Overcoming Ceiling Effects in Defect Prediction

Tim Menzies, Member, IEEE, Zach Milton, Burak Turhan, Yue Jiang, Gregory Gay, Bojan Cukic, Ayşe Bener

Abstract

While data miners can learn defect predictors from static code features, the performance improvement in such detectors is curiously static. One explanation for this ceiling effect is that static code features have limited information content. If so, then it should be useful to give our learners more knowledge about the problems they are processing. Data miners build predictors using a performance criteria $P$ (e.g. accuracy), then assess them using another criteria $Q$ (e.g. probability of detection). Typically, $P \neq Q$, so learners are blind to their purpose. Therefore, it is hardly surprising that they cannot find ways to better their performance. WHICH is a new data miner with $P \approx Q$; i.e. a similar evaluation criteria is applied during training and testing. After matching $P$ to the goals of a particular business application, we found that commonly used data miners perform no better than simple manual methods; and that WHICH outperforms other methods, coming close to a theoretical upper bound in performance (50 and 75 percentile of 70.9% and 80%, respectively). That is, with knowledge of the business application, it is possible to build new data miners that overcome ceiling effects.

Index Terms

product metrics, defect prediction, data mining, decision-tree learning, rule-learning, C4.5, RIPPER.
Manuscript received April 1, 2008; revised XXX.

Dr. Menzies, Dr. Cukic, Mr. Milton, Mr. Gay, and Ms. Jiang are with the Lane Department of Computer Science and Electrical Engineering, West Virginia University, USA: tim@menzies.us, bojan.cukic@mail.csee.wvu.edu; zmilton,yjiang1@mix.wvu.edu; greg@4colorrebellion.com

Dr. Bener and Mr. Turhan are with the Dept. of Computer Engineering, Boğaziçi University, Turkey: bener,turhanb@boun.edu.tr

For enquiries on this work, email tim@menzies.us. For an earlier draft, see http://menzies.us/pdf/08ceffects.pdf.

The research described in this paper was supported by Boğaziçi University research fund under grant number BAP-06HA104 and at West Virginia University under grants with NASAs Software Assurance Research Program. Reference herein to any specific commercial product, process, or service by trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government.

April 30, 2008
I. INTRODUCTION

For several years, the authors have relied upon straightforward application of standard data mining algorithms to learn defect predictors from static code features; see [1]–[10] Recent results, discussed in the paper, now suggest that there has been too much emphasis on machine learning, and too little attention on the use of specific software engineering knowledge.

Automatically generated detect predictors are demonstrably useful and, as described in §2, can operate better than current industrial best practices [9], [11]. However, we have observed that these AI methods have hit a performance ceiling; i.e., some inherent upper bound on the amount of information standard data miners can extract from, say, static code features. For example, after a careful study of 19 data miners for learning defect predictors, Lessmann et.al. [12] conclude

> the importance of the classification model (our emphasis) is less than generally assumed and that practitioners are free to choose from a broad set of candidate models when building defect predictors.

Note that, if all learners yield similar results, then the value of further research into automatic defect predictors is questionable. However, if the performance goal is changed from classification to some application-aware criteria then, contrary to Lessmann’s advice, the selection of learner becomes critical. For example, this paper examines a specific business context we call application\(_1\): inspect the fewest lines of code; find the most number of defective modules. In the context of application\(_1\), we show that several standard learners are demonstrably inferior to simple manual methods. The same cannot be said of our new learner, called WHICH [13], that performs better than both standard learners and manual methods. In fact, WHICH’s performance also comes close to a theoretical upper bound on the performance of any learner tackling application\(_1\).

This paper is structured as follows. After a literature review on defect predictors, we document the ceiling effect mentioned above and hypothesize that it is due to limited information content in static code attributes. Two data reduction experiments (that shrink the size of training sets) will confirm this hypothesis and show that the performance of our defect predictors stabilizes after just a handful of training examples\(^1\). If limited information is the problem, the solution is clear: give our learners more information. The details of application\(_1\) will be used to design an application-aware performance criteria for WHICH. An experiment is performed where WHICH

\(^1\)Note that the data reduction experiments of §4 have been reported previously [14]. However, the rest of this paper, including the definition and exploration of the WHICH learner, is all new work.
and three standard data miners are applied to application. WHICH will be shown to perform better than manual methods and standard data miners. We hypothesize that WHICH’s superior performance comes from detailed knowledge of the necessary business context. Unlike other learners, WHICH applies that knowledge at all levels of its reasoning.

Our conclusion will be that knowledge of the business application can overcome ceiling effects. We hope that this result prompts a new cycle of defect prediction research focused on discovering the best learner(s) for particular business context(s).

II. ABOUT DEFECT PREDICTION

Data miners learn defect predictors from static code features, either from projects previously developed in the same environment or from a continually expanding base of the current project’s artifacts. To do so, tables of examples are formed where one column has a boolean value for “defects detected” and the other columns describe software features such as lines of code, number of unique symbols [15], or max. number of possible execution pathways [16]. Each row in the table holds data from one “module”, the smallest unit of functionality. Depending on the language, these may be called “functions”, “methods”, or “procedures”. Static code features are described in Figures 1,2,3.

The data mining task is to find combinations of features that predict for the value in the defects column. The value of static code features as defect predictors has been widely debated. While some researchers vehemently oppose them [17], [18], many others endorse their use [6], [9], [10], [15], [16], [19]–[35]. Standard verification and validation (V&V) textbooks [36] advise using static code complexity attributes to decide which modules are worthy of manual inspections. For several years, the authors have worked on-site at the NASA Independent software Verification and Validation facility where large government software contractors won’t review software modules unless tools like the McCabe static source code analyzer predict that they exhibit high code complexity measures.

Nevertheless, static code attributes can never be a full characterization of a program module. Fenton offers an insightful example where the same functionality is achieved using different programming language constructs resulting in different static measurements for that module [38]. Fenton used this example to argue the uselessness of static code attributes for fault prediction.

If Fenton is right, then the performance of predictors learned by data mining static code features should be poor. However, this is not true, at least for the code we have studied [2], [4], [6], [7], [9], [10], [27], [29], [30], [39]. Using NASA data, our fault prediction models find
\[ m = \text{McCabe} \]

\[ v(g) \quad \text{cyclomatic complexity} \]

\[ iv(G) \quad \text{design complexity} \]

\[ ev(G) \quad \text{essential complexity} \]

<table>
<thead>
<tr>
<th>locs</th>
<th>loc</th>
<th>loc_total (one line = one count)</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc(\text{other})</td>
<td>loc_blank</td>
<td></td>
</tr>
<tr>
<td></td>
<td>loc_code_and_comment</td>
<td></td>
</tr>
<tr>
<td></td>
<td>loc_comments</td>
<td></td>
</tr>
<tr>
<td></td>
<td>loc_executable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>number_of_lines (opening to closing brackets)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Halstead</th>
<th>( h )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N_1 )</td>
<td>num_operands</td>
</tr>
<tr>
<td>( N_2 )</td>
<td>num_operands</td>
</tr>
<tr>
<td>( \mu_1 )</td>
<td>num_unique_operators</td>
</tr>
<tr>
<td>( \mu_2 )</td>
<td>num_unique_operands</td>
</tr>
</tbody>
</table>

\[ H \]

- \( N \) length: \( N = N_1 + N_2 \)
- \( V \) volume: \( V = N \cdot \log_2 \mu \)
- \( L \) level: \( L = V^*/V \) where \( V^* = (2 + \mu_2) \log_2 (2 + \mu_2) \)
- \( D \) difficulty: \( D = 1/L \)
- \( I \) content: \( I = \hat{L} \cdot V \) where \( \hat{L} = \frac{2}{\mu_1} + \frac{2}{N_2} \)
- \( E \) effort: \( E = V/L \)
- \( B \) error_est
- \( T \) prog_time: \( T = E/18 \) seconds

Fig. 1. Features used in this study. The Halstead features are explained in Figure 2 and the McCabe features are explained in Figure 3.

