Defect Prediction from Static Code Features: Current Results, Limitations, New Approaches

TIM MENZIES, ZACH MILTON
West Virginia University
BURAK TURHAN
NRC
BOJAN CUKIC, YUE JIANG
West Virginia University
AYŞE BENER
Boğaziçi University

Building quality software is expensive and software quality assurance (QA) budgets are limited. Data miners can learn defect predictors from static code features which can be used to control QA resources; e.g. to focus on the parts of the code predicted to be more defective.

Recent results show that better data mining technology is not leading to better defect predictors. We hypothesize that we have reached the limits of the standard learning goal of maximizing area under the curve (AUC) of the probability of false alarms and probability of detection "AUC(pd,pf)"; i.e. the area under the curve of a probability of false alarm versus probability of detection.

Accordingly, we explore changing the standard goal. Learners that maximize "AUC(effort,pd)" find the *smallest* set of modules that contain the *most* errors. WHICH is a meta-learner framework that can be quickly customized to different goals. When customized to AUC(effort,pd), WHICH out-performs all the data mining methods studied here. More importantly, measured in terms of this new goal, certain widely used learners perform *much worse* than simple manual methods.

Hence, we advise against the indiscriminate use of learners. Learners must be chosen and customized to the goal at hand. With the right architecture (e.g. WHICH), tuning a learner to specific local business goals can be a simple task.

Categories and Subject Descriptors: D.2.8 [**Software Engineering**]: Metrics—*Product metrics*; *Complexity metrics*; U.2.8 [**Computer Methodologies**]: Learning

General Terms: Algorithms, Measurement

Additional Key Words and Phrases: defect prediction, static code features, WHICH

1. INTRODUCTION

A repeated result is that static code features such as lines of code per module, number of symbols in the module, etc., can be used by a data miner to predict

Corresponding author: Tim Menzies. Author contacts: tim@menzies.us, zmilton@mix.wvu.edu, bojan.cukic@mail.csee.wvu.edu, yjiang1@mix.wvu.edu, burak.turhan@nrc-cnrc.gc.ca, bener@boun.edu.tr. This research was supported by NSF grant CCF-0810879 and the Turkish Scientific Research Council (Tubitak EEEAG 108E014). For an earlier draft, see http://menzies.us/pdf/08bias.pdf.

which modules are more likely to contain defects[1]. Such defect predictors can be used to allocate the appropriate verification and validation budget assigned to different code modules.

The current high water mark in this field has been curiously static for several years. For example, for three years we have been unable to improve on our 2006 results [Menzies et al. 2007]. Other studies report the same *ceiling effect*: many methods learn defect predictors that perform statistically insignificantly different to the best results. For example, after a careful study of 19 data miners for learning defect predictors seeking to maximize the area under the curve of detection-vs-false alarm curve, Lessmann et al. [2008] conclude

> *...the importance of the classification model is less than generally assumed ... practitioners are free to choose from a broad set of models when building defect predictors.*

This article argues for a very different conclusion. The results of Lessmann et al. are certainly correct for the goal of maximizing detection and minimizing false alarm rates. However, this is not the only possible goal of a defect predictor. WHICH [Milton 2008] is a meta-learning scheme where domain specific goals can be inserted into the core of the learner. When those goals are set to one particular business goal (e.g. "finding the *fewest* modules that contain the *most* errors") then the ceiling effect disappears:

—WHICH significantly out-performs other learning schemes.
—More importantly, certain widely used learners perform *worse than simple manual methods*.

That is, contrary to the views of Lessmann et al, the selection of a learning method appropriate to a particular goal is very critical. Learners that appear useful when pursuing certain goals, can be demonstrably inferior when pursuing others. We recommend WHICH as a simple method to create such customizations.

The rest of this paper is structured as follows. §2 describes the use of static code features for learning defect predictors. §3 documents the ceiling effect that has stalled progress in this field. After that, §IV and §V discuss a novel method to break through the ceiling effect.

2. BACKGROUND

This section motivates the use of data mining for static code features and reviews recent results. The rest of the paper will discuss limits with this approach, and how to overcome them.

2.1 Blind Spots

Our premise is that building high quality software is expensive. Hence, during development, developers *skew* their limited quality assurance (QA) budgets towards

[1]e.g. [Weyuker et al. 2008; Halstead 1977; McCabe 1976; Chapman and Solomon 2002; Nagappan and Ball 2005a; Hall and Munson 2000; Nikora and Munson 2003; Nagappan and Ball 2005b; Khoshgoftaar 2001; Tang and Khoshgoftaar 2004; Khoshgoftaar and Seliya 2003; Porter and Selby 1990; Tian and Zelkowitz 1995; Khoshgoftaar and Allen 2001; Srinivasan and Fisher 1995]

artifacts they believe most require extra QA. For example, it is common at NASA to focus QA more on the on-board guidance system than the ground-based database which stores scientific data collected from a satellite.

This skewing process can introduce an inappropriate bias to quality assurance (QA). If the QA activities concentrate on project artifacts, say *A,B,C,D*, then that leaves *blind spots* in *E,F,G,H,I,....* Blind spots can compromise high assurance software. Leveson remarks that in modern complex systems, unsafe operations often result from an unstudied interaction between components [Leveson 1995]. For example, Lutz and Mikulski [Lutz and Mikulski 2003] found a blind spot in NASA deep-space missions: most of the mission critical *in-flight* anomalies resulted from errors in *ground software* that fails to correctly collect in-flight data.

To avoid blind spots, one option is to rigorously assess all aspects of all software modules, however, this is impractical. Software project budgets are finite and QA effectiveness increases with QA effort. A *linear* increase in the confidence C that we have found all faults can take *exponentially* more effort. For example, to detect one-in-a-thousand module faults, moving C from 90% to 94% to 98% takes 2301, 2812, and 3910 black box tests (respectively)[2]. Lowry et.al. [Lowry et al. 1998] and Menzies and Cukic [Menzies and Cukic 2000] offer numerous other examples where assessment effectiveness is exponential on effort.

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. We endorse this approach. Clearly, the most critical sections require the best known assessment methods, in hope of minimizing the risk of safety or mission critical failure occurring post deployment. 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.

2.2 Lightweight Sampling

2.2.1 *Data Mining.* One method for building a lightweight sampling policy is *data mining* over *static code features*. For this paper, we define *data mining* as the process of summarizing tables of data where rows are *examples* and columns are the *features* collected for each example[3] One special feature is called the *class*. The appendix to this paper describes various kinds of data miners including:

[2]A randomly selected input to a program will find a fault with probability x. Voas observes [Voas and Miller 1995] that after N random black-box tests, the chance of the inputs not revealing any fault is $(1-x)^N$. Hence, the chance C of seeing the fault is $1-(1-x)^N$ which can be rearranged to $N(C,x) = \frac{log(1-C)}{log(1-x)}$. For example, $N(0.90, 10^{-3}) = 2301$.

[3]Technically, this is *supervised learning* in the *absence of a background theory*. For notes on *unsupervised learning*, see papers discussing *clustering* such as [Bradley et al. 1998]. For notes on *using a background theory*, see (e.g.) papers discussing the learning or tuning of Bayes nets [Fenton and Neil 1999].

m = Mccabe		$v(g)$	cyclomatic_complexity
		$iv(G)$	design_complexity
		$ev(G)$	essential_complexity
locs	loc		loc_total (one line = one count
	loc(other)		loc_blank
			loc_code_and_comment
			loc_comments
			loc_executable
			number_of_lines (opening to closing brackets)
Halstead	h	N_1	num_operators
		N_2	num_operands
		μ_1	num_unique_operators
		μ_2	num_unique_operands
	H	N	length: $N = N_1 + N_2$
		V	volume: $V = N * log_2 \mu$
		L	level: $L = V^*/V$ where $V^* = (2 + \mu_2{}^*)log_2(2 + \mu_2{}^*)$
		D	difficulty: $D = 1/L$
		I	content: $I = \hat{L} * V$ where $\hat{L} = \frac{2}{\mu_1} * \frac{\mu_2}{N_2}$
		E	effort: $E = V/\hat{L}$
		B	error_est
		T	prog_time: $T = E/18$ seconds

Fig. 1. Static code features.

—Näive Bayes classifiers use statistical combinations of features to predict for class value. Such classifiers are called "naive" since they assume all the features are statistically independent. Nevertheless, a repeated empirical result is that, on average, seemingly näive Bayes classifiers perform as well as other seemingly more sophisticated schemes (e.g. see Table 1 in [Domingos and Pazzani 1997]).
—Rule learners like RIPPER [Cohen 1995a] generate lists of rules. When classifying a new code module, we take features extracted from that module and iterate over the rule list. The output classification is the first rule in the list whose condition is satisfied.
—Decision tree learners like C4.5 [Quinlan 1992b] build one single-parent tree whose internal nodes test for feature values and whose leaves refer to class ranges. The output of a decision tree is a branch of satisfied tests leading to a single leaf classification.

There are many alternatives and extensions to these learners. Much recent work has explored the value of building *forests* of decision trees using randomly selected subsets of the features [Breimann 2001; Jiang et al. 2008]. Regardless of the learning method, the output is the same: combinations of standard features that predict for different class values.

