Computers and Electrical Engineering 67 (2018) 15–24

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

Empirical analysis of change metrics for software fault prediction☆

Garvit Rajesh Choudhary[a], Sandeep Kumar[b,*], Kuldeep Kumar[c,*], Alok Mishra[d], Cagatay Catal[e]

[a] Google Inc., Mountain View California, USA
[b] Department of Computer Science and Engineering, Indian Institute of Technology Roorkee, India
[c] Department of Computer Science and Engineering, Dr B R Ambedkar National Institute of Technology Jalandhar, Punjab, India
[d] Department of Software Engineering, Atilim University, Ankara, Turkey
[e] Information Technology Group, Wageningen University, Wageningen, The Netherlands

ARTICLE INFO

Article history:
Received 17 November 2017
Revised 24 February 2018
Accepted 26 February 2018
Available online 8 March 2018

Keywords:
Software fault prediction
Eclipse
Change log
Metrics
Software quality
Defect prediction

ABSTRACT

A quality assurance activity, known as software fault prediction, can reduce development costs and improve software quality. The objective of this study is to investigate change metrics in conjunction with code metrics to improve the performance of fault prediction models. Experimental studies are performed on different versions of Eclipse projects and change metrics are extracted from the GIT repositories. In addition to the existing change metrics, several new change metrics are defined and collected from the Eclipse project repository. Machine learning algorithms are applied in conjunction with the change and source code metrics to build fault prediction models. The classification model with new change metrics performs better than the models using existing change metrics. In this work, the experimental results demonstrate that change metrics have a positive impact on the performance of fault prediction models, and high-performance models can be built with several change metrics.

1. Introduction

Software fault prediction studies aim to create prediction models that detect software components with more likelihood of having faults. Software metrics data and fault information from previous software releases are used to train the classification model and this model is then used to predict the fault-proneness of the modules in new releases [1]. Each software metrics value can be used to evaluate the software quality; for example, the average value of the Cyclomatic Complexity (CC) metrics in a class can help quality assurance experts to know whether or not that class is risky and needs changes. Risky class indicates that there is a high potential for a bug if that class is modified by software changes.

From a broader perspective, software fault prediction activity might be considered as a way of reliability prediction [2]. Some reliability prediction approaches predict the fault-prone modules before the testing phase; others use reliability

☆ Reviews processed and recommended for publication to the Editor-in-Chief by Associate Editor Dr. A. Isazadeh.

* Corresponding authors.
E-mail addresses: iitgarvit@gmail.com (G.R. Choudhary), sgargfec@iitr.ac.in (S. Kumar), kumark@nitj.ac.in (K. Kumar), alok.mishra@atilim.edu.tr (A. Mishra), cagatay.catal@wur.nl (C. Catal).

https://doi.org/10.1016/j.compeleceng.2018.02.043
0045-7906/

growth models to understand how the reliability changes over time. This study mainly focuses on the prediction of fault-prone modules before the testing phase and does not address reliability growth models. In addition to the fault prediction models, researchers recently developed new models to predict the security dimension of software quality. This research area is known as software vulnerability prediction and machine learning algorithms have also been applied to build these vulnerability prediction models. In this study, we do not address vulnerabilities directly but focus on general software faults.

Instead of using single metrics for the evaluation, these kinds of metrics can be combined in a prediction model. In order to build such a fault prediction model, software metrics and previous fault data must be extracted from software repositories [3]. If not automated, this task can be time consuming, error prone, and costly. However, there are several utilities in the industry that help to automate this process.

Software quality assurance (SQA) involves monitoring the software development process to ensure quality. In this respect, testing is the most time-consuming activity. Organizations are still looking for alternative solutions to reduce these testing efforts and ensure software quality at a lower cost. Software fault prediction models help project managers to distribute verification resources effectively. Recent studies show that the combination of novel metrics, such as product and process metrics, provide better performance than those that only use one type of metric. Madeyski and Jureczko (2015) [4] suggest using Number of Distinct Committers (NDC) metrics in addition to product metrics when building fault prediction models.

From the machine learning perspective, building models using software metrics in conjunction with the fault data is considered as supervised learning. However, there are some cases where either the previous fault data do not exist or there are very limited fault data. Also, it might be the first project undertaken by an organization in that domain, or the organization might not have access to the historical data. Either way, the organization will have no previous fault data. In a global software engineering project, some companies might not necessarily compile and store these kinds of data, and, hence, only limited data exist.

Recently researchers developed several models to address these problems [5]. In these works, product metrics are generally investigated to build a prediction method. Researchers are primarily concerned with these metrics because they are collected more easily from the repositories than other metrics types, such as process metrics. Public prediction datasets mostly contain product metrics.

Researchers have also investigated software metrics threshold values to build prediction models and design noise detection techniques [6]. As shown in these studies, software metrics are one of the most important components in building high-performance fault prediction models. A recent work [7] has emphasized the importance of the characteristics belonging to input metrics datasets in the selection of suitable software fault prediction models. For this reason, the authors of the present paper focus on change metrics to increase the performance of models. Recent studies apply metrics that are related to the developers, organization metrics [8], and network metrics.