The Halstead features were derived by Maurice Halstead in 1977. He argued that modules that are hard to read are more likely to be fault prone [15]. Halstead estimates reading complexity by counting the number of operators and operands in a module; see the \( h \) features of Figure 1. These three raw \( h \) Halstead features were then used to compute the \( H \): the eight derived Halstead features using the equations shown in Figure 1. In between the raw and derived Halstead features are certain intermediaries:

- \( \mu = \mu_1 + \mu_2 \);
- minimum operator count: \( \mu_1 = 2 \);
- \( \mu_2 \) is the minimum operand count (number of module parameters).

Fig. 2. Notes on the Halstead features

defect predictors [9] with a probability of detection \((pd)\) and probability of false alarm \((pf)\) of \(\text{mean}(pd, pf) = (71\%, 25\%)\).

These values can be compared to baselines in data mining and industrial practice. Raffo (personnel communication) found that industrial reviews discover \( pd = TR(35, 50, 65)\%^2 \) of a systems errors’ (for full Fagan inspections [40]) to \( pd = TR(13, 21, 30)\% \) for less-structured inspections. Similar values were reported at an IEEE Metrics 2002 panel. That panel declined to endorse claims by Fagan [41] and Schull [42] regarding the efficacy of their inspection or

\(^2TR(a, b, c)\) is a triangular distribution with min/mode/max of \( a, b, c \).
An alternative to the Halstead features of Figure 2 are the complexity features proposed by Thomas McCabe in 1976. Unlike Halstead, McCabe argued that the complexity of pathways between module symbols are more insightful than just a count of the symbols [16]. The first three lines of Figure 1 shows McCabe three main features for this pathway complexity. These are defined as follows.

- A module is said to have a flow graph; i.e. a directed graph where each node corresponds to a program statement, and each arc indicates the flow of control from one statement to another.
- The cyclomatic complexity of a module is \( v(G) = e - n + 2 \) where \( G \) is a program’s flow graph, \( e \) is the number of arcs in the flow graph, and \( n \) is the number of nodes in the flow graph [37].
- The essential complexity, \( ev(G) \) of a module is the extent to which a flow graph can be “reduced” by decomposing all the subflowgraphs of \( G \) that are D-structured primes (also sometimes referred to as “proper one-entry one-exit subflowgraphs” [37]). \( ev(G) = v(G) - m \) where \( m \) is the number of subflowgraphs of \( G \) that are D-structured primes [37].
- Finally, the design complexity \( iv(G) \) of a module is the cyclomatic complexity of a module’s reduced flow graph.

Fig. 3. Notes on the McCabe features

directed inspection methods. Rather, it concluded that manual software reviews can find \( \approx60\% \) of defects [43].

As to comparing our defect predictors with standard results from the data mining community, in prior work, we have checked the efficacy of data mining on standard machine learning data sets such as the UCI data repository [44]. For each UCI data set, ten experiments were conducted, using a state-of-the-art decision tree learner [45] from 90\% of the data, selected at random. The experiments tested the learned decision tree on remaining 10\% of the data. On average, state of the art data miners perform at \((pd,pf) = (81\%,20\%)\). This is close to the results we have obtained via data mining on static code attributes \((pd,pf) = (71\%,25\%)\). Note that if static code attributes capture so little about source code (as argued by Fenton), then we would expect lower probabilities of detection and much higher false alarm rates.

Overall, there are two reasons to recommend static code predictors. Firstly, our \((pd, pf)\) results are better than currently used industrial methods such as:

- the \(pd\approx60\%\) reported at the 2002 IEEE Metrics panel or
- the median \((pd) = 21.50\) reported by Raffo.

Secondly, static code defect predictors can be built quickly, even for very large systems [31]. Other methods such as manual code reviews may be more labor-intensive. Depending on the review method, 8 to 20 lines of code (LOC) per minute can be inspected. This effort repeats for all members of the review team (typically, four or six [46]). Our defect detectors, on the other hand, can be generated using automatic methods.
If \{A, B, C, D\} are the true negatives, false negatives, false positives, and true positives (respectively) found by a defect predictor, then:

\[ \begin{align*}
pd &= \text{recall} = \frac{D}{B + D} \\
pf &= \text{precision} = \frac{C}{A + C} \\
bal &= \text{balance} = 1 - \sqrt{\frac{(0 - pf)^2 + (1 - pd)^2}{2}}
\end{align*} \]

All these values range zero to one. Better and larger balances fall closer to the desired zone of no false alarms and 100% detection.

Other measures such as accuracy and precision were not used since, as shown in Figure 7, the percent of defective examples in our tables was usually very small (median value around 8%). Accuracy and precision are poor indicators of performance for data were the target class is so rare (for more on this issue, see [9], [10]).

Fig. 4. Performance measures.

### III. Ceiling Effects

A “ceiling effect” may be observed when different treatments all yield similar upper-bounds on their performance. As documented in this section, there is much evidence for a ceiling effect in defect prediction. Specifically, even after years of exploring different learners and data preprocessing methods, the performance of our learners has not improved.

In January 2007, we published in TSE [9] a study that defined a repeatable experiment in learning defect predictors. The aim of that work was a benchmark result that other researchers could repeat/improve/refute. That experiment used public domain data sets\(^3\) and open source data mining tools (the WEKA toolkit [47]); Data order was randomly sorted (to stop order effects). Data mining was performed using 10-way cross-validation (to test on data not used in training). Learner assessment was via multiple criteria such as probability of detection (\(pd\)), probability of false alarm (\(pf\)), and balance that combines \{pd, pf\} (balance is defined in Figure 4). The experiment also included statistical hypothesis tests over the assessment criteria; novel visualization methods for the results; feature subset selection to find important subsets of features; and learning via multiple types of machine learning algorithms: rule learners, decision tree learners, naïve Bayes classifiers.

Surprisingly, naïve Bayes classifiers (with a simple pre-processor for the numerics) outperformed the other studied methods. For details on naïve Bayes classifiers, see Appendix I.

Since that study, we have tried to find better data mining algorithms for defect prediction. To date, we have failed. Our recent experiments [48] have found little or no improvement from

\(^3\)From http://promisedata.org
Fig. 5. Box plot for area under PD-vs-PF curves seen with 9 learners when, 100 times, a random 90% selection of the data is used for training and the remaining data is used for testing. The rectangles show the inter-quartile range (the 25% to 75% quartile range). The line shows the minimum to maximum range, unless that range extends beyond 1.5 times the inter-quartile range (in which case dots are used to mark these extreme outliers). From [48].

Fig. 6. Range of “ranks” seen in 19 learners building defect predictors when, 10 times, a random 66% selection of the data is used for training and the remaining data is used for testing. In ranked data, values from one method are replaced by their rank in space of all sorted values (so smaller ranks means better performance). In this case, the performance value was area under the false positive vs true-positive curve (and larger values are better). Vertical lines divide the results into regions where the results are statically similar. For example, all the methods whose top ranks are 4 to 12 are statistically insignificantly different. From [12].
the application of numerous data mining methods. Figure 5 shows some of those results using (in order, left to right) aode average one-dependence estimators [49]; bag bagging [50]; bst boosting [51]; IBk instance-based learning [52]; j48 j48 [45]; jrip RIPPER [53]; lgi logistic regression [54]; nb naïve Bayes (second from the right); and rf random forests [55]. A statistical analysis shows that only boosting on discretized data offers a statistically better result than naïve Bayes. However, we cannot recommend boosting:

- Boosting is orders of magnitudes slower than naïve Bayes;
- The median improvement over naïve Bayes is negligible.

