Received August 22, 2017, accepted September 15, 2017, date of publication October 5, 2017, date of current version October 25, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2759180

Integrated Approach to Software Defect Prediction

EBUBEOGU AMARACHUKWU FELIX, (Graduate Student Member, IEEE) AND SAI PECK LEE, (Member, IEEE)
Faculty of Computer Science and Information Technology, University of Malaya, Kuala Lumpur 50603, Malaysia

Corresponding author: Sai Peck Lee (saipeck@um.edu.my)

This work was supported in part by the Ministry of Higher Education under a University of Malaya High Impact Research under Grant UM.C/625/1/HIR/MOHE/FCSIT/13 and in part by FRGS under Grant FP001-2016.

ABSTRACT Software defect prediction provides actionable outputs to software teams while contributing to industrial success. Empirical studies have been conducted on software defect prediction for both cross-project and within-project defect prediction. However, existing studies have yet to demonstrate a method of predicting the number of defects in an upcoming product release. This paper presents such a method using predictor variables derived from the defect acceleration, namely, the defect density, defect velocity, and defect introduction time, and determines the correlation of each predictor variable with the number of defects. We report the application of an integrated machine learning approach based on regression models constructed from these predictor variables. An experiment was conducted on ten different data sets collected from the PROMISE repository, containing 22 838 instances. The regression model constructed as a function of the average defect velocity achieved an adjusted R-square of 98.6%, with a p-value of < 0.001. The average defect velocity is strongly positively correlated with the number of defects, with a correlation coefficient of 0.98. Thus, it is demonstrated that this technique can provide a blueprint for program testing to enhance the effectiveness of software development activities.

INDEX TERMS Software defect prediction, machine learning, number of defects, defect velocity, class imbalance.

I. INTRODUCTION

Software defect prediction helps to ensure that testing and debugging remain in a fast-track mode by providing advance information on the number of faults that are likely to be found in a new program. Both stakeholders and software companies spend substantial resources on repairing the damage caused by defects in software products. The need for software defect prediction in software engineering is driven by the importance of the proper use of available resources in software testing and in delivering quality software products to the user.

However, existing studies have yet to demonstrate a method of predicting the number of bugs in an upcoming release using derived variables. Such an ability to predict the number of software defects would assist software teams in planning software testing and in maintaining software standards. Therefore, considerable effort is still needed to develop an acceptable prediction model that can predict the number of software defects in a future software project. The primary purpose of a prediction model is to provide cost-effective support and guidance during software testing [1]. To ensure cost-effectiveness, it is important to be able to predict the number of defects in a new product release. Such a prediction method should focus on guiding software testing efforts by predicting the possible number of software defects in a new product release before testing begins. Identifying and correcting faults in software can be expensive if performed only after the delivery of the software product to stakeholders. This is because software developers face substantial risks and difficulties in identifying the root causes of faults and correcting those faults during the requirement, design and implementation phases of the software life cycle [2].

Prediction of defects in error-prone software can provide a blueprint for program testing and can also improve the effectiveness of development activities [3]. If the developer fails to address a problem early in the software development process, then the problem can become more complicated, with a corresponding increase in cost in later phases. Conversely, if the complexity is kept low, then the software can be more easily understood and modified during its life cycle [4].

To ensure that software stakeholders make good decisions about future programs and properly allocate resources to software projects, the output of a prediction model must provide a practical result to managers [5]. Currently, software companies apply a series of previously proposed techniques or custom approaches for predicting software defects. However, software companies continue to face unexpected faults that could have been exposed during the initial stages of development if suitable prediction practices had been implemented. Detecting fault-prone software components as early as possible can enable software experts to remain focused and to direct their resources toward addressing possible issues that may arise in a software system that is yet to be developed [6]. The metrics used early in the software development life cycle (SDLC) can play a significant role in project management by helping to reduce the defect density of a software product; specifically, these metrics can be used to determine whether increased quality monitoring is necessary during development. They can also be used in the planning of verification and validation activities [7].

In addition, it is essential to consider the cost and benefits of predicting the number of software defects before program testing starts. If the cost value of a prediction is not assessed first, then poor prediction results may be obtained. Furthermore, inappropriate resource allocation strategies can significantly increase the testing effort [8]. This study proposes a machine-learning-inspired (**MACLI**) approach that can be used to predict the number of software defects in a cost-effective manner. The outcome of such a prediction can help managers and software development teams to appropriately allocate the available resources to software testing and maintenance.

In this study, we used an integrated machine learning approach based on regression models constructed using a set of predictor variables. Based on these variables, our regression models estimate the number of defects in a software product prior to testing through multiple and simple linear regression techniques. Our approach is distinct from previously proposed prediction models for binary defect classification because our technique can estimate the number of defects in a defective program, whereas in binary classification, a program is merely labeled defective or non-defective without estimating the number of defects. Although many techniques for software defect prediction have been proposed in previous studies and have helped software developers to achieve remarkable binary classification results [9], these techniques have not considered the estimation of the number of defects using the derived predictor variables applied in this study. An experiment was conducted on 10 different datasets collected from the PROMISE repository, containing 22,838 instances. The results indicate that the number of defects shows a strong positive correlation with the average defect velocity, a weak positive correlation with the average defect density, and a negative correlation with the average defect introduction time, with correlation coefficients of 0.98, 0.22 and −0.30, respectively. The regression model constructed as a function of the average defect velocity achieves an adjusted R-square of 98.6%, with a p-value of <0.001.

The remainder of this paper is organized as follows. Section II discusses related work. Section III explains our motivation. In Section IV, we present our proposed framework for defect prediction. Section V describes our experiment and the construction of our prediction models. We then present our results and discussion in Section VI. Section VII discusses the potential threats to the validity of this study. Finally, Section VIII presents our conclusion and directions for future work.

II. RELATED WORK

Previous studies have proposed models for both within-project and across-project defect prediction at the source code level based on the software development process. Although these studies have achieved noteworthy performance in terms of accuracy, reliability, fault detection and performance improvement in binary defect classification, the literature lacks an approach for estimating the number of software defects, and no previous studies have considered the predictor variables applied in this study.

For example, Han *et al.*, as reported in [10], proposed a sustainable program reliability estimation model based on the software development process. Their method comprises an integrated software reliability model, a program construction forecast model, a Rayleigh model, and computer-aided software safety estimation to improve prediction outcomes.

Taba *et al.* [11] enhanced the accuracy of defect prediction by applying metrics based on anti-patterns. They used re-factoring to correct poor designs and used anti-patterns to identify weaknesses in a design that might increase the risk of future defects. If defects can be predicted using anti-pattern information, then the development team can use re-factoring to reduce the risk of defects in the system.

Mori [12] developed a prediction model with high accuracy and explanatory power by superposing a naïve Bayes model on an ensemble model.

Jing *et al.* [13] achieved improved software prediction accuracy using a software defect prediction technique based on collaborative representation classification. Their proposed metric-based software defect prediction method resulted in a considerably larger number of defect-free modules compared with the number of faulty modules. Although class imbalance was encountered in that study, the outcome of the study was not affected because the class imbalance was properly addressed through Laplace score sampling for sample training, which resulted in an improved prediction accuracy.

Mahmood *et al.* [14] analyzed the predictive performance achieved using imbalanced data in the prediction of software defects. When the data used for classification are of unequal proportions among different classes, the predictive accuracy of defect prediction studies appears to be low, whereas balanced data result in increased predictive performance. One measure that can be used to address such imbalance problems was reported in [13].

Islam and Sakib, as reported in [15], used package-based clustering to enhance the accuracy of software defect prediction. They grouped software packages into multiple clusters according to their relationships and similarities and proposed a prediction model using this package-based clustering approach that achieved prediction rates of 54%, 71%, and 90%, which were higher than those obtained using a prediction model based on BorderFlow and k-means clustering.

Tantithamthavorn *et al.* [16] argued that the outcome and accuracy of any prediction model are functions of the data used for training. Therefore, prediction models may be overfitted and produce untrustworthy results if the datasets are not reliable.

Mausa *et al.* [17] applied a standard procedure for collecting data to be used in a defect prediction study. Their approach limits the factor of knowledge related to a biased dataset by providing details on the capabilities of a bug-code analyzer. This provides all of the functionalities necessary to create a software defect prediction dataset using a defect monitoring device capable of analyzing defects from the contents of source code management repositories.

The erroneous use of training datasets can lead to poor models and biased prediction results. To overcome this problem, Siebra and Mellof [18] applied a simple pre-processing technique to achieve reliable results. Meanwhile, Rahman *et al.* [19] applied a feature-space transformation process in combination with data pre-processing and normalization to improve prediction accuracy.

Malhotra and Raje, as reported in [20], addressed the problem of incorrect interpretations when verifying the performance of a defect prediction model. They used an object-oriented metric design suite and compared various machine learning techniques to investigate the impact of object-oriented metrics on erroneous classification. The naïve Bayes technique was found to be suitable for predicting the defect-proneness of a class using object-oriented metrics. Their results are consistent with the finding of Menzies *et al.* that naïve Bayes appears to be the best technique for constructing defect prediction models.

Cavezza *et al.* [21] examined defect prediction performance during the software development process. The findings of the study suggest that when a standard approach is applied for defect prediction, promising results can be obtained through continuous refinement of the prediction model using new commit data and by predicting whether any action introduced into a program introduces a bug.

Lu *et al.* [22] investigated the development of a semi-supervised learning technique for program defect prediction using a variant of a self-study algorithm. The study confirmed that confidence fitting can be used as a substitute for established supervised algorithms. The semi-supervised algorithm in combination with dimensional reduction performed considerably better than a random forest model when modules with typical faults were used for training.

Xuan *et al.* [23] comprehensively studied within-project defect prediction performance in a practical and sophisticated manner. They used a massive set of evaluation metrics and reported that a Bayesian network achieves good performance. Other classification models may also perform better in different scenarios because no single model dominates in binary defect classification performance.

Fukushima *et al.* [24] evaluated a cross-project model using just-in-time (JIT) prediction via a case study of open-source projects. The study reported that within-project defect prediction models that are able to achieve high accuracy are uncommon compared with high-accuracy cross-project prediction models. However, cross-project prediction models trained on projects with identical correlations between the predictor and dependent variables often exhibit good performance.

Nam and Kim, as reported in [25], applied a collection of metric equalities to construct a prediction model for projects with diverse metric sets. They combined metric selection and metric combination to achieve a forecast rate of 68%, which was higher than or comparable to the rates achieved for within-project defect prediction. Their proposed method also showed statistical significance.

Jing *et al.* [26] presented a successful solution for mixed cross-company defect prediction by means of combined metric representations for data origin and destination. The performance of this approach depends on the correlation analysis that is established to achieve effective transfer learning for cross-company defect prediction. In this way, similar initial and resulting data distributions can be obtained.

Panichella *et al.* [27] improved the detection of defect-prone entities among software projects by means of a unified defect predictor that considers the groupings produced by various machine learning techniques.

Lessmann *et al.* [28] proposed techniques for predicting fault-prone modules by prioritizing quality assurance efforts and used these techniques for selecting modules in accordance with their fault probabilities. To date, none of these techniques has demonstrated the ability to predict the number of defects that may exist in an upcoming product release. Our proposed **MACLI** approach attempts to fill this gap to expedite actions taken for quality assurance by predicting the number of software defects using the average defect density, average defect velocity, average defect introduction time and module design complexity. Notably, software complexity significantly affects the cost and time of software development and maintenance [29].

Shepperd *et al.* [30] conducted a meta-analysis of all relevant factors that influence predictive performance. They verified the performance of their defect prediction model by determining the factors that significantly influence the predictive outcomes of software defect classifiers, as determined based on the Matthews correlation coefficient. They found that the choice of classifier only slightly influences the performance, whereas the model building factors (that is, factors related to the research group) exert a significant effect. This is because the research group is responsible for data pre-processing. If the data applied in a study are not properly cleaned, that study may produce a biased outcome.

Consequently, the performance achieved in a prediction study depends predominantly on the research group and not on the choice of classifier.

Zhang *et al.* [31] attempted to construct a universal defect prediction prototype. However, developing a universal model of the primary connections between software metrics and defects is challenging because of the variations among predictors.

Caglayan *et al.* [32] constructed a sensitive defect prediction model based on fault categories. By separating defect statistics into different classes for consideration in a defect prediction algorithm, practitioners are able to take proper actions to improve their prediction accuracy. Therefore, grouping and segmenting the errors in a program before applying a prediction model can also improve accuracy. Maneerat and Muenchaisri [33] analyzed bad smell prediction in an early phase of software development by comparing the performances of different machine learning algorithms through statistical significance testing. Halim [3] proposed a model that can compute the complexity of object-oriented software in the design phase for the prediction of error-prone classes. He applied naïve Bayes and K-nearest-neighbor models to identify the link between complexity and bug-proneness in a design.

Parthipan *et al.* [4] also proposed an evaluation model that captures the symptoms of design complexity using an aspect-oriented complexity evaluation model. Note that in the design phase and at the code level, defect prediction models are primarily designed either to discriminate between defective and non-defective modules (binary classification) or to forecast the number of defects (regression analysis) [32].

Despite extensive efforts, these previous studies have demonstrated various achievements with regard to the prediction of software defects at the code level only. Additional research on software defect prediction will be required to produce a pragmatic prediction model for managers and development teams. To the best of our knowledge, a model that can predict the number of possible defects in an upcoming product release has not yet been proposed for either cross-project or within-project defect prediction. Therefore, this study focuses on predicting the number of defects in a software project to provide managers and development teams with knowledge of future software trends. This study also considers the impact of module design complexity, among other predictor variables, in predicting the number of defects in a software project. We evaluated the impact of module design complexity to determine its effect on the number of defects in a software product. Module design complexity is defined as the difficulty of constructing a detailed design, which often results in various issues and leads to complicated software products.

