Merits of Using Repository Metrics in Defect Prediction for Open Source Projects

Bora Çağlayan[1], Ayşe Bener[2], Stefan Koch[3]
Boğaziçi University
Department of Computer Engineering[1,2], Department of Management[3]
Istanbul, Turkey
{bora.caglayan[1], bener[2], stefan.koch[3]}@boun.edu.tr

Abstract

Many corporate code developers are the beta testers of open source software.They continue testing until they are sure that they have a stable version to build their code on. In this respect defect predictors play a critical role to identify defective parts of the software. Performance of a defect predictor is determined by correctly finding defective parts of the software without giving any false alarms. Having high false alarms means testers/ developers would inspect bug free code unnecessarily. Therefore in this research we focused on decreasing the false alarm rates by using repository metrics. We conducted experiments on the data sets of Eclipse project. Our results showed that repository metrics decreased the false alarm rates on the average to 23% from 32% corresponding up to 907 less files to inspect.

1. Introduction

In recent years, free and open source software (OSS) has drawn increasing interest, both from the business and academic world. The main ideas of this development model are described in the seminal work of Raymond [17], 'The Cathedral and the Bazaar', first published in 1997. Raymond contrasts the traditional model of software development, which he likens to a few people planning a cathedral in splendid isolation, with the new 'collaborative bazaar' form of open source software development. In the latter model, a large number of developer-turned users come together without monetary compensation to cooperate under a model of rigorous peer-review and take advantage of parallel debugging which altogether leads to innovation and rapid advancement in developing and evolving software products. In order to enable this while minimizing duplicated work, the source code of the software needs to be accessible, which necessitates suitable licenses, and new versions need to be released often. Most often, the licence a software is under is used to define whether it is open source software or not[15, 18]. Nevertheless, usually a certain development style and culture are also implicitly assumed, although no formal definition or description of an open source development process exists, and there is considerable variance in the practices actually employed by open source projects.

Many software development companies increasingly use OSS to write scripts and IDE s in order to adopt OSS to their own environment. They usually use the beta versions of OSS; report bugs or fix bugs. In a way those commercial users (developers) act as the test team for OSS. Although OSS is free, there is an opportunity cost for everyone involved. So effective use of testing effort in OSS is as important as commercial software development. Any model or method to be proposed should not cause developers to allocate a large portion of their debugging effort to the unfruitful exploration of erroneous alarms. Therefore in this research we would like to answer the following research question: *"How can we decrease the false alarm rates in open source projects?"* Many researchers have proposed predictive models using machine learning techniques[9, 20, 14]. Compared to other verification, validation and testing methods defect predictors are shown to be effective tools to support critical decisions such as "when to stop testing" in software testing [9]. Traditionally static code attributes are used for modeling software data in defect prediction. Although static code attributes can automatically be extracted from the source code their information content is limited. Since software is a complex system with many possible approaches for abstracting it, there is a need to explore alternative metric sets such as history metrics and organizational metrics for defect prediction.

In our research we used Eclipse Project up to version 3.0

[1]This research is supported in part by Turkish Scientific Research Council (Tubitak) under grant number EEEAG 108E014.

[2]. We conducted experiments by using bug data, static code attributes, and repository metrics from Eclipse Project. Our results showed that repository metrics are more cost effective leading to 28% less code inspection without compromising on the prediction accuracy.