Other researchers have also failed to improve our results. For example, Figure 6 shows results from a study by Lessmann et al. on statistical differences between 19 learners used for defect prediction [12]. At first glance, our preferred naïve Bayes method (shown as “NB” on the sixth line of Figure 6) seems to perform poorly; it is ranked in the lower third of all 19 methods. However, as with all statistical analysis, it is important to examine not only central tendencies but also the variance in the performance measure. The vertical dotted lines in Figure 6 show Lessmann et al.’s statistical analysis that divided the results into regions where all the results are significantly different: the performance of the top 16 methods are statistically insignificantly different from each other (including our preferred “NB” method). Lessmann et al. comment:

“Only four competitors are significantly inferior to the overall winner (k-NN, K-start, BBF net, VP). The empirical data does not provide sufficient evidence to judge whether RndFor (Random Forest), performs significantly better than QDA (Quadratic Discriminant Analysis) or any classifier with better average rank.

In other words, Lessmann et al. are reporting a ceiling effect where a large number of learners exhibit performance results that are indistinguishable.

IV. THE “LIMITED INFORMATION CONTENT” HYPOTHESIS

How to explain all these failed attempts to improve fault prediction? One possibility is that these static code features have a limited information content. If so, then simple learning methods will uncover all that can be found. Further, more sophisticated data mining methods will yield no more information.

The rest of this section offers evidence for the limited information hypothesis.
A. Random Reduction Experiments

In our RandomReduction study, defect predictors were learned from $N = 100$, $N = 200$, $N = 300$, etc., instances (selected at random) then Tested on another 100 instances. For all experiments, the Train and Test instances were selected at random using a 10-way cross-validation study\(^4\) for each value of \(N\).

This RandomReduction study was conducted on the NASA sets of Figure 7 using a naïve Bayes classifier (since it performed so well in the above experiments). Space does not permit showing all the results but a representative sample is shown in Figure 8 [14]. In that figure, the X-axis is the size of training set and the Y-axis is the balance measure defined in Figure 4.

Note that the performance does not change much regardless of whether the model is inferred from 100 instances or from up to several thousand instances. In fact, learning from too many training examples may even be detrimental (witness the widening variance as the training set increases). A Mann Whitney U test [56] (95% confidence)\(^5\) confirms the visual pattern apparent in Figure 8: static code features used as the basis for predicting module’s fault content revealed all that they can reveal after as little as 100 instances.

B. Structured Reduction Experiments

The Figure 8 RandomReduction experiment randomly discarded training data. Perhaps a more StructuredReduction method would not damage the information content of the data? If so then, contrary to Figure 8, increasing the sample size will improve performance and break through the ceiling effect reported above.

Three examples of StructuredReduction are micro-, over-, and under-sampling [59], [60]. All of these methods build datasets with an equal number of defective and non-defective classes:

- In the case of under-sampling, random instances from the majority classification are removed. This results in a much smaller dataset, but the minority class is no longer buried

\(^4\)In 10-way cross-validation, 10 experiments are performed where training is conducted on \(|\text{Train}| = 90\% \times N\) instances, then tested on data not used during training (so \(\text{Train} \cap \text{Test} = \{\}\)).

\(^5\)In the rest of this paper, all statistical tests will be via the Mann-Whitney non-paired non-parametric test. Non-parametric tests are used since Demsar advises that parametric assumptions have conflated much prior data mining research [57]. Non-paired tests are used since, as in Figure 8, all the experiments from here onwards apply the same treatment to different populations. Mann-Whitney is used instead of, say, the Wilcoxon test [58] since (a) Wilcoxon is a paired test and (b) a single Mann-Whitney test can compare one learner \(L_1\) against rival learners \(L_2, L_3, \ldots\) (since Mann-Whitney does not require that all samples being compared have the same cardinality). Hence, Mann-Whitney supports very succinct summaries of the results without the post-processing required for Wilcoxon (see Demsar [57] or Lessmann et al. [12] for details on the Wilcoxon post-processing [12]).
<table>
<thead>
<tr>
<th>source</th>
<th>project</th>
<th>language</th>
<th>(# modules) examples</th>
<th>features</th>
<th>% defective</th>
</tr>
</thead>
<tbody>
<tr>
<td>NASA</td>
<td>pc1</td>
<td>C++</td>
<td>1,109</td>
<td>21</td>
<td>6.94</td>
</tr>
<tr>
<td>NASA</td>
<td>kc1</td>
<td>C++</td>
<td>845</td>
<td>21</td>
<td>15.45</td>
</tr>
<tr>
<td>NASA</td>
<td>kc2</td>
<td>C++</td>
<td>522</td>
<td>21</td>
<td>20.49</td>
</tr>
<tr>
<td>NASA</td>
<td>cm1</td>
<td>C++</td>
<td>498</td>
<td>21</td>
<td>9.83</td>
</tr>
<tr>
<td>NASA</td>
<td>kc3</td>
<td>JAVA</td>
<td>458</td>
<td>39</td>
<td>9.38</td>
</tr>
<tr>
<td>NASA</td>
<td>mw1</td>
<td>C++</td>
<td>403</td>
<td>37</td>
<td>7.69</td>
</tr>
<tr>
<td>SOFTLAB</td>
<td>ar4</td>
<td>C++</td>
<td>107</td>
<td>30</td>
<td>18.69</td>
</tr>
<tr>
<td>SOFTLAB</td>
<td>ar3</td>
<td>C++</td>
<td>63</td>
<td>30</td>
<td>12.70</td>
</tr>
<tr>
<td>NASA</td>
<td>mc2</td>
<td>C++</td>
<td>61</td>
<td>39</td>
<td>32.29</td>
</tr>
<tr>
<td>SOFTLAB</td>
<td>ar5</td>
<td>C++</td>
<td>36</td>
<td>30</td>
<td>22.22</td>
</tr>
</tbody>
</table>

Fig. 7. Tables of data, sorted in order of number of examples. The rows labeled “NASA” come from NASA aerospace projects while the rows labeled “SOFTLAB” come from a Turkish software company writing applications for domestic appliances. For details on the features used in each data set, see Figure 1.

Fig. 8. RandomReduction results. experimenting with training set size vs balance (and balance is defined in Figure 4).

inside a larger set of other classes.

- In over-sampling, randomly selected instances from the minority class are copied. Where as under-sampling produces smaller data sets, over-sampling grows the size of the data.
- Micro-sampling combines under-sampling with data reduction. Given \( N \) defective modules in a data set, \( M \in \{25, 50, 75, \ldots\} \leq N \) defective modules are selected at random. Another \( M \) non-defective modules are then selected, at random. Note that under-sampling is a micro-sampling where \( M = N \). Micro-sampling explores training sets of increasing size \( 2M \ldots2N \), while standard under-sampling just explores one data set of size \( 2N \).

Under-sampling, over-sampling, micro-sampling, and “no-sampling” were applied to the NASA data sets from Figure 7 using 10-way cross-validation. For the “no-sampling” experiments, the raw data was used for training and testing without any adjustment to the class frequencies.

In these studies, two data miners were used: the naïve Bayes classifier we recommended above and the j48 decision tree learner [45], [47]. j48 was used to test our rig against known over- and under- sampling results described in the literature [59], [61]. Other learners were not used since, as discussed above, all of our experiments with other learners have not be productive. For
Fig. 9. *StructuredReduction* results. Over- & under- & no sampling results. Sorted descending by median balance results (balance is defined in Figure 4). The right-hand side show median values (as a circle) within a 25% to 75% percentile range. The rank, shown left-hand-side, come from the statistical analysis of Figure 10. Three methods share top rank: NB/none, NB/under, j48/under.