In this study, we analyze the impact of change metrics on fault prediction models in detail. Several change metrics are extracted from software repositories, and models are built using machine learning algorithms, namely Decision Tree, K-Nearest Neighbor, and Random Forest.

Software metrics can be classified into the following types:

- Code and complexity metrics [9–11]: Complexity metrics indicate how complex a code block is, and it is widely known that the more complex the code is, the more vulnerability and risk exist for that module. So far, researchers have suggested and applied many complexity metrics (e.g., the well-known CC metric of Thomas McCabe), which are mostly calculated based on the source code. Therefore, we show this category as code and complexity metrics. CC metric shows how many independent paths, which must be tested separately, exist for that module (method/class/package). These metrics are the traditional product metrics derived directly from the software. Some examples for code metrics are lines of code and lines of comment. Examples for complexity metrics are system complexity, McCabeâs cyclomatic complexity, and essential complexity. If a procedural programming language such as C language is used in the project, McCabe and Halstead metrics [12] are the most popular.
- Object-oriented metrics [13,14]: These metrics are valid for an object-oriented programming paradigm, and they are related to specific programming concepts, such as inheritance, class, cohesion, and coupling. Several object-oriented metrics suites have been proposed. The most popular is the CK (Chidamber-Kemerer) metrics suite. The CK metrics are: weighted method count (WMC), coupling between object classes (CBO), number of children (NOC), lack of cohesion in methods (LCOM), depth of inheritance tree (DIT), and response for a class (RFC).
- Change or process metrics [15–18]: These metrics are related to the changes made during the software development process. They are collected throughout the software lifecycle across its multiple releases. Some process metrics are code churn measures, change bursts, and code deltas. Code churn metric shows the rate at which your code evolves [19]. Change burst metric takes into account the consecutive change sequences [18]. Code delta metric computes the difference between two builds in terms of a specific metric such as lines of code.
- Developer metrics [20,21]: These metrics are directly related to the software developer and are extracted for the different developers who contribute to the software. These metrics include the cumulative number of developers revising a module, the cumulative number of developers who changed the file over all the releases, and the lines of code modified by a developer.

- Network metrics [22–24]: These metrics are the most recent ones used for fault prediction. Network metrics are extracted from the dependency relation between different entities. These entities are treated as nodes in a graph, and a dependency graph is created using the interactions between these entities. The application of network analysis on this graph provides the network metrics values, with the aim of finding a correlation between the dependencies and defects. These network metrics include degree centrality, closeness centrality, betweenness centrality, eigenvector centrality, size measures, constraint measures, and ego network measures.

These categories and their underlying software metrics set the scene for a vast area of research in software metrics selection, and better prediction models can be built when these subsets are combined appropriately. In detail, the main objective of this study is to investigate several change metrics in conjunction with code metrics to improve the performance of fault prediction models. The following research questions (RQ) are addressed in this study.

- RQ1: How do the existing change metrics perform in software fault prediction on Eclipse project datasets?
- RQ2: How do the newly proposed change metrics (proposed in this work) perform in software fault prediction on Eclipse project datasets?
- RQ3: How do the change metrics in conjunction with the code metrics perform in software fault prediction when Eclipse datasets are used?

The contribution of this paper is three-fold:

- The effects of change metrics on software fault prediction are evaluated.
- New change metrics are proposed and used successfully, and
- Change metrics in conjunction with code metrics are shown to improve the performance of fault prediction models.

The rest of the paper is organized as follows: Section 2 presents the related work. Section 3 explains the methodology. Section 4 shows the experimental results. Threats to validity are discussed in Section 5. Section 6 presents the conclusion and discusses future work.

2. Related work

Historical data from the Eclipse project regarding releases 2.0, 2.1 and 3.0 were investigated by researchers in the past to build a fault prediction model, and code metrics were used in conjunction with complexity metrics [9]. They shared this dataset, performed experiments at file and package levels, and showed that a combination of complexity metrics helps project managers to predict faults. It was demonstrated that even simple static code metrics such as lines of code are useful to evaluate software quality [10].

The Eclipse and NASA dataset were used in that study, and the benefits of the lines of code for defect prediction models were shown. Machine learning algorithms such as Multilayer Perceptron, Logistic Regression, and Naive Bayes were used to build models based on the lines of code metric. Recently, researchers have been investigating various learning models to predict the number of faults [7,25], with some also proposing ensemble methods [26] for this purpose, but none have carried out any studies on the use of change metrics.

Complexity metrics have also been addressed in previous studies. For example, Zhou et al. [11] analyzed ten software metrics related to object-oriented, complexity, and size metrics, and built a model based on the binary logistic regression method in Eclipse versions 2.0, 2.1 and 3.0. After the application of the prediction model, it was observed that the lines of code (LOC) and the weighted methods per class (WMC) are the most important metrics to predict the fault-proneness of classes. They concluded that complexity metrics have a positive impact on the performance of fault prediction models.

