DETECTING SOFTWARE BUGS IN SOURCE CODE USING DATA MINING APPROACH

Pravin A.[1], Srinivasan S.[2]

[1]Research Scholar, Sathyabama University, Chennai, India
[2]Director Affiliation, Anna University of Technology, Madurai, India
Email: [1]pravin_ane@rediffmail.com

Abstract

In a large software system knowing which files are most likely to be fault-prone is valuable information for project managers. They can use such information in prioritizing software testing and allocating resources accordingly. However, our experience shows that it is difficult to collect and analyze fine grained test defects in a large and complex software system. On the other hand, previous research has shown that companies can safely use cross-company data with nearest neighbor sampling to predict their defects in case they are unable to collect local data. In this paper the discussion is done to predict software bugs in the source code by using data mining approach by training the models that are perfect and that are defect. In our experiments we used ranking method (RM) as well as nearest neighbor sampling for constructing method level defect predictors. Our results suggest that, for the analyzed projects, at least 70% of the defects can be detected by inspecting only (i) 4% of the code using a Naïve model, (ii) 6% of the code using RM framework.

Key words: Testing, Defect predictors, Software bugs, Training, Ranking method

I. INTRODUCTION

User configurable software — software that can be customized through a set of options by the user — is becoming increasingly prevalent. Often these options are read at program start-up or can be changed at run-time, meaning that configurations can be modified by users' on-the-fly. A single user configurable software application can often be instantiated in an enormous number of ways. From a testing perspective, each configuration may appear largely similar, but the underlying execution of code for the same set of test cases may differ widely across configurations [10]. This increases the burden on software engineers, who must consider not just which inputs to utilize in testing, but also which configurations.

Software testing is one of the most critical and costly phases in software development. Project managers need to know "when to stop testing?" and "which parts of the code to test?". The answers to these questions would directly affect defect rates and product quality as well as resource allocation (i.e. experience of test staff, how many people to allocate for testing) and the cost. As the size and complexity of software increases, manual inspection of software becomes a harder task. In this context, defect predictors have been effective secondary tools to help test teams locate potential defects accurately [7]. These tools are built using historical defect databases and are expected to generalize the statistical patterns for unseen projects. Thus, collecting defect data from past projects is the key challenge for constructing such predictors.

In this paper, we share our experience for building defect predictors in a large system. In order to improve the code quality, a prediction engine will be used to predict the code defects before it enters into a testing phase. The underlying system is standard 3-tier architecture, which has a presentation, application and data layers. Our analysis focuses on the presentation and application layers. However, the content in these layers cannot be separated as a distinct project. Also a defect prediction model which is used on a static code attributes like lines of code etc.

Some researchers have argued against the use of static code attributes claiming that their information content is very limited [1]. However, static code attributes are easy to collect, interpret and many recent research have successfully used them to build defect predictors [2]. Therefore, we decided to use module level cross-company data to predict defects in software projects. Specifically, we have used module level defect information from projects to train defect predictors and then to obtain predictions for projects. Previous researches have shown that cross-company data gives

stable results using nearest neighbor sampling techniques further improves the prediction performance when used [10].

Our experiment results with data on projects, estimate that we can detect 70% of the defects with an investigation effort. While nearest neighbor algorithm improves the detection rate of predictors built on projects data, false alarm rates remain high. In order to decrease false alarm rates, we included ranking method (RM) framework in our analysis based on our previous research. Using of RM framework improves the estimated results with effort decreased from 6% to 3%. The rest of the paper is organized as follows: In section 2 we briefly review the related works; in Section 3 we explain the project data. Section 4 explains our rule-based analysis is discussed in Section 5. The last section gives conclusion.

II. RELATED WORKS

Software failures greatly reduce system reliability. As software becomes more and more complex, there is an urgent need to explore more effective software testing and debugging tools and software engineering methods to minimize the number of bugs that escape into production runs. According to a report from NIST in 2002, improvements in software testing could eliminate about $22.2 billions of business loss in US caused by software errors annually. Since some bugs still slip through even the strictest testing, system designers also need to provide fault tolerant mechanisms to recover from these inevitable software failures.

Fault mining can be performed on any Java program representation. It only requires some notion of interesting program events and a partial order among these events. Different analysis techniques can be used to generate events that can be the basis for invariant inference. Our contribution in this research is to analyze a large-scale industrial system at the module level. To accomplish this, we use a state-of-the-art cross-company defect predictor. We further demonstrate its practical use by improving its performance with nearest neighbor sampling technique. We also use a predictor that not only models intra module complexities, but also inter module connections. We used ranking method (RM) framework and show that combining inter and intra module metrics not only increases the performance of defect predictors but also decreases the required testing effort for manual inspection of the source code.

Also the importance of software is increasing in scientific research and our daily life. Meanwhile, the cost and consequences of software failure caused by software bugs become more and more serious. This research emphasizes a standard process for data mining based software debugging. This proposed process provides guidelines for software testing engineers and researchers on how to apply data mining techniques and software testing theory on real life software testing projects. Data mining based software debugging projects is a five step process: Establish the software testing project; data collecting, cleaning and transformation; select, train and verify the data mining models; classify, locate and describe the software bug found in previous steps; and deploy the knowledge gained into real life software testing project.