<table>
<thead>
<tr>
<th>rank</th>
<th>treatment</th>
<th>median</th>
<th>2nd quartile, median and 3rd quartile</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NB/ none</td>
<td>74.6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>NB/ under</td>
<td>74.1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>j48/ under</td>
<td>73.6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>NB/ over</td>
<td>62.5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>j48/ over</td>
<td>45.6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>j48/ none</td>
<td>42.3</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 10. *StructuredReduction* results: statistical tests on the Figure 9 results. Column two lists six treatments. Each row shows how the results for one treatment compare to the other five. This table is sorted in ascending order on the number of losses (so better methods appear at the top of the table. The first column shows a comparison of one treatment against the other eight. Two treatments have the same rank if their median ranks are statistically insignificantly different (Mann-Whitney, 95% confidence).

<table>
<thead>
<tr>
<th>rank</th>
<th>treatment</th>
<th>win</th>
<th>loss</th>
<th>ties</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>nb / none</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>nb / under</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>j48/ under</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>nb / over</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>j48/ over</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>j48/ none</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

more details on j48, see Appendix II.

Figure 9 shows the under- and over- balance results (balance was defined in Figure 4). The pattern of results is very clear:

- Over-sampling did not improve classifier performance. This result is consistent with Drummond & Holte’s sub-sampling experiments [59] and the sub-sampling classification tree experiments of Kamei et.al. [61].
- The method with the highest medium performance was, yet again, the simple naïve Bayes we recommended previously [9].
- Just like the Figure 8 results, throwing away data (i.e. under-sampling) does not degrade the performance of the learner. In fact, in the case of j48, throwing away data improved the median balance performance from around 40% to over 70%.

This last point motivated the micro-sampling experiment. Recall that micro-sampling is an under-sampling method that discards most of the majority class while keeping only $M$ examples of the minority class. Figure 11 shows the results of an under-sampling study where $M \in \{25, 50, 75, \ldots\}$ defective modules were selected at random, along with an equal $M$ number of defect-free modules. Note the same visual pattern as before: increasing data does not necessarily
improve balance.

Mann-Whitney tests were applied to test the visual pattern of Figure 11. Detectors learned from small $M$ instances do as well as detectors learned from any other number of instances.

- For five data sets, \{CM1,KC2, KC3,MC2,PC1\}, micro-sampling at $M = 25$ did just as well as anything larger sample size.
- For one data set, KC1, best results were seen at $M = 575$. However, in hundreds of repeats for that data set, in all by one case, $M = 25$ did as well as any larger value.

V. ADDING BUSINESS KNOWLEDGE

The above results offer much support for the limited information content hypothesis. In those results, our learners’ performance did not improve after:

- RandomReduction: 100 randomly selected examples;
- StructuredReduction: 25 examples each of defective/non-defective modules.

If limited information content is the problem, then the solution is clear: give the data miner more information. However, as shown above, it is not useful to use more examples of the same kinds of data. Rather, we need to give our learners different kinds of information.

One different kind of knowledge, not found in the training data of Figure 7, is the criteria by which a learner will be assessed. We will call this evaluation criteria $Q$ since it is applied after another criteria $P$ is used by the learner to build a model.

In theory, data mining could use the evaluation criteria $Q$ to guide their search for better predictors. In practice, this is often not the case. For example:
Fig. 12. Properties of 30 defect predictors learned from Figure 7’s NASA data, sorted by effort. For example, the detector shown at X=1 has (accuracy, effort, pd, pf) of (80, 20, 10, 5)% (respectively). From [6], [28].

\[ P \] : During training, a decision-tree learner may stop branching if the diversity of the instances in a leaf of a branch\(^6\) falls below some heuristic threshold.

\[ Q \] : During testing, the learned decision-tree might be tested on any of the criteria shown in Figure 4.

Predictors that satisfy some intra-learning assessment criteria \( P \) may not satisfy some post-learning assessment criteria \( Q \). For example, Figure 12 shows properties of some defect predictors we have learned from the NASA data of Figure 7 (while only 30 predictors are shown in that figure, we have other results where thousands of other predictors follow exactly the same pattern [6], [28]). Observe how accuracy remains stable over a wide range of changes to \( pd \) and \( effort \); e.g. the predictors labeled \( A \) and \( B \) have similar accuracies while \( pd \) changes by a factor of five. If \( P \) maximizes for accuracy, while \( Q \) prefers detectors with high \( pd \) values, then it is conceivable that the learner will return a detector that satisfies \( P \), but not \( Q \) (e.g. all the detectors on the left-hand-side of Figure 12).

Hence, we say that it is important that the learner’s internal evaluation criteria \( P \) reflects business concern \( Q \) since the latter can change markedly from application to application. For example, in application\(_1\), a quality assurance (QA) team has insufficient budget to inspect all the code. Therefore, they need some sorting policy that increases the defect frequency in modules that are ranked high in the sort. In this application, \( Q \) rewards minimizing the inspection effort while maximizing the number of defective modules they discover.

\(^6\)For numeric classes, this diversity measure might be the standard deviation of the class feature. For discrete classes, the diversity measure might be the entropy measure used in j48 [47].
On the other hand, in application$_2$, the manager of a new contract between a V&V consultancy and a client may feel the need to impress the client. In which case, she may direct her engineers to quickly skim all the code looking for some single high severity error. The priorities of application$_2$ are different to application$_1$ in that the former will condone a review of all the code (at least, at very high speed) while the latter only wants to see some of the code.

Note that some learning schemes support biasing the learning according to the overall goal of the system; for example:

- The cost-sensitive learners discussed by Elkan [62];
- The ROC ensembles discussed by Fawcett [63] where the conclusion is computed from some summation of the conclusions of the ensemble of ROC curves$^7$, proportionally weighted, to yield a new learner.

At best, such biasing is only an indirect control of the $P$ criteria. If the underlying criteria used to guide the search is orthogonal to the success criteria of, say, application$_1$ then cost-sensitive learning and ensemble combinations will not be able to generate a learner that supports that business application.

By this line of reasoning, we have an explanation for the ceiling effect described above: when $P \neq Q$, our learners are blind to their purpose and it is hardly surprising that they cannot find ways to improve their performance. What is required is a new kind of learner-one where a similar user-specified criteria is applied during training and testing. The rest of this paper tests the speculation that such a learner can overcome performance ceilings.

VI. WHICH

The WHICH [13] rule learner loops over the space of possible feature ranges, evaluating various combinations of features. In terms of this discussion, the most important feature of WHICH is that WHICH’s $P$ criteria is very close to application$_1$’s $Q$ (denoted $P \approx Q$) and is wired into the inner loop of WHICH’s learning:

1) WHICH maintains a stack of of combinations of features, sorted by an evaluation criteria $P$. WHICH’s design allows for the easy modification to $P$. The exact $P$ used in this study is discussed below in §IV.C.

2) Initially, WHICH’s “combinations” are just each range of each feature. Subsequently, they can create conjunctions of two or more features.

$^7$ROC= receiver-operator characteristic curves such as graphs of PD-vs-PF or PD-vs-precision
3) Two combinations are picked at random, favoring those combinations that are ranked highly by $P$.

4) The two combinations are themselves combined, scored, then sorted into the stacked population of prior combinations.

5) Go to step 1.

After numerous loops, WHICH returns the highest ranked combination of features. During testing, if a new module satisfies this combination then it is predicted as being “defective”.