Certain design violations can lead to increased complexity in software module design and the bad smell phenomenon [34]. In this context, the Unified Modeling Language (UML) standard is helpful for predicting and understanding software systems [35]. We also consider module design complexity because the implementation phase follows the design phase in the SDLC. Therefore, decisions made during the design phase can ultimately affect the success of software implementation. Reducing the complexity of any software design can result in fewer complications during coding, thereby simplifying software maintenance. Software design is important in ensuring software quality. A proper design can ensure a software taxonomy that facilitates the improvement of software quality while providing the basis for all subsequent software activities. Although most software practitioners do not pay significant attention to the need for a detailed design, knowledge of the role of sustainable design remains an important issue [36], [37].

UML provides a high level of detail about software modules, such as class attributes, operations, association names, association directions, and multiplicity [38]. UML also represents the relationships among classes of objects in a system. Khalid *et al.* [39] observed that analyzing complexity at the class level can prevent defects from appearing later in software development and can help to improve the details of a design. The approach proposed in this study can also be applied in the design phase of the SDLC for predicting the number of possible defects in a new version of a software product based on the predictor variables used to construct the prediction models.

III. MOTIVATION

Software defect prediction plays an important role in software standards. Zhang *et al.* [40] reported that predicting the number of software defects can minimize worry during software testing and inspection, thereby enhancing software quality. From the machine learning perspective, improving software quality through defect prediction has become an interesting aspect of software engineering and has been attracting increasing attention for more than three decades. Our inspiration comes from the possibilities enabled by predicting the number of defects in an upcoming product release based on derived metrics in the form of predictor variables derived from the defect acceleration, namely, the defect density, defect velocity and defect introduction time. To this end, we wish to determine the effect of each predictor variable on the number of defects. Studies on predicting the potential number of software errors in a newly released product are lacking. Consequently, software companies continue to suffer from software defects despite the remarkable achievements made in defect prediction.

We hypothesized that the choice of metrics used for prediction also influences the prediction results. Based on this hypothesis, we carefully selected and integrated our predictor variables. The defect density g is the ratio of the number of defects to the project size. The defect velocity v is the rate of change in the defect status with time t. We chose these metrics, which have not previously been applied in research on software defect prediction, as our predictor variables. We selected these metrics because the defect density of a software project depends on the rate at which defects occur.

Thus, the number of defects is a function of the defect acceleration, as shown in Equations (1) to (5). Another reason for selecting these metrics was our desire to produce practical outputs for managers and development teams for software defect prediction.

The number of defects in a new software product needs to be predicted before testing begins. This will enable development teams to deliver more reliable software products and help managers to better allocate the available resources throughout the software development process. To this end, machine learning algorithms are capable of exposing prediction models to new data, that is, training and validation datasets. As the prediction models interact with new data, they can learn independently from the previous computations to produce reliable results. Notably, software defect prediction remains an important aspect of software engineering. Hence, the ability to predict the number of software defects that are likely to be present in a new release is important for ensuring cost-effectiveness and can lead to the delivery of nearly defect-free software products to stakeholders. Furthermore, the advance prediction of the number of software defects can help a software testing team to better utilize the available resources for software testing based on the predicted outcome.

A simple modeling technique is needed to evaluate the effects of our chosen metrics on the number of defects in an upcoming product release. No previous study has considered these predictor variables for predicting the number of software defects. We conducted this study to investigate the prediction of the number of defects in software during the development process. Our modeling technique is based on the relationship between the number of defects and the defect density as a function of defect acceleration. Therefore, in the present study, we used a **MACLI** approach capable of predicting the number of defects to provide managers with useful outputs. This method also enables us to fill a gap in the existing literature. We propose a novel idea for performing defect prediction based on predictor variables that have not been applied in any previous study on defect prediction. We believe that the selected metrics can influence the outcome of any prediction study because these metrics can serve as the independent variables in prediction models. Careful selection of the variables used in prediction has the potential to improve prediction outcomes [41]. Therefore, we strongly believe that our carefully selected variables will produce significant outcomes in predicting the number of defects in a new product release.

IV. PROPOSED FRAMEWORK FOR DEFECT PREDICTION

In this section, we present our proposed framework for early software defect prediction. Figure 1 illustrates the proposed **MACLI** framework. The framework describes the systematic process that we apply for predicting the number of software defects. The first phase is the data pre-processing phase. During this phase, we ensure that detailed information about the source of the data is obtained. This is followed by data

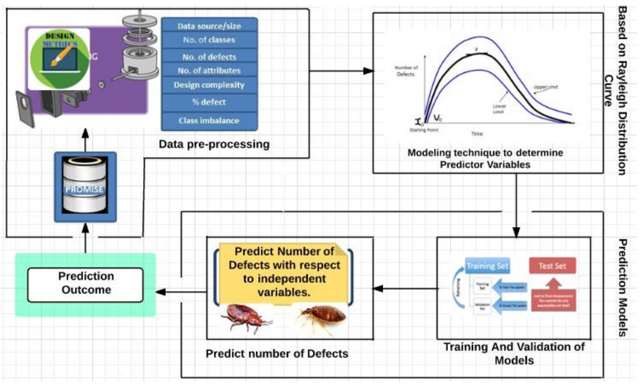

FIGURE 1. Proposed MACLI framework for defect prediction.

analysis, in which we obtain several metrics from the datasets. These metrics include the number of classes in the datasets, the number of attributes, the number of defects, and the percentage of defects. In addition, we determine the design complexity for each dataset. In this study, the effect of the design complexity was compared with the effects of the chosen predictor variables. The next phase in the proposed framework is the application of the modeling technique to derive the predictor variables.

The modeling technique is based on the Rayleigh distribution curve. The prediction modeling phase consists of a model training phase and a model validation phase. The training phase consists of using part of the datasets to train both multiple and simple linear regression models. In this study, the predictor variables derived via our modeling technique were used in training and validating both multiple and simple linear regression models, with the purpose of enhancing software defect prediction and providing practical results to managers and software development teams.

A. DATA PRE-PROCESSING

The first phase of our proposed framework is the data pre-processing phase, which addresses class imbalance and data cleanness. In machine learning studies, class imbalance is a known problem that can affect the outcome of a prediction study [42]–[50]. If the datasets applied in a prediction study are properly cleaned and pre-processed, this can facilitate the prediction of defects early in the SDLC, allowing the software team to focus on achieving improved results. Therefore, such measures can promote improved software quality. Recent defect prediction studies have challenged the quality of datasets used in defect prediction and have also demonstrated the need to properly pre-process such datasets [51]. Our dataset pre-processing technique is further discussed in the experimental section.

B. PROPOSED MODELING TECHNIQUE

In this section, we explain the development of the proposed modeling technique in detail. Our modeling technique is based on the Rayleigh distribution curve, which represents the number of defects over time throughout a project [52],

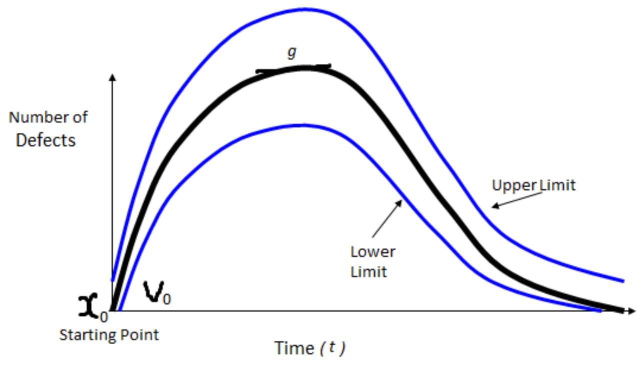

FIGURE 2. The Rayleigh distribution curve, where x_0 = initial defect status, v_0 = initial defect velocity, t = time, and g = defect density.

as shown in Figure 2. This curve reveals how software defects evolve with time throughout the development process. As the phases of software development proceed, the number of errors increases if these errors are not caught and eliminated. Furthermore, the Rayleigh model also shows the relationships between other variables and the number of defects over time throughout the SDLC. These predictor variables were integrated into our models. First, we analyzed the datasets to extract the values of the chosen variables from existing projects. Then, we modeled these variables and applied them in constructing our prediction models.

1) RAYLEIGH DISTRIBUTION CURVE

Figure 2 shows the path representing the defect status of a project over time. Based on this model, we identified the defect density g, defined as the ratio of the number of defects to the size of the project, and the defect velocity v over time t. These metrics were used in constructing our prediction models.

Definitions of Metrics: **Defect density**: The defect density g is the ratio of the number of defects to the project size. The defect density has no unit of measure. **Defect velocity**: The defect velocity v is the rate of change in the defect status with time t and is measured in units of number per day. **Defect introduction time**: The defect introduction time is the time at which defects occur in a software product and is measured in days. **Design complexity**: The module design complexity is defined as the difficulty of constructing a detailed design, which often results in various issues and leads to complicated software products. We derived our predictor variables from the Rayleigh distribution curve presented in Figure 2, with the exception of the design complexity, which is already present in the dataset as one of the attributes. The Rayleigh distribution curve depicts how the number of defects increases over time in a software project. Based on the Rayleigh model, we identified the defect density g, which is defined as the ratio of the number of defects to the size of the project and thus is a function of the number of defects. We also identified the defect velocity v over time t. These metrics were used in constructing our prediction models. The defect density, as defined above, is also illustrated in Figure 2. We evaluated the defect densities of 10 different past projects based on the datasets collected from the PROMISE repository [53]. We also analyzed these datasets to verify their completeness and to determine our other predictor variables. We then calculated the average defect density, average defect velocity and average defect introduction time for each dataset. At the starting point of a project, the number of defects is zero, as shown in Figure 2. The chance of defect introduction increases over time as the project proceeds from one phase to the next. With further phase transitions during software development, the defect acceleration increases. The defect acceleration is the change in the defect velocity at a given instant of time. The increases in the number of defects and the defect density are therefore functions of the defect acceleration, as derived in Equations (1) to (5). The progression through the project phases is accompanied by increases in the numbers of both files and defects. Furthermore, the defects present in these files change from one phase of software development to another. Our model determines the defect status of the files, which enables us to determine the defect introduction time. Accordingly, the average time at which errors appear is obtained, as derived in Equations (9) to (16).

In these equations, the variable x_0 represents the initial defect status of the project, and v_0 and t_0 are the initial defect velocity and the start time of the project, respectively. Using the proposed modeling techniques, the numbers of defects corresponding to the subsequent defect statuses $x_1, \ldots, x_n$ are obtained. These variables are derived based on the software defect density, which is influenced by the defect acceleration of the software project.

The software defect density typically decreases during software development as the competence of the development team increases [10]. Therefore, the relationship of the defect density g with time t depends on the defect acceleration of the project, that is, the velocity v at which defects are introduced with respect to the time of introduction. This implies that the defect density g is a function of the number of defects, which is influenced by the defect acceleration of the project. In this study, the defect acceleration is the basis of all independent variables used to construct our prediction models, and based on these independent variables, our models can be used to predict the number of defects in a new product release. In addition, we believe that these independent variables are directly related to the number of defects. Petrić [54] reported that the potential to predict the number of defects depends on the independent variables considered. Under the assumption of a constant project size, the number of defects depends on the acceleration characterizing defect occurrence.

$$f(no.\ of\ defects) \Rightarrow g$$

This indicates that the defect density g is affected by the number of defects in a project, which, in turn, depends on the defect acceleration.

Thus,

$$f(\textit{defect acceleration}) \Rightarrow \textit{no. of defects}$$

If the defect acceleration is low, then the number of defects will also be low. Accordingly, if the defect acceleration increases, this increased acceleration will lead to an increase in the number of defects in a project and, hence, an increase in the defect density g.

Therefore,

$$f(\textit{defect acceleration}) \Rightarrow g$$

The defect density g is a function of the acceleration characterizing defect occurrence given a constant project size. The defect acceleration can be calculated as the change in defect velocity over the change in time as follows:

$$\textit{defect acceleration} = \frac{\Delta\ \textit{velocity}\ (v)}{\Delta\ \textit{time}\ (t)} \tag{1}$$

At a constant project size, the number of defects and, consequently, the defect density g of a project vary only as functions of the defect acceleration. Thus,

$$g = \frac{\textit{no. of defects}}{\textit{const. project size}} \propto \frac{f(\textit{defect acceleration})}{\textit{const. project size}} \tag{2}$$

For a constant project size, the defect density g, which depends on the defect acceleration, can be expressed as shown in Equations (3) and (4):

$$g = \frac{f(\textit{defect acceleration})}{\textit{const. project size}} \tag{3}$$

$$g = \frac{\Delta \textit{velocity}\ (v)}{\Delta \textit{time}\ (t)} \tag{4}$$

$$v = gt \tag{5}$$

By integrating the defect velocity v over time t, we obtain

$$\int vdt = \int gdt \tag{6}$$

$$g \int dt = gt + c \tag{7}$$

Replacing the constant c with the initial velocity v_0 yields

$$g \int dt = gt + v_0 \tag{8}$$

To predict the defect status x as a function of time t, we use the defect velocity, expressed as

$$v = \frac{\Delta \textit{status}\ (x)}{\Delta \textit{time}\ (t)} \tag{9}$$

Therefore, the target defect status x as a function of time t can be obtained by integrating v over t according to Equation (9). Accordingly, we derive

$$x\ (t) = \int vdt \tag{10}$$

Integrating the defect velocity over time, similar to Equation (6), yields

$$\int vdt = \int gt + v_0 dt \tag{11}$$

Through suitable manipulations of the expression for the defect status x given in Equation (11), we can determine the average defect introduction time t:

$$\int gt + v_0 dt = \int gtdt + \int v_0 dt \tag{12}$$

$$= g \int tdt + \int v_0 dt \tag{13}$$

$$= g\left(\frac{1}{2}t^2\right) + v_0 t + c \tag{14}$$

The constant of integration c becomes the initial defect status x_0. At this initial point, the number of defects and the defect velocity are both equal to zero.

$$x\ (t) = g\left(\frac{1}{2}t^2\right) + v_0 t + x_0 \tag{15}$$

The average defect introduction time t can then be calculated by solving Equation (15) for t. Consequently, we obtain

$$t = \sqrt{\frac{2x}{g}} \tag{16}$$

where x is the defect status and g is the average defect density.

