Implications Of Ceiling Effects In Defect Predictors

Tim Menzies[a], Burak Turhan[b], Ayse Bener[b], Gregory Gay[a], Bojan Cukic[a], Yue Jiang[a] *

[a] Dept of CS& EE, West Virginia University, Morgantown, WV, USA
[b] Dept. of Computer Engineering, Bogazici University, Turkey
tim@menzies.us; turhanb@boun.edu.tr; bener@boun.edu.tr
greg@4colorrebellion.com; yue, cukic@csee.wvu.edu

ABSTRACT

Context: There are many methods that input static code features and output a predictor for faulty code modules. These data mining 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.

Objective: We seek an explanation for this ceiling effect. Perhaps static code features have "limited information content"; i.e. their information can be quickly and completely discovered by even simple learners.

Method: An initial literature review documents the ceiling effect in other work. Next, using three sub-sampling techniques (under-, over-, and micro-sampling), we look for the lower useful bound on the number of training instances.

Results: Using micro-sampling, we find that as few as 50 instances yield as much information as larger training sets.

Conclusions: We have found much evidence for the limited information hypothesis. Further progress in learning defect predictors may not come from better algorithms. Rather, we need to be improving the information content of the training data, perhaps with case-based reasoning methods.

Categories and Subject Descriptors

i.5 [**learning**]: machine learning; d.2.8 [**software engineering**]: product metrics

General Terms

algorithms,experimentation, measurement

Keywords

Naive Bayes,under-sampling, over-sampling,defect prediction

*The research described in this paper was supported by Bogazici 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.

PROMISE'08, May 12–13, 2008, Leipzig, Germany.

1. INTRODUCTION

The current state of the art in learning defect predictors is curiously static. As shown below, better results have not been forthcoming, despite the application of supposedly better data miners.

If better algorithms are not useful, perhaps it is time to better understand the training data. Accordingly, instead of:

- Changing the *data miners*...
- .... this paper changes the *training data*.

Specifically, we study how *throwing training data away* effects the performance of our defect predictors. The results are quite surprising. In experiments with under/over-sampling and incremental cross-validation, we found that *much of the training data can be discarded without losing effectiveness in defect prediction.* This leads to the following notion:

Hypothesis: Static code features have *limited information content*.

This, in turn, leads to three predictions:

Prediction1: The information from static code features can be quickly and completely discovered by even simple learners.

Prediction2: More complex learners will not find new information.

Prediction3: Further progress in learning defect predictors will not come from better algorithms, but from improving the information content of the training data.

We provide empirical evidence for and discuss the validity of **Prediction1** and **Prediction2**. It turns out that a simple learner like Naive Bayes can extract the embedded information in static code features better than a number of sophisticated learners, including more complex Bayesian learners. **Prediction3** can be used to greatly simplify the collection of more insightful training data. We show that simple techniques like log filtering and feature weighting can improve the prediction performance. The final discussion section of this paper discusses methods for increasing the information in the training data in detail.

2. BACKGROUND

For some time now, we have applied data miners to build defect predictors from static code measures [4, 8, 20, 27–31, 33–37]. To learn such predictors, tables of historical examples (like those in Figure 1) 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 [18], or (iii) max. number of possible execution pathways [26].

data	language	(# modules) examples	features	%defective
pc5	C++	17,186	38	3.0
mc1	C++	9,466	38	0.71
pc2	C++	5,589	36	0.41
kc1	C++	2,109	21	15.45
pc3	C++	1,563	37	10.23
pc4	C	1,458	37	12.2
pc1	C++	1,109	21	6.94
kc2	C++	522	21	20.49
cm1	C++	498	21	9.83
kc3	JAVA	458	39	9.38
mw1	C++	403	37	7.69
mc2	C++	61	39	32.29
		40,422		

Figure 1: Twelve tables of data, sorted in order of number of examples. All these tables are in the PROMISE repository.

If $\{A, B, C, D\}$ are the true negatives, false negatives, false positives, and true positives (respectively) found by a defect predictor, then:

$$pd = \quad recall = \quad D/(B+D) \tag{1}$$

$$pf = \quad C/(A+C) \tag{2}$$

$$bal = \quad balance = \quad 1 - \frac{\sqrt{(0-pf)^2 + (1-pd)^2}}{\sqrt{2}} \tag{3}$$

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 1, 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 [33, 35]).

Figure 2: Performance measures.

