IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received January 14, 2021, accepted January 22, 2021, date of publication January 27, 2021, date of current version February 2, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3054948

Improving Software Defect Prediction by Aggregated Change Metrics

LUCIJA ŠIKIĆ, PETAR AFRIĆ, ADRIAN SATJA KURDIJA, (Member, IEEE), AND MARIN ŠILIĆ, (Member, IEEE)
Faculty of Electrical Engineering and Computing, University of Zagreb, 10000 Zagreb, Croatia
Corresponding author: Lucija Šikić (lucija.sikic@fer.hr)

This work was supported in part by the Croatian Science Foundation through the Reliable Composite Applications Based on Web Services (HRZZ-IP-01-2018-6423) research project, and in part by the European Regional Development Fund under Grant KK.01.2.1.01.0111 (OperOSS).

ABSTRACT To ensure the delivery of high quality software, it is necessary to ensure that all of its artifacts function properly, which is usually done by performing appropriate tests with limited resources. It is therefore desirable to identify defective artifacts so that they can be corrected before the testing process. So far, researchers have proposed various predictive models for this purpose. Such models are typically trained on data representing previous project versions of a software and then used to predict which of the software artifacts in the new version are likely to be defective. However, the data representing a software project usually consists of measurable properties of the project or its modules, and leaves out information about the timeline of the software development process. To fill this gap, we propose a new set of metrics, namely aggregated change metrics, which are created by aggregating the data of all changes made to the software between two versions, taking into account the chronological order of the changes. In experiments conducted on open source projects written in Java, we show that the stability and performance of commonly used classification models are improved by extending a feature set to include both measurable properties of the analyzed software and the aggregated change metrics.

INDEX TERMS Classification, feature engineering, process metrics, change metrics, software defect prediction.

I. INTRODUCTION

Defect-prone software modules can help optimize the allocation of test resources if they are identified before testing. This has motivated many researchers to develop models to identify defective modules and thereby reduce both the time and cost of software testing.

Depending on the amount and type of information about a software project, the models for identifying defective modules can be can be examined in one of the following contexts: within-project or cross-project. The Within-Project Defect Prediction (WPDP) models are created if defect data are available for some previous modules of the analyzed software project. These models are mainly trained on the previous project version and then used to predict defective modules in the current version, which is called cross-version defect prediction. However, if such data are not available or are insufficient, the Cross-Project Defect Prediction (CPDP) models can be developed instead by using knowledge from data collected by compiling known external projects [1]. In our study, we will consider cross-version defect prediction models, as many recent studies [2]–[5] have demonstrated the benefits of developing these particular models.

For the purpose of developing defect prediction models, researchers have derived features primarily from product or code metrics that reflect the static properties of the modules, rather than from process or change metrics that describe the process of software development between the versions analyzed. In fact, process metrics, which is often referred to as change metrics in the following context, are often used for just-in-time (JIT) defect prediction, which is fundamentally different from the module defect prediction considered in this paper because it is used to predict whether a code

The associate editor coordinating the review of this manuscript and approving it for publication was Lefei Zhang.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

change contains defects, rather than a module. But, even when used for module defect prediction, process metrics are usually calculated based on the differences between the same module from two analyzed versions of the project, ignoring information about how exactly the module was changed during the development steps. There are some previous studies [6]–[8] that proposed some process metrics derived from the changes made to a file between two project versions, but they did not include a chronology of changes in the definition of the proposed metrics, which could provide a more complete insight into the defect-proneness of the file. Under this assumption, and taking into account the significant results recently achieved in JIT defect prediction [9], [10], and the fact that combining product and process metrics in the development of a defect prediction model can improve its performance [11]–[13], we have developed features for representing software modules that use both the chronological order and the content of module changes. In addition to the reasons mentioned above, the suspicion that traditional metric prediction models had reached a performance limit [14], [15] was another motive for creating the features.

The goal of our study was to evaluate whether and to what extent the features generated from the information about the changes of the modules between two versions can be useful to identify defect-prone software modules of the newer version. As part of this effort, a set of features has been defined. It has been modeled after the pattern of existing metrics for predicting defective changes, and consists of features that we have called *aggregated change metrics* because they are calculated by aggregating the changes made in the development process. Unlike the existing process metrics, which are calculated based on the final differences between the modules of the two versions analyzed, or which neglect the sequential order of the changes, or both, the proposed metrics take into account all changes made to the modules between these versions, as well as their chronological order, and thus reflect a complete overview of the development process.

In summary, this work provides the following contributions:

- Implementation of a framework for extracting all changes to the project modules made between the selected project versions of seven open source Java projects, which are commonly used in defect prediction studies. The framework supports the extraction of relevant data from the projects available on GitHub and SourceForge.
- A set of fourteen metrics that can be calculated for each software module using all changes made to the module between the versions analyzed. We show that the performance of different classifiers is improved when using features that consist of the proposed metrics and existing traditionally used software features that are considered relevant to defect-proneness according to standard feature selection methods.

The rest of the paper is structured as follows. Section II describes the background to predicting software defects, focusing on software metrics, feature selection methods and classifiers. Section III introduces related work on defect prediction using process metrics and software development information. Section IV describes the proposed metrics. The Section V assesses the impact of using such metrics alongside traditional metrics on the performance of commonly used classifiers and analyzes the results obtained. Section VI discusses the threats to validity. Conclusions are drawn in Section VII.

II. BACKGROUND

In general, the task of predicting defect-prone modules can be divided into three subtasks: creating or/and collecting a features for describing the software modules, deciding which subset of the features are most relevant to the defect-proneness of the modules, and choosing a suitable classifier. The first subtask is based on the available information on the software modules and is usually taken separately from the other two steps. The second subtask is performed before of within the third subtask, which depends on how the decision on the features to be used for describing the module is made. However, regardless of how the subtasks are performed, it is crucial to extract the features of software modules that provide insights into the nature of the classification problem under consideration.

This section introduces the whole development process of a defect prediction model. The metrics used for describing software modules are briefly described in Section II-A. Section II-B provides a comprehensive overview of the feature selection methods applicable to modeling the software defect prediction problem, with emphasis on the most commonly used methods. Section II-C presents an overview of the classifiers that have been developed to identify defect-prone software modules.

A. FEATURES FOR REPRESENTING SOFTWARE MODULES

Software modules are represented by features that are generated by qualitative or quantitative description of either the software or its development process. These features are derived from the software information, which is usually a collection of product, process, and project metrics that describe the artifacts of the software [16]. Product metrics describe the characteristics of the product such as size, complexity, and performance. Process metrics include the effectiveness of defect correction during development, the pattern of testing defect arrival, and the response time of the correction process. Project metrics describe project characteristics such as number of software developers, cost, and schedule.

Previous research on software defect prediction has tended to focus on product and process quality aspects that are expected to be strongly correlated with defect-proneness. Such aspects describe product quality, in-process quality, and maintenance quality [16]. Product quality refers to intrinsic product quality, which is usually measured by defect density rate and mean time to failure, and customer satisfaction, which can be determined by customer survey data. In-process quality is measured by evaluating the programming process,

which is estimated by analyzing defect arrival patterns. Maintenance quality metrics reflect the ability to fix software defects as quickly as possible and with excellent quality, which significantly improves customer satisfaction.

In the majority of the studies [2], [3], [17]–[21], product quality is quantified using manually designed features, e.g. Halstead metrics [16], the McCabe complexity measure [22], the Chidamber and Kemerer (CK) metrics [23], [24] and metrics for object-oriented design (MOOD) [25]. To measure process quality, researchers typically use the change metrics, code churn and developer metrics [5], [9], [26], [27]. In addition, it has been shown that the combination of product features and process metrics can be beneficial for defect prediction [11], [28], [29]. In addition, results from several studies have indicated that the use of some process metrics instead of some product metrics can be helpful in identifying the source code that is prone to defects [30], [31].

In addition to or instead of these features, many recent studies have used features that have been extracted directly from the source codes of the modules. In general, this process consists of applying a deep learning algorithm to a representation of a software source code in the form of an Abstract Syntax Tree (AST) or Control Flow Graph (CFG). Although the syntactic structure and semantic information of a source code can be captured by both of these forms, CFG has rarely been used [32]. This is probably because it shows all paths that can be traversed during code execution, ignoring the block structure of the code, which can be advantageous when integrating it into code features. Much research effort has been done to extract features from ASTs that represent software modules using deep neural networks, such as Convolutional Neural Networks [2], [33], Deep Belief Network [4], Long-Short Term Memory networks [34], [35], attention-based Recurrent Neural Network [36], and Transformer model [37].

B. FEATURE SUBSET SELECTION

Like many real-world problems, software defect prediction is generally modeled as a classification problem. To solve it successfully, it is preferable to find out which of the features describing software modules are relevant to the defect-proneness of modules. For this purpose, before or within the development of a classifier, researchers apply some of the feature subset selection techniques to the given feature set.

In general, feature subset selection methods can be divided into filters, wrappers, embedded, and hybrid methods [38]. The decision on which method to use is usually made based on to whether the classifier is given or not. In the case the classifier is not given, feature subset selection method is selected independently of the classifier; otherwise, the feature subset is selected based on the performance of the given classifier.

1) SELECTING FEATURE SUBSET AND CLASSIFIER INDEPENDENTLY

Before selecting a classifier suitable for the software defect prediction problem, it is recommended to perform some of the appropriate feature subset selection methods to discover the optimal subset of features and to remove the features that are not relevant to the target concept. Similarly, it is recommended to remove or reduce the multicollinearity from the existing feature set [39] to ultimately increase the legitimacy and predictive power of the classifier [40]. For this purpose, filter methods are usually applied.

Filter subset selection methods can reduce the runtime of classifier learning and lead to a more general concept [41]. Two categories of these methods can be distinguished: univariate and multivariate methods. In univariate feature filters, individual features are ranked by their scores on various statistical tests for their correlation to the target variable. Pearson's correlation coefficient or Spearman's rank correlation coefficient is usually used for this purpose. In multivariate feature filters the entire subset of features is evaluated according to certain criteria. For this purpose, Variable Inflation Factor (VIF) [42] and the Condition Number (CN) method [19], [43] are widely used in software defect prediction. However, recent research has shown that subsets of metrics generated by commonly used feature selection methods are often inconsistent and correlated, and should be avoided when interpreting defect models [45], [46]. In such cases, they suggest the use of an approach called AutoSpearman as an alternative to other feature selection methods because it achieved the highest consistency of subsets of metrics among training samples and attenuated correlated metrics in their experiment.

2) FEATURE SUBSET SELECTION AS PART OF BUILDING A CLASSIFIER

Feature subset selection can be done as part of the construction of a classifier. In this setting, it is applied to improve the performance of given classifier. Depending on whether it is performed before or during the training of the classifier, it can be done with wrappers or embedded methods.

Wrappers evaluate feature subsets based on the performance of the chosen classifier. In previous work on software defect prediction, features were typically selected as classifiers with Support Vector Machine (SVM) [47], [48], C4.5 [49], [50], Random Forest [51] and Artificial Neural Network (ANN) [52]. The embedded methods do not separate learning from the feature selection part, but perform feature selection as part of the model building process. Previously, regularization [53] and BlogReg algorithm [54] were used as embedded feature selection methods to find features that affect defect prediction in software modules.

Apart from these two groups, feature subset selection can be performed by some hybrid methods. Such methods usually consist of the filter method, which is used to obtain multiple candidate subsets, and the wrapper, which is used to find the best of the candidate subsets. Such a combination can improve the classification accuracy of pure filter methods and also reduce the processing time of pure wrappers. In the past, various hybrid techniques have been presented to select features for software defect prediction. Some of these

techniques are automatic hybrid search [55], hybrid selection method combining different feature sorting techniques [56], and cluster-based hybrid filter-wrapper feature selection [57].

C. SOFTWARE DEFECTS CLASSIFIERS

Finding a suitable classifier depends on the type and size of the features selected in the step described in Section II-A. In general, when software modules are represented by a relatively small number of hand-crafted features, these features are used to develop simpler classifiers in terms of a number of parameters, while in the case of a large number of features, which are usually extracted by neural networks, more complex classifiers have been developed. In both cases, the classifiers are trained using either search-based approaches, statistical methods or machine learning algorithms.

Because of their superior ability to deal with noisy data, their better tolerance of missing data, and their ability to search globally with less probability of returning the locally optimal solution, search-based approaches are considered to be well suited for software engineering problems [58], especially for the area of software defect prediction [59]. In fact, the applicability of these approaches to predicting software defects has been evaluated over the last decade, while most studies have focused on Artificial Immune Recognition Systems [60], [61], Ant Colony Optimization [62], [63], and Genetic Programming [64], [65].

Previous approaches to predicting software defects treated the classification problem as an empirical experiment and tried to predict defect-prone modules by applying some statistical approaches to data representing past experience [66], [67]. In more recent studies, however, machine learning has been used instead of statistical approaches to predict defective software modules. This is probably due to the fact that machine learning algorithms are distribution-free and more robust than statistical approaches because they rely less on assumptions about the data, such as normality and/or linearity [68], which may not necessarily be true for defect prediction data sets. Nevertheless, machine learning algorithms are widely used in predicting software defects and have proven to be very successful in this area of research. In particular, the logistic regression classifier has proven to be particularly suitable for identifying defect-prone software modules that are based either on hand-crafted features [47], [69] or on features that have been extracted using neural networks [2], [4], [34], [36]. In addition, Naïve Bayes [70], [71] and Bayesian Net [3], [28], [72] proved successful in identifying defect-prone software modules. Significant results in predicting software defects have also been achieved using the Decision Tree technique, such as C.45 [73], J48 [74], [75] and Random Forest [19], [19], [20], as a classifier. Similarly, a Support Vector Machine classifier has proven to be suitable for the task of predicting defective modules [21], [76]. Furthermore, a number of studies have successfully used Artificial Neural Networks to solve this task [17], [18], [52], [64]. However, the most significant classification results are obtained when neural networks are used for feature extraction rather than classification.

III. RELATED WORK

We present the current state of research regarding the prediction of defect modules in developing projects using process metrics in Section III-A. However, since the proposed metrics are calculated from the accumulated historical data of software development, in Section III-B we review the most relevant existing proposals for using this type of data not only to predict defective software modules, but also to improve the overall software quality.