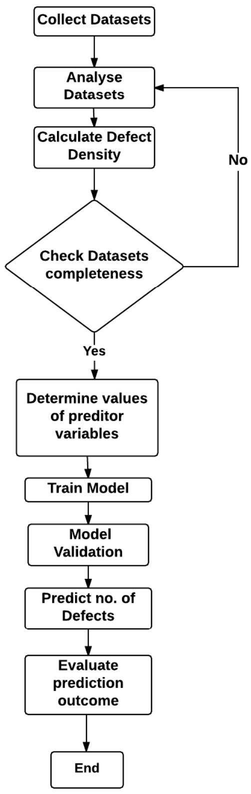

FIGURE 3. Flowchart of the proposed defect prediction model.

2) FLOWCHART OF THE PREDICTION PROCESS

This section presents a simple step-by-step flow diagram for the prediction of the number of software defects. The flowchart of our methodology is shown in Figure 3. This flow diagram begins with data collection and ends with model evaluation for assessing the outcome of the prediction models. The data collection phase is the first step in the flow diagram, in which the source and format of the data are identified. This is followed by data analysis, involving data cleaning and any methods applied to address class imbalance, which remains a challenge in machine learning and data mining studies [55]–[58]. Subsequently, we calculate the average defect density of each dataset. Prior to implementing our modeling technique, we ensure the proper evaluation of our variables from the datasets, including module design complexity. If this evaluation is unsatisfactory, it is necessary to re-examine and analyze the corresponding dataset before applying our modeling technique.

This flowchart clearly illustrates our procedure, from data collection to the prediction of the number of software defects. In this study, in accordance with the first stage of our proposed methodology, after the datasets were collected, we analyzed them to determine the number of defective modules and subsequently calculated the average defect density of each dataset. Before implementing our modeling technique, we ensured that all variables, such as the module design complexity, were properly evaluated for each dataset; otherwise, the dataset was re-examined and analyzed before our modeling technique was applied.

C. PROCEDURE FOR DEFECT PREDICTION

In this sub-section, we present the steps of constructing the models used to predict the number of defects and the steps of pre-processing the datasets. The proposed algorithm can serve as a means to overcome the challenges posed by dataset pre-processing, which is important because software defect prediction models strongly depend on the quality of the data on which they are built.

First, the input variables are the predictor variables derived via our modeling technique, as shown in Equations (1) to (16). Our prediction models are constructed based on these predictor variables, namely, the average defect density g, the average defect velocity v, and the average defect introduction time t, because we believe that these variables are related to the number of defects as a function of the defect acceleration. The defect acceleration is the basis of all of the independent variables used in this study to construct our prediction models, and based on these independent variables, our models can predict the number of defects in a new product release.

The defect density g over time t depends on the defect acceleration of the project, that is, the velocity v at which defects are introduced with respect to the time of introduction. For all datasets 1 to n, denoted by (M_{1-n}), we perform pre-processing and also apply operations $M(i, j)$, where i represents each predictor variable and j represents the calculation operations performed to obtain the value of each predictor variable. Thus, we determine the predictor variables corresponding to each dataset.

Steps 1 to 7 of Algorithm 1 represent the pre-processing of datasets 1 to n to ensure accurate determination of the predictor variables. These predictor variables are the independent variables used to construct our prediction models. The predicted number of defects depends on these independent variables. Steps 8 to 12 represent the construction of a model using the predictor variables. Based on these independent variables, the constructed models can independently learn from the pre-processed data how to predict the number of defects in a new product release.

Algorithm 1 Steps of Predicting the Number of Defects

```
1:  procedure Model Construction For Defect Prediction
    Input: Predictor variables, namely, defect density $g$, defect velocity $v$, and defect introduction time $t$
    Output: Predicted number of defects
2:      pre-process datasets $(M_{1-n})$
3:      determine $(g, v, t)$; see Equations (1) - (16)
4:      for $i = 1$ to $n$ do
5:          for $j = 1$ to $n - 1$ do
6:              if $M(g, v, t)$ could not be determined then
7:                  for every $(M_{1-n})$, recheck $M(i, j)$ to determine $(g, v, t)$ do
8:                      if $M(g, v, t)$ have been determined then
9:                          build a model $f(g, v, t)$
10:                     end if
11:                 end for
12:             end if
13:             Divide $M(i, j)$ into two parts, 80% and 20%, and store in variables Tdata and Vdata, respectively
14:             Tdata ← training (80%)
15:             Vdata ← validation (20%)
16:             train model with $M(i, j)$ (80%)
17:             validate model with $M(i, j)$ (20%)
18:             if $M(i, j)$ validation = successful then
19:                 determine prediction outcome
20:             else
21:                 clean and pre-process dataset $(M_{1-n})$ again
22:             end if
23:         end for
24:     end for
25: end procedure
```

Step 13 represents the division of the datasets into training and validation (or testing) sets. The training datasets are used to train our prediction models. The training data are independent of the testing data; separate training and testing datasets are used to enable the prediction models to learn independently and to ensure that our prediction models are free of overfitting. The datasets selected for training and testing are stored in separate variables.

Steps 14 and 15 indicate the variables in which the training and validation datasets, respectively, are stored. **Tdata** is the variable in which the training datasets are stored, and **Vdata** is the variable in which the validation datasets are stored. Step 16 represents the training of a model using the

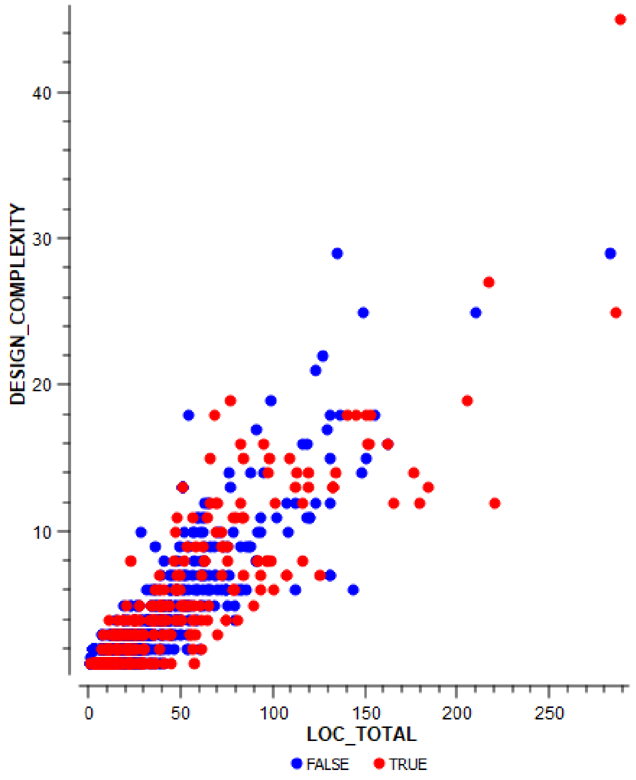

FIGURE 4. Scatter plot showing data imbalance in KC1.

training datasets stored in the **Tdata** variable, which constitute 80% of the total data. Step 17 represents the validation of the trained model using the validation datasets stored in the **Vdata** variable, which constitute 20% of the total data. Finally, steps 18 to 25 represent the generation of the prediction outcome.

V. EXPERIMENTS

We conducted our experiments using ten different NASA datasets that are widely used in software defect prediction studies. In this section, we present the pre-processing of the datasets used to train and validate our models for predicting the number of defects. The experiments were conducted on a desktop computer running 64-bit Windows 10 with an Intel(R) Core(TM) i5-3317U CPU at 1.70 GHz and 4 GB of RAM.

A. DATA COLLECTION PROCEDURE

We collected 10 different datasets from the PROMISE repository [53] for the implementation of our proposed technique. These NASA metric datasets have been widely used in software defect prediction. However, these datasets require a significant amount of pre-processing and cleaning to be suitable for defect prediction. Accordingly, we performed data cleaning as described in this section and data pre-processing as described in the following section.

1) NASA DATASETS ARE OFTEN PROBLEMATIC

Several studies, such as [59] and [60], have indicated that the NASA datasets are problematic. Nevertheless, no software product is free from defects, and no dataset is free from difficulties related to the imbalanced nature of such data. Obviously, data imbalance remains a challenge in defect prediction studies and thus can affect prediction results [42], [44], [45], [48]–[50]. We visualize the imbalanced nature of the NASA datasets in the form of scatter plots in Figures 4-13. To make the datasets suitable for use in this study, we cleaned them using our proposed approach described in Algorithm 2.

Algorithm 2 Data Cleaning for Defect Prediction

```
1:  procedure Steps of dataset cleaning
    Input: Datasets $M$
    Output: Cleaned datasets suitable for prediction based on $A \times B$ cross-validation
2:      A = 10 /* number of folds for cross-validation
3:      B = 10 /* number of repetitions of training/testing
4:      Analyze datasets ($M_{1-n}$) by analyzing each module $CD_{1-n}$ of each ($M_{1-n}$)
5:      for $i = 1$ to $n$ do
6:          Assign a unique ID to each $CD_{1-n}$ of each ($M_{1-n}$)
7:          Check existing attributes in each dataset
8:          Check incomplete and missing values in each module
9:          Check outliers and inliers
10:         Identify defective and defect-free modules $CD_{1-n}$
11:         for $i = 1$ to $n$ do
12:             $f : CD_{1-n} \rightarrow (0, 1)$
13:             if error_count $\geq 1$ then
14:                 $f : (CD) = 1$
15:                 $CD =$ defective
16:                 Record number of defects
17:                 Calculate percentage of defects
18:             else
19:                 $f : (CD) = 0$
20:                 $CD =$ defect-free
21:             end if
22:             Repeat the entire algorithm on each module $CD$ in each dataset $M_{1-n}$ until all datasets have been properly cleaned
23:             Divide $M_{1-n}$ into two parts: 80% and 20%, for training and testing, respectively
24:         end for
25:     end for
26: end procedure
```

B. DATA PRE-PROCESSING

The datasets that served as the input in this study were cleaned as presented in Algorithm 2. We applied a cross-validation sampling strategy with an $A \times B$ validation technique to obtain accurate results, where A is the number of folds and B is the number of repetitions of training or testing. In the cross-validation approach, the data were split into 10 folds, with a relative training set size of 80% and a relative testing set

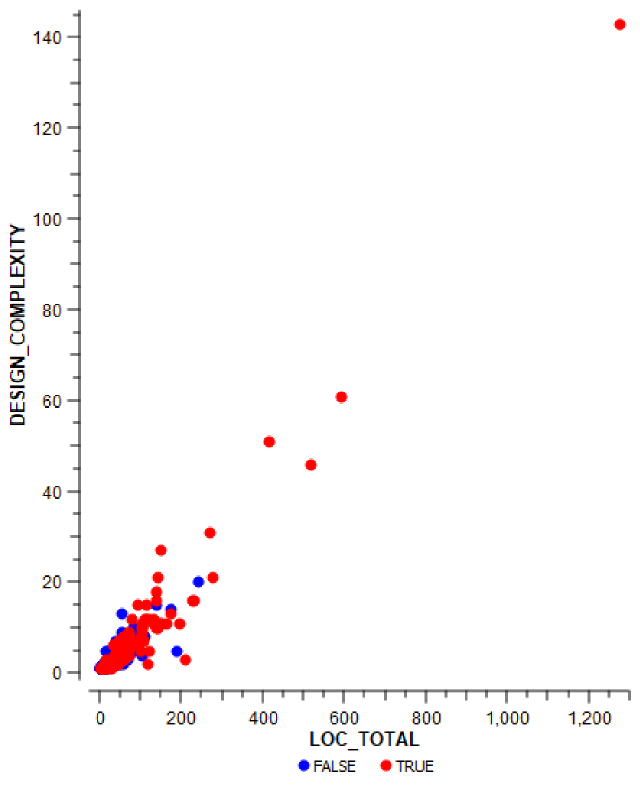

FIGURE 5. Scatter plot showing data imbalance in KC2.

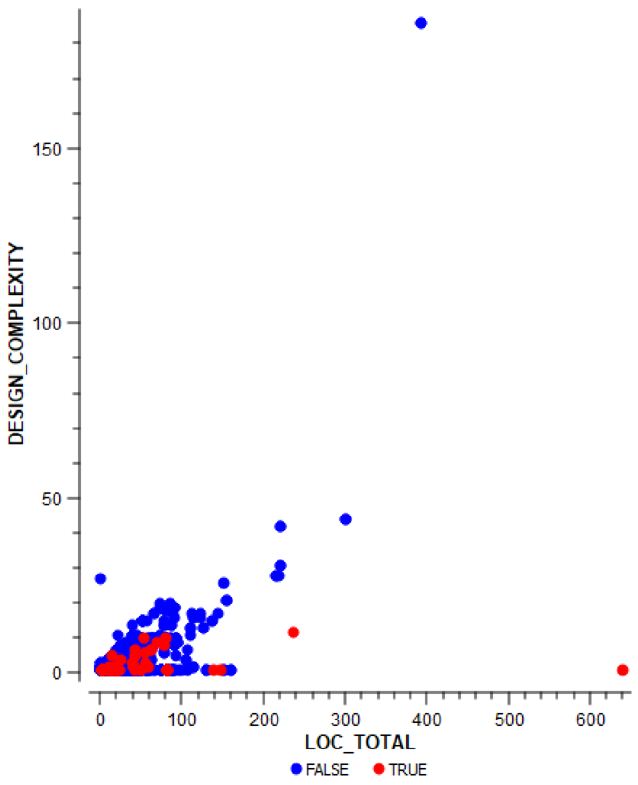

FIGURE 7. Scatter plot showing data imbalance in MC1.