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 need the most attention. Note that these data miners do not predict the total number of defects, just the number of modules containing more than zero defects.

In theory, such defect predictors are not useful. Fenton offers an insightful example where *the same* functionality is achieved using *different* programming language constructs resulting in *different* static measurements for that module [13].

In practice, they are quite effective, at least for the NASA code we have studied [20, 28, 30, 31, 33–37, 41–43]. In those data sets, our fault prediction models find defect predictors [35] with a probability of detection (pd) and probability of false alarm (pf) of $mean(pd, pf) = (71\%, 25\%)$. These values are much higher then known industrial averages for manual defect detection [11, 40].

In January 2007, we published a study [35] that defined 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 used:

- Public domain data sets (from the the PROMISE repository;
- Open source data mining tools (the WEKA toolkit [44]);
- Randomly sorting training data rows (stops order effects);
- 10-way cross-validation (to test on 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 $\{pd, pf\}$ ($balance$ is defined in Figure 2);
- 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;

Surprisingly, very simple Bayes classifiers (with a simple preprocessor for the numerics) out-performed the other studied methods. 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

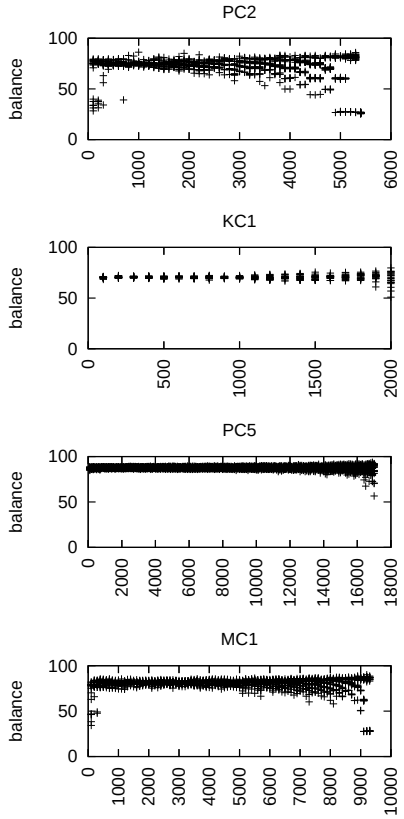

Figure 3: Experiments with training set size vs $balance$.

data mining methods: logistic regression; average one-dependence estimators [15]; under- or over-sampling [10] random forests [2], RIPPER [6], J48 [39], OneR [19] and bagging [3]. Only boosting [14] 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. Due to our connection with PROMISE, we are aware of studies by other researchers (currently under review) that tried other data mining methods. Those studies investigated the statistical difference of the

results from two dozen learners on the same datasets. The simple 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 lesson from the above is that exploring better algorithms may not be productive. Hence, we explored the data that the algorithms were processing.

Elsewhere [32] we checked how *little* information was required to learn a defect predictor. Defect predictors were learned from $N = 100, N = 200, N = 300$.. instances then $Test$ed on another 100 instances. For each N, 10 experiments were performed where training was conducted on $|Train| = 90\% * N$ instances and testing on $|Test| = 100$ (and $Train \cap Test =$). For all experiments, the $N, Train, Test$ instances were selected at random.

This study was conducted on the twelve data sets of Figure 1. Space does not permit showing all the results but a representative sample are shown in Figure 3 [32]. In that figure, the X-axis is the size of training set and the Y-axis is the $balance$ measure defined in Figure 2. 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 was actually detrimental (witness the widening variance as the training set increases). A Mann Whitney U test [25] (95% confidence) confirms the visual pattern apparent in Figure 3: 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.

The above results motivated new experiments, reported below.

3. EXPERIMENTS

The goal of these experiments was two-fold:

1. Can we confirm the effects of Figure 3; i.e. that ignoring large amounts of the training data is not harmful?
2. Can we exploit that effect to build better defect predictors?

Goal #1 was achieved and we have a promising lead on Goal #2.

3.1 Experiment #1: Over- & Under- Sampling

The Figure 3 experiment randomly discarded training data. Perhaps a *more controlled sub-sampling method* would not damage the information content of the data. If so then, contrary to Figure 3, increasing the sample size will improve performance.

Over- and under-sampling [5, 10] are examples of "more controlled sub-sampling methods". Both methods might be useful in data sets with highly imbalances class frequencies. For example, in Figure 1, the frequency of the target class is very low (median value lies between 6.94% and 7.69%).

