Limits of Learning Defect Predictors

***Abstract*—**

I. Introduction

Exponential cost increase quickly exhausts finite QA resources. Hence, blind spots can't be avoided and must be managed. Standard practice is to apply the best available assessment methods on the sections of the program that the best available domain knowledge declares is the most critical. However, this focus on certain sections can blind us to defects in other areas which, through interactions, may cause similarly critical failures. Therefore, the standard practice should be augmented with a *lightweight sampling policy* that (a) explores the rest of the software and (b) raises an alert on parts of the software that appear problematic. This sampling approach is incomplete by definition. Nevertheless, it is the only option when resource limits block complete assessment.

One such lightweight sampling policy is the use of *fault predictors*. To learn such prediction models, either from projects previously developed in the same environment or from a continually expanding base of current project's artifacts, tables of examples are formed where one column has a boolean value for "faults detected" and the other columns describe software features such as (i) lines of code, (ii) number of unique symbols [1], or (iii) max. number of possible execution pathways [2]. Each row in the table holds data from one "module"; i.e. the unit of functionality. Depending on the language, modules may be called "functions", "methods", "procedures" or "files". The data mining task is to find combinations of features that predict for the value in the defects column. Once such combinations are found, managers can use them to determine where to best focus their QA effort. Better yet, if they have already focused their QA effort on the most critical portions of the system, the detectors can "nudge" them towards areas that are also in need of quality improvement.

For many years we relied upon straightforward application of data mining algorithms to learn quality predictors from the artifacts generated by a software project; see [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43], [44].

Standard verification and validation (V&V) textbooks [45] advise using static code complexity attributes to decide which modules are worthy of manual inspections. For several years, the PIs 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 predicts that they exhibit high code complexity measures.

Nevertheless, static code attributes are hardly a complete characterization of the internals 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 [46]. Fenton uses this example to argue the uselessness of static code attributes for fault prediction.

An *alternative interpretation* of Fenton's example is that static attributes can never be a certain indicator of the presence of a fault. Nevertheless, they are useful as probabilistic statements that the frequency of faults tends to increase in code modules that trigger the predictor.

Shepperd & Ince and Fenton & Pfleeger might reject the *alternative interpretation*. They present empirical evidence that the McCabe static attributes offer nothing more than uninformative attributes that are mostly the derivatives of the lines of code. Fenton & Pfleeger note that the main McCabe's attribute (cyclomatic complexity, or $v(g)$) is highly correlated with lines of code [46]. Also, Shepperd & Ince remark that "for a large class of software it (cyclomatic complexity) is no more than a proxy for, and in many cases outperformed by, lines of code" [47].

If Shepperd & Ince and Fenton & Pfleeger are 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 NASA code we have studied [10], [17], [19], [25], [27], [30], [31], [40], [41], [48]. Using NASA data, our fault prediction models find defect predictors [40] with a probability of detection (pd) and probability of false alarm (pf) of $mean(pd, pf) = (71\%, 25\%)$. These values can be best understood via a comparison against known industrial baselines [49], [50].

The research work referenced above has produced useful results. However, there is much room for improvement; e.g. probabilities of detection reported in recent repeatable studies, should not only be achievable on a regular basis but improved.

It has proved difficult to achieve those improvements. In our January 2007 TSE study [40] we set out to define a repeatable experiment in learning defect predictors. The intent of that work was to offer a benchmark in defect prediction that other researchers could repeat/ improve/ refute. That experiment included:

- Public domain data sets (from the the PROMISE repository[1]);
- Open source data mining tools (the WEKA toolkit [51]);
- Repeated randomization of the order of training data (to avoid order effects);
- 10-way cross-validation (to assess the results via data *not* used in training);
- Learning via multiple types of machine learning algorithms (rule learners, decision tree learners, Bayes classifiers);
- Assessment via multiple criteria such as probability of detection (pd), probability of false alarm (pf), and $balance$ that combines $\{pd, pf\}$;
- Statistical hypothesis tests over the assessment criteria;
- Novel visualization methods for the results;
- Feature subset selection to find the most important subset of the static code features;

Since that study, we have tried to find better data mining algorithms for defect prediction. To date, we have failed. Our recent (as yet, unpublished) experiments have found no additional statistically significant improvement from the application of the following data mining methods: logistic regression; average one-dependence estimators [52]; under- or over-sampling [53] random forests [54], RIPPER [55], J48 [56], OneR [57] and bagging [58]. Only boosting [59] on discretized data offers a statistically better result than a Bayes classifier. However, we cannot recommend boosting: the median improvement is quite negligible and boosting is orders of magnitudes slower than a simple Bayes classifier.

