Baselines in Software Defect Detection

Tim Menzies[*]
Computer Science
Portland State University.
tim@menzies.us

ABSTRACT

The goal of the PROMISE workshop is to create a set of reproducible software engineering experiments with the hope that researchers will repeat and improve on prior results. To show that something has improved, there must *first* be some defined baseline. This paper lists the baselines known for data mining results of defect detectors from static code measures and defect logs found in the PROMISE data repository [6]. Many of these results have appeared previously [2] but the *stratification study* at the end of this paper is a new result.

1. EVALUATION CRITERIA

A defect detector hunts for a *signal* that a software module is defect prone. In Figure 1, if a detector senses a signal, sometimes the signal is present (cell D) and sometimes it is not (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:

$$\text{probability detection (a.k.a. recall)} = PD = \frac{D}{B+D}$$

In the other case, the *probability of a false alarm*, or "*PF*", is the percentage of detections when no signal was present:

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

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:

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

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

Figure 1: Statistics.

For example, the numbers in the ABCD cells of Figure 1 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 [4], 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 in the modules falling into each cell of Figure 1 is LOC_A, LOC_B, LOC_C, and LOC_D, then:

$$Inspection\ Effort = \frac{LOC_C + LOC_D}{LOC_A + LOC_B + LOC_C + LOC_D} \quad (1)$$

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 1, these represent an $Effort$ of 56% of the code.

2. TRADE-OFFS WITHIN THE CRITERIA

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 2 has been observed in hundreds of defect detectors generated from various subsets of the available static measures from the PROMISE defect 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" [3, 5]. 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. In that figure:

[*] See http://menzies.us/pdf/05baselinedefects.pdf for a draft of this paper.

PROMISE '05 St. Louis, Missouri USA

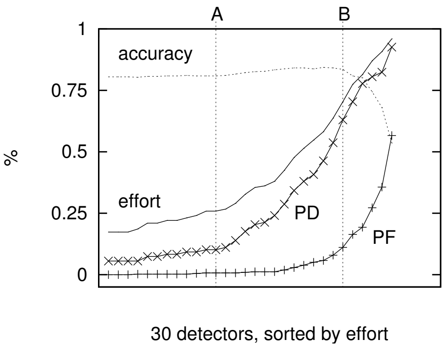

Figure 2: At each x-axis value, x describes the $pf, pd, effort, accuracy$ of one detector. From [3].

- Not surprisingly, to find *more* faults, we need to 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 for predicting detection, false alarms, and effort. Consider the detectors marked A and B on Figure 2. 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.

3. VALIDATION STUDIES

If the goal is to generate detectors that have some useful future validity, then we should test the detector 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 [7]. 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 [8]. 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.

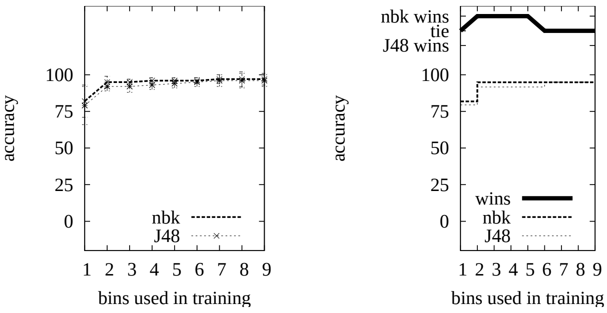

Figure 3: Results (left); summarized (right).

- 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 discussed below has 10885 modules).

Figure 3 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 [1]. 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 3 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 3. 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 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 *plateaus* at $X = \alpha$ if there is are no significant changes to the performance of the learner after training on α examples. In Figure 3, 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

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					

Figure 4: Data sets used in this study. ***File*** **denotes a data set file name available in** ***ARFF*** **format from** **`http://promise.site.uottawa.ca/SERepository/datasets-page.html`.**

to the other learner. In Figure 3 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 3 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.

3.1 Sequence Studies on PROMISE data sets

Figure 5 shows the results for a *<L=500,M=N=10, X=1. . . 9>*. sequence study on the Figure 4 data using a NaiveBayes (with kernel estimation) classifier to an entropy-based decision tree learner (J48) [8][1] 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 2 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 Figure 5, 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 2 and Figure 5 results, we hence *advise against using accuracy to assess defect detectors.*

Figure 5 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 $\frac{90-1=89}{90}$≈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 5 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.*

In Figure 5, 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[2]. These PDs are reached at a much lower cost than what is required for the equivalent manual task. Menzies and Raffo [4] 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[3], 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.*

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, thereby incurring the *effort* cost described above.

At the plateau point, the height of the plateau seems to be determined by how *stratified* is the data set. Figure 5 is sorted in according to maximum reached PD and this order

[1] This study used the J48 and NaiveBayes implementations that come with the WEKA toolkit.

