AN EMPIRICAL DESIGN AND CODE METRICS FOR PREDICTION OF SOFTWARE DEFECTS

S. Maheswari
Research Scholar, Dept of Computer Science, Bharathiar University, Coimbatore, TN, India
rkmahes04@gmail.com

Dr. K Chitra
Asst. Professor Dept of Computer Science, Govt Arts College, Melur, Madurai Dt, TN, India
manikandan.chitra@gmail.com

Abstract—Performance bug detection of the software is a required non-functional requirement, appearing in many fields such as complex applications and real time applications development. In this work focused on early detection of performance bugs. The proposed work finds a robust and reliable to implement solution, predicting performance bugs. We built several methods using machine learning algorithms compare with the proposed novel algorithm, used for defect prediction: C4.5 Decision Trees, Naïve Bayes, Bayesian Networks, and Logistic Regression. Our empirical results show that a proposed model, using lines of code changed, MCCABE/HALSTEAD, can be used to predict performance bugs with the accuracy of 0.94 and precision of 0.96. We show that reducing the number of changes provided in the confirmation can reduce the chance of injecting performance bugs. In this approach can help professionals eliminate performance bugs early in the development cycle. Our results are also of interest to theorists, establishing a link between functional errors and performance errors, and clearly showing that the attributes used to predict functional metrics.

Index Terms—Automation, Performance bug, Quality, Software engineering, Software Metrics.

I. Introduction

Data mining is the undertaking of analyzing information from different points of view and consolidating or compressing the information into important and significant data.

Data mining techniques like feature extraction and classification technique have proved very efficient in predicting defects in biological substances, irregularities in clinical data and revealing significant medical facts that raised interest in exploring such avenues for drug therapy and clinical decision making. Feature extraction is the technique of deciding on a subset of effective methods or features for creating robust unsupervised learning models. Classification is a data analysis technique that is used to distinguish important data classes/categories. This work aims at identifying the optimal and minimal set of software defect models that could identify the fault proneness of software in programming systems with increased accuracy. The final performance measures used to evaluate the current approach include the accuracy, sensitivity and specificity.

Software errors are usually not found until the late stages of the development cycle, when it turns expensive to return and fix them. Addressing these bugs is highly essential faultiness which, application developers build a reputation for delivering products or, create life-critical situations when the software is part of larger systems or devices, such as defense equipments or medical treatment plants. Hence, predicting faults in software methods to improve the quality of software utilized in designing defense equipments was the rationale for this present research work.

A few papers on software faults utilizing mining strategies through prediction methods have been proposed in literature. A portion of the papers talked about fault prediction methods, for example, size and complexity measurements, multivariate examination, and multi co-linearity utilizing Bayesian conviction systems. NB is broadly utilized for building classifiers. When building up a defect predictor, the likelihood of each class is computed, given the extracted attributes from a module, utilizing measurements are applicable to anticipating defective modules, for example, Halstead and McCabe ones and so forth., This work developed prediction rules with Naïve Bayes (NB) classifier for fault attributes. This work placed focus on an efficient method to build for the software fault prediction. We summarize this work for three fundamental reasons: The work is creating the dataset from publically available program resources; the precision reported by combined techniques revealed great scope for
development; and design of more accurate fault prediction techniques could significantly improved the quality of software currently utilized in defense framework.

**Related Work**

A number of previous studies have analysed the effect of current contribution behavior on software quality.

Rahman & Devanbu [30] examined the effects of ownership & experience on quality in several open-source projects, using a fine-grained approach based on fix-inducing fragments of code, and report findings similar to those of our paper. However, they operationalize ownership differently, and ownership policies and practices in OSS and commercial software are quite different. Thus the similarity of effect is striking. Furthermore, Rahman & Devanbu do not study the relationship of minor contribution on software dependencies; nor do they consider social network measures.

Weyuker et al. [35] studied the effect of including team size in the prediction model. They use the number of developers for each component but do not check the percentage of work we explain. They found that the accuracy of fault prediction was improved by adding device dimensions to the model in a negligible way. What makes us different is that we check the proportion of each developer’s contribution to the component. In addition, we are not interested in forecasting, but there is a statistically significant relationship between the determination of property and failure.

Similarly, Menely and Williams studied the relationship between some developers and security holes in the Linux kernel [20]. They found that if more than nine developers contribute a source file, the likelihood of a security vulnerability is sixteen times.

New methods such as Extreme Programming (XP) [4] claim ownership of collective code, but there is little empirical evidence or support for such information in reasonably more difficult or large applications. Our research is the first to empirically quantify the effect of code owners (and inexperienced taxpayers) on the overall quality of the code.

Performance bugs are harder to expose during testing phase, because they do not cause fatal symptoms and do not affect the overall outcome [16]. Furthermore, it is difficult to find the root cause of performance bugs (in comparison with other types of bugs); they also need more time to get resolved [5].