Other researchers have also failed to improve our results. For the past four years, we participated in the PROMISE workshop on repeatable software engineering experiments. The rule of that workshop reauires that if a paper offers an empirical conclusion, then it must also offer the data used to reach the conclusions. That data is stored on-line in the PROMISE repository, which we administer and maintain for the PROMISE community. Our work on that web site, plus our interaction with the PROMISE community, gives us a unique insight into the empirical analysis as well as access to, as yet, unpublished results. Consequently, we are aware of studies by other researchers (currently under review) that tried 25 other

[1] `http://promisedata.org/repository`

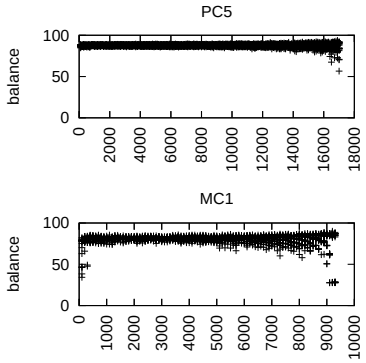


Fig. 1. Training set size vs $balance$.

data mining methods in fault prediction experiments. Those studies investigated the statistical difference of the results from 25 learners on the same datasets using the Wilcoxon signed ranked test [60]. The Bayesian method discussed above ties in first place along with 15 other methods.

How can we explain all these failed attempts to improve fault prediction models in repeatable experiments using same (PROMISE) data sets? One possibility is that all these experiments use simplistic static code features such as lines of code, number of unique symbols in the module, etc. We agree with Shepperd & Ince and Fenton & Pfleeger that such simplistic static code features are hardly a complete characterization of the internals of a module. Static code features might be best described having *limited information content*, with two properties. Firstly, they can be quickly and completely discovered by even simple Bayesian classifiers. Secondly, more sophisticated methods will discover no further information.

Based on all that work we make the following assertion:

> *There has been too much emphasis on machine learning, and too little attention on the use of specific software engineering knowledge.*

Decades of AI research into data mining have resulted in automatic and rapid methods, which can be easily applied to the learning of software quality predictors. While the current generation of detectors are demonstrably useful and better than current industrial best practice [37], [40], we believe that these AI methods have hit a *performance ceiling*; i.e., some inherent upper bound on the amount of information offered by, say, static code features when identifying modules which contain faults. Hence, we doubt that better quality predictors can be found just by trying better machine learning methods.

II. Validation of Limited Information Content

We have made two tests of this *limited information* conjecture. *Test one* checked how *little* information was required to learn a defect predictor. *Test two* augmented the static code features with other features. In *test one*, fault predictors were learned from $N \in \{100, 200, 300, ...\}$ instances then $Test$ed on another 100 instances. For each N, 10 experiments were performed where training was conducted on $Train = 90\% * N$ instances. For all experiments, the $N, Train, Test$ instances were selected at random.

Test one was conducted on twelve data sets, all of which yielded results like those depicted in Figure 1 [61]. In that figure, the X-axis is the size of training set and the Y-axis is the $balance$ measure. Note that the performance foes 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 was actually detrimental (witness the widening variance as the training set increases). A Mann Whitney U test [62] (95% confidence) confirms the visual pattern apparent in Figure 1: static code features used as the basis for predicting module's fault content reveal all that they can reveal after as little as 100 instances.

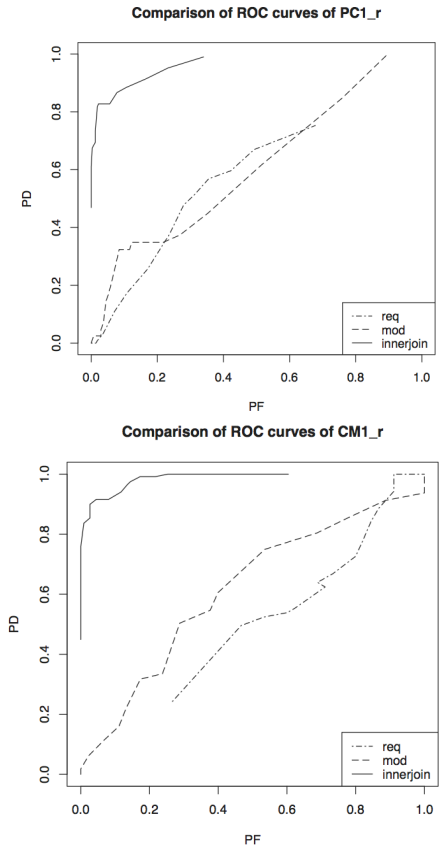


Fig. 2. $\{pd, pf\}$ curves seen when using code and/or requirements features. From [10].

To implement *test two*, we linked code modules to the requirements that prompted their development. Features were extracted from the requirements using a lightweight linear-time text parser. Those features included *weak words* such as "TBD" and "but not limited to"; *imperatives* such as "must", "should"; and *options* such as "can", and "may". Training instances for the fault prediction models were then created using just static code features, just requirements features, or both (using a database inner join operation to connect requirements with modules in which they have been implemented).

The results are shown in Figure 2. This figure plots pd vs pf seen in the two data sets after a 10-way cross validation. The ideal spot on these ROC curves is top left; i.e. no false alarms and perfect detection ($\{pd, pf\} = \{1, 0\}$). The dashed lines on those plots show $\{pd, pf\}$ results when fault prediction models used requirements or code features in isolation. The solid lines show the results of models which used these two kinds of features in combination. Note the remarkable improvement: learning from *multiple perspectives* such as code *and* requirements measures in conjunction, lead to dramatically improved pd and pf.