2.2.2 *Static Code Features.* Defect predictors can be learned from tables of data containing static code features, whose class label is *defective* and whose values are *true* or *false*. In those tables:

—Rows describe data from one *module*. Depending on the language, modules may be called "functions", "methods", "procedures" or "files".
—Columns describe one of the static code features of Figure 1. The appendix of this paper offers further details on these features.

These static code features are collected from prior development work. The *defective* class summarizes the results of a whole host of QA methods that were applied to that historical data. If any manual or automatic technique registered a problem with this module, then it was marked "defective=true". For these data sets, the data mining goal is to learn a binary prediction for $defective$ from past projects that can be applied to future projects.

This paper argues that such defect predictors are useful and describes a novel method for improving their performance. Just in case we overstate our case, it is important to note that defect predictors learned from static code features can only *augment*, but never *replace*, standard QA methods. Given a limited budget for QA, the manager's task is to decide which set of QA methods $M_1, M_2, ...$ that cost $C_1, C_2, ...$ should be applied. Sometimes, domain knowledge is available that can indicate that certain modules deserve the most costly QA methods. If so, then some subset of the system may receive more attention by the QA team. We propose defect predictors as a rapid and cost effective lightweight sampling policy for checking if the rest of the system deserves additional attention. As argued above, such a sampling method is essential for generating high-quality systems under the constraint of limited budgets.

2.3 Frequently Asked Questions

2.3.1 *Why Binary Classifications?.* The reader may wonder why we pursue such a simple binary classification scheme ($defective \in \{true, false\}$) and not, say, *number of defects* or *severity of defects*. In reply, we say:

—We do not use *severity of defects* since in large scale data collections, such as those used below, it is hard to distinguish defect "severity" from defect "priority". All too often, we have found that developers will declare a defect "severe" when they are really only stating a preference on what bugs they wish to fix next. Other authors have the same reservations:
  —Nikora cautions that "without a widely agreed upon definition of severity, we can not reason about it" [Nikora 2004].
  —Ostrand et al. make a similar conclusion: "(severity) ratings were highly subjective and also sometimes inaccurate because of political considerations not related to the importance of the change to be made. We also learned that they could be inaccurate in inconsistent ways" [Ostrand et al. 2004].
—We do not use *number of defects* as our target variable since, as shown in Figure 2, only a vanishingly small percent of our modules have more than one issue report. That is, our data has insufficient examples to utilize (say) one method in the kc1 data set with a dozen defects.

2.3.2 *Why Not Use Regression?.* Other researchers (e.g. [Mockus et al. 2005; T. Zimmermann and Murphy 2009]), use a logistic regression model to predict software quality features. Such models have the general form

$$Probability(Y) = \frac{e^{(c+a_1X_1+a_2X_2+...)}}{1+e^{(c+a_1X_1+a_2X_2+...)}}$$

where a_i are the logistic regression predicted constants and the X_i are the independent variables used for building the logistic regression model. For example, in

	Percentage of modules with N defects				
N defects	cm1	kc1	kc3	mw1	pc1
1	10.67	6.50	1.96	5.69	4.15
2	02.17	3.04	1.53	0.74	1.53
3	01.19	2.18	2.83		0.45
4	00.99	0.76		0.25	0.09
5	00.40	0.33			0.09
6	00.20	0.43			0.09
7	00.40	0.28			0.09
8		0.24			
9		0.05			0.09
10		0.05			
11					
12		0.05			
totals	16.01	13.90	6.32	6.68	6.58

Fig. 2. Sample of percentage of defects seen in different modules. Note that only a very small percentage of modules have more than one defect. For more details on these data sets, see Figure 3.

the case of Zimmermann et al,'s work, those variables are measures of code change, complexity, and pre-release bugs. These are used to predict number of defects.

Another regression variant is the negative binomial regression (NBM) model used by Ostrand et al. [Ostrand et al. 2004] to predict defects in AT&T software. Let y_i equal the number of faults observed in file i and x_i be a vector of characteristics for that file. NBM assumes that y_i given x_i has a Poisson distribution with mean λ_i computed from $\lambda_i = \gamma_i e^{\beta x_i}$ where γ_i is the gamma distribution with mean 1 and variance $\sigma^2 \geq 0$ (Ostrand et al. compute σ^2 and β using a maximum likelihood procedure).

Logistic regression and NBM fit one model to the data. When data is multi-modal, it is useful to fit multiple models. A common method for handling arbitrary distributions to approximate complex distributions is via a set of piecewise linear models. *Model tree learners*, such as Quinlan's M5' algorithm [Quinlan 1992a], can learn such piecewise linear models. M5' also generates a decision tree describing when to use which linear model.

We do not use regression for several reasons:

—Regression assumes a continuous target variable and, as discussed above, our target variable is binary and discrete.
—There is no definitive result showing that regression methods are better/worse than the data miners used in this study. In one of the more elaborate recent studies, Lessmann et al. found no statistically significant advantage of logistic regression over a large range of other algorithms [Lessmann et al. 2008] (the Lessmann et al. result is discussed, at length, below).
—In previous work, we have assessed various learning methods (including regression methods and model trees) in terms of their ability be guided by various business considerations. Specifically, we sought learners that could tune their conclusions to user-supplied utility weights about false alarms, probability of detection, etc. Of the fifteen defect prediction methods used in that study, regression and model trees were remarkably *worst* at being able to be guided in this way [Menzies and Stefano 2003]. The last section of this paper discusses a new learner, called

WHICH, that was specially designed to support simple tuning to user-specific criteria.

2.3.3 *Why Static Code Features?.* Another common question is why just use static code features? Fenton [Fenton et al. 1994] divides software metrics into process, product, and personnel and uses these to collect information on how the software was built, what was built, and who built it. Static code measures are just product metrics and, hence, do not reflect process and personnel details. For this reason, other researchers use more that just static code measures. For example:

—Reliability engineers use knowledge of how the frequency of faults seen in a running system changes over time [Musa et al. 1987; Littlewood and Wright 1997].
—Other researchers explore *churn*; i.e. the rate at which the code base changes [Hall and Munson 2000].
—Other researchers reason about the development team. For example, Nagappan et al. comment on how organizational structure effects software quality [Nagappan et al. 2008] while Weyuker et al. document how large team sizes change defect rates [Weyuker et al. 2008].

When replying to this question, we say that static code features are one of the few measures we can collect in a consistent manner across many projects. Ideally, data mining occurs in some CMM level 5 company where processes and data collection is precisely defined. In that ideal case, there exists extensive data sets collected over many projects and many years. These data sets are in a consistent format and there is no ambiguity in the terminology of the data (e.g. no confusion between "severity" and "priority").

We do not work in that ideal situation. Since 1998, two of the authors (Menzies and Cukic) have been research consultants to NASA. Working with NASA's Independent Software Verification and Validation Facility (IV&V), we have tried various methods to add value to the QA methods of that organization. As we have come to learn, NASA is a very dynamic organization. The NASA enterprise has undergone major upheavals following the 2003 loss of the Columbia shuttle, then President Bush's new vision for interplanetary space exploration in 2004, and now (2010) the cancellation of that program. As research consultants, we cannot precisely define data collection in such a dynamic environment. Hence, we do not ask "what are the right features to collect?". Instead, we can only ask "what features can we access, right now?" This question is relevant to NASA as well as any organization where data collection is not controlled by a centralized authority such as:

—agile software projects;
—out-sourced projects;
—open-sourced projects;
—and organizations that make extensive use of sub-contractors and sub-sub contractors.

In our experience, the one artifact that can be accessed in a consistent manner across multiple different projects is the source code (this is particularly true in large projects staffed by contractors, sub-contractors, and sub-sub contractors). Static code features can can be automatically and cheaply extracted from source code,

even for very large systems [Nagappan and Ball 2005a]. By contrast, other methods such as manual code reviews are labor-intensive. Depending on the review methods, 8 to 20 LOC/minute can be inspected and this effort repeats for all members of the review team, which can be as large as four or six [Menzies et al. 2002].

For all the above reasons, many industrial practitioners and researchers (including ourselves) use static attributes to guide software quality predictions (see the list shown in the introduction). Verification and validation (V&V) textbooks [Rakitin 2001] advise using static code complexity attributes to decide which modules are worthy of manual inspections. At the NASA IV&V facility, we know of several large government software contractors that will not review software modules *unless* tools like McCabe predict that some of them might be fault prone.

2.3.4 *What Can be Learned from Static Code Features?.* The previous section argued that, for pragmatic reasons, all we can often collect are static code measures. This is not to say that if we *use* those features, then they yield useful or interesting results. Hence, a very common question we hear about is "what evidence is there that anything useful can be learned from static code measures?".

There is a large body of literature arguing that static code features are an inadequate characterization of the internals of a function:

—Fenton offers an insightful example where *the same* functionality is achieved via *different* language constructs resulting in *different* static measurements for that module [Fenton and Pfleeger 1997]. Using this example, Fenton argues against the use of static code features.
—Shepperd & Ince present empirical evidence that the McCabe static attributes offer nothing more than uninformative attributes like lines of code. They comment "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" [Shepperd and Ince 1994].
—In a similar result, Fenton & Pfleeger note that the main McCabe attributes (cyclomatic complexity, or $v(g)$) are highly correlated with lines of code [Fenton and Pfleeger 1997].

If static code features were truly useless, then the defect predictors learned from them would satisfy two predictions:

Prediction1:. They would perform badly (not predict for defects);

Prediction2:. They would have no generality (predictors learned from one data set would not be insightful on another).

