On the Value of Learning From Defect Dense Components for Software Defect Prediction

Hongyu Zhang
School of Software,Tsinghua University
Beijing 100084, China
hongyu@tsinghua.edu.cn

Adam Nelson, Tim Menzies
CS & EE, WVU, Morgantown, USA
rabituckman@gmail.com,
tim@menzies.us

ABSTRACT

BACKGROUND: Defect predictors learned from static code measures can isolate code modules with a higher than usual probability of defects.
AIMS: To improve those learners by focusing on the defect-rich portions of the training sets.
METHOD: Defect data CM1, KC1, MC1, PC1, PC3 was separated into components. A subset of the projects (selected at random) were set aside for testing. Training sets were generated for a NaiveBayes classifier in two ways. In sample the *dense* treatment, the components with higher than the median number of defective modules were used for training. In the *standard* treatment, modules from any component were used for training. Both samples were run against the test set and evaluated using recall, probability of false alarm, and precision. In addition, under sampling and over sampling was performed on the defect data. Each method was repeated in a 10-by-10 cross-validation experiment.
RESULTS: Prediction models learned from defect dense components out-performed *standard* method, *under* sampling, as well as *over* sampling. In statistical rankings based on recall, probability of false alarm, and precision, models learned from dense components won *4-5 times* more often than any other method, and also lost the least amount of times.
CONCLUSIONS: Given training data where most of the defects exist in small numbers of components, better defect predictors can be trained from the defect dense components.

Categories and Subject Descriptors

I.5 [**learning**]: machine learning; D.2.8 [**software engineering**]: product metrics

General Terms

Algorithms, experimentation, measurement

Keywords

defect prediction, sampling, defect dense components, ceiling effect

PROMISE2010, Sep 12-13, 2010. Timisoara, Romania

1. INTRODUCTION

It is widely believed that some internal properties of software (e.g., metrics) have relationship with the external properties (e.g., defects). Many prediction models have been proposed based on software metrics. For example, Menzies et al. [20] performed defect predictions for five NASA projects using static code metrics. In their work, probability of detection (recall) and probability of false alarm (pf) are used to measure the accuracy of a defect prediction model. Their models generate the average results of recall = 71% and pf = 25%, using a Naive Bayes classifier.

Zhang and Zhang [40] pointed out that the Menzies et al. results are not satisfactory when precision is considered. They found that high recall and low pf do not necessarily lead to high precision. In reply, Menzies et al. [19] noted that for the defect data in the PROMISE repository, the percentage of modules with defects is low. They showed that, for such data, it is difficult to achieve both high precision *and* high recall (that mathematical argument is reproduced in §3.1).

This paper disputes the conclusions of Menzies et al. We achieve better defect predictors with an approach that originated in the following observation:

> *With software projects, most of the defects occur in a minority of components.*

In the general research community, this is not a new observation. It has been repeatedly demonstrated that most defects occur in a small number of components (e.g. see [9]). However, within the PROMISE community, this is a new finding. To the best of our knowledge, this effect has not been reported previously, even though the PROMISE defect data sets have been the subject of intense scrutiny (perhaps this community has been so focused on the *learner* that they miss important aspects of the *data*).

What is new in this paper is the recognition that we can *exploit* this distribution to build better defect predictors. We test the speculation that if we focus more on the densely defective components, we might generate better defect predictors.

The rest of this paper explores this possibility. In the following, we say that a software *project* is divided into *components* and *components* have *modules*. In this terminology:

- The *module* is the smallest unit of compilation; e.g. a function in "C", or a method in "JAVA";
- A *component* is a large groupings of many modules.

Typically, one *project* contains dozens to hundreds of *components*, and each *component* contains dozens to hundreds of *modules*.

Using this terminology, we can summarize our findings as follows: training on densely defective components produces better defect predictors than otherwise.

2. BACKGROUND

Before we present our specific results, this section reviews the general area of defect prediction.

One method for assessing software is to extrapolate the curve of post-release failures. In this *reliability engineering* approach, analysts use knowledge of how the frequency of failures seen in a running system changes over time [26, 15]. Given a history of defects in a running system, reliability engineers extrapolate a *reliability growth curve* that predicts the future probability of failure on demand of a developing system. Such curves can predict how long before a system reaches a required level of reliability.

Reliability engineering assumes that the system is running; i.e. not until the entire system is assembled. It is undesirable to delay software rework until after the system is running since the longer a defect remains in code, the more expensive it is to remove [4].

Prior to building a fully functioning system, software engineers build modules then, usually, apply some quality assurance (QA) technique to assess the quality of those modules (those techniques include inspections, unit tests, static source code analyzers, etc). A record of the results of those QA techniques is a *defect log*. Note that this log can accumulate from very early in the development cycle (i.e. even before the entire system is running).

