Data Mining Static Code Attributes to Learn Defect Predictors

Tim Menzies, *Member*, *IEEE*, Jeremy Greenwald, and Art Frank

Abstract—The value of using static code attributes to learn defect predictors has been widely debated. Prior work has explored issues like the merits of "McCabes versus Halstead versus lines of code counts" for generating defect predictors. We show here that such debates are irrelevant since *how* the attributes are used to build predictors is much more important than *which* particular attributes are used. Also, contrary to prior pessimism, we show that such defect predictors are demonstrably useful and, on the data studied here, yield predictors with a mean probability of detection of 71 percent and mean false alarms rates of 25 percent. These predictors would be useful for prioritizing a resource-bound exploration of code that has yet to be inspected.

Index Terms—Data mining detect prediction, McCabe, Halstead, artifical intelligence, empirical, naive Bayes.

1 Introduction

Given recent research in artificial intelligence, it is now practical to use *data miners* to automatically learn predictors for software quality. When budget does not allow for complete testing of an entire system, software managers can use such predictors to focus the testing on parts of the system that seem defect-prone. These potential defect-prone trouble spots can then be examined in more detail by, say, model checking, intensive testing, etc.

The value of static code attributes as defect predictors has been widely debated. Some researchers endorse them ([1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20]) while others vehemently oppose them ([21], [22]).

Prior studies may have reached different conclusions because they were based on different data. This potential conflation can now be removed since it is now possible to define a *baseline experiment* using public-domain data sets[1] which different researchers can use to compare their techniques.

This paper defines and motivates such a baseline. The baseline *definition* draws from standard practices in the data mining community [23], [24]. To *motivate* others to use our definition of a baseline experiment, we must demonstrate that it can yield interesting results. The baseline experiment of this article shows that the rule-based or decision-tree learning methods used in prior work [4], [13], [15], [16], [25] are clearly outperformed by a *naive Bayes* data miner with a *log-filtering* preprocessor on the numeric data (the terms in italics are defined later in this paper).

Further, the experiment can explain *why* our preferred Bayesian method performs best. That explanation is quite technical and comes from information theory. In this introduction, we need only say that the space of "best" predictors is "brittle," i.e., minor changes in the data (such as a slightly different sample used to learn a predictor) can make different attributes appear most useful for defect prediction.

This brittleness result offers a new insight on prior work. Prior results about defect predictors were so contradictory since they were drawn from a large space of competing conclusions with similar but distinct properties. Different studies could conclude that, say, lines of code are a better/worse predictor for defects than the McCabes complexity attribute, just because of small variations to the data. Bayesian methods smooth over the brittleness problem by polling numerous Gaussian approximations to the numerics distributions. Hence, Bayesian methods do not get confused by minor details about candidate predictors.

Our conclusion is that, contrary to prior pessimism [21], [22], data mining static code attributes to learn defect predictors is useful. Given our new results on naive Bayes and log-filtering, these predictors are much better than previously demonstrated. Also, prior contradictory results on the merits of defect predictors can be explained in terms of the brittleness of the space of "best" predictors. Further, our baseline experiment clearly shows that it is a misdirected discussion to debate, e.g., "lines of code versus McCabe" for predicting defects. As we shall see, *the choice of learning method* is far more important than *which subset of the available data* is used for learning.

2 Background

For this study, we learn defect predictors from static code attributes defined by McCabe [2] and Halstead [1]. McCabe and Halstead are "module"-based metrics, where a module

1. http://mdp.ivv.nasa.gov and http://promise.site.uottawa.ca/SERepository.

- *T. Menzies is with the Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV 26506-610. E-mail: tim@menzies.us.*
- *J. Greenwald and A. Frank are with the Department of Computer Science, Portland State University, PO Box 751, Portland, OR 97207-0751. E-mail: jegreen@cecs.pdx.edu, arf@cs.pdx.edu.*

Manuscript received 2 Jan. 2006; revised 9 Aug. 2006; accepted 13 Sept. 2006; published online 30 Nov. 2006.
Recommended for acceptance by M. Harman.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0001-0106.

is the smallest unit of functionality.[2] We study defect predictors learned from static code attributes since they are *useful*, *easy to use*, and *widely used*.

Useful. This paper finds defect predictors with a probability of detection of 71 percent. This is markedly higher than other currently used industrial methods such as manual code reviews:

- A panel at *IEEE Metrics 2002* [26] concluded that manual software reviews can find ≈ 60 percent of defects.[3]
- Raffo found that the defect detection capability of industrial review methods can vary from

$$pd = TR(35, 50, 65)\%$$

for full Fagan inspections[4] [29] to

$$pd = TR(13, 21, 30)\%$$

for less-structured inspections.

Easy to use. Static code attributes like lines of code and the McCabe/Halstead attributes can be automatically and cheaply collected, even for very large systems [6]. 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 [30].

Widely used. Many researchers use static attributes to guide software quality predictions (see [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20]). Verification and validation (V&V) textbooks ([31]) advise using static code complexity attributes to decide which modules are worthy of manual inspections. For several years, T. Menzies worked on-site at the NASA software Independent Verification and Validation facility and he knows of several large government software contractors that will not review software modules *unless* tools like McCabe predict that they are fault prone.

Nevertheless, static code attributes are hardly a complete characterization of the internals of a function. Fenton and Pfleeger offer an insightful example where *the same* functionality is achieved using *different* programming language constructs resulting in *different* static measurements for that module [32]. They use this example to argue the uselessness of static code attributes.

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

Shepperd and Ince [22], as well as Fenton and Pfleeger, might reject the *alternative interpretation*. They present empirical evidence that the McCabe static attributes offer nothing more than uninformative attributes such as lines of code. Fenton and Pfleeger note that the main McCabe's attribute (cyclomatic complexity, or $v(g)$) is highly correlated with lines of code [32]. Also, Shepperd and Ince remark that "for a large class of software it (cyclomatic complexity) is no more than a proxy for, and in many cases outperformed by, lines of code."

	probability of	
data	detection	false alarm
pima diabetes	60	19
sonar	71	29
horse-colic	71	7
heart-statlog	73	21
rangeseg	76	30
credit rating	88	16
sick	88	1
hepatitis	94	56
vote	95	3
ionosphere	96	18
mean	81	20

Fig. 1. Some representative pds and pfs for prediction problems from the University of California Irvine machine learning database [33]. These values were generated using the standard settings of a state-of-art decision tree learner (J48). For each data set, 10 experiments were conducted where a decision tree was learned on 90 percent of the data, then tests of the remaining 10 percent. The numbers shown here are the average results across 10 such experiments.

If Shepperd and Ince and Fenton and Pfleeger are right, then

- the supposedly better static code attributes, such as Halstead and McCabes, should perform no better than just simple thresholds on lines of code, and
- the performance of a predictor learned by a data miner should be very poor.

Neither of these are true, at least for the data sets used in this study. Our experimental method seeks the "best" subsets of the available attributes that are most useful for predicting defects. We will show that the best size for the "best" set is larger than 1; i.e., predictors based on single lines of code counts do not perform as well as other methods.