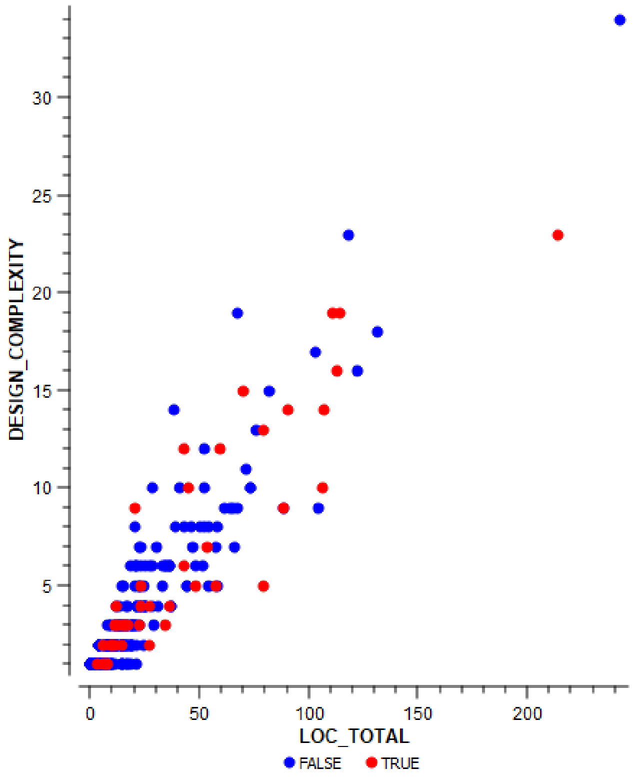

FIGURE 6. Scatter plot showing data imbalance in KC3.

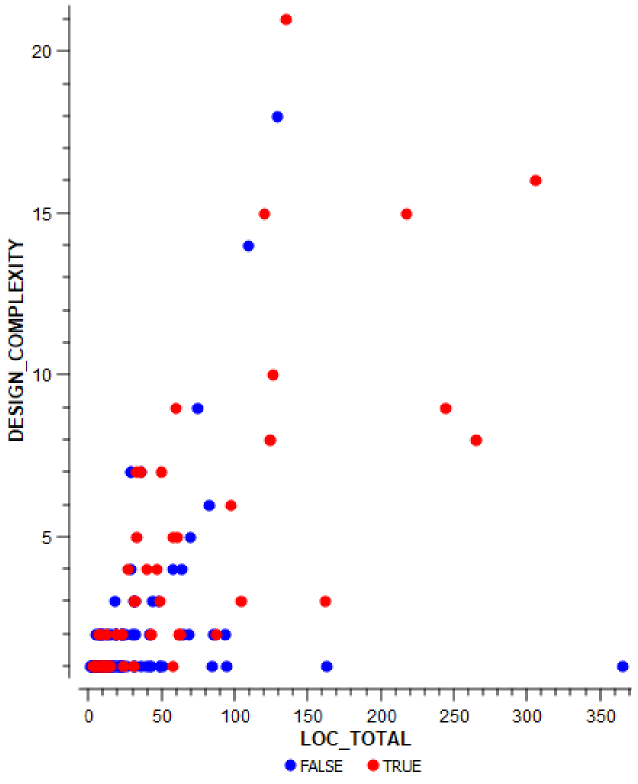

FIGURE 8. Scatter plot showing data imbalance in MC2.

size of 20%. The training data and testing data were independent of each other to ensure unbiased prediction outcomes. We made great efforts to properly pre-process, analyze, and clean the datasets before applying them for the training and validation of our models. Previous studies have faced many criticisms because of the use of biased or noisy datasets.

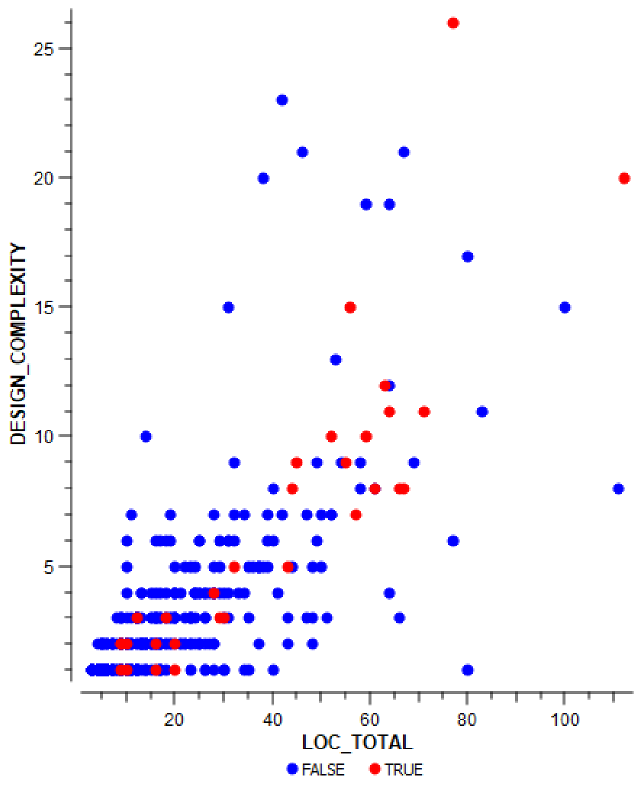

FIGURE 9. Scatter plot showing data imbalance in MW1.

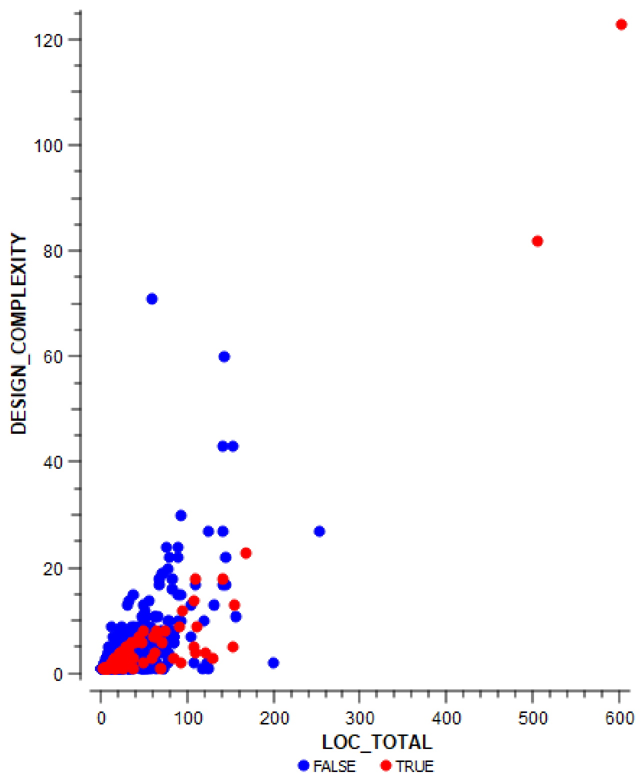

FIGURE 10. Scatter plot showing data imbalance in PC1.

Defect prediction research will not add value to the software research community if the researchers do not spend sufficient time pre-processing their datasets before conducting their experiments. Such negligence may result in poor research findings. To avoid any bias in the results of our study, we fully cleaned and pre-processed our datasets using our systematic technique. To the best of our knowledge, this technique helps to achieve fair results.

First, we determined the source and size of each dataset. In this study, we relied on the widely used NASA datasets from the PROMISE repository. We analyzed each dataset by importing it into a Microsoft Excel spreadsheet to easily identify faulty modules, files, or instances and to identify missing values. We then assigned a numeric identifier to each instance in the dataset; this process was repeated for all datasets. Unique identifiers were assigned to both defective and non-defective modules, which allowed us to easily identify the number of defective instances in each dataset and to avoid repeated data points. Outliers were separated from inliers because only the clustered inliers are suitable to be used as a basis for prediction. The number of defects in each dataset was recorded as well; if the error count among the attributes was greater than or equal to 1, the corresponding module instance was considered defective, whereas an error count of 0 indicated a defect-free module. Thus, we were able to accurately determine the percentage of defective modules in each dataset for comparison with the predicted number of defects.

Second, we verified the number of attributes in each project by importing the pre-processed datasets into Orange 2.7. Files with incomplete attributes were not counted. This process was repeated for each file in every dataset. Although time-consuming, this procedure is necessary to determine whether datasets are suitable for use in training and validation.

We divided the samples into training and testing sets with a 10-fold cross-validation structure to ensure unbiased results. The training sets consisted of 80% of the data and were used to train our models. If models are properly trained, they will produce reliable results during validation; this is why we applied cross-validation sampling when training our models to ensure highly accurate results with regard to model performance.

We constructed nine regression models and applied these models to our datasets, which were divided into training and validation sets at proportions of 80% and 20%, respectively. We first implemented multiple linear regression models, in which multiple independent variables were considered to predict the number of defects. Thereafter, simple linear regression models were applied; in each of these models, only a single independent variable was considered to predict the number of software defects. The results of the predictions were recorded. The details are shown in Tables 3 to 11.

1) HYPOTHESES

The following hypotheses were formulated based on the derived predictor variables:

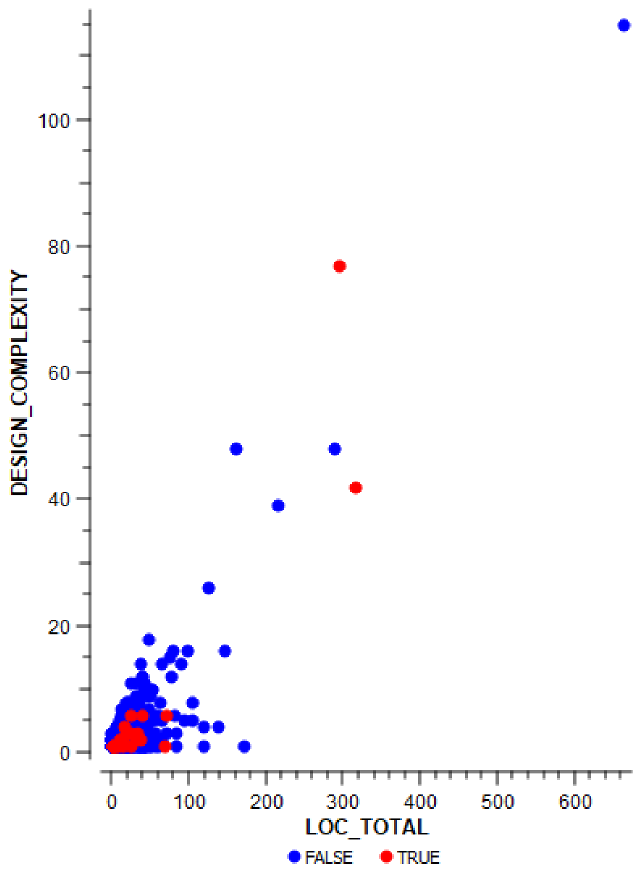

FIGURE 11. Scatter plot showing data imbalance in PC2.

(a). The software defect velocity is expected to influence the number of defects in a software project.

(b). The defect density is not expected to significantly influence the number of defects.

(c). The defect introduction time is not expected to influence the number of defects.

(d). The defect density is expected to increase in a software project with an increasing number of defects.

(e). An increase in the defect acceleration is expected to lead to an increase in the number of defects.

These hypotheses were tested using the models presented as Models 1 to 9. Models 1 to 6 are multiple linear regression models, whereas Models 6 to 9 are simple linear regression models used to enable comparisons with prediction models constructed using the individual metrics. The purpose of both the simple and multiple linear regression models is to predict the number of software defects.

C. PREDICTION MODELS

In this work, both simple and multiple linear regression models were constructed for the prediction of the number of defects in a software project. The multiple linear regression models were constructed for prediction based on two different independent variables, whereas in each simple linear

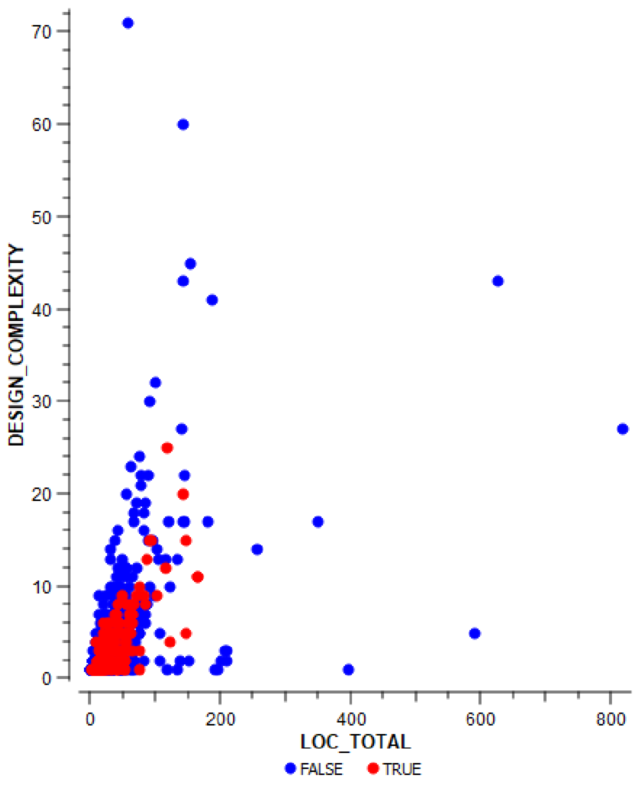

FIGURE 12. Scatter plot showing data imbalance in PC3.

