# An Enhanced Fault Prediction Model for Embedded Software based on Code Churn, Complexity Metrics, and Static Analysis Results

Safa Omri
Karlsruhe Institute of Technology
Germany
Email: safa.omri@kit.edu

Carsten Sinz
Karlsruhe Institute of Technology
Germany
Email: carsten.sinz@kit.edu

Pascal Montag
Daimler AG
Boeblingen
Germany
Email: pascal.montag@daimler.com

***Abstract*—Software systems evolve over time because of functionality extensions, changes in requirements, optimization of code, fixes for security and reliability bugs, etc., and it is commonly known that software quality assurance is thus a continuous issue and is often extremely time-consuming. Therefore, techniques to obtain early estimates of fault-proneness can help in increasing the efficiency and effectiveness of software quality assurance. The ability to predict which components in a large software system are most likely to contain the largest numbers of faults in the next release helps to better manage projects, including early estimation of possible release delays, and affordably guide corrective actions to the quality of the software. This paper extends our previous work, where we demonstrated that the combination of code complexity metrics together with static analysis results allows accurate prediction of fault density and to build classifiers discriminating faulty from non-faulty components. The extension presented in this paper augments our predictor and classifier with code churn metrics. We applied our methodology to C++ projects from Daimler's head unit development. In experiments to separate fault-prone from non-fault-prone components, our new approach achieved a classification accuracy of $89\%$, and the regressor predicted the fault density with an accuracy of $85.7\%$. This is an improvement of $7.5\%$ with respect to the accuracy of fault density prediction, and an improvement of $10\%$ to the accuracy of fault classification compared to our previous approach that did not take code churn metrics into account.**

***Keywords–Software defects mining; static analysis tools; statistical methods; complexity metrics; churn metrics; fault proneness.***

## I. INTRODUCTION

Software plays an important role in automotive product development and in embedded systems in general. As such software is often safety-critical, considerable efforts have to be put into quality assurance. Increasing the effectiveness and efficiency of this effort hence becomes more and more essential.

It is generally acknowledged that software quality assurance is a pressing concern for embedded software development [1]. Given the size, complexity, time and cost pressure in automotive development projects, efficiency is of prime importance. Nowadays, quality assurance is overall the most expensive activity for nearly all software developing companies, since team members need to spend a significant amount of their time inspecting the entire software in detail rather than, for example, implementing new features. If bugs are detected, the fixing of those consumes further development time.

Numerous research studies have analyzed code churn as a variable for predicting faults in large software systems [2], [3], [4]. Code churn is a measure of the quantity of code modification occurring within a software component gradually. But not only code churn is an indicator of problematic code. In our previous work [5], we investigated whether defects detected by static analysis tools combined with code complexity metrics can be used as software quality indicators and employed these measures to build pre-release fault prediction models. We showed in a case study from the automotive domain that the combination of these two measures can be used to predict the pre-release fault density with an accuracy of $78.3\%$. We have also shown that this combination can be used to separate high and low-quality components with a classification accuracy of $79\%$.

High churn is typically related to more faults showing up in code that has actually been changed frequently. And studying these changes that take place during software evolution via code churn is also important. We thus make use of code churn to predict the fault density in software components. We have mined the version control database of a large software system to collect code churn variables. We create as well as validate a collection of relative churn variables as early indicators of software fault density. Relative churn variables are normalized values of the numerous measures acquired throughout the churn procedure [4].

In this article, we develop a prediction model based on the following hypothesis: the history of code changes between different releases (code churn) when combined with our two previous measures can improve the prediction accuracy of software faults density. Another contribution is to apply our prediction model to automotive software, where we obtain improved results compared to our previous approach.

The organization of the paper is as follows. After discussing the state of the art in Section II, we describe the design of our approach in Section III. Our results are reported and discussed in Section IV. Section V concludes and discusses future work.

