How Simple is Software Defect Detection?

Tim Menzies (tim@menzies.us)
Kareem Ammar (kammar@csee.wvu.edu)
Lane Department of Computer Science,West Virginia University

Allen Nikora (allen.p.nikora@jpl.nasa.gov)
Jet Propulsion Laboratory, California Institute of Technology

Justin Di Stefano (justin@lostportal.net)* † ‡
Galaxy Global Corporation, Fairmont, West Virginia

Abstract. Software *defect detectors* input structural metrics of code and output a prediction of how faulty a code module might be. Previous studies have shown that such metrics many be confused by the high correlation between metrics. To resolve this, *feature subset selection* (FSS) techniques such as *principal components analysis* can be used to reduce the dimensionality of metric sets in hopes of creating smaller and more accurate detectors. This study benchmarks several FSS techniques and reports several studies where a large set metrics were reduced to a handful with little loss of detection accuracy. This result raises the possibility that software defect detection may be much simpler than previously believed.

Keywords: empirical studies and metrics; software testing and verification; principal components analysis. fault models; metrics: product metrics; defect detectors; artificial intelligence: learning; feature subset selection.

1. Introduction

Perfection is reached, not when there is no longer anything to add, but when there is no longer anything to take away. – Antoine de Saint-Exupery

Over the past several years, many sophisticated structural measurements of software systems have been used to identify fault-prone components and predict their fault content. Examples of this work include the classification

* Submitted to the Journal of Empirical Software Engineering, October, 2003
† Available from http://menzies.us/pdf/03simpled.pdf.
‡ Work sponsored by the NASA OSMA SAS Program led by the NASA IV&V Facility. The work was conducted at West Virginia University (partially supported by NASA contract NCC2-0979/NCC5-685) and at the Jet Propulsion Laboratory, California Institute of Technology (under a NASA contract). The JPL activity is managed locally at JPL through the Assurance and Technology Program Office. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology

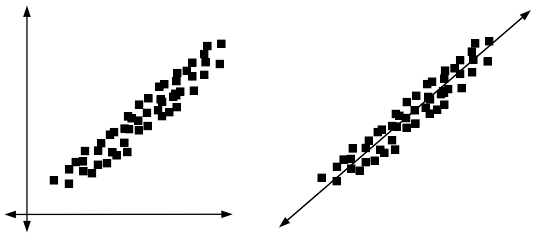

Figure 1. Transformation of axis.

methods proposed by Khoshgoftaar and Allen (Khoshgoftaar and Allen, 1999) and by Ghokale and Lyu (Gokhale and Lyu, 1997); Schneidewind's work on Boolean Discriminant Functions (Schneidewind, 1997), Khoshgoftaar's application of zero-inflated Poisson regression to predicting software fault content (Khoshgoftaar, 2001), and Schneidewind's investigation of logistic regression as a discriminant of software quality (Schneidewind, 2001).

An evident trend found within the above work is the increasing sophistication and complexity of the analysis techniques. Increasing the sophistication of our defect detection is not necessarily the best approach. This paper will argue that such increasing complexity is unnecessary. It will be shown that, at least for the data sets studied here, that very unsophisticated and very simple methods can generate good defect detectors.

Many researchers have explored methods to reduce modeling complexity. In the reliability engineering literature, principal components analysis (PCA) (Dillon and Goldstein, 1984) has been widely applied to resolve problems with structural code measurements; e.g. (Munson and Khoshgoftaar, 1990; Munson and Khoshgoftaar, 1991). PCA eliminates the problem of highly correlated measures by identifying the distinct orthogonal sources of variation and mapping the raw measurements onto a set of uncorrelated features that represent essentially the same information contained in the original measurements. For example, the data shown in two dimensions of Figure 1 (left-hand-side) could be approximated in a single transformed dimension, (right-hand-side).

PCA has its drawbacks. Fault models developed from PCA results are expressed in terms that are not directly visible to users of the model. Such models relate fault content or fault-proneness to the "domain scores" resulting from the PCA. These domain scores are weighted sums of the structural measurements standardized with respect to a chosen baseline. The structure of these models may be very simple. For example, we have used PCA and a decision tree learner to find the following defect detector:

if $domain_1 \leq 0.180$
then NoDefects
elseif $domain_1 > 0.180$
    then if $domain_1 \leq 0.371$ then NoDefects
    else if $domain_1 > 0.371$ then Defects

Here, "$domain_1$" is one of the domains found by PCA. This tree seems very simple, yet is very hard to explain to business clients users since "$domain_1$" is calculated using the following, somewhat intimidating, weighted sum:

$$\begin{aligned} domain_1 = & \ 0.241 * loc + 0.236 * v(g) \\ & +0.222 * ev(g) + 0.236 * iv(g) + 0.241 * n \\ & +0.238 * v - 0.086 * l + 0.199 * d \\ & +0.216 * i + 0.225 * e + 0.236 * b + 0.221 * t \\ & +0.241 * lOCode + 0.179 * lOComment \\ & +0.221 * lOBlank + 0.158 * lOCodeAndComment \\ & +0.163 * uniq_Op + 0.234 * uniq_Opnd \\ & +0.241 * total_Op + 0.241 * total_Opnd \\ & +0.236 * branchCount \end{aligned}$$

(Here, $v(g), ev(g), iv(g)$ are the standard McCabe structural metrics (McCabe, 1976) while the rest are either Halstead metrics (Halstead, 1977) or simple variants on lines of code count. The appendix of this article contains a brief tutorial on these metrics.)

This problem with explaining domain scores encouraged us to look for alternatives to PCA. Our reading of the data mining literature suggested that PCA belongs to a class of *feature subset selection* (FSS) techniques which aim to remove superfluous features (Hall and Holmes, 2003; Kohavi and John, 1997b; Gunnalan et al., 2003). The goal of FSS is to drastically reduce the dimensionality of the data, thus simplifying any subsequent processing. The dimensionality reduction of FSS means that any subsequent processing can ignore irrelevant, redundant and noisy features and focus on only relevant, highly predictive ones to improve its performance. Lastly, detectors learnt from reduced dimensionality are more compact, easily understandable representations of the underlying concept.

To the best of our knowledge, it has not been previously noted in the reliability literature that PCA is one member of a large set of FSS techniques. This study benchmarks PCA against those FSS techniques, in terms of accuracy of the learnt defect detectors. We will show that in the special case of generating defect detectors, very simple FSS methods can out-perform PCA both in

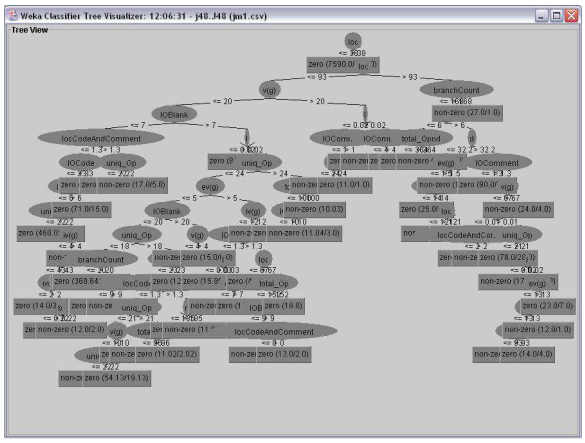

Figure 2. A large decision tree produced by the C4.5 decision learner (Quinlan, 1992) using all 22 metrics in the JM1 data set analyzed in this article.