Defect predictors can be learned from these historical logs, if the logs contain tables of data. In those tables:

- Rows describe data from one *module*; a.k.a. "functions" or "methods" or "procedures".
- Columns describe static code features such as lines of code measures, the McCabe measures, and the Halstead measures. The class label "*defective*" is also a column, whose values are either *true* or *false*

For these data sets, the data mining goal is to learn a binary predictor for $defective$ modules from past projects that can be applied to future projects.

2.1 How can these defect predictors be used?

Defect predictors can be used to adjust QA budgets that are inappropriately focused on the wrong parts of the code.

During development, developers *skew* their limited quality assurance (QA) budgets towards artifacts they believe most require extra QA. For example, it is common at NASA to focus QA more on the on-board guidance system than the ground-based database storing scientific data collected from a satellite.

This skewing process can introduce an inappropriate bias to QA. If the QA activities concentrate on project artifacts, say *A,B,C,D*, then that leaves *blind spots* in *E,F,G,H,I,....* Blind spots can compromise high assurance software. Leveson remarks that in modern complex systems, unsafe operations often result from an unstudied interaction between components [14]. For example, Lutz and Mikulski [17] found a blind spot in NASA deep-space missions: most of the mission critical *in-flight* anomalies resulted from errors in *ground software* that fails to correctly collect in-flight data.

To avoid blind spots, one option is to rigorously assess all aspects of all software modules. But this is impractical. Software project budgets are finite and QA effectiveness increases with QA effort:

- A *linear* increase in the confidence C of finding faults takes *exponentially* more effort. For example, to detect one-in-a-thousand module faults, moving C from 90% to 94% to 98% takes 2301, 2812, and 3910 black box tests (respectively)[1].
- The infamous state space explosion problem imposes strict limits on how much a system can be explored via automatic formal methods [21]. Lowry et.al. [16] and Menzies and Cukic [18] offer numerous other examples where assessment effectiveness is exponential on effort.

Exponential costs quickly exhausts finite QA resources. Hence, blind spots can't be avoided and must be managed. Standard practice is to apply the best available assessment methods on the sections of the program that the best available domain knowledge declares is the most critical. We endorse this approach. Clearly, critical sections require the best known assessment methods, in hope of minimizing the risk of safety or mission critical failure occurring post deployment. However, this focus on certain sections can blind us to defects in other areas which, through interactions, may cause similarly critical failures. Therefore, the standard practice should be augmented with a *lightweight sampling policy* like defect predictors that (a) explores the rest of the software and (b) raises an alert on parts of the software that appear problematic. This sampling approach is incomplete by definition. Nevertheless, it is the only option when resource limits block complete assessment.

2.2 But Does It Work?

Compared to certain reports of standard industrial practice, defect predictors are better than humans are identifying which modules will require further QA. Previously we have reported studies where defect predictors had probabilities of detection (recall) and probabilities of false alarm (pf) of (recall,pf)=(71%,25%), on average [20]. These values are higher than known results from manual inspection methods. For example, a panel at *IEEE Metrics 2002* [31] concluded that manual software reviews can find ≈60% of defects[2]. In other work, Raffo found that the defect detection capability of industrial review methods can vary from

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

for full Fagan inspections [6] to

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

for less-structured inspections [30] (note: "$TR(a, b, c)$" is a triangular distribution with min/mode/max of a, b, c).

Defect prediction can also be faster than human-intensive methods. Static code features can be automatically and cheaply extracted from source code, even for very large systems [27]. By contrast, other methods such as manual code reviews are labor-intensive. Depending on the review methods 8 to 20 LOC/minute can be inspected and this effort repeats for all members of the review team, which can be as large as four or six [22].

Also, there are industrial reports where this style of defect prediction was found pragmatically useful by industry:

- Defect predicting technology has been commercialized in the *Predictive* tool and sold across the United States. One company used it to manage safety critical software for fighter aircraft (the software controlled a lithium ion battery, which can overcharge and possibly explode). After applying a more expensive tool for structural code coverage was applied, the

[1] A randomly selected input will find a fault with probability x. Voas observes [35] that after N random black-box tests, the chance of the inputs not revealing any fault is $(1-x)^N$. Hence, the chance C of seeing the fault is $1-(1-x)^N$ which can be rearranged to $N(C, x) = \frac{log(1-C)}{log(1-x)}$. For example, $N(0.90, 10^{-3}) = 2301$.

[2] That panel supported neither Fagan's claim [7] that inspections can find 95% of defects before testing or Shull's claim that specialized directed inspection methods can catch 35% more defects that other methods [32].

company ran Predictive on the same code base. Predictive produced consistent results with the more expensive tools while being able to faster process a larger code base than the more expensive tool [34].
- Defect predictors developed at NASA [20] have been applied to software development companies in another country (in Turkey). When inspection teams focused on the modules that trigger the defect predictors, they found up to 70% of the defects using 40% of the effort (measured in staff hours) [2].
- A subsequent study on the Turkish software compared how much code needs to be inspected using random selection vs selection via our defect predictors. Using random testing, 87% of the files would have to be inspected in order to detect 87% of the defects. However, if the inspection process was restricted to the 25% of the files that trigger the defect predictors, then 88% of the defects could be found. That is, the same level of defect detection (after inspection) can be achieved using $\frac{87-25}{87} = 71\%$ less effort [1].