## II. RELATED WORK

This section discusses the state of the art and the research results in software fault prediction techniques:

 ISBN: 978-1-61208-752-8 

### A. Faults, Bugs and Failures

In this work, we use the term *fault* to refer to a bug (an error) in the source code. A bug is a fault in a program which causes it to behave abruptly. We refer to an observable error at program run-time as *failure*. That is, every failure can be traced back to a fault, but a fault does not necessarily result in a failure. In recent years, researchers have learned to exploit the vast amount of data that is contained in software repositories such as version and bug databases [4], [6], [7], [8]. The key idea is that one can map problems (in the bug database) to fixes (in the version database) and thus to those locations in the code that caused the problem [9], [10], [11]. The focus of this work is these faults to obtain an early estimate of software component's fault-proneness in order to guide software quality assurance towards inspecting testing the components most likely to contain faults. Fault-proneness is defined as the probability of the presence of faults in the software. Past research on fault-proneness has focused on (i) the definition of code complexity and testing thoroughness metrics, and (ii) the definition and experimentation of models relating metrics with fault-proneness. Moreover, extracting data when mining the software repositories help to better identify the fault-proneness of software components.

### B. Mining Software Repositories

Mining software repositories allows researchers to analyze the information produced throughout the software development process, such as source code, version control system's metadata, as well as issue reports [12], [13], [14]. With such evaluation, researches can empirically examine, understand, and also discover valuable and also actionable insights for software engineering. The extracted data when mining the software repositories help to understand the impact of code smells [15], [16], explore exactly how developers are doing code reviews [17], [18], [19], [20] as well as which testing practices they comply with [21]. Furthermore, historical information extracted from software repositories allows researchers to predict classes that are more susceptible to defects [11], [22], [23], [24], and also determining the core developers of a software team, e.g., to transfer knowledge [25]. Our basic hypothesis is that while these works used only the change and historical information from the source code, it is highly likely that these detected information from software repositories, combined with code complexity metrics and with static analysis faults would be a good indicator of the overall code quality, and help to enhance the fault prediction model presented in our previous work [5].

### C. Fault Prediction

Fault prediction is an active research area in the field of software engineering. Many techniques and metrics have been developed to improve fault prediction performance.

Object-oriented metrics were initially suggested by Chidamber and Kemerer [26]. Basili et al. [27] and Briand et al. [28] were among the first to use such metrics to validate and evaluate fault-proneness. Subramanyam and Krishnan [29] and Tang et al. [30] showed that these metrics can be used as early indicators of externally visible software quality. D'Ambros et al. have compared popular fault prediction approaches for software systems [31], namely, process metrics [32], previous faults [33] and source code metrics [27]. Nagappan et al. [34] presented empirical evidence that code complexity metrics can predict post-release faults. They found that sets of complexity metrics are correlated with post-release defects using five major Microsoft products, including Internet Explorer 6.

Omri et al.'s work [5] builds on the study of Nagappan et al. [34] and focuses on pre-release faults while taking into consideration not only the code complexity metrics but also the faults detected by static analysis tools to build accurate pre-release fault predictors [5].

Furthermore, faults are closely related to changes made in the software systems and studying the changes that take place during software evolution via code churn is also important. Khoshgoftaar et al. [2] were among the first to use past changes for bug prediction. Their objective was to classify the modules as fault-prone or not. Therefore, they identified modules where debug code churn exceeded a threshold. They showed, by studying the change history of two consecutive releases of a large legacy software system of telecommunications, that a high code churn, i.e., a high amount of lines added and removed, is a good indicator of fault-prone modules. The system studied contain over 38,000 procedures in 171 modules. Ohlsson et al. [35], Graves et al. [3] studied the evolution of changes in the software systems to understand their relationship with software quality. Based on a study on eight large-scale open source systems (Eclipse, Postgres, KOFFICE, gcc, Gimp, JBOSS, JEdit and Python), Zimmermann et al. [8] mined the version histories and predicted the location of future changes in systems with an accuracy of 70%. Closely related to our study is the work performed by Nagappan and Ball [4] on predicting defect density in software systems using relative code churn metrics, i.e., code churn weighted by lines of code. They analyze different code churn measures in isolation, and show that relative code churn is better than absolute code churn values to predict defects at statistically significant levels. Their approach is similar to ours in the sense that we are also considering relative churn variables to predict fault potential. However, we focus on predicting fault density on an extended number of variables including code complexity metrics and the faults detected by static analysis tools.

To the best of our Knowledge, this work is the first to combine code churn metrics with code complexity metrics and with static analysis results to predict software defect density.

## III. Approach

Our approach, represented in Figure 1, can be summarized in the following two steps:

### A. Data Pre-Proprocessing

First, we collect the data required to train and test the fault prediction models (the regressor and the classifier) out of a git versioned software project. Git versioning allows us to capture the required data for all software releases. The data required to train our fault prediction models is:

1) **Independent variables**: The independent variables are the input variables to the prediction models. (a) *Static analysis faults*: we execute static code analysis on each component for each release. We define the static analysis fault density of a software component as the number of faults found by static analysis tools, per KLOC (thousand lines of code).