To sub-sample, a target class is selected; in this study, we selected defective modules as the target class. The sampling program runs in two passes through the data. In pass 1, 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 selected 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 the desired goal is no longer buried inside a larger set of other classes.
- *Over-sampling* follows a similar procedure but, instead of limiting the number of instances with other classes, the sampling program adds randomly selected examples of the target class back into the data set. Where as under-sampling produces smaller data sets, over-sampling grows the size of the data set until the minority class has the same frequency as the majority class.

	quartiles				
	min		median		max
treatment	0	25%	50%	75%	100%
NB/ none	21.9	67.7	74.6	81.9	100.0
NB/ under	19.9	67.1	74.1	81.6	100.0
J48/ under	21.6	64.8	73.6	82.6	100.0
NB/ over	17.5	42.0	62.5	72.2	100.0
J48/ over	0.0	29.3	45.6	56.2	100.0
J48/ none	0.0	29.3	42.3	54.5	100.0

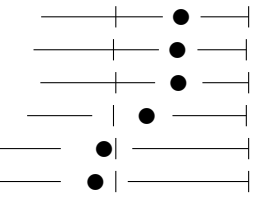

Figure 4: Over- & under- & no sampling results. Sorted descending by median $balance$ results. Visually, there is clear loser (J48/none) and a three-way tie for the winning method (NB/none, NB/under, J48/under). These visual impression are confirmed by the statistical tests of Figure 5.

Regardless of the which of sub-sampling method is used, the result is a data set with an equal number of target and non-target classes.

Over- and under-sampling experiments were conducted on the Figure 1 data using 10*10-way cross-validation[1]. This study used a simple Bayes classifier (since it was useful in our prior experiment [35]) plus a C4.5-like decision tree learner, J4.8 [44] (since that was used in prior under- and over- sampling experiments [10]).

The $balance$ results (defined in Figure 2 for each $treatment$ (data miner, plus sampling policy) were sorted and displayed as *quartile charts* of Figure 4. To generate these charts, the $balance$ results for some treatment are sorted and labeled as follows:

$$\overbrace{\underbrace{4}_{min}, 7, 15}^{q1}, 20, 31, \underbrace{40}_{median}, 52, 64, \overbrace{70, 81, \underbrace{90}_{max}}^{q4}$$

In a quartile chart, the upper and lower quartiles are marked with black lines; the median is marked with a black dot; and a vertical bar is added to mark the 50% value. The above numbers would therefore be drawn as follows:

$$0\% \,|\!-\!\!\bullet\!|\!-\!| \, 100\%$$

We prefer quartile charts of performance deltas to other summarization methods for M*N studies. They offer a very succinct summary of a large number of experiments.

The Figure 4 results are consistent with certain prior results:

- The simple Naive Bayes we recommended previously [35] performed as well as anything else. In a result consistent with the limited information hypothesis discussed in the introduction, seemingly cleverer learning schemes did not outperform simple Bayesian classifiers.
- Just like the Figure 3 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%.
- Under-sampling beat over-sampling for both J48 and Naive Bayes. This result is consistent with Drummond & Holte's

[1]Ten times, randomize the order of the instances in the data. Each time, divide data into i bins. For each bin $j \in 1..i$. Let $test$ be bin j and let $train$ be the remaining bins. Learn on $train$ then evaluate on $test$.