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

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

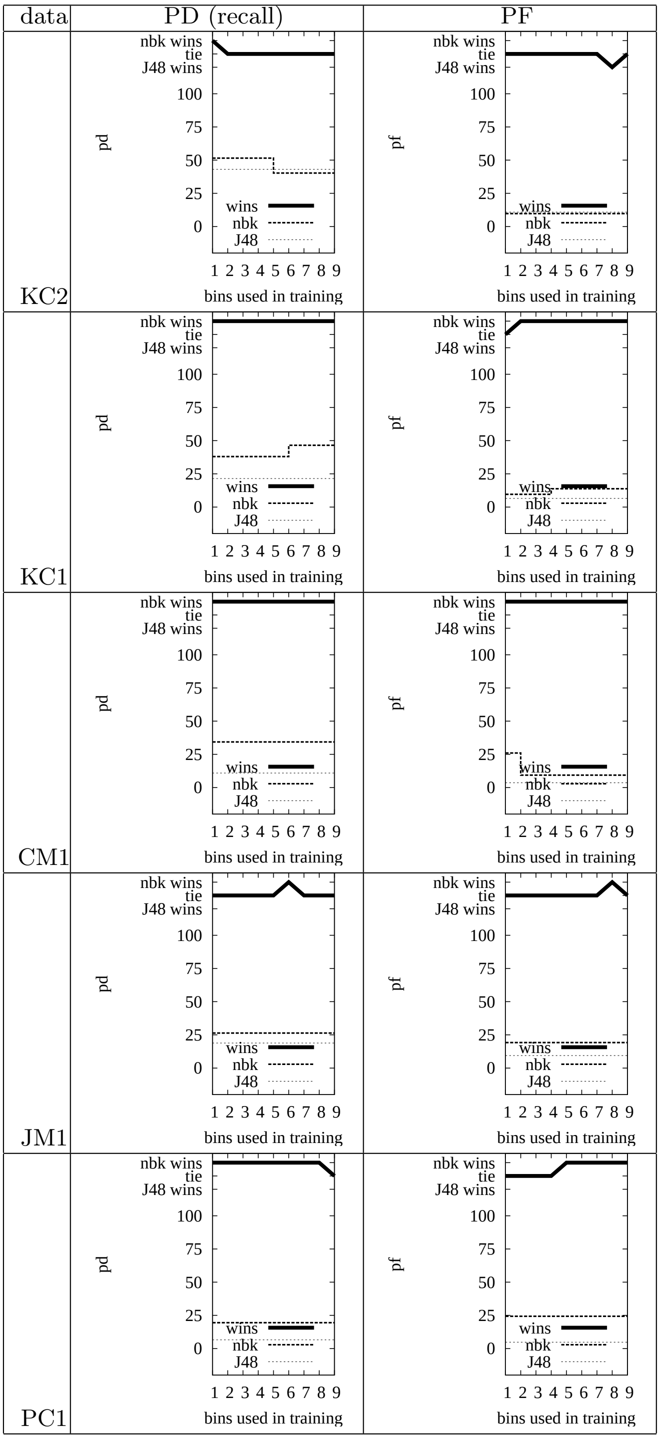

Figure 5: PD and PF results from a sequence study for Figure 6 data.

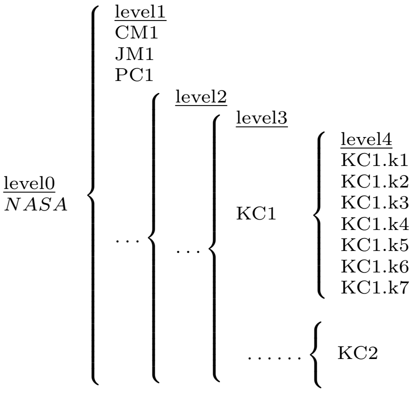

Figure 6: Data sets used in this study.

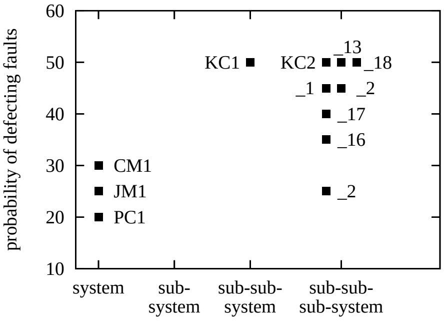

Figure 7: Naive Bayes PD plateaus.

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. This sort order corresponds to the directory structure where the source code was stored (see Figure 6). These datasets come from NASA repositories and standard practice at NASA is to use directories to represent system/sub-systems/sub-sub-systems (etc). Note that 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.

4. STRATIFICATION

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, Naive-Bayes won/lost/ties 16/6/32 times against J4.8.
- *Detectors usually exhibit very early plateau.* In all sub-divisions of KC1, plateau was reached at 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 7. 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, . . . , _18 etc. With the exceptions of _2 and _16, sub-division improves PD.

5. SUMMARY

Using public domain datasets and public domain learners, this paper has defined a set of baseline results for learning defect detectors. The challenge to the reader is now to repeat and improve (or refute) these results.

Acknowledgments

This research was conducted at 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.

6. REFERENCES

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

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

[3] 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`.

[4] 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`.

[5] 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`.

[6] J. Sayyad Shirabad and T.J. Menzies. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005. Available from `http://promise.site.uottawa.ca/SERepository`.

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

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