2.3 Building Better Detectors

There are several proposals for improving defect predictors. One argument is to augment these static code attributes with *social metrics* describing patterns of interactions between developers. This has proved useful in some domains [28] but not in others [36].

Another approach is to use *other static code measures* such as *churn* (the rate at which the code base changes) [10]. As far as we are aware, there is no definitive result showing that some static code measures are better than any other.

As to *use of better data mining algorithms*, the current state of the art is curiously static:

- Khoshgoftaar and Seliya [12] performed an extensive study on NASA JM1 and KC2 datasets using 25 classification techniques with 21 static code metrics. They observed low prediction performance, and they did not see much improvement by using different classification techniques.
- Lessmann et al. report an extensive study on the statistical differences between 19 data miners commonly used for defect prediction [13]. The learners' performance was remarkably similar: all but four shared top ranking in their analysis.

Recently, Milton and Menzies et. al. have had some success with *tuning data miners to the local business goals*. Their WHICH learner performs a stochastic beam search across combinations of attribute ranges to maximize a domain-specific scoring function. In a study with one scoring function, WHICH significantly outperformed standard methods like C4.5 and Naive Bayes [33].

While the WHICH study is indeed interesting, it only wins by changing the rules of the game. In standard defect prediction, the learned detectors are assessed via their precision, false alarm rate, or recall (probability of detection). Note that by "standard defect prediction" we mean every piece of research referenced in this paper (with the sole exception of WHICH) and every defect prediction paper ever published at PROMISE. Therefore, having noted the WHICH experiments, the rest of this paper will only explore standard defect prediction.

3. THE CEILING EFFECT

The high water mark in this field has been static for several years. For example, for four years we have been unable to improve on our 2006 results [20]. Other studies report the same *ceiling effect*: many methods learn defect predictors that perform statistically insignificantly different to the best results. After a careful study 19 data miners for learning defect predictors seeking to maximize the area under the curve of detection-vs-false alarm curve, [13] concludes

> *...the importance of the classifier model is less than generally assumed ... practitioners are free to choose from a broad set of models.*

If better data miners do not offer improvements, perhaps it is time to explore other approaches. If we cannot *improve the data miners*, perhaps we can *improve the data*? To understand this approach, in this section:

- we review the mathematics of defect prediction. This mathematical analysis will highlight the importance of the neg/pos ratio. The neg/pos ratio expresses the percentage of defective modules in the training set: if a training set contains 10% defective modules, then $neg/pos = 9$.
- we then present experiments that carefully select the training data in order to change the neg/pos ratio.

3.1 Defect Prediction Mathematics

Let $\{A, B, C, D\}$ denote the true negatives, false negatives, false positives, and true positives (respectively) found by a binary detector. Certain standard measures can be computed from A, B, C, D:

$pd = recall =$	$D/(B+D)$
$pf =$	$C/(A+C)$
$prec = precision =$	$D/(D+C)$
$acc = accuracy =$	$(A+D)/(A+B+C+D)$
$bal = balance =$	$1 - \sqrt{recall^2 + (1-pf)^2}$
$f - measure =$	$2 * recall * prec/(recall + precision)$
$neg/pos =$	$(A+C)/(B+D)$

The last measure (neg/pos) is most important to the subsequent discussion. The following expression offers a relationship between $prec, recall, pf$. Note that this relationship changes according to the ratio of the target class in the training data; i.e. the $\frac{neg}{pos}$ ratio.

$$prec = \frac{D}{D+C} = \frac{1}{1+\frac{C}{D}} = \frac{1}{1 + neg/pos \cdot pf/recall} \quad (1)$$

which can be rearranged to

$$pf = \frac{pos}{neg} \cdot \frac{(1-prec)}{prec} \cdot recall \quad (2)$$

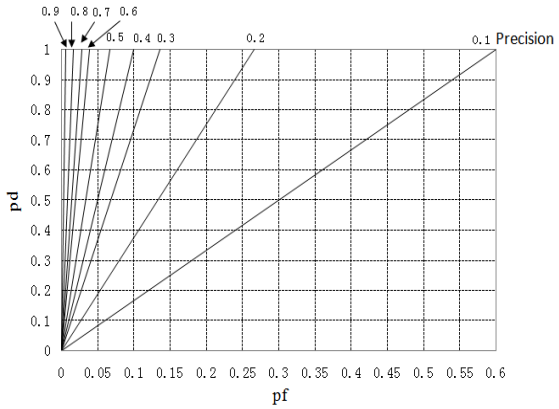
Menzies et al. [19] called this expression *Zhangs' equation* after the authors of the paper that first reported it [40]. Observe how, when $recall$ is fixed then the false alarm rate becomes controlled by precision and a fixed constant determined by the data set being examined; i.e. when ($\alpha = neg/pos$) and $recall = 1$ then:

$$pf = \alpha \cdot \frac{1-prec}{prec} \quad (3)$$

From this equation, it is clear that for any targeted recall value, *increasing* precision requires *decreasing* false alarm rates; e.g. for $prec \in \{0.5, 0.70, 0.9, 0.95\}$, pf becomes $\{1, 0.43, 0.11, 0.005\}$, respectively. As neg/pos increases, high recall&precision is only possible when pf becomes vanishingly small. For example, in the ranges $0.65 \leq prec, recall \leq 0.8$, Zhangs' equation reports that pf falls into the following ranges:

