Survey on Bug Prediction Based on Features Selection

T. Vishnu Vardhan Reddy[1], *N. Sambasiva Rao*[2], *Ch. Srikanth*[3]
[1] Vardhaman College of Engineering, Hyderabad, M.Tech Computer Science and Engineering,,
vishnu.talla34@gmail.com

[2]Vardhaman College of Engineering, Hyderabad, Principal of Vardhaman College of engineering,
snandam@gmail.com
[3]Vardhaman College of Engineering, Hyderabad,M.Tech Computer Science and Engineering,
srikanthch.tillu43@gmail.com

Abstract: The automated detection of faulty modules contained by software systems might lead to reduced development expenses and additional reliable software. In this effort design and development metrics has been used as features to predict defects in given software module using SVM classifier. A rigorous progression of pre-processing steps were applied to the data preceding to categorization, including the complementary of in cooperation classes (faulty or otherwise) and the elimination of a large numeral of repeating instances. The Support Vector Machine in this trial yields a standard accuracy that miles ahead over existing defect prediction models on previously unseen data.

Keywords: Defect Prediction, Feature Selection, F-Measure, SVM.

1. Introduction

Data mining techniques from the field of artificial intelligence now make it probable to forecast software defects; undesired outputs or personal property produced by software, from static code metrics. Views toward the value of using such metrics for imperfection prediction are as varied within the software manufacturing community as those toward the value of static code metrics. However, the conclusions suggest that such predictors are helpful, as on the information used in this learn they predict defective modules with an standard accuracy that is miles ahead of existing defect prediction models. In this regard the consequences observed during this training and testing processes are purely proportional to the properties of the features such as type and count of the selected. Hence it indicates the research scope in the direction of developing scalable feature extraction utilization models.

Shivkumar Shivaji et al attempted to reducing features to Improve Code Change Based Bug Prediction, they attempted to investigate multiple feature selection techniques that are generally applicable to classification-based bug prediction methods. The techniques discard less important features until optimal classification performance is reached. The total number of features used for training is substantially reduced, often to less than 10% of the original. The performance of Na¨ıve Bayes and Support Vector Machine (SVM) classifiers when using this technique is characterized on eleven software projects. Na¨ıve Bayes using feature selection provides significant improvement in buggy F-measure (21% improvement) over prior change classification bug prediction results (by the second and fourth authors [10]). The SVM's improvement in buggy F-measure is 9%. Interestingly, an analysis of performance for varying numbers of features shows that strong performance is achieved at even 1% of the original number of features.

2. Software Quality Models

2.1 Purposes, Usage Scenarios and Requirements

There is a huge amount of work on various forms of quality models. However, comprehensive overviews and classifications are scarce. A first, broad classification of what he called "quality evaluation models" was proposed by Tian. He distinguishes between the specificness levels generalized and product-specific. These classes are further partitioned along unclear dimensions. For example, he distinguishes segmented models for different industry segments from dynamic models that provide quality trends. Two of the authors built on Tian's work and introduced further dimension. Wagner discussed in the dimensions purpose, quality view, specificness and measurement where the purposes are construction, assessment and prediction. This was further extended by Wagner and Deissenboeck with

the dimensions phase and technique. A thorough discussion of critique, usage scenarios and requirements along these dimensions is not provided in any of these contributions.
An impressive development of quality models has taken place over the last decades. These efforts have resulted in many achievements in research and practice. As an example, take a look at the field of software reliability engineering that performed a wide as well as deep investigation of reliability growth models. In some contexts these models are applied successfully in practice. The developments in quality definition models even led to the standardization in ISO 9126 that is well known and serves as the basis for many quality management approaches.

However, the whole field of software quality models is diverse and fuzzy. There are large differences between many models that are called “quality models”. Moreover, despite the achievements made, there are still open problems, especially in the adoption in practice. Because of this, current quality models are subject to a variety of points of criticism that have to be acted on.

We provide a comprehensive definition of a quality model based on the purpose the model has. Using this tripartion in definition models, assessment models and prediction models (DAP), we summarized the existing critique and collected a unique collection of usage scenarios of quality models. From this, we derived a comprehensive set of requirements, again ordered in terms of the DAP classification, that can be used in two contexts: (1) evaluation of existing models in a specific context or (2) further developments and improvements of software quality models.

2.2. An Empirical Examination of 100 mature Open Source Projects

Starting with Eric Raymond's ground-breaking work, "The Cathedral and the Bazaar”, open-source software (OSS) has commonly been regarded as work produced by a community of developers. Ghosh's cooking pot markets, similarly, point to a communal product development system. Certainly, this is a good label for some OSS products that have been featured prominently in the news. For instance, Moon and Sproull point out that by July 2000, about 350 contributors to LINUX were acknowledged in a credit list in the source code of the kernel.
However, my goal in this paper is to ask if the community-based model of product development holds as a general descriptor of the average OSS product. I systematically look at the actual number of developers involved in the production of one hundred mature OSS products. What I found is more consistent with the lone developer (or cave) model of production rather than a community model (with a few glaring exceptions, of course).