In the following experiment, we wanted to isolate the value of unifying the train/test evaluation criteria against all other treatments. Accordingly, we “crippled” WHICH and disabled various internal options. For example, WHICH can optimize each combination via a back select that discards superfluous parts of the combination. Hence, the following results were obtained by a data miner that knows nothing special about over-fitting avoidance or any other AI search technique. The only intelligence used by WHICH to find better defect predictors is $P \approx Q$.

For more details on WHICH, see Appendix IV.

A. About $application_1$

Our study will focus on $application_1$, the details of which are shown in Figure 13. In $application_1$, a QA team working on a limited budget wants to sort the modules that the data miner predicts are defective in order to find (a) those that require urgent inspection, and (b) others than could be inspected later (or never).

We make no presumption that $application_1$ is the only possible way to use data miners. There are many other business applications of defect predictors that do not conform to $application_1$. However, $application_1$ was chosen for two reasons. Firstly, it is a common usage of automatic
defect predictors. For example, if a V&V company is hired to audit the code from some new off-shore client, they may have a large code base to inspect in the shortest possible time.

Secondly, application$_1$ addresses current concerns in the defect prediction literature. Arisholm & Briand [64] argue against certain standard measures of predictor performance such as accuracy (defined in Figure 4), saying that a highly accurate predictor can be undesirable in other ways. For example, accuracy says nothing about the appropriate sort order for reading modules. In application$_1$ we want a QA team to read less while finding more defects. For such a budget-conscious team, if X% of the modules are predicted to be faulty and if those modules contain less than X% of the defects, then the costs of generating the defect predictor is not worth the effort.

Koru et al. [65] have much to say about the relative defect frequency (RDF) of different biasing strategies for selecting which modules should be inspected next. Based on a literature review and empirical studies, they make a strong case the relationship between module size and number of defects is not linear, but logarithmic; i.e. smaller modules are proportionally more troublesome. Accordingly, they argue that LOC can be be used to create a biasing strategy with higher RDF. For example: if one has the resources to inspect 10,000 LOC, then their logarithm defect hypothesis would say that it is better to pick 100 classes of size 100-LOC as opposed to picking 10 classes of 1,000 LOC.

We can use application$_1$ to test the logarithmic defect hypothesis by exploring two sort orders:

- In Koru’s preferred manualUp policy, the smaller modules (those with less LOC) will be inspected first.
- In the opposite manualDown policy, the larger modules will be inspected first.

B. Effort-vs-PD Curves

The relative merits of biasing strategies like manualUp and manualDown can be compared using the effort-vs-pd diagram of Figure 14. The curves in that figure are generated as follows:

- Some oracle selects a set of modules to inspect. In the case of automatic data mining, this would be the modules predicted to be defective. In the case of manualUp and manualDown, it would be all modules.
- The selected are sorted. For example, except for manualDown, we sort all modules ascending on LOC.
The selected set is explored in the sorted order $1 \leq i \leq |\text{selected}|$. Each $x$ value in Figure 14 shows $\sum_i LOC_i$ found in the first $i$ modules.

For each $x$ value, the $y$ value is the percentage of the defective modules seen in the first $i$ modules.

Typically, defect detectors do not trigger on all modules, and have some false alarm rate. For example, the good curve of Figure 14 triggers on $B=43\%$ of the code while only detecting $85\%$ of the defective modules. Similarly, the bad curve stops after finding $30\%$ of the defective modules in $24\%$ of the code. To compute the area under the effort-vs-pd curve, we must fill in the gap between the termination point and $X = 100$. In the sequel, we will assume the QA team only inspects the modules referred to by the data miner. Visually, for the good curve, this assumption would correspond to a flat line running to the right from point $C = 85$.

The effort-vs-pd curve of Figure 14 lets us define reasonable lower bounds on the performance of an automatic data miner being used for application$_1$. For Arisholm & Briand to approve of a data miner, its curve must fall above the diagonal line marked as minimum. This is the region where $pd > effort$; i.e where the QA team can read less and finds more. Also, if Koru et al. are right then the manualUp and manualDown curves should appear as drawn in Figure 14; i.e. manualUp should find defective modules faster than manualDown. A bad automatic method performs worse than manual methods; i.e. its effort-vs-pd curve falls below the performance curves of the manual methods. In application$_1$, there would be no business
justification for a learner that generates (e.g.) the bad curve of Figure 14 since it falls below one of the manual curves.

Figure 14 also lets us define a theoretical upper bound on the performance of any learner tackling application\textsubscript{1}. Imagine an omniscient oracle that restricts the inspections to just the $A\%$ defective modules (in Figure 14, $A = 30\%$). If manual\texttt{Up} was then applied to just those defective modules, then that would result in the best curve. Realistically, defect predictors can approach the best curve, but never reach it. Hence, the most we can hope for is something like the good curve that falls below the best curve and above the manual\texttt{Up} and manual\texttt{Down} curves.

Two more details will complete our discussion of Figure 14. Firstly, in the sequel, the following observation will become a significant point. Even though Figure 14 shows effort-vs-pd, it can also indirectly show pf. Consider the plateau in the good curve of Figure 14, marked with “D”, at around effort = 10, pd = 45. Such plateaus mark false alarms where the detectors are selecting modules that have no defects. That is, one way to maximize the area under an effort-vs-pd curve is to assign a heavy penalty against false alarms that lead to plateaus.

Secondly, when comparing supposedly good defect predictors, it is useful to express their performance in terms of the area under the effort-vs-pd curve, expressed as a ratio of the area under the best curve. To be complete, that evaluation should contain the “$\Delta$” factor that models the effectiveness of QA teams that inspect modules according to the defect predictors’ recommendation (and at $\Delta = 1$, the inspection teams are perfect at recognizing defective modules). Note that that factor applies to the activity that occurs after the data miners runs and the modules are sorted in ascending order by LOC. Hence, it is the same across all data miners. By expressing the value of a defect predictor as a ratio of the area under the best curve, $\Delta$ cancels out. In this way, for application\textsubscript{1}, we can assess the relative merits of different defect predictors independently of $\Delta$.

We use $Q'$ to denote this performance measure; i.e. area under the effort-vs-pd curve, measured as a percentage of the area under the best curve.

\textit{C. Designing P for application\textsubscript{1}}

Recall that $P$ is the criteria used internally by the learner to grow a model. When the learner terminates and outputs a model, $Q$ is used to assess the outputted model.

Our preferred $Q'$ was described in the last section. This section describes the $P'$ used by WHICH to incrementally grow candidate rule sets. In summary, our experiments will using a $P'$ criteria that approximates the intent of $Q'$, even if it does not implement it directly.
The $Q'$ criteria maximizes $pd$ while minimizing $effort$. To emulate that criteria during rule generation, we experimented with:

$$P' = 1 - \frac{\sqrt{pd^2 + \alpha + (1-pf)^2 \times \beta + (1-effort)^2 \times \gamma}}{\sqrt{\alpha + \beta + \gamma}}$$

(4)

where $(pd,pf,effort)$ were normalized to fall between zero and one. The $\alpha, \beta, \gamma$ terms of this expression represent the relative utility of $pd, pf, effort$ respectively. Clearly, $0 \leq (P', \alpha, \beta, \gamma, pd, pf, effort) \leq 1$ and larger values of $P'$ are better. The $pf$ term was absent from the first version of $P'$ but was added after some initial experiments that returned rules with high false alarm rates. Note that increasing the $effort$ or $pf$ leads to a decrease in $P'$.

Initially, we gave $pd$ and $effort$ equal weights; i.e. $\alpha = \gamma = 1$. This yielded disappointing results: the performance of the learned detectors varied wildly across our cross-val experiments. Figure 15 explains why: there exists a small number of modules with very large LOCs. For example, there are 126 modules in the $kc4$ data set, most of them are under 100 lines of code but a few of them are over 1000 lines of code long. The presence of small numbers of very large modules means that $\gamma = 1$ is not recommended. If the very large modules fall into a particular subset of some cross-val, then the performance associated with WHICH’s rule can vary unpredictably from one run to another.