The knowledge of the values of multiple feature sets from different sources along the project's development life cycle leads to conclusion that the resulting models likely represent domain-specific, if not project-specific solutions. Figure 2 strongly suggest in favor of for using multiple feature sets from different software artifacts. While an

r3.5in

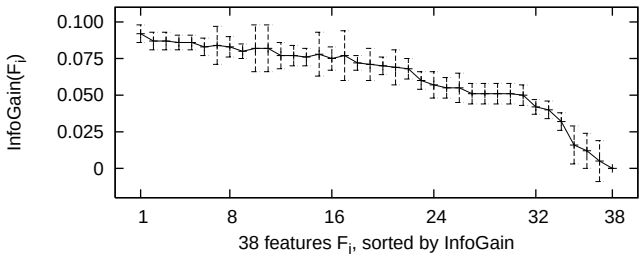


Fig. 3. Mean and standard deviation (t-bars) of InfoGain in 10*90% random samples of defect data.

exciting result, it has some drawbacks. Different projects use different tools and build software using different processes. Hence, different projects have access to different sets of features which may be useful for predicting where faults hide; e.g. an agile process may have no access to the detailed requirement documents used to build model in Figure 2.

Other experiments also suggest that the *best* set of features are domain specific. *Feature subset selection* (FSS) checks if feature subsets performs as well as supersets. In all the data sets studied in [40], 38 static code features could be pruned down to three *without* reducing the efficacy of the defect predictor. Significantly, in each data sets, FSS selected *different features*. That is, FSS found no universally *best* features for defect prediction. An analogous result has been been reported by Shepperd and Ince [47]. They reviewed 18 publications in which no *best* predictive feature was found. In those publications, an equal number of studies reported that cyclomatic complexity is the same, is better, or is worse than LOC in predicting defects.

Figure 3 explains the FSS results and the results of Shepperd and Ince. It also cautions that the best set of features will change from domain to domain. This figure plots the *information gain* associated with 38 static code features. This *infogain* measure is used in decision tree learners [56] to select the feature that best divides up the examples[2]. The first 16 features have very similar information content. In fact, given the standard deviations shown in the error bars of that figure, the 16th-ranked feature is statistically insignificantly different to the top-ranked features. Hence, minor changes in the training sample change what features are found to be the *best* software quality predictors.

Based on the above, in contrast with the current trend in the literature, we recommend against trying supposedly better AI methods to learn defect predictors. We make this recommendation for two reasons.

Firstly, different data miners have failed to improve the state of the art in learning defect predictors. The early plateau effect of Figure 1 suggests that static code features yield all they can yield after as little as a few hundred examples. This information can be found by relatively simple learners such as Naive Bayes. That is, the added value of more sophisticated AI data miners must be questioned.

Secondly, there is a limited generality in the predictors learned by standard AI data miners. Figure 2 showed that combining data sources can yield fault predictors that out-perform predictors learned from either source in isolation. Different domains and different projets offer different data sources and, consequently, the *best* set of features to use in a quality prediction will be domain-dependent. Further evidence for the domain dependent nature of quality predictors comes from Figure 3. Given a set of features, many of them compete to be the *"best"* one and minor variations in the sampling methods can yield different selection of *best* predictors.

[2]Infogain compares the number of bits required to encode the distribution of defective/non-defective classes before and after dividing the examples amongst the range of feature F.

III. Over-Under Sampling

Over and under-sampling are well known techniques in the field of Data Mining. Both of these preprocessing techniques have the potential to improve the results of a learner. To sample, you must first pick a particular class as a goal. For the MDP datasets, we picked true. Your sampling program will take two passes through the data. During the first pass, a table of frequency counts is built. If the desired goal is reached in fewer instances than the other classes, a second pass is taken through the data. This is where the two sampling techniques differ. In the case of under-sampling, instances are printed until the combined number of instances with other classes is equal to the number of instances with the desired goal. This results in a much smaller dataset, but your desired goal is not lost in the great big data soup.

Over-sampling follows a similar procedure. Instead of limiting the number of instances with other classes, the sampling program instead add new instances with your desired goal. Ultimately, you have a much larger dataset. We have used a fairly simple sampling method. For both sampling techniques, half of the final dataset will consist of instances with the desired goal.

In this field, many hold the opinion that more data is better. If you are looking for the diamonds in the dust, youll find more diamonds when you have more dust. Through this experiment, we hope to show the effects of over and under-sampling on the MDP datasets as they are processed with the Nave Bayes classifier and the J48 implementation of the C4.5 tree learner. We actually believe that under-sampling will yield the best results, challenging the more is better mantra.

A. Experiment

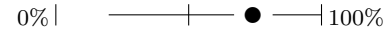
To confirm our hypothesis, and to test the general effects of the sampling techniques, we put together a 10-way cross-validation experiment. We first created two sets, each with a different sampling method. For each of these and the original dataset, we split into a training set (90%) and a test set (10%). We then learned on the data using the Weka implementations of Nave Bayes and J48. We repeated each of these steps ten times for each of the MDP datasets.

We compared the results based on the balance (a combination of $\{pd, pf\}$that decreases if pd decreases or pf increases). We ran Mann-Whitney U tests and formed win-loss-tie tables based on the learner and the sampling method used. We also created win-loss-tie tables for learner, sampling method, and dataset.

