

How Simple is Software Defect Detection?

Kareem Ammar, Tim Menzies
Lane Department of Computer Science,
West Virginia University
PO Box 6109, Morgantown,
WV, 26506-6109, USA
kamar@csee.wvu.edu, tim@menzies.us

Allen Nikora
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Dr
Pasadena, CA 91109
allen.P.Nikora@jpl.nasa.gov

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 may 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; principal components analysis. fault models; metrics: product metrics; defect detectors; artificial intelligence: learning; feature subset selection.*

1 Introduction

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 methods proposed by Khoshgoftaar and Allen [14] and by Ghokale and Lyu [6]; Schneidewind's work on Boolean Discriminant Functions [35], Khoshgoftaar's application of zero-inflated Poisson regression

to predicting software fault content [13], and Schneidewind's investigation of logistic regression as a discriminant of software quality [36].

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 modelling complexity. In the reliability engineering literature, principal components analysis (PCA) [3] has been widely applied to resolve problems with structural code measurements; e.g. [29, 30]. 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

⁰Submitted to the 14th. IEEE International Symposium on Software Reliability Engineering ISSRE 2003, Denver, Colorado, Nov 17-20, 2003 issre2003.cs.colostate.edu/. April 18, 2003. Wp ref: wp/03/issre/simpledefects. Available on-line at <http://menzies.us/pdf/03simpledef.pdf>.

⁰This work was sponsored by the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program led by the NASA IV&V Facility and conducted at the West Virginia University (partially supported by NASA contract NCC2-0979/NCC5-685) and at the Jet Propulsion Laboratory, California Institute of Technology (under a contract with the National Aeronautics and Space Administration). The JPL work was funded by NASA's Office of Safety and Mission Assurance, Center Initiative UPN 323-08. That 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.

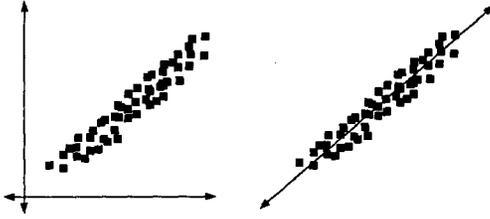


Figure 1. Transformation of axis.

