (repeated at the top of Figure 15) and the results from Scenario I. As can be seen, using defect detectors breaks-even (i.e. the $Delta$ goes from negative to positive) if the above approach can detect an additional 1 to 2% of the latent defects in the code and starts showing a positive benefit in both effort and latent defects at 3%. The worst case is that an additional 1.16 person months would be expended doing inspections that do not find any new defects.

Moreover, if 5% or 10% of the latent defects are found, the quality of the code would be improved by an average 15.5 and 32.5 defects respectively and an an average 2.5 and 6.5 person months of effort respectively could be saved.

To repeat, the *minimum* performance target for defect detectors to be beneficial for IV&V would be 3% additional defects detected and the *maximum* exposure would be 1.16 person months of effort. Based on our commercial work

	Total Size (KLOC)	Total Effort + IV&V (PM)	Total Effort (PM)	Total Rework Effort (PM)	Total Duration (Month)	Average Duration	Total Defect Corrected	Total Latent Defects	Code Inspection Effort (PM)
Baseline									
Average	99.79	781.41	781.41	160.56	32.81	28.51	5,907.08	507.81	9.67
Std Dev	4.00	27.66	27.66	6.95	1.43	1.22	257.00	22.00	0.29
Case 1: detcap IV&V = 0									
Average	99.79	782.57	781.41	160.56	32.85	28.51	5,907.08	507.81	10.83
Std Dev	4.00	27.67	27.66	6.95	1.43	1.22	257.00	22.00	0.30
Deltas		-1.16 ◄	0.00	0.00	-0.05	0.00	0.00	0.00 ◄	-1.16
Case 2: detcap IV&V = 0.01									
Average	99.79	782.06	780.90	160.05	32.85	28.50	5,928.55	505.79	10.83
Std Dev	4.00	27.67	27.65	6.93	1.43	1.22	258.00	22.00	0.30
Deltas		-0.65 ◄	0.51	0.51	-0.04	0.00	-21.47	2.02 ◄	-1.16
Case 3: detcap IV&V = 0.02									
Average	99.79	781.27	780.11	159.29	32.83	28.49	5,931.95	502.40	10.83
Std Dev	4.00	27.66	27.64	6.89	1.43	1.22	258.00	21.00	0.30
Deltas		0.14 ◄	1.30	1.27	-0.03	0.02	-24.87	5.41 ◄	-1.16
Case 4: detcap IV&V = 0.03									
Average	99.79	780.48	779.32	158.53	32.82	28.47	5,935.34	499.00	10.83
Std Dev	4.00	27.64	27.63	6.86	1.43	1.22	258.00	21.00	0.30
Deltas		0.93 ◄	2.09	2.03	-0.01	0.03	-28.26	8.81 ◄	-1.16
Case 5: detcap IV&V = 0.04									
Average	99.79	779.68	778.52	157.77	32.80	28.46	5,938.73	495.61	10.83
Std Dev	4.00	27.64	27.62	6.83	1.43	1.22	258.00	21.00	0.30
Deltas		1.72 ◄	2.88	2.79	0.00	0.05	-31.65	12.20 ◄	-1.16
Case 6: detcap IV&V = 0.05									
Average	99.79	778.89	777.73	157.01	32.79	28.44	5,942.13	492.22	10.83
Std Dev	4.00	27.63	27.61	6.80	1.43	1.21	259.00	21.00	0.30
Deltas		2.51 ◄	3.67	3.55	0.02	0.06	-35.05	15.59 ◄	-1.16
Case 7: detcap IV&V = 0.10									
Average	99.79	774.94	773.78	153.21	32.72	28.37	5,959.10	475.24	10.83
Std Dev	4.00	27.58	27.57	6.63	1.43	1.21	259.00	20.00	0.30
Deltas		6.47 ◄	7.63	7.35	0.09	0.14	-52.02	32.57 ◄	-1.16

Fig. 15. Operational Scenario I: Using Defect Detectors in IV&V Mode

with companies exploring re-inspections of their code, we are confident that the 3% additional defects threshold can be easily surpassed. However, in this forum, we cannot publish supportive evidence for this claim since it is based on proprietary data.