A. TYPES OF PROCESS METRICS

Many studies have so far focused on process metrics rather than product or code metrics when developing models to predict cross-version defects. In this context, they have proposed different process metrics that can be divided into the following three types: developer metrics, code change metrics and development process metrics [77].

When used to describe a software module, developer metrics are extracted for the developers who contribute to that specific software. Generally, these metrics are used to quantify the developers experience with the software projects or the details of their commit activity. However, although some studies emphasize that using developer metrics to describe software modules has negligible impact on the success of the defect prediction process [30], [78], others claim that some of these metrics can be useful for identifying defective software modules [79]–[81]. A more explicit contradiction is observed between studies that refer to the relationship between the developer's experience and the defective module [82], [83]. Nevertheless, some of the developer-related metrics, such as a number of developers who modified the software module [5], [29], [78], [84], and a number of developer commits that changed the module source code [79], [85], are commonly used to predict software defects.

Changes made to source code during software development can be quantified in the form of code change metrics based on data collected throughout the software lifecycle across successive versions. Such metrics have proven useful for identifying defect-prone modules [7], [86], especially code churn measures [30], [81], change bursts [11], [87], and code deltas [29], [30], [88]. In most studies of software defect prediction, the code change metrics representing the module of a project are calculated using differences between the source code of two successive versions of the module [6], [89], [90], but it can also be calculated taking into account changes from the several continuous versions of the module [8], [86], [87], [91].

Development process metrics describe the evolution of a software project. Many of these metrics have been shown to be related to defect-proneness, specifically the number of revisions [5], [30], [92] and the number of refactorings [6], [7] of a particular software module during development of the version of the project to which the module

belongs. Similarly, a frequency of changes [93], a number of historical defects [5], [29], a number of repaired defects [6], and a code change complexity [94] can be indicators for a defective software module.

For the purposes of this paper, we extracted the following process metrics to represent the software module of the analyzed project: the number of commits that included the module, the number of developers that modified the module, the geometric mean of the experience of all developers that modified the module, the number of lines that were added to the module, and the number of lines that were deleted from the module. All of these metrics come from previous research and have proven to be good indicators of software defects [6], [7], [92].

B. HISTORICAL SEQUENCES OF SOFTWARE METRICS

In the following, the most important research work on how the inclusion of historical information on software metrics can be used to evaluate software quality and identify defective software modules is presented.

Moser *et al.* [7] have made a significant contribution to the development of process metrics using the commit history of a project. In the experiment conducted with the three successive versions from the Eclipse project data set, they have shown that the performance of defect prediction models improves when they are built using process metrics instead of or in conjunction with code metrics. From the set of seventeen process metrics they defined, the following metrics have been emerged as features with the highest predictive power for defect-proneness: a maximum or average number of files committed together, a number of revisions of a file, a number of times a file has been refactored and a number of times a file has been involved in bug-fixing.

A similar research on the same data set was conducted by Choudhary *et al.* [6], where they defined and extracted twenty new code and change metrics. The extracted metrics are based on a number of lines of code for commits, a number of lines of code a developer has worked on, a time difference between commits to a module, and so on. Based on the experiment, in which they measured the impact of both the existing change metrics defined by Moser *et al.* [7] and the extracted metrics on software defect prediction, the extracted metrics provide additional power for defect predictors.

In their work [8], Rhmann *et al.* used a set of code change metrics describing modules of the Android project to create commonly used models for predicting software defects. The set consists of the following metrics: added, deleted, or changed lines of code, maximum number of these three values for all commits, and total, maximum, and average code churn for a software module considering all its versions. Depending on the results obtained, using commit data from more than one previous project version can improve the performance of the models created.

Liu *et al.* [91] have recently pointed out that the existing models for predicting software defects are created using software metrics that describe only the information between two adjacent project versions. Assuming that information about how software modules change during project development is important for defect prediction, they have developed a new predictor that instead uses features created by combining module metrics from multiple consecutive versions. In experiments they used data from PROMISE projects, whose modules are described using code and process metrics.

Gradišnik *et al.* [24] have shown that the performance of software maintenance prediction can be improved by using a history of changes in software metrics over time rather than relying on measurements of individual versions of a software product. Their study focused on CK metric changes between subsequent versions of the analyzed open source projects, taken from the Maven Repository.[1]

In an experiment conducted on twelve evolving Java projects from the publicly available data set, Madeyski and Jureczko [5] examined which of the following process metrics are correlated with the number of defects in a software module: a number of revisions, a number of distinct committers, a number of modified lines, and a number of defects in a previous version. According to the results obtained, despite the fact that a number of defects and a number of defects in the previous version are strongly correlated with the number of defects in a module, a number of distinct committers and a number of modified lines have proven to be the most useful metrics in terms of defect prediction models.

Using the same data set, Jiang *et al.* [77] investigated which process metrics are significantly important for altering defects. In their study, the following process metrics were considered: a number of revisions, a number of distinct committers, a number of modified lines, a degree of code modification, and an average number of modified lines. Based on the results of the study, they concluded that a number of distinct committers and a number of revisions play an important role in changing the defect state.

As can be seen from the current state of research on the inclusion of historical information in the defect prediction of evolving projects, there is still work to be done that focuses on extracting process metrics from a commit history to predict software modules. This is also supported by the fact that, taking into account all of the above-mentioned studies that deal exclusively with the prediction of defect-prone software modules using the commit history, only in the study [5] conducted by Madeyski and Jureczko was more than one project used in the experiment.

Since recent research [24], [77] has shown that the quality of a software module can be estimated based on the commit history of a project, we decided to extract more process metrics and analyze their impact on the performance of software defect predictors. The proposed metrics of the same name are proposed by Madeyski and Jureczko [5], but they are not designed to identify defect-prone modules, but to detect defective commits, and therefore differ in their formulas. However, since such a set of metrics has often been selected

[1] https://mvnrepository.com/

by experts in previous research [27], we have decided to adapt it to describe software modules, not commits. While there were some process metrics in earlier research [6], [7] that were derived from a project's commit history, these metrics do not take into account a chronology of commits and changes across commits, which we strongly believe should be used to create features that better describe the project's development process. Also, to the best of our knowledge, this is the first set of metrics representing software modules from the PROMISE data which takes into account any commit that changes the module between two adjacent project versions.

IV. PROPOSED SET OF METRICS

In this section we describe a process of collecting and calculating metrics that can improve the performance of predictive models for identifying defect-prone software modules or components. The process consists of two steps: collecting change history of software modules and calculating aggregated change metrics. In the first stage, which is described in Section IV-A, the software project's development history data are extracted in numerical form. The data so extracted are then used to calculate the proposed metrics for each module in the project, which is described in Section IV-B.

A. COLLECTING CHANGE HISTORY OF SOFTWARE PROJECT

The replacement of a traditionally used static code metric by the process metric for creating defect prediction models has gained increasing popularity in the recent past [5]. This is not surprising, considering that in several recent studies [7], [12], [92], [95] defect prediction models based on process metrics have performed better than static code metrics. However, unlike product metrics, which are widely and publicly available or have good tool support, process metrics are sometimes difficult to collect [5].

Process metrics reflect the history of project development, which provides additional information that can be used to improve the prediction of software defects. Such information is usually extracted using a Version Control System (VCS), such as CVS, SVN, or GIT, that stores details of every change made to the project's source code.

In most cases, both process and product metrics have version-duration. They are usually extracted at the class level from two adjacent versions of the analyzed project, and thus reflect only the final differences between the versions. More comprehensive information about the development and quality of the project can be provided by calculating such metrics for each commit made between the analyzed versions. Such information is generally used by just-in-time prediction models, which exploit the properties of a commit to predict whether it will introduce a defect. We believe, however, that it can also be used to create the models for identifying defect-prone modules that are introduced between two versions.

However, since there was no explicit commit history, we had to extract all the necessary commits from GitHub and SourceForge. There is Commit Guru [96], a web application that extracts the commits from a GitHub project and calculates change metrics defined by Kamei *et al.* [10] for each commit. Unfortunately, it does not meet our needs for two reasons. First, the jEdit project we analyze in this paper does not exist on GitHub, so we cannot extract its commit history using Commit Guru. Second, we need to capture all changes in the source code that have been committed between specific versions of the project: not on a specific branch, but on all existing branches. These obstacles have motivated us to develop the proposed framework, which can be easily adapted to collect the commit history of projects other than those analyzed in this paper. Since very few of the existing studies have contributed to defect prediction in the form of a universal approach that allows new metrics to be extracted from the existing commit history regardless of the project, we consider the framework to be an important research contribution.

B. CALCULATING AGGREGATED CHANGE METRICS

Unlike previous work on predicting defective software modules, where process metrics were calculated either based on either the final differences between the analyzed versions or using all commits between the versions without considering the chronological order of these commits, or both, we integrated both the content and the chronological order of all commits made to the source code between the two project versions into the proposed metrics. We believe that such defined metrics provide a more nuanced picture of both the development process and the quality of the software than existing process metrics. With this in mind, we have defined a set of fourteen metrics that can be derived from existing, commonly used class-level change metrics [10], whose notation we have adopted by adding an arrow representing a timeline to indicate that our metrics take into account a chronology of commits. The proposed metrics, which we have referred to as *aggregated change metrics*, indicating how they are calculated, are described below.

1) AGGREGATED NUMBER OF SUBSYSTEMS ($\overrightarrow{NS}$)

The metric $\overrightarrow{\mathrm{NS}}$ is an average number of different subsystems that are modified before each commit that changed the file. It can be calculated for a file F by using (1), where ss_k represents a number of different subsystems that were modified *in all commits before* the commit k, and where C_F represents the number of commits that changed the file between the analyzed project versions.

$$\overrightarrow{\mathrm{NS}} = \frac{1}{C_F}\sum_{k=1}^{C_F} ss_k \tag{1}$$

2) AGGREGATED NUMBER OF DIRECTORIES($\overrightarrow{ND}$)

The Aggregated Number of Directories is an average number of different directories that have been modified before each commit that changed the file. If d_k denotes a number of different directories that were changed *in all commits before* the commit k, and if C_F represents the number of commits

that changed the file between the analyzed project versions, the metric $\overrightarrow{\mathrm{ND}}$ can be calculated for a file F with (2).

$$\overrightarrow{\mathrm{ND}} = \frac{1}{C_F}\sum_{k=1}^{C_F} d_k \tag{2}$$

3) AGGREGATED NUMBER OF FILES ($\overrightarrow{\mathrm{NF}}$)

The Aggregated Number of Files refers to an average number of different directories that were modified along with the file before each commit that modified that particular file. For a file F, it can be calculated using (3), where f_k refers to a number of different files that were changed along with the file *in all commits before* the commit k and where C_F represents the number of commits that changed the file between the analyzed project versions.

$$\overrightarrow{\mathrm{NF}} = \frac{1}{C_F}\sum_{k=1}^{C_F} f_k \tag{3}$$

4) AGGREGATED ENTROPY ($\overrightarrow{\mathrm{ENT}}$)

As a change metric, entropy represents a distribution of modified code in a commit. To define it for an individual file F, we have aggregated the entropy values for each commit that changes the file between the analyzed project versions. In particular, if the file F has been changed in C_F commits between the analyzed versions, the metric $\overrightarrow{\mathrm{ENT}}$ can be calculated for a file F using the recursive formula (4) up to $n = C_F$, where $f_{k,i}$ denotes a portion of the commit k that modifies the file f_i (with $\sum_i^n f_{k,i} = 1$ for each commit k that modifies n files).

$$\begin{aligned}\overrightarrow{\mathrm{ENT}}(F,1) &= \sum_{i=1}^{n} f_{1,i}\log_2 f_{1,i}\\ \overrightarrow{\mathrm{ENT}}(F,n) &= \frac{1}{2}\Big(\overrightarrow{\mathrm{ENT}}(F,n-1) + \sum_{i=1}^{n} f_{n,i}\log_2 f_{n,i}\Big)\end{aligned} \tag{4}$$

5) AGGREGATED LINES ADDED ($\overrightarrow{\mathrm{LA}}$)

The metric $\overrightarrow{\mathrm{LA}}$ is calculated by averaging the aggregated averages of the number of lines of code added to the file according to the formula (5), where la_k represents a number of lines of code added to the file in the commit k. For a file F that was changed in C_F commits, the metric value is equal to the value of $\overrightarrow{\mathrm{LA}}(F, C_F)$.

$$\begin{aligned}\overrightarrow{\mathrm{LA}}(F,1) &= la_1\\ \overrightarrow{\mathrm{LA}}(F,n) &= \frac{1}{2}\Big(\overrightarrow{\mathrm{LA}}(F,n-1) + la_n\Big)\end{aligned} \tag{5}$$

6) AGGREGATED LINES DELETED ($\overrightarrow{\mathrm{LD}}$)

For a file F modified in C_F commits, the metric Aggregated Lines Deleted can be calculated with the formula (6) up to $n = C_F$, where ld_k represents a number of lines of code that were deleted from the file in the commit k.

$$\begin{aligned}\overrightarrow{\mathrm{LD}}(F,1) &= ld_1\\ \overrightarrow{\mathrm{LD}}(F,n) &= \frac{1}{2}\Big(\overrightarrow{\mathrm{LD}}(F,n-1) + ld_n\Big)\end{aligned} \tag{6}$$

7) AGGREGATED LINES BEFORE ($\overrightarrow{\mathrm{LT}}$)

The metric $\overrightarrow{\mathrm{LT}}$ reflects a custom average of the number of lines of code in the file before a particular commit for each commit that changes a file. If C_F denotes the number of commits that include the file F, and if lt_k represents a number of lines of code in the file before the commit k, it can be calculated using the recursive formula (7) up to $n = C_F$.

$$\begin{aligned}\overrightarrow{\mathrm{LT}}(F,1) &= lt_1\\ \overrightarrow{\mathrm{LT}}(F,n) &= \frac{1}{2}\Big(\overrightarrow{\mathrm{LT}}(F,n-1) + lt_n\Big)\end{aligned} \tag{7}$$

8) AGGREGATED BUG FIX ($\overrightarrow{\mathrm{FIX}}$)

For a commit k, a metric Bug Fix bf_k takes the value 1 if it is a defect-fix, and 0 otherwise. To calculate it for an individual file F in C_F commits, we suggest using the formula (8) up to $n = C_F$.

$$\begin{aligned}\overrightarrow{\mathrm{FIX}}(F,1) &= bf_1\\ \overrightarrow{\mathrm{FIX}}(F,n) &= \frac{1}{2}\Big(\overrightarrow{\mathrm{FIX}}(F,n-1) + bf_n\Big)\end{aligned} \tag{8}$$

However, to calculate bf_k for a commit k, it should be defined when a commit is considered a fix and when it is not. The categorization of this type depends on the way commit messages are written, so it can be defined in different, but probably similar, ways. Nevertheless, for the purposes of this paper, we have defined defect-fix commits as those whose message contains at least one word consisting of one of the following root words: "defect", "fix", "proper", "work", "issue", "closed", "problem".