Also, the predictors learned from those "best" sets perform surprisingly well. Formally, learning a defect predictor is a *binary prediction problem* where each module in a database has been labeled "defect-free" or "defective." The learning problem is to build some predictor which guesses the labels for as-yet-unseen modules. Using the methods described below, this paper offers new defect predictors with a probability of detection (pd) and probability of false alarm (pf) of

$$mean(pd, pf) = (71\%, 25\%).$$

Fig. 1 lets us compare our new results against standard binary prediction results from the University of California Irvine machine learning repository of standard test sets for data miners [33]. Our new results of $(pd, pf) = (71\%, 25\%)$ are close to the standard results of $(pd, pf) = (81\%, 20\%)$, which is noteworthy in four ways:

1. It is unexpected. If static code attributes capture so little about source code (as argued by Shepherd and

2. In other languages, modules may be called "function" or "method."

3. That panel supported neither Fagan's claim [27] that inspections can find 95 percent of defects before testing or Shull's claim that specialized directed inspection methods can catch 35 percent more defects than other methods [28].

4. $TR(a, b, c)$ is a triangular distribution with min/mode/max of a, b, c.

	probability of	
data	detection	false alarm
pc1	24	25
jm1	25	18
cm1	35	10
kc2	45	15
kc1	50	15
mean	36	17

Fig. 2. Prior results of learning defect predictors. From [4].

Ince and Fenton and Pfleeger), then we would expect lower probabilities of detection and much higher false alarm rates.

2. These new (pd, pf) figures are much larger than any of our prior results of $mean(pd, pf) = (36\%, 17\%)$ [4] (see Fig. 2). Despite much experimentation [14], [13], the only way we could achieve a $pd > 70\%$ was to accept a 50 percent false alarm rate.
3. These new results of $mean(pd) = 71\%$ are better than currently used industrial methods, such as the $pd \approx 60\%$ reported at the 2002 IEEE Metrics panel or the $median(pd) = 21..50$ reported by Raffo.
4. There is still considerable room for improvement, such as lower pfs and higher pds. We are actively researching better code metrics which, potentially, will yield "better" predictors.

This last point motivates much of this paper. Before we can demonstrate "better," we need to define "better than what?" That is, improvement can only be measured against a well-defined baseline result. That baseline needs to be repeatable and based on a public-domain data set. Further, the basis for comparatively assessing different data mining methods should be well-justified and well-specified so that others can repeat, improve, or refute prior results. Hence, much of the rest of this paper is devoted to a meticulous description of our experimental method.

The baseline experiment was selected in response to certain shortcomings in other work. For example, Nagappan and Ball [6, p. 6] report accuracies of 82.91 percent for their defect predictor. Accuracy attributes the number of times the predicted class of a module (defect-free or defective) is the same as the actual class. These accuracy values were found in a *self-test*; i.e., the learned predictor was applied to the data used to train it. In our study, we use neither accuracy nor self-tests:

- When the target class (defect-free or defective) is in the minority, accuracy is a poor measure of a learner's performance. For example, a learner could score 90 percent accuracy on a data set with 10 percent defective modules, even if it predicts that all defective modules were defect-free.
- Self-tests are deprecated by the data mining community since such self-tests can grossly overestimate performance [23]. If the goal is to understand how well a defect predictor will work on future projects, it is best to assess the predictor via *holdout* modules not used in the generation of that predictor.

Hence, for this study, we use attributes other than accuracy including pd, pf, and several others defined below. Also, our learned predictors will be assessed using *holdout* modules.

3 Threats to Validity

Like any empirical data mining paper, our conclusions are biased according to what data was used to generate them. Issues of *sampling bias* threaten any data mining experiment; i.e., what matters *there* may not be true *here*. For example, the sample used here comes from NASA, which works in a unique market niche.

Nevertheless, we argue that results from NASA are relevant to the general software engineering industry. NASA makes extensive use of contractors who are contractually obliged (ISO-9O01) to demonstrate their understanding and usage of current industrial best practices. These contractors service many other industries; for example, Rockwell-Collins builds systems for many government and commercial organizations. For these reasons, other noted researchers, such as Basili et al. [34], have argued that conclusions from NASA data are relevant to the general software engineering industry.

All inductive generalization suffers from a sampling bias. The best we can do is define our methods and publicize our data so that other researchers can try to repeat our results and, perhaps, point out a previously unknown bias in our analysis. Hopefully, other researchers will emulate our methods in order to repeat, refute, or improve our results. We would encourage such researchers to offer not just their conclusions, but the data used to generate those conclusions. The MDP is a repository for NASA data sets and the PROMISE code repository are places to store and discuss software engineering data sets from other organizations.

Another source of bias in this study is the set of learners explored by this study. Data mining is a large and active field and any single study can only use a small subset of the known data mining algorithms. For example, neural networks [35] and genetic algorithms [36] were not used for this study as they can be very slow. The experiment described in this paper took weeks to debug and a full day to run once debugged. We were therefore not motivated to explore other, slower learners but would encourage other researchers with access to supercomputers or a large CPU-farm to do so.

4 Data

An experiment needs three things:

- data to be processed,
- a processing method, and
- a reporting method.

This section discusses the data used in this study. Processing via data miners and our reporting methods are discussed later.

All our data comes from the MDP. At the time of this writing, 10 data sets are available in that repository. Two of those data sets have a different format from the rest and were not used in this study. This left eight, shown in Fig. 3. Each module of each data sets describes the attributes of

system	language	sub-system data set	total LOC	# modules	% defective
spacecraft instrument	C	cm1-05	17K	506	9
storage management for ground data	Java	kc3	8K	459	9
		kc4	25K	126	49
Db	C	mw1	8K	404	7
Flight software for earth orbiting satellite	C	pc1-05	26K	1,108	6
		pc2	25K	5,590	0.4
		pc3	36K	1,564	10
		pc4	30K	1,458	12

Fig. 3. Data sets used in this study. The data sets cm1-05 and pc1-05 update data sets $cm1$ and $pc1$ processed previously by the authors [15].

that module, plus the number of defects known for that module. This data comes from eight subsystems taken from four systems. These systems were developed in different geographical locations across North America. Within a system, the subsystems shared some a common code base but did not pass personnel or code between subsystems. Fig. 4 shows the module sizes of our data; for example, there are 126 modules in the $kc4$ data set; most of them are under 100 lines of code, but a few of them are more than 1,000 lines of code long.

Each data set was preprocessed by removing the module identifier attribute (which is different for each row). Also, the $error\_count$ column was converted into a Boolean attribute called $defective?$ as follows:

$$defective? = (error\_count \geq 1).$$

Finally, the $error\_density$ column was removed (since it can be derived from line counts and $error\_count$). The preprocessed data sets had 38 attributes plus one target attribute ($defective?$), shown in Fig. 5, and included Halstead, McCabe, lines of code, and other miscellaneous attributes.

The Halstead attributes were derived by Maurice Halstead in 1977. He argued that modules that are hard to read are more likely to be fault prone [1]. Halstead estimates reading complexity by counting the number of operators and operands in a module: See the h attributes of Fig. 5. These three raw h Halstead attributes were then used to compute the H: the eight derived Halstead attributes using the equations shown in Fig. 5. In between the raw and derived Halstead attributes are certain intermediaries (which do not appear in the MDP data sets):

- $\mu = \mu_1 + \mu_2$,
- minimum operator count: $\mu_1^* = 2$, and
- μ_2^* is the minimum operand count and equals the number of module parameters.