B. Results

We can see from Figure 4 that when processing datasets with the J48 learner, under-sampling dramatically improves our results. Oversampling shows a slight advantage over no sampling at all. These results arent quite so clear for Nave Bayes. Under-sampling tied with filter, and over-sampling produced worse results. This didnt quite produce the level of detail that we would like, so we have also looked at quartile results.

Naive Bayes, Under-Sampled [19.9, 67.1, 74.1, 81.6,100.0]

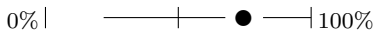


Naive Bayes, Over-Sampled [17.5, 42.0, 62.5, 72.2,100.0]

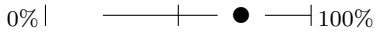
Learner	Sample	W-L-T
J48	Over	1-4-0
	Under	3-0-2
	None	0-5-0
NB	Over	2-3-0
	Under	3-0-2
	None	3-0-2

Fig. 4. W-L-T table for learner and sampling policy.

Naive Bayes, No Sampling [21.9, 67.7, 74.6, 81.9,100.0]

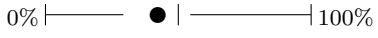


J48, Under-Sampled [21.6, 64.8, 73.6, 82.6,100.0]



J48, Over-Sampled [0.0, 29.3, 45.6, 56.2,100.0]

J48, No Sampling [0.0, 29.3, 42.3, 54.5,100.0]



We can more clearly see the difference from the quartile data. The balance increase from under-sampling for J48 is huge. Over-sampling only presents a small improvement, only a three percent increase in balance. The Nave Bayes results were higher to begin with. On sets with no sampling policy, Nave Bayes had a midpoint of 74.6 compared with J48s 42.3. In fact, Nave Bayes performed slightly better when no sampling was done at all, scoring a midpoint of 74.6 when not sampling and 74.1 when under-sampling. Over-sampling hurt the results, bringing the midpoint down to 62.5.

Figure 7 shows the results of an under-sampling study where $N \in \{25, 50, 75, ...\}$ defective modules were selected along with an equal N 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 this visual pattern. Detectors learned from just N instances do as well as detectors learned from any other number of instances:

- N=25 for {CM1,KC2,KC3,MC1,MC2,MW1,PC1,PC2}
- N=75 for {PC3} (and N=25 losses three times out of 11).
- N=200 for {PC4} (but N=25 losses once out of 13 trials)
- N=575 for {KC1} (for N=25 only lost once out of 25 trials)
- N=1025 for {PC5} (but at N=25, the lost lose only once our of 25 trials)

IV. How much data is required?

A. Design

A curious aspect of the above results is that defect predictors were learned using only a handful of defective modules. For example, consider a 90%/10% train/test split on $pc1$ with 1,109 modules, only 6.94% of which are defective. On average, the test set will only contain $1109 * 0.9 * 6.94/100 = 69$ defective modules. Despite this, $pc1$ yields an adequate median $\{pd, pf\}$ results of $\{88, 34.5\}\%$.

Experiment #2 was therefore designed to check how *little* data is required to learn defect predictors. Experiment #2 was essentially the same as the last experiment, but without treatment #1 (the cross-company study). Instead, experiment #2 took the 12 example tables and learned predictors using:

- Treatment 3 (reduced WC): a randomly selected subset of the 90% rows of this table;

Specially, after randomizing the order of the rows, training sets were built using just the first 100,200,300,... rows in the tables. After training, the learned theory was applied to 100 rows not used in training (selected at random).

Experiment #1 only used the features found in all tables of data. For this experiment, we imposed no such restrictions and used whatever features were available in each data set.

B. Results from Experiment #2

Recall that "balance" is defined to be a combination of $\{pd, pf\}$ that decreases if pd decreases or pf increases. As shown in Figure 5, there was very little change in balanced performance after learning from 100,200,300,... examples. Indeed, there is some evidence that learning from larger training sets had detrimental effects: the more training data, the larger the variance in the performance of the learned predictor. Observe how, in $pc2$, as the training set size increases (moving right along the x-axis) the dots showing the on balance performance start spreading out. A similar, but smaller, *spread effect* can be see in $kc2$ and $mc1$.

The Mann-Whitney U test was applied to check the visual trends seen in Figure 5. For each table, all results from training sets of size 100,200,300... were compared to all other results from the same table. The issue was "how much data is enough?" i.e. what is the minimum training set size that never lost to other training set of a larger size. Usually, that min value was quite small:

- In seven tables $\{cm1, kc2, kc3, mw1, pc3, pc4\}$, $min = 100$;
- In $\{kc1, pc1\}$, $min = \{200, 300\}$ instances, respectively.

In other tables of data, min was much larger. In $\{pc2, mc1, pc5\}$ the min values were found at $\{4900, 8300, 11000\}$, respectively. However, much smaller training set sizes performed nearly as well as these larger training sets:

- In $pc5$, predictors learned from 300 examples only lost to other sizes twice in 169 trials;
- In $mc1$, predictors learned from 400 examples only lost to other sizes once out of 92 trials);
- In $pc2$, predictors learned from 800 examples only lost to other size twice out of 53 trials.