9) AGGREGATED NUMBER OF DEVELOPERS ($\overrightarrow{\mathrm{NDEV}}$)

The Aggregated Number of Developers is an average number of different developers who modified a file before each commit that changed the file. If $ndev_k$ represents a number of different developers who modified the file *in all commits before* the commit k, and if C_F represents the number of commits that modified the file between the analyzed project versions, the metric $\overrightarrow{\mathrm{NDEV}}$ can be calculated for a file F using (9).

$$\overrightarrow{\mathrm{NDEV}} = \frac{1}{C_F}\sum_{k=1}^{C_F} ndev_k \tag{9}$$

10) AGGREGATED TIME INTERVAL BETWEEN COMMITS ($\overrightarrow{\mathrm{AGE}}$)

For a file F modified in C_F commits, the metric $\overrightarrow{\mathrm{AGE}}$ can be calculated with the formula (10) up to $n = C_F$, where age_k represents a number of days that have elapsed between the $(k-1)$th and kth commit.

$$\begin{aligned}\overrightarrow{\mathrm{AGE}}(F,1) &= 0\\ \overrightarrow{\mathrm{AGE}}(F,n) &= \frac{1}{2}\Big(\overrightarrow{\mathrm{AGE}}(F,n-1) + age_n\Big)\end{aligned} \tag{10}$$

11) AGGREGATED NUMBER OF UNIQUE CHANGES ($\overrightarrow{\text{NUC}}$)

The metric $\overrightarrow{\text{NUC}}$ is an average number of different commits made before each commit that changed the file, with the commits differing in a set of files that they change. It can be calculated for a file F by using (11), where nuc_k denotes a number of different commits *in all commits before* the commit k and where C_F represents a total number of commits that have changed the file F between the project versions analyzed.

$$\overrightarrow{\text{NUC}} = \frac{1}{C_F}\sum_{k=1}^{C_F} nuc_k \tag{11}$$

12) AGGREGATED EXPERIENCE ($\overrightarrow{\text{EXP}}$)[2]

When calculated for a developer and a commit k, Experience refers to the number of commits the developer made before the commit k. To calculate it for an individual file F, we have define $\overrightarrow{\text{EXP}}(F, k)$ for each commit k of C_F commits that changed the file F between the analyzed project versions as the average of $\overline{exp_{F,k}}$, which is the average experience of all developers who modified the file F before the commit k, and $\overrightarrow{\text{EXP}}(F, k-1)$. More specifically, the Aggregated Experience can be calculated using (12).

$$\begin{aligned} &\overrightarrow{\text{EXP}}(F, 1) = 0 \\ &\overrightarrow{\text{EXP}}(F, n) = \frac{1}{2}\left(\overrightarrow{\text{EXP}}(F, n-1) + \overline{exp_{F,n}}\right) \end{aligned} \tag{12}$$

13) AGGREGATED RECENT EXPERIENCE ($\overrightarrow{\text{REXP}}$)

[2] For a developer and a commit k, $\overrightarrow{\text{EXP}}$ represents the recent experience of the developer before the commit k. If cy is the current year, fy is the year of the developer's first commit, and c_y is the number of commits made by the developer in the year y, it can be calculated using (13).

$$\text{REXP} = \sum_{y=fy}^{cy} \frac{c_y}{cy-y+1} \tag{13}$$

Using the average recent experience of all the developers that have changed the file F before the commit k, referred to as $\overline{rexp_{F,k}}$, a total number of commits that changed the file F, referred to as C_F, then the metric Aggregated Recent Experience can be calculated using the recursive formula (14) up to $n = C_F$.

$$\begin{aligned} &\overrightarrow{\text{REXP}}(F, 1) = 0 \\ &\overrightarrow{\text{REXP}}(F, n) = \frac{1}{2}\left(\overrightarrow{\text{REXP}}(F, n-1) + \overline{rexp_{F,n}}\right) \end{aligned} \tag{14}$$

14) AGGREGATED EXPERIENCE ON THE SUBSYSTEM ($\overrightarrow{\text{SEXP}}$)[2]

If it is calculated for one developer and a commit k, Developer's Experience on the Subsystem is the number of commits the developer made before the commit k and changing the file(s) belonging to that subsystem. For a file F, a number of commits that change that file C_F, and the average experience on the file's subsystem of all developers who changed the file F before the commit k, reffered to as $\overline{sexp_{F,k}}$ it can be calculated with (15) up to $n = C_F$.

$$\begin{aligned} &\overrightarrow{\text{SEXP}}(F, 1) = 0 \\ &\overrightarrow{\text{SEXP}}(F, n) = \frac{1}{2}\left(\overrightarrow{\text{SEXP}}(F, n-1) + \overline{sexp_{F,n}}\right) \end{aligned} \tag{15}$$

The aggregated change metrics for a file in the nth project version are computed by aggregating process metrics that describe commits that changed the file between the nth and $(n+1)$th versions of the project. In other words, to represent the file from the current project version using the proposed metrics, it is necessary to collect all the commits between the current version and the previous version of the file and then aggregate each of the proposed process metrics extracted from each of the collected commits. Accordingly, to compute the proposed metrics for source files from the software version considered as the training set, it is necessary to extract all the commits made between that version and the previous version. That is, when a model is trained on source files from the nth project version to identify defective source files from the $(n+1)$th project version, the aggregated change metrics for source files from the nth project version were calculated using data from all commits made between the $(n-1)$th and the nth project versions.

The aggregation itself can be described with Figure 1, which illustrates a sequence of commits made between the nth and $(n+1)$th versions of a project whose files are represented by a series of squares `file_1`, `file_2`, ..., `file_j`. For example, considering only commits that are shown in the Figure 1, and with a white square representing a file that has not been changed within a commit that points to it, only a set of change metrics for the commit `commit_2` will not be used to calculate the aggregated change metrics for the `file_3`.

As can be seen from the formulas (1)-(13), (14) and (15), the aggregation process can be carried out in two ways, depending on what a specific metric measures. First, an aggregated change metric can be calculated by averaging values representing sizes of a set that is updated after each commit, as in the formulas (1)-(3), (9), and (11). In this way, we integrate the chronological order of commits implicitly by tracking the amounts of changes across commits. Second, it can be calculated by using a recursive function that calculates the average of its value for the previous commit and the change metric for the current commit. Such function is used in the formulas (4)-(8), (10), (12), (14) and (15).

In this first setting, the amounts of unique changes related to a change metric are tracked across the analyzed commits by updating a set that represents the change metric. By calculating an average of the tracked amounts, we obtain a value that gives a clearer insight into the file development process than the final size of the set representing the analyzed

[2] Only commits within the analyzed versions of the project were taken into account when calculating developer metrics.

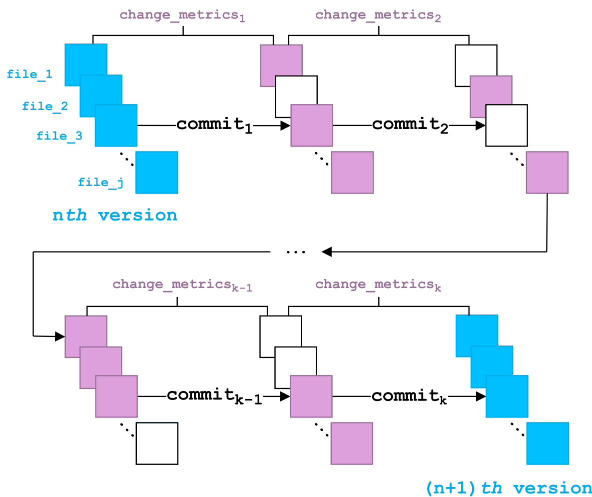


FIGURE 1. **Process of collecting process metrics from commits made between two successive versions of the project. The aggregated change metrics are calculated using the process metrics values from all *k* commits.**

metric. For example, assume that a file was changed by three developers in ten commits between two versions, so that the first seven commits were made by the same developer, the eighth and ninth by the second developer, and the last by the third developer. In this case, the value of the process metric NDEV is 3, since it represents the total number of developers who modified the file between versions. On the other hand, the value of the aggregated change metric $\overrightarrow{\text{NDEV}}$ is 1.4, because the values of the accumulated number of developers across commits are represented by the sequence 1, 1, 1, 1, 1, 1, 1, 2, 2, 3. In this particular case, the majority of commits are performed by only one developer, and since the value of 1.4 is closer to the value of 1 than to the value of 3, it can be concluded that the aggregated change metric $\overrightarrow{\text{NDEV}}$ describes the development of the file more adequately than the process metric NDEV.

The second setting uses a recursive formula to aggregate change metrics that are not represented by the size of some set, which is the case in the first setting. Some previous studies [6]–[8], [77] have defined some process metrics as an average, minimum or maximum of values for some metric over commits. However, they ignore the chronological order of the commits, which we believe should be considered. For example, assume that a file was changed in four commits, with the second commit made two days after the first commit, the third commit made forty-eight days after the second commit, and the fourth commit made one day after the third commit. Then the value of $\overrightarrow{\text{AGE}}$, which is $0.5(0.5(1 + 45) + 9) = 16$, is closer to the exact number of days that have passed between the third and fourth commits, which is 9, than an average value ($\text{avg}\{1, 45, 9\} = 18.33$), a maximum value ($\max\{1, 45, 9\} = 45$), or a minimum value ($\min\{1, 45, 9\} = 1$) of the values for the metric AGE in the analyzed commits.

V. EVALUATION

In this section, we conducted two experiments to evaluate the effectiveness of using the proposed aggregated change metrics as features for representing source code of software modules. In the first experiment we analyzed the quality of the proposed metrics. In particular, we examined which of the proposed metrics show a strong correlation with the defect label. Since the proposed metrics are used together with the traditional features in the development of prediction models in the second experiment, the correlation coefficients between the traditional features and the defect label are also calculated. As part of the first experiment, we have also investigated whether multicollinearity is present in the set of features that are most strongly correlated with the defect label. In the second experiment, we investigated whether and to what extent the performance of commonly used classifiers can be improved by using the proposed metrics together with the traditional features to detect defect-prone source code. Furthermore, since the predictive power of some of the commonly used classifiers can be impaired if there is multicollinearity between the features, the features describing the project modules used in the experiments are tested for multicollinearity before a model is trained.

In particular, our study addresses the following research questions (RQ):

RQ1: Which subset of the aggregated change metrics is the most correlated with the defect-proneness?

RQ2: Can the use of aggregated change metrics as features improve the performance of commonly used classifiers?

We have developed Python scripts to download source code files and extract the information to calculate aggregated change metrics from these files. In case features are selected independently from the classifier, both traditional and proposed metrics used as features have been evaluated using methods from the Python libraries pandas and statsmodels. In contrast, features where selection is part of the creation of the classifier are selected using scikit-learn implementations of selection methods. Regardless of how the features were selected, the classification models were created using scikit-learn modules. All experiments were run on a NVIDIA GeForce Titan Xp GPU with RAM of 12 GB. The data sets used and the experimental results are available at GitLab.[3]

Section V-A describes the experimental setup and Section V-B defines evaluation measures used in the experiments. In Sections V-C and V-D we answer the research questions.

A. EXPERIMENTAL SETUP

To conduct the experiments, we collected historical data for projects from the commonly used PROMISE data set. Using the existing features describing modules from these projects and the proposed metrics, we developed defect prediction models and measured their performance to investigate

[3] https://gitlab.com/LSikic/sdp-using-acm

TABLE 1. Description of 20 traditional features and 5 process metrics.

Traditional features		
Symbol	**Name**	**Description**
WMC	Weighted Methods per Class	The number of methods in the class
DIT	Depth of Inheritance Tree	The number of steps from the class node to the root of the inheritance tree
NOC	Number of Children	The number of immediate descendants of the class
CBO	Coupling Between Object classes	The number of classes to which it is coupled and vice versa
RFC	Response for a Class	Sum of the number of methods called within the class' methods and the number of the class' methods
LCOM	Lack of Cohesion in Methods	Number of pairs of methods that do not share a reference to an instance variable
LCOM3	Lack of Cohesion in Methods (different formula from LCOM)	If m is the number of methods, a the number of attributes in the class and $\mu(a)$ the number accessing an attribute, then $LCOM3 = (\frac{1}{a}\sum_i^a \mu(a_j) - m)/(1-m)$
NPM	Number of Public Methods	The number of all the methods in a class declared as being public
DAM	Data Access Metric	Ratio of the number of private (protected) attributes to the total number of attributes
MOA	Measure of Aggregation	The number of data declarations (class fields) whose types are user defined classes
MFA	Measure of Function Abstraction	Number of methods inherited by a class increased by a number of methods accessible by member methods of the class
CAM	Cohesion among Methods of a Class	Sum of the number of different types of method parameters in every method divided by a product of the number of different method parameter types in the whole class and the number of methods
IC	Inheritance Coupling	The number of parent classes to which a given class is coupled
CBM	Coupling between Methods	Total number of new/redefined methods to which all inherited methods are coupled
AMC	Average Method Complexity	The number of Java byte codes
CA	Afferent Couplings	How many other classes use the specific class
CE	Efferent Coupling	How many other classes are used by the specific class
MAX_{CC}	Maximum McCabe	Maximum McCabe's cyclomatic complexity values of methods in the same class
AVG_{CC}	Average McCabe	Average McCabe's cyclomatic complexity values of methods in the same class
LOC	Lines of Code	Measures the volume of the code
Process metrics		
Symbol	**Name**	**Description**
ADD	Lines Added	The added lines in class normalized by total number of added and deleted lines
DEL	Lines Deleted	The added lines in class normalized by total number of added and deleted lines
ADEV	Number of Developers	The of developers who changed the class
COMM	Number of Commits	The number of commits made to the class
GEXP	Developers' experience	The geometric mean of the experiences of all the developers that changed the class

the usefulness of the proposed metrics. In the following, we describe the data set used and the process of developing the models.