- $0.023 \leq pf \leq 0.062$ for $neg/pos = 7$;
- $0.0108 \leq pf \leq 0.0287$ for $neg/pos = 15$;
- $0.007 \leq pf \leq 0.0017$ for $neg/pos = 250$;

Figure 1: The change of $precision$ with pf and pd when $neg/pos = 15$



In our experience [23, 20, 24], such small pf values are not achievable, at least with the defect data in the PROMISE repository. Figure 1 illustrates how $precision$ changes with pf and pd values, assuming $neg/pos = 15$.

In summary, Zhangs' equation represents a fundamental limit on our ability to improve defect predictors for domains where the defective modules are relatively infrequent. This analysis is highly pertinent to the PROMISE defect data sets that have neg/pos ratios up to 249. That is, unless we can do something about these neg/pos ratios, the mathematics of defect prediction assert that further improvements in defect prediction will require relaxing the goal of building defect predictors with high precision *and* high recall.

3.2 Changing the neg/pos Ratio

One obvious way to change neg/pos ratio is to *over-sample* or *under-sample* the training data. Both methods might be useful in data sets with highly imbalances class distributions. In *over-sampling*, randomly selected instances from the minority class are copied. In under-sampling, instances are selected until the combined number of instances with other classes is equal to the number of instances with the desired goal. Over-sampling typically grows the size of the dataset while under-sampling results in a much smaller dataset.

At PROMISE'08, Menzies et al. [25] reported the following sampling experiment. Different sampling policies (over-sampling, under-sampling, and "no treatment") were applied to the NASA datasets using 10-way cross-validation. For the "no treatment" experiments, the raw data was used for training and testing without any adjustment to the class frequencies.

In these studies, two data miners were used to evaluate known over- and under- sampling results described in the literature [5, 11]: a Naive Bayes classifier and the J48 decision tree learner [37, 29]. Other learners were not used since, as discussed above, Lessmann et al. reported that experiments with other learners have yet to be productive [13].

Menzies et al demonstrated in the paper that:

- "No treatment" performed as well as under-sampling
- Over-sampling did not improve classifier performance. This result is consistent with Drummond & Holte's sub-sampling experiments [5] and the sub-sampling classification tree experiments of Kamei et.al. [11].

Rank	Treatment	f-measure percentiles 25%	50%	75%	2nd quartile median, 3rd quartile
1	j48 / none	27	86	94	
2	j48 / over	33	80	92	
3	nb / none	33	69	81	
4	nb / under	33	67	79	
4	nb / over	33	67	79	
5	j48 / under	30	53	79	

Figure 2: Over- & under- & no sampling results. Sorted descending by median $f - measure$ results ($f - measure$ is defined in §3.1). The right-hand side show median values (as a circle) within a 25% to 75% percentile range. The *rank*, shown left-hand-side, come from the statistical analysis of Figure 3. Three methods share top rank: NB/none, NB/under, j48/under.

rank	treatment	win	loss	ties
1	j48 / none	5	0	0
2	j48 / over	4	1	0
3	nb / none	3	2	0
4	nb / under	1	3	1
4	nb / over	1	3	1
5	j48/ under	0	5	0

Figure 3: Statistical tests on the Figure 2 results: sorted in ascending order on the number of losses (so *better* methods appear at the *top* of the table. First column shows a comparison of one treatment against the others. Two treatments have the same rank if their median ranks are statistically different (Mann-Whitney, 95% confidence).

- Curiously, throwing away data (i.e. under-sampling) did not degrade the performance of the learner. In fact, in the case of j48, throwing away data improved the median $balance$, as defined in §3.1, performance from around 40% to over 70%. The implications of this "under-sampling is ok" result are discussed in [25].

In order to maintain consistency with the main experiment of this paper, we replicated the experiment described in [25] using the same data sets. The results were evaluated using f-measure, which is the weighted average of the performance of a treatment based on $precision$ and $recall$. Figure 2 shows the results of the replicated experiment with f-measures.

Figure 2 and Figure 3 highlight the fact that over- and under-sampling methods did not necessarily improve prediction performance when f-measure was considered. Here we can see that, again, "no treatment" was not only competitive with the other two, it was the best treatment.

In summary, under-sampling/over-sampling approaches do not appear to be promising. However, for rigor, in the remaining sections of this article we will test them alongside the proposed method: learning prediction models from defect-dense components.