At least in our experiences, these predictions do not hold. This evidence falls into two groups: *field studies* and a *controlled laboratory study.* In the *field studies*:

—Our prediction technology was commercialized in the *Predictive* tool and sold across the United States to customers in the telecom, energy, technology, and government markets, including organizations such as Compagnie Financire Alcatel (Alcatel); Chevron Corporation; LogLogic, Inc.; and Northrop Grumman Corporation. As an example of the use of Predictive, one company (GB Tech, Inc.) used it to manage safety critical software for a United States manned strike fighter. This code had to be tested extensively to ensure safety (the software

controlled a lithium ion battery, which can overcharge and possibly explode). First, a more expensive tool for structural code coverage was applied. Later, the company ran that tool and Predictive on the same code. Predictive produced consistent results with the more expensive tools while being able to faster process a larger code base than the more expensive tool [Turner 2006].

—We took the defect prediction technology of this paper (which was developed at NASA in the USA) and applied it to a software development company from another country (a Turkish software company). The results were very encouraging: when inspection teams focused on the modules that trigger our defect predictors, they found up to 70% of the defects using 40% of the effort (measured in staff hours). Based on those results, we were subsequently invited by two companies to build tools to incorporate our defect prediction methods into their routine daily processes [Tosun et al. 2009].

—A subsequent, more detailed, study on the Turkish software compared how much code needs to be inspected using a random selection process vs selection via our defect predictors. Using the random testing strategy, 87% of the files would have to be inspected in order to detect 87% of the defects. However, if the inspection process was restricted to the 25% of the files that trigger our defect predictors, then 88% of the defects could be found. That is, the same level of defect detection (after inspection) can be achieved using $\frac{87-25}{87} = 71\%$ less effort [**?**].

The results of these field studies run counter to **Prediction1**. However, they are not reproducible results. In order to make a claim that other researchers can verify, we designed a *controlled experiment* to assess **Prediction1** and **Prediction2** in a reproducible manner [Turhan et al. 2009]. That experiment was based on the public domain data sets of Figure 3. These data sets are quite diverse and are written in different languages (C,C++,JAVA); written in different countries (United Stated and Turkey); and written for different purposes (control and monitoring of white goods, NASA flight systems, ground-based software).

Before we can show that experiment, we must first digress to define performance measures for defect prediction. When such a predictor fires then $\{A, B, C, D\}$ denotes the true negatives, false negatives, false positives, and true positives (respectively). From these measures we can compute:

$$\begin{array}{ll} pd = recall = & \frac{D}{B+D} \\ pf = & \frac{C}{A+C} \end{array}$$

In the above, *pd* is the *probability of detecting* a faulty module while *pf* is the *probability of false alarm*. Other performance measures are $accuracy = \frac{A+D}{A+B+C+D}$ and $precision = \frac{D}{C+D}$. Figure 4 shows an example of the calculation of these measures.

Elsewhere[Menzies et al. 2007], we show that accuracy and precision are highly unstable performance indicators for data sets like Figure 3 where the target concept occurs with relative infrequency: in Figure 3, only $\frac{1}{7}th$ (median value) of the modules are marked as defective. Therefore, for the rest of this paper, we will not refer to accuracy or precision.

Having defined performance measures, we can now check **Predictions1&2**; i.e. static defect features lead to poor fault predictors and defect predictors have no

project	source	language	description	# modules	features	%defective
pc1	NASA	C++	flight software for earth orbiting satellites	1,109	21	6.94
kc1	NASA	C++	storage management for ground data	845	21	15.45
kc2	NASA	C++	storage management for ground data	522	21	20.49
cm1	NASA	C++	spacecraft instrument	498	21	9.83
kc3	NASA	JAVA	storage management for ground data	458	39	9.38
mw1	NASA	C++	a zero gravity experiment related to combustion	403	37	7.69
ar4	Turkish white goods manufacturer	C	refrigerator	107	30	18.69
ar3	Turkish white goods manufacturer	C	dishwasher	63	30	12.70
mc2	NASA	C++	video guidance system	61	39	32.29
ar5	Turkish white goods manufacturer	C	washing machine	36	30	22.22
				Total: 4,102		

Fig. 3. Tables of data, sorted in order of number of examples, from `http://promisedata.org/data`. The rows labeled "NASA" come from NASA aerospace projects while the other rows come from a Turkish software company writing applications for domestic appliances. All this data conforms to the format of §2.2.2.

		module found in defect logs?	
		no	yes
signal detected?	no	A = 395	B = 67
	yes	C = 19	D = 39

$$pf = Prob.falseAlarm = 5\%$$
$$pd = Prop.detected = 37\%$$
$$acc = accuracy = 83\%$$
$$prec = precision = 67\%$$

Fig. 4. Performance measures

generality between data sets. If D denotes all the data in Figure 3, and D_i denote one particular data set $D_i \in D$, then we can conduct two kinds of experiments:

SELF:. Self-learning experiments where we *train* on 90% of D_i then test on the remaining 10% . Note that such self-learning experiments will let us comment on **Prediction1**.

RR:. Round-robin experiments where we *test* on 10% (randomly selected) of data set D_i after *training* on the remaining nine data sets $D - D_i$. Note that such round-robin experiments will let us comment on **Prediction2**.

experiment	notes	median pd%	median pf%
RR	round-robin	94	68
RR2	round-robin + relevancy filtering	69	27
SELF	self test	75	29

Fig. 5. Results of round-robin and self experiments. From [Turhan et al. 2009]. All the *pd* and *pf* results are statistically different at the 95% level (according to a Mann-Whitney test).

It turns out that the round-robin results are unimpressive due to an *irrelevancy effect*, discussed below. Hence, it is also useful to conduct:

RR2:. Round-robin experiments where a *relevancy filter* is used to filter away irrelevant parts of the training data.

After repeating experiments RR, SELF, RR2 twenty times for each data set $D_i \in D$, the median results are shown in Figure 5. At first glance, the round-robin results of RR seem quite impressive: a 98% probability of detection. Sadly, these high detection probabilities are associated with an unacceptably high false alarm rate of 68%.

In retrospect, this high false alarm rate might have been anticipated. A median sized data set from Figure 3 (e.g. $mw1$) has around 450 modules. In a round-robin experiment, the median size of the training set is over 3600 modules taken from nine other projects. In such an experiment, it is highly likely that the defect predictor will be learned from numerous irrelevant details from other projects.

To counter the problem of irrelevant training data, the second set of round-robin experiments constructed training sets for D_i from the union of the 10 nearest neighbors within $D - D_i$. The RR2 results of Figure 5 show the beneficial effects of relevancy filtering: false alarm rates reduced by $\frac{68}{27} = 252\%$ with only a much smaller reduction in *pd* of $\frac{94}{69} = 136\%$.

Returning now to **Prediction1**, the SELF and RR2 $pd \geq 69\%$ results are much larger than those seen in industrial practice:

—A panel at *IEEE Metrics 2002* [Shull et al. 2002] concluded that manual software reviews can find ≈60% of defects[4].
—Raffo found that the defect detection capability of industrial review methods can vary from $pd = TR(35, 50, 65)\%$[5]. for full Fagan inspections [Fagan 1976] to $pd = TR(13, 21, 30)\%$ for less-structured inspections [Raffo 2005].

That is, contrary to **Prediction1**, defect predictors learned from static code features perform well, relative to standard industrial methods.

Turning now to **Prediction2**, note that the RR2 round-robin results (with relevancy filtering) are close to the SELF:

—The *pd* results are only $1 - \frac{75}{69} = 8\%$ different;
—The *pf* results are only $\frac{29}{27} - 1 = 7\%$ different.

[4]That panel supported neither Fagan claim [Fagan 1986] that inspections can find 95% of defects before testing or Shull's claim that specialized directed inspection methods can catch 35% more defects that other methods [Shull et al. 2000].
[5]$TR(a, b, c)$ is a triangular distribution with min/mode/max of a, b, c.

That is, contrary to **Prediction2**, there is generality in the defect predictions learned from static code features. Learning from local data is clearly best (SELF's *pd* results are better than RR2), however, nearly the same performance results as seen in SELF can be achieved by applying defect data from one site (e.g. NASA fight systems) to another (e.g. Turkish white good software).

2.4 Summary

For all the above reasons, we research defect predictors based on static code features. Such predictors are:

—*Useful:* they out-perform standard industrial methods. Also, just from our own experience, we can report that they have been successfully applied in software companies in the United States and Turkey.
—*Generalizable:* as the RR2 results show, the predictions of these models generalize across data sets taken from different organizations working in different countries.
—*Easy to use:* they can automatically process thousands of modules in a matter of seconds. Alternative methods such as manual inspections are much slower (8 to 20 LOC per minute).
—*Widely-used:* We can trace their use as far back as 1990 [Porter and Selby 1990]. We are also aware of hundreds of publications that explore this method (for a partial sample, see the list shown in the introduction).

3. CEILING EFFECTS IN DEFECT PREDICTORS

Despite several years of exploring different learners and data pre-processing methods, the performance of our learners has not improved. This section documents that ceiling effect and the rest of this paper explores methods to break through the ceiling effect.

In 2006 [Menzies et al. 2007], we defined a repeatable defect prediction experiment which, we hoped, others could improve upon. That experiment used public domain data sets and open source data miners. Surprisingly, a simple näive Bayes classifier (with some basic pre-processor for the numerics) out-performed the other studied methods. For details on näive Bayes classifiers, see the appendix.