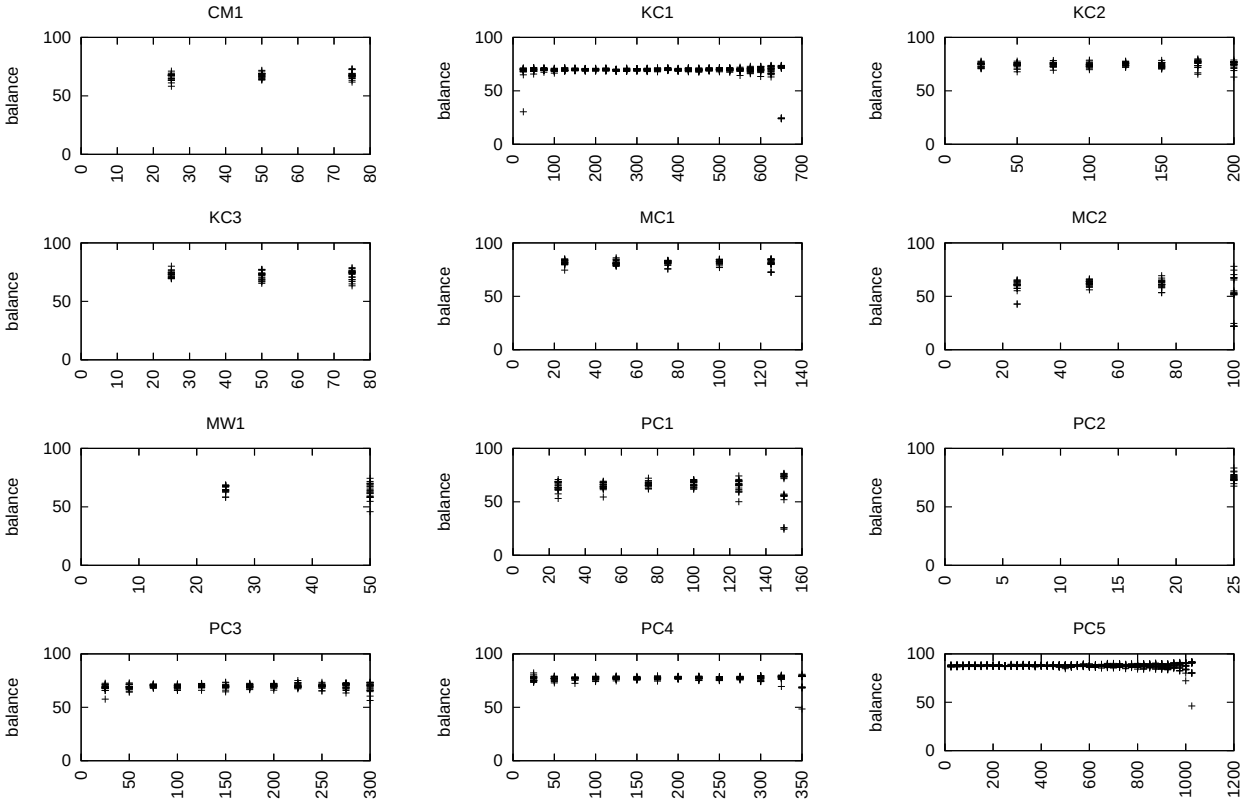

Figure 6: Micro-sampling results

learner / sampler	win - loss	win	loss	ties
NB / none	3	3	0	2
NB / under	3	3	0	2
J48 / under	3	3	0	2
NB / over	-1	2	3	0
J48 / over	-3	1	4	0
J48 / none	-5	0	5	0

Figure 5: Statistical tests on the Figure 4 results. Mann-Whitney [25] compared the median ranks of all the $balance$ values seen in Figure 4. Two treatments tied if their median ranks were statistically insignificantly different (95% confidence). Otherwise, medians were compared numerically to assign "win" or "loss". Table sorted descending on total number of $win - loss$ for each treatment against the other five.

sub-sampling experiments [10] and the sub-sampling classification tree experiments of Kamei et.al. [21][2].

However, Figure 4 has some new results which, as far as we are aware, have not been reported elsewhere:

- Observe how $NB/none$ is one of the topped-ranked methods. That is, sub-sampling decision tree learning does not out-perform Naive Bayes.
- $NB/none$ ties with $NB/under$. That is, while sub-sampling offers no improvement over un-sampled Bayesian learning, under-sampling does not harm classifier performance,

[2]Due to differences in experimental methods, we find we cannot compare our results to the regression tree and LDA analysis of [21].

This last point is the most significant. It means effective detectors can be learned from a very small sample of the available data- an issue we explore below.

3.2 Experiment #2: Micro-sampling

In order to determine the lower-limit on the number of cases that require manual inspection, we performed another experiment. Another under-sampling policy was employed, which we call *micro-sampling*. Given N defective modules in a data set,

$$M \in \{25, 50, 75, .\} \leq N$$

defective modules were selected at random. Another M non-defective modules were selected, at random. The combined $2M$ data set was then passed to a 10*10-way cross validation.

Formally, under-sampling is a micro-sampling where $M = N$. Micro-sampling explores training sets of size up to N, standard under-sampling just explores once data set of size $2N$.

For this study we used Naive Bayes since, in all the above work, it seems to be doing as well as anything else. Figure 6 shows the results of an under-sampling study where $M \in \{25, 50, 75, ...\}$ 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 this visual pattern. Detectors learned from small M instances do as well as detectors learned from any other number of instances.

- For eight data sets, {CM1,KC2, KC3,MC1, MC2, MW1, PC1, PC2}, micro-sampling at $M = 25$ did just as well as anything larger sample size.