C. Scenario II: Defect Detectors and Inspection-based V&V

Scenario II is the case where analysts have a *much weaker* training set; i.e. *only* the results of internal inspections. This is the PD′ case where the data miners only find *some* the defects in defect logs which contain only *some* of the project defects.

One situation where this could happen would be when a team declines to wait for the IV&V team to report issues. Instead, they use their own experience of, say, their code inspections to build defect detectors. For example:

- A minimum set of code was inspected. From assumption A3, 50 modules per sub-sub-system, or 11.5% of the code, would be required to achieve plateau performance.
- Using the defect logs from these inspections and the inspected modules, defect detectors were learned using data miners and applied to the rest of the code.
- Modules were identified as likely candidates for defects.
- Only those portions of the code that are tagged as likely ”hot spots” were inspected.

Under this scenario, the defect detectors would select 61.5% of the code for further inspection. This would result in 38.5% reduction in inspection effort (approximately 3.7 person-months) and inspection schedule savings. However, process simulation shows that the savings in inspection effort would not offset the increase in defect detection and rework costs associated with finding these defects later in the development process. Figure 16 shows the baseline results and results of having an expected defect detection capability of 47% will cause the process using defect detectors during project V&V to break even on effort (but have an overall poorer quality).

If we are learning on an inspection log containing 50% of all defects (the expected case), then such a 47% overall defect detection rate is only possible if a data miner can learn near-perfect detectors with a PD of 98% (i.e. $50\% * 98\% = 47\%$). Since this is highly unlikely (to say the least), the conclusion of Scenario II must be to doubt the value of defect detectors for improving V&V.

D. Scenario III: Fixing Scenario II

If our process simulation methodology is rich enough, we should be able to query our models to find some repair to Scenario II. This section searches for such a repair.

We diagnosed the problem identified in Scenario II as *learning from impoverished training examples*. As discussed above, the effectiveness of the learned defect detectors approach depends upon the training examples used. If the training examples themselves are not effective at detecting defects, the learned defect detectors will not be either.

If a diagnosis is useful, it should suggest a repair. To repair the problem we explored ways to improve the training set. One possibility would be to *improve* the methods used to generate the inspection reports. A candidate technology for that improvement is Shull’s *perspective-based* inspection methods [43] where each reviewer explores the code with different goals; e.g. one reviewer might only check for termination on looping constructs while another might check

	Total Size (KLOC)	Total Effort (PM)	Total Rework Effort (PM)	Total Duration (Month)	Average Duration	Total Defect Corrected	Total Latent Defects	Code Inspection Effort (PM)
Baseline								
Average	99.79	781.41	160.56	32.81	28.51	5,907.08	507.81	9.67
Std Dev	4.00	27.66	6.95	1.43	1.22	257.00	22.00	0.29
Case 1: detcap 2 = 0.48								
Average	99.79	780.24	163.24	32.74	28.44	5,895.33	519.56	5.52
Std Dev	4.00	27.52	7.07	1.43	1.21	257.00	22.00	0.14
Deltas		1.17 ◄	-2.68	0.07	0.07	11.75	-11.75 ◄	4.15
Case 2: detcap 2 = 0.47								
Average	99.79	781.53	164.58	32.78	28.47	5,889.45	525.43	5.42
Std Dev	4.00	27.54	7.13	1.43	1.22	256.00	22.00	0.14
Deltas		-0.13 ◄	-4.02	0.03	0.03	17.63	-17.62 ◄	4.24

Fig. 16. Operational Scenario II: Using Defect Detectors in V&V Mode.

that they have enough information to divide system input into equivalent classes. Shull's key argument is that having such specific goals (which he calls "perspectives") means that a reviewer can work faster and more effectively since their attention is not distracted by numerous side-issues. Further, since multiple reviewers are looking at multiple different issues, there is less likelihood for the different reviewers to discover the same issues.