We made the experiment repeatable in the hope that other researchers could improve or refute our results. So far, to the best of our knowledge, no study using just static code features has out-performed our 2006 result. Our own experiments [Jiang et al. 2008] found little or no improvement from the application of numerous data mining methods. Figure 6 shows some of those results using (in order, left to right) *aode* average one-dependence estimators [Yang et al. 2006]; *bag* bagging [Brieman 1996]; *bst* boosting [Freund and Schapire 1997]; IBk instance-based learning [Cover and Hart 1967]; $C4.5$ C4.5 [Quinlan 1992b]; $jrip$ RIPPER [Cohen 1995b]; lgi logistic regression [Breiman et al. 1984]; *nb* näive Bayes (second from the right); and rf random forests [Breimann 2001]. These histograms show area under the curve (AUC) of a pf-vs-pd curve. To generate such a "AUC(pf,pd)" curve:

—A learner is executed multiple times on different subsets of data;
—The pd, pf results are collected from each execution;
—The results are sorted on increasing order of pf;

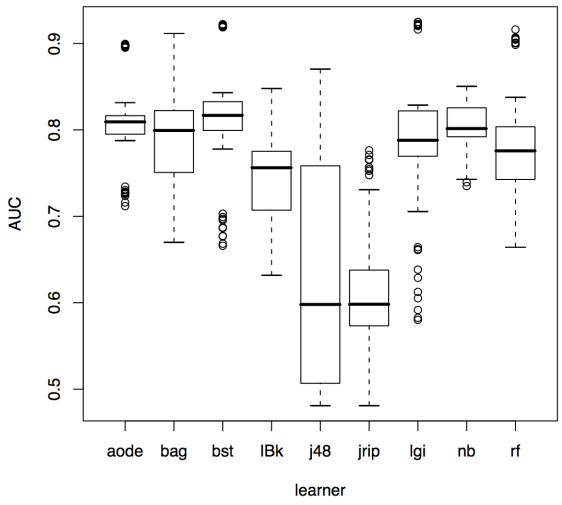

Fig. 6. Box plot for AUC(pf,pd) seen with 9 learners when, 100 times, a random 90% selection of the data is used for training and the remaining data is used for testing. The rectangles show the inter-quartile range (the 25% to 75% quartile range). The line shows the minimum to maximum range, unless that range extends beyond 1.5 times the inter-quartile range (in which case dots are used to mark these extreme outliers). From [Jiang et al. 2008].

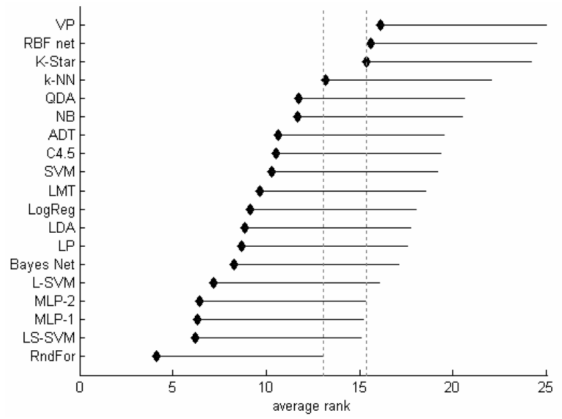

Fig. 7. Range of AUC(pf,pd) ranks seen in 19 learners building defect predictors when, 10 times, a random 66% selection of the data is used for training and the remaining data is used for testing. In ranked data, values from one method are replaced by their rank in space of all sorted values (so *smaller* 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 [Lessmann et al. 2008].

—The results are plotted on a 2-D graph using pf for the x-axis and pd for the y-axis.

A statistical analysis of the Figure 6 results showed that only boosting on discretized data offers a statistically better result than näive Bayes. However, we cannot recommend boosting: boosting is orders of magnitudes slower than näive Bayes; and the median improvement over näive Bayes is negligible.

Other researchers have also failed to improve our results. For example, Figure 7 shows results from a study by Lessmann et al. on statistical differences between 19 learners used for defect prediction [Lessmann et al. 2008]. At first glance, our preferred näive Bayes method (shown as "NB" on the sixth line of Figure 7) 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 7 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.

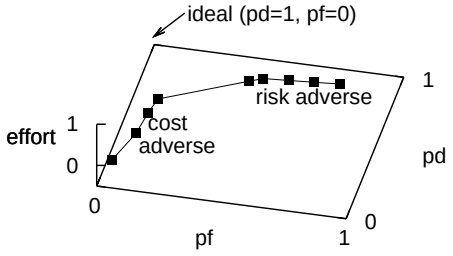

Fig. 8. Pf-vs-pd-vs-$effort$.

4. BREAKING THROUGH THE CEILING

This section discusses methods for breaking through the ceiling effects documented above.

One constant in the results of Figure 6 and Figure 7 is the performance *goal* used in those studies: both those results assumed the goal of the learning was to maximize AUC(pf,pd), i.e. the area under a pf-vs-pd curve. As shown below, if we change the goal of the learning, then we can break the ceiling effect and find better (and worse) methods for learning defect predictors from static code measures.

Depending on the business case that funded the data mining study, different goals may be most appropriate. To see this, consider the typical pf-vs-pd-vs-$effort$ curve of Figure 8:

—The pf,pd performance measures were defined above.
—$Effort$ is the percentage of the code base found in the modules predicted to be faulty (so if all modules are predicted to be faulty, the 100% of the code base must be processed by some other, slower, more expensive QA method).

For the moment, we will just focus on the pf, pd plane of Figure 8. A perfect detector has no false alarm rates and finds all fault modules; i.e. pf, pd=0, 1. As shown in Figure 8, the AUC(pf,pd) can bend towards this ideal point but may never reach there:

—Detectors learned from past experience have to make some inductive leaps and, in doing so, make some mistakes. That is, the only way to achieve high pds is to accept some level of pfs.
—The only way to avoid false alarms is to decrease the probability that the detector will trigger. That is, the only way to achieve low pfs is to decrease pd.

Different businesses prefer different regions of Figure 8 curve:

—Mission-critical systems are *risk averse* and may accept very high false alarm rates, just as long as they catch any life-threatening possibility.
—For less critical software, *cost averse* managers may accept lower probabilities of detection, just as long as they do not waste budgets on false alarms.

That is, different businesses have different $goals$:

Goal1:. Risk averse developments prefer high pd;

Goal2:. Cost averse developments accept mid-range pd, provided they get low pf.

Arisholm & Briand [Arisholm and Briand 2006] propose yet another another goal:

Goal3:. A budget-conscious team wants to know that if X% of the modules are predicted to be defective, then modules contain more than X% of the defects. Otherwise, they argue, the cost of generating the defect predictor is not worth the effort.

The $effort$-based evaluation of **Goal3** uses a dimension not explored by the prior work that reported ceiling effects (Lessmann et al. or our work [Jiang et al. 2008; Menzies et al. 2007]). Hence, for the rest of this paper, we will assess the impacts of the Arisholm & Briand goal of maximizing the "AUC(effort,pd)".

4.1 Experimental Set Up

4.1.1 *Operationalizing AUC(effort,pd).* To operationalize **goal3** from Arisholm & Briand evaluation, we assume that:

—After a data miner predicts a module is defective, it is inspected by a team of human experts.
—This team correctly recognize some subset Δ of the truly defective modules, (and $\Delta = 1$ means that the inspection teams are perfect at their task).
—Our goal is to find learners that find the *most* number of defective modules in the *smallest* number of modules (measured in terms of LOC).

For Arisholm & Briand to approve of a data miner, it must fall in the region $pd > effort$. The *minimum* curve in Figure 9 shows the lower boundary of this region and a "good" detector (according to AUC(effort,pd)) must fall above this line. Regarding the x-axis and y-axis of this figure:

—The x-axis shows all the modules, sorted on size. For example, if we had 100 modules of 10 LOC, 10 modules of 15 LOC, and 1 module of 20 LOC then the x-axis would be 111 items long with the 10 LOC modules on the left-hand side and the 20LOC module on the right-hand side.
—Note that the y-axis of this figure assumes $\Delta = 1$; i.e. inspection teams correctly recognizes all defective modules. Other values of Δ are discussed below.

4.1.2 *Upper and Lower Bounds on Performance.* It is good practice to compare the performance of some technique against theoretical *upper* and *lower* bounds [Cohen 1995a]. Automatic data mining methods are interesting if they out-perform manual methods. Therefore, for a *lower-bound* on expected performance, we compare them against some manual methods proposed by Koru et.al. [Koru et al. 2007; Koru et al. 2008; Koru et al. 2009]:

—They argue that the relationship between module *size* and *number of defects* is not linear, but *logarithmic*; i.e. smaller modules are proportionally more troublesome.
—The *manualUp* and *manualDown* curves of Figure 9 show the results expected by Koru et al. from inspecting modules in increasing/decreasing order of size (respectively).
—With *manualUp*, all modules are selected and sorted in increasing order of size, so that curve runs from 0 to 100% of the LOC.

