Assessing Predictors of Software Defects

Tim Menzies[†§‡], Justin DiStefano[‡§], Andres Orrego[S], Robert (Mike) Chapman[‡]
[‡]Galaxy Global Corporation. USA
[†]Computer Science, Portland State University,USA
[§]Lane Department of Computer Science and Electrical Engineering, West Virginia University, USA
tim@menzies.us | justin@lostportal.net | Andres.S.Orrego@ivv.nasa.gov | Robert.M.Chapman@ivv.nasa.gov

OVERVIEW: When learning defect detectors from static code measures, NaiveBayes learners are better than entrophy-based decision-tree learners. Also, accuracy is not a useful way to assess those detectors. Further, those learners need no more than 200-300 examples to learn adequate detectors, especially when the data has been *heavily stratified*; i.e. divided up into sub-sub-sub systems (and by "adequate", we mean that those detectors perform nearly as well slower, more expensive manual inspections).

The rest of this paper describes how we reached those conclusions after (i) an analysis of known baselines in the literature and (ii) a review of a assessment methods for detectors learned from the NASA defect logs of Figure 1.

BASELINES: If defect detectors are interesting, they must somehow be better than known *baselines* in the literature. For example, consider manual code reviews. These reviews are labor intensive; depending on the methods, 8 to 20 LOC/minute can be inspected and this effort repeats for all members of the review team, which can be as large as four or six [5]. These reviews can also be effective. A recent panel at *IEEE Metrics 2002* [6] concluded that such reviews can find $\approx 60\%$ of defects[1]. That defect detection rate has a wide variance. Raffo found that the defect detection capability of industrial inspection methods can vary from $TR(35, 50, 65)\%$[2] for full Fagan inspections, to $TR(13, 21, 30)$ for some widely-used industrial practices.

PUBLIC DOMAIN PROBLEMS: The data used in this study comes from the CM1, JM1, PC1, KC1 and KC2 NASA applications shown in Figure 1. This data contains static code measures (Halstead, Mccabe, SLOC) and defect rates (which we convert to booleans: $defects \in \{true, false\}$). This code is divided into levels for systems, sub-systems, sub-sub-systems, etc; for example, CM1 is a level1 system and KC2 is is a level4 sub-sub-sub-system.

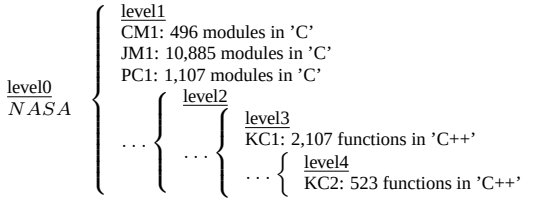

Figure 1. Data sets used in this study.

Ideally, this data set should be larger than the five items shown in Figure 1. Nevertheless, these five data sets are a far larger corpus than seen in most studies[3]

ASSESSMENT MEASURES: Various assessment measures exist for data miners including readability (neural networks cannot succinctly report their theories in a human-readable form); repeatability (genetic algorithms can return different theories after different runs); just to name a few. This study will use the assessment measures shown in Figure 2. This figure defines, amongst other things, the probability of detection PD as the ratio of known defects found divided by all known defects.

META-ASSESSMENT: If the goal of learning is to generate models that have some useful future validity, then the learned theory should be tested on data not used to build it[4]. One standard meta-assessment method a N-way cross

[1] That panel supported neither Fagan claim [3] that inspections can find 95% of defects before testing or Shull's claim that specialized directed inspection methods can catch 35% more defects that other methods [7]

[2] $TR(a, b, c)$ is a triangular distribution with min/mode/max of a, b, c.

[3] e.g. A recent report on software defect detectors in *IEEE Transactions on Software Engineering* used just 2 data sets [4].

[4] Failing to do so can result in a 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 [8]. 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.