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 domain1 ≤ 0.180
then NoDefects
else if domain1 > 0.180
  then if domain1 ≤ 0.371 then NoDefects
  else if domain1 > 0.371 then Defects

```

Here, “domain₁” is one of the domains found by PCA. This tree seems very simple, yet is very hard to explain to business clients users since “domain₁” is calculated using the following, somewhat intimidating, weighted sum:

$$\begin{aligned}
\text{domain}_1 = & 0.241 * \text{loc} + 0.236 * v(g) \\
& + 0.222 * \text{ev}(g) + 0.236 * \text{iv}(g) + 0.241 * n \\
& + 0.238 * v - 0.086 * l + 0.199 * d \\
& + 0.216 * i + 0.225 * e + 0.236 * b + 0.221 * t \\
& + 0.241 * \text{I}OC\text{Code} + 0.179 * \text{I}OC\text{Comment} \\
& + 0.221 * \text{I}O\text{Blank} + 0.158 * \text{I}OC\text{CodeAndComment} \\
& + 0.163 * \text{uni}Q\text{Op} + 0.234 * \text{uni}Q\text{Opnd} \\
& + 0.241 * \text{total}O\text{P} + 0.241 * \text{total}O\text{Pnd} \\
& + 0.236 * \text{branchCount}
\end{aligned}$$

(Here, $v(g)$, $ev(g)$, $iv(g)$ are the standard McCabe structural metrics [19] while the rest are either Halstead metrics [10] or simple variants on lines of code count.)

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

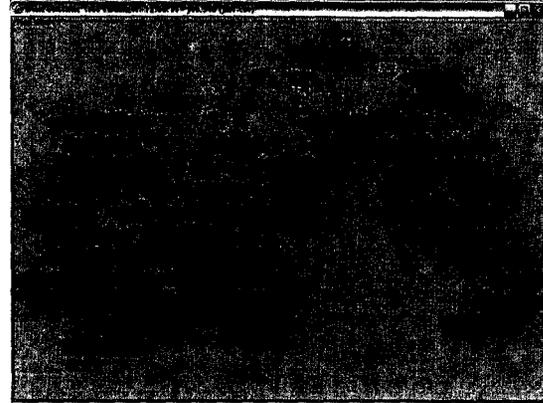


Figure 2. A large decision tree produced by the C4.5 decision learner [34] using all 22 metrics in the JM1 data set analyzed in this article.

to a class of *feature subset selection* (FSS) techniques which aim to remove superfluous features [7,8,17]. 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 representation 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 terms of the number of features rejected and the accuracy of the detectors learnt from the remaining features.

Unlike other studies (e.g. [27]), which contained a mere fifty observations, the experimental data used for this paper is large (hundreds to thousands of records) and is drawn from three different software projects. That is our conclusions are based on a broader experience base than previous work.

Another important feature of this study is that it is a *repeatable* experiment. Two of the three data sets used here publicly available¹ (and the third may be

¹<http://mdp.ivv.nasa.gov>, or <http://menzies.us/data>.

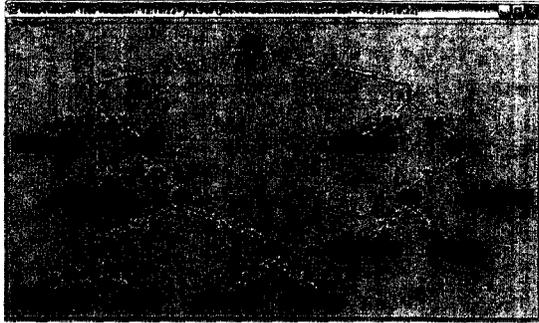


Figure 3. Small decision tree produced by C4.5 from the JM1 data set using just the three metrics selected by the TAR2less FSS method (described later in this paper).

available before the date of ISSRE 2003). These experiments also use freely distributed tools available online, such as the WEKA machine learning toolkit² and the TAR2 treatment learner [22–24]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 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 [21,27] 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<http://www.cs.waikato.ac.nz/~ml/weka/>
³<http://menzies.us/rx.html>

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 4 shows one study where, of the 42 features offered in the data set, only six were deemed significant by the learner.

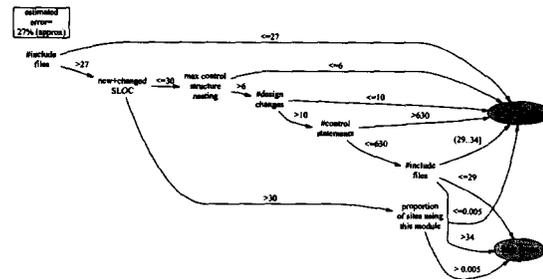


Figure 4. Predicting fault-prone modules [14]. Learned from data collected from a telecommunications system with > 10 million lines of code containing a few thousand modules.

For another example, Figure 6 shows 18 metrics given to a particular learner. Figure 5 shows what that learner generated. The key feature of Figure 5 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 [5]. Figure 7 shows that, with respect to the index of cumulative faults, not all features are equally associated with faults. Figure 7 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

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 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 [34] and J4.8 [38] decision tree algorithms use an information theoretic measure (entropy) to find its splits while the CART [2] decision tree learner uses another measure called the GINA index.

Bayesian Learning: An alternative to decision tree learning is Naive Bayesian learning [38]. 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 8. In that log, the frequency of playing some, or lots of golf is $P(\text{none}) = \frac{5}{14}$, $P(\text{some}) = \frac{3}{14}$ and $P(\text{lots}) = \frac{6}{14}$ respectively. In the special case where it is not windy (i.e. $E = \text{not windy}$) then the probabilities change to $P(\text{not windy}|\text{none}) = \frac{2}{8}$, $P(\text{not windy}|\text{some}) = \frac{3}{8}$, $P(\text{not windy}|\text{lots}) = \frac{3}{8}$. 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:

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

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

$$\begin{aligned} P(\text{none}|\text{not windy}) &= \frac{0.143}{0.143 + 0.214 + 0.214} = 0.250 \\ P(\text{some}|\text{not windy}) &= \frac{0.214}{0.143 + 0.214 + 0.214} = 0.375 \end{aligned}$$

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

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

Figure 8. Attributes that select for golf playing behavior.

$$P(\text{lots}|\text{not windy}) = \frac{0.214}{0.143 + 0.214 + 0.214} = 0.375$$

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 [12, 20, 22–24, 24, 25]. 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 8. 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 8. In the case of *outlook=overcast*, we play *lots* of golf all the time. In the case of *humidity ≥ 90*, we only play *lots* of golf in 20% of the cases. The net effect of these two select statements is shown in Figure 9.

The *WHERE* statements within a select statement can contain conjunctions of arbitrary size. Exploring all such conjunctions manually is a tedious task. TAR2 is an automatic tool for finding the strongest select statements; i.e., the statement that *most* selects for preferred behavior while *most* discouraging undesirable behavior. TAR2 calls this strongest select statement the "treatment" since it is a recommended action for improving the current situation. The algorithm is au-

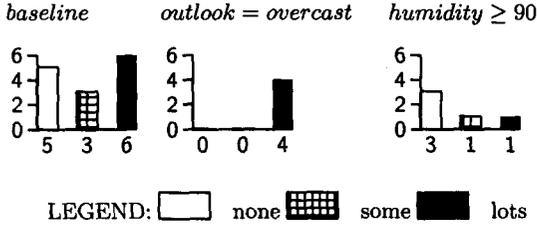


Figure 9. Changes to golf playing behavior from the baseline.

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 9 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 [11]. 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 [26]. Given a set of numeric metrics

$$attribute_1, attribute_2, \dots, attribute_n$$

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. *Thresholds* are then selected corresponding to equal areas under that distribution. For example, in one of the data sets we examine, the McCabe cyclo-matic 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:

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

$$\begin{aligned} Z^{-1}(area) &\approx 0.25 \\ \therefore v(g).threshold(area) &\approx 7.65 \end{aligned}$$

ROCKY generates one detector

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

for the range

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

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

4 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 [17] 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 [38]. 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.* [7] 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 modelling software reliability and fault content in the late 1980s and early 1990s, when Munson and Khoshgoftaar first developed the concept of relative complexity [29, 30], 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 [29]. In a later paper, they examined relationships between the relative complexity and software reliability [30]. 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 [31]. 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 [27], 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 [32], 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 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 [28]. They also initiated a collaboration with the Mission Data System, a mission software technology development effort at the Jet Propulsion Laboratory [5]. 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 methods 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 10 shows some WRAPPER results from ex-

	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%
Δ accuracy:	0%	6%	5%	4%	2%	1%	0.5%	0%	-25%	6%	7%

Figure 10. Feature subset selection using a WRAPPER of a decision tree learner. The Δ Accuracy figure is the difference in the accuracies of the theories found by decision tree learner using the *before* and *after* attributes. From [17].

periments by Kohavi and John [16]. 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 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 [4]. 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 [34]. A final comparison of this measure is used in feature selection.