In a result consistent with Koru et.al., our experiments show $manualUp$ usually defeating $manualDown$. As shown in Figure 9, $manualUp$ scores higher on $effort$-vs-PD than $manualDown$. Hence, we define an *upper bound* on our performance as follows. Consider an optimal oracle that restricts module inspections to just the modules that are truly defective. If *manualUp* is applied to just these modules, then this would show the upper-bound on detector performance. For example, Figure 9 shows this *best* curve where 30% of the LOC are in defective modules.

In our experiments, we ask our learners to make a binary decision ($defective, nonDefective$). All the modules identified as $defective$ are then sorted in order of increasing size (LOC). We then assess their performance by AUC(effort,pd). For example, the *bad* learner in Figure 9 performs worse than the *good* learner since the latter has a larger area under its curve.

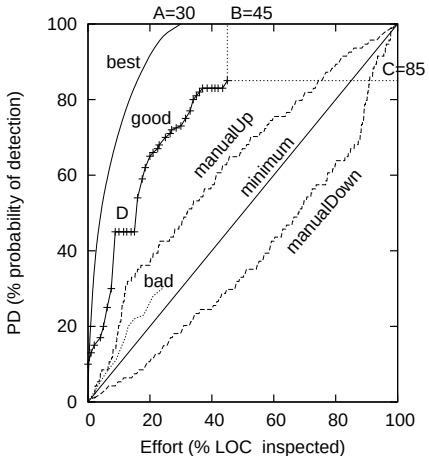

Fig. 9. $Effort$-vs-PD.

In order to provide an upper-bound on our AUC, we report them as a ratio of the area under the *best* curve. All the performance scores mentioned in the rest of this paper are hence *normalized AUC(effort,pd)* values ranging from 0% to 100% of the *best* curve.

Note that normalization simplifies our assessment criteria. If the effectiveness of the inspection team is independent of the method used to select the modules that they inspect, then Δ 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, this Δ cancels out so we can assess the relative merits of different defect predictors *independently* of Δ.

4.1.3 *Details.* Three more details will complete our discussion of Figure 9. Defect detectors usually do not trigger on all modules. For example, the *good* curve of Figure 9 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 complete the $effort$-vs-PD curve, we must fill in the gap between the termination point and $X = 100$. Later in this article, we will assume that test engineers inspect 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 $C = 85$ (i.e. the 85% of the code triggered by the learner that generated the *good* curve).

Secondly, the following observation will become significant when we tune a learner to AUC(effort,pd). Even though Figure 9 shows $effort$-vs-PD, it can also indirectly show false alarms. Consider the plateau in the *good* curve of Figure 9, 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, to maximize the area under an $effort$-vs-PD, we could assign a heavy penalty against false alarms that lead to plateaus.

Thirdly, Figure 9 assumes that inspection effort is linear on size of module. We make this assumption since a previous literature review reported that current in-

spection models all report linear effort models [Menzies et al. 2002]. Nevertheless, Figure 9 could be extended to other effort models as follows: stretch the x-axis to handle, say, non-linear effects such as longer modules that take exponentially more time to read and understand.

4.2 Initial results

Figure 9's *bad* and *manualUp* curves show our first attempt at applying this new evaluation bias. These curves were generated by applying *manualUp* and the C4.5 tree learner [Quinlan 1992b] to one of the data sets studied by Lessmann et al. Observe how the the automatic method performed far worse than a manual one. To explain this poor performance, we comment that data miners grow their models using a search bias B_1, then we assess them using a different evaluation bias B_2. For example:

—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.
—During *testing*, the learned decision-tree might be tested on a variety of criteria such as Lessmann et al.'s AUC measure or our operationalization of AUC(effort,pd).

It is hardly surprising that C4.5 performed so poorly in Figure 9. C4.5 was not designed to optimize AUC(effort,pdf) (since B_1 was so different to B_2). Some learning schemes support biasing the learning according to the overall goal of the system; for example:

—The *cost-sensitive learners* discussed by Elkan [Elkan 2001];
—The *ROC ensembles* discussed by Fawcett [Fawcett 2001] where the conclusion is a summation of the conclusions of the ensemble of ROC curves[7], proportionally weighted, to yield a new learner.
—Our cost curve meta-learning scheme permits an understanding of the performance of a learner across the entire space of pd-vs-pf trade-offs [Jiang et al. 2008].

At best, such biasing only indirectly controls the search criteria. If the search criteria is orthogonal to the success criteria of, say, maximizing $effort$-vs-pd, then cost-sensitive learning or ensemble combinations or cost curve meta-learning will not be able to generate a learner that supports that business application. Accordingly, we decided to experiment with a new learner, called WHICH, whose internal search criteria can be tuned to a range of goals such as AUC(effort,pd).

5. WHICH

The previous section argued for a change in the goals of of data miners. WHICH [Milton 2008] is a meta-learning scheme that uses a configurable search bias to grow its models. This section describes WHICH, how to customize it, and what happened when we applied those customizations to the data of Figure 3.

[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 C4.5.
[7]ROC= receiver-operator characteristic curves such as Lessmann et al.'s plots of PD-vs-PF or PD-vs-precision

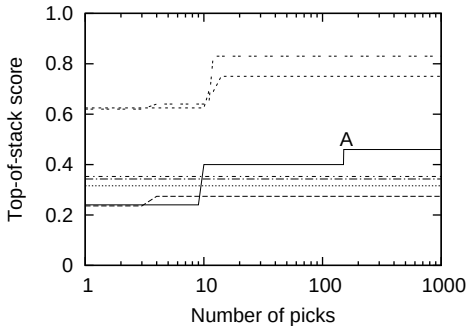

Fig. 10. Top-of-stack scores of the WHICH stack seen after multiple "picks" (selection and scoring of two conditions picked at random, then combined) for seven data sets from the UCI data mining repository [Blake and Merz 1998]. Usually, top-of-stack stabilizes after just a dozen pick. However, occasionally, modest improvements are seen after a few hundred "picks" (see the plot marked with an "A").

5.1 Details

WHICH loops over the space of possible feature ranges, evaluating various combinations of features:

(1) Data from continuous features is discretized into "N" equal width bins. We tried various bin sizes and, for this study, best results were seen using $N \in \{2, 4, 8\}$ bins of width $(max - min)/N$.
(2) WHICH maintains a stack of feature combinations, sorted by a customizable search bias B_1 . For this study, WHICH used the AUC(effort,pd) criteria, discussed below.
(3) Initially, WHICH's "combinations" are just each range of each feature. Subsequently, they can grow to two or more features.
(4) Two combinations are picked at random, favoring those combinations that are ranked highly by B_1.
(5) The two combinations are themselves combined, scored, then sorted into the stacked population of prior combinations.
(6) Go to step 4.

For the reader aware of the artificial intelligence (AI) literature, we remark that WHICH is a variant of beam search. Rather than use a fixed beam size, WHICH uses a fuzzy beam where combinations deeper in the stack are exponentially less likely to be selected. Also, while a standard beam search just adds child states to the current frontier, WHICH can add entire sibling branches in the search tree (these sibling branches are represented as other combinations on the stack).

After numerous loops, WHICH returns the highest ranked combination of features. During testing, modules that satisfy this combination are predicted as being "defective". These modules are sorted on increasing order of size and the statistics of Figure 9 are collected.

The looping termination criteria was set using our engineering judgment. In studies with UCI data sets [Blake and Merz 1998], Milton showed that the score of

top-of-stack condition usually stabilizes in less than 100 picks [Milton 2008] (those results are shown in Figure 10). Hence, to be cautious, we looped 200 times.

The following expression guides WHICH's search:

$$B_1 = 1 - \frac{\sqrt{PD^2 * \alpha + (1-PF)^2 * \beta + (1-effort)^2 * \gamma}}{\sqrt{\alpha + \beta + \gamma}} \quad (1)$$

The $(PD, PF, effort)$ values are normalized to fall between zero and one. The (α, β, γ) terms in Equation 1 model the relative utility of $PD, PF, effort$ respectively. These values range $0 \leq (\alpha, \beta, \gamma) \leq 1$. Hence:

—$0 \leq B_1 \leq 1$;
—larger values of B_1 are better;
—increasing $(effort, PF, PD)$ leads to $(decreases, decreases, increases)$ in B_1 (respectively).

Initially, we gave PD and $effort$ equal weights and ignored PF; i.e. $\alpha = 1, \beta = 0, \gamma = 1$. This yielded disappointing results: the performance of the learned detectors varied wildly across our cross-validation experiments. An examination of our data revealed why: there exists a small number of modules with very large LOCs. For example, in one data set with 126 modules, most have 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-validation, then the performance associated with WHICH's rule can vary unpredictably from one run to another.

Accordingly, we had to use PF as a surrogate measure for $effort$. Recall from the above discussion that we can restrain decreases in PD by assigning a heavy penalty to the false alarms that lead to plateaus in an $effort$-vs-PD curve. In the following experiments, we used a B_1 equation that disables $effort$ but places a very large penalty on PF; i.e.

$$\alpha = 1,\ \beta = 1000,\ \gamma = 0 \quad (2)$$

We acknowledge that the choice Equation 1 and Equation 2 is somewhat arbitrary. In defense of these decisions, we note that in the following results, these decisions lead to a learner that significantly out-performed standard learning methods.

5.2 Results

Figure 11 shows results from experimental runs with different learners on the data sets of Figure 3. 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 C4.5 decision tree learner, the RIPPER rule learner, and our previously recommended näive 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.