An alternative to the Halstead attributes are the complexity attributes proposed by Thomas McCabe in 1976. Unlike Halstead, McCabe argued that the complexity of pathways *between* module symbols is more insightful than just a count of the symbols [2]. The first three lines of Fig. 5 show McCabe's three main attributes for this pathway complexity. These are defined as follows: A module is said to have a *flow graph*; i.e., a directed graph where each node corresponds to a program statement and each arc indicates the flow of control from one statement to another. The

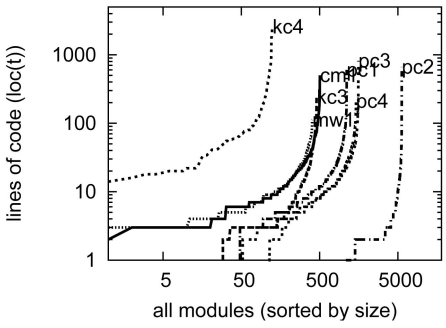

Fig. 4. Log-log plot of module sizes in the Fig. 3 data.

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
misc = Miscellaneous			branch_count
			call_pairs
			condition_count
			decision_count
			decision_density
			design_density
			edge_count
			global_data_complexity
			global_data_density
			maintenance_severity
			modified_condition_count
			multiple_condition_count
			node_count
			normalized_cyclomatic_complexity
			parameter_count
			pathological_complexity
			percent_comments

Fig. 5. Attributes used in this study.

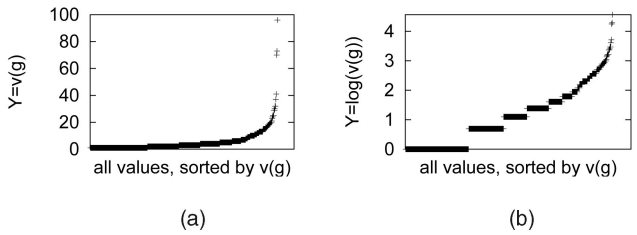

Fig. 6. $v(G)$ from $cm1$. (a) Raw values. (b) Log-filtered on right.

cyclomatic complexity of a module is $v(G) = e - n + 2$, where G is a program's flow graph, e is the number of arcs in the flow graph, and n is the number of nodes in the flow graph [37]. The *essential complexity* ($ev(G)$) of a module is the extent to which a flow graph can be "reduced" by decomposing all the subflowgraphs of G that are *D-structured primes* (also sometimes referred to as "proper one-entry one-exit subflowgraphs" [37]). $ev(G) = v(G) - m$, where m is the number of subflowgraphs of G that are D-structured primes [37]. Finally, the *design complexity* ($iv(G)$) of a module is the cyclomatic complexity of a module's reduced flow graph.

At the end of Fig. 5 are a set of *misc* attributes that are less well-defined than lines of code attributes or the Halstead and McCabe attributes. The meaning of these attributes is poorly documented in the MDP database. Indeed, they seem to be values generated from some unknown tool set that was available at the time of uploading the data into the MDP. Since there are difficulties in reproducing these attributes at other sites, an argument could be made for removing them from this study. A counterargument is that if static code attributes are as weak as suggested by Shepherd and Ince and Fenton and Pfleeger, then we should use all possible attributes in order to make maximum use of the available information. This study took a middle ground: All these attributes were passed to the learners and they determined which ones had the most information.

An interesting repeated pattern in our data sets are *exponential distributions* in the numeric attributes. For example, Fig. 6a shows the sorted McCabe $v(g)$ attributes from *cm1*. These values form an exponential distribution with many small values and a few much larger values. Elsewhere, we have conducted limited experiments suggesting that a *logarithmic filter* on all numeric values might improve predictor performance [14]. Such a filter replaces all numerics n with their logarithms. $ln(n)$. The effects of such a filter are shown in Fig. 6b: The log-filtered values are now more evenly spread across the y-range, making it easier to reason about them. To test the value of log-filtering, all the data was passed through one of two filters:

1. *none*; i.e., no change, or
2. *logNums*; i.e., logarithmic filtering. To avoid numerical errors with $ln(0)$, all numbers under 0.000001 are replaced with $ln(0.000001)$.

5 Learners

The above data was passed to three learners from the WEKA data mining toolkit [23]: OneR, J48, and naive Bayes.[5] This section describes those learners.

Holte's OneR algorithm [38] builds prediction rules using one or more values from a single attribute. For example, OneR executing on the *kc4* data set can return

```
EDGE_COUNT:
< 2.99 -> defect-free
>= 2.99 -> defective
```

which may be read as follows: "A module is defect-free if its edge count is less than 2.99." OneR was chosen to test the value of predictors based on *simple thresholds on single attributes*. For an example of such a simple threshold, recall that McCabe recommends inspected modules that satisfy $v(g) > 10$ or $iv(g) > 4$. Several other example thresholds exist in defect prediction literature:

- Chapman and Solomon advocate predicting defects using $v(g) > 20$ or $ev(g) > 8$ [3].
- In early work [14], we advocated $ev(g) > 7$ or $\text{lines of code} > 118$ (based on this study, we now reject that advice).

OneR can only return simple thresholds on single attributes. If predictors built by OneR were as good as any other, then that would support the use of simple thresholds, such as those advocated by McCabe et al.

One way to view OneR's defect predictions rules is a decision tree of maximum depth 1 whose leaves are either the label *defective* or *defect-free*. The J48 learner builds decision trees of any depth. For example, J48 executing on the *kc4* data set can return

```
CALL_PAIRS <= 0: defect-free
CALL_PAIRS > 0
| NUMBER_OF_LINES <= 3.12: defect-free
| NUMBER_OF_LINES > 3.12
| | NORMALIZED_CYLOMATIC_COMPLEXITY <= 0.02
| | | NODE_COUNT <= 3.47: defective
| | | NODE_COUNT > 3.47: defect-free
| | NORMALIZED_CYLOMATIC_COMPLEXITY > 0.02:
    defective
```

which may be read as follows: "A module is defective if it has nonzero call-pairs and has more than 3.12 lines and does not have a low normalized cyclomatic complexity (0.02) or it has a low normalized cyclomatic complexity and a low node-count (up to 3.47)." Note that J48 predictors can be more complex and explore more special cases than OneR. J48's predictors would outperform OneR if defect prediction required such elaboration.

J48 is a JAVA implementation of Quinlan's C4.5 (version 8) algorithm [39]. The algorithm recursively *splits* a data set according to tests on attribute values in order to separate the possible predictions (although attribute tests are chosen one at a time in a greedy manner, they are dependent on results of previous tests). C4.5/J48 uses information theory to assess candidate splits: The *best split* is

5. http://www.cs.waikato.ac.nz/~ml/weka.

the one that *most simplifies* the target concept. Concept simplicity is measured using information theory. Suppose a data set has 80 percent defect-free modules and 20 percent defective modules. Then, that data set has a class distribution C_0 with classes $c(1) = defect - free$ and $c(2) = defective$ with frequencies $n(1) = 0.8$ and $n(2) = 0.2$. The number of bits required to encode an arbitrary class distribution C_0 is $H(C_0)$, defined as follows:

$$\left.\begin{array}{rcl} N & = & \sum_{c \in C} n(c) \\ p(c) & = & n(c)/N \\ H(C) & = & -\sum_{c \in C} p(c) log_2 p(c) \end{array}\right\}. \quad (1)$$

A split divides C_0 (before the split) into C_1 and C_2 (after the split). The best split leads to the simplest concepts; i.e., maximize $H(C_0) - (H(C_1) + H(C_2))$.