Object-oriented (OO) metrics such as CK metrics have been studied in detail in previous software fault prediction studies. Zhou and Leung [13] showed that CBO, WMC, RFC, and LCOM metrics are useful metrics for fault prediction, but DIT is not useful in NASA datasets. In addition, design metrics can predict low-severity faults better than high-severity faults in faulty classes. Their study validated object-oriented metrics for fault severities. Another empirical analysis on OO metrics was performed by Subramanyam and Krishnan [14] who only dealt with WMC, CBO, and DIT metrics in their experiments, claiming that the predictive power of object-oriented design metrics varies across programming languages such as C++ and Java, and therefore they built two models for each language.

A similar study to our research was performed by Moser et al. [15]. They applied static code metrics and compared the model performance with that of a model based on change metrics extracted from Eclipse CVS change logs. The change metrics are Revisions, Refactorings, BugFixes, Authors, LocAdded, MaxLocAdded, AvgLocAdded, LocDeleted, MaxLocDeleted, AvgLocDeleted, CodeChurn, MaxCodeChurn, AvgCodeChurn, MaxChangeSet, AvgChangeSet, Age, and WeightedAge as explained in detail in Section 3. They observed that change metrics outperformed static code metrics with a 10% increase in accuracy. Decision trees was the best algorithm among the techniques investigated. After analysis of the decision trees, Moser et al. [15] found that the most important predictors were Changeset, Revisions, Refactorings, and BugFixes.

Krishnan et al. [16] investigated whether change metrics are good predictors for software product lines. They used the same change metrics set as used by Moser et al. [15] and chose Eclipse as the evolving product. Their experiments showed

that change metrics are good at predicting the fault-prone files and that the results improved significantly as the product evolved.

Another study on change metrics was performed by Bell et al. [17], who collected code churn and other measurements from 18 releases of a software system. They reported that churn measures that used added lines of code, deleted, and modified lines of code were effective in detecting fault-prone components. The sum of these three metrics were the most effective. Therefore, they suggested using a measure of change metric in building fault prediction models.

Similarly, Nagappan et al. [18] investigated change bursts as a predictor for software defects. Change bursts are defined as consecutive changes over a certain period of time. The study was performed on Windows Vista datasets and change bursts were found to be the best defect predictor, yielding the highest precision and recall value at 90%. They replicated this study on a Windows Server 2003 dataset and achieved precision of 74.4% and recall at 88.0%.

A study on developer metrics was performed by Bell et al. [20], who investigated the impact of data for individual developers on fault prediction performance. Because different programmers might have different coding styles and experiences, it is interesting to evaluate the impact of these individual tasks. In order to investigate the role of each individual developer on the standard prediction model, they suggested a parameter called buggy file ratio. Based on their experiments, they suggested that incorporation of individual developer information provided a very small improvement in the prediction accuracy.

Similarly, Matsumoto et al. [21] studied the effects of developer measures on fault prediction, finding the addition of developer metrics significantly improved the performance of the model in Eclipse datasets. Furthermore, they validated their hypothesis that different developers inject a different number of faults in a module and a module that is revised by more developers tends to contain more faults.

Zimmerman and Nagappan [22] performed a study on the network analysis of dependency graphs, reporting that central binaries have a tendency to be defect prone and those that are part of a larger clique tend to be even more so. They compared the performance of network metrics with complexity metrics on retrieving critical binaries and found that complexity metrics can retrieve only 30% of such binaries, whereas network metrics can retrieve twice as much. In Logistic Regression models, network metrics provided a 10% higher recall value than the one produced with the model using complexity metrics.

This study was replicated by Premraj and Herzig on three open-source Java projects: JRuby, ArgoUML, and Eclipse [23]. In intra-release models, their results were similar to those of Zimmerman and Nagappan whereby network metrics perform better than code metrics. However, when they used these metrics in a different setup, such as cross-project, they found that network metrics provided no advantage over code metrics in terms of accuracy, further adding that code metrics were preferable because they are fewer, easier to collect, and faster to train models.

Zimmerman and Naggappan [24] carried out another study to predict post-release defects by using program dependencies. In that work, they did not consider any metrics derived from dependency data but studied only the actual dependencies. Using the dependency data for each binary file, they built a prediction model that took the dependency targets of a binary as input and determined whether a binary was defect prone or not as well as the probability of its proneness. They achieved the best result with Support Vector Machines with linear kernels. Most dependencies are decided in the design phase and by using the models created, a developer can make better decisions about design alternatives.

He et al. [27] demonstrated that simple classifiers such as Naive Bayes work well with a simplified metric set, and models for within-company defect prediction (WCDP) are better than those for cross-company defect prediction (CCDP).

In summary, it can be seen from the related work that useful information can be extracted from change logs to help in software fault prediction. Therefore, we decided to extract more change metrics and analyze the effect of several existing and new change metrics on software fault prediction.