—*Three versions of WHICH:* This study applied several variants of WHICH. WHICH-2, WHICH-4, and WHICH-8 discretize numeric ranges into 2,4, and 8 bins (respectively).
—*MICRO-20:* MICRO-20 was another variant motivated by the central limit theorem. According to the central limit theorem, the sum of a large enough sample will be approximately normally distributed (the theorem explains the prevalence of the normal probability distribution). The sample can be quite small, sometimes even as low as 20. Accordingly, MICRO-20 was a variant of WHICH-2 that learns from just 20+20 examples of defective and non-defective modules (selected at random).

5.2.1 *Overall Results.* Figure 11 shows the results for *all* the data sets of Figure 3, combined:

—Each row shows the normalized AUC(effort,pdf) results for a particular learner over 30 experiments (10 repeats of a three-way). These results are shown as a 25% to 75% quartile range (and the large black dot indicates the median score).
—The left-hand-side column of each row shows the results of a Mann-Whitney (95% confidence test) of each row. Row i has a different rank to row $i+1$ if their median scores are different and the Mann-Whitney test indicates that the two rows have a different wins+ties results. See the appendix for a discussion on why the Mann-Whitney test was used on these results.

In Figure 11, WHICH performs *relatively* and *absolutely* better than all of the other methods studied in this paper:

—*Relative performance:* WHICH-2 and the MICRO-20 learner have the highest ranks;
—*Absolute performance:* In our discussion of Figure 9, the *best* curve was presented as the upper bound in performance for any learner tackling AUC(effort,pd). WHICH's performance rises close to this upper bound, rising to 70.9 and 80% (median and 75% percentile range) of the *best* possible performance.

Several other results from Figure 11 are noteworthy.

—There is no apparent benefit in detailed discretization: WHICH-2 outperforms WHICH-4 and WHICH-8.
—In a result consistent with our prior publications [Menzies et al. 2007], our näive Bayes classifier out-performs other standard data miners (C4.5 and RIPPER).
—In a result consistent with Koru et.al.'s logarithmic defect hypothesis, manualUp defeats manualDown.
—In Figure 11, standard data miners are defeated by manual method (manualUp). The size of the defeat is very large: median values of 61.1% to 27.6% from manualUp to C4.5.

This last result is very sobering. In Figure 11, two widely used methods (C4.5 and RIPPER) 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.

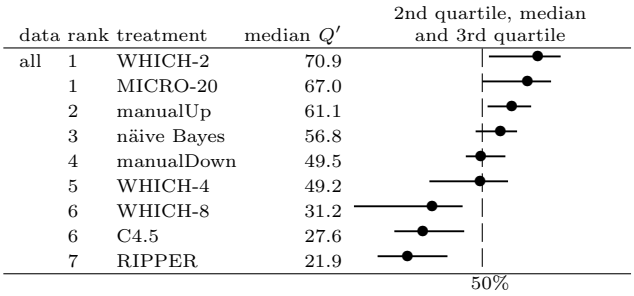

Fig. 11. Results from all data sets of Figure 3, combined from 10 repeats of a 3-way cross-val, sorted by *median* Q'. Each row shows 25 to 75% percentile range of the normalized AUC(effort,pdf) results (and the large black dot indicates the median score). Two rows have different *ranks* (in the left-hand-side column) if their median AUC scores are different and a Mann-Whitney test (95% confidence) indicates that the two rows have a different wins+ties results. Note that we do not recommend WHICH-4 and WHICH-8 since these discretization policies performed much worse than WHICH-2.

5.2.2 *Individual Results.* Figure 11 combines results from all data sets. Figures 12, 13, 14, and 15 look at each data set in isolation. The results divide into three patterns:

—In the eight data sets of pattern #1 (shown in Figure 12 and Figure 13), WHICH-2 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 14), WHICH-2 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 15), WHICH-2 is soundly defeated by manual methods (manualUp). However, in this case, the WHICH-2 variant MICRO-20 falls into the second rank.

In summary, when looking at each data set in isolation, WHICH performs very well in $\frac{9}{10}$ of the data sets.

5.3 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. [Lessmann et al. 2008] and our prior work [Menzies et al. 2007] come from NASA aerospace applications. Here, we use that data, plus three extra data sets from 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 Turkish software was built in a profit- and revenue-driven commercial organization, whereas NASA is a cost-driven government entity
—The Turkish 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 team spread around the United States.
—The Turkish development was carried out in an ad-hoc, informal way rather than

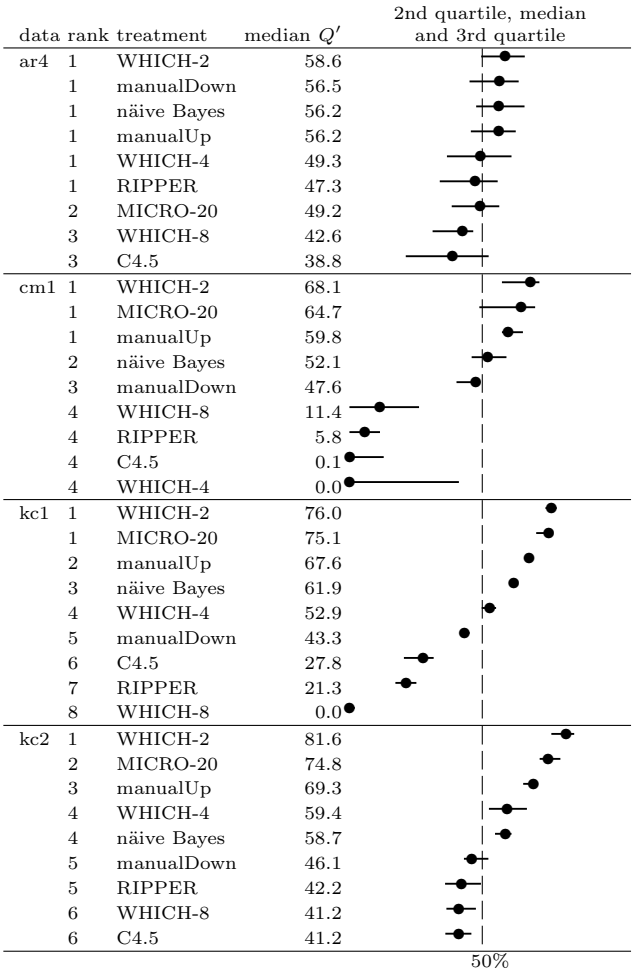

Fig. 12. Four examples of pattern #1: WHICH-2 ranked #1 *and* has highest median. This figure is reported in the same format as Figure 11.

the formal, process oriented approach used at NASA.

Our general conclusion, that WHICH is preferred to other methods when optimzing for AUC(effort,pd), holds for $\frac{6}{7}$ of the NASA data sets and $\frac{3}{3}$ of the Turkish sets. The fact that the same result holds for such radically different organizations is a strong argument for the external validity of our results.

While the above results, based on ten data sets, are no promise of the efficacy of WHICH on future data sets, these results are strong evidence that, when a learner is assessed using AUC(effort, pd), then:

—Of all the learners studied here, WHICH or MICRO-20 is preferred over other learners;
—Standard learners such as näive Bayes, the RIPPER rule learner, and the C4.5

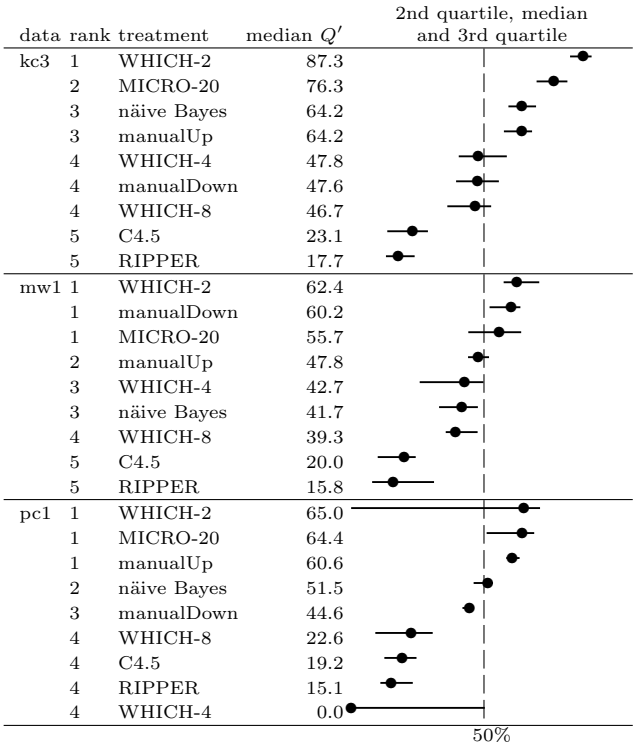

Fig. 13. Three examples of pattern #1: WHICH-2 ranked #1 *and* has the highest median. This figure is reported in the same format as Figure 11.

decision tree learner perform much worse than simple manual methods. Hence, we must strongly depreciate their use when optimizing for AUC(effort,pd).

6. DISCUSSION

This goal of this paper was to comment on Lessmann et al.'s results by offering one example where knowledge of the evaluation biases alters which learner "wins" a comparative evaluation study. The current version of WHICH offers that example.

While that goal was reached, there are many open issues that could be fruitfully explored, in future work. Those issues divide into *methodological issues* and *algorithmic issues*.