4. BREAKING THROUGH THE CEILING

4.1 The Distribution of Defects

In this paper, we exploit the naturally occurring distributions inside defect data sets. A repeated result [38, 39] is that in a large software system, the distribution of defects are skewed - that a small number of modules accounts for a large proportion of the defects. For example:

- In Eclipse 3.0, 20% of the largest packages are responsible for 60.34% of the pre-release defects (defects found six months before the release) and 63.49% of post-release defects (defects found six months after the release).
- At the file level, 20% of the largest Eclipse 3.0 files are responsible for 62.29% pre-release defects and 60.62% post-release defects.
- These results are consistent with those reported by other researchers such as Fenton and Ohlsson [8] and Andersson and Runeson [3].
- We further find that the distribution of defects across modules can be formally expressed using the Weibull function.

The skewed distribution of defects is also found in the PROMISE defect datasets; i.e. a few modules (at the function/method level) have a large number of defects and a large number of modules have a few defects. As an example, Figure 4 shows the distribution of defects over KC1 and PC3 modules from the PROMISE repository. All modules are ranked by the number of defects they are responsible for. Clearly the distributions are highly skewed- a few modules have many defects and most modules have 0 or 1 defect. We find that the top 5% "most defective" PC3 modules contain 68.34% of the defects, the top 10% "most defective" PC3 modules contain 98.84% of the defects. For KC1, the top 5% modules contain 55.81% defects and the top 10% modules contain 77.90% defects. Furthermore, we find that the distribution of defects also follows the Weibull distribution.

For the PROMISE defect datasets, we also perform component-level analysis. For each data set, components are extracted (using a unique identifier) containing both defective and non-defective modules (also labeled with a unique value). The data is shown in Table 1. We found that different components have different defect "densities". For example:

- For PC3, 6 out of 29 (20.69%) "most-defective" components contains 77.61% defects and 70% defective modules. The NEG/POS ratios in these components range from 0.93 to 2.70, while for all modules in PC3, the overall NEG/POS ratio is 8.77. Also, 14 out of 29 (48.28%) "most-defective" components contains 98.07% defects and 97.50% of defective modules. The average NEG/POS ratio in these components is 5.20. The defect dense components have much smaller NEG/POS ratios.
- For KC1, 6 out of 18 (33.33%) "most-defective" components contains 65.71% defects and 77.85% defective modules. These components have NEG/POS ratios ranging from 0.7 to 4.7, while for all modules in KC1, the overall NEG/POS value is 5.5. Also, 9 out of 18 (50%) components contain 91.24% defects and 89.23% defective modules. The average NEG/POS ratio in these components is 4.30. The defect dense components have also smaller NEG/POS ratios.

We obtained similar results for other NASA projects. In summary, most of the defects are found in the minority of components, which have lower NEG/POS ratios.

4.2 Learning from Defect Dense Components - An Experiment

The goal of this experiment is to assess the value of component-level, rather than project-level, training for defect predictors. For the component-level training, we will favor training data from components with higher than usual defect frequencies.

Five NASA defect data sets from the PROMISE repository were used: CM1, KC1, MC1, PC1, PC3. These data sets were chosen because:

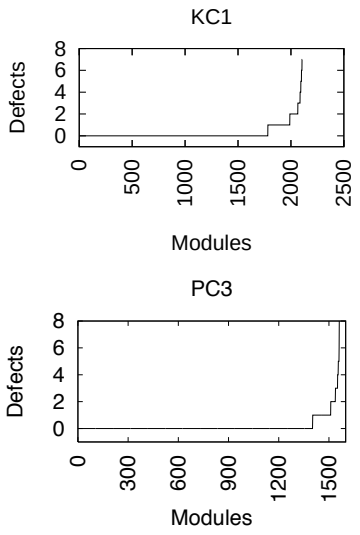


Figure 4: The Distribution of Defects in KC1 and PC3

- They have been studied in the field extensively;
- They are widely available to the PROMISE community;
- They are projects comprised of numerous components.

The subset of components with high defect densities were extracted as follows:

- If the number of defective modules per component exceeds the median number of defective modules across all components in that data set, it is labeled as a defect-$dense$ component.

For example, in Figure 5 the bottom horizontal line represents the median number of defects in the KC1 and PC3 data sets. Components existing above this line are labeled $dense$, while those lying under it are denoted as $sparse$.

A simple description of the experiment can be given as follows. For each data set, dense components are extracted. When a dense component is located, a Naive Bayes classifier is trained on all modules within this component. Testing is conducted on modules belonging to all components of that data set $except$ the one used to train the learner. Each defect-dense component is visited and used in this fashion.