To appear, Workshop on Predictive Software Models http://www.site.uottawa.ca/psm2004; Chicago, USA Co-located with ICSM 2004. WP: 04/psm/psm.tex; Available on the web at http://menzies.us/pdf/04psm.pdf.
This research funded was under contract NCC2-0979 and NCC5-685, sponsored by the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program led by the NASA IV&V Facility.

		module in defect log? 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

$$Acc = accuracy = \frac{A+D}{A+B+C+D} = 83\%$$
$$PF = falsealarm = \frac{C}{A+C} = 5\%$$
$$PD(a.k.a.recall) = detected = \frac{D}{B+D} = 37\%$$
$$prec = Precision = \frac{D}{C+D} = 67\%$$

Figure 2. Assessment statistics the binary classifier $v(g) \geq 10$. Each cell {A,B,C,D} shows the number of modules that fall into each cell.

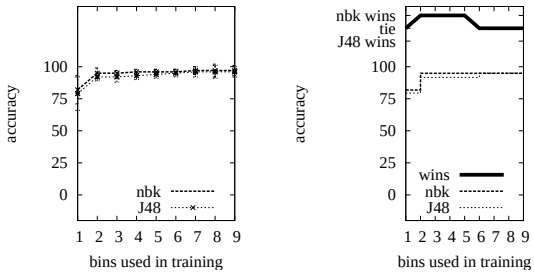

Figure 4. Learning curves (left); summarized (right).

M times the data ordering is randomized and then:

- The first L examples are divided into N buckets. The class distribution within the N buckets 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 a
- The remaining $\frac{N-X}{N} * L$ of the data is used for testing.

The sequence stops at $N-1$ since training on $\frac{N}{N}$ of the data would leave nothing for the test suite ($1 - \frac{N}{N} = 0$).

Figure 3. An LMNX study.

validation study where, the data is sorted into a random order M times. For each order, the data is divided into N buckets and the theory learned on N-1 buckets is tested on the remaining bucket. This procedure allows a prediction on how well the learner will perform on new data.

LMNX STUDIES: In a commercial setting, accessing data is difficult. Data miners need to know how *little* data they need to achieve good results. Hence, we modify the M*N procedure to explore how the learner's performance changes as *less and less data* is available.

Figure 3 defines an L*M*N*X study. In summary, the procedure picks up to L items from a data set (selected at random), divides those items into N buckets, then records the performance of a learner as an increasing number of the buckets is used for training. This process is repeated M-times. The "L" in the LMNX 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 1 which has 10885 examples).

Figure 4 shows the results from a LMNX study where *<L=150,M=N=10,X=1...9>* on the IRIS data set from the UCI repository [1]. The plot on the left shows the mean classifier accuracy improving as more and more of the data is used to train two classifiers from the WEKA toolkit [9] (J48 and a Naive Bayes classifier with kernel estimation).

The error bars in Figure 4 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 *LMNX summary plot* of Figure 4. In a LMNX 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 a LMNX summary plot is 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; and J48 never won; and both learners tied four times.

In an LMNX study, a learner is said to have *plateaued* when the accuracy/PD/PF/precision curves in the summary plot stop changing; i.e. there are no significant changes to the performance of the learner. The plateaus indicate when enough data collection is enough. In Figure 4, the learners plateaus after 40%*150= 60 examples

LMNX STUDIES ON DEFECT DATA: Figure 5 shows a LMNX study for the Figure 1 data comparing a NaiveBayes (with kernel estimation) classifier to an entropy-based decision tree learner (J48) [9]. In that study, *<L=500,M=N=10, X=1...9>*. Figure 5 shows 180 experiments where, 10 times, NaiveBayes and J48 were trained and tested on the same data. At the 95% level, NaiveBayes won 61 times, J48 won 26 times, and in the remaning 180-61-26=93 times, the two methods tied. This means that in $\frac{180-26=154}{180} \approx 85\%$ of cases, NaiveBayes would result in equivalent or better defect detectors than J48.

Figure 5 is sorted in according to PD. 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 and precisions. Note that this sort order