To repair this problem, we had to deemphasize $effort$ and use $pf$ as a surrogate measure. False alarms create plateaus in $effort$-vs-$pd$ curves (recall the above discussion on the point “D” in Figure 14). Hence, in the following experiments, we used a variant of $P'$ that disables $effort$ but places a very large penalty on $pf$; i.e. $\alpha = 1, \beta = 1000, \gamma = 0$. 

Fig. 15. Lines of code in a sample of our data sets.
Fig. 16. Results from all data sets of Figure 7, combined from 10 repeats of a 3-way cross-val, sorted by median $Q'$. The rank values in column 1 were generated in the same manner as Figure 9.

<table>
<thead>
<tr>
<th>data rank treatment</th>
<th>median $Q'$</th>
<th>2nd quartile, median and 3rd quartile</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>70.9</td>
<td></td>
</tr>
<tr>
<td>1 micro20</td>
<td>67.0</td>
<td></td>
</tr>
<tr>
<td>2 manualUp</td>
<td>61.1</td>
<td></td>
</tr>
<tr>
<td>3 naïve Bayes</td>
<td>56.8</td>
<td></td>
</tr>
<tr>
<td>4 manualDown</td>
<td>49.5</td>
<td></td>
</tr>
<tr>
<td>5 which4</td>
<td>49.2</td>
<td></td>
</tr>
<tr>
<td>6 which8</td>
<td>31.2</td>
<td></td>
</tr>
<tr>
<td>6 j48</td>
<td>27.6</td>
<td></td>
</tr>
<tr>
<td>7 jRip</td>
<td>21.9</td>
<td></td>
</tr>
</tbody>
</table>

VII. EXPERIMENTS

Figure 16 shows results from experimental runs with different learners on Figure 7. Each run randomized the order of the data ten times, then performed a N=3-way cross-val study (N=3 was used since some of our data sets were quite small). For each part of the cross-val study, pd-vs-effort curves were generated using:

- **Manual methods**: manualUp and manualDown;
- **Using standard data miners**: the j48 decision tree learner, the jRip rule learner, and our previously recommended naïve Bayes method. For more details on these learners, see Appendices I, II, and III. Note that these standard miners included methods that we have advocated in prior publications [2]–[10].
- Two versions of WHICH: WHICH2 discretized continuous ranges with a log filter, then divided the numerics into two equal width bins; micro20 is WHICH2, plus the micro-sampling strategy discussed in §IV-B. For more details on WHICH, see Appendix IV.

To be accurate, numerous versions of WHICH were explored, each with different discretization policies. We only report WHICH2 and micro20 since they were always on the upper envelope of the results.

A. Overall Results

Figure 16 shows the results for all the data sets of Figure 7, combined. In terms of the title of this paper, the most important result is that WHICH performs relatively better that all of the other methods studied in this paper. That is, unlike our prior results, all the learners in this study do not hover around the same performance ceiling.
<table>
<thead>
<tr>
<th>data</th>
<th>rank</th>
<th>treatment</th>
<th>median $Q'$</th>
<th>2nd quartile, median and 3rd quartile</th>
</tr>
</thead>
<tbody>
<tr>
<td>ar4</td>
<td>1</td>
<td>which2</td>
<td>58.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>manualDown</td>
<td>56.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>naïve Bayes</td>
<td>56.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>manualUp</td>
<td>56.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>which4</td>
<td>49.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>jRip</td>
<td>47.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>micro20</td>
<td>49.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>which8</td>
<td>42.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>j48</td>
<td>38.8</td>
<td></td>
</tr>
<tr>
<td>cm1</td>
<td>1</td>
<td>which2</td>
<td>68.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>micro20</td>
<td>64.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>manualUp</td>
<td>59.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>naïve Bayes</td>
<td>52.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>manualDown</td>
<td>47.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>which8</td>
<td>11.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>jRip</td>
<td>5.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>j48</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>kc1</td>
<td>1</td>
<td>which2</td>
<td>76.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>micro20</td>
<td>75.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>manualUp</td>
<td>67.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>naïve Bayes</td>
<td>61.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>which4</td>
<td>52.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>manualDown</td>
<td>43.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>j48</td>
<td>27.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>jRip</td>
<td>21.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>which8</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>kc2</td>
<td>1</td>
<td>which2</td>
<td>81.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>micro20</td>
<td>74.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>manualUp</td>
<td>69.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>which4</td>
<td>59.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>naïve Bayes</td>
<td>58.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>manualDown</td>
<td>46.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>jRip</td>
<td>42.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>which8</td>
<td>41.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>j48</td>
<td>41.2</td>
<td></td>
</tr>
<tr>
<td>kc3</td>
<td>1</td>
<td>which2</td>
<td>87.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>micro20</td>
<td>76.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>naïve Bayes</td>
<td>64.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>manualUp</td>
<td>64.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>which4</td>
<td>47.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>manualDown</td>
<td>47.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>which8</td>
<td>46.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>j48</td>
<td>23.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>jRip</td>
<td>17.7</td>
<td></td>
</tr>
<tr>
<td>mw1</td>
<td>1</td>
<td>which2</td>
<td>62.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>manualDown</td>
<td>60.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>micro20</td>
<td>55.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>manualUp</td>
<td>47.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>which4</td>
<td>42.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>naïve Bayes</td>
<td>41.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>which8</td>
<td>39.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>j48</td>
<td>20.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>jRip</td>
<td>15.8</td>
<td></td>
</tr>
<tr>
<td>pc1</td>
<td>1</td>
<td>which2</td>
<td>65.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>micro20</td>
<td>64.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>manualUp</td>
<td>60.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>naïve Bayes</td>
<td>51.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>manualDown</td>
<td>44.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>which8</td>
<td>22.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>j48</td>
<td>19.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>jRip</td>
<td>15.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>which4</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 17. Seven examples of pattern #1: WHICH2 ranked #1 and has highest median. This figure is reported in the same format as Figure 9.
Also, measured in absolute terms, WHICH performs very well. In our discussion of Figure 14, the best curve was presented as the upper bound in performance for any learner tackling application\textsubscript{1}. WHICH’s performance rises close to this upper bound, rising to to 70.9 and 80\% (median and 75\% percentile range) of the best possible performance.

Several other results from Figure 16 are noteworthy.

- In a result consistent with our prior publications [9], our naïve Bayes classifier out-performs other standard data miners (j48 and jRip).
- In a result consistent with Kuro et.al.’s logarithmic defect hypothesis, manualUp defeats manualDown.

Fig. 18. Two examples of pattern \#2: While WHICH2 did not achieve the highest medians, it was still ranked \#1 compared to eight other methods. This figure is reported in the same format as Figure 9.

Fig. 19. The only example of pattern \#3: WHICH2 loses (badly) but MICRO20 still ranks high. This figure is reported in the same format as Figure 9.
• Also, in a result consistent with the limited information content hypothesis, micro20 has the same rank as WHICH2 (i.e. their performance is statistically indistinguishable). Recall that, under micro20-sampling, learning is performed using just 40 examples, evenly mixed, of defective and non-defective modules.

• In Figure 16, standard data miners are defeated by a manual method (manualUp). The size of the defeat is very large: median values of 61.1% to 27.6% from manualUp to j48. One very sobering result in Figure 16 is that two widely used methods (j48 and jRip) are defeated by manualDown; i.e. by a a manual inspection method that Koru et al. would argue is the worst possible inspection policy. These results calls into question the numerous prior defect prediction results, including several papers written by the authors [2]–[10].