Perspective-based inspections (PBR) have been shown to detect approximately 85% of latent defects [43]. Using these highly effective inspections for the training data set, and assuming that the PDs of the learned theories remain at 50% of the defects in the training set, then enables the learned defect detectors to achieve $\text{PD}' = 85\% * 50\% = 42.5\%$.

The benefit of using process simulation models is the ability to quickly do "what if" analysis and to optimize the implementation of new technologies so that they achieve good results. In addition, using the simulation model, the break-even point for this operational scenario can also be found.

Hence, in situations where code inspections achieve 47% defect detection capability (effectiveness) or less, the combined approach of using perspective-based inspections on approximately 12% of the code coupled with learned defect detectors can provide a payoff in terms of reduced effort and reduced delivered defects. The farther the baseline inspections are below the 47% break even point, the stronger the benefit with perspective-based methods.

VI. Conclusion

We are not the first to use data mining to develop defect detectors. Prior work in this area includes [31]–[34]. However, with the exception of Porter and Selby [31], much of the prior work has focused on the technical details of the defect detectors and not the business implications of the technology. Our belief is that this report is a far more detailed analysis of the properties of defect detectors learned for metrics repositories than has appeared previously.

An important part of our analysis was process simulation. Process simulation is a powerful tool for conducting what-if queries on software processes. We have shown above two such what-ifs: in Scenario I we did not know the impact of applying data miners to IV&V yet, in Figure 15, we could still identify the break even point where such mining was useful. Also, in Scenario III, we could repair the negative result of Scenario II by finding a break even point in the knowledge content of our defect logs, after which data mining helped V&V.

Apart from process simulation, we also offer several specific conclusions about data mining and software repositories. Firstly, Naive Bayes classifiers seem better suited to handling noisy real-world defect logs than J48.

Secondly, prior pessimism on the utility of static code measures may be incorrect. Halstead and McCabe *are* more informative than mere lines of code (recall Figure 5). Perhaps we need to assess static code measures on more than just correlation or accuracy. We have seen above (in Figure 7) that correlation can be an orthogonal measure to the probabilities of defect detection and false alarm rates. A similar disconnect between accuracy and PD/PF was seen in Figure 9.

Thirdly, our analysis of the NASA data sets found several important effects that are worth checking for in other data sets:

- The *stratification effect* of Figure 12 where learning on data divided below the sub-system level improved PDs;
- The *early plateau effect* of Figure 11 where the conclusions from our data miners stabilized after seeing only 50 modules.

If these effects repeat in other data sets then they promise that good defect detectors can be learned from surprisingly little data.

More generally, we see this kind of analysis as being very beneficial for more than just assessing defect detectors learned from software repositories. The above process simulation is an example of a general technological assessment process where we can:

- Identify the conditions under which application of a new technology would be beneficial.
- As importantly, we can identify situations when applying this technology would *not* be beneficial.
- We have can performance benchmarks or criteria that vendors of a new technology would need to achieve in order for an organization to consider investing and adopting their technology.
- We can diagnose problems associated with implementing a new tool or technology and identify new and creative ways to apply the technology to the benefit of the organization (and the vendors)
- Finally, we can do all this *before* the technology is purchased or applied and therefore can save scarce resources available for process improvement.

ACKNOWLEDGMENTS

This research was conducted at West Virginia University, Portland State University, partially 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] 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.

[2] W. Curtis, "Building a cost-benefit case for software process improvement, notes from tutorial given at the seventh Software Engineering Process Group Conference, Boston, MA,," May 1995.

[3] R. Dion, "Process improvement and the corporate balance sheet," *IEEE Software*, pp. 28–35, July 1993.

[4] W. Harrison, D. Raffo, J. Settle, and N. Eickelmann, "Adapting financial measures: Making a business case for software process improvement," *Software Quality Journal*, vol. 8, no. 3, 1999.

[5] T. McGibbon, "A business case for software process improvement, data analysis center for software state-of-the-art report, prepared for rome laboratory,," September 1996, Available from `http://www.dacs.dtic.mil/techs/roi.soar/soar.html`.

[6] D.M. Raffo, "Modeling software processes quantitatively and assessing the impact of potential process changes of process performance," May 1996, Ph.D. thesis, Manufacturing and Operations Systems.