A more thorough explanation is given by the pseudocode in Figure 6. Lines 1 and 5 of Figure 6 illustrate the use of the 10 X 10-way cross validation used in the experimental process. The standard 10 X 10-way cross validation operates by selecting 90% of the data randomly for training, and the remaining 10% for testing (this process is then repeated 10 times to guard against order effects). Since the objective is to analyze the performance of training on components containing a high number of defective modules, a minute alteration was made to cross-validation for this experiment. A "pool" of training data was constructed by focusing on only those instances $within$ a dense component, as in line 3 of the pseudocode. The available pool of testing instances, thus, are gathered from the $remaining$ components in the data set (line 4). This is employed to prevent training and testing on modules within the

Project	language	#Defects	#Modules	#Defective Modules	#Components	#Defective Components	#Dense Components
CM1	C++	70	505	48	20	9	9
KC1	C++	525	2107	325	18	18	9
MC1	C++	79	9466	68	57	26	26
PC1	C++	139	1107	76	29	17	14
PC3	C++	259	1563	160	29	17	14

Table 1: The component data for NASA defect datasets.

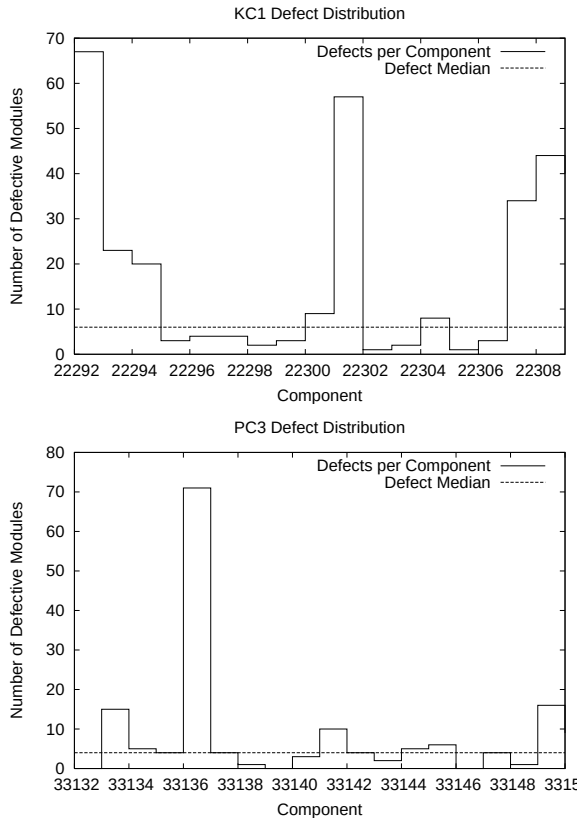


Figure 5: Defect distributions of components found in the KC1 and PC3 data sets. Note that only a small number of components contain a relatively high number of defective modules.

```
1 For run = 1 to 10
2   For each dense component C in data set D
3     Train = C
4     Test  = D - C
5     For bin = 1 to 10
6        Test'  = 10% of Test (picked at random)
7        Train' = 90% of Train (picked at random)
8        Naive Bayes (Train', Test')
9     end bin
10   end component
11 end run
```

Figure 6: The experiment on learning prediction models from dense components. The experiment performs training on modules residing in dense components, and testing on modules contained in all other components in the data set.

Rank	Treatment	recall percentiles 25%	50%	75%	2nd quartile median, 3rd quartile
1	Train on Dense Components	31	69	91	
1	Train on All Components	35	71	93	

0 50 100

Figure 7: $Recall$ values for learning on dense components compared to learning on all components across all data sets, sorted by statistical ranking via a Mann-Whitney test at 95% confidence (row i in the table has a different rank to the one before if it is statistically different to the results from all rows with the next lowest rank; and the median is different).

Rank	Treatment	pf percentiles 25%	50%	75%	2nd quartile median, 3rd quartile
1	Train on Dense Components	0	15	52	
2	Train on All Components	0	26	65	

0 50 100

Figure 8: PF values for learning on dense components compared to learning on all components across all data sets, sorted in the same manner as Figure 7.

same component. Lines 6 and 7 illustrate collecting 90% of the current dense component's instances as the final training set $Train'$, and 10% of the modules from the available instances in components not labeled dense as $Test'$.

Line 8 of Figure 6 executes the classifier (in this case, Naive Bayes) on the previously created training and testing sets $Train'$ and $Test'$. The Naive Bayes classifier was utilized because of its speed, and also for the fact that it has been shown to perform well on PROMISE defect data against other learners [13].

Determining the benefits of training defect prediction models using fewer, but more densely-packed components also requires the comparative analysis of learning from all components (and thus all modules). In addition, we also compare the proposed method with the over- and under- sampling methods. The experimental results for five NASA datasets are shown in the following section.

4.3 Experimental Results

The metrics used in the analysis of comparing results from training on dense components over the traditional method of using all components in the data set are $recall$, pf and $precision$.