2.3. Extracting Facts from Open Source Software

Open source software systems are becoming ever more important these days. Many large companies are investing in open source projects and many of them are also using such software in their own work.

In this paper we describe a framework called Columbus with which we are able to calculate the object oriented metrics validated for fault-proneness detection from the source code of the well-known open source web and e-mail suite called Mozilla. We then compare our results with those presented. One of our aims was to supplement their work with metrics obtained from a real-size software system. We also compare the metrics of the seven most recent versions of Mozilla (1.0–1.6), which covers over one and a half years of development, to see how the predicted fault-proneness of the software system changed during its development.
The Columbus framework has been further improved recently with a compiler wrapping technology that allows us to automatically analyze and extract information from practically any software system that compiles with GCC on the GNU/Linux platform (the idea is applicable as well to other compilers and operating systems). What is more, we can do this without modifying any of the source code or make files. We describe this technique in detail later in this paper.

This paper makes three key contributions: (1) we presented a method and toolset with which facts can be automatically extracted from real-size software; (2) using the collected facts we calculated object oriented metrics and supplemented a previous work [1] with measurements made on the real-world software Mozilla; and (3) using the calculated metrics we studied how Mozilla’s predicted fault proneness has changed over seven versions covering one and a half years of development.

When checking the seven versions of Mozilla we found that the predicted fault-proneness of the software as a whole decreased slightly but we also found example classes with an increased probability of fault-proneness.

In the future we plan to validate the object oriented metrics and hypotheses presented (and here as well) on Mozilla for fault-proneness detection using the reported faults which are available from the Bugzilla database. We also plan to scan Mozilla (and also other open source systems) regularly for fault-proneness and make these results publicly available.

2.4. Assessing the Health of Open Source Communities

The computing world lauds many Free/Libre and Open Source Software offerings for both their reliability and features. Successful projects such as the Apache http Web server and Linux operating system kernel have made FLOSS a viable option for many commercial organizations.
While FLOSS code is easy to access, understanding the communities that build and support the software can be difficult. Despite accusations from threatened proprietary vendors, few continue to believe that open source programmers are all amateur teenaged hackers working alone in their bedrooms. But neither are they all part of robust, well-known communities like those behind Apache and Linux.
If you, as an IT professional, are going to rely on or recommend FLOSS, or contribute yourself, you should first research the community of developers, leaders, and active users behind the software to decide whether it's healthy and suitable for your needs.

3. Life cycle and motivation

Understanding a project's life cycle and its participants' motivations is useful for understanding why a FLOSS community is important to a project's success.

The founders' good ideas expressed in working code facilitate a successful project's second phase: a "creative explosion" in which the product, now public, develops quickly, gathering features and capabilities that in turn attract additional developers and users. What Perl founder Larry Wall calls "learning in public" can be an exhilarating, if difficult, time early in project's life cycle.

Many researchers have examined FLOSS participants' motivations, but these studies often focused on small samples of atypical projects.
• Intellectual engagement;
• Knowledge sharing;
• The product itself; and
• Ideology, reputation, and community obligation.

Although reputation is low on the list, its importance does rise with the length of participant involvement. Projects that have an atmosphere of exploration and intellectual engagement, especially early in their life, are most likely to attract the active user community needed for future success. Also important in attracting good developers is a code base that solves a real need.

Even very successful open source projects often lack detailed roadmaps, explicit work assignments, or feature request prioritizations. A key is to ensure the efficient use of a fixed pool of resources, but FLOSS projects don't face such fixed pools, either in the number of participants or in the amount of time each one can devote.
Therefore, organizing for fun can be more important than organizing for efficiency. In fact, duplication of effort could be a positive sign that the project can attract resources and is in a position to choose the best contributions. On the other hand, a formalized system for prioritizing security issues clearly benefits some applications. And if the processes are discussed, it's important that these discussions regularly end in action that lets people get back to work rather than in an exhausted stalemate.
If your assessment leaves you feeling that the community isn't right for you, be prepared to consider alternatives, no matter how attractive the code is. Trying to change an existing community is likely to end in frustration and undermine the reasons you chose FLOSS in the first place. However, while a rejection of your enthusiastic contributions can seem dictatorial and rude, it can also demonstrate a longterm, cohesive vision—a FLOSS community at its best.

4. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction

Many researchers have sought to analyze the connection between object-oriented metrics and code quality. A summary of the empirical literature was given by Subramanian and Krishnan. It was written in Java and consisted of 123 classes and around 34,000 lines of code. First, they examined the correlation among the metrics and found four highly correlated subsets. Then, they used univariate analysis to find out which metrics could detect faults and which could not. They found that Subramanian and Krishnan chose a relatively large-commerce application developed in C++ and Java and collected metrics from 405 C++ and 301 Java classes. They examined the effect of the size along with the WMC, CBO, and DIT values on the faults by using multivariate regression analysis. Besides validating the usefulness of metrics, they compared the applicability of the metrics indifferent languages; thus, they validated their hypotheses for C++ and Java classes separately. They concluded that the size was a good predictor in both languages, but WMC and CBO could be validated only for C++.

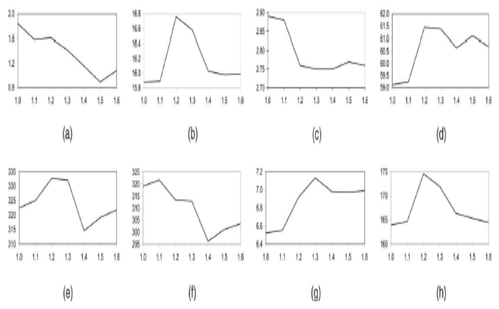

Figure.1 Changes in the mean value of the metrics over seven versions of Mozilla.

The main contributions of this paper are the following:

1. We presented a method and toolset with which metrics (and also other data) can be automatically calculated from the C++ source code of real-size software (the toolset is freely available for academic purposes and can be downloaded from the homepage of Front End ART).
2. By processing the Bugzilla database, we associated the bugs with classes found in the source code.
3. We employed statistical (logical and linear regression) and machine learning (decision tree and neural network) methods to assess the applicability of the well-known object-oriented metrics to predict the number of bugs in classes.
4. Using the calculated metrics, we studied how Mozilla’s predicted fault-proneness changed over seven versions covering one and a half years of development.

Our main observations are the following:

1. All four assessment methods employed yielded very similar results.
2. The CBO metric seems to be the best in predicting the fault-proneness of classes.
3. The LOC metric performed fairly well and, because it can be easily calculated, it seems to be suitable for quick fault prediction. However, for fine-grained analyses, the multivariate models perform much better (e.g., in the case of linear regression, theR2 value of LOC was 0.34, while the R2 value of the multivariate model was 0.43).
4. The correctness of the LCOM metric is good, but its completeness value is low.
5. The DIT metric is untrustworthy, and NOC cannot be used at all for fault-proneness prediction.
6. In Mozilla version 1.2, we noticed significant changes in the metrics—which we believe reflects fall in quality—but it slowly restored in the later versions.

The precision of our models is not yet satisfactory, so we have to analyze what the reasons are for the most common errors in the models and examine whether other metrics can improve them. We will also check whether multiple models perform better when combined in some way (e.g., using voting majority).

We are currently performing the same kind of investigation on other large software systems (OpenOffice.org and two industrial systems). In the future, we plan to scan Mozilla (and other open source systems) regularly for fault proneness and make these results publicly available for the software developer community.

4.2 Classifying Software Changes: Clean or Buggy

The goal of change classification is to use a machine learning classifier to predict bugs in changes. As a result, related work exists in the area of bug prediction, as well as algorithms for source code clustering and for text classification.

4.3 Predicting Buggy and High Risk Modules

There is a rich literature for bug detection and prediction. Existing work falls into one of three categories, depending on the goal of the work. The goal of some work is to identify a problematic module list by analyzing software quality metrics or a project's change history. This identifies those modules that are most likely to contain latent bugs, butTSE-0061-0306 5provides no insight into how many faults may be in each module. Other efforts address this problem, predicting the bug density of each module using its software change history. Work that computes a problematic module list or that determines a fault density are good for determining where to focus quality assurance efforts, but do not provide specific guidance on where, exactly, in the source code to find the latent bugs. In contrast, efforts that detect faults by analyzing source code using static or dynamic analysis techniques can identify specific kinds of bugs in software, though generally with high rates of false positives. Common techniques include type checking, deadlock detection, and pattern recognition.

Hassan and Holt use a caching algorithm to compute the set of fault prone modules, called the top-ten list. They use four factors to determine this list: software that was most frequently modified, most recently modified, most frequently fixed, and most recently fixed. Kim et al. proposed the bug cache algorithm to predict future faults based on previous fault localities .Ostrand et al. identified the top 20% of problematic files in a project. Using future fault predictors and a negative binomial linear regression model, they predict the fault density of each file.

Pan et al. use metrics computed over software slice data in conjunction with machine learning algorithms to find bug-prone software files or functions. Their approach tries to find faults in the whole code, while our approach focuses on file changes.