RLF: Relief: Relief is an instance based learning scheme [15, 18]. 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 [18]. 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 [9]. This technique relies on a heuristic merit calculation that assigns high scores to subsets with features that are highly

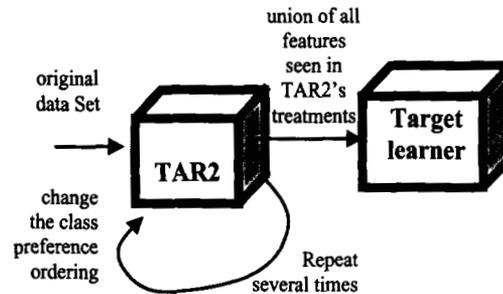


Figure 11. Tar2less algorithm.

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 [1].

TAR2less: Figure 11 shows the TAR2less FSS developed by Gunnalan, Menzies, *et al.* [7]. TAR2less runs TAR2 many times, each time targeting a different class; e.g. defects, no defects:

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

data set	# modules	% with defects
KC2	523	20%
AN1	1719	58%
JM1	10885	19%

Figure 12. Data sets used in this study. Here, a “module” is the equivalent of a C function.

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.

5 Experiments

Data Sets: The remainder of this paper is dedicated to a case study on three NASA projects, which will be referred to as “KC2”, “AN1”, and “JM1”. The projects are NASA C++ programs. All modules analyzed were built by NASA developers excluding several thousands modules that are COTS⁴ software. The McCabe and Halstead structural metrics were extracted from these systems and mapped to the defect logs kept for each project. The defect rate and the number of modules for each data set is shown in Figure 12.

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). This sampling policy would be inappropriate if we were building a customized defect detector for the AN1 project. However, we judge that this sampling policy is adequate for the purposes of testing different FSS strategies.

Methods: FSS was conducted using the PCA, CBS, IG, RLF, WRAPPER, and 1R implementations supplied with the WEKA machine learning toolkit⁵. We used our own implementations of ROCKY and TAR2⁶. TAR2less 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

⁴COTS is an acronym for Commercial Off The Shelf

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

⁶Available from <http://menzies.us/rx.html> and <http://menzies.us/pace.html>

Learner	Attributes	KC2	JM1	AN1
Original (C4.5)	21	82.11%	79.34%	65.90%
Original (Bayes)	21	83.65%	80.41%	58.27%
1R	1	82.95%	79.55%	65.02%
ROCKY	1	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. Bold entries denotes an increase over all baselines.

Bayes classifiers supplied within the WEKA. Assessing FSS via these two learners is quite standard in the FSS literature (e.g. [17]) 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 TAR2less when finding the best treatment of size N.

Results: Figure 13 shows the classification on 10-way generated by ROCKY and 1R. Figure 14, Figure 15 and Figure 16 show the average classification accuracies seen in 10-way cross validation runs of Naive Bayes and WRAPPER using just the features selected by PCA, CBS, IG, RLF, WRAPPER, and TAR2less.

The first line of Figure 14, Figure 15, and Figure 16 shows the results of running all available features through Naive Bayes and C4.5. Underlined entries mark the largest accuracy generated by any method. *Italicized* entries show where features found by FSS generated detectors with a higher accuracy than the baseline. Figure 17 is a summary table showing how often our FSS methods out-performed all baselines.

6 Discussion

Is throwing away information useful for defect detection? Surprisingly, it would seem so. Figure 17 shows

	Attributes	C4.5		Naive Bayes
		accuracy	tree Size	accuracy
Original	15	65.90%	131	58.27%
Tar2less	4	<u>66.82%</u>	119	58.61%
CFS	5	65.02%	17	<u>60.42%</u>
CBS	9	<u>66.07%</u>	81	57.57%
IG	4	64.55%	3	<u>60.65%</u>
RLF	4	<u>67.11%</u>	41	60.77%
PCA	7	65.77%	13	<u>61.87%</u>
Wrapper	5 (C4.5)	<u>66.88%</u>	37	
	2 (bayes)			<u>62.51%</u>
	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).

	Attributes	C4.5		Naive Bayes
		accuracy	tree Size	accuracy
Original	21	79.34%	677	80.41%
Tar2less	4	<u>81.06%</u>	11	<u>80.70%</u>
CFS	7	<u>80.51%</u>	63	<u>80.46%</u>
CBS	19	79.48%	671	80.25%
IG	4	<u>81.25%</u>	15	<u>80.43%</u>
RLF	4	<u>80.98%</u>	25	79.62%
PCA	8	<u>80.09%</u>	17	79.67%
Wrapper	5 (C4.5)	<u>85.57%</u>	15	
	2 (bayes)			<u>80.90%</u>
	mean	81.04%	mean	80.31%

Figure 15. JM1 FSS and learner accuracy runs. Baseline accuracies shown on line one. Italicized and blank entries have the same meaning as in Figure 14.

that in $35/48 = 73\%$ 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, 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 5 features selected from a space of 15 to 21 features. Hence, this study endorses FSS for defect detector generation.