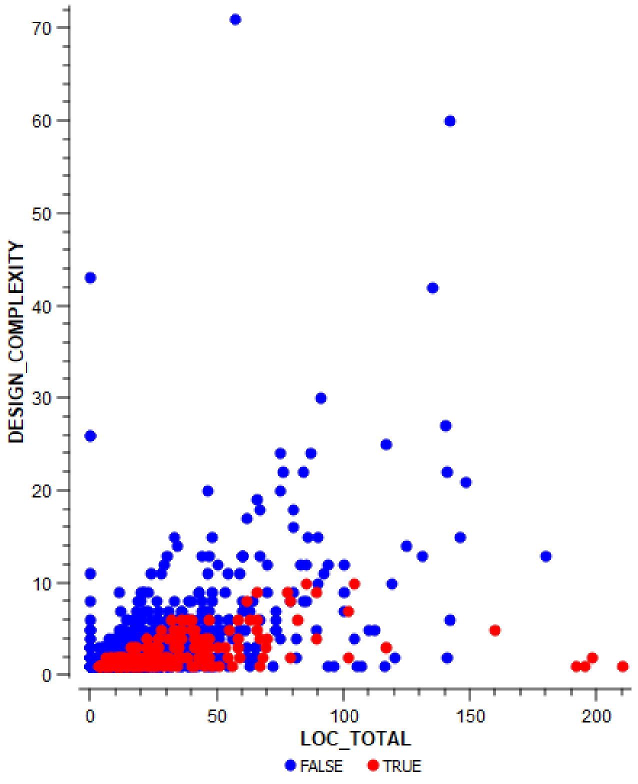

FIGURE 13. Scatter plot showing data imbalance in PC4.

regression model, only one independent predictor variable was considered to investigate the individual influence of each predictor variable.

The training datasets used in building our prediction models are denoted by **Tdata**, and the validation datasets are denoted by **Vdata**. The outcomes of the models built using the training datasets were stored in a variable called **Result**, and the prediction results were stored in a variable called **Pred**.

1) MULTIPLE LINEAR REGRESSION MODELS

The regression models presented as Models 1 to 9 were constructed using the the design complexity, denoted by DC, as well as the derived variables, namely, the average defect introduction time, denoted by ADT; the average defect density, denoted by ADD; and the average defect velocity, denoted by ADV. These variables were used to predict the number of defects, denoted by ND. The models were trained using the 80% of the training data stored in the variable called **Tdata**. The outcomes from training were stored in a variable called **Result**.

Model 1 was built as a function of the design complexity and the average defect introduction time as follows:

$$Result < -lm \sum_{i=1}^{n} ND \sim \sum_{i=1}^{n} DC + \sum_{i=1}^{N} ADT, \; data = Tdata \tag{17}$$

Model 2 was built as a function of the design complexity and the average defect density as follows:

$$Result < -lm \sum_{i=1}^{n} ND \sim \sum_{i=1}^{n} DC + \sum_{i=1}^{n} ADD, \; data = Tdata \tag{18}$$

Model 3 was built as a function of the design complexity and the average defect velocity as follows:

$$Result < -lm \sum_{i=1}^{n} ND \sim \sum_{i=1}^{n} DC + \sum_{i=1}^{n} ADV, \; data = Tdata \tag{19}$$

Model 4 was built as a function of the average defect introduction time and the average defect density as follows:

$$Result < -lm \sum_{i=1}^{n} ND \sim \sum_{i=1}^{n} ADT + \sum_{i=1}^{n} ADD, \; data = Tdata \tag{20}$$

Model 5 was built as a function of the average defect introduction time and the average defect velocity as follows:

$$Result < -lm \sum_{i=1}^{n} ND \sim \sum_{i=1}^{n} ADT + \sum_{i=1}^{n} ADV, \; data = Tdata \tag{21}$$

Model 6 was built as a function of the average defect density and the average defect velocity as follows:

$$Result < -lm \sum_{i=1}^{n} ND \sim \sum_{i=1}^{n} ADD + \sum_{i=1}^{n} ADV, \; data = Tdata \tag{22}$$

2) SIMPLE LINEAR REGRESSION MODELS

Model 7 was built as a function of the average defect density as follows:

$$Result < -lm \sum_{i=1}^{n} ND \sim \sum_{i=1}^{n} ADD, \; data = Tdata \tag{23}$$

Model 8 was built as a function of the average defect velocity as follows:

$$Result < -lm \sum_{i=1}^{n} ND \sim \sum_{i=1}^{n} ADV, \; data = Tdata \tag{24}$$

Model 9 was built as a function of the average defect introduction time as follows:

$$Result < -lm \sum_{i=1}^{n} ND \sim \sum_{i=1}^{n} ADT, \; data = Tdata \tag{25}$$

In the equations above, ND is the number of defects, DC is the design complexity, ADT is the average defect introduction time, ADD is the average defect density, ADV is the average defect velocity, and n is the number of datasets.

The multiple and simple linear regression models were each validated using the 20% of the data stored in the variable **Vdata**, which were independent of the training data used to generate the outcomes stored in the variable **Result**. Thus, the validation process was conducted as expressed below:

$$Pred < -predict \sum_{i=1}^{n} Result, \; Vdata \tag{26}$$

D. MODEL COMPARISON

Throughout our experiments, we applied a cross-validation sampling strategy. Importantly, the performance assessment of any prediction method depends on the data sampling method applied. For our experiments, we chose to apply cross-validation sampling to achieve accurate results and also because, for a large dataset, cross-validation sampling of the data split into 10 folds with a relative training set size of 80% and a testing (validation) set size of 20% performs better than leave-one-out or random sampling.

Estimates of model performance serve to indicate how well a model will perform on unseen data [61]. Li *et al.* [62] used a cross-validation sampling technique for their performance assessment. In cross-validation, separate training and validation datasets are used to prevent the dependence of either one on the other, and consequently, this approach is known to be nearly unbiased. However, smaller samples may tend to show wider variance [63]. Isaksson *et al.* [64] have therefore argued that neither bootstrapping nor cross-validation is reliable when dealing with small samples. Nevertheless, unstable results obtained through cross-validation can be stabilized by repeating the validation process several times, as in, for example, the 10-fold cross-validation applied in this study.

In the cross-validation sampling approach, measures of both fit and prediction error are considered to achieve a more accurate estimation of model prediction performance [65].

It is meaningful to compare the metrics proposed in this study with static code metrics and process metrics. Both static and process metrics are useful in defect prediction studies. On the one hand, static metrics are widely applied in practice as a safe alternative for predicting defective programs and can also assist software developers in locating faults in their programs [66], [67]. These static metrics are useful to a certain extent, but their prediction accuracy decreases over time [68]–[72]. On the other hand, process metrics are applied to measure the quality of a system because they capture more details about faulty code. Process metrics show good potential for making predictions regarding post-product release status and are useful for large datasets [30], [73]. However, despite the potential of the existing metrics, additional studies are still needed to find metrics that show further relevance to the industry that is yet to be provided by the existing static and process metrics [73]. Therefore, our aim with our proposed metrics is to fill this gap by clearly demonstrating their influence on the number of software defects and, thus, their potential to help software companies improve software standards.

We evaluated the performance of our models using the p-value, the adjusted R-square and the F-statistic. The p-value is used to indicate how well a model performs; it represents the statistical significance of a model [74], [75]. The F-statistic enables a comparison of the average significance of the variables used in constructing the models. The adjusted R-square measures the goodness of fit of a model and also indicates the influence of a significant variable in a model. We considered the adjusted R-square in this study because its value increases only when a significant variable is included in a model. The Mean Magnitude of Relative Error (MMRE) is the most widely used evaluation measure for determining the performance of competing software prediction models, and one of its purposes is to help a software team to select the best-performing model. However, we did not apply the MMRE as an evaluation criterion in this study because the findings of several previous studies, such as [76] and [77], suggest that the MMRE may be unreliable under certain conditions, leading to the selection of the worse candidate between two competing models; in particular, the MMRE tends to prefer a model that underestimates to a model that accurately estimates the expected value. These studies cast doubt on the results of previous studies that have relied on the MMRE to compare the accuracy of different prediction models.

VI. RESULTS AND DISCUSSION

In this section, we present our hypotheses and preliminary results. First, we set a null hypothesis H_0 that our chosen predictor variables may not perform well. This hypothesis corresponds to the assumption that our predictor variables do not affect the prediction model. Thus, any metric that cannot achieve a p-value of 0.001 is of negligible significance. This null hypothesis was used for significance testing based on our results. The results indicated that the null hypothesis H_0 does not hold because the alternative hypotheses achieved significant p-values. We used 80% of the samples for training and 20% for validation. A substantial number of samples were used for training to ensure that the models were not built using noisy data and that our training data would not produce any biased output during model validation. The pre-processing of the datasets assisted in obtaining reliable results that could lead to unbiased prediction outcomes. We used both multiple and simple linear regression models to predict the number of defects based on different independent variables. Tables 3 to 11 present the results obtained when using our trained models to predict the numbers of defects for the 20% of the data designated for validation.

Table 1 presents the statistics of the datasets collected from the PROMISE repository in terms of the number of files, the number of defects, the module design complexity, the number of existing attributes, and the percentage of defects in each dataset.

TABLE 1. Statistics of the datasets collected from the PROMISE repository.

Dataset	No. of Instances	No. of Defects	Design Complexity	No. of Attributes	% Defects
KC1	2109	326	3359	21	15.46
KC2	522	107	1905	21	20.50
KC3	458	43	1378	39	9.39
MC1	9466	68	1475	38	0.72
MC2	161	48	429	39	29.81
MW1	403	31	1340	37	7.69
PC1	1109	77	3475	21	6.94
PC2	5589	21	1370	36	0.38
PC3	1563	159	2152	37	10.17
PC4	1458	174	2147	37	11.93
Total	22,838	1054	19,030	326	4.62

Table 2 presents the values of the predictor variables used in our modeling technique, namely, the average defect density, average defect introduction time, and average defect velocity, for each dataset.

TABLE 2. Values of the predictor variables for each dataset.

Dataset	Average Defect Density	Average Defect Introduction Time (Days)	Average Defect Velocity (Number per Day)
KC1	0.1545	165	25.49
KC2	0.2050	71	14.56
KC3	0.0934	99	9.25
MC1	0.0072	1622	11.68
MC2	0.2981	33	9.84
MW1	0.0769	102	7.84
PC1	0.0694	179	12.42
PC2	0.0038	1715	6.52
PC3	0.1017	175	17.80
PC4	0.1193	156	18.61

Table 3 presents the predicted numbers of defects as well as the F-statistic, adjusted R-square, and p-value results for the Model 1 predictions, based on the design complexity and the average defect introduction time. We obtained these results using the 20% of our datasets set aside for validation. We cleaned and trained our datasets repeatedly using the same ratio of 80% for training and 20% for validation. Note that the KC1 and PC2 datasets constitute the 20% of the data used to validate our prediction models. The results presented in Tables 3 to 4 provide an idea of the possible number of software defects that are likely to be present in a new product release, as obtained from the models built using the proposed independent variables. These results can support software project stakeholders in decision-making and in allocating available resources to new software projects.

TABLE 3. Predictions based on design complexity and average defect introduction time (Model 1).

Dataset	No. of Defects	Predicted No. of Defects	F-statistic	Adjusted R-square	p-value
KC1	326	465	0.6116	-0.1248	0.57
PC2	21	15	0.6116	-0.1248	0.57

TABLE 4. Predictions based on design complexity and average defect density (Model 2).

Dataset	No. of Defects	Predicted No. of Defects	F-statistic	Adjusted R-square	p-value
KC1	326	146	0.7821	-0.0664	0.51
PC2	21	12	0.7821	-0.0664	0.51

Table 4 presents the predicted numbers of defects as well as the corresponding F-statistic, adjusted R-square, and p-value results for the Model 2 predictions. The performance of this model is merely fair, with a p-value of 0.51, compared with the p-value of 0.57 achieved by the model represented in Table 3. Thus, the design complexity and average defect density are apparently less strongly correlated with the number of defects than the design complexity and average defect introduction time are, although the former two metrics can nevertheless be used to build a prediction model.

TABLE 5. Predictions based on design complexity and average defect velocity (Model 3).

Dataset	No. of Defects	Predicted No. of Defects	F-statistic	Adjusted R-square	p-value
KC1	326	260	224.7	0.9846	p<0.001
PC2	21	14	224.7	0.9846	p<0.001

Table 5 presents the predicted numbers of defects as well as the corresponding F-statistic, adjusted R-square, and p-value results for the Model 3 predictions. The performance of this model (with a p-value of < 0.001) is significant, unlike that of the previous models. Thus, this prediction model based on the design complexity and average defect velocity achieves high values of the F-statistic and adjusted R-square, and it achieves significant prediction outcomes.

TABLE 6. Predictions based on average defect introduction time and average defect density (Model 4).

Dataset	No. of Defects	Predicted No. of Defects	F-statistic	Adjusted R-square	p-value
KC1	326	315	0.0421	-0.3768	0.96
PC2	21	16	0.0421	-0.3768	0.96

Table 6 shows that Model 4, based on the average defect introduction time and average defect density, achieves a p-value of 0.96 and low values of the adjusted R-square and F-statistic. Consequently, the joint impact of these variables on the prediction outcome appears to be weak based on these measures. However, this combination of variables can still be used to build a prediction model.

TABLE 7. Predictions based on average defect introduction time and average defect velocity (Model 5).

Dataset	No. of Defects	Predicted No. of Defects	F-statistic	Adjusted R-square	p-value
KC1	326	261	236.8	0.9854	p<0.001
PC2	21	7	236.8	0.9854	p<0.001

Table 7 presents the results for Model 5, which shows significant prediction performance in terms of the p-value, adjusted R-square and F-statistic. The high value of the adjusted R-square and the p-value of <0.001 indicate the good performance of this model. Moreover, the significance of this model as measured based on the F-statistic is also high. Therefore, the average defect introduction time and average defect velocity show a high joint influence on the number of defects.

TABLE 8. Predictions based on average defect density and average defect velocity (Model 6).