Another way to build defect predictors is to use a naive Bayes data miner. Such classifiers are based on Bayes' Theorem. Informally, the theorem says $next = old * new$; i.e., what we'll believe *next* comes from how *new* evidence affects *old* beliefs. More formally,

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

i.e., given fragments of evidence E_i and a prior probability for a class $P(H)$, the theorem lets us calculate a posteriori probability $P(H|E)$. When building defect detectors, the posterior probability of each class ("defective" or "defect-free") is calculated, given the attributes extracted from a module such as the lines of code, the McCabe attributes, the Halstead attributes, etc. The module is assigned to the possibility with the highest probability. This is straightforward processing and involves simply estimating the probability of attribute measurements within the historical modules. Simple frequency counts are used to estimate the probability of discrete attribute attributes. For numeric attributes, it is common practice to use the probability density function for a normal distribution [23]:

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

where $\{\mu, \sigma\}$ are the attributes {mean, standard deviation}. To be precise, the probability of a continuous attribute being a particular continuous value x is zero, but the probability that it lies within a small region, say $x \pm \epsilon/2$, is $\epsilon \times f(x)$. Since ϵ is a constant that weighs across all possibilities, it cancels out and needs not to be computed.

The above learning technology can be used to generate defect predictors from data *or* to assess the value of different portions of the data. Various *attribute subset selection* algorithms [24] (hereafter, *subsetting*) find what attributes can be deleted without damaging the performance of the learned predictor. Subsetting can be used independently of the learning technique of choice as a general method for data reduction.

The simplest and fastest subsetting method is to rank attributes from the most informative to least informative. After discretizing numeric data,[6] then if A is a set of attributes, the number of bits required to encode a class after observing an attribute is

$$H(C|A) = -\sum_{a \in A} p(a) \sum_{c \in C} p(c|a) log_2(p(c|a).$$

The highest ranked attribute A_i is the one with the largest *information gain*, i.e., the one that most reduces the encoding required for the data *after* using that attribute:

$$InfoGain(A_i) = H(C) - H(C|A_i), \quad (2)$$

where $H(C)$ comes from (1). In *iterative InfoGain subsetting*, predictors are learned using the $i = 1, 2, \ldots, N$th top-ranked attributes. Subsetting terminates when $i + 1$ attributes perform no better than i. In *exhaustive InfoGain subsetting*, the attributes are first ranked using iterative subsetting. Next, predictors are built using all subsets of the top j ranked attributes. For both iterative and exhaustive subsetting, the process is repeated 10 times using 90 percent of the data (randomly selected). Iterative subsetting takes time linear on the number of attributes N while exhaustive subsetting takes time 2^j (so it is only practical for small $j \leq N$).

```
M    = 10
N    = 10
All  = 38  # all the attributes
DATAS=(cm1 kc3 kc4 mw1 pc1 pc2 pc3 pc4) # data set list
FILTERS= (none logNums)                 # filter list
LEARNERS= (oneR j48 nb)                 # learner list

for data in DATAS
  for filter in FILTERS
      data' = filter(data)
      rank data' attributes via InfoGain # Equation 2
      for i = 1,2,3, All
         attribute' = the i-th highest ranked attributes
         data''     = select attributes' from data'
         repeat M times
             randomized order from data''
             generate N bins from data''
             for i in 1 to N
               tests        = bin[i]
               trainingData = data'' - tests
               for learner in LEARNERS
                  METHOD         = (filter attributes' learner)
                  predictor      = learner(trainingData)
                  RESULT[METHOD] = apply predictor to tests
```

Fig. 7. This study. Data is filtered and the attributes are ranked using InfoGain. The data is then shuffled into a random order and divided into 10 bins. A learner is then applied to a *training set* built from nine of the bins. The learned predictor is tested on the remaining bin.

6 Experimental Design

This study used the $(M = 10) * (N = 10)$-*way cross-evaluation iterative attribute subset selection* shown in Fig. 7. The study is nearly the same as the procedure defined in Hall and Holmes' subsetting experiments [24] (but we have added a data filtering step). The data set is divided into N buckets. For each bucket in a 10-way cross-evaluation, a predictor is learned on nine of the buckets, then tested on the remaining bucket.

Hall and Holmes advise repeating an N-way study M times, randomizing the order each time. Many algorithms exhibit *order effects*, where certain orderings dramatically improve or degrade performance [41] (insertion sort runs slowest if the inputs are already sorted in reverse

6. For example, given an attribute's minimum and maximum values, replace a particular value n with $(n - min)/((max - min)/10)$. For more on discretization, see [40].

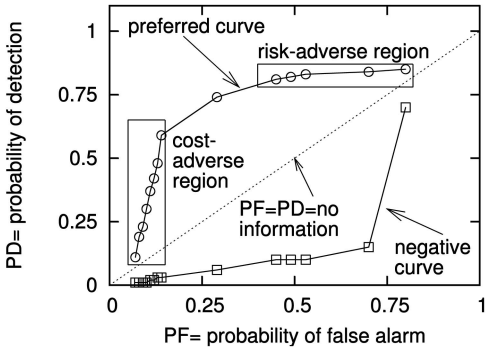

Fig. 8. Regions of a typical ROC curve.

order). Randomizing the order of the inputs defends against order effects.

These $\mathrm{M} * \mathrm{N}$ studies implement a *holdout* study which, as argued above, is necessary to properly assess the value of a learned predictor. Holdout studies assess a learned predictor using data *not* used to generate it. Such holdout studies are the preferred evaluation method when the goal is to produce predictors intended to predict future events [23].

The $10 * 10$-way study was wrapped inside scripts that explored different subsets of the attributes in the order suggested by *InfoGain* (2). In the innermost loop of the study, some *method* was applied to some data set. As shown in the third to the last line of Fig. 7, these *methods* were some combination of *filter*, *attributes'*, and *learner*.

7 Assessing Performance

The performance of the learners on the MDP data was assessed using *receiver-operator* (ROC) curves. Formally, a defect *predictor* hunts for a *signal* that a software module is defect prone. Signal detection theory [42] offers ROC curves as an analysis method for assessing different predictors. A typical ROC curve is shown in Fig. 8. The y-axis shows probability of detection (pd) and the x-axis shows probability of false alarms (pf). By definition, the ROC curve must pass through the points $pf = pd = 0$ and $pf = pd = 1$ (a predictor that never triggers never makes false alarms; a predictor that always triggers always generates false alarms). Three interesting trajectories connect these points:

1. A straight line from (0, 0) to (1, 1) is of little interest since it offers *no information*; i.e., the probability of a predictor firing is the same as it being silent.
2. Another trajectory is the *negative curve* that bends away from the ideal point. Elsewhere [14], we have found that if predictors negate their tests, the negative curve will transpose into a *preferred curve*.
3. The point ($pf = 0$, $pd = 1$) is the ideal position (a.k.a. "sweet spot") on a ROC curve. This is where we recognize all errors and never make mistakes. *Preferred curves* bend up toward this ideal point.