3. Methodology

Our goal was to determine a set of software metrics to help build a fault prediction model that would outperform other fault prediction models, and demonstrate the effectiveness of the proposed model consisting of new change metrics in public datasets. Studies performed on the code and developer metrics [9,10,20,21] show that although they are very useful for building prediction models, there is still good potential for improvement. Network metrics are relatively new metrics and have been evaluated by a number of researchers [22,23]. Our study focuses on change metrics. Nagappan et al. [18] used change burst metrics to build models that yielded promising performances with Windows Vista and Windows Server 2003 datasets, but not so for the Eclipse project data. Moser et al. [15] compared the performance of code and change metrics on the Eclipse data and showed that these change metrics outperform code metrics. By considering these studies, we decided to make use of both code and change metrics.

Because large amount of information can be extracted from code repositories, we decided to analyze change logs to achieve better results and therefore make better decisions. In order to make this study repeatable and testable, the experiments were performed on public datasets for Eclipse versions 2.0, 2.1, and 3.0. High-performance classification algorithms, namely Random Forest, K-Nearest Neighbor, and Decision Tree have been used to build the prediction models. The classification model produces two classes, namely fault-prone and non-fault-prone. Table 1 shows the definitions for all the existing change metrics.

Table 1
Existing change metrics [15,16].

Metrics	Definition
COMMITS	Number of commits a file has gone through
REFACTORINGS	Number of times a file has been refactored
BUGFIXES	Number of times a file has been involved in bug fixing
AUTHORS	Number of distinct authors who made commits to the file
LOC-ADDED	Number of lines of code added to the file
MAX-LOC-ADDED	Maximum number of lines of code added for all commits
AVG-LOC-ADDED	Average lines of code added per commit
LOC-DELETED	Number of lines of code deleted from the file
MAX-LOC-ADDED	Maximum number of lines of code deleted for all commits
AVG-LOC-ADDED	Average lines of code deleted per commit
CODECHURN	Sum of all commits (added lines of code â deleted lines of code)
MAX-CODECHURN	Maximum CODECHURN for all commits
AVG-CODECHURN	Average CODECHURN per commit
MAX-CHANGESET	Maximum number of files committed together with the repository
AVG-CHANGESET	Average number of files committed together with the repository
WEIGHTED-AGE	Age of a file normalized by added lines of code

Table 2
New change metrics.

Metrics	Definition
LOC-WORKED-ON	Lines of code added plus lines of code deleted
MAX-LOC-WORKED-ON	Maximum LOC-WORKED-ON for all commits
AVG-LOC-WORKED-ON	Average LOC-WORKED-ON per commit
MAX-COMMITS	Maximum number of commits made per developer
AVG-COMMITS	Average number of commits made by each developer
MAX-LOC-ADDED	Maximum lines of code added by a developer
AVG-LOC-ADDED	Average lines of code added by each developer
MAX-LOC-DELETED	Maximum lines of code deleted by a developer
AVG-LOC-DELETED	Average lines of code deleted by each developer
MAX-LOC-WORKED	Maximum lines of code worked by a developer
AVG-LOC-WORKED	Average lines of code worked by each developer
MAX-CODECHURN	Maximum CODECHURN by a developer over all developers
AVG-CODECHURN	Average CODECHURN by each developer
MAX-CHANGESET	Maximum CHANGESET by a developer over all developers
AVG-CHANGESET	Average CHANGESET by each developer
MAX-TIME-DIFF	Max time difference between 2 consecutive commits of a file
MIN-TIME-DIFF	Min time difference between 2 consecutive commits of a file
AVG-TIME-DIFF	Avg. time difference between 2 consecutive commits of a file
SINGLE-COMMITS	Number of commits where the file was committed alone
RELATIVE-SINGLE-COMMITS	Single commits / Total number of commits

New change metrics were determined based on the four dimensions we identified namely, standard change metrics, developer-based change metrics, period-based change metrics, and uniqueness-based change metrics. We took into account all the change metrics that can be produced during the software development process. Each dimension provides different information regarding the relevant change. A set of new change metrics is introduced in Table 2, which can be categorized into four main classes:

- Standard: Lines of code worked (total, max, and avg). These metrics are similar to code churn and they must have a direct relationship with the defects. High values indicate the fault-proneness of the module.
- Developer-based: Commits per developer (max, avg), LOC added per developer (max, avg), LOC deleted per developer (max, avg), LOC worked on per developer (max, avg), code churn per developer (max, avg) and change set per developer (max, avg). These metrics are extracted for the developers instead of independent commits.
- Period-based: Time difference between commits (max, min, and avg). These metrics are motivated by the fact that, if a file undergoes a large number of commits in a short period of time, it has a higher probability of being fault-prone. Therefore, the smaller the time difference is, the higher the probability of being defective.
- Uniqueness-based: Single commits and relative single commits. These metrics are motivated by the fact that files undergo both unique changes and non-unique changes, and that unique changes have a greater tendency to demonstrate defects [28]. Therefore, if a file undergoes a commit where it is the only one being committed, the change is bound to be unique, and more single commits means that the file has undergone more unique changes. Hence, these metrics must have a direct effect on the defects.