2 Related Work

There has been a few research that focus on the usage of repository metrics in defect prediction for OSS. Mockus et al.[11] worked on predicting the defect density of Apache as compared to commercial projects. This was done based on a realtively small dataset, and they did not consider the number of developers as a distinguishing factor. Koch and Neumann[7] used process and product measures in a study of open source frameworks. They found that a high number of programmers and commits, as well as a high concentration are associated with problems in quality on class level, mostly to violations of size and design guidelines. This underlines the results of Koru and Tian[8], who have found that modules with many changes rate quite high on structural measures like size or inheritance. On project level, there is a distinct difference: Those projects with high overall quality ranking have more authors and commits, but a smaller concentration have poor quality ranking. This leads to a conclusion on a general level for OSS is that as many people as possible should be attracted to a project. On the other hand, these resources should, from the viewpoint of product quality, be organized in small teams. Ideally, on both levels, the effort is not concentrated on too few of the relevant participants.
Learning-based defect predictors are often built using static code attributes and historical data including the location of defects from completed projects [19, 9, 20, 21, 14]. However, due to limited information content of static code attributes, many algorithms suffer from a ceiling effect in their prediction performances such that they are unable to improve the defect detection performance using available size and complexity metrics [10]. There are studies that focus on other factors affecting an overall software system such as the dependencies, code churn metrics or organizational (repository) metrics related with the development process [13, 10]. Results of these studies show that the ability of process-related factors to identify failures in the system is significantly better than the performance of size and complexity metrics. Zimmermann and Nagappan challenged the limited information content of data in defect prediction. The authors proposed to use network metrics that measure dependencies, i.e. interactions, between binaries of Windows Server 2003. Results of their study show that the network metrics have higher performance measures in finding defective binaries than code complexity metrics. Moser et al. also used repository metrics and they concluded that repository metrics give better prediction results than static code metrics [12]. In Moser et al.'s work only post-release defect data of Eclipse Project defects are extracted. They trained and tested on the same release. They also applied a cost sentitive classification approach to improve the performance of their model. They stated that cost sensitive approach would be hard to implement in real life. It is also difficult to derive clear conclusions from their results such that whether the cost sensitive classification or usage of repository metrics improved the results. In another study the effect of number of committers have been investigated in a large scale commercial project. The outcome of this study revealed that the number of developers did not have any effect on the defect density [23].
Many research so far focused mainly on the performance of the defect predictor in terms of its probability of detection (pd rate). However, probability of false alarms (pf) is equally important. High pf rate means that all defect-free modules are classified as defective, which yields inspection of all these modules unnecessarily and contradicts with the purpose of defect prediction. In our previous work we tackled this problem as elimination of extraneous factors (i.e.irrelevancies) in data that was composed of static code attributes[22]. We achieved lower pf rates by using a filtering algorithm to improve the performance of our model.

3 Methodology

3.1 Data Set

Eclipse project is a widely used multi-language software development platform written mainly in java language[1]. Its source has its roots in IBM VisualAge IDE. The project was made open-source in November 2001 and it was licensed initially under creative-commons license and later Eclipse Public License. Both of these licences are compatible with FSF standards for open source licences.
During the history span of Eclipse project that we examined, main committers to the project were IBM employees. According to [6] this categorizes Eclipse as a strictly governed project with commercial open source characteristics.

3.2 Data Collection

For the purposes of this study Eclipse defect data submitted to Promise dataset[2, 24]was used as a basis. In mining defects data of Eclipse Zimmermann et al.[24] made an assumption to find post-pre release defects: since an open source program life cycle is usually blurry, defects that occurred within six months after a release are assumed to be post-release while defects that are found before a release up to six months are assumed to be pre-release defects.

Table 1. Files and Defects for Eclipse Versions

Eclipse Version	Nr. of Files	Pre Release Defected
2.0	6727	2609
2.1	7886	1785
3.0	10590	1821

We have extracted history metrics from the cvs revision system and matched with Promise data set by writing custom scripts in Python language. Our classification was done on the source files by labeling the files containing one or more defects as defected, and the files containing zero defects as not defected. We used pre-release defective files, so for each version of Eclipse, we constructed 3 metric sets using static and repository metrics alone and by combining them. Number of files and pre-release defects for each release are given in Table 1.

Open source projects usually have a small amount of core developers doing majority of the commit work. Using Sourceforge projects Koch[7] found that on average 10 percent of developers are responsible for 90 percent of the commits. Eclipse project's commit effort does not show this characteristic which is an unusual behavior for an open source project as seen on Figure 1. Top 10% developers of eclipse are responsible for less then 50% of commits. 10 history metrics were extracted from CVS repository of Eclipse project:

1. Commits(Com) $\rightarrow$ Number of commits on file cummulatively
2. Committers (Ctr) $\rightarrow$ Number of committers on file cumulatively
3. Lines Added (Lad) $\rightarrow$ Lines added cummulatively
4. Lines Removed (Lrm) $\rightarrow$ Lines removed cummulatively
5. Commits Last Release (ComLR) $\rightarrow$ Number of commits for last release
6. Committers Last Release (CtrLR) $\rightarrow$ Number of committers on files for last release
7. Lines Added Last Release (LadLR) $\rightarrow$ Lines added on files for last release
8. Lines Removed Last Release (LrmLR) $\rightarrow$ Lines added on files for last release
9. Gini Inequality Coefficient (GINI) $\rightarrow$ Commit inequality coefficient of committers on files
10. Top Committer Percentage (TCPer) $\rightarrow$ Percentage of top committers on files

While first 8 metrics have been in previous studies[13, 23, 12], GINI and Top Committer Percentage are new metrics and to the best of our knowledge they were not used in any previous defect prediction work.