What is a good FSS method for defect detection? This is unclear but the very simple FSS methods (ROCKY and 1R) ran very fast and resulted in highly accurate theories. Further, ROCKY out-performed the other FSS methods in the case of KC2 and JM1 but not AN1. The CPU-intensive WRAPPER method is slow to run but, of the more complex FSS methods, al-

	Attributes	C4.5		Naive Bayes
		accuracy	tree Size	accuracy
Original	21	82.11%	51	83.65%
Tar2less	2	<u>82.89%</u>	5	83.08%
CFS	2	<u>85.25%</u>	9	<u>84.10%</u>
CBS	6	<u>83.39%</u>	23	<u>83.72%</u>
IG	4	<u>84.29%</u>	3	<u>84.10%</u>
RLF	4	81.61%	5	82.18%
PCA	2	<u>82.57%</u>	5	<u>84.87%</u>
Wrapper	1 (C4.5)	<u>85.57%</u>	3	
	4 (bayes)			<u>85.19%</u>
	mean	83.45%	mean	83.86%

Figure 16. 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	TAR2	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
total	17/21	15/21	0/3	3/3	35/48

Figure 17. How often FSS generates theories of higher accuracy than using all available features.

ways generated the most accurate theory (using either C4.5 or Naive Bayes).

Comparatively speaking, how does PCA compare to other FSS methods? 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 [8]. Our results do not contradict their conclusions- we obtained high accuracies with CFS, RLF and WRAPPER, and never with CBS, IG, or PCA.

How simple is defect detection? 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.

Is accuracy the best way to judge the effectiveness of a defect detector? Perhaps not. A striking feature of Figure 14, Figure 15, and Figure 16 is the very small variance in the accuracy figures. In other work [26],

KC2	$ev(g) \geq 4.99$	85.28% accurate
JM1	$unique\ operands \geq 60.48$	81.10% accurate
AN1	$unique\ operands \geq 8.14$	66.08% accurate

Figure 18. Best detectors learnt by ROCKY.

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 [33]. 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 McCabe’s 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.

What is the best detector? 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 18 (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 $unique\ operands$ value was the feature that yielded the most accurate detectors in JM1 and AN1.
- The threshold value for the two detectors that use the Halstead metrics 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.

7 Conclusion

It is important to note the limits of this study. Currently, we have only explored feature subset selection for defect detectors using three data sets. Three data sets is insufficient to declare a general principle. Hence, we encourage other researchers to test our methods of their data.

Also, 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.

That caution 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

- [1] H. Almuallim and T. Dietterich. Learning with many irrelevant features. In *The Ninth National Conference on Artificial Intelligence*, pages pp. 547–552. AAAI Press, 1991.
- [2] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. Classification and regression trees. Technical report, Wadsworth International, Monterey, CA, 1984.
- [3] W. Dillon and M. Goldstein. *Multivariate Analysis: Methods and Applications*. Wiley-Interscience, 1984.
- [4] S. Dumais, J. Platt, D. Heckerman, and M. Sahami. Inductive learning algorithms and representations for text categorization. In *The International Conference on Information and Knowledge Management*, pages pp. 148–155, 1998.
- [5] D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks. Software architecture themes in JPL’s mission data system. In *AIAA Space Technology Conference and Exposition, Albuquerque, NM.*, 1999.
- [6] S. S. Gokhale and M. R. Lyu. 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*, pages 31–36, March 1997.
- [7] R. Gunnalan, T. Menzies, K. Appukutty, S. A., and Y. Hu. Feature subset selection with tar2less. In *Submitted to ICML'03*, 2003. Available from <http://menzies.us/pdf/03tar2less.pdf>.
 - [8] M. Hall and G. Holmes. Benchmarking attribute selection techniques for discrete class data mining. *IEEE Transactions On Knowledge And Data Engineering (to appear)*, 2003.
 - [9] M. A. Hall. *Correlation-based feature selection for machine learning*. PhD thesis, Department of Computer Science, University of Waikato, Hamilton, New Zealand, 1998.
 - [10] M. Halstead. *Elements of Software Science*. Elsevier, 1977.
 - [11] R. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11:63, 1993.