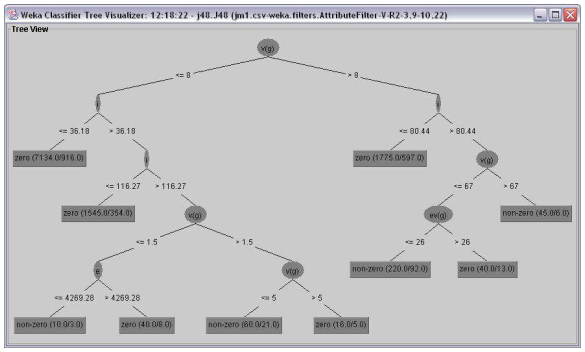

Figure 3. Small decision tree produced by C4.5 from the JM1 data set using just the three metrics selected by the SELECT FSS method (described later in this paper).

terms of the number of features rejected and the accuracy of the detectors learnt from the remaining features.

Unlike other studies (e.g. (Munson and Nikora, 1998)), which contained a mere fifty observations, the experimental data used for this paper is large (hundreds to thousands of records) and is drawn from five different software projects shown in Figure 4. Apart from being written in “C” or “C++”, there is little commonality in our sample. These projects come from five different teams working at four different locations around the country. This software performs a wide range of different functions from spacecaft instrumentation to real-time predictive simulations. That is our conclusions are based on a broader experience base than previous work: specifically, 15,730 modules for which there exist 3975 defect reports.

project	# modules	% with defects	language	developed at	notes notes
AN1	1719	58%	C++	location 1	development project for reusable code libraries
CM1	496	9.7%	C	location 2	a NASA spacecraft instrument
JM1	10885	19%	C	location 3	real-time predictive ground system (uses simulations to generate the predictions)
KC1	2107	15.4%	C++	location 4	storage management for receiving and processing ground data
KC2	523	20%	C++	location 4	science data processing; another part of the same project as KC1; different personnel to KC1. shared some third-party software libraries as KC1, but no other software overlap.

Figure 4. Data sets used in this study. Here, a "module" is the equivalent of a C function or a C++ method. All modules analyzed were built by NASA developers excluding several thousands modules that are COTS software (COTS is an acronym for Commercial Off The Shelf). The McCabe and Halstead structural metrics were extracted from these systems and mapped to the defect logs kept for each project. Note that AN1 is an artificially generated data set. That system has not finished its testing phase so its defect logs are incomplete. For this data set, we hence used all the log entries relating to defects and a nearly equal number of entries with no defects (selected at random).

Another important feature of this study is that it is a *repeatable* experiment. Four of the five data sets used here publicly available[1]. These experiments also use freely distributed tools available online, such as the WEKA machine learning toolkit [2] and the TAR2 treatment learner (Menzies and Hu, 2001; Menzies and Hu, 2002a; Menzies and Hu, 2002b)[3]. Repeatability is an important methodologically principle since it allows other researchers to independently assess our results.

The most important feature of our study was the dramatic reduction in number of features. In all the case studies shown below, over 75% of the

[1] `http://mdp.ivv.nasa.gov`, or `http://menzies.us/data.html`
[2] `http://www.cs.waikato.ac.nz/~ml/weka/`
[3] `http://menzies.us/rx.html`

available features could be ignored, without compromising the detector accuracy. For example our case studies show that the complex defect detector decision tree of Figure 2 can be reduce to simpler tree of Figure 3, with little or no loss in defect detection accuracy. Interestingly, these reductions are obtained using methods much simpler than anything used before in the software reliability literature. This result has made us reevaluate our own previous results (Munson and Nikora, 1998; Menzies and DiStefeno, 2002) that used PCA and other techniques to simplify fault detectors.

This is not to say that the prior research on PCA was useless. On the contrary, claims that method *X* is simpler, but just as effective, as method *Y* is meaningless without knowledge of method *Y*. The only way this paper can claim that something is a better FSS than (e.g.) PCA is to have access to the prior results on PCA. Hence, we say that prior research on PCA was an essential precursor to this work.

2. Related Results

The thesis of this paper is that *many features are ignorable*. That is, most of the available metrics can be omitted from defect detectors without affecting the accuracy of those detectors. There is some evidence for this thesis, scattered throughout the literature. This section reviews that evidence.

A defect detector in this domain is a test that some measured software structural feature has passed some threshold. Different metric ranges may also be combined to form a composite defect detector in order to compose trees or other classifier structures.

Decision tree learning has been frequently applied to the task of generating summaries of defect logs. Often, these summaries use only a small subset of the available features. For example, Figure 5 shows one study where, of the 42 features offered in the data set, only six were deemed significant by the learner.

For another example, Figure 7 shows 18 metrics given to a particular learner. Figure 6 shows what that learner generated. The key feature of Figure 6 is what is *not shown* in the learnt decision tree: of the 18 features available to this learner, only the four underlined metrics appear in the tree.

For yet another example, we can look at the individual domains learnt by PCAs for a mission software technology development effort at JPL (Dvorak et al., 1999). Figure 8 shows that, with respect to the index of cumulative faults, not all features are equally associated with faults. Figure 8 plots the cumulative domain values for each of the system builds, together with the cumulative number of faults for each build. It is quite apparent from this figure that Domain 1, associated with control, is most closely associated with the cumulative fault count. Indeed, the correlation coefficient between Domain 1

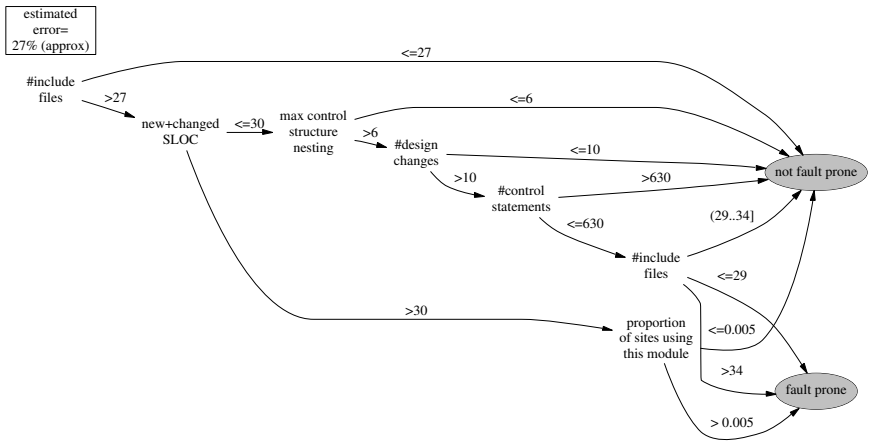

Figure 5. Predicting fault-prone modules (Khoshgoftaar and Allen, 1999). Learned from data collected from a telecommunications system with $>$ 10 million lines of code containing a few thousand modules.

normalized cumulative domain values and the cumulative fault values is 0.94. The correlation between Domain 2 and cumulative faults is -0.20. Finally, correlation between Domain 3 and cumulative faults is 0.71.

The above examples can only be found after reading widely in the literature. The rest of this article checks if the phenomenon that *many features are ignorable* is easily repeatable. A range of data sets will be explored using a range of *feature subset selection* (FSS) techniques. With the exception of PCA, most of these FSS techniques come from the data mining literature. Hence, before we explain FSS, we must offer some background notes on data mining.

3. Data Mining

Data mining is a summarization technique that reduces large sets of examples to a small understandable pattern using a range of techniques taken from statistics and artificial intelligence. It is commonly referred to as searching for pearls in the sand. This next section is a review of data mining methods and algorithms used in this study. The subsequent section describes feature subset selection.