The bug data for the Eclipse project is made publicly available by Zimmerman et al. [9]. The process of data collection begins with cloning the Eclipse GIT repositories from the link https://git.eclipse.org/c/, which has a GIT Tag for the specific

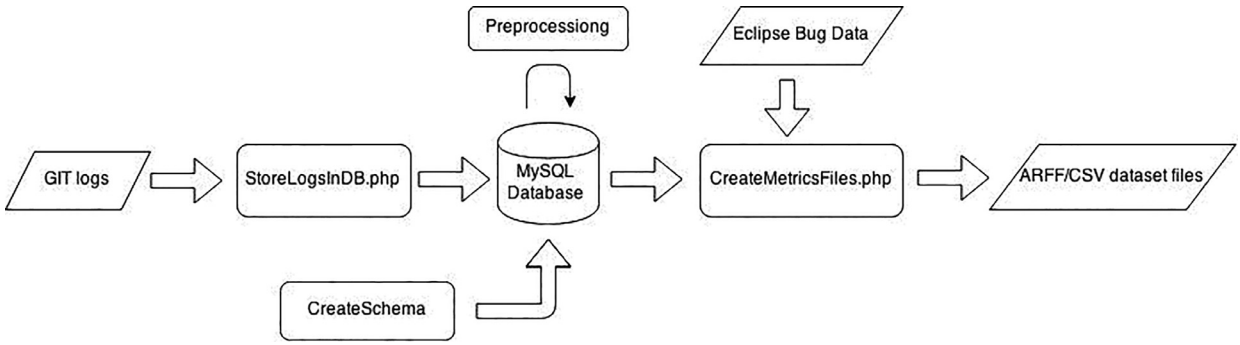

Fig. 1. Flowchart of the data collection.

version. A step-by-step procedure for creating the dataset from the GIT repositories is explained as follows and depicted in Fig. 1:

- Extracting GIT logs: This involves using the GIT log command to extract the logs.
- Creating database schema: This creates a schema for MySQL to store the GIT logs so that metrics calculation becomes easier. The schema includes four tables, namely developers, files, commits, and commit-file-relation.
- Storing GIT logs in the database: GIT logs are stored in MySQL tables. A PHP script is used in this step.
- Pre-processing of data: Pre-processing of the data is required, such as cleaning the developer data because developers sometimes add their user-id in place of their name and at other times, they add both user-id and name separately, creating two separate entries for the same developer.
- Generating dataset files: We calculate the metrics and create ARFF/CSV dataset files.

To select the most relevant metrics in our experimentation, we performed correlation analysis using the Spearman correlation technique in the SPSS tool to determine the correlation between each metric and post-release defects. This helped us to select the best set of metrics (also known as features) before performing fault prediction.

As represented in Fig. 1, first GIT logs are extracted from the repositories and then the database schema is created to store the GIT logs. To do this, StoreLogsinDB.php file is executed. Later, Eclipse bug data are applied and CreateMetricsFiles.php file is run. After this step, datasets are ready for use. The following website contains the datasets and the other generated files. https://sites.google.com/site/sandeepkumariitr002/projects.

4. Experimental results

All the experiments were performed on the WEKA machine learning platform. The following classifiers were used: Decision Tree (J48), K-Nearest Neighbor (KNN), and Random Forest (RF). For the performance evaluation, the following metrics were applied: Precision (P), Recall (R) and F-Measure (F). Recall defines the percentage of defective files that a prediction model is able to predict as defective. Therefore, a higher value of recall is preferred and desirable because the cost of detecting and fixing defects in the later stages of the software development lifecycle increases substantially. In this study, precision, recall, and F-measure calculations were performed at the file level. Measurement of these parameters was performed using the following definitions:

Precision: In the context of this work, precision can be defined as the number of modules predicted correctly as faulty out of the total number of modules predicted as faulty. It is defined in Eq. (1) as follows:

$$Precision = Truepositive/(Falsepositive + Truepositive) \tag{1}$$

Recall: In the context of this work, recall can be defined as the total number of modules predicted correctly as faulty out of all the faulty modules in the given dataset. It is defined in Eq. (2) as follows:

$$Recall = Truepositive/(Falsenegative + Truepositive) \tag{2}$$

Four sets of metrics were investigated during the experiments:

- Static code metrics provided by Zimmerman et al. [9] were analyzed.
- All the existing change metrics explained in the previous section were investigated.
- New change metrics plus existing change metrics were evaluated.
- Static code metrics plus all the change metrics were examined.

All the experiments were performed using the 10-fold cross-validation technique. Two types of software fault prediction models were built to evaluate the metrics sets:

- Intra-version model: The prediction model is trained and tested on the data from the same version of Eclipse. Tables 3–5 show the performance results for Eclipse v2.0, v2.1, and v3.0, respectively.

Table 3
Classification results for Eclipse v2.0.