We explain the experiment #2 results as follows. These experiments used simplistic static code features such as lines of code, number of unique symbols in the module, etc. Such simplistic static code features are hardly a complete characterization of the internals of a function. Fenton offers an insightful example where *the same* functionality is achieved using *different* programming language constructs resulting in *different* static measurements for that module [46]. We would characterize such static code features as having *limited information content*. Limited content is soon exhausted by repeated sampling. Hence, such simple features reveal all they can reveal after a small sample

C. Sanity Checks on Experiment #2

This section checks for precedents on the Experiment #2 results and can be skipped at first reading of this paper.

There is also some evidence that the results of Experiment 2 (that performance improvements stop after a few hundred examples) has

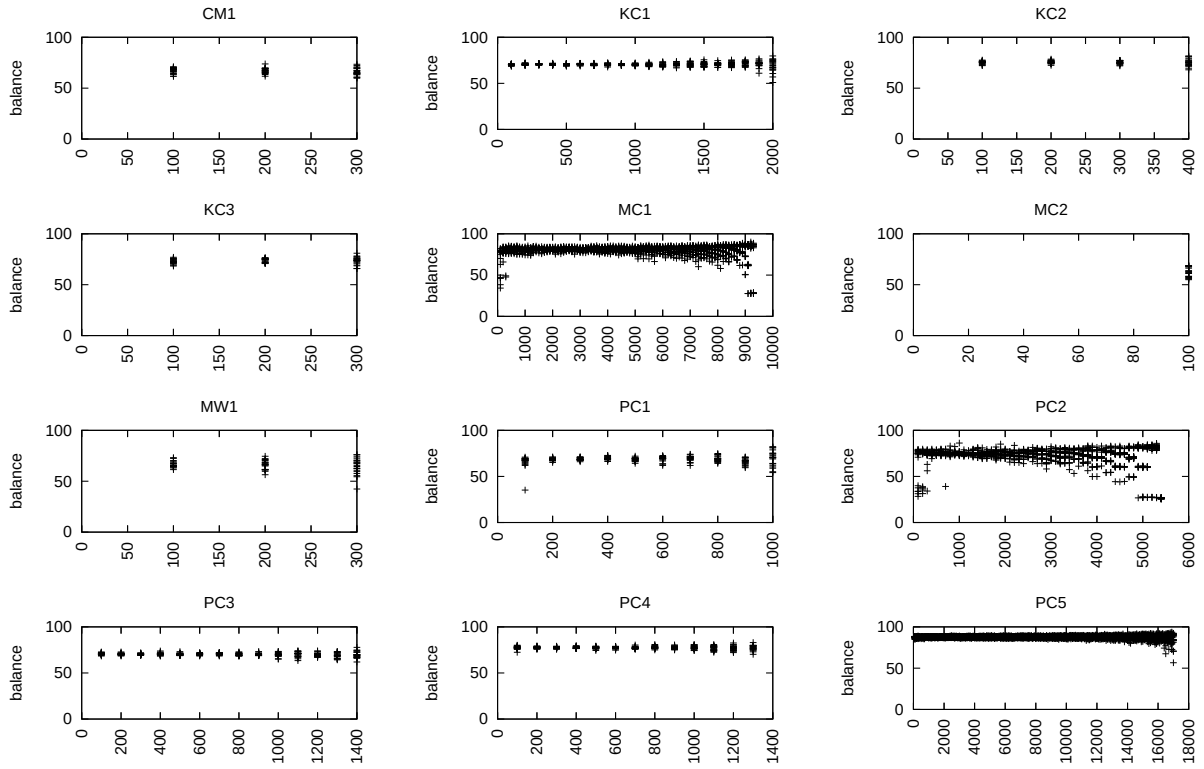


Fig. 5. Results from experiment #2. Training set size grows in units of 100 examples, moving left to right over the x-axis. The MC2 results only appear at the maximum x-value since MC2 has less than 200 examples.

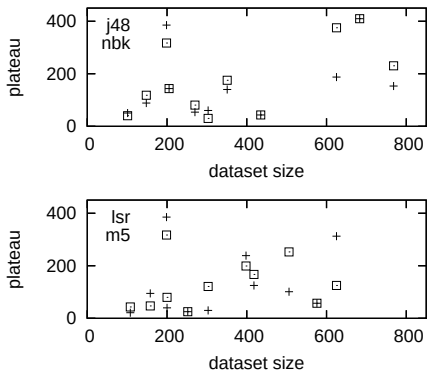


Fig. 6. Y-axis shows plateau point after learning from data sets that have up to X examples (from [63]). The top plot shows results from using Naive Bayes (nbk) or a decision tree learner (j48) [56] to predict for discrete classes. Bottom plot shows results from using linear regression (lsr) or model trees (m5) [64] to learn predictors for continuous classes. In this study, data sets were drawn from the UC Irvine data repository [65].

been seen previously in the data mining literature (caveat: to the best of our knowledge, this is first report of this effect in the defect prediction literature):

- In their discussion on how to best handle numeric features, Langley and John offers plots of the accuracy of Naive Bayes classifiers after learning on 10,20,40,..200 examples. In those plots, there is little change in performance after 100 instances [66].
- Orrego [63] applied four data miners (including Naive Bayes) to 20 data sets to find the *plateau point*: i.e. the point after which there was little net change in the performance of the data miner. To find the plateau point, Oreggo used t-tests to compare the results of learning from Y or $Y + \Delta$ examples. If, in a 10-way cross-validation, there was no statistical difference between Y and $Y + \Delta$, the plateau point was set to Y. As shown in Figure 6, many of those plateaus where found at $Y \leq 100$ and most were found at $Y \leq 200$. Note that these plateau sizes are consistent with the results of Experiment 2.