1) PROMISE DATA SET

The data set used for this research consists of projects developed with the Java programming language. The set consists of projects collected from PROMISE,[4] a publicly accessible repository of research data for software defect prediction, which has been widely used in previous studies [2]–[4], [51], [72]. It contains information about classes from various Java Apache project versions. Specifically, each project version is represented by a list of the classes it consists of, and each class is described by 20 traditional features, such as Lines of Code (LOC) and Weighted Methods per Class (WMC), and the defect label. In addition, we have calculated five process metrics, which are used in previous research [92], from the commit history data extracted for the purpose of the research. As with the *state-of-the-art* approaches [2], [4], [34], we used seven projects from PROMISE in the experiments. Detailed information about the projects used can be found in Tables 2 and 1. Table 2 shows the number of files and the defect rate for each version of the projects, while Table 1 includes a description of the traditional features and process metrics used to describe the project modules.

Data collection for the aggregated change metrics was carried out using a framework developed for this research. Using the framework, we downloaded the relevant versions of the project repositories, which are available on the software platforms GitHub and SourceForge. From the downloaded data, we extracted the commits between project versions, which were then processed and used to calculate fourteen aggregated change metrics.

To evaluate the proposed metrics, we followed the standard approach and extracted features used to develop a classifier from two consecutive versions of each project, with the

[4]https://github.com/opensciences/opensciences.github.io

TABLE 2. PROMISE data set description.

Project	Version	#Files	Defect rate [%]
camel	1.4	872	16.63
	1.6	965	19.48
jedit	4.0	306	22.60
	4.1	312	25.32
lucene	2.0	195	46.67
	2.2	247	58.30
poi	2.5	385	64.42
	3.0	442	63.57
synapse	1.1	222	27.03
	1.2	256	33.59
xalan	2.5	803	48.19
	2.6	885	46.44
xerces	1.2	440	16.14
	1.3	453	15.23

training set from the older version and the test set from the newer version of the project. Finally, each class file presented in the project version that has changed from the previous version is represented either by a defect label and a feature vector of length 34, generated by concatenating the traditional features with the aggregated change metrics, or by a defect label and a feature vector of length 25, generated by concatenating the traditional features with process metrics. Accordingly, we have omitted the files that do not meet these criteria from the experiments, since the values of the process metrics cannot be calculated for such files.

Given the great inconsistency of prediction results between existing software defect prediction models [97], we repeated our experiment 30 times to verify the results. In this way, we reduced the likelihood of errors or anomalous performance results and obtained a more reliable experiment.

2) SOFTWARE DEFECT PREDICTION MODEL DEVELOPMENT

For completeness, we investigated whether the prediction of defect-prone software modules can be improved using the aggregated change metrics, regardless of whether the prediction model is chosen before or after the feature selection step. We used both approaches in our experiments, which showed that the proposed metrics add value in both cases.

Under the first approach, features are selected using AutoSpearman [45], an automated approach to feature selection that first selects the uncorrelated metrics based on a Spearman rank correlation test and then selects the uncorrelated metrics from the resulting subset of metrics based on a VIF analysis. In this way, both correlation and multicollinearity are detected and removed from the feature set. Using the features selected in this way, we have developed an ensemble model for detecting defect-prone software modules, as such a model is likely to perform best in defect prediction [97]. The developed model uses soft voting to classify software modules as defective or not defective, i.e., it identifies a software module as defective if the sum of the probabilities of its classifiers that the module is defective is greater than the sum of the probabilities of its classifiers that the module is not defective. Using the soft voting setting may result in higher model performance than using a hard voting system, where the class that received the most votes from its classifiers is selected as the ensemble predictor because it gives more weight to the votes with high confidence [98].

In the second case, i.e., when feature selection is part of the construction of a classifier, it can be done using wrappers or embedded methods. In this work, the wrapper method is based on the AUC score of an ensemble classifier, while the embedded method uses a random forest performance for feature selection. To determine whether the individual performance of some of the most commonly used classifiers in predicting software defects can be improved by using the proposed metrics in conjunction with existing traditional features, we also used support vector machine, decision tree, and multi-layer perceptron classifiers as part of the wrapper-based approach.

For the wrapper method, we used bidirectional feature elimination. It is similar to the forward feature selection, which starts from an empty feature set and adds a new feature in each iteration. To determine which of the remaining features to select in the current iteration, the classifier is trained separately for each remaining feature on the set of previously selected features that will be augmented by that feature. The feature whose inclusion resulted in the largest increase in classifier performance is selected. In each epoch of bidirectional feature elimination, the described selection is followed by an elimination step. More specifically, after adding a new feature, the importance of all features from the current feature set is checked. If the classifier performs better when trained on the current set without the unimportant features, these features are removed.

Embedded feature selection with a random forest classifier is based on the weighted impurity in a tree, a measure of how much the tree overfits the training data. During the selection process, the importance of each feature is calculated as a measure of how much it reduces the weighted impurity in a tree. The calculated reductions are ranked over all features in the tree and averaged over all trees in a random forest [99]. Finally, the highest ranked features are selected for training a classification model.

B. EVALUATION MEASURES

In order to assess whether and to what extent the proposed metrics are relevant for use in the construction of defect prediction models, in the first experiment we calculated Spearman's correlation coefficient between the defect-proneness of a file and each feature from the set of proposed and traditional features. It is a non-parametric measure of the statistical dependence between the ranks of two variables that makes no assumptions about the distribution of variable values or the linearity of the association between variable values.

In this experiment we also investigated whether there is a correlation between different features. For this purpose we used either Pearson's correlation coefficient or Spearman's correlation coefficient. In particular, we have calculated the Pearson's correlation coefficient between features for whose

values it can be concluded by Shapiro-Wilk's test with 95% certainty that both are normally distributed; otherwise, we have calculated the Spearman's correlation coefficient.

To evaluate the performance of the model for predicting software defects based on a specific set of features, researchers typically use Precision, Recall, F1 score and Area Under the ROC Curve (AUC). However, given the lack of standardized metrics and the existence of a class imbalance in the data set used in this research, we have reported AUC, as suggested by Jiang *et al.* [100]. It has a lower variance, i.e. it is more static than any of the above metrics, and is therefore highly preferable for the evaluation of defect prediction models. In addition, we have calculated the Matthews Correlation Coefficient (MCC) because it is least affected by imbalanced data [101] and gives a more informative and truthful score when evaluating binary classifications than F1 score [102]. Furthermore, we have reported the F1 score together with AUC and MCC in Section V-D, since this measure has been reported in many previous studies (e.g. [2], [4]).

The AUC is based on the area under the ROC (Receiver Operating Characteristic) and can be used to evaluate how well the developed model can distinguish between defect class and non-defect class. A model whose predictions are completely wrong has an AUC of 0, while a model whose predictions are completely correct has an AUC of 1.

The F1 score is the harmonic mean of precision and recall. It is a widely used measure of the accuracy of the test, with values between 0 for the worst accuracy and 1 for the best accuracy.

The MCC is a correlation coefficient between the observed and predicted classifications. Its value ranges from -1 to 1, with a value of 1 representing a perfect prediction, 0 as no better than a random prediction, and -1 as the worst possible prediction.

C. WHICH SUBSET OF THE AGGREGATED CHANGE METRICS IS THE MOST RELEVANT TO THE DEFECT LABEL? (RQ1)

In this experiment we investigate which of the proposed metrics are relevant for use in the construction of defect prediction models by measuring the Spearman's correlation coefficient on the training set consisting of concatenated data from each PROMISE project.

The correlation coefficient, hereinafter referred to as ρ, is calculated between each metric from the set of fourteen proposed metrics and the defect label. The strength of the correlation was estimated according to the absolute value of the correlation $|\rho|$, as suggested by Evans [103], as follows: very weak if $0.0 \leq |\rho| \leq 0.19$, weak if $0.2 \leq |\rho| \leq 0.39$, moderate if $0.4 \leq |\rho| \leq 0.59$, strong if $0.6 \leq |\rho| \leq 0.79$, and very strong if $0.8 \leq |\rho| \leq 1.0$. In order to ensure the credibility of our results, we have assessed the statistical significance of each calculated coefficient at the level of 0.05 and have only considered those with a p-value of less than 5%. In this way, we are 95% confident that only significant correlations are reported. In order to compare the proposed metrics and the traditional features in terms of correlation with defect-proneness, we calculated the correlation between traditional features and process metrics and the defect label as well. The calculated coefficients are shown in Table 3.

TABLE 3. The correlation coefficient (ρ) between traditional features (TF) and defect label, between aggregated change metrics (ACM) and defect label, and between process metrics (PM) and defect label on the projects from PROMISE data set.

TF	ρ
NPM	0.26
LOC	0.26
WMC	0.26
RFC	0.23
LCOM	0.22
CBM	0.20
MAX_{CC}	0.15
AMC	0.14
IC	0.14
MOA	0.10
CE	0.08
AVG_{CC}	0.07
DIT	0.07
CBO	0.07
CA	0.05
CAM	-0.13

ACM	ρ
$\overrightarrow{LT}$	0.36
$\overrightarrow{NF}$	0.25
$\overrightarrow{NS}$	0.23
$\overrightarrow{LD}$	0.23
$\overrightarrow{ND}$	0.22
$\overrightarrow{AGE}$	0.21
$\overrightarrow{NUC}$	0.18
$\overrightarrow{NDEV}$	0.14
$\overrightarrow{LA}$	0.07
$\overrightarrow{FIX}$	-0.06
$\overrightarrow{SEXP}$	-0.09
$\overrightarrow{EXP}$	-0.13
$\overrightarrow{REXP}$	-0.17
$\overrightarrow{ENT}$	-0.21

PM	ρ
COMM	0.18
DEL	0.14
ADEV	0.13
ADD	-0.14
GEXP	-0.15

Based on the results on the PROMISE data set presented in Table 3, we can be 95% certain that there is a positive correlation between the defect-proneness and nine of the thirteen aggregated change metrics marked as ACM. At $\rho = 0.36$, we found that the metric ($\overrightarrow{LT}$) is the most correlated with a defect label when all three sets of metrics are considered. In terms of correlation, this metric has outperformed the second ranked metric, which is the traditional feature NPM, by 0.1. Furthermore, there is a weak correlation between a file defect-proneness and the metric $\overrightarrow{NF}$. Similar results were obtained with $\rho = 0.23$ by metrics $\overrightarrow{NS}$ and $\overrightarrow{LD}$, with $\rho = 0.22$ by metric $\overrightarrow{ND}$, and with $\rho = 0.21$ by metric $\overrightarrow{AGE}$. Moreover, the results show the metrics reflecting aggregated numbers of subsystems and directories that are changed along with the file, the accumulated number of lines of code deleted from the file, and the aggregated time interval between the commits are weakly correlated with the defect-proneness of the file. Finally, with the exception of the metric $\overrightarrow{ENT}$, which was found to be weakly but negatively correlated with a file's defect label, other aggregated change metrics were found to be very weakly correlated with the defect label.

A narrower range of degrees of strength of correlation was observed between the traditional features, marked as TF in Table 3, and the defect label as for between the aggregated change metrics and the defect label. As can be seen from Table 3, fifteen traditional features proved to be positively correlated with the file defect-proneness. Looking only at these features, metrics NPM, LOC, and WMC ($\rho = 0.26$) proved to be the features that are most correlated with a defect label of a file. Moreover, magnitudes of the correlation coefficient ρ of 0.23 and 0.22 indicate the existence of a weak correlation between the defect label and the traditional

features RFC and LCOM, in the order given. A weak correlation was also found between the feature CBM and the file defect-proneness. In addition, a very weak correlation was found between each of the rest of the traditional features and the defect-proneness of the file. With a range of correlation coefficient values between -0.15 and 0.18, process metrics, marked as PM in Table 3, have been shown to be very weakly correlated to the file's defect label.

Some of the commonly used classifiers are sensitive to multicollinearity between features, which means that their performance could be harmed if there is a correlation between the features. To investigate the existence of a correlation between features describing software modules, which will help us to select a suitable value for a parameter of the algorithm used in the second experiment, we calculated the correlation coefficient between traditional features, process metrics, and proposed metrics that proved to be at least weakly correlated with the defect label. More specifically, according to the criteria mentioned in Section V-B, the correlation coefficient is calculated for each pair from a set of features whose correlation, either positive or negative, with the defect label is assessed by ρ greater than 0.2. As shown in Table 3, the set consists of the traditional features NPM, LOC, WMC, RFC, LCOM, and CMB, and the proposed metrics $\overrightarrow{LT}$, $\overrightarrow{NF}$, $\overrightarrow{NS}$, $\overrightarrow{LD}$, $\overrightarrow{ND}$, $\overrightarrow{AGE}$, and $\overrightarrow{ENT}$. The obtained coefficients' values are represented in a form of the table in Figure 2.

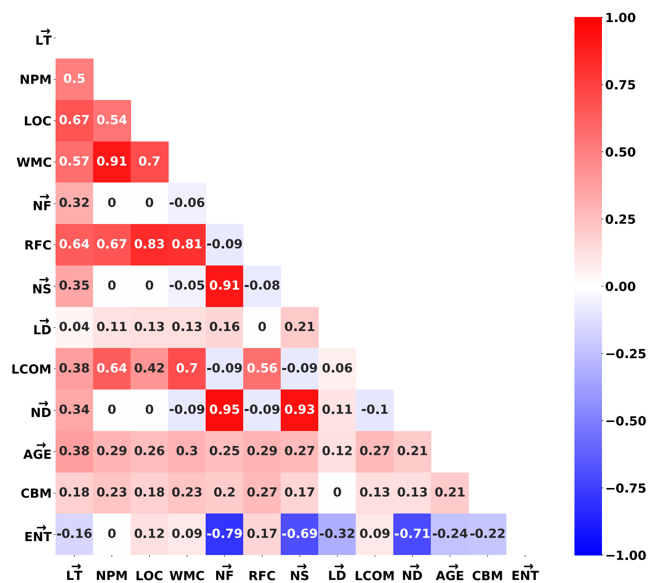


FIGURE 2. The correlation coefficients between features whose correlation coefficient relating the defect label was greater than 0.2.