(b) *Code complexity metrics*: we compute different code complexity metrics for each of the components and for each release as describes in Table I. (c) *Code churn metrics*: we mine the git repositories databases to extract several code churn metrics (e.g, added LOC, removed LOC, etc., see Table I) for each release.

2) **Dependent variable**: The dependent variable is the output that will be predicted by our prediction models. We mine the git repositories using natural language processing techniques to parse and analyze commit messages mentioning bug fixes keywords (e.g, bug fix, bug fixing, etc.). Such bug fix commits are the indicator of the true known fault density of the software components for each release.

### B. Model Training

We train different machine learning models to learn the fault densities of each software component based on the independent variables: (a) static analysis faults densities, (b) code complexity metrics, and (c) code churn metrics.

We split our data into two parts: (1) train data which accounts for 3 successive releases of all software components, and (2) test data representing the fourth release (the last release).

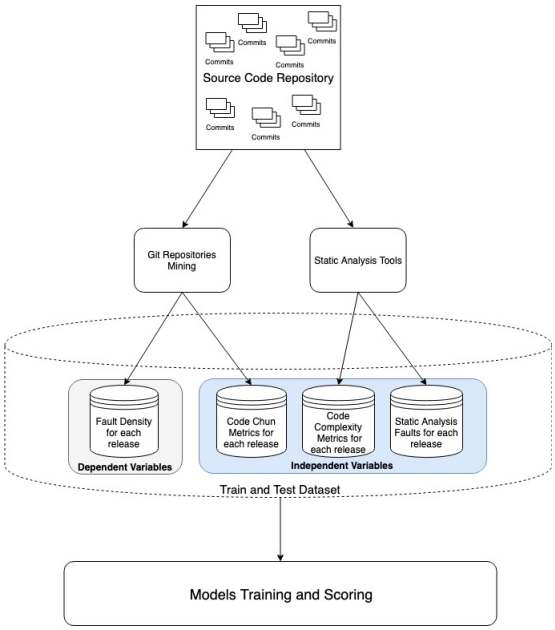


Figure 1. Overview of the fault prediction process

## IV. Emperical Study

In this section, we present the empirical study that we performed to investigate the hypotheses stated in Section I. In this section, we discuss the dataset, the statistical methods and machine learning algorithms we used, and report on the results and findings of the experiments. The experiment was carried out using 70 components of an automotive head unit control system (Audio, Navigation, Phone, etc.). The size of the code base analyzed is 28.71 MLOC (2,871 KLOC). All repositories use the object oriented language C++.

### A. Data Preparation

The goal of this work is to come up with fault predictors that evaluate our hypothesis and enhance our prediction model built in our previous study [5]. The data required to build our fault predictors are:

*1) Faults Data:* We are interested, in this work, in faults that have been detected during the development and mentioned as bugfixes in git commits. For each component, we extracted all detected bugs through mining the git repositories. The extracted faults are then used to compute the **fault density**.

*2) Static Analysis Fault Density:* Moreover, we executed static analysis tools on each component and extracted the identified faults. These faults were then used to compute the **static analysis fault density**. We used commercial non-verifying static analysis tools in this study.

*3) Code Complexity Metrics:* We compute several code complexity metrics for each of the components. The code complexity metrics are represented in Table I. We limit our study to a set of selected metrics that have shown to provide significant quality indicators over a long period of time. Code complexity metrics have been shown to correlate with fault density in several case studies [28], [29], [30], and they have been proposed in different case studies to assess software quality [1], [34].

*4) Code Churn Metrics:* Software repositories contain historic information regarding the overall development of software program systems. Mining software databases is nowadays considered one of the most intriguing expanding areas within software engineering. Different recent works have used past changes as indicators for faults because faults that are introduced by recent changes and the more changes are done to a part of the source code the more likely it will contain faults[36]. Thus, we mine the software repositories databases to extract the churn metrics. We use these code churn metrics, as described in Table I to predict software fault density. This study builds on our work [5] and goes for faults collected through mining the git repositories of all software components, and takes into consideration not only the static faults and the code complexity metrics but also the code churn metrics to build accurate fault predictors.