3.1. Methods and Algorithms

Cross Validation: A common mistake that new data miners make is *over-training*. Over-training happens when a data miner to get *too* specific in its

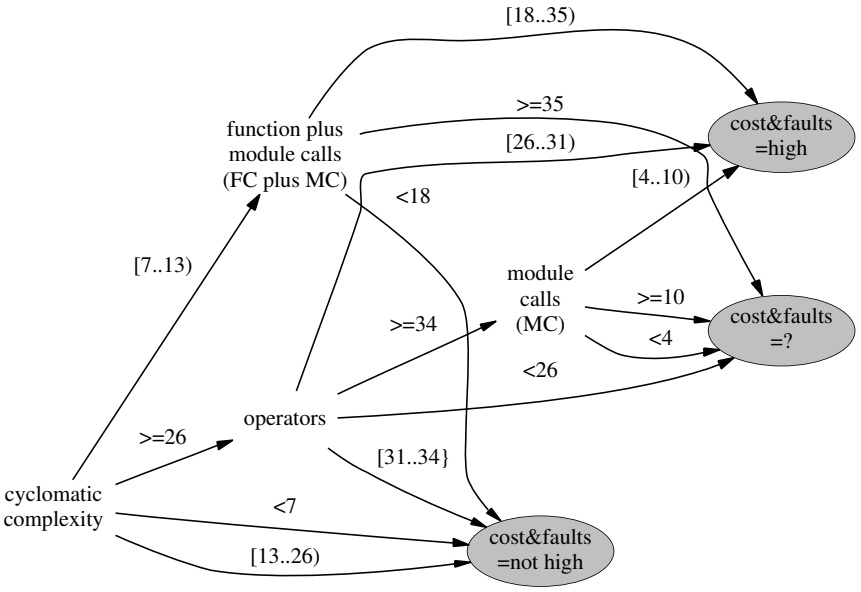

Figure 6. Predicting modules with high cost modules and many faults. Data from 16 NASA ground support software for unmanned spacecraft control (Tian and Zelkowitz, 1995). These systems were of size 3,000 to 112,000 lines of FORTRAN and contained 4,700 modules.

learning. If that happens then your results, while extremely applicable to current data, are unlikely to apply to data seen in the future.

Across whole module:	*Averages per KSLOC:*
total operators	FC plus MC
total operators	IO statements
	IO parameters
Averages per KSLOC:	origin
assignment statements	operands
cyclomatic complexity	operators
executable statements	comments (C)
decision statements	source lines (SL)
function calls (FC)	SL minus C
module calls (MC)	format statements

Figure 7. Metrics available to the learner that generated Figure 6. “Cyclomatic complexity” is a measure of internal program intricacy (McCabe, 1976).

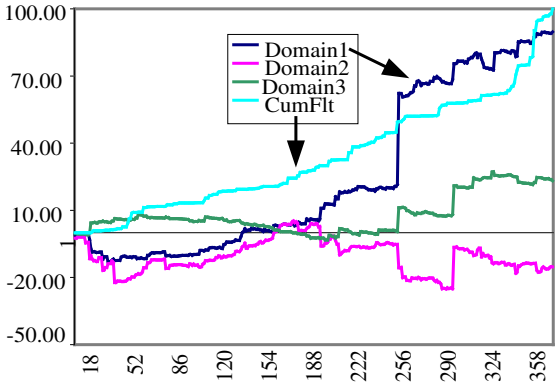

Figure 8. Three domain scores and a cumulative total for one JPL system.

One way of avoiding this pitfall is by assessing the learnt treatments against data not used during training. One method for doing so is N-way cross validation. In this process, a training set is divided into N buckets. For each bucket in turn, a select is learned on the other $N - 1$ buckets, then tested on the bucket that was put aside. A learner is deemed *stable* if it works in the majority of all N turns.

Decision Tree Learning: Figure 2, Figure 3, Figure 5 and Figure 6 were generated via decision tree learners. One way to learn such trees is to *split* the whole example set into subsets based on some metric/threshold comparison. The process then repeats recursively on the subsets. Each splitter value becomes the root of a sub-tree. Splitting stops when either a subset gets so small that further splitting is superfluous, or a subset is contains examples with only one type (e.g. all the remaining examples are about defective modules).

A *good split* decreases the percentage of different types of modules in a subset. Such a good split ensures that smaller subtrees will be generated since less further splitting is required to sort out the subsets. Various schemes have been described in the literature for finding good splits. For example, the C4.5 (Quinlan, 1992) and J4.8 (Witten and Frank, 1999) decision tree algorithms uses an information theoretic measure (entropy) to find its splits while the CART (Breiman et al., 1984) decision tree learner uses another measure called the GINA index.

Bayesian Learning: An alternative to decision tree learning is Naive Bayesian learning (Witten and Frank, 1999). In this approach, a prior probability of an hypothesis H is updated whenever new evidence E comes to hand. Baye's rule tells us how:

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

Such learners are "naive" in that they assume no correlation between attributes. However, this seemingly "naive" assumption has proven to be remarkably robust and useful in many domains.

For example of Bayesian learning, consider the log of golf-playing behavior shown in Figure 9. In that log, the frequency of playing some, or lots of golf is $P(none) = \frac{5}{14}$, $P(some) = \frac{3}{14}$ and $P(lots) = \frac{6}{14}$ respectively. In the special case where it is not windy (i.e. $E = notwindy$) then the probabilities change to $P(not\ windy|none)$= $\frac{2}{8}$, $P(not\ windy|some)$= $\frac{3}{8}$, $P(not\ windy|lots)$= $\frac{3}{8}$. If we have evidence that today is not windy, we can update our prior beliefs about golf-playing behavior. First, we compute the likelihoods that we will play none,some, or lots of golf:

$$likelihood(none|not\ windy) = \frac{2}{5} * \frac{5}{14} = 0.143$$
$$likelihood(some|not\ windy) = \frac{3}{3} * \frac{3}{14} = 0.214$$
$$likelihood(lots|not\ windy) = \frac{3}{6} * \frac{6}{14} = 0.214$$

These likelihoods are then normalized in the standard way to get probabilities:

$$P(none|not\ windy) = \frac{0.143}{0.143 + 0.214 + 0.214} = 0.250$$
$$P(some|not\ windy) = \frac{0.214}{0.143 + 0.214 + 0.214} = 0.375$$
$$P(lots|not\ windy) = \frac{0.214}{0.143 + 0.214 + 0.214} = 0.375$$

That is, on non-windy days, it is least probable that we will play no golf.

Treatment Learning: A new data mining technique is the TAR2 treatment learning technique developed by Menzies and Yu (Hu, 2002; Menzies and Hu, 2002b; Menzies et al., 2002b; Menzies et al., 2002a; Menzies and Hu, 2002b; Menzies and Hu, 2001; Menzies and Hu, 2002a). Treatment learning searches for a strong *select statement* that most *changes* the ratio of classes. To understand the concept of a *strong select statement*, consider the log of golf playing behavior seen in Figure 9. In that log, we only play *lots* of golf in $\frac{6}{5+3+6} = 43\%$ of the cases. To improve our game, we might search for conditions that increases our golfing frequency. Two such searches are shown in the bottom of Figure 9. In the case of `outlook=overcast`, we play *lots* of golf all the time. In the case of `humidity` $\geq$ `90`, we only play *lots* of golf in 20% of the cases. The net effect of these two select statements is shown in Figure 10.

The `WHERE` statements within a select statement can contain conjunctions of arbitrary size. Exploring all such conjunctions manually is a tedious task.

outlook	*temp(°F)*	*humidity*	*windy?*	*class*
sunny	*85*	*86*	*false*	*none*
sunny	*80*	*90*	*true*	*none*
sunny	*72*	*95*	*false*	*none*
rain	*65*	*70*	*true*	*none*
rain	*71*	*96*	*true*	*none*
rain	*70*	*96*	*false*	*some*
rain	*68*	*80*	*false*	*some*
rain	*75*	*80*	*false*	*some*
sunny	*69*	*70*	*false*	*lots*
sunny	*75*	*70*	*true*	*lots*
overcast	*83*	*88*	*false*	*lots*
overcast	*64*	*65*	*true*	*lots*
overcast	*72*	*90*	*true*	*lots*
overcast	*81*	*75*	*false*	*lots*

```
SELECT class FROM original  SELECT class FROM original
WHERE outlook = 'overcast'  WHERE humidity >= 90

lots                        none
lots                        none
lots                        none
lots                        some
                            lots
```

Figure 9. Attributes that select for golf playing behavior.