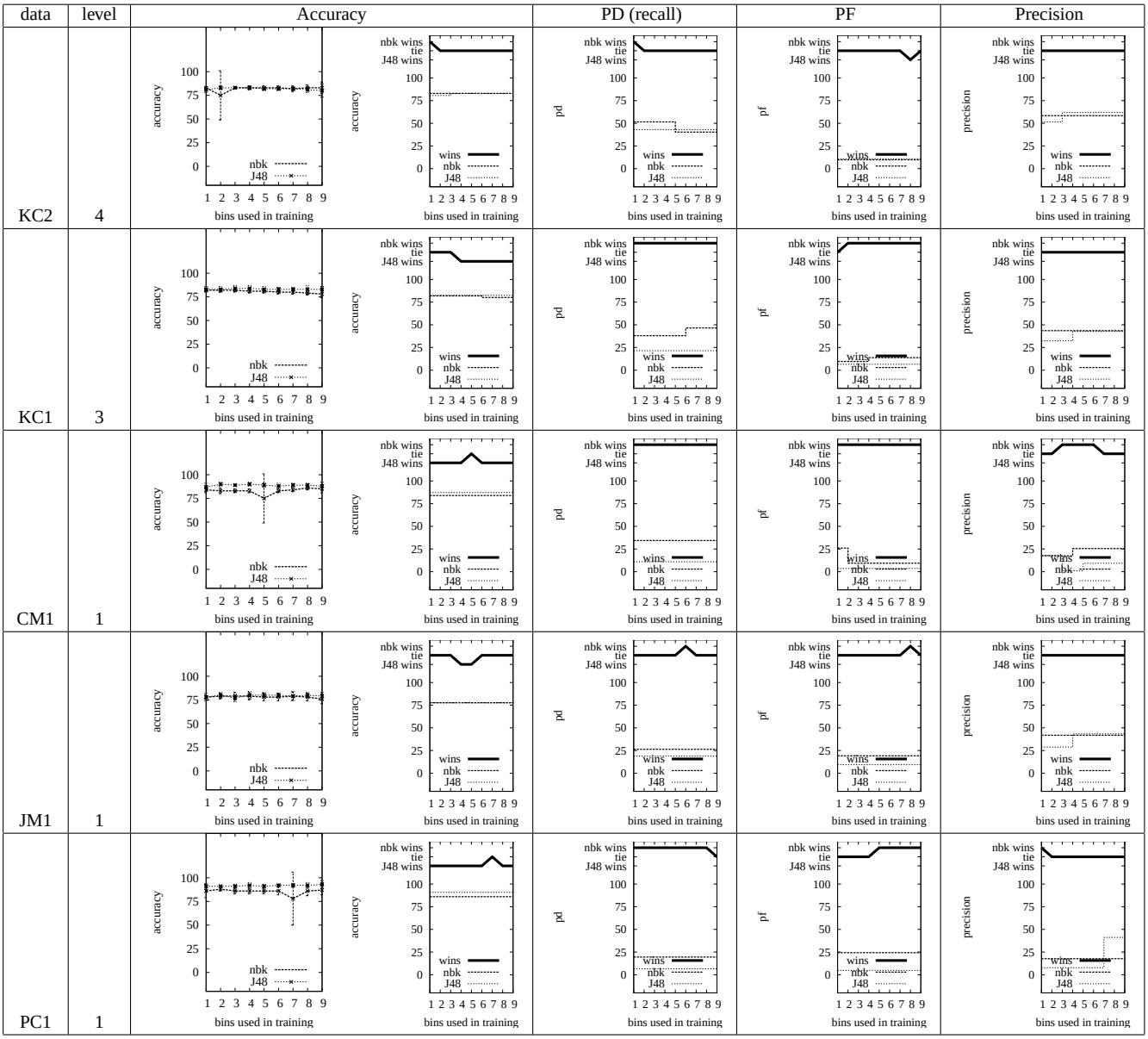

Figure 5. Accuracy, PD, PF, precision results from a LMNX study for Figure 1 data.

corresponds to the system/sub-system break up shown in Figure 1. That is, defect detectors learnt from specialized sub-sub-sub-systems have a higher PD and precision than defect detectors learnt for an entire system. A similar effect has been seen previously in the software cost estimation literature: better results are obtained from data that has been *stratified* into the specialized groups [2, p179]..

PROBLEMS USING "ACCURACY": Data miners usually report their value in terms of *average accuracy* seen in a M*N study and mere accuracy measures can miss numerous important effects. In Figure 5, accuracy is (nearly) the same in all datasets even though other measures varied wildly; e.g CM1 and KC2 have similar accuracies but KC2's PD are much higher than CM1's PD.