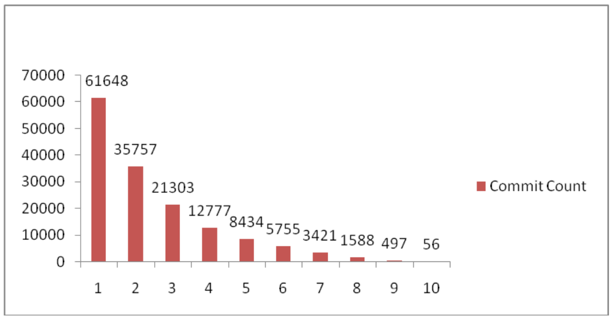

Figure 1. Commit Effort of Nth 10 percent committers on defect count

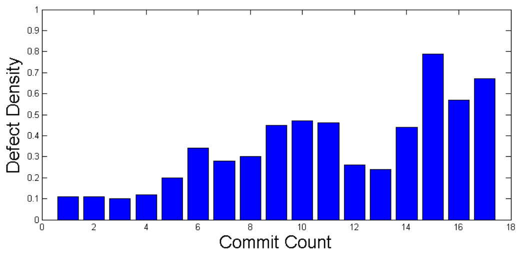

Figure 2. Defect density for Eclipse Release 3.0 wrt Commit Count

Formula for finding Gini inequality of a file is given below:

$$\mu = \frac{\text{Number of Commits}}{\text{Number of Committers}}$$

$$Gini = \frac{\sum \Delta\text{Commits by Committer}}{\mu \text{Total Commits}^2}$$

For the definition of top committers criteria of most commits was used as a top committer criteria.

3.3 Important Trends In Repository Metrics

In Figure 2 it can be seen that defect proneness increases for Eclipse project as number of committers increases. From the Figure it is clear that there is a correlation between defect densities.

In Figure 3 effect of top committer effort to defect proneness is seen. Trend in the effect of top committers effect on defect density is not as clear. Top defect rates are seen when top committers are supported by other committers with a small number of commits (1-10% other developer commits and 90-99% top developer commits).

Table 2. Confusion Matrix

	actual	defected	defect free
predicted			
defected		A	B
defect free		C	D

3.4 Defect Prediction Model

In this study, we have focused on increasing information content of the input data by using additional metrics extracted from repository. We have used the Naive Bayes data mining algorithm that achieves significantly better results than many other mining algorithms for defect prediction [9]. One of the reasons for the success of Naive Bayes algorithm is that it manages to combine signals coming from multiple attributes. It simply uses attribute likelihoods derived from historical data to make predictions for the modules of a software system [4]. As the inputs to the model, we have mined repository metrics together with the actual defect information and static code attributes from Eclipse Project on three versions.

3.5 Performance Measures

In order to assess the performance of our defect predictor on various metric sets, we have calculated well-known performance measures: probability of detection (pd), and probability of false alarms (pf) rates [4]. Pd, which is also defined as recall, measures how good our predictor is in finding actual defective modules. Pf, on the other hand, measures the false alarms of the predictor, when it classifies defect-free modules as defective. In the ideal case, we expect from a predictor to catch all defective modules (pd = 1). Moreover, it should not give any false alarms by misclassifying actual defect-free modules as defective (pf = 0). The ideal case is very rare, since the predictor is activated more often in order to get higher probability of detection rates [4]. This, in turn, leads to higher false alarm rates. Thus, we need to achieve a prediction performance which is as near to (1,0) in terms of (pd,pf) rates as possible. These parameters are found from the confusion matrix shown in table2. The optimum pf,pd rate combinations can vary from project to project. On a safety critical project pd rate can be more important while on a budget constrained project low pf rate can be more important for resource allocation.

$$pd = \frac{A}{A+C} \tag{1}$$

$$pf = \frac{B}{B+D} \tag{2}$$

Open source Weka Machine Learning program and its Java API was used to implement our experiments[3]. When

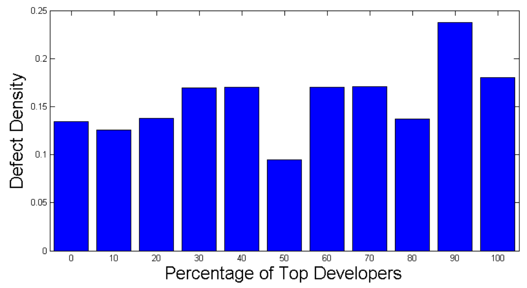

Figure 3. Eclipse Top Committer Effort wrt Error Density

same dataset was used for training and testing 10-fold cross validation was used to eliminate bias. Promise data of Eclipse had around 200 metrics with various degrees of independence so supervised feature selection algorithm of Weka was employed to reduce the number of metrics as a preprocessing step.
Since data distribution can not be assumed to be normal Mann-Whitney U test was done on data to test the statical significance of using different techniques.
We have also conducted a cost-benefit analysis of our results based on the work of Arisholm et al.[5]. Since our focus in this research is to achieve lower pf rates we would like to find out the benefit of having low pf rate in terms of number of lines inspected keeping the pd rate constant. The below formula takes all of our predictions and determines the savings in inspection time in terms of effort spent. The formula contains the files we marked correctly as defective as well as our false alarms. Therefore it includes the pf rate in it. The lower the pf rate the higher the benefit would be. In other words the real value comes from the increase in "D" and decrease in "B" which is the definition of pf.