TAR2 is an automatic tool for finding the strongest select statements; i.e., the statement that *most* selects for preferred behavior while *most* discouraging

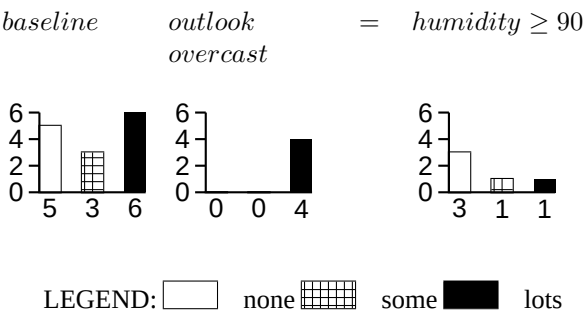

Figure 10. Changes to golf playing behavior from the baseline.

undesirable behavior. TAR2 calls this strongest select statement the "treatment" since it is a recommended action for improving the current situation. The algorithm is automatic and, as used in this study, searched the entire range of possible conditions. TAR2's configuration file lets an analyst search for the best select statement using conjunctions of size 1,2,3,4, etc. Since TAR2's search is elaborate, an analyst can automatically find the *best* and *worst* possible situation within a data set. For example, the select statements seen in Figure 10 were learnt by TAR2 and show the *best* and *worst* possible situation for playing *lots* of golf.

1R: Simpler than any of the above techniques is the 1R machine learner (Holte, 1993). It creates a set of rules from a single attribute. First 1R selects an attribute then branches within the attribute to create a set of divisions based on class value. For each division it assigns the most frequent class and then computes the error rate. Finally, 1R simply chooses the attribute with the total least error rate.

ROCKY: Simpler even than 1R is ROCKY (Menzies et al., 2003). Given a set of numeric metrics

$$attribute_1, attribute_2, ...attribute_n$$

ROCKY exhaustively explores all singleton rules of the form

$$attribute \geq threshold$$

Threshold is found as follows. Every numeric attribute is assumed to come from a gaussian distribution. $Threshold$s are then selected corresponding to equal areas under that distribution. For example, in one of the data sets we examine, the McCabe cyclomatic complexity $v(g)$ had a mean of $\mu = 4.9$ and a standard deviation of $\sigma = 11$. If this Gaussian is converted to a unit Gaussian (by subtracting the mean and dividing by the standard deviation), then standard Z-tables could be used to calculate a $v(g)\ threshold$ value of 7.65 could be found as follows:

$$\begin{aligned} area &= 0.6 \text{ (just for example)} \\ Z^{-1}(area) &= \frac{v(g) - \mu}{\sigma} \\ Z^{-1}(area) &\approx 0.25 \\ \therefore v(g).threshold(area) &\approx 7.65 \end{aligned}$$

ROCKY generates one detector

$$attribute_i \geq attribute_i[threshold(area)]$$

for the range

$$area \in \{0.05, 0.1, 0.15, \ldots 0.9, 0.95\}$$

A key point that will be important below is that ROCKY and 1R can only ever find detectors based on a single attribute.

3.2. Feature Subset Selection

Feature subset selection finds what subset of the available features is most informative. PCA is the FSS method best known to the reliability engineering community. However, as we shall see, numerous other FSS methods have been evolved in the data mining community.

A repeated empirical observation is that ignoring features can improve classifier accuracy. How can ignoring information be useful? Kohavi & John (Kohavi and John, 1997b) review studies with Naive Bayes classifiers. The accuracy of such classifiers decreases very slowly as irrelevant features are added to an instance set. However, the accuracy of the same classifiers can degrade sharply as the number of correlated features increase. Note that this observation is similar to the original motivations for using PCA: i.e. learning is simpler when highly correlated features don't conflate the learning process.

Another explanation for the success of FSS is offered by Witten & Frank (Witten and Frank, 1999). They note that effective generalization requires numerous examples. Decision tree learners recursively split instances by ranking features according to how much they decrease the diversity of the classes in the split sets. As learning progresses, fewer and fewer instances are available to learn the next sub-tree. If the instances contain too many features of similar rank, then many splits are quickly generated. Hence, instances become sparser in the sub-trees, and effective generalization becomes harder.

Yet another explanation for the success of FSS comes from Gunnalan, Menzies, *et.al.* (Gunnalan et al., 2003) who argue that solvable problems have an average case property called *small backbones*. Small backbone problems contain a small number of variables that control all other variables in the system. Learning the essential features of small backbone problems means finding the variables that are either in the small backbone or highly correlated to the backbone variables.

PCA: Principal Component Analysis: PCA first began to be used in modeling software reliability and fault content in the late 1980s and early 1990s, when Munson and Khoshgoftaar first developed the concept of relative complexity (Munson and Khoshgoftaar, 1990; Munson and Khoshgoftaar, 1991), which is described as a weighted sum of the domain scores resulting from the application of PCA to raw structural measurements. Unlike other complexity metrics, relative complexity simultaneously combines all feature dimensions of all structural measures. In an early paper, they identified clear relationships between complexity metric domains and software quality (Munson and Khoshgoftaar, 1990). In a later paper, they examined re-

lationships between the relative complexity and software reliability (Munson and Khoshgoftaar, 1991). This study concluded that:

– The relative complexity measure is appropriate for the comparison and classification of software modules, and

– It is feasible to include relative complexity as a parameter in software reliability models.

In particular, they noted that relative complexity could be used to represent the complexity of a particular software module for a particular build, which laid the foundation for measuring the evolution of software system.

In 1996, Munson and Werries presented a methodology for measuring software evolution that extended the notion of software complexity across sequential builds (Munson and Werries, 1996). In this paper, they addressed the issue of establishing a baseline against which all change to a software system will be measured. To properly account for the amount of change that occurs between subsequent builds of a system, it is necessary to measure each build with respect to a baseline that remains constant across all builds. This is accomplished by choosing one particular build as the baseline, and then standardizing the measurements from all other builds with respect to the means and standard deviations of the baseline measurements. They also developed a mechanism wherein the precise manner in which builds differ from each other may be measured. This is accomplished by computing the difference in relative complexity between subsequent versions of a module within the system. The measurement mechanism also takes into account the situation in which a module is present in one of the builds but not the other.