*5) Relative Code Churn Metrics:* For each of the component, we compute a number of relative code churn metrics, as described in Table I. We show in this paper that using relative code churn as fault predictor is better than using (absolute) code churn predictors. Furthermore, combining relative code churn metrics with code complexity metrics and static analysis faults can accurately predict the fault density with a high degree of sensitivity. Our metric suite in this work is able to discriminate between fault and not fault-prone components with an accuracy of 89.0 percent.

### B. Model Fitting and Regression Analysis

In this section we compare predictive models built using the different metrics presented in Table I in order to find the best model for accurate fault prediction. We fit several models to the absolute code churn data as well as the relative code churn data separately as predictors, with the fault density as the dependent variable. We tested our data on the four main regression models families (Generalized Linear models, Deep Learning Models, Random Forest Models and Boosted Models). The experiment

 ISBN: 978-1-61208-752-8 

TABLE I. METRICS USED FOR THE STUDY

| Metrics | Description |
|---|---|
| Static Analysis Fault Density | # faults found by static analysis tools per KLOC (thousand lines of code). |
| **Code Churn Metrics** | |
| Added LOC | # lines of code added |
| Removed LOC | # lines of code deleted |
| Modified Files | # files modified |
| Files count | # files compiled to create a software component |
| Developers | # developers |
| **Relative Code Churn Metrics** | |
| Added LOC / Relevant LOC | We expect the larger the proportion of added code to the Relevant LOC, the higher is the probability of the presence of faults in the software component. |
| Removed LOC / Relevant LOC | We expect the larger the proportion of removed code to the Relevant LOC, the higher is the probability of the presence of faults in the software component. |
| Modified Files / Files count | We expect the larger the proportion of files in a component that get modified, the higher is the probability of these files introducing faults. |
| **Code Complexity Metrics** | |
| Relevant LOC | # relevant LOCs without comments, blanks, expansions, etc. |
| Complexity | cyclomatic complexity of a method |
| Nesting | # nesting levels in a method |
| Statements | # statements in a method |
| Paths | # non-cyclic paths in a method |
| Parameters | # function parameters in a method |

shows that boosted models are showing the best fitting and generalized accuracy. This result can be explained by the fact that the relation between the independent variables is highly non-linear. The boosted models include RGBoost (also known as regularized gradient boosting), Distributed Random Forests (DRF) as well as Gradient Boosting Machines (GBM). We will shortly explain the model that we used in this study to predict the fault density. RGBoost is a supervised learning algorithm that implements a process called boosting to yield accurate models [37]. Boosting refers to the ensemble learning technique of building many models sequentially, with each new model attempting to correct for the deficiencies in the previous model [38]. In tree boosting, each new model that is added to the ensemble is a decision tree. RGBoost provides parallel tree boosting that solves many data science problems in a fast and accurate way. For many problems, RGBoost is one of the best gradient boosting machine frameworks today [37]. Both RGBoost and GBM follows the principle of gradient boosting. There are, however, differences in modeling details. Specifically, RGBoost uses a more regularized model formalization to control over-fitting, which gives it better performance, especially when the correlation between the independent variables is non-linear. Distributed Random Forest (DRF) is a powerful classification and regression tool. When given a set of data, DRF generates a forest of classification or regression trees, rather than a single classification or regression tree [39]. As a measure of the regression fits, we compute $R^2$. $R^2$ measures the variance in the predicted variable that is accounted by the regression built using the predictors. As a measure of the unbiased error estimate of the error variance, we use the mean squared error ($MSE$). The regression model fit for absolute code churn metrics has an $R^2$ value of $0.473$, an $MSE$ value of $0.235$. Nevertheless, using the relative code churn metrics as fault predictors shows a better fit; the $R^2$ value increases to $0.730$, the $MSE$ decreases to $0.113$. We then combined relative code churn metrics with code complexity metrics and with static analysis fault density as predictors for the fault density. Table II shows that when using the combination relative code churn metrics with code complexity metrics and with static analysis fault density as fault predictors, we obtain the best fit using the Regularized Gradient Boosting (RGBoost) model; the $R^2$ value increases to $0.857$, the $MSE$ decreases to $0.015$. Therefore, we conclude that it is more beneficial to combine relative code churn metrics with code complexity metrics and static analysis fault density to explain software faults. The validation of the model goodness is repeated 10 times using the 10-fold cross-validation technique. A benefit of using ensembles of decision tree methods like regularized gradient boosting is that they can automatically provide estimates of feature importance from a trained predictive model, as presented in Figure 2.