$$pf \propto \frac{A+B}{A+B+C+D} \tag{3}$$

$$\text{Benefit } \% = pd - \frac{A+B}{A+B+C+D} \tag{4}$$

$$\text{Benefit (files)} = (pd - \frac{A+B}{A+B+C+D}) * \text{Total Files} \tag{5}$$

3.6 Results

We have compared our results for different metric sets. In the first case same versions are used for test and training to estimate the pre-release defects. The results of same version testing can be seen at Table 3. We see that when repository metrics used alone both pd and pf improves. When

Table 3. Comparison of Static,Repository and Both Metrics Using Cross Validation

Metric Set	Version	Defect	pd	pf
All	2.0	Pre Release	0.68	0.35
All	2.1	Pre Release	0.68	0.27
All	3.0	Pre Release	0.68	0.29
		Average	0.68	0.29
		Std Dev	0	0.04
Repository	2.0	Pre Release	0.74	0.29
Repository	2.1	Pre Release	0.66	0.17
Repository	3.0	Pre Release	0.66	0.21
		Average	0.67	0.23
		Std Dev	0.05	0.09
Static	2.0	Pre Release	0.68	0.37
Static	2.1	Pre Release	0.68	0.29
Static	3.0	Pre Release	0.68	0.30
		Average	0.68	0.32
		Std Dev	0	0.04

Table 4. Comparison of Static,Repository and Both Metrics For Different Versions

Metric Set	Version	Test Version	pd	pf
All	2.0	2.1	0.68	0.35
All	2.1	3.0	0.69	0.32
Repository	2.0	2.1	0.74	0.35
Repository	2.1	3.0	0.74	0.31
Static	2.0	2.1	0.68	0.37
Static	2.1	3.0	0.68	0.34

they are used in combination with static code attributes only pf improves. In the case of static code attributes modules with extreme characteristics (i.e. complexity, size) may have been high and and hence their cumulative effect on the overall model increase significantly. However, factors such as number of commits, committers, lines added, lines removed, top committer percentage, etc. may have impacts on the module characteristics. Therefore these characteristics (i.e. repository metrics) contain more relevant information to detect the defective files. In the second case successive versions are used for estimating pre-release defects in successive releases. The result of these tests can be seen at Table 4. We see that whether we train on the same version or use successive versions for training our results do not change. Again the effect of repository metrics in lowering pf rate is positive. All of our reported results passed Mann-Whitney U test, P<0.5 so that low pfs are significantly better in the case of repository metrics where pds are statistically indifferent.

Cost-Benefit Analysis: As explained in section 3.4 we conducted a cost-benefit analysis on our results. We looked at the experiment results of training from the defects of previous release. We see that 30% was gained in inspection effort that corresponds to 3173 less files to inspect for release 3.0. The detailed results can be seen at Table 5.

Table 5. Comparison of Benefits for Using Different Metric Sets with Successive Releases

Metric Set	Train Version	Test Version	Benefit %	Benefit (Files)
Repository	2.0	2.1	0.28	2174
	2.1	3.0	0.3	3173
Static	2.0	2.1	0.27	2094
	2.1	3.0	0.27	2827
All Metrics	2.0	2.1	0.26	2031
	2.1	3.0	0.28	2935

4. Threats to Validity

Threats to validity in experimental studies, which include retrospective artifact analysis, can be categorized into construct, internal, and external validity [16]. The experimental design follows the framework suggested as a baseline by Menzies et al. [9]. We have used 10-fold cross-validation in all experiments. That is, data sets are divided into 10 bins, 9 bins are used for training and 1 bin is used for testing. Repeating these 10 folds ensures that each bin is used for training and testing while minimizing the sampling bias. Each holdout experiment is also repeated 10 times and in each repetition the data sets are randomized to overcome any ordering effect and to achieve reliable statistics. We have applied Mann-Whitney U test with a p<0.5 in order to determine the statistical significance of mean results. One of the common threats to external validity of empirical research is the difficulty of generalizing results. Open source software projects may have distinct characteristics arising from the organizational structure they evolve from or their application domain. For this reason additional work should be done before generalizing these findings for open source projects.