Recent investigations have focused on identifying relationships between the measured structural evolution of a software system and the rate at which faults are inserted into it during development (i.e., the number of faults inserted per unit of structural change). In a small study (Munson and Nikora, 1998), Nikora and Munson analyzed the flight software and software failure reports for the command and data handling subsystem of a NASA planetary exploration spacecraft, and found strong indications that measurements of a system's structural evolution could serve as predictors of the fault insertion rate. However, this study had two limitations: The study was relatively small - fewer than 50 observations were used in the regression analysis relating the number of faults inserted to the amount of structural change. The definition of faults that was used was not quantitative. The ad-hoc taxonomy, first described in (Nikora and Munson, 1997), was an attempt to provide an unambiguous set of rules for identifying and counting faults. The rules were based on the types of changes made to source code in response to failures reported in the system. Although the rules provided a way of classifying the faults by type, and attempted to address faults at the level of individual

	number of attributes										
before:	10	13	15	180	22	8	25	36	6	6	6
after:	2	2	2	11	2	1	3	12	1	1	2
reduction:	80%	84%	87%	94%	90%	87%	88%	67%	83%	83%	67%
Δaccuracy:	0%	6%	5%	4%	2%	1%	0.5%	0%	-25%	6%	7%

Figure 11. Feature subset selection using a WRAPPER of a decision tree leaner. The ΔAccuracy figure is the difference in the accuracies of the theories found by decision tree learner using the $before$ and $after$ attributes. From (Kohavi and John, 1997b).

modules, they were not sufficient to enable repeatable and consistent fault counts by different observers to be made. The rules in and of themselves were unreliable. To overcome these limitations, the investigators developed a quantitative definition of software faults, based on the grammar of the language of the software system (Munson and Nikora, 2002). They also initiated a collaboration with the Mission Data System, a mission software technology development effort at the Jet Propulsion Laboratory (Dvorak et al., 1999). They were able to collect significantly more information than for the previous study; over the time interval during which they study was conducted, there were over 1500 builds of the MDS. The total number of distinct versions of all modules was greater than 65,000, and over 1400 problem reports were included in the analysis. This study agreed with the earlier study's conclusions that there appear to be strong relationships between measurements of a software system's structural evolution and the number of faults inserted into that system, and extended the earlier work by identifying types of structural change more likely to result in the introduction of faults and types less likely to do so.

WRP: Wrapper Subset Evaluation: PCA is a common FSS method used by statisticians. WRAPPER is a common FSS method used by data miners. In this method, a *target learner* is augmented with a pre-processor that used a heuristic search to grow subsets of the available features. At each step in the growth, the target learner is called to find the accuracy of the model learned from the current subset. Subset growth is stopped when the addition of new features did not improve the accuracy.

Figure 11 shows some WRAPPER results from experiments by Kohavi and John (Kohavi and John, 1997a). In their experiments, 83% (on average) of the measures in a domain could be ignored with only a minimal loss of accuracy.

The advantage of the this approach is that, if some target learner is already implemented, then the WRAPPER is simple to implement. The disadvantage

of the wrapper method is that each step in the heuristic search requires another call to the target learner; i.e. it may be very slow.

For the results shown below, we will use a WRAPPER of two target learners: a decision tree learner (C4.5) and a Naive Bayes classifier.

IG: Information Gain Attribute Ranking: This is a simple and fast method for feature ranking (Dumais et al., 1998). This method measures the split criteria of the class before and after observing a feature. The differences in the split criteria gives a measure of the information gained because of that attribute (Quinlan, 1992). A final comparison of this measure is used in feature selection.

RLF: Relief: Relief is an instance based learning scheme (Kira and Rendell, 1992; Kononenko, 1994). It works by randomly sampling one instance within the data. It then locates the nearest neighbors for that instance from not only the same class but the opposite class as well. The values of the nearest neighbor features are then compared to that of the sampled instance and the feature scores are maintained and updated based on this. This process is specified for some user-specified M number of instances. Relief can handle noisy data and other data anomalies by averaging the values for K nearest neighbors of the same and opposite class for each instance (Kononenko, 1994). For data sets with multiple classes, the nearest neighbors for each class that is different from the current sampled instance are selected and the contributions are determined by using the class probabilities of the class in the dataset.

CFS:Correlation-based Feature Selection: CFS uses subsets of features (Hall., 1998). This technique relies on a heuristic merit calculation that assigns high scores to subsets with features that are highly correlated with the class and poorly correlated with each other. Merit can find the redundant features since they will be highly correlated with the other features. It can also identify ignorable features since they will be poor predictors of any class. To do this CFS informs a heuristic search for key features via a correlation matrix.

CBS: Consistency-based Subset Evaluation: CBS is really a set of methods that use class consistency as an evaluation metric. The specific CBS studied by Hall and Holmes method finds the subset of features whose values divide the data into subsets with high class consistency (Almuallim and Dietterich, 1991).

SELECT: Figure 12 shows the SELECT FSS developed by Gunnalan, Menzies, *et.al.* (Gunnalan et al., 2003). SELECT runs TAR2 many times, each time targeting a different class; e.g. defects, no defects:

1. Initialize the SELECTED features to nil.
2. For each class in turn, declare it to be TAR2's "best" class. Then enter the following loop:
   - Set treatment size N to 1

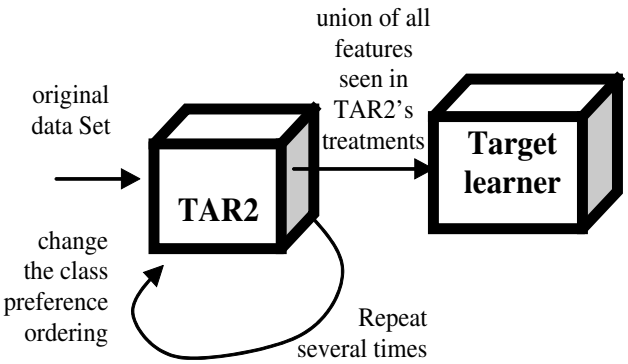

Figure 12. SELECT algorithm.

- Find the "best" treatment of size N via TAR2.
- If the score of the best treatment is no better than that of the best treatment of size N-1, then
  - Add the features seen in the best treatment to SELECTED.
- Else, N++ and loop.

3. Collect the average accuracy seen in a 10-way cross validation of the target learner using
   - just the features seen in SELECTED
   - all features

Note that each single run of TAR2 finds features that most selected for one class. Over all the runs, TAR2less finds the union of all the features that most selected for every class.

1R and ROCKY: Most FSS methods input M features and output some subset $N, N < M$. An extreme form of feature subset selection is to use learners that can only output theories containing $N = 1$ features. Two such learners are the 1R and ROCKY systems described above.

Note that this method is far less general than the other methods described above since it will fail if $N > 1$ features must be selected.

4. Experiments

The remainder of this paper is dedicated to a case study on the AN1, CM1, KC1, KC2, JM1 datasets described in Figure 4.

Learner	Attributes	CM1	KC1	KC2	JM1	AN1
Original (C4.5)	21	89.52%	83.77%	82.11%	79.34%	65.90%
Original (Bayes)	21	84.88%	82.34%	83.65%	80.41%	58.27%
1R	1	89.30%	83.10%	82.95%	79.55%	65.02%
ROCKY	1	90.50%	85.63	*85.28%*	*81.10%*	*66.14%*

Figure 13. 1R and ROCKY runs. Baseline accuracies generated from all features via C4.5 and Naive Bayes shown on lines one and two. Underlined entries denotes an increase over all baselines.

Methods: FSS was conducted using the PCA, CBS, IG, RLF, WRAPPER, and 1R implementations supplied with the WEKA machine learning toolkit[4]. We used our own implementations of ROCKY and TAR2[5]. SELECT was applied manually using TAR2.

Each FSS method *generated* candidate features which were then *selected* and *assessed*. Usually, the selected features were *assessed* by running them through a 10-way cross validation over the C4.5 and Naive Bayes classifiers supplied within the WEKA. Assessing FSS via these two learners is quite standard in the FSS literature (e.g. (Kohavi and John, 1997b)) since these are widely used and understood learning systems. Also, these two classifiers are very different kinds of learners so results that repeat in both C4.5 and Naive Bayes are guaranteed not to be the result of quirks in (e.g.) decision tree learning.

Sometimes, however, other methods were required to assess the selected features. For example, if we were "wrapping" learner "X" then we assessed the WRAPPER's output only on learner "X". Also, in the case of ROCKY and 1R, those learners have their own cross-val facilities to assess the accuracies of their learnt theories.

The *generation* methods varied. In the usual case, the WEKA environment offered options to conduct FSS via a 10-way cross validation. We disabled this option for WRAPPER since that was impractically slow, especially for the 10,000 records in JM1. 10-way cross validation was also used within SELECT when finding the best treatment of size N.