The work that is closest to ours is by Jin et al. [2], which studied a set of 109 real-world performance bugs. They studied the bugs’ lifetime from inception to fix, their root causes and introduction mechanisms, in order to create rule-based detectors. We used a comparable approach to create detectors (referred to as “patterns” in our study) with the aim to identify similar performance bugs. They studied software written in Java, C, C++, and JavaScript; ours was written in MATLAB. Moreover, we focused on finding an efficient method to automatically detect performance bugs based on data extracted through the use of patterns (197 real-world performance bugs in our case).

Summarizing, Jin et al. [2] focused on analyzing characteristics of performance bugs, while in this study, our focus is on understanding the contribution of each source code attribute, as extracted from the source code repository, to the predicting power of the several machine learning algorithms that we used. Lastly, our general approach and the methodology for creating the prediction model were meant to be easily reproducible in the future. Therefore, our work is complementary.

Automated detection of performance bugs have been implemented in the past [3], [18]. However, the authors used dynamic analysis tools, namely execution traces and historical performance data, to detect slowly executing code. In addition, dynamic analysis tools often require dedicated testing environment to get accurate performance readings. We, on the other hand, are leveraging code attributes extracted from source code repository and automatically detecting performance bugs before the code executes (and reaches production environment).

Thus, our work is complementary. Static analysis tools can also eliminate performance bugs [20]. However, this approach requires knowledge of pattern template, while ours does not (as we omit pattern type info in our models), hence the complementarity of our work. There exist formal and thorough coding standards for RTS. However, the standards are, typically, language specific (e.g., C [21] and C++ [22]). To the best of our knowledge, no thorough and formal RTS coding standard exists for MATLAB.

Finally, since in our study we relate performance with the real-time nature of the system, one could argue that programming language might not be
appropriate for such a case. Although this is a problem that falls out of the scope of this study, we understand its importance and validity. This programming language (with a proper runtime [22]) is used in well-known large-scale RTS.

II. METHODOLOGY

A. Data Preparation

We eliminated all files that did not go into metrics extraction base, namely: readme files, testing scripts, and help files. Additionally, we removed a minor 0.2% (9 out of 4623 unique tuples) of "commit id – file name" records related to source files. These records were outliers, extreme cases. For example, we excluded source files that were moved or removed. To be more specific, the version control system by default identifies directory changes/refactorings as complete removals of the files themselves. Therefore, whenever a file is moved one or more levels up or down in the directory structure, we noticed abnormal numbers of lines added and/or removed. In some of these cases (and especially in directories including large files) more than 10,000 lines were added or removed on a single commit. The cleansing described above, resulted in a more precise model creation, mainly because of the removal of entries that will not be seen on future commits [31].

B. Metrics Extraction:

The features are extracted based on McCabe and Halstead measures. The following table defines the defect detectors assessments.

<table>
<thead>
<tr>
<th>classifier predicts no defects</th>
<th>No</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>classifier predicts Some defects</td>
<td>tp</td>
<td>fn</td>
</tr>
</tbody>
</table>

Accuracy = (tp+tn)/(tp+tn+fp+fn)

Probability of detection (recall) = tn/(tp+tn)

Precision = tn/(fp+tn)

Effort = (fp.LOC + tn.LOC)/(Total LOC)

The Four McCabe software [19] metrics are: Essential complexity, cyclomatic complexity, design complexity and LOC.

D. Attribute Information

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loc</td>
<td>McCabe’s line count of code</td>
</tr>
<tr>
<td>iv(g)</td>
<td>McCabe “essential complexity”</td>
</tr>
<tr>
<td>v</td>
<td>McCabe “cyclomatic complexity”</td>
</tr>
<tr>
<td>l</td>
<td>McCabe “volume”</td>
</tr>
<tr>
<td>D</td>
<td>McCabe “difficulty”</td>
</tr>
<tr>
<td>i</td>
<td>McCabe “intelligence”</td>
</tr>
<tr>
<td>e</td>
<td>McCabe “effort”</td>
</tr>
<tr>
<td>b</td>
<td>Halstead</td>
</tr>
<tr>
<td>T</td>
<td>Halstead’s time estimator</td>
</tr>
<tr>
<td>IOCode</td>
<td>Halstead’s line count</td>
</tr>
<tr>
<td>I0Comment</td>
<td>Halstead’s count of lines of comments</td>
</tr>
<tr>
<td>I0Blank</td>
<td>Halstead’s count of blank lines</td>
</tr>
<tr>
<td>I0CodeAndComment</td>
<td>Numeric</td>
</tr>
<tr>
<td>uniq_Op</td>
<td>unique operators</td>
</tr>
<tr>
<td>uniq_Opnd</td>
<td>unique operands</td>
</tr>
<tr>
<td>total_Op</td>
<td>total operators</td>
</tr>
</tbody>
</table>

E. Rule Mining

According to the recommended interval, we define simple rules for each metric. If the module's metrics are not within the specified time interval (this means manually checking the module), these rules are activated. It shows 12 basic rules and corresponding indicators, as well as 2 derived rules. The first derivative rule, rule 13, defines the separation of 12 basic rules. If you trigger some basic rules, that is the rule 13 trigger. The reason is that the corresponding comments and the intervals associated with the Halstead metrics do not conform to the characteristics of Turkcell coding. One solution is to define new ranges for these metrics. However, this is not possible because there is no defect data to derive these detection activation intervals. In order to overcome this problem, we defined rule 14, if all the basic rules were activated, but Halsted fire. This reduces the frequency of extraction of rules. However, Rule 14 states that 9556 modules corresponding to 461.655 LOC must be checked to detect possible defects. It is not practical to check 45% of the total LOC. On the other hand, it will be more effective to show learning-based models.