Dataset	No. of Defects	Predicted No. of Defects	F-statistic	Adjusted R-square	p-value
KC1	326	260	200.3	0.9827	p<0.001
PC2	21	14	200.3	0.9827	p<0.001

Table 8 presents the results for Model 6, based on the average defect density and average defect velocity. This model performs similarly to the models represented in Tables 5 and 7, with a high p-value and adjusted R-square of <0.001 and 98.27%, respectively. Moreover, the derived predictor variables show consistent performance in terms of the p-value, adjusted R-square and F-statistic, as seen from Tables 5, 7, 8 and 11. The models presented in all of these tables show consistent performance as a result of the influence of the average defect velocity, which is included in all of these prediction models.

TABLE 9. Predictions based on average defect density (Model 7).

Dataset	No. of Defects	Predicted No. of Defects	F-statistic	Adjusted R-square	p-value
KC1	326	315	0.00086	-0.1665	0.98
PC2	21	19	0.00086	-0.1665	0.98

Tables 9 to 11 present the outcomes of the simple linear regression models, which enable a comparison of the influences of the individual predictor variables.

TABLE 10. Predictions based on average defect introduction time (Model 8).

Dataset	No. of Defects	Predicted No. of Defects	F-statistic	Adjusted R-square	p-value
KC1	326	322	0.0583	-0.1554	0.82
PC2	21	16	0.0583	-0.1554	0.82

Tables 9 and 10 present the prediction outcomes based on the average defect density and the average defect introduction time, respectively. The average defect density individually achieves a p-value of 0.98 and an F-statistic of **0.00086**. Similarly, as shown in Table 10, the average defect introduction time individually achieves a p-value of 0.82 and an F-statistic of **0.0583**. Although these metrics show low correlations with the number of defects, the models built using these metrics nevertheless predict a noteworthy number of defects. Thus, both metrics can be useful when building defect prediction models.

TABLE 11. Predictions based on average defect velocity (Model 9).

Dataset	No. of Defects	Predicted No. of Defects	F-statistic	Adjusted R-square	p-value
KC1	326	261	479.9	0.9856	p<0.001
PC2	21	14	479.9	0.9856	p<0.001

Finally, Table 11 presents the best-performing prediction model in terms of the F-statistic, adjusted R-square, and p-value. The performance of this model is outstanding, with a p-value of <**0.001** and an F-statistic of **479.9**, which indicate that this model yields significant results in predicting the number of defects. Unlike the other individual variables, the average defect velocity is strongly positively correlated with the number of defects, with a high positive correlation coefficient of 0.98.

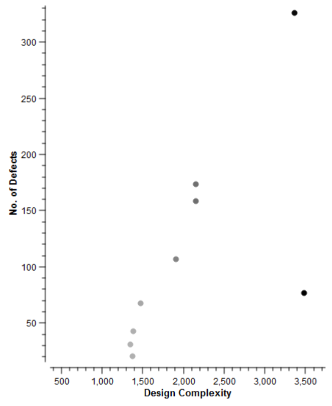

FIGURE 14. **Effect of design complexity on number of defects.**

Figure 14 presents the effect of the design complexity on the number of defects. From this graph, we conclude that the design complexity exhibits a moderate positive correlation of 0.67 with the number of defects. By contrast, Figure 15 shows that the average defect velocity presents a strong positive correlation of 0.98 with the number of defects because the number of defects monotonically increases as the defect velocity increases. The effect of time on the number of defects is shown in Figure 16. This graph shows that time exhibits a weak negative correlation of -0.30 with the number of defects. Meanwhile, the defect density shows a weak positive correlation coefficient of 0.22 (Figure 17).

The predictor variables considered in this study, namely, the design complexity, average defect velocity, average defect introduction time, and average defect density, are presented in Figures 14 to 17, respectively. However, none of these variables except the average defect velocity shows a linear relationship with the number of defects. This result indicates that if defects are not eliminated early in the SDLC, then correcting them in a later phase may prove difficult. We therefore conclude that the average defect velocity is strongly positively correlated with the number of defects in any given project. By contrast, the other variables show moderate, negative, or weak correlations with the number of defects. Figures 18 to 21 present the statistics of the variables of design complexity, average defect velocity, average defect introduction time and average defect density, respectively. These figures show the range of values for each variable as well as the mean and median values of the distributions.

The novelty of this work lies in the chosen variables, the applied modeling technique and analysis, and the cleaning and pre-processing of the datasets. The novel ideas presented in this paper can also serve as the foundation for a new area of research that may yield more reliable and actionable outputs

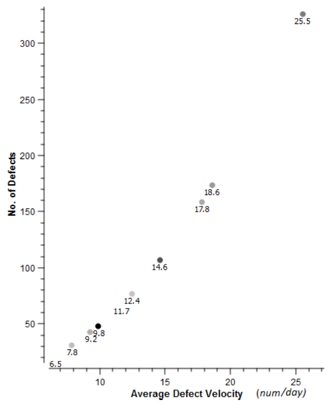

FIGURE 15. **Effect of average defect velocity on number of defects.**

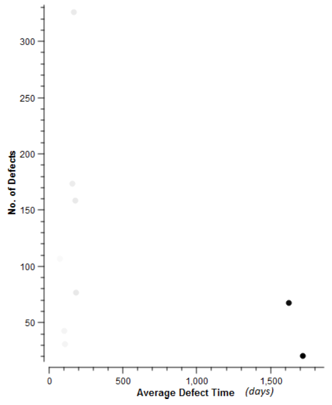

FIGURE 16. **Effect of average defect introduction time on number of defects.**

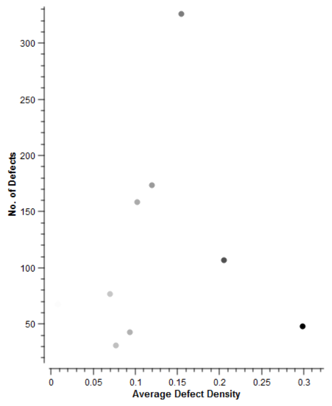

FIGURE 17. **Effect of average defect density on number of defects.**

within the software engineering community. Consequently, our ideas can improve the quality of software products. If software companies are provided with a proper technique for predicting the number of software defects in advance, then such a technique can be beneficial for decision-making by software managers, thereby reducing the cost of software development and improving the quality of software products. In addition, if defects can be predicted early in the SDLC,

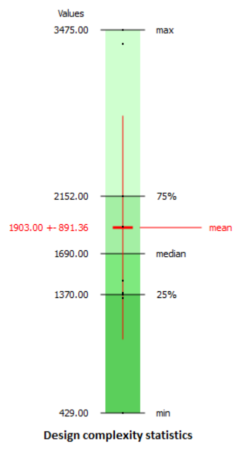

FIGURE 18. Design complexity.

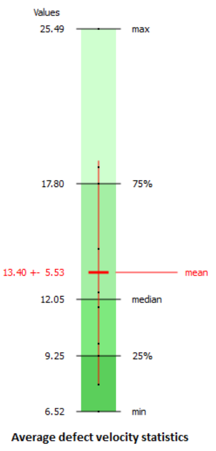

FIGURE 19. Average defect velocity.

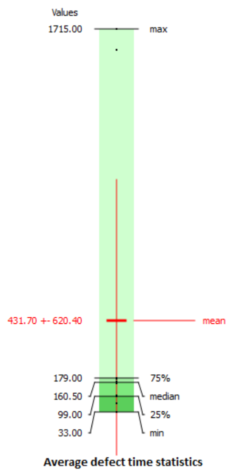

FIGURE 20. Average defect introduction time.

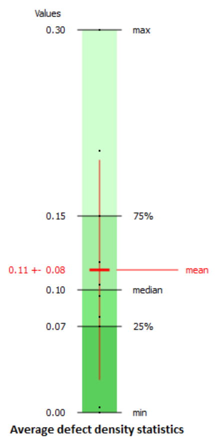

FIGURE 21. Average defect density.

development teams can focus on improving on their ability to deliver more reliable software products by preventing these defects from ever arising in their software systems. Such efforts will also address the majority of software complexity issues caused by software defects.

Throughout the experiments conducted in this study, the pre-processing of the data played a significant role and required substantial attention because if a model is trained with noisy data, then that model will never produce accurate results; therefore, any report on such a study will always raise eyebrows among the research community. Consequently, we applied great effort in pre-processing our datasets before dividing them into training and validation sets. Data quality is one of our major concerns because if the data used in a study are of low quality, then that study will be unable to produce viable discoveries. In our study, we found that most studies are biased because the data used are often noisy. In addition, we observed that the pre-processing of datasets is a challenging task, which is why in most studies, insufficient time is spent on refining datasets.

Considering the number of predictor variables applied in this work, our findings obtained using the proposed models demonstrate that the software defect velocity variable is the predominant contributor to the emergence of software defects, as seen from its strong positive correlation coefficient. These defects affect software quality. Therefore, there is a need to predict software defects early in the SDLC to address the complexity issues facing software products.

By comparing the results obtained in this study with existing work, we can conclude that our results represent a unique approach for estimating the number of software errors in a new product release. Although state-of-the-art classifiers previously applied in defect prediction have achieved noteworthy performance on binary classification tasks, these classifiers do not provide methods for estimating the numbers of defects in the software modules that are predicted to be defective. This study has presented an approach that can be used to predict the number of faults in a defective program or in a new software product based on the predictor variables used to construct the prediction models proposed herein.

These predictor variables, namely, the defect density, defect velocity and defect introduction time, have not been fully integrated into previous prediction models to demonstrate how they influence the number of defects in an upcoming product release. This is why the purpose of this study was to investigate the use of these variables in predicting the number of defects in a new software product to assist software managers in their decision-making. This study considered a multiple linear regression approach for estimating the number of software defects to permit the integration of more than one variable when building our prediction models. The results suggest that the defect velocity strongly influences the number of defects in a software project. Given the strong positive correlation between the defect velocity and the number of defects, software development teams should better focus on the rate at which software projects transition from one phase to the next. This is because the number of defects that arise in a project is a function of the rate at which defects occur. In addition, the defect density of a project depends on the number of defects present in the software.

The number of software defects depends on the rate at which these defects occur. Therefore, there is a need to investigate this rate, called the defect velocity in this study. This study also indicates that any software product with a high defect density shows a high number of defects and vice versa. If the factors determining the rate at which software defects occur can be identified, this will enable developers to restructure their testing and debugging efforts into a more cost-effective approach to deliver higher-quality software products in subsequent releases.

The proposed technique attempts to predict not only the number of defects but also the rate at which these defects occur over time. This achievement is a result of the careful selection, integration and modeling of the predictor variables considered in this study for predicting the number of software defects as well as the determination of the influence of each of these variables. Based on these carefully selected variables, this study indicates that using datasets as they exist in the repository without adequate cleaning and pre-processing could yield deceiving results. Therefore, researchers are encouraged to pre-process their datasets carefully before using them as inputs to models. In this way, properly cleaned and pre-processed datasets will allow models to learn independently of the peculiarities of the specific data used, thereby making the prediction outcomes more reliable.

A cross-validation approach, such as that applied in this study, relies on data cleanliness to avoid overfitting because if a model is applied to the same dataset on which it was trained, the model will achieve a perfect score, but its prediction outcomes may be useless. Unclean data are, essentially, biased data that can affect decision-making in industry. Consequently, the quality of decision-making can be a function of the quality of the dataset used in the study on which that decision-making is based. Hence, overfitting on unclean data may have a negative impact on decision-making; therefore, it is important to properly clean and pre-process datasets before applying them in any prediction model.

The approach proposed in this study attempts to solve the problem of dataset reliability through the top-down pre-processing and cleaning of the datasets. The datasets used in this study were thoroughly cleaned before our predictor variables were calculated. The number of defects in a software project is sensitive to these variables. The findings based on these predictor variables suggest that the number of defects in a software project is a function of the defect acceleration; when a software project begins, the likelihood of defects occurring is high.

TABLE 12. Correlation coefficients of the predictor variables with the number of defects.

Variable	Correlation Coefficient
Design complexity	0.67
Average defect time	-0.30
Average defect density	0.22
Average defect velocity	0.98

Table 12 presents the correlation coefficients between the number of defects and the variables used to build the models. Among the considered predictor variables, the defect velocity exhibits the strongest correlation with the number of software defects, with a correlation coefficient of 0.98, as shown in Table 12. The rate at which defects occur in a software project is characterized by the defect velocity. Thus, this study has proven that the defect velocity contributes to the number of defects in a software product. Clearly, the defect density increases as the rate at which defects occur increases and thus is a function of the defect velocity. If the defect density of a software project is low, then this means that the rate at which defects occur is also low, suggesting that the defect density is also a function of the defect velocity with respect to time. In addition, the results of this study indicate that among the tested models, the prediction model based on the average defect velocity is the best-performing model. This model achieved a high adjusted R-square of 98.6% and a p-value of <0.001, indicating significant performance. These results confirm that the defect velocity affects the number of defects in a software project, as shown by the correlation coefficient of 0.98. In this study, we have demonstrated how the number of software defects in an upcoming product release can be predicted using the derived variables. This demonstration clearly illustrates that the proposed prediction approach provides actionable outputs for software companies as well as a blueprint for program testing to enhance the effectiveness of software development activities.

A. LIMITATIONS OF STATISTICAL METHODS IN MACHINE LEARNING

Most existing prediction models show weak performance as a result of their inability to capture the as-yet-unknown correlation between defects and failures [78]. In our opinion, this is because the predictor variables that actually influence the number of defects in a software product have not been fully integrated in the construction of a suitable prediction model. In this study, we have integrated such predictor variables when building our proposed models; according to our findings, the average defect velocity, that is, the rate at which defects occur over time, exerts the greatest influence on the number of defects in a software product. Based on our findings, a prediction model built using only the defect velocity or the defect velocity in combination with other predictor variables tends to produce reliable prediction outcomes. However, statistical approaches such as those applied in this study are subject to certain limitations, including limitations related to multicollinearity, model fitting and the quality of the data points.