6.1 Methodological Issues

This paper has commented that the use of a new goal (AUC(effort,pd)) resulted in improved performance for certain learners tuned to that new goal. It should be noted that trying different goals for learners randomly is perhaps too expensive. Such an analysis may never terminate since the space of possible goals is very large.

We do not recommend random goal selection. Quite the reverse, in fact. We would propose that:

—*Before* commencing data mining, there must be some domain analysis with the

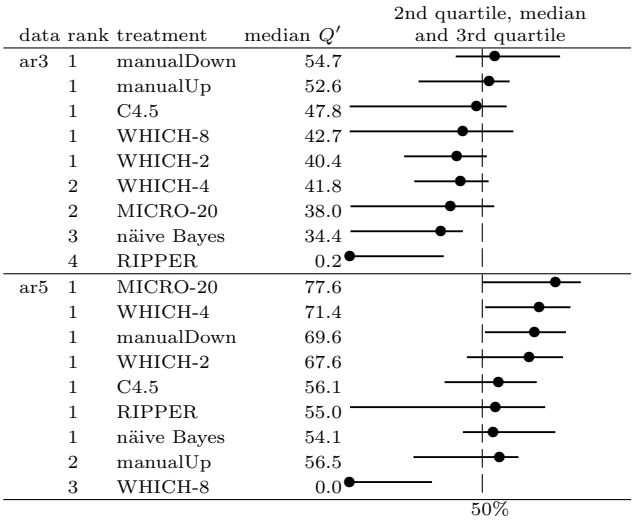

Fig. 14. Two examples of pattern #2: While WHICH-2 did not achieve the highest medians, it was still ranked #1 compared to eight other methods. This figure is reported in the same format as Figure 11.

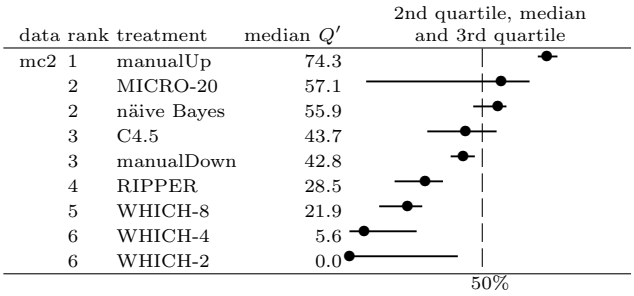

Fig. 15. The only example of pattern #3: WHICH-2 loses (badly) but MICRO-20 still ranks high. This figure is reported in the same format as Figure 11.

goal of determining the success criteria that most interests the user population. (for a sample of such goals, recall the discussion at the start of §4 regarding *mission-critical* and *other* systems).

—Once the business goals have been modeled, then the data miners should be customized to those goals.

That is, rather than conduct studies with randomized business goals, we argue that it is better to *let business considerations guide the goal exploration.* Of course, such an analysis would be pointless unless the learning tool can be adapted to the business goals. WHICH was specially designed to enable the rapid customization of the learner to different goals. For example, while the current version supports AUC(effort,pd), that can be easily changed to other goals.

6.2 Algorithmic Issues

The algorithmic issues concerning the inner details of WHICH:

—Are there better values for (α, β, γ) than Equation 2?
—The above study only explored AUC(effort,pd) and this is only one possible goal of a defect predictor. It could be insightful to explore other goals.
—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 unused an unrestricted stack size.
—Currently, WHICH sorts new items into the stack using a linear time search from 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 [Pugh 1990].
—Other rule learners employ a greedy back-select to prune conditions. To implement such a search, check if removing any part of the combined condition improves the score. If not, terminate the back select. Otherwise, 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 $MaxLoops$ is 200. This may be an overly cautious setting. Given the results of Figure 10, $MaxLoops$ might be safely initialized to 20 and only increased if no dramatic improvement seen in the first loop. For most domains, this would yield a ten-fold speed up of our current implementation.

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

7. CONCLUSION

Given limited QA budgets, it is not possible to apply the most effective QA method to all parts of a system. The manager's job is to decide what needs to be tested most, or tested least. Static code defect predictors are one method for auditing those decisions. Learned from historical data, these detectors can check which parts of the system deserve more QA effort. As discussed in §2.4, defect predictors learned from static code measures are *useful* and *easy to use*. Hence, as shown by a list offered in the introduction, they are very *widely-used*.

Based on our own results, and those of Lessmann et al., it seems natural to conclude that many learning methods have equal effectiveness at learning defect predictors from static code features. In this paper, we have shown that this *ceiling effect* does not necessarily hold when studying performance criteria other than AUC(pf,pd). When defect predictors are assessed by other criteria such as "read less, see more defects" (i.e. AUC(effort,pd)), then the selection of the appropriate learner becomes critical:

—A learner tuned to "read less, see more defects" performs best;
—A simple manual analysis out-performs certain standard learners such as NB, C4.5, RIPPER. The use of these learners is therefore depreciated for "read less, see more defects".

Our conclusion is that knowledge of the *goal* of the learning can and should be used to select a preferred learner for a particular domain. The WHICH meta-learning framework is one method for quickly customizing a learner to different goals.

We hope that this paper prompts a new cycle of defect prediction research focused on selecting the *best* learner(s) for *particular* business goals. In particular, based on this paper, we now caution that it is an open and urgent question whether or not many of our learners perform any better than simple manual methods.

REFERENCES

Arisholm, E. and Briand, L. 2006. Predicting fault-prone components in a java legacy system. In *5th ACM-IEEE International Symposium on Empirical Software Engineering (ISESE), Rio de Janeiro, Brazil, September 21-22*. Available from `http://simula.no/research/engineering/publications/Arisholm.2006.4`.

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

Bradley, P. S., Fayyad, U. M., and Reina, C. 1998. Scaling clustering algorithms to large databases. In *Knowledge Discovery and Data Mining*. 9–15. Available from `http://citeseer.ist.psu.edu/bradley98scaling.html`.

Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. 1984. Classification and regression trees. Tech. rep., Wadsworth International, Monterey, CA.

Breimann, L. 2001. Random forests. *Machine Learning*, 5–32.

Brieman, L. 1996. Bagging predictors. *Machine Learning 24,* 2, 123–140.

Chapman, M. and Solomon, D. 2002. The relationship of cyclomatic complexity, essential complexity and error rates. Proceedings of the NASA Software Assurance Symposium, Coolfont Resort and Conference Center in Berkley Springs, West Virginia. Available from `http://www.ivv.nasa.gov/business/research/osmasas/conclusion2002/Mike_C%hapman_The_Relationship_of_Cyclomatic_Complexity_Essential_Complexity_and_Erro%r_Rates.ppt`.

Cohen, P. 1995a. *Empirical Methods for Artificial Intelligence*. MIT Press.

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

Cover, T. M. and Hart, P. E. 1967. Nearest neighbour pattern classification. *IEEE Transactions on Information Theory*, 21–27.

Demsar, J. 2006. Statistical comparisons of clasifiers over multiple data sets. *Journal of Machine Learning Research 7*, 1–30. Avaliable from `http://jmlr.csail.mit.edu/papers/v7/demsar06a.html`.

Dietterich, T. 1997. Machine learning research: Four current directions. *AI Magazine 18,* 4, 97–136.

Domingos, P. and Pazzani, M. J. 1997. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning 29,* 2-3, 103–130.

Elkan, C. 2001. The foundations of cost-sensitive learning. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI01)*. Available from `http://www-cse.ucsd.edu/users/elkan/rescale.pdf`.

Fagan, M. 1976. Design and code inspections to reduce errors in program development. *IBM Systems Journal 15,* 3.

Fagan, M. 1986. Advances in software inspections. *IEEE Trans. on Software Engineering*, 744–751.

Fawcett, T. 2001. Using rule sets to maximize roc performance. In *2001 IEEE International Conference on Data Mining (ICDM-01)*. Available from `http://home.comcast.net/~tom.fawcett/public_html/papers/ICDM-final.pdf`.

Fenton, N., Pfleeger, S., and Glass, R. 1994. Science and Substance: A Challenge to Software Engineers. *IEEE Software*, 86–95.

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

Fenton, N. E. and Pfleeger, S. 1995. *Software Metrics: A Rigorous & Practical Approach (second edition)*. International Thompson Press.

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

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

Hall, G. and Munson, J. 2000. Software evolution: code delta and code churn. *Journal of Systems and Software*, 111 – 118.

Halstead, M. 1977. *Elements of Software Science*. Elsevier.

Huang, J. and Ling, C. 2005. Using auc and accuracy in evaluating learning algorithms. *IEEE Transactions on Knowledge and Data Engineering 17,* 3 (March), 299–310.

Jiang, Y., Cukic, B., and Ma, Y. 2008. Techniques for evaluating fault prediction models. *Empirical Software Engineering*, 561–595.

Jiang, Y., Cukic, B., and Menzies, T. 2008. Does transformation help? In *Defects 2008*. Available from `http://menzies.us/pdf/08transform.pdf`.

Khoshgoftaar, T. 2001. An application of zero-inflated poisson regression for software fault prediction. In *Proceedings of the 12th International Symposium on Software Reliability Engineering, Hong Kong*. 66–73.

Khoshgoftaar, T. and Allen, E. 2001. Model software quality with classification trees. In *Recent Advances in Reliability and Quality Engineering*, H. Pham, Ed. World Scientific, 247–270.