C. Performance Bug Prediction

The methodology proposed in this work for software defect prediction comprises of two phases: Data processing phase; and validation phase. This work involves data pre-processing, metric extraction, rule mining, weighted factor prediction and classification of the data information. The next phase comprises of validating the performance of the classifiers and its rules investigated in this study using cross-validation technique and ranking the performance of the classifiers based on the classification accuracy and recall ratio. The methodology followed in this research work is detailed in next section.
The automated prediction of software fault is depending on two criteria. One is sensitivity of the prediction and the other one is specificity. The both were measured in confusion matrix and calculated as:

- **Sensitivity** = \( \frac{\text{number of true positives}}{\text{number of true positives} + \text{number of false negatives}} \)
- **Specificity** = \( \frac{\text{number of true negatives}}{\text{number of true negatives} + \text{number of false positives}} \)

The above two tests are helpful to know the accurate prediction.

### III. EXPERIMENTAL RESULTS

#### A. Removed Constant Attributes

An attribute which has a constant/fixed value throughout all instances is easily identifiable as it will have a variance of zero. Such attributes do not have any information with which to discern modules apart, and are at best a waste of classifier resources. This data set has 5 constant attributes out of a total of 22, thus 45 percent of available recorded values contain no information upon which to data mine.

#### B. Removed Redundant Attributes

In addition to constant attributes, repeated attributes occur where two attributes have identical values for each instance. This effectively results in a single attribute being overrepresented. Amongst the data sets there is only one pair of repeated attributes, namely the 'number of lines' and 'loc total' attributes in data set. For this stage we removed one of the attributes so that the values were only being represented once. We chose to keep the 'loc total' attribute label as this is common usage of our data set.

#### Replaced Missing Values

Values may or may not be problematic for machine learners depending on the classification method used. 19 records in the data set contain missing values, but all in the same single attribute: 'decision density'. This attribute is defined as 'condition count' divided by 'decision count', and for each missing value both these base attributes have a value of zero.

#### D. Results

In this present work, the software metric 21 are form McCabe's and Halstead metrics and one goal metric were taken to measure. Using Matlab tool, dataset were applied to classifiers Naive Bayes and proposed algorithm algorithms. The dataset has taken combination in basis of structure and object oriented. Most of the source codes were written in C and C++ language. Study had compared average of accuracy (the values taken in to table as from 0 to 1), true positive rate, false positive rate, sensitivity and specificity. Accuracy calculated based on number of instances classified correctly. Based on these analysis results, proposed method was suitable for both large and small datasets.

The following tables have given the instances correctly classified and in-accurately classified with total number instances in dataset using different classifiers. Also it provides best classifier by highlighted based on sensitivity and specificity value.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Accurately Classified Instances</th>
<th>In-accurately Classified Instances</th>
<th>Total Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training Set</td>
<td>3792</td>
<td>222</td>
<td>4014</td>
</tr>
<tr>
<td>Testing Set</td>
<td>4519</td>
<td>481</td>
<td>5000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Sensitivity</th>
<th>Specificity</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training Set</td>
<td>0.982</td>
<td>0.978</td>
<td>94.36</td>
</tr>
<tr>
<td>Testing Set</td>
<td>0.996</td>
<td>0.999</td>
<td>97.81</td>
</tr>
</tbody>
</table>

Performance Evaluation of Proposed Work
IV. CONCLUSION AND FURTHER ENHANCEMENT

In this proposed work predict performance bug modules in a large software system. This work have performed an average case analysis for the 25 projects in order to determine the characteristics of the implemented code base and observed that there were contradicting measurements with the company objectives. Our initial data analysis showed that a simple rule-based model based on the recommendation criteria in the static code attributes estimated the defect rate of the code to be checked. Taking into account the scope of the system, this is an unrealistic result. Therefore, we established learning-based defect predictors and performed additional analysis. Due to the lack of defect data at the local module level, we use synthetic data to learn predictors of defects. The initial analysis confirmed the mean defect rate for all projects. Although simple rule-based modules require code inspection, the learning-based model suggests that we only need to check 6% of the code. This is because rule-based models are biased toward more complex and larger modules, while learning-based models predict that smaller modules contain most of the defects. The results of our second analysis used frame-adjusted data. The external validation of the data did not change the median probability of detection and significantly reduced the median probability of false positives. The second analysis further improved the estimate and suggested that only 93% of the code could be detected and 93% of defects could be detected.

Our future work is to gather local software level defects in order to build predictors for this large telecommunication system within the company. Further to use block level code to metrics to predict both semantic and performance defects between successive software releases.

REFERENCES