Results: Figure 13 shows the classification on 10-way generated by ROCKY and 1R. Figures 14 to 18 show the average classification accuracies seen in 10-way cross validation runs of Naive Bayes and WRAPPER using just the features found by our FSS methods.

The first line of Figure 14, Figure 15, Figure 16, Figure 17 and Figure 18 shows the results of running all available features through Naive Bayes and C4.5. Underlined entries mark the largest accuracy generated by any method.

[4] `http://www.cs.waikato.ac.nz/~ml/weka/`

[5] Available from `http://menzies.us/rx.html` and `http://menzies.us/pace.html`

		C4.5		Naive Bayes
	Attributes	accuracy	tree Size	accuracy
Original	15	65.90%	131	58.27%
SELECT	4	*66.82%*	119	*58.61%*
CFS	5	65.02%	17	*60.42%*
CBS	9	*66.07%*	81	57.57%
IG	4	64.55%	3	*60.65%*
RLF	4	*67.11%*	41	*60.77%*
PCA	7	65.77%	13	*61.87%*
Wrapper	5 (c4.5)	*66.88%*	37	
	2 (bayes)			*62.51%*
	mean	*66.02%*	mean	*60.08%*

Figure 14. AN1 FSS and learner accuracy runs. Baseline accuracies shown on line one. Underlined entries mark the largest accuracy generated by any method. *Italicized entries mark an increase over the baseline. WRAPPER's selected features were only assessed on the "wrapped" learner- either C4.5 or Naive Bayes (hence the blank cells on the WRAPPER line).*

		C4.5		Naive Bayes
	Attributes	accuracy	tree Size	accuracy
Original	24	89.52%	33	84.88%
SELECT	6	*90.32%*	1	*86.49%*
CFS	6	89.2%	1	*86.90%*
CBS	1	*89.92%*	1	*89.11%*
IG	4	*90.12%*	1	*87.90%*
RLF	4	89.52%	1	83.48%
PCA	7	89.52%	13	*86.29%*
Wrapper	0 (C4.5)	N/A	1	
	0 (bayes)			N/A
	mean	*89.84%*	mean	*86.43%*

Figure 15. CM1 FSS and learner accuracy runs. Baseline accuracies shown on line one. *Italicized* and blank entries have the same meaning as in Figure 14. N/A denotes runs were the feature subset selector returned no attributes

Italicized entries show where features found by FSS generated detectors with a higher accuracy than the baseline. Figure 19 is a summary table showing how often our FSS methods out-performed all baselines.

Of all our results, CM1 is most unusual. Figure 15 shows that the accuracy after FSS was nearly the same as before FSS. Further, in that dataset some FSS methods (WRAPPER) selected no feature at all; i.e. WRAPPER found that no feature was more informative than any other. This result can be partially explained by the nature of that data. Of all our datasets, CM1 is

		C4.5		Naive Bayes
	Attributes	accuracy	tree Size	accuracy
Original	21	79.34%	677	80.41%
SELECT	4	*81.06%*	11	*80.70%*
CFS	7	*80.51%*	63	*80.46%*
CBS	19	*79.48%*	671	80.25%
IG	4	*81.25%*	15	*80.43%*
RLF	4	*80.98%*	25	79.62%
PCA	8	*80.09%*	17	79.67%
Wrapper	5 (C4.5)	*85.57%*	15	
	2 (bayes)			*80.90%*
	mean	*81.04%*	mean	80.31%

Figure 16. JM1 FSS and learner accuracy runs. Baseline accuracies shown on line one. *Italicized* and blank entries have the same meaning as in Figure 14.

		C4.5		Naive Bayes
	Attributes	accuracy	tree Size	accuracy
Original	21	83.79%	163	82.34%
SELECT	3	*85.00%*	5	*83.82%*
CFS	7	*84.24%*	99	*82.91%*
CBS	18	83.29%	135	82.34%
IG	4	*85.38%*	5	*84.20%*
RLF	4	*85.19%*	9	80.83%
PCA	6	82.05%	21	*83.00%*
Wrapper	5 (C4.5)	*85.29%*	13	
	1 (bayes)			*85.52%*
	mean	*84.65%*	mean	*83.12%*

Figure 17. KC1 FSS and learner accuracy runs. Baseline accuracies shown on line one. *Italicized* and blank entries have the same meaning as in Figure 14.

the smallest and has the lowest defect rate (see Figure 4). Perhaps the target concept in CM1 is too small to be found by the methods discussed here.

5. Questions and Answers

Our results let us comment on the the following issues.

Q: Is throwing away information useful for defect detection?

A: Surprisingly, it would seem so. Figure 19 shows that in $57/80 = 71\%$ of our experiments, using any FSS method improved the accuracy over the baseline. Also, in all our experiments, if all the FSS methods described here are used, then detectors were found with a higher accuracy than the baseline,

		C4.5		Naive Bayes
	Attributes	accuracy	tree Size	accuracy
Original	21	82.11%	51	83.65%
SELECT	2	*82.89%*	5	83.08%
CFS	2	*85.25%*	9	*84.10%*
CBS	6	*83.33%*	23	*83.72%*
IG	4	*84.29%*	3	*84.10%*
RLF	4	81.61%	5	82.18%
PCA	2	*82.57%*	5	*84.87%*
Wrapper	1 (C4.5)	*85.57%*	3	
	4 (bayes)			*85.19%*
	mean	*83.45%*	mean	*83.86%*

Figure 18. KC2 FSS and learner accuracy runs. Baseline accuracies shown on line one. *Italicized* and blank entries have the same meaning as in Figure 14.

	C4.5	Naive Bayes	1R	ROCKY	total
JM1	7/7	4/7	0/1	1/1	12/16
AN1	4/7	6/7	0/1	1/1	11/16
KC2	6/7	5/7	0/1	1/1	12/16
KC1	6/7	5/7	0/1	1/1	12/16
CM1	4/7	5/7	0/1	1/1	10/16
total	27/35	25/35	0/5	5/5	57/80

Figure 19. How often FSS generates theories of higher accuracy than using all available features.

while using far fewer features. If we look at the best detector (the underlined entries), then we see that the best detectors used between 1 to 6 features selected from a space of 15 to 21 features. Hence, this study endorses FSS for defect detector generation.

Q: What is a good FSS method for defect detection?

A: This is unclear but the very simple FSS methods (ROCKY and 1R) ran very fast and resulted in highly accurate theories. Further, ROCKY outperformed the other FSS methods in four of our five case studies. The CPU-intensive WRAPPER method is slow to run but, of the more complex FSS methods, always generated the most accurate theory (using either C4.5 or Naive Bayes).

Q:Comparatively speaking, how does PCA compare to other FSS methods?

A: In this study, PCA has not scored well. In none of our experiments were the highest accuracy detectors learnt via FSS. Hall & Holmes have assessed a similar set of FSS methods as this study, but on a broader set of data (none of

which related to software defect detection). Their results are hardly supportive of PCA. They conclude that if the slow run times of WRAPPER can be tolerated, then it usually generates the most accurate theories. Otherwise, in their opinion, CFS and RLF are best (Hall and Holmes, 2003). Our results do not contradict their conclusions- we obtained high accuracies with CFS, RLF and WRAPPER, and never with CBS, IG, or PCA.

Q: How simple is defect detection?

A: Apparently, very simple. The "clean-up" offered by FSS was very small. In all cases the accuracy found using all features was less than 7% of the best accuracy found after FSS. This suggests that the correlations between variables in these data sets do not greatly confound defect detector generation. Note that if most defect data sets lack such correlations then tools designed to handle highly-correlated datasets (e.g. PCA) are not necessary.

Q: Is accuracy the best way to judge the effectiveness of a defect detector?