B. Internal Validity

Figure 16 showed combined results from all of the data sets in Figure 7. From it, we concluded that WHICH is preferred to other methods for application.

Figures 17, 18, and 19 check the internal validity of this conclusion by looking at each data set in isolation. The results divide into three patterns:

• In the eight data sets of pattern #1 (shown in Figure 17), WHICH2 has both the highest median \( Q' \) performance and is found to be in the top rank by the Mann-Whitney statistical analysis.

• In the two data sets of pattern #2 (shown in Figure 18), WHICH2 does not score the highest median performance, but still is found in the top-rank.

• In the one data set that shows pattern #3 (shown in Figure 19), WHICH2 is soundly defeated by manual methods (manualUp). However, in this case, the WHICH variant micro20 falls into the second rank.

In summary, when looking at each data set in isolation, WHICH2 performs very well in 10 of the data sets. Hence, we say that the general conclusion of the last section almost always holds for specific data sets.

C. External Validity

We argue that the data sets used in this paper are far broader (and hence, more externally valid) than seen in prior defect prediction papers. All the data sets explored by Lessmann et al. [12] and our prior work [9] come from NASA aerospace applications. Here, we use that data, plus three extra data sets from SOFTLAB, a Turkish company writing software controllers for
dishwashers (ar3), washing machines (ar4) and refrigerators (ar5). The development practices from these two organizations are very different:

- The SOFTLAB software was built in a profit- and revenue-driven commercial organization, whereas NASA is a cost-driven government entity.
- The SOFTLAB software was developed by very small teams (2-3 people) working in the same physical location while the NASA software was built by much larger teams spread around the United States.
- The SOFTLAB development was carried out in an ad-hoc, informal way rather than the formal, process-oriented approach used at NASA.

Our general conclusion, that WHICH is preferred to other methods for application, holds for \( \frac{7}{8} \) of the NASA data sets and \( \frac{3}{4} \) of the SOFTLAB sets. The fact that the same result holds for such radically different organizations is a strong argument for the external validity of our results.

VIII. Conclusion

Since 2002 [27], we have been using standard data miners such as j48, jRip and naïve Bayes to learn defect predictors. The general pattern of those results was documented above:

- From Figure 6, we see Lessmann et al. [12] documenting an effect that we have also observed [48]: our learners suffer from a ceiling effect where supposedly more sophisticated learners do no better than simple ones.
- From Figure 12, we see that our prior work generated defect detectors from the Figure 7 data with a \( pd \) performance less than the \( effort \). Arisholm & Briand [64] might dismiss these results, arguing that since we are finding X% of the faulty modules after reading more than X% of the code, then the cost of generating the defect predictor is not worth the effort.

These results are troubling, on two counts. Firstly, our prior results do not add value to a budget-conscious test engineering team using the defect predictors to prioritize their inspection process (a task we have called \( application_1 \)). Secondly, neither our own research [48] nor the research of others [12] has resulted in methods that improve our prior results (at least, for the purposes of extracting defect predictors from Figure 7 data).

The above negative results prompted the basic rethinking of the defect prediction problem presented in this paper. Based on:

- the RandomReduction results of Figure 8;
- the StructuredReduction results of Figure 9;
- and the micro20 results shown in Figures 17 & 18 & 19
We assert that the performance of our learners will not improve merely by passing more of the *same* kinds of data to our learners. The *RandomReduction* and *StructuredReduction* experiments offer support for a *limited information content* hypothesis in static code features. i.e. (i) simple learning methods will uncover all that can be found and (ii) more sophisticated data mining methods will yield no more information.

Accordingly, we explored the value of giving the learners *different* kinds of knowledge. Standard learners use some intra-learning assessment criteria \( P \) to generate predictors. The learned model is assessed by a post-learning assessment criteria \( Q \). In all our prior work, \( P \neq Q \). Therefore, it seemed important to try a new kind of learner, called WHICH, where \( P \approx Q \).

The version of WHICH used in this study was deliberately designed to be unsophisticated. For example, it has no over-fitting operator that prunes away superfluous parts of a model. WHICH’s search for predictors is completely random, biased only by some \( P \) function. That is, the performance of this study’s WHICH learner was *solely* determined by \( P \). As such, it is an ideal tool for assessing the value of learners where \( P \approx Q \).

Our experiments used \( P \) criteria that approximates the intent of *application\(_1\)'s* \( Q \) criteria, even though it does not implement it directly. Initially, we tried \( P = Q \) but, under cross-val, the presence of a small number of very large modules (see Figure 15) resulted in wild variations in the measured performance. In the discussion around Figure 13, it was observed that false alarms lead to plateaus in an *effort-vs-pd* curves. That is, one way to increase the area under a *effort-vs-pd* curve is to assign a heavy penalty against false alarms. In the above results, we used the utility function shown in Equation 4 where \( pd:pf \) was weighted 1:1000.

The performance measure used in this study was area under an *effort-vs-pd* curve, expressed as a ratio of the area under the same curve for a theoretical upper bound on any learner tackling *application\(_1\)* (i.e. the *best* curve of Figure 13). WHICH’s results were compared to manual methods and standard learners using *median* performance measures plus statistical tests (Mann-Whitney, 95%) to *rank* the performance of all learners/ manual methods. The results are highly supportive of our hypothesis that learners that use \( P \approx Q \) can out-perform other learners where \( P \neq Q \):

- In 7/11 data sets, WHICH scored the highest median & rank.
- In 2/11 data sets, WHICH scored the highest ranking.
- In the remaining data set, the standard WHICH algorithm (WHICH2) performed very badly but the micro-sampling version of WHICH (micro20) scored the highest rank.
Hence, we recommend the use of WHICH2 or the micro20 variant for learning defect predictors for application$_1$.

This recommendation may not hold beyond the context of application$_1$. This is not necessarily a flaw with this study. Rather, it our view that it is a mistake to assess a learner except in the context of a specific business context. The case for this view has been made above but we add one further point here. The business acceptance of our data mining technology will be greater if we can offer the learners alongside results relating to some business-relevant application.

Finally, we find that the conclusion of Lessmann et al. (quoted in the introduction) is correct in certain contexts. They advise that practitioners are free to choose from a broad set of candidate models when building defect predictors. This is certainly true when defect detectors are assessed via accuracy since, in that case, they all exhibit a ceiling effect. However, when they are assessed by other criteria (e.g. maximizing effort-vs-pd) then some learners such as WHICH can break through that ceiling. Overall, WHICH’s performance was very good and rose to within 70.9 and 80% (median and 75% percentile range) of the best possible performance. Measured on the same scale, other learners such as j48, jRip, and näive Bayes, perform much worse.

REFERENCES


**APPENDIX I**

NÄIVE BAYES CLASSIFIERS

Naive Bayes classifiers offer a relationship between fragments of evidence $E_i$, a prior probability for a class $P(H)$, and a posteriori probability $P(H|E)$:

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

April 30, 2008  
DRAFT
For numeric features, a feature’s mean $\mu$ and standard deviation $\sigma$ are used in a Gaussian probability function [47]:

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

Simple naïve Bayes classifiers are called “naïve” since they assume independence of each feature. Potentially, this is a significant problem for data sets where the static code measures are highly correlated (e.g., the number of symbols in a module increases linearly with the module’s lines of code). However, Domingos and Pazzini have shown theoretically that the independence assumption is a problem in a vanishingly small percent of cases [66]. This result explains (a) the repeated empirical result that, on average, seemingly naïve Bayes classifiers perform as well as other seemingly more sophisticated schemes (e.g., see Table 1 in [66]); and (b) our prior experiments where naïve Bayes did not perform worse than other learners that continually re-sample the data for dependent instances (e.g., decision-tree learners that recurse on each “split” of the data [45]).