Metrics	J48 (P) (R) (F)	KNN (P) (R) (F)	RF (P) (R) (F)
Static code metrics	0.450 0.294 0.356	0.400 0.397 0.399	0.648 0.301 0.411
Existing change metrics	0.594 0.373 0.458	0.485 0.474 0.479	0.702 0.387 0.499
All the change metrics	0.574 0.394 0.467	0.496 0.488 0.492	0.717 0.382 0.499
Code plus all the change metrics	0.508 0.417 0.458	0.507 0.495 0.501	0.777 0.375 0.506

Table 4
Classification results for Eclipse v2.1.

Metrics	J48 (P) (R) (F)	KNN (P) (R) (F)	RF (P) (R) (F)
Static code metrics	0.407 0.160 0.229	0.268 0.242 0.254	0.571 0.136 0.220
Existing change metrics	0.556 0.255 0.349	0.338 0.320 0.329	0.596 0.215 0.316
All the change metrics	0.477 0.252 0.330	0.360 0.344 0.352	0.644 0.218 0.326
Code plus all the change metrics	0.416 0.320 0.362	0.343 0.323 0.333	0.701 0.184 0.292

Table 5
Classification results for Eclipse v3.0.

Metrics	J48 (P) (R) (F)	KNN (P) (R) (F)	RF (P) (R) (F)
Static code metrics	0.486 0.246 0.327	0.353 0.339 0.346	0.627 0.207 0.311
Existing change metrics	0.576 0.294 0.389	0.405 0.387 0.396	0.659 0.278 0.391
All the change metrics	0.519 0.317 0.393	0.423 0.397 0.409	0.664 0.270 0.384
Code plus all the change metrics	0.457 0.375 0.412	0.448 0.415 0.431	0.730 0.252 0.374

Table 6
Classification results trained using v2.0 and tested on v2.1.

Metrics	J48 (P) (R) (F)	KNN (P) (R) (F)	RF (P) (R) (F)
Static code metrics	0.320 0.218 0.260	0.263 0.323 0.290	0.312 0.254 0.280
Existing change metrics	0.491 0.127 0.201	0.253 0.136 0.177	0.493 0.116 0.188
All the change metrics	0.467 0.197 0.277	0.288 0.144 0.192	0.565 0.122 0.201
Code plus all the change metrics	0.365 0.243 0.292	0.295 0.210 0.246	0.542 0.175 0.264

Table 7
Classification results trained using v2.0 and tested on v3.0.

Metrics	J48 (P) (R) (F)	KNN (P) (R) (F)	RF (P) (R) (F)
Static code metrics	0.379 0.191 0.254	0.320 0.281 0.299	0.464 0.218 0.296
Existing change metrics	0.448 0.148 0.222	0.324 0.234 0.272	0.454 0.125 0.196
All the change metrics	0.444 0.172 0.248	0.346 0.248 0.289	0.581 0.110 0.185
Code plus all the change metrics	0.379 0.236 0.291	0.344 0.286 0.312	0.598 0.189 0.287

Table 8
Classification results trained using v2.1 and tested on v3.0.

Metrics	J48 (P) (R) (F)	KNN (P) (R) (F)	RF (P) (R) (F)
Static code metrics	0.538 0.105 0.175	0.341 0.214 0.263	0.473 0.118 0.189
Existing change metrics	0.392 0.246 0.303	0.298 0.183 0.227	0.529 0.152 0.236
All the change metrics	0.378 0.222 0.280	0.301 0.194 0.236	0.500 0.112 0.183
Code plus all the change metrics	0.312 0.237 0.269	0.300 0.220 0.254	0.564 0.106 0.178

- Cross-version model: The prediction model is trained using data from an older version to predict the defects in a newer version of Eclipse. This type of experiment represents a more practical scenario because when we apply the defect prediction model on a version, we use the data from the previous version. Table 6 presents the performance results when the model was trained using v2.0 and tested on v2.1. Table 7 shows the classification results when the model was trained using v2.0 and tested on v3.0. Table 8 presents the classification results when the model was trained using v2.1 and tested on v3.0.

According to Tables 3, 4, 5, we observed the following results:

- KNN and J48 classifiers perform best among all the classifiers tried with high recall and F-measure. Random Forest gives the highest precision but low recall.

- The classification model containing the new change metrics shows an improvement in the results in the J48 and KNN classifier compared with the one with the existing change metrics, i.e. 2–3 percentage increase in recall in most cases. Also the new metrics perform as well as the existing metrics in the Random Forest classifier.
- The improvement with the new change metrics over static code metrics is very significant, for instance, a 10% increase in both recall and precision.
- The combined model of code and change metrics shows a significant increase in recall in the J48 classifier, as high as 7% compared with change metrics and 16% compared with static code metrics in v2.1.
- The combined model outperforms static code metrics in all cases and change metrics in most cases.
- The combined model shows a significant increase in precision (as high as 77%) in the Random Forest classifier but with a small decrease in recall value.