 - [12] Y. Hu. Treatment learning, 2002. Masters thesis, University of British Columbia, Department of Electrical and Computer Engineering. In preparation.
 - [13] T. Khoshgoftaar. An application of zero-inflated poisson regression for software fault prediction. In *Proceedings of the 12th International Symposium on Software Reliability Engineering, Hong Kong*, pages 66–73, Nov 2001.
 - [14] T. M. Khoshgoftaar and E. B. Allen. Model Software Quality with Classification Trees. In H. Pham, editor, *Recent Advances in Reliability and Quality Engineering*. World Scientific, 1999.
 - [15] K. Kira and L. Rendell. A practical approach to feature selection. In *The Ninth International Conference on Machine Learning*, pages pp. 249–256. Morgan Kaufmann, 1992.
 - [16] R. Kohavi and G. John. Wrappers for feature subset selection. *Artificial Intelligence*, pages 273–324, 1997.
 - [17] R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324, 1997.
 - [18] I. Kononenko. Estimating attributes: Analysis and extensions of relief. In *The Seventh European Conference on Machine Learning*, pages pp. 171–182. Springer-Verlag, 1994.
 - [19] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.
 - [20] T. Menzies, E. Chiang, M. Feather, Y. Hu, and J. Kiper. Condensing uncertainty via incremental treatment learning. In *Annals of Software Engineering*, 2002. Available from <http://menzies.us/pdf/02itar2.pdf>.
 - [21] T. Menzies and J. S. DiStefano. Metrics that matter. In *27th NASA SEL workshop on Software Engineering (submitted)*, 2002.
 - [22] T. Menzies and Y. Hu. Reusing models for requirements engineering. In *First International Workshop on Model-based Requirements Engineering*, 2001. Available from <http://menzies.us/pdf/01reusere.pdf>.
 - [23] T. Menzies and Y. Hu. Agents in a wild world. In C. Rouff, editor, *Formal Approaches to Agent-Based Systems, book chapter*, 2002. Available from <http://menzies.us/pdf/01agents.pdf>.
 - [24] T. Menzies and Y. Hu. Just enough learning (of association rules). In *WVU CSEE tech report*, 2002. Available from <http://menzies.us/pdf/02tar2.pdf>.
 - [25] T. Menzies, D. Raffo, S. on Setamanit, Y. Hu, and S. Tootoonian. Model-based tests of truisms. In *Proceedings of IEEE ASE 2002*, 2002.
 - [26] T. Menzies, J. D. Stefano, K. Ammar, K. McGill, P. Callis, R. Chapman, and D. J. When can we test less? In *Submitted to IEEE Metrics'03*, 2003. Available from <http://menzies.us/pdf/03metrics.pdf>.
 - [27] J. Munson and A. Nikora. 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.*, pages pp. 263–269, August 12-14 1998.
 - [28] J. Munson and A. Nikora. Toward a quantifiable definition of software faults. In *Proceedings of the 13th IEEE International Symposium on Software Reliability Engineering*. IEEE Press, 2002.
 - [29] J. C. Munson and T. M. Khoshgoftaar. Regression modeling of software quality. *Information and Software Technology*, 32(2):105–114, 1990.
 - [30] J. C. Munson and T. M. Khoshgoftaar. The use of software complexity metrics in software reliability modeling. In *Proceedings of the International Symposium on Software Reliability Engineering, Austin, TX*, May 1991.
 - [31] J. C. Munson and D. S. Werries. Measuring software evolution. In *Proceedings of the 1996 IEEE International Software Metrics Symposium*, pages pp. 41–51. IEEE Computer Society Press, May 1996.
 - [32] A. Nikora and J. Munson. Finding fault with faults: A case study. In *proceedings of the Annual Oregon Workshop on Software Metrics, Coeur d'Alene, ID*, May 11-13 1997.
 - [33] F. Provost, T. Fawcett, and R. Kohavi. The case against accuracy estimation for comparing induction algorithms. In *Proc. 15th International Conf. on Machine Learning*, pages 445–453. Morgan Kaufmann, San Francisco, CA, 1998. Available from <http://citeseer.nj.nec.com/provost98case.html>.
 - [34] R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, 1992. ISBN: 1558602380.
 - [35] N. F. Schneidewind. Software metrics model for integrating quality control and prediction. In *Proceedings of the 8th International Symposium on Software Reliability Engineering, Albuquerque, New Mexico*, pages 402–415, November 1997.
 - [36] N. F. Schneidewind. Investigation of logistic regression as a discriminant of software quality. In *Proceedings of the 7th International Software Metrics Symposium, London*, pages 328–337, April 2001.
 - [37] J. Tian and M. Zelkowitz. Complexity measure evaluation and selection. *IEEE Transaction on Software Engineering*, 21(8):641–649, Aug. 1995.
 - [38] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.