V. Conclusions

From these results, it is clear that J48 is improved by both over and under-sampling. Under-sampling, in fact, represents a huge improvement. Less data is clearly better than more. It is less clear how Nave Bayes is affected by sampling policies. When looking at results by dataset, under-sampling and no sampling policy are usually indistinguishable. In some cases, one outperforms the other. Over-sampling universally produced worse results. This does clearly show that for Nave Bayes, more data is not better. It also shows that narrowing your current data will not hurt the results, even if it does not improve them.

In summary, the research challenge is clear:

- We should stop trying out supposedly better AI data miners. Instead, we must look to other sources of information based on the software engineering knowledge and project's business environment if we are going to improve on our current generation of quality predictors.
- We must acknowledge that it is insufficient to merely apply one machine learning method and then use the model on all future problems. Rather, we need to find cost-effective methods to quickly learn domain dependent quality predictors.

References

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

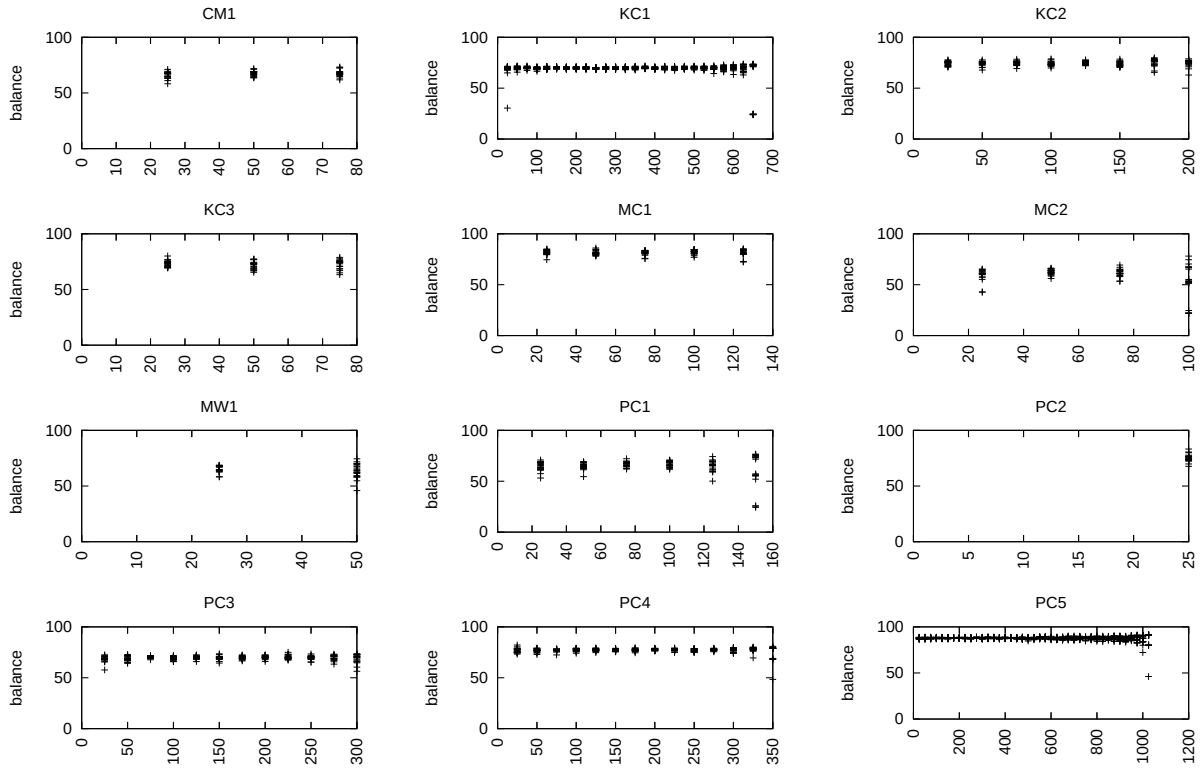


Fig. 7. Under-sampling results

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

[3] Z. Chen, T. Menzies, and D. Port, "Feature subset selection can improve software cost estimation," in *Proceedings, PROMISE workshop, ICSE 2005*, 2005, available from http://menzies.us/pdf/05/fsscocomo.pdf.

[4] M. Connell and T. Menzies, "Quality metrics: Test coverage analysis for smalltalk," in *Tools Pacific, 1996, Melbourne*, 1996, available from http://menzies.us/pdf/96conel.pdf.