As can be seen from the result in Figure 2, a very high correlation is measured between the proposed metrics $\overrightarrow{NF}$ and $\overrightarrow{ND}$ ($\rho = 0.95$), between the proposed metrics $\overrightarrow{NS}$ and $\overrightarrow{ND}$ ($\rho = 0.93$), between the proposed metrics $\overrightarrow{NF}$ and $\overrightarrow{NS}$ ($\rho = 0.91$), and between the traditional features WMC and NPM ($\rho = 0.91$). Such high correlation values between pairs from a set $\{\overrightarrow{NF}, \overrightarrow{ND}, \overrightarrow{NS}\}$ are expected because an increase in the aggregated number of modified subsystems presupposes an increase in the aggregated number of modified directories, whose increase requires an increase in the aggregated number of modified files. A slightly lower values, yet still classified as 'very strong', are detected between traditional features LOC and RFC (0.83), and between RFC and WMC (0.81).

In summary, the proposed metric $\overrightarrow{LT}$, are the most positively correlated with a defect label when features from all three sets of metrics are considered. This is an expected result of this experiment, as a larger module contributes more defects [104]. In addition, the results of the experiment show that as the number of files, directories, and subsystems modified within commits containing a file increases, the likelihood of a defect being introduced into the file increases, which is supported by existing research [105]. In addition to these metrics, the proposed metric $\overrightarrow{LD}$ has also been shown to be positively correlated with the file's defect-proneness, which is a reasonable result assuming that changes to a file that involve deleting a large amount of its code could indicate that the file is defect-prone. This is also supported by the fact that the large number of code changes, i.e., code churn, is related to the file's defect-proneness [87]. Moreover, experimental results have shown that the likelihood of a file being defective increases as the proposed metric $\overrightarrow{AGE}$ increases. Since the metric $\overrightarrow{AGE}$ places more weight on recent code changes, this result is consistent with previous research that concluded that recent changes contribute more defects than older changes [29]. Finally, the aggregated entropy has been shown to be weakly negatively correlated with the defect label of a file, which contradicts the result obtained for the metric Entropy used to classify commits as defective or non-defective, indicating that scattered changes are likely to introduce defects [94]. However, such a result should not necessarily be expected for the metric $\overrightarrow{ENT}$, since it does not consider just one commit, but all commits that modify a given file. In fact, a small value of aggregated entropy for a file might indicate not only that the file was modified within commits that have changed a small number of files, but also that the proportions of the file that was changed within commits are relatively large. Since relative code churn measures are a good indicator of defective modules [87], one might expect a small value of $\overrightarrow{ENT}$ for a file can be expected to indicate its defectiveness.

> According to the results obtained on the seven projects from the PROMISE data set, the most relevant aggregated change metric for the defect label is $\overrightarrow{LT}$. A significant correlation with the defect label was also observed between metrics $\overrightarrow{NF}$, $\overrightarrow{NS}$, $\overrightarrow{LD}$, $\overrightarrow{ND}$, $\overrightarrow{AGE}$, and $\overrightarrow{ENT}$.

D. CAN THE USE OF AGGREGATED CHANGE METRICS IMPROVE THE PERFORMANCE OF COMMONLY USED CLASSIFIERS? (RQ2)

To answer this research question, we compared the performance of a classifier created with traditional features

and process metrics with the performance of a classifier created with traditional features and aggregated change metrics. The comparison was performed in two different settings - when the features are selected independently of the classifier and when they are selected as part of the classifier building process using a wrapper or embedded feature selection method. In the first setting and as part of the wrapper method, we tested the performance of an ensemble of the following frequently used classifiers: Random Forrest Classifier (RFC), classifier based on linear discriminant analysis (LDAC), multi-layer perceptron classifier (MLPC) with one hidden layer, and support vector machine classifier (SVMC). The embedded method was based on the performance of a random forest classifier. Finally, we performed an additional experiment to investigate whether the classifiers commonly used in software defect prediction, which are listed in Section V-A2, could benefit from the use of the proposed metrics.

The hyper-parameters of the ensemble were tuned by a random search. To be precise, we tuned `gamma` and `C` for SVMC, `solver` for LDAC, `max_depth` and `n_estimators` for RFC and `hidden_layer_sizes` for MLPC by randomly examining the combinations of the values of these hyper-parameters within a set of ranges typical of each of them. The values of the other hyper-parameters for each classifier were set to the default values provided by scikit-learn. Specifically, for each project in the data set, the optimal hyper-parameters' values are selected as one of ten combinations of randomly selected hyper-parameters' values that give the ensemble the best performance in terms of AUC score on the training set over 5-fold cross validation. The optimal values for `max_depth` and `n_estimators`, the hyper-parameters' of RFC used within the embedded feature selection, were obtained in the same way. In addition, when examining the performance of an individual classifier, we tuned the hyper-parameters for MLPC and SVMC using the same procedure as for the same models within the ensemble. Similarly, we found optimal values for the decision tree classifier's (DTC) hyper-parameters `max_depth`, `max_features`, and `min_samples_leaf`. The exact ranges of values for the hyper-parameters we examined can be found in GitLab[3].

The other parameters of the experimental settings were chosen taking into account the results from Section V-D and with the aim of providing the models with sufficient features that contain information about modules that are relevant for the defect-proneness. Specifically, in the first setting the ensemble was built using the features selected by AutoSpearman with a threshold value of 0.6 for a Spearman rank test and a threshold value of 10 for a Variance Inflation Factor analysis. In addition to eliminating highly correlated features [106], the values were chosen to avoid serious multicollinearity problems [101]. In the second setting, we have set the number of features the model selects within both the wrapper and the embedded method to 20, which is the number of traditional features. This allows us to test not only whether the proposed metric can replace the process metrics, but also whether a classifier selects either some of the process metrics or some of the aggregated change metrics against some of the traditional features improve its performance.

The AUC, MCC, and F1 scores achieved by the models have for each project from the PROMISE data set, and the mean value of these scores, are listed in Tables 4-6. Each table compares the performance of a model using features selected from the set consisting of traditional features and process metrics (TPM), and the performance of a model using features selected from the set consisting of traditional features and proposed aggregated change metrics (TACM). Table 4 shows results obtained with the ensemble classifier built on features selected by AutoSpearman, Table 5 shows results obtained with the ensemble classifier built within the wrapper method for feature selection, while Table 6 shows results achieved by the RFC model built within the embedded method for feature selection.

In order to assess the statistical significance between the models, we conducted pair-wise tests for all performance measures. To do this, we firstly tested the normality of the obtained values of the measures by both models using the Shapiro-Wilk test with 95% confidence. In the case where the obtained values of the measures of both models follow a normal distribution, the statistical significance between the performance measures was tested with a paired t-test. Otherwise, the paired Wilcoxon test was performed. In both the t-test and the Wilcoxon test a p-value of less than 0.05 was assumed as significant. This means that the values of a performance measure obtained by two classifiers analyzed can only be interpreted as significantly different if the p-value is less than 0.05, in which case these classifiers can be considered to be different in terms of the performance. The decision on the difference of the classifiers with the corresponding p-value obtained in the test performed is reported in the column SD (which stands for "Significantly Different") of the Tables 4-6, where the value of "T" (True) indicates that the classifiers are different in terms of the performance analyzed, while the value of "F" (False) indicates that we cannot be 95% sure their performances are different.

The results presented in Tables 4-6 show that the TACM model in the embedded setting achieved the best overall results in all performance measurements for the projects of the PROMISE data set. Specifically, the RFC built within the embedded feature selection method using traditional and proposed features achieved an average AUC value of 63%, an average MCC value of 0.3 and an F1 score of 54%, i.e. it outperformed the RFC built using traditional features and process metrics by 2% for AUC, 4% for F1 and 0.07 for MCC. As can be seen from the last columns of Table 6, the difference in the performance of the models in terms of all evaluation measures can be considered significant for all projects, except for the *poi* project, where we cannot be 95% sure that the models perform equally in terms of AUC and MCC. Nevertheless, it is particularly important to highlight the improvement achieved by the model on *xerces* project, which suffers most from the problem of class imbalance,

TABLE 4. Performance comparison of the ensemble classifier based on features selected by AutoSpearman from traditional features and process metrics (TPM), and from traditional features and proposed aggregated metrics (TACM) on PROMISE.

AUC

Project	TPM	TACM	SD (p-value)
camel	0.55 ± 0.00	**0.56 ± 0.01**	**T** (< 0.001)
jedit	**0.68 ± 0.01**	0.63 ± 0.01	**T** (< 0.001)
lucene	0.49 ± 0.01	**0.52 ± 0.01**	**T** (< 0.001)
poi	0.49 ± 0.00	**0.64 ± 0.01**	**T** (< 0.001)
synapse	**0.55 ± 0.01**	**0.55 ± 0.02**	F (0.727)
xalan	0.54 ± 0.01	**0.60 ± 0.02**	**T** (< 0.001)
xerces	**0.51 ± 0.00**	**0.51 ± 0.00**	F (0.522)
Average	0.55 ± 0.00	**0.57 ± 0.01**	**5/7**

MCC

Project	TPM	TACM	SD (p-value)
camel	**0.23 ± 0.01**	0.18 ± 0.01	**T** (< 0.001)
jedit	**0.39 ± 0.02**	0.37 ± 0.01	**T** (0.001)
lucene	-0.03 ± 0.02	**0.06 ± 0.02**	**T** (< 0.001)
poi	-0.06 ± 0.01	**0.32 ± 0.01**	**T** (< 0.001)
synapse	**0.16 ± 0.02**	0.11 ± 0.05	**T** (< 0.001)
xalan	0.11 ± 0.01	**0.24 ± 0.02**	**T** (< 0.001)
xerces	0.10 ± 0.02	**0.13 ± 0.03**	**T** (< 0.001)
Average	0.13 ± 0.02	**0.20 ± 0.02**	**7/7**

F1

Project	TPM	TACM	SD (p-value)
camel	0.19 ± 0.02	**0.23 ± 0.02**	**T** (< 0.001)
jedit	**0.58 ± 0.03**	0.43 ± 0.03	**T** (< 0.001)
lucene	0.74 ± 0.01	**0.75 ± 0.01**	**T** (< 0.001)
poi	0.77 ± 0.00	**0.79 ± 0.00**	**T** (< 0.001)
synapse	0.28 ± 0.05	**0.39 ± 0.04**	**T** (< 0.001)
xalan	0.34 ± 0.03	**0.44 ± 0.06**	**T** (< 0.001)
xerces	**0.06 ± 0.01**	0.04 ± 0.02	**T** (< 0.001)
Average	0.42 ± 0.02	**0.44 ± 0.03**	**7/7**

TABLE 5. Performance comparison of the ensemble classifier within wrapper feature selection on a set of traditional features and process metrics (TPM), and on a set of traditional features and proposed aggregated metrics (TACM) on PROMISE.

AUC

Project	TPM	TACM	SD (p-value)
camel	**0.57 ± 0.01**	**0.57 ± 0.00**	**T** (0.009)
jedit	0.65 ± 0.01	**0.67 ± 0.01**	**T** (< 0.001)
lucene	0.49 ± 0.01	**0.54 ± 0.01**	**T** (< 0.001)
poi	0.55 ± 0.01	**0.63 ± 0.01**	**T** (< 0.001)
synapse	0.59 ± 0.01	**0.60 ± 0.01**	**T** (< 0.001)
xalan	0.62 ± 0.01	**0.63 ± 0.01**	**T** (0.001)
xerces	**0.52 ± 0.01**	0.51 ± 0.00	**T** (< 0.001)
Average	0.57 ± 0.01	**0.59 ± 0.01**	**7/7**

MCC

Project	TPM	TACM	SD (p-value)
camel	**0.27 ± 0.01**	0.26 ± 0.01	**T** (0.004)
jedit	0.39 ± 0.01	**0.47 ± 0.01**	**T** (< 0.001)
lucene	-0.02 ± 0.01	**0.13 ± 0.02**	**T** (< 0.001)
poi	0.16 ± 0.01	**0.25 ± 0.01**	**T** (< 0.001)
synapse	0.24 ± 0.01	**0.33 ± 0.01**	**T** (< 0.001)
xalan	0.25 ± 0.02	**0.29 ± 0.03**	**T** (< 0.001)
xerces	0.08 ± 0.02	**0.16 ± 0.00**	**T** (< 0.001)
Average	0.20 ± 0.01	**0.27 ± 0.01**	**7/7**

F1

Project	TPM	TACM	SD (p-value)
camel	**0.26 ± 0.02**	0.25 ± 0.01	**T** (0.012)
jedit	0.50 ± 0.02	**0.53 ± 0.02**	**T** (< 0.001)
lucene	0.70 ± 0.01	**0.76 ± 0.01**	**T** (< 0.001)
poi	**0.78 ± 0.01**	0.70 ± 0.03	**T** (< 0.001)
synapse	**0.36 ± 0.03**	0.34 ± 0.02	**T** (< 0.001)
xalan	**0.53 ± 0.02**	0.51 ± 0.04	**T** (0.008)
xerces	**0.10 ± 0.02**	0.06 ± 0.00	**T** (< 0.001)
Average	**0.46 ± 0.02**	0.45 ± 0.02	**7/7**

when looking at projects from the PROMISE data set. On this project, the proposed metrics allowed the model to improve its performance by 7% for AUC, 0.24 for MCC and 13% for F1. It follows that the proposed features can be useful in overcoming the class imbalance that is considered one of the major problems in predicting software defects [107].