The Tables 6, 7, 8 are related to the second type of analysis, called the cross-version model.
According to these tables, we observed the following results:

- In terms of recall, static code performs better than change metrics in most cases.
- The classification model with the new change metrics still performs better than the existing change metrics in most cases with increase in recall as high as 7% using the J48 classifier in the model trained using v2.0 and tested on v2.1.
- The new change metrics also display a good improvement in precision over existing change metrics as well as static code metrics in most cases.
- The combined model of static code and change metrics performs better than the individual models.
- The results are quite sporadic and there is no single model that performs better than the others in all cases. This prevents us from reaching a definitive conclusion.

Our responses to the research questions we identified are as follows:

- Answer to RQ1: Existing change metrics provide acceptable performance in software fault prediction when Eclipse project datasets are used.
- Answer to RQ2: Newly proposed metrics provide better performance than models based on existing change metrics on Eclipse project datasets.
- Answer to RQ3: The use of change metrics in conjunction with code metrics provides better performance than the models that have individual metrics set on Eclipse datasets.

5. Threats to validity

Experimental studies are associated with some potential threats for the validity of their results [29,30]. In this section, some of the possible threats to the validity are discussed. In this work, the dataset was accumulated from Eclipse GIT repositories. The bug data for Eclipse versions 2.0, 2.1, and 3.0 were made publicly available by Zimmerman et al. [9]. Therefore, open-source software projects have been analyzed in this work. However, the performance of the prediction model using the proposed metrics for systems developed in-house or in an industrial environment might be different than the current results.

In addition, the experiments in this study were performed on Eclipse project data. The results obtained for the proposed metrics are encouraging, but more work is needed before the results of the model can be generalized outside this context, such as on different types of software applications/domains. Furthermore, in this study, all the experiments were performed using binary classification models for software fault prediction, but the performance may vary for fault count classification models and fault severity models. Another threat might be the programming language. The software systems for which the fault data were collected in this work were developed in Java and experiments using other programming languages might provide different results.

6. Conclusion and future work

The results demonstrate that change metrics provide extra performance for fault predictors and they complement static code metrics for the prediction models. Newly introduced change metrics improve the performance of models and help to build high-performance fault predictors. The classification results show that the new metrics provide an important increase in recall, that is, approximately 10% over static code metrics and approximately 23% over existing change metrics. Even a small increase in the recall value is crucial in terms of the resources that can be saved in an organization. The suggested machine learning model using static code metrics and change metrics outperforms the model using only static code metrics.

Also, we observed that the Random Forest classifier provides better precision value, but the decision tree (J48) and KNN classifiers provide better recall and better overall F-measure. For the cross-project prediction models, the new change metrics provide an improvement over the existing change metrics, but the model using the combination of new change metrics and code metrics performs best in most cases. Therefore, it can be concluded that the new change metrics proposed in this work are promising and models built on these metrics provide better prediction results.

In order to obtain better generalization of the results presented in this work, we plan to analyze other kinds of software projects (i.e., industrial settings) and projects implemented in different programming languages. Furthermore, it would be interesting to build fault count models and fault severity models using the metrics and the models introduced in this study.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.compeleceng.2018.02.043.