[5] A. Dekhtyar, J. H. Hayes, and T. Menzies, "Text is software too," in *International Workshop on Mining Software Repositories (submitted)*, 2004, available from http://menzies.us/pdf/04msrtext.pdf.

[6] M. Feather and T. Menzies, "Converging on the optimal attainment of requirements," in *IEEE Joint Conference On Requirements Engineering ICRE'02 and RE'02, 9-13th September, University of Essen, Germany*, 2002, available from http://menzies.us/pdf/02re02.pdf.

[7] P. Haynes, T. Menzies, and G. Phipps, "Using the size of classes and methods as the basis for early effort prediction; empirical observations, initial application; a practitioners experience report," in *OOPSLA Workshop on OO Process and Metrics for Effort Estimation*, 1995.

[8] P. Haynes and T. Menzies, "C++ is Better than Smalltalk?" in *Tools Pacific 1993*, 1993, pp. 75–82.

[9] ——, "The Effects of Class Coupling on Class Size in Smalltalk Systems," in *Tools '94*. Prentice Hall, 1994, pp. 121–129.

[10] Y. Jiang, B. Cukic, and T. Menzies, "Fault prediction using early lifecycle data," in *ISSRE'07*, 2007, available from http://menzies.us/pdf/07issre.pdf.

[11] K. Lum, J. Hihn, and T. Menzies, "Sudies in software cost model behavior: Do we really understand cost model performance?" in *ISPA Conference Proceedings*, 2006, available from http://menzies.us/pdf/06ispa.pdf.

[12] T. Menzies and E. Sinsel, "Practical large scale what-if queries: Case studies with software risk assessment," in *Proceedings ASE 2000*, 2000, available from http://menzies.us/pdf/00ase.pdf.

[13] T. Menzies, "Practical machine learning for software engineering and knowledge engineering," in *Handbook of Software Engineering and Knowledge Engineering*. World-Scientific, December 2001, available from http://menzies.us/pdf/00ml.pdf.

[14] T. Menzies and H. Singh, "How AI can help SE; or: Randomized search not considered harmful," in *AI'2001: the Fourteenth Canadian Conference on Artificial Intelligence, June 7-9, Ottawa, Canada*, 2001, available from http://menzies.us/pdf/00funnel.pdf.

[15] T. Menzies, E. Chiang, M. Feather, Y. Hu, and J. Kiper, "Condensing uncertainty via incremental treatment learning," in *Software Engineering with Computational Intelligence*, T. M. Khoshgoftaar, Ed. Kluwer, 2003, available from http://menzies.us/pdf/02itar2.pdf.

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

[17] T. Menzies, R. Lutz, and C. Mikulski, "Better analysis of defect data at NASA," in *SEKE03*, 2003, available from http://menzies.us/pdf/03superodc.pdf.

[18] T. Menzies and J. D. Stefano, "More success and failure factors in software reuse," *IEEE Transactions on Software Engineering*, May 2003, available from http://menzies.us/pdf/02sereuse.pdf.

[19] J. D. Stefano and T. Menzies, "Machine learning for software engineering: Case studies in software reuse," in *Proceedings, IEEE Tools with AI, 2002*, 2002, available from http://menzies.us/pdf/02reusetai.pdf.

[20] Y. Liu, T. Menzies, and B. Cukic, "Data sniffing - monitoring of machine learning for online adaptive systems," in *IEEE Tools with AI*, 2002, available from http://menzies.us/pdf/03datasniffing.pdf.

[21] T. Menzies and Y. Hu, "Data mining for very busy people," in *IEEE Computer*, November 2003, available from http://menzies.us/pdf/03tar2.pdf.

[22] E. Chiang and T. Menzies, "Position paper: Summary of simulations for very early lifecycle quality evaluations," in *Prosim '03*, 2003, available from http://menzies.us/pdf/03prosim.pdf.

[23] S. L. Cornford, M. S. Feather, J. Dunphy, J. Salcedo, and T. Menzies, "Optimizing spacecraft design optimization engine development: Progress and plans," in *Proceedings of the IEEE Aerospace Conference, Big Sky, Montana*, 2003, available from http://menzies.us/pdf/03aero.pdf.

[24] E. Chiang and T. Menzies, "Simulations for very early lifecycle quality evaluations," *Software Process: Improvement and Practice*, vol. 7, no. 3-4, pp. 141–159, 2003, available from http://menzies.us/pdf/03spip.pdf.

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

[26] T. Menzies, J. Kiper, and M. Feather, "Improved software engineering decision support through automatic argument reduction tools," in *SEDECS'2003: the 2nd International Workshop on Software Engineer-*

ing Decision Support (part of SEKE2003), June 2003, available from http://menzies.us/pdf/03star1.pdf.

[27] T. Menzies and J. S. D. Stefano, "How good is your blind spot sampling policy?" in *2004 IEEE Conference on High Assurance Software Engineering*, 2003, available from http://menzies.us/pdf/03blind.pdf.

[28] D. Geletko and T. Menzies, "Model-based software testing via incremental treatment learning," in *28th Annual NASA Goddard Software Engineering Workshop (SEW'03)*, December 2003.