Khoshgoftaar, T. M. and Seliya, N. 2003. Fault prediction modeling for software quality estimation: Comparing commonly used techniques. *Empirical Software Engineering 8,* 3, 255–283.

Koru, A., Emam, K. E., Zhang, D., Liu, H., and Mathew, D. 2008. Theory of relative defect proneness: Replicated studies on the functional form of the size-defect relationship. *Empirical Software Engineering*, 473–498.

Koru, A., Zhang, D., El Emam, K., and Liu, H. 2009. An investigation into the functional form of the size-defect relationship for software modules. *Software Engineering, IEEE Transactions on 35,* 2 (March-April), 293 –304.

Koru, A., Zhang, D., and Liu, H. 2007. Modeling the effect of size on defect proneness for open-source software. In *Proceceedings PROMISE'07 (ICSE)*. Available from `http://promisedata.org/pdf/mpls2007KoruZhangLiu.pdf`.

Lessmann, S., Baesens, B., Mues, C., and Pietsch, S. 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*.

Leveson, N. 1995. *Safeware System Safety And Computers*. Addison-Wesley.

Littlewood, B. and Wright, D. 1997. Some conservative stopping rules for the operational testing of safety-critical software. *IEEE Transactions on Software Engineering 23,* 11 (November), 673–683.

Lowry, M., Boyd, M., and Kulkarni, D. 1998. Towards a theory for integration of mathematical verification and empirical testing. In *Proceedings, ASE'98: Automated Software Engineering*. 322–331.

Lutz, R. and Mikulski, C. 2003. Operational anomalies as a cause of safety-critical requirements evolution. *Journal of Systems and Software*. Available from `http://www.cs.iastate.edu/~rlutz/publications/JSS02.ps`.

McCabe, T. 1976. A complexity measure. *IEEE Transactions on Software Engineering 2,* 4 (Dec.), 308–320.

Menzies, T. and Cukic, B. 2000. When to test less. *IEEE Software 17,* 5, 107–112. Available from `http://menzies.us/pdf/00iesoft.pdf`.

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

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

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

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

Milton, Z. 2008. Which rules. M.S. thesis.

Mockus, A., Zhang, P., and Li, P. L. 2005. Predictors of customer perceived software quality. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*. ACM, New York, NY, USA, 225–233.

Musa, J., Iannino, A., and Okumoto, K. 1987. *Software Reliability: Measurement, Prediction, Application*. McGraw Hill.

Nagappan, N. and Ball, T. 2005a. Static analysis tools as early indicators of pre-release defect density. In *ICSE 2005, St. Louis*.

Nagappan, N. and Ball, T. 2005b. Static analysis tools as early indicators of pre-release defect density. In *ICSE*. 580–586.

Nagappan, N., Murphy, B., and V, B. 2008. The influence of organizational structure on software quality: An empirical case study. In *ICSE'08*.

Nikora, A. 2004. Personnel communication on the accuracy of severity determinations in nasa databases.

Nikora, A. and Munson, J. 2003. Developing fault predictors for evolving software systems. In *Ninth International Software Metrics Symposium (METRICS'03)*.

Ostrand, T. J., Weyuker, E. J., and Bell, R. M. 2004. Where the bugs are. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. ACM, New York, NY, USA, 86–96.

Porter, A. and Selby, R. 1990. Empirically guided software development using metric-based classification trees. *IEEE Software*, 46–54.

Pugh, W. 1990. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM 33,* 6, 668–676. Available from ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.pdf.

Quinlan, J. R. 1992a. Learning with Continuous Classes. In *5th Australian Joint Conference on Artificial Intelligence*. 343–348. Available from http://citeseer.nj.nec.com/quinlan92learning.html.

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

Raffo, D. 2005. Personnel communication.

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

Shepperd, M. and Ince, D. 1994. A critique of three metrics. *The Journal of Systems and Software 26,* 3 (September), 197–210.

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

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

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

T. Zimmermann, N. Nagappan, H. G. E. G. and Murphy, B. 2009. Cross-project defect prediction. In *ESEC/FSE'09*.

Tang, W. and Khoshgoftaar, T. M. 2004. Noise identification with the k-means algorithm. In *ICTAI*. 373–378.

Tian, J. and Zelkowitz, M. 1995. Complexity measure evaluation and selection. *IEEE Transaction on Software Engineering 21,* 8 (Aug.), 641–649.

Tosun, A., Bener, A., and Turhan, B. 2009. Practical considerations of deploying ai in defect prediction: A case study within the turkish telecommunication industry. In *PROMISE'09*.

Turhan, B., Menzies, T., Bener, A., and Distefano, J. 2009. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering 68,* 2, 278–290. Available from `http://menzies.us/pdf/08ccwc.pdf`.

Turner, J. 2006. A predictive approach to eliminating errors in software code. Available from `http://www.sti.nasa.gov/tto/Spinoff2006/ct_1.html`.

Voas, J. and Miller, K. 1995. Software testability: The new verification. *IEEE Software*, 17–28. Available from `http://www.cigital.com/papers/download/ieeesoftware95.ps`.

Weyuker, E., Ostrand, T., and Bell, R. 2008. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*.

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

Yang, Y., Webb, G. I., Cerquides, J., Korb, K. B., Boughton, J. R., and Ting, K. M. 2006. To select or to weigh: A comparative study of model selection and model weighing for spode ensembles. In *ECML*. 533–544.

APPENDIX

Learners Used in This Study

WHICH, *manualUp*, and *manualDown* was described above. The other learners used in this study come from the WEKA toolkit [Witten and Frank 2005] and are described below.

Naive Bayes classifiers, or NB, offer a relationship between fragments of evidence E_i, a prior probability for a posteriori probability an hypothesis given some evidence $P(H|E)$; and a class hypothesis $P(H)$ probability (in our case, we have two hypotheses: $H \in \{defective, nonDefective\}$). The relationship comes from Bayes Theorem: $P(H|E) = \prod_i P(E_i|H)\frac{P(H)}{P(E)}$ For numeric features, a feature's mean μ and standard deviation σ are used in a Gaussian probability function [Witten and Frank 2005]: $f(x) = 1/(\sqrt{2\pi}\sigma)e^{-\frac{(x-\mu)^2}{2\sigma^2}}$. Simple naive Bayes classifiers are called "naive" 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 [Domingos and Pazzani 1997]. This result explains (a) the repeated empirical result that, on average, seemingly näive Bayes classifiers perform as well as other seemingly more sophisticated schemes (e.g. see Table 1 in [Domingos and Pazzani 1997]); and (b) our prior experiments where naive Bayes did not perform worse than other learners that continually re-sample the data for dependent instances (e.g. decision-tree learners like C4.5 that recurse on each "split" of the data [Quinlan 1992b]).

This study used J48 [Witten and Frank 2005], a JAVA port of Quinlan's C4.5 decision tree learner C4.5, release 8 [Quinlan 1992b]. C4.5 is an *iterative dichotomiza-*

tion algorithm that seeks 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. C4.5 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 C4.5 stores the most frequent 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. C4.5 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.

JRip is a JAVA port of the RIPPER [Cohen 1995b] *rule-covering* algorithm. One rule is learned at each pass for one 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. JRip 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[Dietterich 1997]. 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 naive method).

Details on Static Code Features

This section offers some details on the Halstead and Mccabe features.

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 [Halstead 1977]. 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: $\mu_1^* = 2$;
—μ_2^* is the minimum operand count (number of module parameters).

An alternative to the Halstead features 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 [McCabe 1976]. The McCabe measures 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 [Fenton and Pfleeger 1995].

—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" [Fenton and Pfleeger 1995]). $ev(G) = v(G) - m$ where m is the number of subflowgraphs of G that are D-structured primes [Fenton and Pfleeger 1995].
—Finally, the *design complexity* ($iv(G)$) of a module is the cyclomatic complexity of a module's reduced flow graph.

Choice of Statistical Test

For several reasons, this study uses the Mann Whitney test. Firstly, many authors, including Demsar [Demsar 2006], remark that ranked statistical tests such as Mann-Whitney are not susceptible to errors caused by non-Gaussian performance distributions. Accordingly, we do not use t-tests since they make a Gaussian assumption.

Also, recall that Figure 9 shows the results of a two-stage process: first, select some detectors; second, rank them and watch the $effort$-vs-pd curve grow as we sweep right across Figure 9 (this two-stage process is necessary to baseline the learners against $manualUp$ and $manualDown$, as well as allowing us to express the results as the ratio of a *best* curve). The second stage of this process violates the paired assumptions of, say, the Wilcoxon tests since different test cases may appear depending on which modules are predicted to be defective. Accordingly, we require a non-paired test like Mann Whitney to compare distributions (rather than pairs of treatments applied to the same test case).

Further, while much has been written of the inadequacy of other statistical tests [Demsar 2006; Huang and Ling 2005], to the best of our knowledge, there is no current negative critique of Mann Whitney as a statistical test for data miners.

Lastly, unlike some other tests (e.g. Wilcoxon), Mann-Whitney does not demand that the two compared populations are of the same size. Hence, it is possible to run one test that compares each row of (e.g.) Figure 12 to every other row in the same division. This simplifies the presentation of the results (e.g. avoids the need for a display of, say, the Bonferroni-Dunn test shown in Figure 2 of Demsar [Demsar 2006]).

Received November 2009; revised April 2010; accepted May 2010.