- For one data sets, {PC3}, best results were seen at $M = 75$. However , in many cases $\frac{8}{11}$, $M = 25$ did just as well as anything else.
- For three data sets {PC4,KC1,PC5} best results were seen at $M = \{200, 575, 1025\}$, respectively. However, for these three data sets, in all by one case $M = 25$ did as well as any larger value.

In summary, the number of cases that must be reviewed in order to arrive at the performance ceiling of defect predictor is very small: as low as 50 randomly selected modules (25 defective and 25 non-defective).

4. DISCUSSION

There exist numerous incremental case-based reasoning tools [7, 23, 24] that ask humans to audit a stochastic sample of real-world cases. Insights gained from those sessions are automatically generalized and applied to another random sample, Experts then review the classifications made on the new sample, and offer further refinements. As far back as 1999, software metrics experts Fenton and Neil [12] postulated that such human-machines-based system might out-perform systems based on on static code measures (since other features/metrics could be accounted for that cannot currently be addressed using static code metrics).

When is case-based reasoning preferable to automatic data mining? While there are many answers to this question, this paper can make specific comments about one particular issue. Case-based reasoning methods require humans to examine and comment on specific cases. This is impractical if learning adequate theories requires examining a very large number of cases.

The results of experiment #1 suggest that, for static code measures, it is not necessary to manually inspect thousands of cases. In fact, just a few hundred may suffice. Consider the 9,466 modules of MC1. This data set has a defective module rate of 0.71% ; i.e. 9.466*0.71/100=67 modules:

- When under-sampling, a data set of 134 modules is created (all the defective, plus 67 others, selected at random).
- When micro-sampling, the results of Experiment #2 suggest that 50 modules would suffice (any 25 defective modules, but any other 25- selected at random).

These results raise the possibility that a human-in-the-loop case-based reasoning environment might *perform as well as* automatic methods, despite the automatic methods exploring more examples.

Based on other recent research, we go further and hypothesize that such an environment might *perform better* than automatic methods. Elsewhere, we have explored combining static code measures with other measures that, serendipitously, a particular domain may contain. For example, at ISSRE 2007 [20], we reported experiments where static code measures were combined with results from an ultra-lightweight text miner. Figure 7 shows how a remarkable improvement in learner performance was achieved by applying *combinations* of requirements and code features. Other results are analogous to Figure 7:

- In the JMLR special issue on feature selection, Guyon and Elisseeff provide simple examples showing that the information content of data can be significantly increased when features are used together rather than individually [17]. In the context of this paper, we would re-express that result as: *using features from different sources, e.g. requirements and source code, can significantly increase the information content of SE data.*