**APPENDIX II**

**THE J48 DECISION TREE LEARNER**

j48 [47] is a JAVA port of Quinlan’s decision tree learner C4.5, release 8 [45]. j48 is a iterative dichotomization algorithm that seek the best attribute value splitter that most simplifies the data that falls into the different splits. Each such splitter becomes a root of a tree. Sub-trees are generated by calling iterative dichotomization recursively on each of the splits. j48 is defined for discrete class classification and uses an information-theoretic measure to describe the diversity of classes within a data set. A leaf generated by j48 stores the most frequency class seen during training. During test, an example falls into one of the branches in the decision tree and is assigned the class from the leaf of that branch. J48 tends to produce big “bushy” trees so the algorithm includes a pruning step. Sub-trees are eliminated if their removal does not greatly change the error rate of the tree.

**APPENDIX III**

**THE RIPPER RULE LEARNER**

RIPPER [53] is a rule-covering algorithm; i.e. one rule is learned at each pass for the majority class. All the examples that satisfy the rule condition are marked as covered and are removed from the data set. The algorithm then recurses on the remaining data.
RIPPER takes a rather unique stance to rule generation and has operators for *pruning, description length* and *rule-set optimization*. For a full description of these techniques, see [67]. In summary, after building a *rule*, RIPPER performs a back-select to see what parts of a *condition* can be pruned, without degrading the performance of the rule. Similarly, after building a *set of rules*, RIPPER tries pruning away some of the rules. The learned rules are built while minimizing their *description length*; the size of the learned rules, as well as a measure of the rule errors. Finally, after building rules, RIPPER tries replacing straw-man alternatives (i.e. rules grown very quickly by some naïve method).

**APPENDIX IV**

**THE WHICH RULE LEARNER**

WHICH inputs a set of training examples and a evaluation criteria $P \approx Q$ and outputs a single rule that is best for maximizing $P$.

Following the recommendations of [9], all numeric data is transformed using $x = \log(max(0.00001, x))^8$ then divided into $N$ ranges using equal-width discretization. Given a value $X$ from a column of data with minimum and maximum values $min, max$ (respectively), then

$$RANGE(X) = \text{floor} \left( \frac{X - \text{min}}{(\text{max} - \text{min})/N} \right) + 1$$

(6)

This discretization policy divides the values from a continuous variable into $N$ ranges. In this study, we explored several variants of WHICH using different number of ranges including WHICH2, WHICH4, and WHICH8 that use $N=2,4,8$ ranges (respectively).

Internally, WHICH maintains a stack of conditions, sorted by the evaluation criteria $P$. For example, if $P_{pd}$ is the probability of detection (see Equation 1 in Figure 4), then the condition $LOC == 0$ has the score $P_{pd} = 0$ (since all modules have at least one line of code). Hence, according to $P_{pd}$, this condition will be sorted to the bottom of the stack.

WHICH initializes the stack by scoring all the discretized ranges from Equation 6 generated from the discretization. As shown in Figure 20, all the raw scores are accumulated and normalized using

$$\text{normalized}(i) = \frac{\sum_i \text{score}(i)}{\text{sum of all scores}}$$

After initialization, WHICH enters a loop:

---

Many static code attributes have an exponential distribution with a small number of very large outliers. This “log transform” flattens the distribution and makes the learning simpler. For more details, see [9].
Fig. 20. Example of scores on the WHICH stack.

<table>
<thead>
<tr>
<th>i</th>
<th>raw score</th>
<th>cumulative</th>
<th>normalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>100</td>
<td>100 / 220 = 0.45</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>200</td>
<td>200 / 220 = 0.91</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>210</td>
<td>210 / 220 = 0.95</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>213</td>
<td>213 / 220 = 0.97</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>216</td>
<td>216 / 220 = 0.99</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>218</td>
<td>218 / 220 = 0.99</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>220</td>
<td>220 / 220 = 1.00</td>
</tr>
</tbody>
</table>

1) *Old* = score of the top of the stack
2) *Loop* number of times, repeat:
   2a : Pick two items from the stack, favoring those with higher score.
   2b : Combine them and score the combination using $E$
   2c : Sort the new combination into the stack
3) *New* = score of top of stack
4) If *New* > *Old* goto 1
5) Else return top of stack

Note that, initially, all the stack items are single conditionals. As the loop continues, singletons may be combined into doubles which might then combine into triples, etc.

We set the *Loop* variable using our engineering judgment. Figure 21 shows typical results from WHICH (running over sample UCI data sets [44]). The score of the top-of-stack condition usually stabilizes in a remarkably short time (after less than a dozen “picks”; i.e. applications of step 2, described above). Occasionally, modest improvement is seen after 100 picks (see plot marked with an “A”). Hence, to be cautious, we set *Loop* to 200.

When selecting items for combination, WHICH (a) generates a random number $0 \leq r \leq 1$; (b) runs down the stack from top to bottom; (c) returns the first condition for which the normalized cumulative score is less than or equal to $r$. On average, this approach returns conditions nearer top-of-stack (i.e. those with higher score).

Note that, when combining conditions, ranges from *different* features are joined using a *conjunction* and ranges from the *same* feature are joined with a *disjunction*. For example, adding $a = 1$ to “$a = 2 \land b = 3$” results in “$(a = 1 \lor a = 2) \land b = 3$”.

Also, when adding new combinations to the stack, if a counter holds the total score of all conditions added to the stack, then the stack needs only hold the sorted raw score column. All
of the other columns can be computed-on-the-fly during by the condition selection algorithm described in the last point. This avoids tedious updates of cumulative scores over the whole stack.

 WHICH is fully described elsewhere [13]. For the reader conversant with AI literature, we remark:

• WHICH is a stochastic variant of beam search where the maximum height of the stack is the size of the beam. A standard beam search sweeps out over a tree of possibilities and performs cautious additions to each leaf (at most, only graft one more test to the current branch). WHICH’s original design was a non-cautious beam search that copied entire sub-trees from other branches in the search, then grafted them onto the current branch. On reflection, we realized that the the above stack structure achieves the same goal, while being much simpler to implement.

• The early stabilization of the top-of-stack is consistent with the back door variable effect discussed in the constraint satisfaction literature [68]; i.e. many domains have a small number of variables that control everything else. If a domain has such “back doors” then (a) all solutions must use them; (b) all changes to the output variables will be associated with different ranges for the back doors; (c) a stochastic search like WHICH will suffice to find them. Elsewhere, we offer extensive discussions on the implications of back doors on decision making in software engineering [69], [70].

 WHICH would be a fruitful workbench for further experimentation. Our current results leaves
many unexplored possibilities:

- It is possible to restrict the size of the stack to some maximum depth (and new combinations that score less than bottom-of-stack are discarded). For the study shown here, we used an unrestricted stack size.

- Currently, WHICH sorts new items into the stack using a linear time search from the top-of-stack. This is simple to implement via a linked list structure but a faster alternative would be a binary-search over skip lists [71].

- Other rule learners employ a greedy back-select to prune conditions. To implement such a search, check to see if removing any part of the combined condition improves the score. If not, terminate the back select. Else, remove that part and recurse on the shorter condition. Such a back-select is coded in the current version of WHICH, but the above results were obtained with back-select disabled.

- Currently our default value for \textit{Loop} is 200. This may be an overly cautious setting. \textit{Loop} might be safely initialized to, say, 20 and only increased if no dramatic improvement is seen in the first loop. Our initial experiments suggest that, for most domains, this would yield a ten-fold speed increase.

We encourage further experimentation with WHICH. The current release is released under the GPL3.0 license and can be downloaded from \url{http://unbox.org/wisp/tags/which}. 