TABLE II. REGRESSION FITS

| | | RGBoost | | DRF | | GBM | |
|---|---|---|---|---|---|---|---|
| | | $R^2$ | MSE | $R^2$ | MSE | $R^2$ | MSE |
| Predictors | Absolute Code Churn Metrics **alone** | 0.473 | 0.235 | 0.325 | 0.337 | 0.592 | 0.195 |
| | Relative Code Churn Metrics **alone** | 0.730 | 0.113 | 0.651 | 0.267 | 0.694 | 0.221 |
| | Relative Code Churn Metrics **Combined With** Code Complexity Metrics and Static Analysis Fault Density | **0.857** | **0.015** | 0.683 | 0.103 | 0.784 | 0.067 |

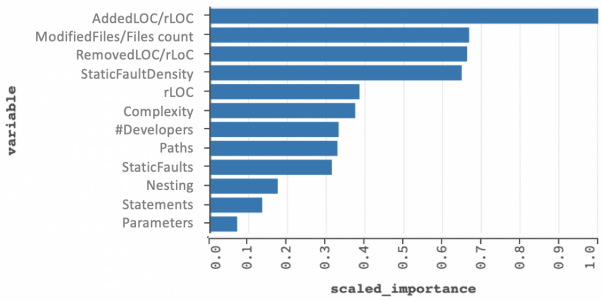


Figure 2. Variable Importances

### C. Fault-Proneness Analysis

In order to classify software components into fault-prone and not fault-prone components, we applied several statistical classification techniques. The classification techniques include the same techniques that we considered for the regression; RGBoost, DRF and GBM. The independent variables for the classifiers are the relative code churn metrics combined with the code complexity metrics and the static analysis fault density. The dependent variable is the result of binarizing (i.e., fault-prone vs. not fault-prone) the fault density. A confusion matrix, as defined in Table III, is used to store the correct and incorrect decisions made by a classification model. For instance, if a component is classified as fault-prone when it is truly fault-prone, the classification is true positive (tp). If the component is classified as fault-prone when it is actually

 ISBN: 978-1-61208-752-8 

TABLE III. COMPARING OBSERVED AND PREDICTED COMPONENT CLASSES IN A CONFUSION MATRIX. USED TO COMPUTE PRECISION AND RECALL VALUES OF CLASSIFICATION MODEL

| | | Observed class: fault prone | Observed class: non-fault prone |
|---|---|---|---|
| Predicted class | fault prone | True negative (TN) | False negative (FN) |
| | non-fault prone | False positive (FP) | True positive (TP) |

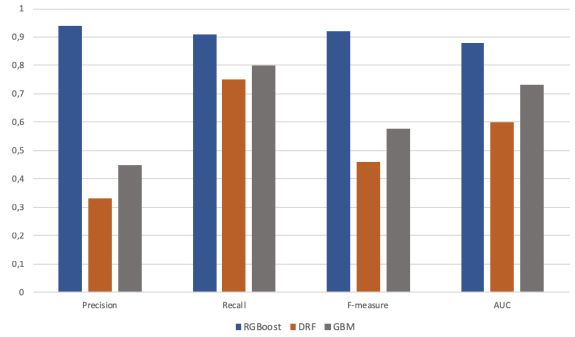


Figure 3. Classification performance of our approach

clean (not fault-prone), then the classification is a false positive (fp). If the file is classified as clean when it is in fact fault-prone, the classification is a false negative (fn). Finally, if the issue is classified as clean and it is, in fact, clean, the classification is true negative (tn). In order to compare the actual observed and predicted classes for each component, we categorized each predicted class into four individual categories as shown in Table III. As evaluation measures, we compute precision, recall, and F-measure defined as:

- Precision: how many of the components classified by our classifiers as fault-prone are actually fault-prone.
$$precision = \frac{tp}{tp+fp}$$
- Recall: how many fault-prone components our classifiers were able to identify correctly as fault-prone.
$$recall = \frac{tp}{tp+fn}$$
- F-measure: measures the weighted harmonic mean of the precision and recall.
$$F\text{-}measure = 2 * \frac{precision*recall}{precision+recall}$$