In the ideal case, a predictor has a high probability of detecting a genuine fault (pd) and a very low probability of false alarm (pf). This ideal case is very rare. The only way to achieve high probabilities of detection is to trigger the predictor more often. This, in turn, incurs the cost of more false alarms.

		module found in defect logs?	
		no	yes
signal detected?	no (i.e. $v(g) < 10$)	A = 395	B = 67
	yes (i.e. $v(g) \geq 10$)	C = 19	D = 39

$$pd = \quad Prop.detected = \quad 37\%$$
$$pf = \quad Prob.falseAlarm = \quad 5\%$$
$$notPf = \quad 1 - pf = \quad 95\%$$
$$bal = \quad Balance = \quad 45\%$$
$$Acc = \quad accuracy = \quad 83\%$$

Fig. 9. A ROC sheet assessing the predictor $v(g) \geq 10$. Each cell {A,B,C,D} shows the number of modules that fall into each cell of this ROC sheet. The bal (or balance) variable is defined below.

Pf and pd can be calculated using the ROC sheet of Fig. 9. Consider a predictor which, when presented with some signal, either triggers or is silent. If some oracle knows whether or not the signal is actually present, then Fig. 9 shows four interesting situations. The predictor may be silent when the signal is absent (cell A) or present (cell B). Alternatively, if the predictor registers a signal, sometimes the signal is actually absent (cell C) and sometimes it is present (cell D).

If the predictor registers a signal, there are three cases of interest. In one case, the predictor has correctly recognized the signal. This probability of this detection is the ratio of detected signals, true positives, to all signals:

$$\text{probability detection} = pd = recall = D/(B + D). \quad (3)$$

(Note that pd is also called $recall$.) In another case, the probability of a false alarm is the ratio of detections when no signal was present to all nonsignals:

$$\text{probability false alarm} = pf = C/(A + C). \quad (4)$$

For convenience, we say that $notPf$ is the complement of pf:

$$\text{notPf} = 1 - C/(A + C). \quad (5)$$

Fig. 9 also lets us define the *accuracy*, or acc, of a predictor as the percentage of true negatives and true positives:

$$\text{accuracy} = acc = (A + D)/(A + B + C + D). \quad (6)$$

If reported as percentages, these attributes have the range

$$0 \leq \ acc\%, \ pd\%, \ , notPf\% \leq 100.$$

Ideally, we seek predictors that maximize *acc percent*, *pd percent*, and *notPf percent*.

Note that maximizing any one of these does not imply high values for the others. For example, Fig. 9 shows an example with a high accuracy (83 percent) but a low probability of detection (37 percent). Accuracy is a good measure of a learner's performance when the possible outcomes occur with similar frequencies. The data sets used in this study, however, have very uneven class distributions (see Fig. 3). Therefore, this paper will assess its learned predictors using bal, pd, and $notPf$ and not acc.

In practice, engineers *balance* between pf and pd. To operationalize this notion of *balance*, we define bal to be the

Euclidean distance from the sweet spot $pf = 0, pd = 1$ to a pair of $< pf, pd >$. For convenience, we 1) normalize bal by the maximum possible distance across the ROC square ($\sqrt{2}$), 2) subtract this from 1, and 3) express it as a percentage; i.e.

$$balance = bal = 1 - \frac{\sqrt{(0-pf)^2+(1-pd)^2}}{\sqrt{2}}. \quad (7)$$

Hence, better and *higher* balances fall *closer* to the desired sweet spot of $pf = 0, pd = 1$.

8 Quartile Charts of Performance Deltas

Recall from Fig. 7 that a *method* is some combination of *filter*, *attributes'*, and *learner*. The experiment described above explored numerous combinations of filters, attributes, and learners all within a $(M = 10) * (N = 10)$-way cross-evaluation study. Hence, this experiment generated nearly 800,000 *performance deltas* (defined below) comparing pd, $notPf$, and bal values from different *methods* applied to the same $test$ data.

The performance deltas were computed using simple subtraction, defined as follows: A *positive performance delta* for method X means that method X has outperformed some other method in *one* comparison. Using performance deltas, we say that the best method is the one that generates the largest performance deltas over *all* comparisons.

The performance deltas for each method were sorted and displayed as *quartile charts*. To generate these charts, the performance deltas for some $method$ were sorted to find the lowest and highest quartile as well as the median value, e.g.,

$$\overbrace{\underbrace{-59}_{min}, -19, -19}^{lowest\ quartile}, -16, -14, -10, \underbrace{-10}_{median}, 5, 14, 39, \overbrace{42, 62, \underbrace{69}_{max}}^{highest\ quartile}.$$

In a quartile chart, the upper and lower quartiles are marked with black lines, the median is marked with a black dot, and vertical bars are added to mark 1) the zero point, 2) the minimum possible value, and 3) the maximum possible value (in our case, -100 percent and 100 percent). The above numbers would therefore be drawn as follows:

$$-100\%| \quad \text{——} \quad \bullet| \quad \text{——} \quad |100\%.$$

We prefer quartile charts of performance deltas to other summarization methods for $\mathrm{M} * \mathrm{N}$ studies. First, they offer a very succinct summary of a large number of experiments. For example, Fig. 10 displays 200,000 performance deltas in $\frac{1}{4}$ of page. Second, they are *nonparametric* displays; i.e., they make no assumptions about the underlying distribution. Standard practice in data mining is to compare the mean performance of different methods using t-tests [23]. T-tests are a *parametric method* that assume that the underlying population distribution is a Gaussian. Recent results suggest that there are many statistical issues left to explore regarding how to best to apply those t-tests for summarizing $\mathrm{M} * \mathrm{N}$-way studies [43]. Such t-tests assume Gaussian distributions and some of our results are clearly non-Gaussian:

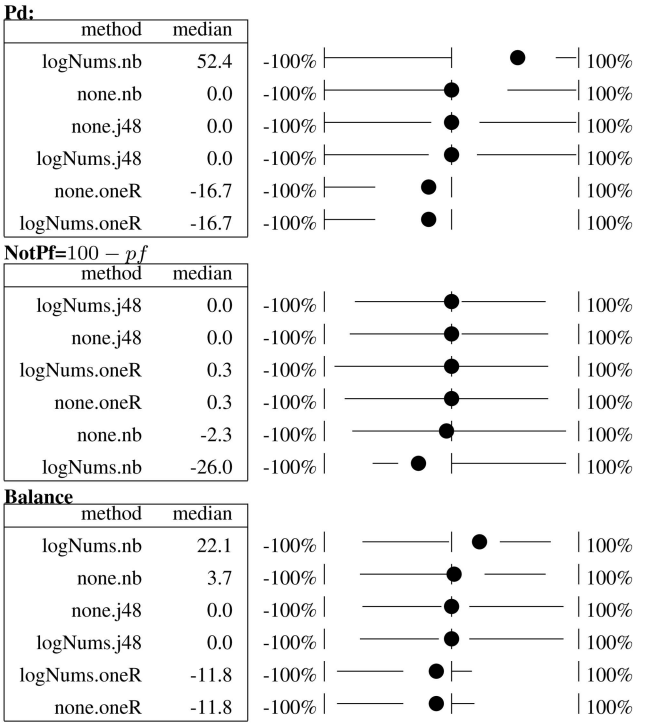

Fig. 10. Performance deltas for pd, $notPf$, and bal using all 38 attributes.