Figure 7, Figure 8 and Figure 9 show statistical rankings of each treatment, as well as *quartile charts* displaying the median and variance of each metric for the $combined$ data sets, as a whole, used in the experiment:

- The black circle in the center of each plot is the median;
- The line going from left to right from this circle shows the second and third quartile respectively.
- Each chart is $ranked$ (see left-hand-side column) by a Mann-

Project	Recall		0% 50% 100%	Prob. False Alarm (Pf)		0% 50% 100%	Precision		0% 50% 100%
CM1	1	All		1	Dense		1	All	
	2	Over		2	Over		1	Over	
	2	Under		2	Under		1	Under	
	3	Dense		2	All		1	Dense	
KC1	1	Dense		1	All		1	Dense	
	1	Under		1	Over		1	All	
	1	All		1	Under		1	Under	
	1	Over		1	Dense		1	Over	
MC1	1	Over		1	Over		1	Dense	
	1	Under		1	Under		1	All	
	2	All		2	All		1	Under	
	3	Dense		3	Dense		1	Over	
PC1	1	Dense		1	Dense		1	Dense	
	2	Over		2	Over		2	All	
	2	All		2	All		2	Under	
	3	Under		3	Under		3	Over	
PC3	1	Under		1	Dense		1	Dense	
	1	Over		2	Over		1	Under	
	2	All		2	Under		1	All	
	2	Dense		3	All		1	Over	

Table 2: Result statistics per data set. The numeric value next to each treatment represents its Mann-Whitney rank, to the right of each treatment lies the quartile chart for each. Each metric is either sorted by ranking, or in the case of a tie, descending pd and $prec$ or ascending pf.

Rank	Treatment	precision percentiles 25%	50%	75%	2nd quartile median, 3rd quartile
1	Train on All Components	20	78	95	
1	Train on Dense Components	12	75	96	
					0 50 100

Figure 9: $Precision$ values for learning on dense components compared to learning on all components across all data sets, sorted in the same manner as Figure 7.

Whitney test (95% confidence).

We prefer quartile charts of performance to other summarization methods for a multitude of studies, as they offer a very succinct summary of a large number of experiments.

Over all the data sets studied here, training on dense components (those containing a higher number of defective modules) yields similar medians (for $recall$ and $precision$). However, dramatic gains are seen with pf: the median error rates decrease by nearly half.

The overall results of Figure 7, Figure 8 and Figure 9 obscure some interesting effects. If we look at the results for individual data sets, as shown in Table 2, a more interesting result appears. This table is divided into five data sets (CM1, KC1, MC1, PC1, PC3), three evaluation criteria (recall, pf, precision), and four treatments ($dense$, all, $over$, $under$). Quartile charts are shown, along with a statistical analysis that ranks pairs of treatments:

- If two treatments score $1, 1$, then there is no statistically significant difference in their performance.
- If two treatments score $1, 2$, then they are statistically different and the first one has a better median than the other.

A summary of these statistical rankings is shown in Table 3. Each treatment is assigned one of $\{+, 0, -\}$ depending on if it won, tied, lost in the statistical rankings of Table 2 (based on a Mann-Whitney test at 95% confidence). For example, in CM1:

- The four treatments tied on precision;
- Training on all produced a better $recall$ than training on $dense$, $over$, and $under$;
- Training on $dense$ produced a better pf than training on all, $over$, and $under$.

The bottom of Table 3 shows a summary of these comparisons: $dense$ wins more than all ($5/15$ times vs $1/15$ times), as well as $over$ and $under$ samplings ($5/15$ times vs $0/15$ times). $dense$ also loses less than all ($4/15$ times vs $8/15$ times), as well as $over$ and $under$ samplings ($4/15$ times vs $6/15$ times)

In summary, in the majority cases, training on $dense$ components yielded statistically significantly better detectors than training on all components. Also, these detectors performs better than those learned from over- and under- samplings.

5. CONCLUSION

A previously unexplored feature of the PROMISE defect data sets are the small number of components containing most of the defects. This skewed defect distribution has implications on our data analysis of the PROMISE defect datasets. We have been reporting neg/pos ratios for those data sets as varying from 1 to 249 (see §3.1). This statistic is clearly incorrect since it is a mean value across a highly skewed distribution:

- An entire *project* like CM1 may have a high overall neg/pos ratio.
- This mean ratio does not characterize the neg/pos ratio of individual components.

Mathematically, this skewed distribution is an interesting region of the data since such components have lower neg/pos ratios which, according to the maths of §3.1, means that they could (in theory) yield detectors with high precision and recall.