All two measures are values between zero and one. A precision of one indicates that the classification model does not report any false positives. A recall of one implies that the model does not report any false negatives. The F-measure can be interpreted as a weighted average of the precision and recall, where an F-measure reaches its best value at one and worst at zero. Furthermore, we investigate the use of the area under the receiver operating characteristic (ROC) curve (AUC) as a performance measure for approach. The area under the ROC curve (AUC) equals the probability that the classifiers predict a randomly chosen true positive higher than a randomly chosen false negative. The larger the AUC, the more accurate is the classification model. As shown in Figure 3, the classification model which uses RGBoost as the classifier produced an impressive result with all four performance indicators (Precision, Recall, F-measure and AUC) being well above $0.9$. Using DRF or GBM achieved very high recall, but at the same time it appeared to produce many false positives, and thus their precision is much lower than the precision produced by RGBoost. All studied classifiers achieved an AUC well above the $0.5$ threshold; $0.89$ for RGBoost, $0.6$ for DRF and $0.73$ for GBM.

### D. Threats to Validity

The validity of credibility problems occur when there are mistakes in measurement. This is negated to an extent by the reality that the whole data collection procedure is automated through the version control systems through mining the git repositories. Nevertheless, the version control systems only documents data upon developer check-out or check-in of files. If a developer made several overlapping edits to a file in a single check-out/check-in period then a certain amount of changes will not be visible. Moreover, a developer may have a file checked out for a very long period of time throughout which few churns were made. These worries are reduced somewhat by the cross-check among the measures to recognize irregular values for any of the measures, as well as the significant dimension and diversity of our dataset. In our study, we give proof for utilizing all the relative code churn metrics rather than a subset of values or principal components. This study is particular and ought to be improved based upon further result.

## V. Conclusion And Future Work

In this paper we verified the hypothesis that history of code changes between different commits and releases (code churn) when combined with static analysis fault density and code complexity metrics are a good predictor of pre-release fault density. Moreover, adding code churn metrics increases the prediction accuracy.

For future work we plan to further validate our study by analyzing additional software projects. Attributing fault density to smaller units of code (e.g., files, functions), we consider also an interesting direction of research. To achieve that it might be needed to take additional features of the source code, such as the abstract syntax tree (AST), control- and dataflow into account. For this, we also plan to train deep learning models to predict software faults not only on the component level, but also on the method level. We also plan the generalizability of the presented approach on different open source projects.

## Acknowledgement

We would like to thank Steffen Görzig and Pascal Montag from Daimler AG for their support and for providing the data underlying our case study.

## References

[1] R. Rana, M. Staron, J. Hansson, and M. Nilsson, “Defect prediction over software life cycle in automotive domain state of the art and road map for future,” in 2014 9th International Conference on Software Engineering and Applications (ICSOFT-EA), Aug 2014.

 ISBN: 978-1-61208-752-8 

[2] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan, "Detection of software modules with high debug code churn in a very large legacy system," in Proceedings of the The Seventh International Symposium on Software Reliability Engineering, ser. ISSRE '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 364–.

[3] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," IEEE Transactions on Software Engineering, vol. 26, no. 7, July 2000, pp. 653–661.

[4] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in Proceedings of the 27th International Conference on Software Engineering, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 284–292.

[5] S. Omri, P. Montag, and C. Sinz, "Static analysis and code complexity metrics as early indicators of software defects," Journal of Software Engineering and Applications, vol. 11, no. 4, april 2018.

[6] A. Mockus, P. Zhang, and P. L. Li, "Drivers for customer perceived software quality," in ICSE 2005, 2005.

[7] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," IEEE Trans. Softw. Eng., vol. 31, no. 4, Apr. 2005, pp. 340–355.

[8] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in Proceedings of the 26th International Conference on Software Engineering, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 563–572.

[9] D. Čubranić and G. C. Murphy, "Hipikat: Recommending pertinent software development artifacts," in Proceedings of the 25th International Conference on Software Engineering, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 408–418.

[10] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in Proceedings of the International Conference on Software Maintenance, ser. ICSM '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 23–.

[11] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in Proceedings of the 2005 International Workshop on Mining Software Repositories, ser. MSR '05. New York, NY, USA: ACM, 2005, pp. 1–5.

[12] K. K. Chaturvedi, V. B. Sing, and P. Singh, "Tools in mining software repositories," in Proceedings of the 2013 13th International Conference on Computational Science and Its Applications, ser. ICCSA '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 89–98.