When trained using traditional and proposed features, the ensemble classifier created within the wrapper method has a slightly worse results than the RFC developed within the embedded method using these features. Although the TPM model outperformed the TACM model in terms of average F1 score by 1% in this particular setting, the TACM model did achieve better average AUC and MCC scores by 2% and 0.07. Considering that MCC is a more appropriate performance measure for binary classifications than F1 score [102], the TACM model can be considered a winner in this setting.

When an ensemble is trained on a subset of features selected by AutoSpearman (Table 4), extending the set of traditional features with the proposed metrics rather than with the process metrics improved ensemble performance by 4% in terms of the average F1 score, 2% in terms of average AUC score and 0.07 in terms of average MCC score. In conclusion, when comparing the results from all three Tables 4-6, it can be observed that, when considering the projects analyzed, a filter-based feature selection may prevent the classifiers from fully exploiting the potential of the features and subsequently achieving better results.

To find out which of the proposed metrics contributed to this improvement, we extracted the proposed metrics that were most frequently selected as the 20 most important characteristics according to the RFC model for each of the seven PROMISE projects. The result is represented by a histogram in Figure 3.

As can be seen in Figure 3, nine of the aggregated change metrics were selected by the RFC model from the 20 most important features for more than half, i.e. for four out of seven PROMISE projects. It can be concluded that, apart from those that proved to be highly correlated with the defect label in Section V-C (e.g. $\overrightarrow{\text{LT}}$, $\overrightarrow{\text{ND}}$, $\overrightarrow{\text{NS}}$), several other aggregated change metrics, in particular the metric $\overrightarrow{\text{NDEV}}$, have also provided the RFC model with valuable information about the modules, which ultimately helped it to identify defective modules more successfully than when using traditional features and process metrics.

TABLE 6. Performance comparison of the random forest classifier within embedded feature selection on a set of traditional features and process metrics (TPM), and on a set of traditional features and proposed aggregated metrics (TACM) on PROMISE.

AUC

Project	TPM	TACM	SD (p-value)
camel	0.57 ± 0.01	$\mathbf{0.59 \pm 0.01}$	**T** (< 0.001)
jedit	0.69 ± 0.01	$\mathbf{0.71 \pm 0.01}$	**T** (< 0.001)
lucene	0.59 ± 0.01	$\mathbf{0.63 \pm 0.01}$	**T** (< 0.001)
poi	$\mathbf{0.62 \pm 0.01}$	$\mathbf{0.62 \pm 0.01}$	F (0.429)
synapse	0.60 ± 0.01	$\mathbf{0.61 \pm 0.01}$	**T** (< 0.001)
xalan	0.69 ± 0.01	$\mathbf{0.70 \pm 0.00}$	**T** (< 0.001)
xerces	0.50 ± 0.02	$\mathbf{0.57 \pm 0.01}$	**T** (< 0.001)
Average	0.61 ± 0.01	$\mathbf{0.63 \pm 0.01}$	**6/7**

MCC

Project	TPM	TACM	SD (p-value)
camel	0.18 ± 0.01	$\mathbf{0.24 \pm 0.01}$	**T** (< 0.001)
jedit	0.40 ± 0.02	$\mathbf{0.48 \pm 0.01}$	**T** (< 0.001)
lucene	0.18 ± 0.02	$\mathbf{0.25 \pm 0.01}$	**T** (< 0.001)
poi	$\mathbf{0.24 \pm 0.02}$	$\mathbf{0.24 \pm 0.01}$	F (0.929)
synapse	0.25 ± 0.01	$\mathbf{0.26 \pm 0.01}$	**T** (< 0.001)
xalan	0.38 ± 0.02	$\mathbf{0.41 \pm 0.01}$	**T** (< 0.001)
xerces	0.00 ± 0.03	$\mathbf{0.24 \pm 0.02}$	**T** (< 0.001)
Average	0.23 ± 0.02	$\mathbf{0.30 \pm 0.01}$	**6/7**

F1

Project	TPM	TACM	SD (p-value)
camel	0.30 ± 0.01	$\mathbf{0.33 \pm 0.01}$	**T** (< 0.001)
jedit	0.60 ± 0.02	$\mathbf{0.62 \pm 0.01}$	**T** (< 0.001)
lucene	0.68 ± 0.01	$\mathbf{0.70 \pm 0.01}$	**T** (< 0.001)
poi	$\mathbf{0.75 \pm 0.01}$	$\mathbf{0.75 \pm 0.01}$	**T** (< 0.001)
synapse	0.40 ± 0.01	$\mathbf{0.45 \pm 0.01}$	**T** (< 0.001)
xalan	0.65 ± 0.01	$\mathbf{0.69 \pm 0.01}$	**T** (< 0.001)
xerces	0.13 ± 0.01	$\mathbf{0.26 \pm 0.02}$	**T** (< 0.001)
Average	0.50 ± 0.01	$\mathbf{0.54 \pm 0.01}$	**7/7**

TABLE 7. Performance comparison of the SVMC within the wrapper feature selection on a set of traditional features and process metrics (TPM), and on a set of traditional features and proposed aggregated metrics (TACM) on PROMISE.

AUC

Project	TPM	TACM	SD (p-value)
camel	0.47 ± 0.02	$\mathbf{0.53 \pm 0.02}$	**T** (< 0.001)
jedit	0.49 ± 0.01	$\mathbf{0.63 \pm 0.06}$	**T** (< 0.001)
lucene	0.48 ± 0.03	$\mathbf{0.60 \pm 0.02}$	**T** (< 0.001)
poi	0.49 ± 0.01	$\mathbf{0.60 \pm 0.04}$	**T** (< 0.001)
synapse	0.40 ± 0.02	$\mathbf{0.62 \pm 0.03}$	**T** (< 0.001)
xalan	0.43 ± 0.02	$\mathbf{0.55 \pm 0.06}$	**T** (< 0.001)
xerces	0.49 ± 0.02	$\mathbf{0.59 \pm 0.03}$	**T** (< 0.001)
Average	0.46 ± 0.02	$\mathbf{0.59 \pm 0.04}$	**7/7**

MCC

Project	TPM	TACM	SD (p-value)
camel	-0.13 ± 0.02	$\mathbf{0.13 \pm 0.03}$	**T** (< 0.001)
jedit	-0.06 ± 0.06	$\mathbf{0.32 \pm 0.12}$	**T** (< 0.001)
lucene	-0.05 ± 0.07	$\mathbf{0.21 \pm 0.03}$	**T** (< 0.001)
poi	-0.08 ± 0.02	$\mathbf{0.23 \pm 0.07}$	**T** (< 0.001)
synapse	-0.24 ± 0.05	$\mathbf{0.27 \pm 0.08}$	**T** (< 0.001)
xalan	-0.27 ± 0.05	$\mathbf{0.12 \pm 0.22}$	**T** (< 0.001)
xerces	-0.10 ± 0.08	$\mathbf{0.22 \pm 0.02}$	**T** (< 0.001)
Average	-0.13 ± 0.05	$\mathbf{0.21 \pm 0.08}$	**7/7**

F1

Project	TPM	TACM	SD (p-value)
camel	$\mathbf{0.34 \pm 0.02}$	0.15 ± 0.09	**T** (< 0.001)
jedit	$\mathbf{0.49 \pm 0.13}$	0.43 ± 0.17	F (0.159)
lucene	$\mathbf{0.71 \pm 0.13}$	0.48 ± 0.06	**T** (< 0.001)
poi	$\mathbf{0.76 \pm 0.01}$	0.63 ± 0.19	**T** (< 0.001)
synapse	0.39 ± 0.03	$\mathbf{0.48 \pm 0.05}$	**T** (< 0.001)
xalan	$\mathbf{0.58 \pm 0.02}$	0.45 ± 0.16	**T** (< 0.001)
xerces	0.22 ± 0.06	$\mathbf{0.27 \pm 0.07}$	**T** (0.007)
Average	$\mathbf{0.50 \pm 0.06}$	0.41 ± 0.12	**6/7**

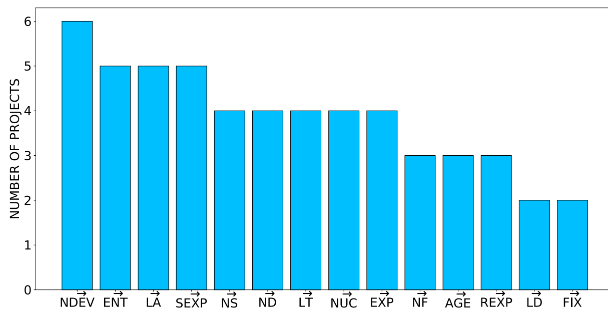


FIGURE 3. Number of PROMISE projects for which the proposed metrics were selected among 20 most relevant features.

Finally, it remains to be seen whether the proposed metrics can improve the performance of the individual classifiers. For this purpose, each of the individual classifiers is trained within the wrapper-based feature selection, using the same setting as for training an ensemble classifier. We decided to use the wrapper method because such method was considered to be well suited for PROMISE projects [50]. The results are shown in Table 7 for SVMC, in Table 8 for DTC, and in Table 9 for MLPC. The identifiers used in Tables 4-6, i.e. TPM and TACM, are also used in Tables 7-9.

As shown in Tables 7-9, the DTC achieved the highest average AUC, MCC and F1-scores on the PROMISE data set when trained using traditional features together with the proposed aggregated change metrics. It outperformed both SVMC and MLPC trained using the same set of features by 14% and 8% in terms of the average F1 score, 5% and 3% in terms of the average AUC score, and 0.07 and 0.02 in terms of the average MCC. The results presented in Table 8 also show that the DTC improved significantly in all performance measures for each PROMISE project, except for the F1-score for *lucene* project,[5] when the process metrics are replaced with the proposed aggregated metrics for its development. The replacement has resulted in a significant improvement in average AUC (from 0.52 to 0.64), F1 (from 43% to 55%) and MCC (from 0.04 to 0.28). As can be seen in the Table 9, such replacement also proved beneficial for the MLPC model. The

[5]Even if the DTC within the TACM model has reached a higher F1-score than the DTC within the TPM model on *lucene* project, we cannot be 95% certain that there is a significant difference between the F1-scores obtained from these two models for *lucene* project.

TABLE 8. Performance comparison of the DTC within the wrapper feature selection on a set of traditional features and process metrics (TPM), and on a set of traditional features and proposed aggregated metrics (TACM) on PROMISE.

AUC

Project	TPM	TACM	SD (p-value)
camel	0.56 ± 0.02	$\mathbf{0.61 \pm 0.01}$	**T** (< 0.001)
jedit	0.60 ± 0.02	$\mathbf{0.69 \pm 0.02}$	**T** (< 0.001)
lucene	0.53 ± 0.03	$\mathbf{0.62 \pm 0.01}$	**T** (< 0.001)
poi	0.45 ± 0.06	$\mathbf{0.68 \pm 0.02}$	**T** (< 0.001)
synapse	0.51 ± 0.03	$\mathbf{0.61 \pm 0.02}$	**T** (< 0.001)
xalan	0.56 ± 0.04	$\mathbf{0.67 \pm 0.01}$	**T** (< 0.001)
xerces	0.43 ± 0.07	$\mathbf{0.62 \pm 0.02}$	**T** (< 0.001)
Average	0.52 ± 0.04	$\mathbf{0.64 \pm 0.02}$	**7/7**

MCC

Project	TPM	TACM	SD (p-value)
camel	0.09 ± 0.03	$\mathbf{0.20 \pm 0.02}$	**T** (< 0.001)
jedit	0.20 ± 0.03	$\mathbf{0.37 \pm 0.03}$	**T** (< 0.001)
lucene	0.07 ± 0.05	$\mathbf{0.23 \pm 0.02}$	**T** (< 0.001)
poi	-0.10 ± 0.12	$\mathbf{0.36 \pm 0.04}$	**T** (< 0.001)
synapse	0.03 ± 0.06	$\mathbf{0.23 \pm 0.05}$	**T** (< 0.001)
xalan	0.12 ± 0.07	$\mathbf{0.35 \pm 0.02}$	**T** (< 0.001)
xerces	-0.10 ± 0.10	$\mathbf{0.23 \pm 0.04}$	**T** (< 0.001)
Average	0.04 ± 0.07	$\mathbf{0.28 \pm 0.03}$	**7/7**

F1

Project	TPM	TACM	SD (p-value)
camel	0.34 ± 0.02	$\mathbf{0.40 \pm 0.02}$	**T** (< 0.001)
jedit	0.51 ± 0.04	$\mathbf{0.61 \pm 0.02}$	**T** (< 0.001)
lucene	0.61 ± 0.09	$\mathbf{0.63 \pm 0.05}$	F (0.304)
poi	0.48 ± 0.14	$\mathbf{0.76 \pm 0.03}$	**T** (< 0.001)
synapse	0.37 ± 0.04	$\mathbf{0.48 \pm 0.03}$	**T** (< 0.001)
xalan	0.51 ± 0.07	$\mathbf{0.64 \pm 0.04}$	**T** (< 0.001)
xerces	0.17 ± 0.04	$\mathbf{0.34 \pm 0.04}$	**T** (< 0.001)
Average	0.43 ± 0.06	$\mathbf{0.55 \pm 0.03}$	**6/7**

TABLE 9. Performance comparison of the MLPC within the wrapper feature selection on a set of traditional features and process metrics (TPM), and on a set of traditional features and proposed aggregated metrics (TACM) on PROMISE.