1) MULTICOLLINEARITY

One of the most common problems encountered with the application of statistical methods in the existing literature, multicollinearity occurs when two or more predictor variables are highly positively or negatively correlated with each other [78]. By contrast, linear regression models rely on the assumption that the correlation between predictor variables is always zero, meaning that they are independent of each other [79]. When multicollinearity arises in a statistical analysis, it may lead to inconsistent signs of the coefficients and misleading conclusions. In our study, the correlation coefficient between the number of defects and the average defect introduction time had a negative value of $\mathbf{-0.30}$; this negative correlation was consistent when tested on both the NASA metric data and the ELFF datasets. Notably, collinearity between variables has previously been reported, namely, a negative correlation between the defect density and the module size [80]. The author of that study reported that since there must be a negative correlation between the module size X and $\frac{1}{X}$, a negative correlation is consequently expected between X and the defect density $\frac{Y}{X}$ (number of defects/size). We can similarly conclude that the average defect introduction time should have a negative correlation with the number of defects. Both the simple and multiple linear regression models applied in our study predicted the number of defects with little or no multicollinearity between the predictor variables. The values of the correlation coefficients reported in Table 12 confirm the influence of each predictor variable as presented in Figures 14-17.

2) MODEL FITTING

Almost all evaluations of regression models are primarily concerned with model fitting, with less focus on the successful prediction of the expected result. Furthermore, attempts are typically made to demonstrate how well these models explain the historical data through least-squares fitting and the goodness of fit. In this study, we instead focused on accurately predicting the number of defects and determining the effect of each predictor variable on the number of defects. A reliable model is one that is capable of predicting the number of defects in a software module [78]. Because of a lack of reliable data, the authors of some existing studies may have used only their own data for model fitting without performing a proper evaluation of these models on newer datasets, such as in [81] and [82]. The development of the prediction models proposed in this study was aimed at accurately predicting the number of software defects rather than determining how well a model fit the data.

3) QUALITY OF DATA POINTS

It is somewhat challenging to determine which studies in the existing literature have included procedures for controlling the quality of the data points during pre-processing, although if such procedures were applied, some justification for doing so should be provided [78]. In our case, we addressed the quality of the data points when applying our models to the NASA datasets; some data points were removed during the training phase due to the identification of incomplete attributes because such incomplete data may impact model performance. These data points were removed to enable us to gain full knowledge of the data while ensuring that the data analysis was performed using datasets with complete attributes and that the imbalanced nature of the data could be properly characterized. The data imbalance is clearly presented in Figures 1-10, and our data pre-processing steps are presented in Algorithm 2.

B. BENEFITS OF THE PROPOSED MACLI APPROACH

In terms of cost and quality, our proposed **MACLI** approach represents an easy-to-use and cost-effective data pre-processing technique that can assist software companies in gaining a comprehensive and accurate understanding of their existing datasets and in performing appropriate pre-processing thereof. At the same time, our proposed approach requires fewer lines of code and less time and energy than existing methods. Generally, good prediction models are capable of learning independently and accurately from reliable data such that they are able to produce reliable prediction results. Furthermore, this approach can assist managers in decision-making regarding the allocation of available resources to software projects. The experimental results obtained in this study suggest that the defect velocity is the primary factor responsible for the number of defects in a software project; consequently, software companies of all sizes must pay adequate attention to the rate at which software projects transition from one phase to another. This implies that a software project must not be executed using a slow approach; rather, more attention should be paid in every stage of the development process to reduce errors. However, the initial phase of a software project may require more than the available number of experts to accurately determine whether a project contains nearly zero defects and thus is ready to transition to the next development phase. If an enormous software project is handled by fewer than the required number of experts, then that project may be more prone to errors, whereas a project that is attended to by the required number of experts is capable of approaching nearly zero defects and can be delivered to the end user on time.

The challenges posed by software defects remain an important issue to be addressed in software engineering. Software companies continue to seek possible ways of reducing or eliminating errors in software products. The regression models applied in this study can assist managers in forecasting the future status of software products by predicting the number

TABLE 13. **ELFF dataset statistics.**

Project Name	No. of Classes	No. of Defects	No. of Attributes	% Defects	Defect Density	Defect Time (days)	Defect Velocity (num/day)
AutoPlot 2012	2800	192	42	6.86	0.0686	286	19.62
Cdk1	1678	0	42	0	0	0	0
Cdk1.1	1670	261	42	15.63	0.1563	146	22.82
Cdk1.2	1717	0	42	0	0	0	0
Cmusphinx3.6	665	15	42	2.26	0.0226	243	5.49
Cmusphinx3.7	665	10	42	1.5	0.015	298	4.47
Controltier3	1659	0	42	0	0	0	0
Controltier3.1	1658	117	42	7.06	0.0706	217	15.32
Controltier3.2	1683	0	42	0	0	0	0
Drjava2008	2697	1013	42	37.56	0.3756	120	45.07
Drjava2009	3196	774	42	24.22	0.2422	162	39.24
Drjava2010	3549	403	42	11.36	0.1136	250	28.4
Eclemma2	196	9	42	4.59	0.0459	92	4.22
Eclemma2.1	233	37	42	15.88	0.1588	54	8.58
Genoviz5.4	1111	827	42	74.44	0.7444	55	40.94
Genoviz6	1077	840	42	77.99	0.7799	53	41.33
Genoviz6.1	1059	817	42	77.15	0.7715	52	40.12
Genoviz6.2	1156	306	42	26.47	0.2647	93	24.62
Genoviz6.3	1242	226	42	18.2	0.182	117	21.29
HTMLUnit2008	497	427	42	85.92	0.8592	34	29.21
HTMLUnit2009	1059	121	42	11.43	0.1143	136	15.54
HTMLUnit2010	1296	364	42	28.09	0.2809	96	26.97
JEdit5.2	1265	13	42	1.03	0.0103	496	5.11
Jikesrvm2	1332	107	42	8.03	0.0803	182	14.61
Jikesrvm3	2098	440	42	20.97	0.2097	141	29.57
Jikesrvm3.1	2290	71	42	3.1	0.031	384	11.9
Jitterbit1.1	6141	533	42	8.68	0.0868	376	32.64
Jitterbit1.2	12247	145	42	1.18	0.0118	1441	17
Jmol2	268	38	42	14.18	0.1418	61	8.65
Jmol3	275	35	42	12.73	0.1273	66	8.4
Jmol4	297	81	42	27.27	0.2727	47	12.82
Jmol5	324	92	42	28.4	0.284	48	13.63
Jmol6	378	294	42	77.78	0.7778	31	24.11
Jmol7	405	248	42	61.23	0.6123	36	22.04
Jmol8	453	145	42	32.01	0.3201	53	16.97
Jmol9	459	176	42	38.34	0.3834	49	18.79
Jmol10	556	107	42	19.24	0.1924	76	14.62
Jmri2	2730	124	42	4.54	0.0454	347	15.75
Jmri2.2	3337	175	42	5.24	0.0524	357	18.71
Jmri2.4	3727	1388	42	37.24	0.3724	141	52.51
Jmri2.6	4059	252	42	6.21	0.0621	362	22.48
Jppf4	1933	258	42	13.35	0.1335	170	22.69
Jppf4.1	1934	133	42	6.88	0.0688	237	16.31
Jppf4.2	1956	135	42	6.9	0.069	238	16.42
Jppf5	1879	274	42	14.58	0.1458	161	23.47
Jppf5.1	1912	55	42	2.88	0.0288	364	10.48
Jtds23072009	156	35	42	22.44	0.2244	37	8.3
Jump1.5	2791	131	42	4.69	0.0469	345	16.18
Jump1.6	2909	104	42	3.58	0.0358	403	14.43
Jump1.7	3016	96	42	3.18	0.0318	436	13.86
Jump1.8	3064	107	42	3.49	0.0349	419	14.62
Jump1.9	3231	281	42	8.7	0.087	273	23.75
OmegaT3.1	1204	45	42	3.74	0.0374	254	9.49
OmegaT3.5	1391	96	42	6.9	0.069	201	13.87
OmegaT3.6	1476	97	42	6.57	0.0657	212	13.93
Runawfe3.5	5029	0	42	0	0	0	0
Runawfe3.6	5237	5	42	0.1	0.001	3236	3.24
Runawfe4.1	2882	369	42	12.8	0.128	212	27.14
Runawfe4.2	3408	0	42	0	0	0	0
Saros1.0.6	329	69	42	20.97	0.2097	56	11.74
Tango2008	4299	151	42	3.51	0.0351	495	17.37
Unicore1.2	412	90	42	21.84	0.2184	61	13.32
Unicore1.3	466	54	42	11.59	0.1159	90	10.43
Unicore1.4	477	202	42	42.35	0.4235	47	19.9
Unicore1.5	728	69	42	9.48	0.0948	124	11.76
Unicore1.6	834	234	42	28.06	0.2806	77	21.61
Xaware5	882	120	42	13.61	0.1361	114	15.52
Xaware5.1	994	51	42	5.13	0.0513	197	10.11
Xaware6	1001	0	42	0	0	0	0

of defects that are likely to be present in a new product release. In addition, these prediction models can support decision-making regarding the proper allocation of available resources in a software project, which will lead to a more profitable future for software companies. Because of the variety of choices of regression models provided in this study for predicting the number of defects, our proposed models also have the potential to yield new insights for software teams by uncovering the relationships between the variables that influence the number of defects; these variables had not previously been considered, but our prediction models close this gap.

C. VALIDATION OF THE PROPOSED APPROACH

To validate the proposed approach, we present experimental results obtained using newer datasets. We have applied the proposed technique to the **ELFF** datasets, as presented in Table 13. The results obtained further support our findings based on the **NASA** datasets. From the results, we can conclude that as the software defect velocity increases, the number of defects in a software product also increases, and vice versa. In addition, we have presented the steps of our data cleaning process in Algorithm 2 to clearly illustrate the approach we applied to make these data suitable for use in defect prediction.

Table 13 presents the statistics of the 69 datasets used in this study. The columns in the table, from the first to the last, report the project name, the number of classes, the number of defects, the number of attributes, the percentage of defects, the average defect density per class, the defect introduction time, and the average defect velocity per class.

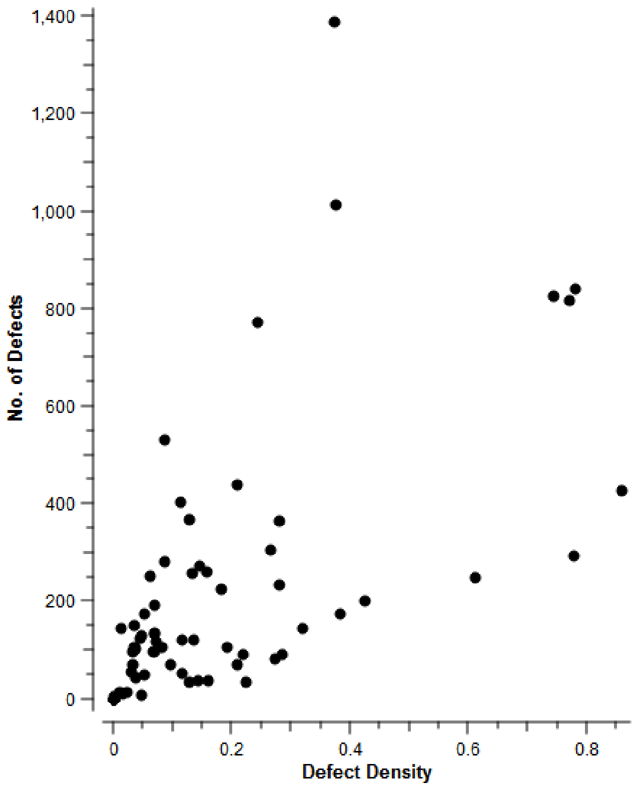

FIGURE 22. Effect of defect density on number of defects.

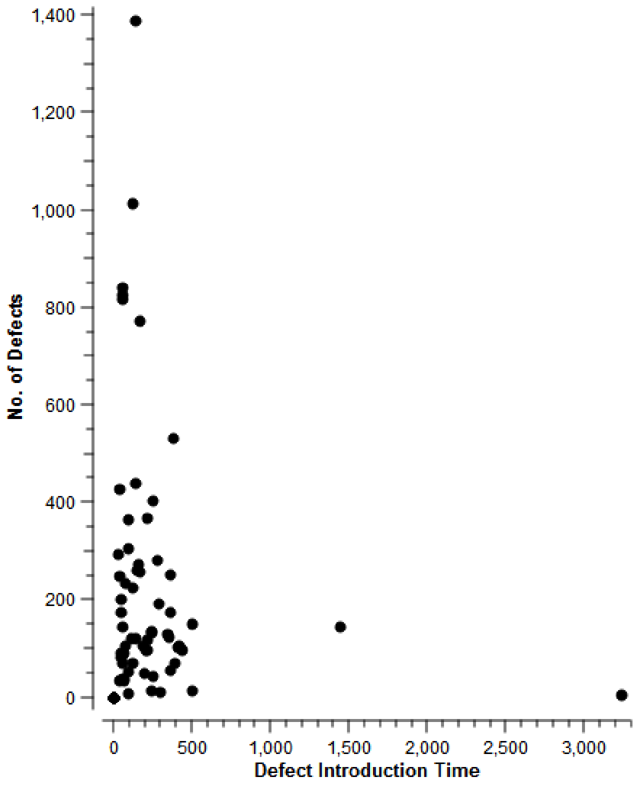

FIGURE 23. Effect of defect introduction time on number of defects.