III. PROPOSED METHODOLOGY

The Fig. 1 illustrates a general architecture of the Ranking method Prediction System. Data is captured periodically from the software vendors' Online Transaction Processing (OLTP) systems which collect defect report and escalation information. Data may be captured weekly and stored indefinitely in a data mart. While the information in the OLTP systems continues to change from moment to moment, the information in the data mart remain constant so as to provide the historical data required for training predictive models in Prediction engine. All the historical data may be pre-processed before being fed into data-mining rules. Derived fields, historical information, and statistics are added to yield the set of available input fields. These fields or attributes may include data fields which come directly from the defect report tracking system and derived fields obtained from the data and other sources. The next step is to train and validate predictive models. Once one or more satisfactory models have been found, the most appropriate model may be selected and run against the most recent snapshot of defect report data. The predicted escalations may be then reported to the product group for evaluation and proactive resolution. The product group may provide feedback to allow for ongoing improvement of the overall escalation prediction and prevention effort for that two mechanism is been implemented one is the average case analysis and the other is the rule based approach. Here the test projects

are implemented in Java and are gathered with 30 static code metrics from each. In total, there are approximately 5 modules spanning with 3,000 lines of code. All projects are from presentation and application layers. The characteristic of projects with dataset has 22 static code attributes. In our analysis also we have used only the common attributes that are available in different projects

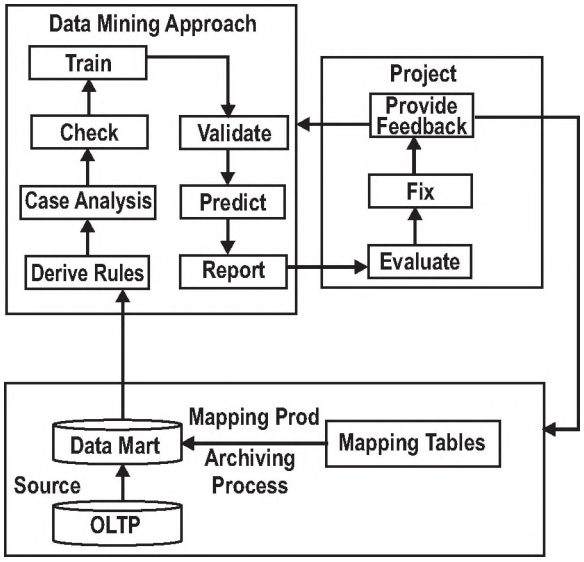

Fig. 1. General architecture of the Ranking method Prediction System

3.1 Average-case analysis

The average values of static code metrics collected from the different projects are used in this paper. It also shows the recommended intervals (i.e. minimum and maximum values) based on statistics from projects, when applicable. The system explains that the developers do not write comments throughout the source code. In case of the Low number of operands and operators which are indicated with small, modular methods. While the latter observation can be interpreted as an objective to decrease maintenance effort, the former contradicts such as an objective which requires action. Note that, this shows how a simple average case analysis can point out conceptual problems in company objectives as long as measurement is performed.

3.2 Rule-based Approach

Based on the recommended intervals of the simple rules for each metric can be defined. These rules fire, if a module's metric is not in the specified interval, indicating the manual inspection of the module. The basic rules and corresponding metrics, along with derived rules are identified. The first derived rule, Rule 1 is defined as a disjunction among other basic rules. That is Rule 1 fires if any basic rule fires. Frequently rules that are caused are defined as rule 1 which fires all the time. A solution would be to define new intervals for these metrics; however, this is not possible since there are no defect data to derive these inspection-triggering intervals. In order to overcome this problem we have defined Rule 2 that fires if all basic rules are been fired. This reduces the firing frequency of the disjunction rule. However, Rule 2 states that 6 modules (14%) corresponding to be inspected in order to detect potential defects. On the other hand, learning based model will be shown to be far more effective. Also we have designed two types of analysis using the learning based model:

- Rule #1 uses the cross-company predictor with k-nearest neighbor sampling for predicting fault-prone modules.
- Rule #2 combines inter and intra module metrics, in other words incorporate RM framework into static code attributes and than apply the model of Analysis

IV. RESULTS & DISCUSSION

In this analysis we used the data miner that achieves significantly better results than many other algorithms for defect prediction. We selected a random 90% subset of data to train the model. From this subset, we have selected similar projects that are similar to trained data in terms of Euclidean distance in the dimensional metric space. The nearest neighbors in the random subset are used to train a predictor, which then made predictions on the data. We repeated this procedure 20 times and raised a flag for modules that are estimated as defective at least in 10 trials. The estimated defect rate is 15% that is consistent with the rule-based model's estimation.