- The naive Bayes performance delta pd results (using $logNums$) of Fig. 10 exhibit an extreme skewness (a median point at 52.4 with a quarter of the performance deltas pushed up toward the maximum figure of 100 percent).
- All the OneR performance delta pd results of Fig. 10 are highly skewed. OneR's pd performance delta was *never* higher than 16.7 and over half the performance deltas for that method had that value (hence, the missing upper arms in the OneR results of Fig. 10).

For the sake of completeness, we applied t-tests when sorting quartile charts: One quartile chart appears above its neighbor if it was statistically different (at the 95 percent confidence level) and has a larger mean. However, given the skews we are seeing in the data, we base our conclusions on *standout* effects seen in the nonparametric quartile diagrams. A *standout* effect is a large and positive median with a highest quartile bunched up toward the maximum figure. The pd results for naive Bayes (with $logNums$) are an example of such a standout effect. On the other hand, OneR's pd results are a *negative standout*: Those performance deltas tend to bunch down toward -100 percent; i.e., in terms of pd, OneR usually performs much worse than anything else.

9 Results

Naive Bayes with a log-transform has both a positive standout result for pd and a negative standout result for $notPf$. This result, of winning on pd but losing on pf, is to be expected. Fig. 8 showed that the cost of high pds are higher pfs. The other learning methods cannot emulate the high

Balance:

#attributes	median		
3	10.0	-100%	100%
38	8.0	-100%	100%
2	0.0	-100%	100%
1	-30.7	-100%	100%

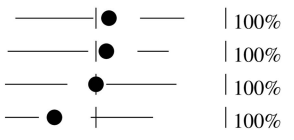

Fig. 11. On balance performance deltas of naive Bayes (with logNums) using just the best one, two, or three attributes, or all 38 attributes.

pds of naive Bayes (with log-transforms) since they take fewer chances (hence, have lower false alarm rates).

The *balance* results of Fig. 10 combines the pd and pf results using (7). On balance, with 38 attributes:

- OneR loses more often than it wins: Observe that OneR has a negative median $balance$.
- The best method, on balance, is clearly naive Bayes with log-transforms since it has a minority of negative balance performance deltas (only 25 percent), and it beats other methods by 22.1 percent (or more) half the time.

A review of the J48 and OneR quartile charts in Fig. 10 shows that J48 outperforms OneR in terms of pd, $notPf$ and bal. That is, for these data sets, predictors that use simple threshold comparisons (OneR) perform worse than predictors built from more elaborate decision trees (J48).

Since, on balance *logNums.nb* performs best, the rest of this article only presents the subsetting results for that method. Initial experiments with iterative InfoGain subsetting showed that all but one of the data sets (pc1) could be reduced from 38 to 3 attributes without degrading the on-balance performance. However, iterative subsetting selected seven attributes for $PC1$. Therefore, for that data set only, exhaustive subsetting was performed on 2^7 subsets to find the three best attributes.

These InfoGain results were then compared to various other subsetting methods: CFS [44]; Relief [45], [46]; and CBS [47]. Measured in terms of pd, $notPf$, $balance$, or number of selected attributes, there was no apparent advantage in using these other subsetting methods instead of InfoGain.

Fig. 11 shows the InfoGain results for naive Bayes with logNums. On balance, large reductions in the number of attributes are possible without compromising the performance of the learned predictor. Using 2 or 3 attributes worked as well as using 38 attributes. However, using only one attribute resulted in inferior performance.

All the results up to this point have been *comparisons between* different methods. Having determined that naive Bayes (with logNums) is our preferred method, the next question is how well does that method perform in *absolute* terms. To test that, in Fig. 12, a standard $10 * 10$-way experiment with attribute subset-selection was performed (hence, each line in Fig. 12 shows the results of $10 * 10 = 100$ experiments using just the two or three attributes shown in the *selected attributes* column of Fig. 12). On average, naive Bayes (with logNums) built predictors with mean $pd = 71\%$, and mean $pf = 25\%$.

		%		selected attributes	selection
data	N	pd	pf	(seeFigure 13)	method
pc1	100	48	17	3, 35, 37	exhaustive subsetting
mw1	100	52	15	23, 31, 35	iterative subsetting
kc3	100	69	28	16, 24, 26	iterative subsetting
cm1	100	71	27	5, 35, 36	iterative subsetting
pc2	100	72	14	5, 39	iterative subsetting
kc4	100	79	32	3, 13, 31	iterative subsetting
pc3	100	80	35	1, 20, 37	iterative subsetting
pc4	100	98	29	1, 4, 39	iterative subsetting
all	800	71	25		

Fig. 12. Best defect predictors learned in this study. Mean results from Näive Bayes after a 10 repeats of 1) randomize the order of the data; 2) divide that data into 10 90 percent:10 percent splits for training:test. Prior to learning, all numerics where replaced with logarithms. InfoGain was then used to select the best two or three attributes shown in the right-hand column (and if "three" performed as well as "two," then this table shows the results using "two").

The Fig. 12 results are better than they first appear:

- Recall from Fig. 3 that the number of defective modules may be very small: The most extreme example of this is $PC2$ with only 0.4 percent defective modules. It is somewhat of an achievement that, for $PC2$, our methods yielded $\{pd = 72\%, pf = 14\%\}$ for such a tiny target.
- The best we have achieved in the past with cross-validation was a mean pd under 50 percent [48] (recall Fig. 2). In those experiments, the only way to achieve a $pd > 70\%$ was to accept around a 50 percent false alarm rate [13], [14]. The results of Fig. 11 results have much higher pds and lower pfs.

One interesting aspect of Fig. 12 is that different data sets selected very different "best" attributes (see the *selected attribute* column (see the selected attribute column of Fig. 12 and Fig. 13)). This aspect can be explained by Fig. 14, which shows the InfoGain of all the attributes in an MDP data set (KC3). Note how the highest ranked attributes (those on the left) offer very similar information. That is, there are no clear winners, so minor changes in the training sample (the 90 percent subsampling used in subsetting or a cross-validation study) can result in the selection of very different "best" attributes.

The pattern of InfoGain values of Fig. 14 (where there are many alternative "best" attributes) repeats in all the MDP

ID	frequency in Figure 12	what	type
1	2	loc_blanks	locs
3	2	call_pairs	misc
4	1	loc_code_and_command	locs
5	2	loc_comments	locs
13	1	edge_count	misc
16	1	loc_executable	locs
20	1	I	H (derived Halstead)
23	1	B	H (derived Halstead)
24	1	L	H (derived Halstead)
26	1	T	H (derived Halstead)
31	2	node_count	misc
35	3	μ_2	h (raw Halstead)
36	1	μ_1	h (raw Halstead)
37	2	number_of_lines	locs
39	2	percent_comments	misc

Fig. 13. Attributes used in Fig. 12, sorted into the groups of Fig. 5.

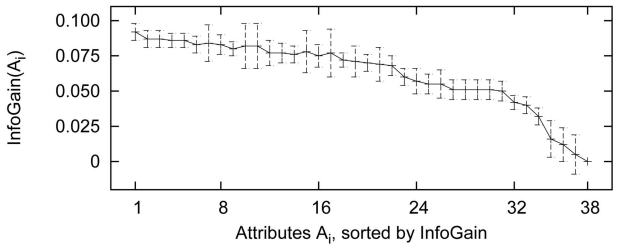

Fig. 14. InfoGain for KC3 attributes. Calculated from (2). Lines show means and t-bars show standard deviations after 10 trials on 90 percent of the training data (randomly selected).

data sets. This pattern explains a prior observation of Shepperd and Ince, who found 18 publications of which an equal number of studies reported that the McCabe cyclomatic complexity is the same, is better, or is worse than lines of code in predicting defects [22]. Fig. 14 motivates the following principles:

- Do not seek "best" subsets of static code attributes.
- Rather, seek learning methods that can combine multiple partial defect indicators, like the statistical methods of naive Bayes.