[13] F. Z. Sokol, M. F. Aniche, and M. A. Gerosa, "Metricminer: Supporting researchers in mining software repositories," in 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM), Sep. 2013, pp. 142–146.

[14] A. Zaidman, B. V. Rompaey, S. Demeyer, and A. v. Deursen, "Mining software repositories to study co-evolution of production & test code," in Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation, ser. ICST '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 220–229.

[15] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjoberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in Proceedings of the 2010 IEEE International Conference on Software Maintenance, ser. ICSM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[16] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "The scent of a smell: An extensive comparison between textual and structural smells," in Proceedings of the 40th International Conference on Software Engineering, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 740–740.

[17] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in Proceedings of the 2013 International Conference on Software Engineering, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 712–721.

[18] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in Proceedings of the 11th Working Conference on Mining Software Repositories, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 202–211.

[19] V. J. Hellendoorn, P. T. Devanbu, and A. Bacchelli, "Will they like this?: Evaluating code contributions with language models," in Proceedings of the 12th Working Conference on Mining Software Repositories, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 157–167.

[20] P. Thongtanunam, S. Mcintosh, A. E. Hassan, and H. Iida, "Review participation in modern code review," Empirical Softw. Engg., vol. 22, no. 2, Apr. 2017, pp. 768–817.

[21] D. Spadini, M. Aniche, M.-A. Storey, M. Bruntink, and A. Bacchelli, "When testing meets code review: Why and how developers review tests," in Proceedings of the 40th International Conference on Software Engineering, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 677–687.

[22] A. Bacchelli, M. D'Ambros, and M. Lanza, "Are popular classes more defect prone?" in Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering, ser. FASE'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 59–73.

[23] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in Proceedings of the 2010 10th International Conference on Quality Software, ser. QSIC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 23–31.

[24] L. Pascarella, F. Palomba, and A. Bacchelli, "Re-evaluating method-level bug prediction," in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), March 2018, pp. 592–601.

[25] A. Mockus, R. T. Fielding, and J. Herbsleb, "A case study of open source software development: The apache server," in Proceedings of the 22Nd International Conference on Software Engineering, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 263–272.

[26] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," IEEE Trans. Softw. Eng., vol. 20, no. 6, Jun. 1994.

[27] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," IEEE Trans. Softw. Eng., vol. 22, no. 10, Oct. 1996, pp. 751–761.

[28] L. C. Briand, J. Wüst, S. V. Ikonomovski, and H. Lounis, "Investigating quality factors in object-oriented designs: An industrial case study," in Proceedings of the 21st International Conference on Software Engineering, ser. ICSE '99. New York, NY, USA: ACM, 1999.

[29] R. Subramanyam and M. S. Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects," IEEE Trans. Softw. Eng., vol. 29, no. 4, Apr. 2003.

[30] M.-H. Tang, M.-H. Kao, and M.-H. Chen, "An empirical study on object-oriented metrics," in Proceedings of the 6th International Symposium on Software Metrics, ser. METRICS '99. Washington, DC, USA: IEEE Computer Society, 1999.

[31] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," Empirical Softw. Engg., vol. 17, no. 4-5, Aug. 2012, pp. 531–577.

[32] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in Proceedings of the 30th International Conference on Software Engineering, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 181–190.

[33] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting faults from cached history," in Proceedings of the 29th International Conference on Software Engineering, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 489–498.

[34] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in Proceedings of the 28th International Conference on Software Engineering, ser. ICSE '06. New York, NY, USA: ACM, 2006.

[35] M. C. Ohlsson, A. von Mayrhauser, B. McGuire, and C. Wohlin, "Code decay analysis of legacy software through successive releases," in 1999 IEEE Aerospace Conference. Proceedings (Cat. No.99TH8403), vol. 5, March 1999, pp. 69–81 vol.5.

[36] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, "Automatic identification of bug-introducing changes," in Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, ser. ASE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 81–90.

[37] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in Proceedings of the 22Nd ACM SIGKDD International Conference

on Knowledge Discovery and Data Mining, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 785–794.

[38] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," Annals of Statistics, vol. 29, 2000, pp. 1189–1232.

[39] M. Guillame-Bert and O. Teytaud, "Exact distributed training: Random forest with billions of examples," ArXiv, vol. abs/1804.06755, 2018.

 ISBN: 978-1-61208-752-8 