AUC

Project	TPM	TACM	SD (p-value)
camel	0.54 ± 0.02	$\mathbf{0.58 \pm 0.01}$	**T** (< 0.001)
jedit	0.54 ± 0.06	$\mathbf{0.68 \pm 0.02}$	**T** (< 0.001)
lucene	0.50 ± 0.02	$\mathbf{0.57 \pm 0.02}$	**T** (< 0.001)
poi	0.55 ± 0.01	$\mathbf{0.65 \pm 0.03}$	**T** (< 0.001)
synapse	0.46 ± 0.03	$\mathbf{0.60 \pm 0.02}$	**T** (< 0.001)
xalan	0.62 ± 0.03	$\mathbf{0.70 \pm 0.01}$	**T** (< 0.001)
xerces	0.42 ± 0.02	0.51 ± 0.01	**T** (< 0.001)
Average	0.52 ± 0.03	$\mathbf{0.61 \pm 0.02}$	**7/7**

MCC

Project	TPM	TACM	SD (p-value)
camel	0.11 ± 0.04	$\mathbf{0.23 \pm 0.02}$	**T** (< 0.001)
jedit	0.11 ± 0.13	$\mathbf{0.40 \pm 0.03}$	**T** (< 0.001)
lucene	0.00 ± 0.05	$\mathbf{0.17 \pm 0.04}$	**T** (< 0.001)
poi	0.12 ± 0.03	$\mathbf{0.30 \pm 0.05}$	**T** (< 0.001)
synapse	-0.08 ± 0.06	$\mathbf{0.26 \pm 0.03}$	**T** (< 0.001)
xalan	0.26 ± 0.07	$\mathbf{0.42 \pm 0.02}$	**T** (< 0.001)
xerces	-0.14 ± 0.03	$\mathbf{0.07 \pm 0.04}$	**T** (< 0.001)
Average	0.05 ± 0.06	$\mathbf{0.26 \pm 0.03}$	**7/7**

F1

Project	TPM	TACM	SD (p-value)
camel	0.19 ± 0.11	$\mathbf{0.31 \pm 0.03}$	**T** (< 0.001)
jedit	0.29 ± 0.16	$\mathbf{0.58 \pm 0.04}$	**T** (< 0.001)
lucene	$\mathbf{0.70 \pm 0.09}$	0.57 ± 0.19	**T** (0.002)
poi	0.70 ± 0.11	$\mathbf{0.71 \pm 0.05}$	F (0.638)
synapse	0.27 ± 0.04	$\mathbf{0.40 \pm 0.04}$	**T** (< 0.001)
xalan	0.56 ± 0.06	$\mathbf{0.65 \pm 0.03}$	**T** (< 0.001)
xerces	0.05 ± 0.02	$\mathbf{0.06 \pm 0.05}$	F (0.564)
Average	0.39 ± 0.08	$\mathbf{0.47 \pm 0.06}$	**5/7**

MLPC within the TACM model performed better than the MLPC within the TPM model in all performance measurements for each PROMISE project, except for the F1-score for *lucene* project. However, for this specific project, the TACM model has outperformed the MLPC model in terms of MCC by 0.17. Since the MCC is a more adequate performance assessor for binary classifiers than the F1-score, the MLPC within the TACM model can be considered more suitable for use in *lucene* project than the MLPC within the TPM model. Finally, Table 7 shows that despite a lower average F1-score (by 9%), the performance of SVMC is significantly higher in terms of an average MCC, i.e. by 0.34, when trained using the proposed aggregated change metrics instead of process metrics. A similar improvement is also indicated by a higher average AUC score (by 13%).

It is worth noting that there have been a small number of experiments where we cannot claim the performance of the compared models differed significantly in the projects analyzed, but considering a much greater number of experiments where the models differed significantly, we believe this is sufficient to support our conclusion.

The performance of commonly used classifiers can be improved by adding aggregated change metrics to the existing set of traditional features used to train classifiers.

VI. THREATS TO VALIDITY

As with any empirical study, there are threats to validity that should be discussed. Below we discuss the external, internal, construct and conclusion validity of our study.

A. EXTERNAL VALIDITY

In the context of external validity, it must be examined whether and to what extent a generalization of the research results is possible. Since the quality of the experimental results depends on the data set used, we have chosen to use the data set commonly used in software defect prediction studies. As such, it should be suitable for developing and validating models for identifying defect-prone software modules. Nevertheless, the experiments conducted in this research can also be performed with a different data set.

B. INTERNAL VALIDITY

To assess the quality of the proposed metrics, it should be examined whether the performance of the defect prediction models will be improved when they are developed using these metrics. For this purpose, we had to decide which classifiers would be developed in the experiments and which methods for feature selection would be used. Even though our decisions are made according to the common practice in software defect prediction and are also justified in this paper, it is possible that the internal validity of our study is influenced by the preference of classifiers and feature selection methods. In order to minimize this possibility, we have decided to use different but commonly used classifiers and feature selection methods in this research. In future research, however, more different classifiers and feature selection methods may be used within the framework for developing software defect prediction model used in this research.

C. CONSTRUCT VALIDITY

The evaluation of the proposed metrics depends directly on the measure used to assess the correlation between these metrics and the defect label, and on the performance measures used to assess the performance of the classifiers. To make the assessment fair, we used the widely used Spearman's rank correlation coefficient for the first assessment, while for the second assessment we reported various performance measures, including those used in the majority of studies regarding software defect prediction. Nevertheless, other appropriate correlation measures and evaluation metrics may also be used for evaluation purposes.

D. CONCLUSION VALIDITY

In order to assess the validity of the conclusions drawn in this research, we have ensured statistical significance of our results by meeting a requirement of the central limit theorem. More specifically, since 30 repetitions of an experiment according to the central limit theorem are considered sufficient to minimize the statistical bias and variance of the experimental results, we repeated each experiment 30 times. However, to make the research results more reliable, we also performed appropriate significance tests to demonstrate that the results provided by the models are significantly improved when they are trained using the proposed metrics instead of the process metrics. For this purpose, we have used the paired t-test and the paired Wilcoxon test, but it is possible to perform a different statistical test as long as the data meet the requirements of the test.

VII. CONCLUSION

The need for an effective method to detect potential defects in software is growing steadily as the number, size and complexity of software systems in today's world increases. Many research groups have made considerable efforts to develop such a method, but the results they have achieved suggest that there is still room for improvement, especially in extracting features from the software information which will faithfully represent its modules.

In this work, we have proposed a set of features that provide information about the development process of software modules. In contrast to the existing metrics for representing a software module, the proposed features take into account all changes made in the module's source code, i.e. they are calculated by aggregating the change metrics extracted from each change, taking into account the chronological order of the changes. As such, they provide a more nuanced picture of the development of the module that has proven to be relevant for defect-proneness, compared to existing features.

In the experiments conducted on seven frequently used projects from the data set, we investigated which of the proposed features are most correlated with defect-proneness. In addition, we have shown that the performance of defect prediction models improves in terms of AUC, MCC and F1 score when they are trained on a combination of traditional and proposed features rather than on traditional features and process metrics. Most importantly, however, the proposed metrics have allowed the models to improve their performance on the data that suffer most from the class imbalance, which is identified as problematic in defect prediction research.

Encouraged by the experimental results obtained in this paper, we will try to extract more metrics that reflect the development process of the modules in our future work. Furthermore, we will investigate the applicability of the proposed metrics in detecting the defect-prone software modules for source code written in other programming languages.

ACKNOWLEDGMENT

The Titan X Pascal used for this research was donated by NVIDIA Corporation.

REFERENCES

[1] F. R. Porto, "Cross-project defect prediction with meta-learning," Ph.D. dissertation, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Paulo, Brazil, 2017.

[2] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Jul. 2017, pp. 318–328.

[3] A. Okutan and O. T. Yıldız, "Software defect prediction using Bayesian networks," *Empirical Softw. Eng.*, vol. 19, no. 1, pp. 154–181, Feb. 2014.

[4] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. 38th Int. Conf. Softw. Eng.*, May 2016, pp. 297–308.

[5] L. Madeyski and M. Jureczko, "Which process metrics can significantly improve defect prediction models? An empirical study," *Softw. Qual. J.*, vol. 23, no. 3, pp. 393–422, Sep. 2015.

[6] G. R. Choudhary, S. Kumar, K. Kumar, A. Mishra, and C. Catal, "Empirical analysis of change metrics for software fault prediction," *Comput. Electr. Eng.*, vol. 67, pp. 15–24, Apr. 2018.

[7] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proc. 13th Int. Conf. Softw. Eng. ICSE*, 2008, pp. 181–190.

[8] W. Rhmann, B. Pandey, G. Ansari, and D. K. Pandey, "Software fault prediction based on change metrics using hybrid algorithms: An empirical study," *J. King Saud Univ. Comput. Inf. Sci.*, vol. 32, no. 4, pp. 419–424, May 2020.

[9] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur.*, Aug. 2015, pp. 17–26.

[10] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, Jun. 2013.

[11] Q. He, B. Shen, and Y. Chen, "Software defect prediction using semi-supervised learning with change burst information," in *Proc. IEEE 40th Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, Jun. 2016, pp. 113–122.

[12] Y. Xia, G. Yan, and H. Zhang, "Analyzing the significance of process metrics for TT&C software defect prediction," in *Proc. IEEE 5th Int. Conf. Softw. Eng. Service Sci.*, Jun. 2014, pp. 77–81.

[13] D. Wahyudin, A. Schatten, D. Winkler, A. M. Tjoa, and S. Biffl, "Defect prediction using combined product and project metrics-a case study from the open source 'apache' myfaces project family," in *Proc. 34th Euromicro Conf. Softw. Eng. Adv. Appl.*, Sep. 2008, pp. 207–215.

[14] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: Current results, limitations, new approaches," *Automated Softw. Eng.*, vol. 17, no. 4, pp. 375–407, Dec. 2010.

[15] T. J. Shippey, "Exploiting abstract syntax trees to locate software defects," Ph.D. dissertation, School Comput. Sci., Univ. Hertfordshire, Hatfield, U.K., 2015.

[16] M. H. Halstead, *Elements of Software Science*. Haarlem, The Netherlands: North-Holland, 1977.

[17] C. Manjula and L. Florence, "Deep neural network based hybrid approach for software defect prediction using software metrics," *Cluster Comput.*, vol. 22, no. 4, pp. 9847–9863, 2019.

[18] Ö. F. Arar and K. Ayan, "Software defect prediction using cost-sensitive neural network," *Appl. Soft Comput.*, vol. 33, pp. 263–277, Aug. 2015.

[19] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust prediction of fault-proneness by random forests," in *Proc. 15th Int. Symp. Softw. Rel. Eng.*, Nov. 2004, pp. 417–428.

[20] A. Kaur and R. Malhotra, "Application of random forest in predicting fault-prone classes," in *Proc. Int. Conf. Adv. Comput. Theory Eng.*, Dec. 2008, pp. 37–43.

[21] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *J. Syst. Softw.*, vol. 81, no. 5, pp. 649–660, May 2008.

[22] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[23] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[24] M. Gradišnik, T. Beranič, and S. Karakatič, "Impact of historical software metric changes in predicting future maintainability trends in open-source software development," *Appl. Sci.*, vol. 10, no. 13, p. 4624, Jul. 2020.

[25] R. Harrison, S. J. Counsell, and R. V. Nithi, "An evaluation of the MOOD set of object-oriented software metrics," *IEEE Trans. Softw. Eng.*, vol. 24, no. 6, pp. 491–496, Jun. 1998.

[26] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *Proc. 33rd Int. Conf. Softw. Eng. ICSE*, May 2011, pp. 481–490.

[27] L. Qiao and Y. Wang, "Effort-aware and just-in-time defect prediction with neural network," *PLoS ONE*, vol. 14, no. 2, Feb. 2019, Art. no. e0211359.

[28] B. Stanic, "Static code metrics vs. Process metrics for software fault prediction using Bayesian network learners," M.S. thesis, Dept. Softw. Eng., School Innov., Des. Eng., Mälardalen Univ., Västerås, Sweden, 2015. [Online]. Available: http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A874471&dswid=-4377

[29] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, pp. 653–661, Jul. 2000.

[30] D. Radjenović, M. Heričko, R. Torkar, and A. Živković, "Software fault prediction metrics: A systematic literature review," *Inf. Softw. Technol.*, vol. 55, no. 8, pp. 1397–1418, Aug. 2013.

[31] S. Majumder, P. Mody, and T. Menzies, "Revisiting process versus product metrics: A large scale analysis," 2020, *arXiv:2008.09569*. [Online]. Available: http://arxiv.org/abs/2008.09569

[32] J. Nam, "Survey on software defect prediction," Dept. Compter Sci. Eng., Hong Kong Univ. Sci. Technol., Tech. Rep. 2014.

[33] C. Pan, M. Lu, B. Xu, and H. Gao, "An improved CNN model for within-project software defect prediction," *Appl. Sci.*, vol. 9, no. 10, p. 2138, May 2019.

[34] H. K. Dam, T. Pham, S. Wee Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, "A deep tree-based model for software defect prediction," 2018, *arXiv:1802.00921*. [Online]. Available: http://arxiv.org/abs/1802.00921

[35] H. Liang, Y. Yu, L. Jiang, and Z. Xie, "Seml: A semantic LSTM model for software defect prediction," *IEEE Access*, vol. 7, pp. 83812–83824, 2019.

[36] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, "Software defect prediction via attention-based recurrent neural network," *Sci. Program.*, vol. 2019, pp. 1–14, Apr. 2019.

[37] Q. Zhang and B. Wu, "Software defect prediction via transformer," in *Proc. IEEE 4th Inf. Technol., Netw., Electron. Autom. Control Conf. (ITNEC)*, Jun. 2020, pp. 874–879.

[38] N. Hoque, D. K. Bhattacharyya, and J. K. Kalita, "MIFS-ND: A mutual information-based feature selection method," *Expert Syst. Appl.*, vol. 41, no. 14, pp. 6371–6385, Oct. 2014.

[39] S. Kan, *Metrics and Models in Software Quality Engineering*, 2nd ed. Reading, MA, USA: Addision-Wesley, 2002.

[40] I. E. Frank and R. Todeschini, *The Data Analysis Handbook*. Amsterdam, The Netherlands: Elsevier, 1994.

[41] M. Dash and H. Liu, "Feature selection for classification," *Intell. Data Anal.*, vol. 1, nos. 1–4, pp. 131–156, 1997.

[42] R. Shatnawi and W. Li, "The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process," *J. Syst. Softw.*, vol. 81, no. 11, pp. 1868–1882, Nov. 2008.

[43] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proc. 3rd Int. Workshop Predictor Models Softw. Eng. (PROMISE, ICSE Workshops)*, May 2007, p. 9.

[44] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust prediction of fault-proneness by random forests," in *Proc. 15th Int. Symp. Softw. Rel. Eng.*, Nov. 2004, pp. 417–428.