References

[1] Radjenović D, Heričko M, Torkar R, Živkovič A. Software fault prediction metrics: a systematic literature review. Inf Softw Technol 2013;55(8):1397–418.
[2] Pandey AK, Goyal NK. Early software reliability prediction. Springer; 2015.
[3] Catal C, Diri B. A systematic review of software fault prediction studies. Expert Syst Appl 2009;36(4):7346–54.
[4] Madeyski L, Jureczko M. Which process metrics can significantly improve defect prediction models? an empirical study. Softw Qual J 2015;23(3):393–422.
[5] Catal C, Sevim U, Diri B. Metrics-driven software quality prediction without prior fault data. In: Electronic engineering and computing technology. Springer; 2010. p. 189–99.
[6] Catal C, Alan O, Balkan K. Class noise detection based on software metrics and roc curves. Inf Sci 2011;181(21):4867–77.
[7] Rathore SS, Kumar S. A decision tree regression based approach for the number of software faults prediction. ACM SIGSOFT Softw Eng Notes 2016;41(1):1–6.
[8] Caglayan B, Turhan B, Bener A, Habayeb M, Miransky A, Cialini E. Merits of organizational metrics in defect prediction: an industrial replication. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering, 2. IEEE; 2015. p. 89–98.
[9] Zimmermann T, Premraj R, Zeller A. Predicting defects for eclipse. Predictor models in software engineering, 2007. PROMISE'07: ICSE workshops 2007. International workshop on. IEEE; 2007. 9–9
[10] Zhang H. An investigation of the relationships between lines of code and defects. In: Software maintenance, 2009. ICSM 2009. IEEE international conference on. IEEE; 2009. p. 274–83.
[11] Zhou Y, Xu B, Leung H. On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. J Syst Softw 2010;83(4):660–74.
[12] Rodriguez D, Ruiz R, Riquelme JC, Harrison R. A study of subgroup discovery approaches for defect prediction. Inf Softw Technol 2013;55(10):1810–22.
[13] Zhou Y, Leung H. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. IEEE Trans Softw Eng 2006;32(10):771–89.
[14] Subramanyam R, Krishnan MS. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. IEEE Trans Softw Eng 2003;29(4):297–310.
[15] Moser R, Pedrycz W, Succi G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: 2008 ACM/IEEE 30th international conference on software engineering. IEEE; 2008. p. 181–90.
[16] Krishnan S, Strasburg C, Lutz RR, Goševa-Popstojanova K. Are change metrics good predictors for an evolving software product line?. In: Proceedings of the 7th international conference on predictive models in software engineering. ACM; 2011. p. 7.
[17] Bell RM, Ostrand TJ, Weyuker EJ. Does measuring code change improve fault prediction?. In: Proceedings of the 7th international conference on predictive models in software engineering. ACM; 2011. p. 2.
[18] Nagappan N, Zeller A, Zimmermann T, Herzig K, Murphy B. Change bursts as defect predictors. In: 2010 IEEE 21st international symposium on software reliability engineering. IEEE; 2010. p. 309–18.
[19] Shin Y, Meneely A, Williams L, Osborne JA. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Trans Softw Eng 2011;37(6):772–87.
[20] Bell RM, Ostrand TJ, Weyuker EJ. The limited impact of individual developer data on software defect prediction. Emp Softw Eng 2013;18(3):478–505.
[21] Matsumoto S, Kamei Y, Monden A, Matsumoto K-i, Nakamura M. An analysis of developer metrics for fault prediction. In: Proceedings of the 6th international conference on predictive models in software engineering. ACM; 2010. p. 18.
[22] Zimmermann T, Nagappan N. Predicting defects using network analysis on dependency graphs. In: Proceedings of the 30th international conference on Software engineering. ACM; 2008. p. 531–40.
[23] Premraj R, Herzig K. Network versus code metrics to predict defects: a replication study. In: 2011 international symposium on empirical software engineering and measurement. IEEE; 2011. p. 215–24.
[24] Zimmermann T, Nagappan N. Predicting defects with program dependencies. In: Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement. IEEE Computer Society; 2009. p. 435–8.
[25] Rathore SS, Kumar S. An empirical study of some software fault prediction techniques for the number of faults prediction. Soft Comput 2016:1–18.
[26] Rathore SS, Kumar S. Linear and non-linear heterogeneous ensemble methods to predict the number of faults in software systems. Knowl-Based Syst 2017;119:232–56.
[27] He P, Li B, Liu X, Chen J, Ma Y. An empirical study on software defect prediction with a simplified metric set. Inf Softw Technol 2015;59:170–90.
[28] Ray B, Nagappan M, Bird C, Nagappan N, Zimmermann T. The uniqueness of changes: characteristics and applications. In: Proceedings of the 12th working conference on mining software repositories. IEEE Press; 2015. p. 34–44.
[29] Javanmardi S, Shojafar M, Shariatmadari S, Ahrabi SS. Fr trust: a fuzzy reputation–based model for trust management in semantic p2p grids. Int J Grid Utility Comput 2014;6(1):57–66.
[30] Chiaraviglio L, Blefari-Melazzi N, Canali C, Cuomo F, Lancellotti R, Shojafar M. A measurement-based analysis of temperature variations introduced by power management on commodity hardware. In: Transparent optical networks (ICTON), 2017 19th international conference on. IEEE; 2017. p. 1–4.

Garvit Rajesh Choudhary works at Google Inc., Mountain View California, USA. He received an Integrated Dual Degree, BTech in Computer Science and MTech in Information Technology from the Indian Institute of Technology Roorkee in 2015. His research interests include software fault prediction, database management systems, and cloud networking.

Sandeep Kumar works at the Computer Science and Engineering Department, Indian Institute of Technology Roorkee, India. He has supervised three PhD theses. He is a senior member of IEEE. He has received a Young Faculty Research Fellowship-MeitY and NSF/TCPP early adopter award. His research interests include semantic web and software engineering.

Kuldeep Kumar works at the Department of Computer Science and Engineering, National Institute of Technology Jalandhar, India. He received his PhD degree in Computer Science from the National University of Singapore in 2016. Prior to joining the institute, he worked for 2 years in the Birla Institute of Technology and Science, Pilani, India.

Alok Mishra works at the Department of Software Engineering in Atilim University in Turkey. He is visiting research chair in University of Technology of Compiegne (Sorbonne Universities) in France. He is also adjunct professor in GlobalNxt University in Malaysia and visiting professor in Belgrade Metropolitan University. His research interests are software engineering and information systems.

Cagatay Catal works at the Information Technology Group in Wageningen University in the Netherlands. He received BS & MSc degrees in Istanbul Technical University and a PhD degree in Yildiz Technical University in Istanbul. He worked for 6 years in Istanbul Kultur University. Before joining the university, he worked for 8 years at the Scientific and Technological Research Council of Turkey.