[7] D. Raffo and M. Kellner, "Impact of potential process changes: A quantitative approach to process modeling," in *Elements of Software Process Assessment and Improvement*, K. El Emam and N. Madhavji, Eds. 199, IEEE Computer Society.

[8] D. Harter S. Slaughter and M. Krishnan, "Evaluating the cost of software quality," *Communications of the ACM*, pp. 67–73, August 1998.

[9] R. Vienneau, "The present value of software maintenance," *Journal of Parametrics*, pp. 18–36, April 1995.

[10] G.A. Hansen, *Automating Business Process Reengineering*, Prentice Hall, 1997.

[11] M. Laguna and J. Marklund, *Business Process Modeling, Simulation, and Design*, Pearson Prenctice Hall, 2004.

[12] Institute of Electrical and Inc. Electronics Engineers, "Iso/iec 12207 standard for information technology - software lifecycle process," 1998.

[13] C. Jones, *Applied Software Measurement (second edition)*, McGraw Hill, 1991.

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

[15] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.

[16] G. Boetticher, "An assessment of metric contribution in the construction of a neural network-based effort estimator," in *Second International Workshop on Soft Computing Applied to Software Engineering, Enschade, NL*, 2001, Available from: `http://nas.cl.uh.edu/boetticher/publications.html`.

[17] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, "Classification and regression trees," Tech. Rep., Wadsworth International, Monterey, CA, 1984.

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

[19] M.A. 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.

[20] James Dougherty, Ron Kohavi, and Mehran Sahami, "Supervised and unsupervised discretization of continuous features," in *International Conference on Machine Learning*, 1995, pp. 194–202.

[21] Z. Z. Zheng and G. Webb, "Lazy learning of bayesian rules," *Machine Learning*, vol. 41, no. 1, pp. 53–84, 2000, Available from `http://www.csse.monash.edu/~webb/Files/ZhengWebb00.pdf`.

[22] G.H. John and P. Langley, "Estimating continuous distributions in bayesian classifiers," in *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence Montreal, Quebec: Morgan Kaufmann*, 1995, pp. 338–345, Available from `http://citeseer.ist.psu.edu/john95estimating.html`.

[23] U M Fayyad and I H Irani, "Multi-interval discretization of continuous-valued attributes for classification learning," in *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, 1993, pp. 1022–1027.

[24] Ying Yang and Geoffrey I. Webb, "A comparative study of discretization methods for naive-bayes classifiers," in *Proceedings of PKAW 2002: The 2002 Pacific Rim Knowledge Acquisition Workshop*, 2002, pp. 159–173.

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

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

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

[28] Tim Menzies, Justin S. DiStefeno, Mike Chapman, and Kenneth Mcgill, "Metrics that matter," in *27th NASA SEL workshop on Software Engineering*, 2002, Available from `http://menzies.us/pdf/02metrics.pdf`.

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

[30] Tim Menzies and Justin S. Di 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`.

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

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

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

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

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

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

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

[38] Ron Kohavi and George H. John, "Wrappers for feature subset selection," *Artificial Intelligence*, vol. 97, no. 1-2, pp. 273–324, 1997.

[39] T. Menzies, K. Ammar, A. Nikora, and Justin Di Stefano, "How simple is software defect detection?," in *Tech report, Computer Science, Portland State University*, 2004, Available from `http://menzies.us/pdf/03simpled.pdf`.

[40] Tim Menzies, David Raffo, Siri on Setamanit, Ying Hu, and Sina Tootoonian, "Model-based tests of truisms," in *Proceedings of IEEE ASE 2002*, 2002, Available from `http://menzies.us/pdf/02truisms.pdf`.

[41] C.L. Blake and C.J. Merz, "UCI repository of machine learning databases," 1998, URL: `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

[42] M. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, 1976.

[43] F. Shull, I. Rus, and V.R. Basili, "How perspective-based reading can improve requirements inspections," *IEEE Computer*, vol. 33, no. 7, pp. 73–79, 2000, Available from `http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/82.77.pdf`.