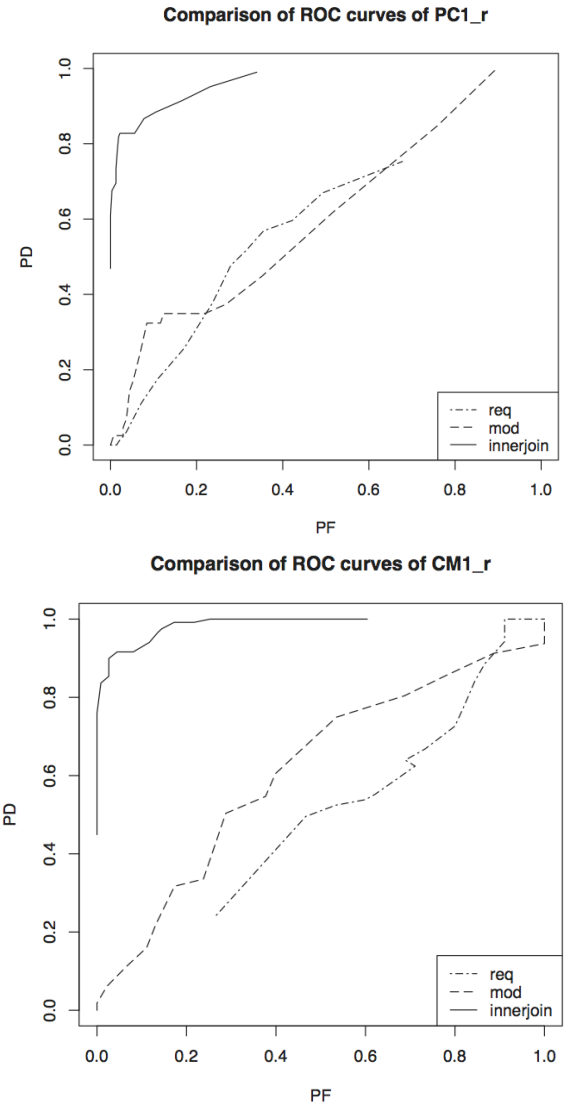

Figure 7: $\{pd, pf\}$ curves seen when using code and/or requirements features. Results from CM1 and PC1 after a 10-way cross validation. The ideal spot on these ROC curves is top left; i.e. no false alarms and perfect detection ($\{pd, pf\} = \{1, 0\}$). The dashed lines on those plots show $\{pd, pf\}$ results when fault prediction models used features mined from requirements text, or features mined from static code measures (in isolation). The solid lines show the results of models which used these two kinds of features in combination. From [20].

- Face recognition differentiates from other image processing research by using facial properties such as the location of the nose, lips, eyes etc. Combination of facial characteristics from such different sources improves the performance of face detectors that solely use i.e. pixel values of the image [16]. *We argue that SE domain should make use of domain specific knowledge (when possible) to differentiate from general data mining tasks.*

Note that the lesson of Figure 7 is not "always augment static code features with requirements features". This is impractical ad-

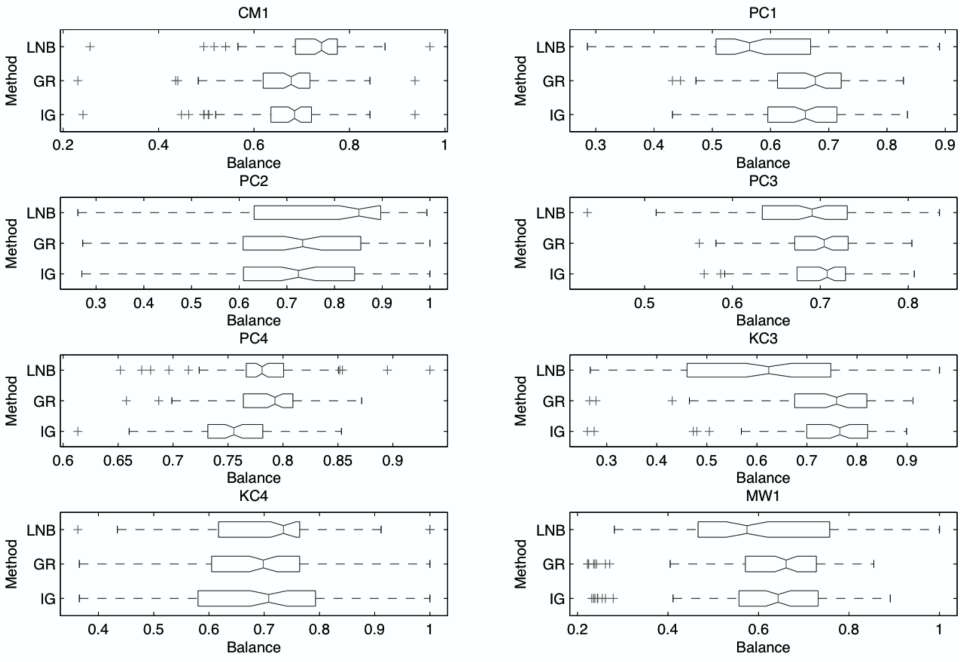

Figure 8: Comparison of three Naive Bayes schemes. LNB is the log-filtered Naive Bayes used in our IEEE TSE 2007 paper [35]. IG and GR are two variants on Naive Bayes where some oracle has weighted the attributes: IG uses an InfoGain oracle and GR uses a Gain Ratio oracle. Box plots for three learners after 10x10 cross validation. Balance values can be improved by weighting static code features. Feature subset selection corresponds to $0-1$ *hard* weight assignment. Weights of non-informative features are automatically assigned 0. Other features are assigned a *soft* weight value in $[0..1]$ interval according to their defect related information content. Note that, in $\frac{7}{8}$ of these results, the weighted schemes have higher medians or higher maximum values, or both.

vice since not all domains allow data mining specialists access to source code and the requirements that produced them; e.g. very few open source projects have detailed textual requirements documents.