10 Conclusion

These results strongly endorse building defect predictors using niave Bayes (with logNums). The combination of $\text{learner} + \text{filter}$ generated predictors with average results of $pd = 71\%$ and $pf = 25\%$ (see Fig. 12). This is an interesting result since, as mentioned above, if static code attributes capture so little about source code (as argued by Shepherd and Ince and Fenton and Pfleeger), then we would expect much lower probabilities of detection and much higher false alarm rates.

Our results also comment on the relative merits of certain learners. Based on these experiments, we would reject the use of simple thresholds for defect prediction. If simple thresholds such as $v(g) > 10 \vee iv(g) > 4$ were the best defect predictors, then two results would be predicted. First, the single attribute tests of OneR would perform as well as the multiple tests of J48. Second, the subsetting methods would select attribute sets of size 1. Neither of these results were seen in Fig. 10 and Fig. 11.

This experiment was also negative regarding the merits of building intricate decision trees to predict defects. Recalling Fig. 10, naive Bayes (with logNums) outperformed J48. We offer two explanations why naive Bayes with logNums outperforms our prior work:

- Recalling Fig. 6, it is possible that code defects are actually associated in some log-normal way to static code attributes. Of all the methods studied here, only naive Bayes (with *logNums*) was able to directly exploit this association.
- Recalling Fig. 14, many of the static code attributes have similar information content. Perhaps defect detection is best implemented as some kind of thresholding systems, i.e., by summing the signal from several partial indicators. Of all the learners used in this study, only the statistical approach of naive Bayes can sum information from multiple attributes.

The best attributes to use for defect prediction vary from data set to data set. Hence, rather than advocating a particular subset of possible attributes as being the *best attributes*, these experiments suggest that defect predictors should be built using *all available* attributes, followed by subsetting to find the most appropriate particular subset for a particular domain.

In summary, we endorse the use of static code attributes for predicting detects with the following caveat: Those predictors should be treated as probabilistic, not categorical, indicators. While our best methods have a nonzero false alarm, they also have a usefully high probability of detection (over two-thirds). Just as long as users treat these predictors as *indicators* and not definite *oracles*, then the predictors learned here would be pragmatically useful for focusing limited verification and validation budgets on portions of the code base that are predicted to be problematic.

Since we are optimistic about using static code attributes, we need to explain prior pessimism about such attributes [21], [22]:

- Prior work would not have found good predictors if that work had focused on attribute subsets rather than the learning methods. Fig. 12 shows that the best attribute subsets for defects predictors can change dramatically from data set to data set. Hence, conclusions regarding the *best attribute(s)* are very brittle, i.e., may not still apply when we change data sets.
- Also, prior work would not have found good predictors if that work had not explored a large space of learning methods. For example, Fig. 10 shows that, of the six methods explored here, only *one* (naive Bayes with logNums) had a median performance that was both large and positive.

More generally, our high-level conclusion is that it is no longer adequate to assess defect learning methods using only one data set and only one learner. Further research should assess the merits of their proposed techniques via extensive experimentation.

11 Future Work

Our hope is that numerous researchers repeat our experiments and discover learning methods that are superior to the one proposed here. Paradoxically, this paper will be a success if it is quickly superseded.

There are many ways to design learning methods that could outperform the results of this paper. Here, we list just three:

- Data mining is a dynamic field and new data miners and continually being developed. For example, Webb et al. have proposed an improvement to naive Bayes that aggregates 1-dependence estimators [49]. It would be interesting to check if these newer learners improved the results of this paper.

- With regard to preprocessing the numerics, we might be able to do even better than our current results. Dougherty et al. [40] report spectacular improvements in the performance of naive Bayes via the use of better numeric preprocessing than just simple log-filtering.
- The Halstead and McCabe attributes were defined in the 1970s and complier technology has evolved considerably since then. Halstead and McCabe are *intramodule* metrics and, with modern intraprocedural data flow analysis, it should be possible to define a new set of 21st-century *intermodule* metrics that yield better defect predictors.

ACKNOWLEDGMENTS

The research described in this paper was carried out at West Virginia University and Portland State University under contracts and subcontracts with NASA's Software Assurance Research Program and the Galaxy Global Corporation, Fairmont West Virginia. 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. An earlier draft of this paper is available at http://menzies.us.pdf/06learnPredict.pdf.