4.4 Mining Buggy Patterns

One thread of research attempts to find buggy or clean code patterns in the history of development of a software project. Williams and Hollingsworth use project histories to improve existing bug finding tools .Using a return value without first checking its validity may be a latent bug. In practice, this approach leads to many false positives, as typical code has many locations where return 8are used without checks. To remove the false positives, Williams and Hollingsworth use project histories to determine which kinds of function return values must be checked. For example, if the return value of foo was always verified in the previous project history, but was not verified in the current source code, it is very suspicious. Livsh its and Zimmermann combined software repository mining and dynamic analysis to discover common use patterns and code patterns that are likely errors in Java applications. Similarly, PR-Miner mines common call sequences from a code snapshot, and then marks all non-common call patterns as potential bugs.

These approaches are similar to change classification, since they use project specific patterns to determine latent software bugs. However, the mining is limited to specific patterns such as return types or call sequences, and hence limits the type of latent bugs that can be identified.

4.5 Classification, Clustering, Associating, and Traceability Recovery

Several research efforts share a similarity with bug classification in that they also extract features (terms) from source code, and then feed them into classification or clustering algorithms. These efforts have goals other than predicting bugs, including classifying software into broad functional categories, clustering related software project documents, and associating source code to other artifacts such as design documents.
Krovtez et al. use terms in the source code (as Research that categorizes or associates source code with other documents (traceability recovery) is similar to ours in that it gathers terms from the source code and then uses learning or statistical approaches to find associated documents. For example, Maletic et al. extracted all features available in the source code via Latent Semantic Analysis (LSA), and then used this data to cluster software and create relationships between source code and other related software project documents. In a similar vein, Kuhn et al. used partial terms from source code to cluster the code to detect abnormal module structures. The goal of the classification is to place a bug report into a specific category of bug report, or to find the developer best suited to fix a bug. This work, along with change classification, highlights the potential of using machine learning techniques in software engineering.

If an existing concern –such as assigning bugs to developers, can be recast as a classification problem, then it is possible to leverage the large collection of data stored in bug tracking and software configuration management systems.

4.6 Text Classification

Text classification is a well-studied area with a long research history. Using text terms as features, researchers have proposed many algorithms to classify text documents, such as classifying news articles into their corresponding genres. Among existing work on text classification, spam filtering is the most similar to ours. Spam filtering is a classification problem to identify email as spam or ham (not spam). This paper adapts existing text classification algorithms into the domain of source code change classification. Our research focuses on generating and selecting features related to buggy source code changes.

5. Conclusion

Change classification differs from previous bug prediction work since it Classifies changes **is** Previous bug prediction work focuses on finding prediction or regression models to identify fault-prone or buggy modules, files, and functions. Change classification predicts whether there is a bug in any of the lines that were changed in one file in one SCM commit transaction. This can be contrasted with making bug predictions at the module, file, or method level. Bug predictions are immediate, since change classification can predict buggy changes as soon as a change is made.

6. Reference

1. Shivkumar Shivaji, E. James Whitehead, Jr., Ram Akella, Sunghun Kim, “Reducing Features to Improve Code Change Based Bug Prediction” in IEEE 2012.
2. V. Challagulla, F. Bastani, I. Yen, and R. Paul. Empirical Assessment of Machine Learning Based Software Defect Prediction Techniques. In *Object-Oriented Real-Time Dependable Systems, 2005. WORDS 2005. 10th IEEE International Workshop on*, pages 263–270. IEEE, 2005.
3. M. DAmbros, M. Lanza, and R. Robbes. Evaluating Defect Prediction Approaches: a Benchmark and an Extensive Comparison. *Empirical Software Engineering*, pages 1–47, 2011.
4. B. Efron and R. Tibshirani. *An Introduction to the Bootstrap*, volume 57. Chapman & Hall/CRC, 1993.
5. K. Elish and M. Elish. Predicting Defect-Prone Software Modules Using Support Vector Machines. *Journal of Systems and Software*, 81(5):649–660, 2008.
6. R. Fan, K. Chang, C. Hsieh, X. Wang, and C. Lin. Liblinear: A Library for Large Linear Classification. *The Journal of Machine Learning Research*, 9:1871–1874, 2008.
7. T. Fawcett. An Introduction to ROC Analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006.
8. J. Friedman, T. Hastie, and R. Tibshirani. *The Elements of Statistical Learning*, volume 1. Springer Series in Statistics, 2001.
9. K. Gao, T. Khoshgoftaar, H. Wang, and N. Seliya. Choosing Software Metrics for Defect Prediction: an Investigation on Feature Selection Techniques. *Software: Practice and Experience*, 41(5):579–606, 2011.
10. T. Gyim´othy, R. Ferenc, and I. Siket. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. *IEEE Trans. Software Eng.*, 31(10):897–910, 2005.