5. Conclusions and Future Work

In this study we focused on improving the prediction performance of our learning based model in terms of achieving lower pf rates. In our previous work we had seen that algorithms have reached a ceiling such that we have failed to find a better algorithm to improve pf rates. On the other hand static code attributes have limited information content. Descriptions of software modules only in terms of static code attributes can overlook some important aspects of software including: the type of application domain; the skill level of the individual programmers involved in system development; contractor development practices; the variation in measurement practices; and the validation of the measurements and instruments used to collect the data. For this reason we augmented, and replaced static code measures with repository metrics such as past faults or changes to code or number of developers who have worked on the code, etc. We used data from 3 versions of Eclipse Project. Bug data was already available at Promise Repository. We mined and extracted the repository metrics by writing our own scripts. Repository metrics on these projects are now available in Promise Repository. Our results show that repository metrics give better insight to software product and hence we were able to lower pf rate on the average from 32% to 23% corresponding up to 907 less files to inspect compared to using only static code attributes. Also while answering our research question we have noted some other interesting characteristics. We have seen that there is a linear relationship between the number of cumulative counts and defect density. Also for the Eclipse project we did not find any significant relationship between the ratio of "top" committers in terms of commit count and defect density. Open source development does not follow the same process and patterns for different projects. As a future work a large scale study of open source projects from different set of backgrounds would be useful to validate our findings.

References

[1] Eclipse project website. `http://www.eclipse.org`.

[2] Promise website. `http://promisedata.org`.

[3] Weka website. `http://www.cs.waikato.ac.nz/ml/weka/`.

[4] E. Alpaydin. *Introduction to Machine Learning*. The MIT Press, October 2004.

[5] E. Arisholm and L. C. Briand. Predicting fault-prone components in a java legacy system. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 8–17, New York, NY, USA, 2006. ACM.

[6] E. Capra, C. Francalanci, and F. Merlo. An empirical study on the relationship between software design quality, development effort and governance in open source projects. *Software Engineering, IEEE Transactions on*, 34(6):765–782–, 2008.

[7] S. Koch and C. Neumann. Exploring the effects of sourceforge.net coordination and communication tools on the efficiency of open source projects using data envelopment analysis. *Empirical Software Engineering*.

[8] A. Koru and J. Tian. Comparing high-change modules and modules with the highest measurement values in two large-scale open-source products. *IEEE Trans. Softw. Eng.*, 31(8):625–642, 2005.

[9] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *Software Engineering, IEEE Transactions on*, 33(1):2–13–, 2007.

[10] T. Menzies, B. Turhan, A. B. Bener, G. Gay, B. Cukic, and Y. Jiang. Implications of ceiling effects in defect predictors. In *Proceedings of the 4th international workshop on Predictor models in software engineering*, pages –, Leipzig, Germany, 2008. ACM.

[11] A. Mockus, R. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, 2002.

[12] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 181–190, 2008.

[13] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, pages –, 2007.

[14] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *Software Engineering, IEEE Transactions on*, 31(4):340–355–, 2005.

[15] B. Perens. *Open Sources: Voices from the Open Source Revolution*, chapter The Open Source Definition, pages 246–256. O'Reilly and Associates, 1999.

[16] D. E. Perry, A. A. Porter, and L. G. Votta. Empirical studies of software engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 345–355, New York, NY, USA, 2000. ACM.

[17] E. S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'reilly and Associates, 1999.

[18] R. M. Stallman. *Free Software, Free Society: Selected Essays of Richard M. Stallman*. GNU Press, 2002.

[19] A. Tosun, B. Turhan, and A. Bener. Ensemble of software defect predictors: a case study. In *ESEM '08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 318–320, New York, NY, USA, 2008. ACM.

[20] B. Turhan and A. Bener. A multivariate analysis of static code attributes for defect prediction. In *Quality Software, 2007. QSIC '07. Seventh International Conference on*, pages 231–237–, 2007.

[21] B. Turhan and A. Bener. Software defect prediction: Heuristics for weighted naive bayes. In *Proceedings of the 2nd International Conference on Software and Data Technologies (ICSOFT'07)*, pages 244–249, 2007.

[22] B. Turhan, T. Menzies, A. Bener, and J. Distefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering Journal*, 2009. in print. DOI 10.1007/s10664-008-9103-7.

[23] E. Weyuker, T. Ostrand, and R. 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–, 2008.

[24] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, May 2007.