REFERENCES

[1] M. Halstead, *Elements of Software Science.* Elsevier, 1977.
[2] T. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.,* vol. 2, no. 4, pp. 308-320, Dec. 1976.
[3] M. Chapman and D. Solomon, "The Relationship of Cyclomatic Complexity, Essential Complexity and Error Rates," *Proc. NASA Software Assurance Symp.,* http://www.ivv.nasa.gov/business/research/osmasas/conclusion2002/Mike_Chapman_The_Relationship_of_Cyclomatic_Complexity_Essential_Complexity_and_Error_Rates.ppt, 2002.
[4] T. Menzies, J. DiStefano, A. Orrego, and R. Chapman, "Assessing Predictors of Software Defects," *Proc. Workshop Predictive Software Models,* 2004.
[5] Polyspace verifier®, http://www.polyspace.com/, 2005.
[6] N. Nagappan and T. Ball, "Static Analysis Tools as Early Indicators of Pre-Release Defect Density," *Proc. Int'l Conf. Software Eng.,* 2005.
[7] G. Hall and J. Munson, "Software Evolution: Code Delta and Code Churn," *J. Systems and Software,* pp. 111-118, 2000.
[8] A. Nikora and J. Munson, "Developing Fault Predictors for Evolving Software Systems," *Proc. Ninth Int'l Software Metrics Symp. (METRICS '03),* 2003.
[9] N. Nagappan and T. Ball, "Static Analysis Tools as Early Indicators of Pre-Release Defect Density," *Proc. Int'l Conf. Software Eng.,* pp. 580-586, 2005.
[10] T. Khoshgoftaar, "An Application of Zero-Inflated Poisson Regression for Software Fault Prediction," *Proc. 12th Int'l Symp. Software Reliability Eng.,* pp. 66-73, Nov. 2001.
[11] W. Tang and T.M. Khoshgoftaar, "Noise Identification with the K-Means Algorithm," *Proc. Int'l Conf. Tools with Artificial Intelligence (ICTAI),* pp. 373-378, 2004.
[12] T.M. Khoshgoftaar and N. Seliya, "Fault Prediction Modeling for Software Quality Estimation: Comparing Commonly Used Techniques," *Empirical Software Eng.,* vol. 8, no. 3, pp. 255-283, 2003.
[13] T. Menzies, J.D. Stefano, K. Ammar, K. McGill, P. Callis, R. Chapman, and J. Davis, "When Can We Test Less?" *Proc. IEEE Software Metrics Symp.,* 2003.
[14] T. Menzies, J.S. DiStefeno, M. Chapman, and K. Mcgill, "Metrics that Matter," *Proc. 27th NASA SEL Workshop Software Eng.,* 2002.
[15] T. Menzies, J.D. Stefano, and M. Chapman, "Learning Early Lifecycle IV and V Quality Indicators," *Proc. IEEE Software Metrics Symp.,* 2003.
[16] T. Menzies and J.S.D. Stefano, "How Good Is Your Blind Spot Sampling Policy?" *Proc. 2004 IEEE Conf. High Assurance Software Eng.,* 2003.
[17] A. Porter and R. Selby, "Empirically Guided Software Development Using Metric-Based Classification Trees," *IEEE Software,* pp. 46-54, Mar. 1990.
[18] J. Tian and M. Zelkowitz, "Complexity Measure Evaluation and Selection," *IEEE Trans. Software Eng.,* vol. 21, no. 8, pp. 641-649, Aug. 1995.
[19] T. Khoshgoftaar and E. Allen, "Model Software Quality with Classification Trees," *Recent Advances in Reliability and Quality Eng.,* pp. 247-270, 2001.
[20] K. Srinivasan and D. Fisher, "Machine Learning Approaches to Estimating Software Development Effort," *IEEE Trans. Software Eng.,* pp. 126-137, Feb. 1995.
[21] N. Fenton and N. Ohlsson, "Quantitative Analysis of Faults and Failures in a Complex Software System," *IEEE Trans. Software Eng.,* pp. 797-814, Aug. 2000.
[22] M. Shepped and D. Ince, "A Critique of Three Metrics," *J. Systems and Software,* vol. 26, no. 3, pp. 197-210, Sept. 1994.
[23] I.H. Witten and E. Frank, *Data Mining,* second ed. Morgan Kaufmann, 2005.
[24] M. Hall and G. Holmes, "Benchmarking Attribute Selection Techniques for Discrete Class Data Mining," *IEEE Trans. Knowledge and Data Eng.,* vol. 15, no. 6, pp. 1437-1447, June 2003.
[25] T. Menzies "21st Century AI: Proud, Not Smug," *IEEE Intelligent Systems,* vol. 18, no. 3, pp. 18-24, May-June 2003, http://menzies.us/pdf/03airpride.pdf.
[26] F. Shull, V.B. Boehm, A. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What We Have Learned About Fighting Defects," *Proc. Eighth Int'l Software Metrics Symp.,* pp. 249-258, 2002.
[27] M. Fagan, "Advances in Software Inspections," *IEEE Trans. Software Eng.,* pp. 744-751, July 1986.
[28] F. Shull, I. Rus, and V. Basili, "How Perspective-Based Reading Can Improve Requirements Inspections," *IEEE Computer,* vol. 33, no. 7, pp. 73-79, July 2000, http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/82.77.pdf.
[29] M. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems J.,* vol. 15, no. 3, 1976.
[30] T. Menzies, D. Raffo, S. Setamanit, Y. Hu, and S. Tootoonian, "Model-Based Tests of Truisms," *Proc. IEEE Automated Software Eng. Conf.,* 2002.
[31] S. Rakitin, *Software Verification and Validation for Practitioners and Managers,* second ed. Artech House, 2001.
[32] N.E. Fenton and S. Pfleeger, *Software Metrics: A Rigorous and Practical Approach.* Int'l Thompson Press, 1997.
[33] C. Blake and C. Merz, "UCI Repository of Machine Learning Databases," http://www.ics.uci.edu/~mlearn/MLRepository.html, 1998.
[34] V. Basili, F. McGarry, R. Pajerski, and M. Zelkowitz, "Lessons Learned from 25 Years of Process Improvement: The Rise and Fall of the NASA Software Engineering Laboratory," *Proc. 24th Int'l Conf. Software Eng. (ICSE),* 2002.
[35] D. Rumelhart, G. Hinton, and R. Williams, "Learning Representations by Back-Propagating Errors," *Nature,* no. 323, pp. 533-536, 1986.
[36] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.
[37] N.E. Fenton and S. Pfleeger, *Software Metrics: A Rigorous and Practical Approach,* second ed. Int'l Thompson Press, 1995.
[38] R. Holte, "Very Simple Classification Rules Perform Well on Most Commonly Used Data sets," *Machine Learning,* vol. 11, p. 63, 1993.
[39] R. Quinlan, *C4.5: Programs for Machine Learning.* Morgan Kaufman, 1992.
[40] J. Dougherty, R. Kohavi, and M. Sahami, "Supervised and Unsupervised Discretization of Continuous Features," *Proc. Int'l Conf. Machine Learning,* pp. 194-202, 1995.
[41] D. Fisher, L. Xu, and N. Zard, "Ordering Effects in Clustering," *Proc. Ninth Int'l Conf. Machine Learning,* 1992.
[42] D. Heeger, "Signal Detection Theory," http://white.stanford.edu/~heeger/sdt/sdt.html, 1998.
[43] R. Bouckaert, "Choosing between Two Learning Algorithms Based on Calibrated Tests," *Proc. Int'l Conf. Machine Learning (ICML '03),* 2003, http://www.cs.pdx.edu/~timn/dm/10x10way.

[44] M.A. Hall, "Correlation-Based Feature Selection for Machine Learning," PhD dissertation, Dept. of Computer Science, Univ. of Waikato, Hamilton, New Zealand, 1998.
[45] K. Kira and L. Rendell, "A Practical Approach to Feature Selection," *Proc. Ninth Int'l Conf. Machine Learning*, pp. pp. 249-256, 1992.
[46] I. Kononenko, "Estimating Attributes: Analysis and Extensions of Relief," *Proc. Seventh European Conf. Machine Learning*, pp. 171-182, 1994.
[47] H. Almuallim and T. Dietterich, "Learning with Many Irrelevant Features," *Proc. Ninth Nat'l Conf. Artificial Intelligence*, pp. 547-552, 1991.
[48] T. Menzies, J.S.D. Stefano, C. Cunanan, and R.M. Chapman, "Mining Repositories to Assist in Project Planning and Resource Allocation," *Proc. Int'l Workshop Mining Software Repositories*, 2004.
[49] G. Webb, J. Boughton, and Z. Wang, "Not So Naive Bayes: Aggregating One-Dependence Estimators," *Machine Learning*, vol. 58, no. 1, pp. 5-24, 2005.

Tim Menzies received the CS and PhD degrees from the University of New South Wales and is the author of more than 160 publications. He is an associate professor at the Lane Department of Computer Science at West Virginia University (USA), and has been working with NASA on software quality issues since 1998. His recent research concerns modeling and learning with a particular focus on lightweight modeling methods. His doctoral research explored the validation of possibly inconsistent knowledge-based systems in the QMOD specification language. He is a member of the IEEE.

Jeremy Greenwald received the BS degree in physics and astronomy from the University of Pittsburgh in 2001. He is a graduate student in the Computer Science Department at Portland State University. He has more than six years of research experience in numeric methods and data mining. His master thesis focuses on the comparative study of data mining techniques and equivalences with numeric optimization techniques. He has also interned at a software development firm in Beaverton, Oregon.

Art Frank is an undergraduate student pursuing the BS degree in computer science at Portland State University. He is currently working as an IT manager and database administrator at a large nonprofit organization in Portland, Oregon.