Rather, the lesson of Figure 7 is that it can be very useful to let experts access and combine features from whatever sources are locally available. Such an "explore whatever" environment is not an automatic black box data miner. Rather, it is a human-in-the-loop case-based reasoning (CBR) environment where humans reflect on the specifics of particular cases, connect to different data sources, and (sometimes) run automatic data miners on combinations or subsets of a variety of types of features.

Two prior PROMISE papers [1, 9] suggest how such CBR environments might operate:

- At PROMISE 2006, Amor et.al. [1] describe an incremental classification cycle where, in round i, an expert classifies a random selection of the available data. A classifier learned from this examples is then applied to other, randomly selected examples. Then, in round $i + 1$, the expert reviews the results of that those classifications to fix any incorrect classifications from round i (formally, this is boosting where humans are used to identify examples that were harder to classify in the proceeding rounds).
- At PROMISE 2007, Dekhtyar et.al. [9] showed improvement in the performance of a traceability environment after several rounds of an expert reviewing results automatically inferred in a prior round.

Both Amor et.al. and Dekhtyar et.al. report that it took hundreds of examples and several rounds to generate useful detectors. Neither team experimented with (a) decreasing the number of examples seen in each round; or (b) micro-sampling the examples from each round. Based on this paper, we would speculate that policies (a) and (b) would significantly decrease the time required by an expert to achieve performance plateaus.

Another role for human experts in a CBR environment is to instruct the learner how *combinations* of attributes can work together to provide solutions. For example a standard Naive Bayes classifier gives equal weights to all attributes then uses frequency counts to learn the relative importance of each attribute. Elsewhere [42, 43] we have allowed an oracle to offer unequal attribute weights. In this scheme, instead of feature subset selection [22], we have used all attributes with weights assigned to them. We have converted the information gain and gain ratio of features into weight values. The results are compelling such that the performance can improve over standard Naive Bayes, and there is also no need for dealing with feature subset selection. Though the improvements in Figure 8 are not ground-breaking, they provide a hint regarding the value of unequal treatment of information sources. In the case of Figure 8, weights are assigned by the model. As Fenton and Neil warn us, the weights provided by models may not be meaningful to humans in the process [12]. However, we argue that weights assigned to different information sources by human experts *with business knowledge* can increase the quality of solutions.

5. CONCLUSIONS

The development of fault prediction models has been a very active research area. The reason for such a significant attention to automated quality predictors lays in their practical importance. Cur-

rent models are useful, as they allow software project managers to better guide the allocation usually meager quality assurance resources to artifacts which need them the most.

Recent results now indicate that this current research paradigm, which relied on relatively straightforward application of machine learning tools, has reached its limits. Building software quality predictors via data mining is essentially an inductive generalization over past experience. According to Mitchell's classic model of data mining [38], any inductive generalization explores a version space of possible theories. All data miners hit a performance ceiling effect when they cannot find additional information that better relates software measure with fault occurrence.

To build better quality predictors that break through ceiling effects, we must introduce more topology into the search space. Standard machine learning algorithms lack the business knowledge which characterizes software projects. To add that business knowledge, we propose human-in-the-loop CBR tools. We expect that this approach will allow software managers to safely focus on the application of quality assurance techniques of choice, confident that automated quality predictors will raise alerts about the artifacts where quality issues actually exist.

Three results make us advocate this kind of CBR tool:

- The environment would impractical if the human operator must examine a very large number of cases. The micro-sampling results suggest that this might not be the case. Indeed, as few as 50 randomly selected instances might suffice.
- The environment would run fast. If 50 instances are enough, then training should be virtually instantaneous. A human user could explore a very large number of features or combinations of features, all the while getting very rapid feedback.
- The environment, or some other new direction, is required. As argued above, our current generation of AI algorithms have hit a ceiling effect. We doubt that further progress will be made using better algorithms. Instead, we would advocate methods (like the CBR tool briefly described above), that increase the information content of the training data.

6. FUTURE WORK

In summary, we think it is time to change the subject of the *which* question. Rather than *which learner* we should focus on *which data*.

In this context, our future direction is clear- build a human-in-the-loop CBR environment for learning defect predictors, then benchmark that environment against automatic methods.

Other candidates for future work are

- To better understand the attribute weighting schemes explored in Figure 8.
- To improve on the randomized sampling strategies used in this paper. It may be possible to increase the performance of defect predictors with wiser sampling strategies (e.g. some initially clustering to find a small number of most representative, or most unusual, instances).