[45] J. Jiarpakdee, C. Tantithamthavorn, and C. Treude, "AutoSpearman: Automatically mitigating correlated metrics for interpreting defect models," 2018, *arXiv:1806.09791*. [Online]. Available: http://arxiv.org/abs/1806.09791

[46] J. Jiarpakdee, C. Tantithamthavorn, and A. E. Hassan, "The impact of correlated metrics on the interpretation of defect models," *IEEE Trans. Softw. Eng.*, early access, Jan. 10, 2019, doi: 10.1109/TSE.2019.2891758.

[47] T. M. Khoshgoftaar and K. Gao, "Feature selection with imbalanced data for software defect prediction," in *Proc. Int. Conf. Mach. Learn. Appl.*, Dec. 2009, pp. 235–240.

[48] H. Wang, T. M. Khoshgoftaar, K. Gao, and N. Seliya, "Mining data from multiple software development projects," in *Proc. IEEE Int. Conf. Data Mining Workshops*, Dec. 2009, pp. 551–557.

[49] S.-J. Lee, Z. Xu, T. Li, and Y. Yang, "A novel bagging C4.5 algorithm based on wrapper feature selection for supporting wise clinical decision making," *J. Biomed. Informat.*, vol. 78, pp. 144–155, Feb. 2018.

[50] D. Rodriguez, R. Ruiz, J. Cuadrado-Gallego, J. Aguilar-Ruiz, and M. Garre, "Attribute selection in software engineering datasets for detecting fault modules," in *Proc. 33rd EUROMICRO Conf. Softw. Eng. Adv. Appl. (EUROMICRO)*, Aug. 2007, pp. 418–423.

[51] H. Osman, M. Ghafari, and O. Nierstrasz, "The impact of feature selection on predicting the number of bugs," 2018, *arXiv:1807.04486*. [Online]. Available: http://arxiv.org/abs/1807.04486

[52] I. Gondra, "Applying machine learning to software fault-proneness prediction," *J. Syst. Softw.*, vol. 81, no. 2, pp. 186–195, Feb. 2008.

[53] H. Osman, M. Ghafari, and O. Nierstrasz, "Automatic feature selection by regularization to improve bug prediction accuracy," in *Proc. IEEE Workshop Mach. Learn. Techn. for Softw. Qual. Eval. (MaLTeSQuE)*, Feb. 2017, pp. 27–32.

[54] K. Muthukumaran, A. Rallapalli, and N. L. B. Murthy, "Impact of feature selection techniques on bug prediction models," in *Proc. 8th India Softw. Eng. Conf.*, Feb. 2015, pp. 120–129.

[55] K. Gao, T. M. Khoshgoftaar, H. Wang, and N. Seliya, "Choosing software metrics for defect prediction: An investigation on feature selection techniques," *Softw., Pract. Exper.*, vol. 41, no. 5, pp. 579–606, Apr. 2011.

[56] L. Jia, "A hybrid feature selection method for software defect prediction," *IOP Conf. Ser., Mater. Sci. Eng.*, vol. 394, Aug. 2018, Art. no. 032035.

[57] F. Wang, J. Ai, and Z. Zou, "A cluster-based hybrid feature selection method for defect prediction," in *Proc. IEEE 19th Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Jul. 2019, pp. 1–9.

[58] M. Harman, "The relationship between search based software engineering and predictive modeling," in *Proc. 6th Int. Conf. Predictive Models Softw. Eng. PROMISE*, 2010, pp. 1–13.

[59] L. Son, N. Pritam, M. Khari, R. Kumar, P. Phuong, and P. Thong, "Empirical study of software defect prediction: A systematic mapping," *Symmetry*, vol. 11, no. 2, p. 212, Feb. 2019.

[60] C. Catal and B. Diri, "Software defect prediction using artificial immune recognition system," in *Proc. 25th Conf. IASTED Int. Multi-Conf., Softw. Eng.*, 2007, pp. 285–290.

[61] C. Catal, B. Diri, and B. Ozumut, "An artificial immune system approach for fault prediction in object-oriented software," in *Proc. 2nd Int. Conf. Dependability Comput. Syst. (DepCoS-RELCOMEX)*, Jun. 2007, pp. 238–245.

[62] D. Azar and J. Vybihal, "An ant colony optimization algorithm to improve software quality prediction models: Case of class stability," *Inf. Softw. Technol.*, vol. 53, no. 4, pp. 388–393, Apr. 2011.

[63] K. Kumar, D. Jayadev Gyani, and G. Narsimha, "Software defect prediction using ant colony optimization," *Int. J. Appl. Eng. Res.*, vol. 13, no. 19, pp. 14291–14297, 2018.

[64] M. Akour and W. Y. Melhem, "Software defect prediction using genetic programming and neural networks," *Int. J. Open Source Softw. Processes*, vol. 8, no. 4, pp. 32–51, Oct. 2017.

[65] S. S. Rathore and S. Kumar, "Predicting number of faults in software system using genetic programming," *Procedia Comput. Sci.*, vol. 62, pp. 303–311, Jan. 2015.

[66] G. Denaro and M. Pezze, "An empirical evaluation of fault-proneness models," in *Proc. 24th Int. Conf. Softw. Eng., ICSE*, May 2002, pp. 241–251.

[67] T. M. Khoshgoftaar and N. Seliya, "Tree-based software quality estimation models for fault prediction," in *Proc. 8th IEEE Symp. Softw. Metrics*, Jun. 2002, pp. 203–214.

[68] J. Galindo and P. Tamayo, "Credit risk assessment using statistical and machine learning: Basic methodology and risk modeling applications," *Comput. Econ.*, vol. 15, no. 1, pp. 107–143, 2000.

[69] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan, "Heterogeneous defect prediction," *IEEE Trans. Softw. Eng.*, vol. 44, no. 9, pp. 874–896, Sep. 2018.

[70] B. Turhan and A. Bener, "Analysis of naive bayes' assumptions on software fault data: An empirical study," *Data Knowl. Eng.*, vol. 68, no. 2, pp. 278–290, Feb. 2009.

[71] R. T. Asmono, R. S. Wahono, and A. Syukur, "Absolute correlation weighted naïve Bayes for software defect prediction," *J. Softw. Eng.*, vol. 1, no. 1, pp. 38–45, 2015.

[72] X. Xuan, D. Lo, X. Xia, and Y. Tian, "Evaluating defect prediction approaches using a massive set of metrics: An empirical study," in *Proc. 30th Annu. ACM Symp. Appl. Comput.*, Apr. 2015, pp. 1644–1647.

[73] J. Wang, B. Shen, and Y. Chen, "Compressed C4.5 models for software defect prediction," in *Proc. 12th Int. Conf. Qual. Softw.*, Aug. 2012, pp. 13–16.

[74] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, "Micro interaction metrics for defect prediction," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng. SIGSOFT/FSE*, 2011, pp. 311–321.

[75] A. G. Koru and H. Liu, "Building effective defect-prediction models in practice," *IEEE Softw.*, vol. 22, no. 6, pp. 23–29, Nov. 2005.

[76] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson, "Software defect prediction using static code metrics underestimates defect-proneness," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2010, pp. 1–7.

[77] L. Jiang, S. Jiang, L. Gong, Y. Dong, and Q. Yu, "Which process metrics are significantly important to change of defects in evolving projects: An empirical study," *IEEE Access*, vol. 8, pp. 93705–93722, 2020.

[78] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Programmer-based fault prediction," in *Proc. 6th Int. Conf. Predictive Models Softw. Eng. PROMISE*, 2010, pp. 1–10.

[79] S. Matsumoto, Y. Kamei, A. Monden, K.-I. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in *Proc. 6th Int. Conf. Predictive Models Softw. Eng. - PROMISE*, 2010, pp. 1–9.

[80] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, Nov. 2012.

[81] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, Nov. 2011.

[82] J. Eyolfson, L. Tan, and P. Lam, "Do time of day and developer experience affect commit bugginess," in *Proc. 8th Work. Conf. Mining Softw. Repositories MSR*, 2011, pp. 153–162.

[83] F. Rahman and P. Devanbu, "Ownership, experience and defects: A fine-grained study of authorship," in *Proc. 33rd Int. Conf. Softw. Eng. ICSE*, 2011, pp. 491–500.

[84] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "The limited impact of individual developer data on software defect prediction," *Empirical Softw. Eng.*, vol. 18, no. 3, pp. 478–505, Jun. 2013.

[85] S. O. Kini and A. Tosun, "Periodic developer metrics in software defect prediction," in *Proc. IEEE 18th Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2018, pp. 72–81.

[86] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Does measuring code change improve fault prediction?" in *Proc. 7th Int. Conf. Predictive Models Softw. Eng. Promise*, 2011, pp. 1–8.

[87] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change bursts as defect predictors," in *Proc. IEEE 21st Int. Symp. Softw. Rel. Eng.*, Nov. 2010, pp. 309–318.

[88] S. A. Ajila and R. T. Dumitrescu, "Experimental use of code delta, code churn, and rate of change to understand software product line evolution," *J. Syst. Softw.*, vol. 80, no. 1, pp. 74–91, Jan. 2007.

[89] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy," *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, Mar. 2008.

[90] Y. Kastro and A. B. Bener, "A defect prediction method for software versioning," *Softw. Qual. J.*, vol. 16, no. 4, pp. 543–562, Dec. 2008.

[91] Y. Liu, Y. Li, J. Guo, Y. Zhou, and B. Xu, "Connecting software metrics across versions to predict defects," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Mar. 2018, pp. 232–243.

[92] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 432–441.

[93] T. Illes-Seifert and B. Paech, "Exploring the relationship of a file's history and its fault-proneness: An empirical method and its application to open source programs," *Inf. Softw. Technol.*, vol. 52, no. 5, pp. 539–558, May 2010.

[94] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, May 2009, pp. 78–88.

[95] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *J. Syst. Softw.*, vol. 83, no. 1, pp. 2–17, Jan. 2010.

[96] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: Analytics and risk prediction of software commits," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, Aug. 2015, pp. 966–969.

[97] D. Bowes, T. Hall, and J. Petrić, "Software defect prediction: Do different classifiers find the same defects?" *Softw. Qual. J.*, vol. 26, no. 2, pp. 525–552, Jun. 2018.

[98] A. Géron, *Hands-on Machine Learning With Scikit-Learn, Keras, and Tensorflow: Concepts, Tools, and Techniques to Build Intelligent Systems*. Newton, MA, USA: O'Reilly Media, 2019.

[99] Q. Huang, C. Fang, S. Mittal, and R. D. Blanton, "Improving diagnosis efficiency via machine learning," in *Proc. IEEE Int. Test Conf. (ITC)*, Oct. 2018, pp. 1–10.

[100] Y. Jiang, J. Lin, B. Cukic, and T. Menzies, "Variance analysis in software fault prediction models," in *Proc. 20th Int. Symp. Softw. Rel. Eng.*, Nov. 2009, pp. 99–108.

[101] R. M. O'brien, "A caution regarding rules of thumb for variance inflation factors," *Qual. Quantity*, vol. 41, no. 5, pp. 673–690, Sep. 2007.

[102] J. Akosa, "Predictive accuracy: A misleading performance measure for highly imbalanced data," in *Proc. SAS Global Forum*, vol. 12, 2017, pp. 2–5.

[103] J. D. Evans, *Straightforward Statistics for the Behavioral Sciences*. San Francisco, CA, USA: Thomson Brooks/Cole Publishing Co, 1996.

[104] T. M. Khoshgoftaar, X. Yuan, and E. B. Allen, "Balancing misclassification rates in classification-tree models of software quality," *Empirical Softw. Eng.*, vol. 5, no. 4, pp. 313–330, 2000.

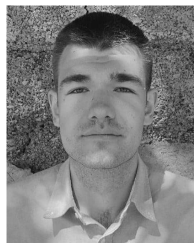
[105] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Tech. J.*, vol. 5, no. 2, pp. 169–180, Apr. 2000.

[106] P. Schober, C. Boer, and L. A. Schwarte, "Correlation coefficients: Appropriate use and interpretation," *Anesthesia Analgesia*, vol. 126, no. 5, pp. 1763–1768, May 2018.

[107] I. Arora, V. Tetarwal, and A. Saha, "Open issues in software defect prediction," *Procedia Comput. Sci.*, vol. 46, pp. 906–912, Jan. 2015.

LUCIJA ŠIKIĆ received the master's degree in computer science from the University of Zagreb, in 2018. Her Ph.D. project is source code defect prediction. She has published in *Journal of Software and Systems*. She is currently a Research Associate with the Consumer Computing Laboratory, Faculty of Electrical Engineering and Computing, University of Zagreb.

PETAR AFRIĆ received the master's degree in computer science from the University of Zagreb, in 2018. His Ph.D. project is source code defect prediction. He has published in *Journal of Software and Systems* and has presented at the IEEE International Conference on Software Quality, Reliability and Security. He is currently a Research Associate at the Consumer Computing Laboratory, Faculty of Electrical Engineering and Computing, University of Zagreb.



ADRIAN SATJA KURDIJA (Member, IEEE) received the Ph.D. degree in computer science from the Faculty of Electrical Engineering and Computing, University of Zagreb, in 2020. His Ph.D. project deals with service selection and QoS prediction. He has published in the IEEE COMMUNICATIONS LETTERS, the *European Journal of Operational Research*, the *International Journal of Web and Grid Services*, the *Knowledge-Based Systems*, and the IEEE TRANSACTIONS ON SERVICES COMPUTING. He is currently a Research Assistant at the Consumer Computing Laboratory, Faculty of Electrical Engineering and Computing, University of Zagreb.

MARIN ŠILIĆ (Member, IEEE) received the Ph.D. degree in computer science from the Faculty of Electrical Engineering and Computing, University of Zagreb, in 2013. He is currently an Associate Professor at the Faculty of Electrical Engineering and Computing, University of Zagreb. His research interests span machine learning, data mining, service-oriented computing, and software engineering. He has published several papers in the IEEE TRANSACTIONS ON SERVICES COMPUTING, IEEE TRANSACTIONS ON DEPANDABLE AND SECURE COMPUTING, *Journal of Systems and Software*, and *Knowledge-Based Systems*. Also, he has published his research results at the *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* and at the *IEEE International Conference on Software Quality, Reliability and Security*.