Another drawback with accuracy is that these authors can't find in the literature baseline accuracy measurements for software defect detectors. On the other hand, the introduction to this paper reported PD's for manual code inspection from the literature ranging from 13% to 30% (for simple inspections) to 35% to 65% (for more elaborate inspection methods). The PDs shown in Figure 5 range up to to 55% (in KC1) in the more stratified data sets. To the best

of our knowledge, Figure 5 is the first report in the literature that static code detectors learnt from highly stratified data work as well as middle to high-effort manual inspection methods. This is a significant result since automatically data mining detectors are far cheaper and faster than manual inspections that requires up to half a dozen participants working for many days.

EARLY PLATEAUS: In $\frac{15}{20}^{ths}$ of the Figure 5 plots, there is no significant improvement after learning from 40%*500=400 examples. Further, in $\frac{19}{20}^{ths}$ of the plots in Figure 5, there is no improvement after 60%*500=300 examples[5] Such an *early plateau* effect can have several benefits.

If a learner plateaus early, then standard data mining methods can scale to very large data sets. Our SAWTOOTH learner (under development) finds the number of examples N needed before plateau. SAWTOOTH then changes to *cruise mode* where learning is disabled and the system runs down the rest of the data, testing the new examples on the theory learned before entering cruise mode, all the while maintaining a cache of the last N examples. If the test performance statistics ever significantly change, then the data has fallen off the plateau and more learning is required. SAWTOOTH then passes the N examples to a some learner in order to find the next plateau. The process repeats till the data is exhausted. Note that, at all times, SAWTOOTH only needs the memory required to handle N examples.

SAWTOOTH can mine large data sets and handle *concept drift*. When collecting data, sometimes the data mining team is unaware that the underlying distributions have changed (e.g. when there are major personnel changes amongst the programmers). These distribution changes can imply that the learned theory needs to be changed.

In domains with early plateaus, SAWTOOTH can handle concept drift. Suppose a domain generates I instances per time T at the rate $\frac{I}{T}$. Suppose further that learners in that domain stabilize after S instances which takes time $\frac{S}{\frac{I}{T}}$ to collect. If concept drift takes *longer* than stabilization time, then concept drift can be detected by dividing the data chronologically into units of size S and testing $unit_j$ using the theory learned from $unit_{j-1}$. If the domain is stable, then the *same* performance should be seen in $unit_j$ as $unit_{j-1}$. If concept drift has occurred then the performance will be different.

CONCLUSIONS: Based on a LMNX analysis of the Figure 1 data, we say that (1) measuring detector performance in terms of accuracy can be uninformative; (2) NaiveBayes is better than J48 at finding defect detectors; (3) learners need only 200-300 examples to find defect detectors; (4) stratification into sub-sub-systems (and below) improves the probability of detection and precision; and for stratified datasets, (5) defect detectors performing nearly as well slower, more expensive manual inspections

FUTURE WORK: The generality of our conclusions should be checked. We hence recommend the establishment of a public domain database of data mining examples for defect detection. These data sets should include stratification information. For our part, the Figure 1 data is available from NASA's Metrics Data Program web site[6] in normalized table data format, or from the first author's web site[7], in ARFF format.

References

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

[2] B. Boehm, E. Horowitz, R. Madachy, D. Reifer, B. K. Clark, B. Steece, A. W. Brown, S. Chulani, and C. Abts. *Software Cost Estimation with Cocomo II*. Prentice Hall, 2000.

[3] M. Fagan. Advances in software inspections. *IEEE Trans. on Software Engineering*, pages 744–751, July 1986.

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

[5] T. Menzies, D. Raffo, S. on Setamanit, Y. Hu, and S. Tootoonian. Model-based tests of truisms. In *Proceedings of IEEE ASE 2002*, 2002. Available from `http://menzies.us/pdf/02truisms.pdf`.

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

[7] F. Shull, I. Rus, and V. Basili. How perspective-based reading can improve requirements inspections. *IEEE Computer*, 33(7):73–79, 2000. Available from `http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/82.77.pdf`.

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

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

[5] J48's precision results jumps 30% in PC1- but even after this jump, J48's mean is not significantly different to the NaiveBayes precision results for PC1.

[6] `http://mdp.ivv.nasa.gov`

[7] `http://scant.org/2/eg/arff/defects*`