A: Perhaps not. A striking feature of Figures 14 to 18 is the very small variance in the accuracy figures. In other work (Menzies et al., 2003), we have assessed hundreds of learnt theories and found that accuracy can remain stable while other important features can vary wildly. For example, two detectors with the same accuracy can have very different probabilities of false alarms. Other data mining research suggests that accuracy alone is not a good indicator of learner performance in many domains (Provost et al., 1998). This may be attributed to greatly skewed class distributions or domain related anomalies. We are currently repeating our study, but this time assessing detectors via:

- The *cost* of collecting the data for the detectors (collecting cyclomatic complexity using Mccabes can be very expensive due to licensing issues);
- The probability of false alarms, given that the detector has been triggered;
- The probability of true detection alarms, given that the detector has been triggered;
- The probability that a defect has been missed, given that the detector has not been triggered;
- The stability of the detector under N-way cross validation;
- The stability of the detector when applied to different data sets;

Our current thinking is that finding a "best" detector judged on all the above criteria will require some kind of N-dimensional optimization toolkit.

Q: What is the best detector?

dataset	detector	accuracy
AN1	$unique\ operands >= 8.14$	66.08%
CM1	$I \geq 167.94$	90.54%
JM1	$unique\ operands >= 60.48$	81.10%
KC1	$V \geq 1106.55$	85.62%
KC2	$ev(g) >= 4.99$	85.28%

Figure 20. Best detectors learnt by ROCKY.

A: We did not find that McCabe's standard detector of $v(g) > 10$ was the most accurate. However, we cannot offer an external valid alternative. The defect detectors did not stabilize across the different data sets. For example, the most accurate defect detectors found by ROCKY are shown in Figure 20 (we report ROCKY's output here since that output is small enough to read and ROCKY's best accuracies were always very close to the best overall accuracies). Note that variations in the learnt detectors:

- The McCabe $ev(g)$ value was the most accurate in KC2
- The Halstead values of *intelligence content* (I), the *volume* of the program (V) or the $unique\ operands$ value were the feature that yielded the most accurate detectors in the other data sets (for an explanation on these metrics, see the appendix).
- The threshold value for the two detectors that use the same Halstead metric were wildly different: 60.48 in JM1 and 8.14 in AN1.

Clearly, the distributions of variables seen in JM1 and AN1 are very different. If distributions always vary so wildly between defect data sets, then we it may be folly to imagine that a single defect detector rule such as $v(g) > 10$ will suit all software development. Instead, companies should tune their defect detectors according to their own historical logs describing their own people building their own kind of application.

6. Discussions

We offer our results with two cautions. Firstly, we have only explored feature subset selection for defect detectors using five data sets. To the best of our knowledge, our sample is much larger and more repeatable than previous studies. While it would be preferrable to base our analysis on more data sets, to the best of our knowledge, our study is based on a broader sample size than previous reports about defect detectors. Most of our data is public domain and encourage other researchers to test their methods on our data.

Secondly, we caution researchers against restricting their analysis of structural measurements and failure data to the simple techniques described in this paper. Although software defect detection may be a simple task, and simple fault models may be deployed as part of production software development efforts, the research underlying such models must still apply the full range of measurement and analysis techniques to develop the models in the first place. This ensures that a richer set of relationships between structural measures and fault content will be developed, and allows the development of meaningful benchmarks for the simpler models.

Those two cautions notwithstanding, our conclusion must be as follows. If in the usual case we see that accurate defect detectors can be found after trivially simple algorithms have rejected most of the structural features, then software defect detection is a very simple task indeed.

References

Almuallim, H. and T. Dietterich: 1991, 'Learning with Many Irrelevant Features'. In: *The Ninth National Conference on Artificial Intelligence*. pp. pp. 547–552, AAAI Press.

Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone: 1984, 'Classification and Regression Trees'. Technical report, Wadsworth International, Monterey, CA.

Dillon, W. and M. Goldstein: 1984, *Multivariate Analysis: Methods and Applications*. Wiley-Interscience.

Dumais, S., J. Platt, D. Heckerman, and M. Sahami: 1998, 'Inductive learning algorithms and representations for text categorization'. In: *The International Conference on Information and Knowledge Management*. pp. pp. 148–155.

Dvorak, D., R. Rasmussen, G. Reeves, and A. Sacks: 1999, 'Software Architecture Themes In JPL's Mission Data System'. In: *AIAA Space Technology Conference and Exposition, Albuquerque, NM*.

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

Gokhale, S. S. and M. R. Lyu: 1997, 'Regression Tree Modeling for the Prediction of Software Quality'. In: *Proceedings of the Third ISSAT International Conference on Reliability and Quality in Design, Anaheim, CA*. pp. 31–36.

Gunnalan, R., T. Menzies, K. Appukutty, S. A, and Y. Hu: 2003, 'Feature Subset Selection with TAR2less'. In: *Submitted to ICML'03*. Available from `http://menzies.us/pdf/03tar2less.pdf`.

Hall, M. and G. Holmes: 2003, 'Benchmarking Attribute Selection Techniques for Discrete Class Data Mining'. *IEEE Transactions On Knowledge And Data Engineering (to appear)*.

Hall., M. A.: 1998, 'Correlation-based feature selection for machine learning'. Ph.D. thesis, Department of Computer Science, University of Waikato, Hamilton, New Zealand.

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

Holte, R.: 1993, 'Very Simple Classification Rules Perform Well on Most Commonly Used Datasets'. *Machine Learning* **11**, 63.

Hu, Y.: 2002, 'Treatment learning'. Masters thesis, Unviersity of British Columbia, Department of Electrical and Computer Engineering. In preperation.

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*. pp. 66–73.

Khoshgoftaar, T. M. and E. B. Allen: 1999, 'Model Software Quality with Classification Trees'. In: H. Pham (ed.): *Recent Advances in Reliability and Quality Engineering*. World Scientific.

Kira, K. and L. Rendell: 1992, 'A Practical Approach to Feature Selection'. In: *The Ninth International Conference on Machine Learning*. pp. pp. 249–256, Morgan Kaufmann.

Kohavi, R. and G. John: 1997a, 'Wrappers for feature subset selection'. *Artificial Intelligence* pp. 273–324.

Kohavi, R. and G. H. John: 1997b, 'Wrappers for Feature Subset Selection'. *Artificial Intelligence* **97**(1-2), 273–324.

Kononenko, I.: 1994, 'Estimating attributes: Analysis and extensions of relief'. In: *The Seventh European Conference on Machine Learning*. pp. pp. 171–182, Springer-Verlag.

McCabe, T.: 1976, 'A Complexity Measure'. *IEEE Transactions on Software Engineering* **2**(4), 308–320.

Menzies, T., E. Chiang, M. Feather, Y. Hu, and J. Kiper: 2002a, 'Condensing uncertainty via Incremental Treatment Learning'. In: *Annals of Software Engineering*. Available from `http://menzies.us/pdf/02itar2.pdf`.

Menzies, T. and J. S. DiStefeno: 2002, 'Metrics that Matter'. In: *27th NASA SEL workshop on Software Engineering (submitted)*.

Menzies, T. and Y. Hu: 2001, 'Reusing models for requirements engineering'. In: *First International Workshop on Model-based Requirements Engineering*. Available from `http://menzies.us/pdf/01reusere.pdf`.

Menzies, T. and Y. Hu: 2002a, 'Agents in a Wild World'. In: C. Rouff (ed.): *Formal Approaches to Agent-Based Systems, book chapter*. Available from `http://menzies.us/pdf/01agents.pdf`.

Menzies, T. and Y. Hu: 2002b, 'Just enough learning (of association rules)'. In: *WVU CSEE tech report*. Available from `http://menzies.us/pdf/02tar2.pdf`.