7. REFERENCES

[1] J. Amor, G. Robles, J. Gonzalez-Barahona, and A. Navarro. Discriminating development activities in versioning systems: A case study. In *Proceedings, PROMISE 2006*, 2006. Available from `http://promisedata.org/pdf/phil2006AmorRoblesGonzales_BarahonaNavarro.p%df`.

[2] L. Breiman. Random forests. *Machine Learning*, pages 5–32, October 2001.

[3] L. Brieman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[4] Zhihoa Chen, Tim Menzies, and Dan 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`.

[5] Alexander Yun chung Liu. The effect of oversampling and undersampling on classifying imbalanced text datasets. Master's thesis, 2004. Available from `http://www.lans.ece.utexas.edu/~aliu/papers/aliu_masters_thesis.pdf`.

[6] W.W. Cohen. Fast effective rule induction. In *ICML'95*, pages 115–123, 1995. Available on-line from `http://www.cs.cmu.edu/~wcohen/postscript/ml-95-ripper.ps`.

[7] P. Compton, L. Peters, G. Edwards, and T.G. Lavers. Experience with ripple-down rules. *Knowledge-Based Systems*, 19(5):356–362, September 2006. Available from `http://www.cse.unsw.edu.au/~compton/#Starter_Papers`.

[8] A. Dekhtyar, J. Huffman 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`.

[9] A. Dekhtyar, J.H. Hayes, and J. Larsen. Make the most of your time: How should the analyst work with automated traceability tools? In *3rd International Workshop on Predictive Modeling in Software Engineering (PROMISE'2007)*, 2007.

[10] 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.

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

[12] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999. Available from `http://citeseer.nj.nec.com/fenton99critique.html`.

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

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

[15] G.I.Webb, J. Boughton, and Z. Wang. Not so naive bayes: Aggregating one-dependence estimators. *Machine Learning*, 58(1):5–24, 2005. Available from `http://www.csse.monash.edu.au/~webb/Files/WebbBoughtonWang05.pdf`.

[16] B. Gokberk, M. O. Irfanoglu, L. Akarun, and E. Alpaydin. Learning the best subset of local features for face recognition. *Pattern Recogn.*, 40(5):1520–1532, 2007.

[17] I. Guyon and A. Elisseeff. An introduction to variable and feature selection'. *Journal of Machine Learning Research*, pages 1150–1182, March 2003.

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

[19] R.C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11:63, 1993.

[20] 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.

[21] Yasutaka Kamei, Akito Monden, Shinsuke Matsumoto, Takeshi Kakimoto, and Ken-ichi Matsumoto. The effects of over and under sampling on fault-prone module detection. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 196–204, 20-21 Sept. 2007.

[22] Ron Kohavi and George H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324, 1997.

[23] J. Kolodner. *Case-Based Reasoning*. Morgan Kaufmann, 1993.

[24] J.L. Kolodner. Improving Human Decision Making Through Case-Based Decision Aiding. *AI Magazine*, page 68, Summer 1991.

[25] 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.*, 18(1):50–60, 1947. Available on-line at http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&hand%le=euclid.aoms/1177730491.

[26] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[27] 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.

[28] 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.

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

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

[31] T. Menzies, Justin S. Di Stefano, Chris Cunanan, and Robert (Mike) 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.

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

[33] Tim Menzies, Alex Dekhtyar, Justin Distefano, and Jeremy Greenwald. Problems with precision. *IEEE Transactions on Software Engineering*, September 2007. http://menzies.us/pdf/07precision.pdf.

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

[35] Tim Menzies, Jeremy Greenwald, and Art 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.

[36] Tim Menzies, Robyn Lutz, and Carmen Mikulski. Better analysis of defect data at NASA. In *SEKE03*, 2003. Available from http://menzies.us/pdf/03superodc.pdf.

[37] Tim Menzies and Justin S. Di Stefano. How good is your blind spot sampling policy? In *2004 IEEE Conference on High Assurance Software Engineering*, 2003. Available from http://menzies.us/pdf/03blind.pdf.

[38] T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

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

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

[41] J.S. Di 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.

[42] B. Turhan and A. Bener. A multivariate analysis of static code attributes for defect prediction. *qsic*, 0:231–237, 2007.

[43] B. Turhan and A. Bener. Software defect prediction: Heuristics for weighted naive bayes. In *Second International Conference in Software and Data Technologies (ICSOFT 2007)*, pages 244–249, 2007.

[44] Ian H. Witten and Eibe Frank. *Data mining. 2nd edition*. Morgan Kaufmann, Los Altos, US, 2005.