[29] T. Menzies, S. Setamanit, and D. Raffo, "Data mining from process models," in *PROSIM 2004*, 2004, available from http://menzies.us/pdf/04dmpm.pdf.

[30] T. Menzies, J. S. D. Stefano, C. Cunanan, and R. M. Chapman, "Mining repositories to assist in project planning and resource allocation," in *International Workshop on Mining Software Repositories*, 2004, available from http://menzies.us/pdf/04msrdefects.pdf.

[31] T. Menzies, J. DiStefano, A. Orrego, and R. Chapman, "Assessing predictors of software defects," in *Proceedings, workshop on Predictive Software Models, Chicago*, 2004, available from http://menzies.us/pdf/04psm.pdf.

[32] T. Menzies, D. Port, Z. Chen, J. Hihn, and S. Stukes, "Validation methods for calibrating software effort models," in *Proceedings, ICSE*, 2005, available from http://menzies.us/pdf/04coconut.pdf.

[33] T. Menzies, Z. Chen, D. Port, and J. Hihn, "Simple software cost estimation: Safe or unsafe?" in *Proceedings, PROMISE workshop, ICSE 2005*, 2005, available from http://menzies.us/pdf/05safewhen.pdf.

[34] Z. Chen, T. Menzies, D. Port, and B. Boehm, "Finding the right data for software cost modeling," *IEEE Software*, Nov 2005.

[35] T. Menzies, D. Port, Z. Chen, J. Hihn, and S. Stukes, "Specialization and extrapolation of induced domain models: Case studies in software effort estimation," 2005, iEEE ASE, 2005, Available from http://menzies.us/pdf/05learncost.pdf.

[36] T. Menzies and J. Richardson, "Making sense of requirements, sooner," *IEEE Computer*, October 2006, available from http://menzies.us/pdf/06qrre.pdf.

[37] T. Menzies, Z. Chen, J. Hihn, and K. Lum, "Selecting best practices for effort estimation," *IEEE Transactions on Software Engineering*, November 2006, available from http://menzies.us/pdf/06coseekmo.pdf.

[38] T. Menzies and J. Hihn, "Evidence-based cost estimation for better quality software," *IEEE Software*, July/August 2006, available on-line at http://menzies.us/pdf/06costs.pdf.

[39] T. Menies, K. Lum, and J. Hihn, "The deviance problem in effort estimation," 2006, available from http://menzies.us/06deviations.pdf.

[40] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, January 2007, available from http://menzies.us/pdf/06learnPredict.pdf.

[41] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with precision," *IEEE Transactions on Software Engineering*, September 2007, http://menzies.us/pdf/07precision.pdf.

[42] T. Menzies, O. Elwaras, J. Hihn, F. n. B. B. M, and R. Madachy, "The business case for automated software engineerng," in *IEEE ASE*, 2007, available from http://menzies.us/pdf/07casease-v0.pdf.

[43] S. Ramakrishnan and T. Menzies, "An ongoing experiment in o-o software process and product measurements," in *Proceedings SEEP'96, New Zealand*, 1996.

[44] S. Ramakrishnan, T. Menzies, M. Hasslinger, P. Bok, H. McCarthy, B. Devakadadcham, and D. Moulder, "On building an effective measurement system for oo software process, product and resource tracking," in *Tools Pacific, 1996*, 1996.

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

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

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

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

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

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

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

[52] G.I.Webb, J. Boughton, and Z. Wang, "Not so naive bayes: Aggregating one-dependence estimators," *Machine Learning*, vol. 58, no. 1, pp. 5–24, 2005, available from http://www.csse.monash.edu.au/~webb/Files/WebbBoughtonWang05.pdf.

[53] C. Drummond and R. C. Holte, "C4.5, class imbalance, and cost sensitivity: why under-sampling beats over-sampling," in *Workshop on Learning from Imbalanced Datasets II*, 2003.

[54] L. Breiman, "Random forests," *Machine Learning*, pp. 5–32, October 2001.

[55] W. Cohen, "Fast effective rule induction," in *ICML'95*, 1995, pp. 115–123, available on-line from http://www.cs.cmu.edu/~wcohen/postscript/ml-95-ripper.ps.

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

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

[58] L. Brieman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

[59] Y. Freund and R. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *JCSS: Journal of Computer and System Sciences*, vol. 55, 1997.

[60] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics*, vol. 1, pp. 80–83, 1945.

[61] T. Menzies, B. Turhan, A. Bener, and J. Distefano, "Cross- vs within-company defect prediction studies," Computer Science, West Virginia University, Tech. Rep., 2007, available from http://menzies.us/pdf/07ccwc.pdf.

[62] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Ann. Math. Statist.*, vol. 18, no. 1, pp. 50–60, 1947, available on-line at http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&handle=euclid.aoms/1177730491.

[63] A. Orrego, "Sawtooth: Learning from huge amounts of data," Master's thesis, Computer Science, West Virginia University, 2004.

[64] J. R. Quinlan, "Learning with Continuous Classes," in *5th Australian Joint Conference on Artificial Intelligence*, 1992, pp. 343–348, available from http://citeseer.nj.nec.com/quinlan92learning.html.

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

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