Menzies, T., D. Raffo, S. on Setamanit, Y. Hu, and S. Tootoonian: 2002b, 'Model-based Tests of Truisms'. In: *Proceedings of IEEE ASE 2002*.

Menzies, T., J. D. Stefano, K. Ammar, K. McGill, P. Callis, R. Chapman, and D. J: 2003, 'When Can We Test Less?'. In: *Submitted to IEEE Metrics'03*. Available from `http://menzies.us/pdf/03metrics.pdf`.

Munson, J. and A. Nikora: 1998, 'Estimating Rates Of Fault Insertion And Test Effectiveness In Software Systems'. In: *Proceedings of the Fourth ISSAT International Conference on Reliability and Quality in Design.* pp. pp. 263–269.

Munson, J. and A. Nikora: 2002, 'Toward a Quantifiable Definition of Software Faults'. In: *Proceedings of the 13th IEEE International Symposium on Software Reliability Engineering*. IEEE Press.

Munson, J. C. and T. M. Khoshgoftaar: 1990, 'Regression Modeling of Software Quality'. *Information and Software Technology* **32**(2), 105–114.

Munson, J. C. and T. M. Khoshgoftaar: 1991, 'The Use of Software Complexity Metrics in Software Reliability Modeling'. In: *Proceedings of the International Symposium on Software Reliability Engineering, Austin, TX*.

Munson, J. C. and D. S. Werries: 1996, 'Measuring Software Evolution'. In: *Proceedings of the 1996 IEEE International Software Metrics Symposium*. pp. pp. 41–51, IEEE Computer Society Press.

Nikora, A. and J. Munson: 1997, 'Finding Fault with Faults: A Case Study'. In: *proceedings of the Annual Oregon Workshop on Software Metrics, Coeur d'Alene, ID*.

Provost, F., T. Fawcett, and R. Kohavi: 1998, 'The case against accuracy estimation for comparing induction algorithms'. In: *Proc. 15th International Conf. on Machine Learning*. pp. 445–453, Morgan Kaufmann, San Francisco, CA. Available from `htpp://citeseer.nj.nec.com/provost98case.html`.

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

Schneidewind, N. F.: 1997, 'Software Metrics Model for Integrating Quality Control and Prediction'. In: *Proceedings of the 8th International Symposium on Software Reliability Engineering, Albuquerque, New Mexico*. pp. 402–415.

Schneidewind, N. F.: 2001, 'Investigation of Logistic Regression as a Discriminant of Software Quality'. In: *Proceedings of the 7th International Software Metrics Symposium, London*. pp. 328–337.

Tian, J. and M. Zelkowitz: 1995, 'Complexity Measure Evaluation and Selection'. *IEEE Transaction on Software Engineering* **21**(8), 641–649.

Witten, I. H. and E. Frank: 1999, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann.

Appendix

A. Metrics

This appendix shows our standard tutorial on commonly used structure metrics.

A.1. McCabe

In order to facilitate the search for error-prone modules or functions, many tools have evolved over the past few years. One of the most popular ones (and the one being used extensively at NASA IV&V) is the McCabe IQ© package. This package can evaluate Ada, C and C++ source code, and provides many different types of software metrics.

The McCabe metrics are a collection of four software metrics: essential complexity, cyclomatic complexity, design complexity and LOC (McCabe, 1976). Of these four, all but LOC are metrics which were developed by T. J. McCabe. McCabe & Associates claim that these complexity measurements provide insight into the reliability and maintainability of a module. For example, around NASA IV&V, a cyclomatic complexity of over 10 or an essential complexity of over 4 is flagged as a module that will be difficult to maintain and/or debug. This paper will not attempt to make any refutation to those claims and practices; however, these metrics are also commonly used as predictors for error-prone modules. As this paper will demonstrate, these complexity measurements do not *always* point the way towards modules with increased error density.

The following paragraphs present a short overview of the three complexity metrics mentioned previously.

Cyclomatic Complexity, or $v(G)$, measures the number of *linearly independent paths*[6] through a program's flow graph[7]. $v(G)$ is calculated by $v(G) = e - n + 2$, 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).

Essential Complexity, or $ev(G)$, is the extent to which a flow graph can be "reduced" by decomposing all the sub-flow-graphs of G that are D-structured primes [8]. $ev(G)$ is calculated by $ev(G) = v(G) - m$ where m is the number of sub-flow-graphs of G that are D-structured primes. (Fenton and Pfleeger, 1995)

Design Complexity, or $iv(G)$, is the cyclomatic complexity of a module's reduced flow graph. The flow graph, G, of a module is reduced to eliminate any complexity which does not influence the interrelationship between design modules. This complexity measurement reflects the modules calling patterns to its immediate subordinate modules.

A.2. HALSTEAD

Another commonly used collection of software metrics are the Halstead Metrics (Halstead, 1977). They are named after their creator, Maurice H. Halstead. Halstead felt that software (or the writing of software) could be related to the themes which were being advanced at that time in the psychology literature. He created several metrics which are meant to encapsulate these properties; these metrics can be extracted by use of the McCabe IQ tool mentioned previously, and are discussed in detail below.

Halstead began by defining some basic measurements (these measurements are collected on a per module basis):

$$\begin{aligned}
\mu_1 &= \text{number of unique operators} \\
\mu_2 &= \text{number of unique operands} \\
N_1 &= \text{total occurrences of operators} \\
N_2 &= \text{total occurrences of operands} \\
\mu_1^* &= \text{potential operator count} \\
\mu_2^* &= \text{potential operand count}
\end{aligned}$$

These six metrics are self explanatory, with the possible exception of the potential operator/operand counts. Halstead defines μ_1^* and μ_2^* as the *minimum*

[6] A set of paths is linearly independent if no path in the set is a linear combination of any other paths in the set

[7] A flow graph is a directed graph where each node corresponds to a program statement, and each arc indicates the flow of control from one statement to another

[8] D-structured primes are also sometimes referred to as "proper one-entry one-exit sub-flow-graphs". For a more thorough discussion of D-primes, see (Fenton and Pfleeger, 1995)

possible number of operators and operands for a module. This minimum number would occur in a (potentially fictional) language in which the required operation already existed, possibly as a subroutine, function, or procedure. In such a case, $\mu_1^* = 2$, since at least two operators must appear for any function; one for the name of the function, and one to serve as an assignment or grouping symbol. μ_2^* represents the number of parameters, without repetition, which would need to be passed to the function or procedure.

Using these measurements, Halstead defined the *length* of a program P as: $N = N_1 + N_2$; the vocabulary of P is $\mu = \mu_1 + \mu_2$; and the *volume* of P (akin to the number of mental comparisons needed to write a program of length N) is $V = N * log_2\mu$.

A variant of V is V^* is the potential volume - the volume of the minimal size implementation of P: $V^* = (2+{\mu_2}^*)log_2(2+{\mu_2}^*)$. The *program level* of a program P with volume V is $L = V^*/V$. The inverse of level is *difficulty*; i.e. $D = 1/L$

According to Halstead's theory, we can calculate an estimate $\hat{L}$ of L as $\hat{L} = 1/D = \frac{2}{\mu_1} * \frac{\mu_2}{N_2}$. The intelligence content of a program, I, is $I = \hat{L} * V$ and the effort required to generate P is given by $E = \frac{V}{\hat{L}} = \frac{\mu_1 N_2 N log_2\mu}{2\mu_2}$ where the unit of measurement E is elementary mental discriminations needed to understand P. From E we can generate B (an estimate of the number of errors) using $B = K * E^{0.67}$ (where K is a language-specfic constant). Finally, according to Halstead, the required programming time T for a program of effort E is $T = E/18seconds$.