To test this possibility, we restricted training to just the components with higher than the median number of defective modules. This was compared to training using all the data (which, according

data set	performance measure	all components	dense components	over sampling	under sampling
CM1	precision	0	0	0	0
	recall	+	-	-	-
	pf	-	+	-	-
KC1	precision	0	0	0	0
	recall	0	0	0	0
	pf	0	0	0	0
MC1	precision	0	0	0	0
	recall	-	-	0	0
	pf	-	-	0	0
PC1	precision	-	+	-	-
	recall	-	+	-	-
	pf	-	+	-	-
PC3	precision	0	0	0	0
	recall	-	-	0	0
	pf	-	+	-	-
summary	+	1	5	0	0
	0	6	6	9	9
	-	8	4	6	6

Table 3: Each treatment is assigned one of $\{+, 0, -\}$ depending on if it $won, tied, lost$ in the statistical rankings of Table 1 (based on a Mann-Whitney test at 95% confidence). Note that $dense$ won most often, and lost the least amount of times compared to all other treatments ($4/15$ vs. $8/15$ and $6/15$).

to our understanding of the literature, is the more usual treatment). We found that training via $dense$ sampling is useful for generating better defect prediction models. The improvements are not startlingly better - the recall and precision mean values are close to standard practice (see Table 2). Nevertheless, in the majority cases, training on $dense$ components yielded statistically significantly better detectors than other methods (training on all data, over-sampling, and under-sampling).

Given the simplicity of the technique (just select training data from a subset of the components), we recommended the proposed method whenever it becomes apparent that components have different defect densities. Although for the very first project, we do not know the proportions of defect dense components, the future projects can benefit from defect prediction models learned from the dense components in the first project.

This result has negative and positive implications. On the positive side, we have shown here that the skewed defect distributions within a project can be exploited to generate better defect predictors. On the negative side, it means that there are aspects of the PROMISE data sets that have not been noticed in dozens of prior publications. The PROMISE repository data is used widely in the literature, usually in papers of the form "here is a new learner applied to data sets X,Y,Z from PROMISE". Such papers may be so focused on the *learner* that they miss important aspects of the *data*. In the future, we would recommend visualizations of data *before* applying data miners, lest we miss other important aspects of our data.

Acknowledgments

This work was partially funded by the United States National Science Foundation (CCF-1017263), and the Chinese NSF grant (90718022).

6. REFERENCES

[1] A. B. A. Tosun and R. Kale. Ai-based software defect predictors: Applications and benefits in a case study. In *Proc. Twenty-Second IAAI Conference on Artificial Intelligence*, 2010.

[2] A. B. A. Tosun and B. Turhan. Practical considerations of deploying ai in defect prediction: A case study within the turkish telecommunication industry. In *Proc. PROMISEą́09*, 2009.

[3] C. Andersson and P. Runeson. A replicated quantitative analysis of fault distributions in complex software systems. *IEEE Transactions on Software Engineering*, 33(5):273–286, May 2007.

[4] B. Boehm and P. Papaccio. Understanding and controlling software costs. *IEEE Trans. on Software Engineering*, 14(10):1462–1477, October 1988.

[5] C. Drummond and R. C. Holte. C4.5, class imbalance, and cost sensitivity: why under-sampling beats over-sampling. In *Workshop on Learning from Imbalanced Datasets II*, 2003.

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

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

[8] N. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, pages 797–814, August 2000.

[9] K. Goseva-Popstojanova and M. Hamill. Architecture-based software reliability: Why only a few parameters matter? In *Computer Software and Applications Conference, 2007*, pages 423–430, 2007.

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

[11] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K.-i. Matsumoto. The effects of over and under sampling on fault-prone module detection. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 196–204, 20-21 Sept. 2007.

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

[13] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, May 2008. Available from http://iccle.googlecode.com/svn/trunk/share/pdf/lessmann08.pdf.

[14] N. Leveson. *Safeware: System Safety And Computers*. Addison-Wesley, 1995.

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

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

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

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

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

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

[21] T. Menzies, J. Powell, and M. E. Houle. Fast formal analysis of requirements via 'topoi diagrams'. In *ICSE 2001*, 2001. Available from http://menzies.us/pdf/00fastre.pdf.

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

[23] T. Menzies, B. Turhan, A. Bener, and J. Distefano. Cross- vs within-company defect prediction studies. 2007. Available from http://menzies.us/pdf/07ccwc.pdf.

[24] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang. Implications of ceiling effects in defect predictors. In *PROMISE '08: Proceedings of the 4th international workshop on Predictor models in software engineering*, pages 47–54, New York, NY, USA, 2008. ACM.

[25] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang. Implications of ceiling effects in defect predictors. In *PROMISE '08: Proceedings of the 4th international workshop on Predictor models in software engineering*, pages 47–54, New York, NY, USA, 2008. ACM.

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

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

[28] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: an empirical case study. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 521–530, New York, NY, USA, 2008. ACM.

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

[30] D. Raffo. Personal communication. 2005.

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

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

[33] B. T. B. C. Y. J. A. B. T. Menzies, Z. Milton. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, May 2010.

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

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

[36] E. Weyujer, T. Ostrand, and B. Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5):539–559, October 2008.

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

[38] H. Zhang. On the distribution of software faults. *IEEE Transactions on Software Engineering*, 34(2):301–302, 2008.

[39] H. Zhang. An investigation of the relationships between lines of code and defects. In *Proc. ICSM'09*, Sep 2009.

[40] H. Zhang and X. Zhang. Comments on 'data mining static code attributes to learn defect predictors'. *IEEE Transactions on Software Engineering*, September 2007.