Table 1. Explains about the Average-case analysis of the projects

Project	Language	Estimated Defect Rate (%)	Inspection (%)
PRJ1	Java	0.05	0.04
PRJ2	Java	0.08	0.05
PRJ3	Java	0.07	0.06
PRJ4	Java	0.08	0.06
PRJ5	Java	0.05	0.08

In the results it explains about the Table 1 which defines the Average-case analysis of the projects. The average values of static code metrics collected from the different projects are used in this result. Table 2 explains the Rule-based analysis. In each and every rule corresponds to the recommended interval for the corresponding metric and the % of defects that is been reduced.

Table 2. Rule-based analysis for each rule corresponds to the recommended interval

Rule	Metric	% Defects
Rule1	Intelligent Content	0.05
Rule2	Maximum Nesting Depth	0.02
Rule3	Volume	0.05
Rule4	Total Operators	0.06
Rule5	Time	0.05

V. CONCLUSION

In this paper the investigation is done on predicting the Software bugs in software source code using data mining approach. We have also performed analysis on different projects in order to determine the characteristics of implementing code and observations that are been conducted in order to measure the company objectives. Specifically, the software modules were written using relatively low number of operands and operators to increase modularity and to decrease maintenance effort. However, we have also observed that the code base was purely commented, which makes maintenance a difficult task. Our initial data analysis revealed that a simple rule-based model based on recommended standards on static code attributes estimates a defect rate of 14% and requires 46% of the code to be inspected. This is an impractical outcome considering the scope of the system. Thus, we have constructed learning based defect predictors and performed further analysis. We have used data to learn defect predictors, due to lack of local module level defect data. This is from the fact that rule-based model has a bias towards more complex and larger modules, whereas learning based model predicts that smaller modules contain most of the defects.

REFERENCES

[1] Tao Xie, Suresh Thummalapenta, David lo, Chao Liu,"Data Mining for Software Engineering", IEEE Computer, August 2009

[2] Hamid Abdul BAsit, Stan Jarzabek, " A Data Mining approach for detecting higher-level clones in Software", IEEE Transactions on Software Engineering, Vol. 35, No. 4, July/August 2009

[3] Ivano Malavelta, Henry Muccini, Patrizio Pellicciona, Damien Andrew Tamburri, "Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies", IEEE Transactions on Software Engineering, Vol. 36, No. 4, January/February 2010

[4] Tao Xie, Jain Pei, Ahmed E Hassen, "Mining Software Engineering Data", IEEE 29 th International Conference on Software Engineering ICSE 07.

[5] Francisco P.Romero, Jose A.Olivas, MArcele Genero, Mario Piattini, "Automatic Extraction of the main terminology used in Empirical Software Engineering through Text Mining Techniques" ACM ESEM 08

[6] Mohammed J Zaki, Christopher D Carothes, Boleslan K Szymaski, "VOGUE: A Variable Order hidden Markov Model with duration based on Frequent Sequence Mining", ACM Transactions on Knowledge Discovery from Data, Vol. 4 No.1, Article 5, January 2010.

[7] Francine Bermas, "Got Data? A guide to data preservation in the Information Age", Communications of the ACM, December 2008

[8] Nizar R Mabroukeh, Christe I Ezeite," Using Domain Ontology for Semantic Web Usage Mining and Next Page Prediction", ACM CIKM 08

[9] Tim Menzein, Gary D Boettiecher, "Smarter Software Engineering: Practical Data Mining Approaches", IEEE/NASA 27 Th Annual Software Engineering Workshop 2002.

[10] Josh Eno, Craig W Thompson," Generating Synthetic Data to Match Data Mining Patterns", IEEE Internet Computing May/June 2008

[11] O.Maqbool, A Karim, H.A.Babri, Misarwar, "Reverse Engineering using Association Rules", IEEE INMIC 2004, pp. 389 -395.

[12] Gang Kou Yipeng, "A Standard for Data Mining based Software Debugging", IEEE 4 Th International Conference on Networked Computing and advanced Information Management.

[13] Qi Wang, Bo yo, Jie Zhu, "Extract Rules from Software Engineering Quality Prediction Model based on Neural Networks", ICTAI 2004.

[14] Ngoavel Moha, Yann-Gael Gueheneu, Laurence Duchien, Anne-Fran Coisele Mew, "DÉCOR – A Method for the Specification and Detection of Code and Design Smells", IEEE Transactions on Software Engineering, Vol. 36, No. 4, January/February 2010

A.Pravin: received the B.E degree in Computer Science & Engineering from Bharath Niketan Engineering College, Madurai Kamaraj University, Madurai, India in 2003 and M.E degree in Computer Science & Engineering from Sathyabama University, Chennai, India in 2005 .Where he is currently working towards the Ph.D degree in Computer Science & Engineering at Sathyabama University, Chennai, India.

He works currently as an Assistant Professor for the Department of Computer Science and Engineering at SRR Engineering College, chennai and he has more than 7 Years of teaching experience.He has participated and presented many Research Papers in International and National Conferences. His area of interests includes Software Engineering, Data mining and Data warehouse.