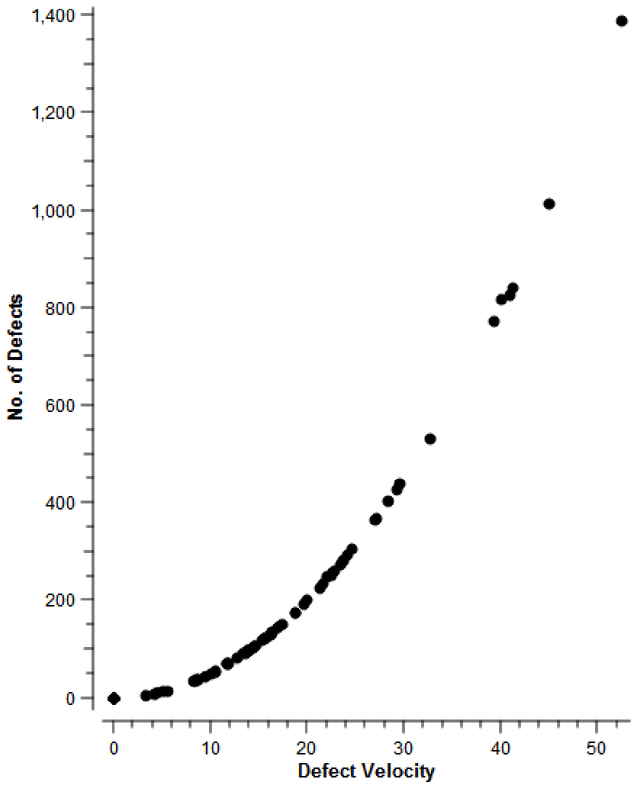

FIGURE 24. Effect of defect velocity on number of defects.

Figures 22, 23 and 24 compare the results of our experiments on the ELFF datasets with regard to the influence of the proposed metrics, namely, the defect density, defect introduction time and defect velocity, on the number of defects per class across these 69 open-source Java projects. The results show that the defect density, defect introduction time and defect velocity exhibit correlation coefficients of 61.41%, −11.413% and 93.56%, respectively, with the number of defects. These findings further support that the number of defects in a software project is a function of the defect velocity, as demonstrated in this paper using both the NASA datasets and the more recent ELFF datasets.

VII. THREATS TO VALIDITY

In this section, we present the potential threats to the validity of this study. We discuss both the internal and external threats that may have affected the study's results.

A. INTERNAL THREATS TO VALIDITY

Although previous studies have achieved noteworthy performance in terms of metric selection for defect prediction, we carefully selected novel metrics that we believed might influence the number of defects in a software product. We also constructed the equations for deriving the corresponding predictor variables. These equations and the derived predictor variables may have affected our results, and to clarify these potential effects, we evaluated the individual performance of each metric and its influence on the number of defects. We considered only a few predictor variables for predicting the number of defects in a new program; further studies may consider other meaningful variables that might influence the number of defects in an upcoming software project. For this purpose, the careful selection of additional metrics that may impact the number of software defects will be needed.

B. EXTERNAL THREATS TO VALIDITY

We conducted our experiments using 10 different datasets containing 22,838 instances. These datasets contain imbalanced classes, which may pose a threat to the generalizability of the results of this study. Note that these datasets are widely used NASA projects collected from the PROMISE repository and have been applied in several previous defect prediction studies. For instance, Laradji *et al.* [41], Czibula *et al.* [83], and Ghotra *et al.* [84] applied these NASA datasets in their early studies. The accuracy of our results depends on the preprocessing and data cleaning techniques applied in this study. Although the results varied between projects, the dataset preprocessing was an important phase of this study. Nevertheless, additional replication studies would be advantageous. In this paper, class imbalance has been identified as a challenge in software defect prediction, but we did not specifically address this issue in the current study because our focus was the prediction of the number of defects.

VIII. CONCLUSION AND FUTURE WORK

Several concerns that arise in software defect prediction have yet to be resolved. Therefore, we have presented a **MACLI** approach for predicting the number of defects in an upcoming software product using predictor variables derived from the defect acceleration, and we have determined the correlation of each predictor variable with the number of defects. The number of defects shows a strong positive correlation with the average defect velocity, a weak positive correlation with the average defect density, and a negative correlation with the average defect introduction time, with correlation coefficients of 0.98, 0.22 and −0.30, respectively. The proposed method can provide practical outputs to managers and software development teams. In our experiments, we used 10 PROMISE datasets, containing 22,838 instances in total. Our experimental results show that a prediction model based on the average defect velocity achieves an adjusted R-square of 98.6% and a p-value of < 0.001, indicating that the average defect velocity is strongly positively correlated with the number of defects. The advance prediction of the potential number of defects in a software product, as presented in Tables 3 to 1, can serve as an actionable output for a software development team.

We implemented a systematic modeling technique for predicting the number of software defects. We considered various combinations of our predictor variables, namely, the average defect velocity, average defect introduction time, and average defect density, for use in predicting the number of defects in an upcoming product release. In addition, we evaluated the individual effect of each of these variables on the number of software defects. Our results suggest that among these predictor variables, only the average defect velocity shows a strong positive correlation with the number of possible defects in an upcoming product release. Therefore, to reduce defects, software managers can focus on the rate at which a project transitions from one phase to another over time.

The results of our work must be confirmed to verify the suitability of our approach for defect prediction. Future studies can use the most recent datasets from any software company to validate this method for predicting the number of defects in an upcoming product release while also considering additional predictor variables. The proposed **MACLI** approach can enable managers and development teams to restructure their testing and debugging efforts based on a highly cost-effective and actionable prediction model.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their comments.

REFERENCES

[1] K. Herzig, "Using pre-release test failures to build early post-release defect prediction models," in *Proc. IEEE 25th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Nov. 2014, pp. 300–311.

[2] A. Nugroho, M. R. V. Chaudron, and E. Arisholm, "Assessing UML design metrics for predicting fault-prone classes in a Java system," in *Proc. 7th IEEE Work. Conf. Mining Softw. Repositories (MSR)*, May 2010, pp. 21–30.

[3] A. Halim, "Predict fault-prone classes using the complexity of UML class diagram," in *Proc. IEEE Int. Conf. Comput., Control, Inform. Appl. (IC3INA)*, Nov. 2013, pp. 289–294.

[4] S. Parthipan, S. Senthil Velan, and C. Babu, "Design level metrics to measure the complexity across versions of AO software," in *Proc. IEEE Int. Conf. Adv. Commun. Control Comput. Technol. (ICACCCT)*, May 2014, pp. 1708–1714.

[5] B. Caglayan, A. T. Misirli, A. B. Bener, and A. Miranskyy, "Predicting defective modules in different test phases," *Softw. Quality J.*, vol. 23, no. 2, pp. 205–227, 2015.

[6] Y. Ma and B. Cukic, "Adequate and precise evaluation of quality models in software engineering studies," in *Proc. IEEE Int. Workshop Predictor Models Softw. Eng., ICSE Workshops (PROMISE)*, May 2007, p. 1.
[7] Y. Jiang, B. Cukic, and T. Menzies, "Fault prediction using early lifecycle data," in *Proc. 18th IEEE Int. Symp. Softw. Rel. (ISSRE)*, Nov. 2007, pp. 237–246.
[8] A. Monden *et al.*, "Assessing the cost effectiveness of fault prediction in acceptance testing," *IEEE Trans. Softw. Eng.*, vol. 39, no. 10, pp. 1345–1357, Oct. 2013.
[9] Y. Zhang, D. Lo, X. Xia, and J. Sun, "An empirical study of classifier combination for cross-project defect prediction," in *Proc. IEEE 39th Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 2. Jul. 2015, pp. 264–269.
[10] K. Han, J.-H. Cao, S.-H. Chen, and W.-W. Liu, "A software reliability prediction method based on software development process," in *Proc. IEEE Int. Conf. Quality, Rel., Risk, Maintenance, Safety Eng. (QR2MSE)*, Jul. 2013, pp. 280–283.
[11] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan, "Predicting bugs using antipatterns," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2013 pp. 270–279.
[12] T. Mori, "Superposed naive Bayes for accurate and interpretable prediction," in *Proc. IEEE 14th Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2015, pp. 1228–1233.
[13] X.-Y. Jing, Z.-W. Zhang, S. Ying, F. Wang, and Y.-P. Zhu, "Software defect prediction based on collaborative representation classification," in *Proc. Companion 36th Int. Conf. Softw. Eng.*, 2014, pp. 632–633.
[14] Z. Mahmood, D. Bowes, P. C. Lane, and T. Hall, "What is the impact of imbalance on software defect prediction performance?" in *Proc. 11th Int. Conf. Predictive Models Data Anal. Softw. Eng.*, 2015, p. 4.
[15] R. Islam and K. Sakib, "A package based clustering for enhancing software defect prediction accuracy," in *Proc. IEEE 17th Int. Conf. Comput. Inf. Technol. (ICCIT)*, Dec. 2014, pp. 81–86.
[16] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto, "The impact of mislabelling on the performance and interpretation of defect prediction models," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng. (ICSE)*, vol. 1. May 2015, pp. 812–823.
[17] G. Mauša, T. G. Grbac, and B. D. Bašić, "Software defect prediction with Bug-Code analyzer—A data collection tool demo," in *Proc. IEEE 22nd Int. Conf. Softw., Telecommun. Comput. Netw. (SoftCOM)*, Sep. 2014, pp. 425–426.
[18] C. A. Siebra and M. A. B. Mello, "The importance of replications in software engineering: A case study in defect prediction," in *Proc. Conf. Res. Adapt. Convergent Syst.*, 2015, pp. 376–381.
[19] M. H. Rahman, S. Sharmin, S. M. Sarwar, and M. Shoyaib, "Software defect prediction using feature space transformation," in *Proc. Int. Conf. Internet Things Cloud Comput.*, 2016, p. 72.
[20] R. Malhotra and R. Raje, "An empirical comparison of machine learning techniques for software defect prediction," in *Proc. 8th Int. Conf. Bioinspired Inf. Commun. Technol.*, 2014, pp. 320–327.
[21] D. G. Cavezza, R. Pietrantuono, and S. Russo, "Performance of defect prediction in rapidly evolving software," in *Proc. IEEE/ACM 3rd Int. Workshop Release Eng. (RELENG)*, May 2015, pp. 8–11.
[22] H. Lu, B. Cukic, and M. Culp, "Software defect prediction using semi-supervised learning with dimension reduction," in *Proc. 27th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, Sep. 2012, pp. 314–317.
[23] 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.*, 2015, pp. 1644–1647.
[24] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi, "An empirical study of just-in-time defect prediction using cross-project models," in *Proc. 11th Work. Conf. Mining Softw. Repositories*, 2014, pp. 172–181.
[25] J. Nam and S. Kim, "Heterogeneous defect prediction," in *Proc. 10th Joint Meet. Found. Softw. Eng.*, 2015, pp. 508–519.
[26] X. Jing, F. Wu, X. Dong, F. Qi, and B. Xu, "Heterogeneous cross-company defect prediction by unified metric representation and CCA-based transfer learning," in *Proc. 10th Joint Meet. Found. Softw. Eng.*, 2015, pp. 496–507.
[27] A. Panichella, R. Oliveto, and A. De Lucia, "Cross-project defect prediction models: L'Union fait la force," in *Proc. Softw. Evol. Week-IEEE Conf. Softw. Maintenance, Reeng. Reverse Eng. (CSMR-WCRE)*, Feb. 2014, pp. 164–173.
[28] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, Jul. 2008.
[29] S. Yousefi and N. Modiri, "Deployment of integrated design for the reduction of software complexity," in *Proc. IEEE 7th Int. Conf. Netw. Comput. Adv. Inf. Manage. (NCM)*, Jun. 2011, pp. 172–174.
[30] M. Shepperd, D. Bowes, and T. Hall, "Researcher bias: The use of machine learning in software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 40, no. 6, pp. 603–616, Jun. 2014.
[31] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model," in *Proc. 11th Work. Conf. Mining Softw. Repositories*, 2014, pp. 182–191.
[32] B. Caglayan, A. Tosun, A. Miranskyy, A. Bener, and N. Ruffolo, "Usage of multiple prediction models based on defect categories," in *Proc. 6th Int. Conf. Predictive Models Softw. Eng.*, 2010, p. 8.
[33] N. Maneerat and P. Muenchaisri, "Bad-smell prediction from software design model using machine learning techniques," in *Proc. IEEE 8th Int. Joint Conf. Comput. Sci. Softw. Eng. (JCSSE)*, May 2011, pp. 331–336.
[34] J. Wuest. (2017). *The Software Design Metrics Tool for the UML.* [Online]. Available: http://www.sdmetrics.com/LoR.html
[35] C. F. J. Lange and M. R. V. Chaudron, "Managing model quality in UML-based software development," in *Proc. 13th IEEE Int. Workshop Softw. Technol. Eng. Pract.*, Sep. 2005, pp. 7–16.
[36] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th ed. Basingstoke, U.K.: Palgrave Macmillan, 2005.
[37] C. Becker *et al.*, "Sustainability design and software: The Karlskrona manifesto," in *Proc. 37th Int. Conf. Softw. Eng.*, vol. 2. May 2015, pp. 467–476.
[38] A. M. Fernández-Sáez, M. Genero, and M. R. Chaudron, "Empirical investigation on the benefits of using UML in software maintenance," in *Proc. 12th Int. Conf. Product Focused Softw. Develop. Process Improvement*, 2011, pp. 44–47.
[39] S. Khalid, S. Zehra, and F. Arif, "Analysis of object oriented complexity and testability using object oriented design metrics," in *Proc. Nat. Softw. Eng. Conf.*, 2010, p. 4.
[40] Z.-W. Zhang, X.-Y. Jing, and T.-J. Wang, "Label propagation based semi-supervised learning for software defect prediction," *Automated Softw. Eng.*, vol. 24, no. 1, pp. 47–69, 2017.
[41] I. H. Laradji, M. Alshayeb, and L. Ghouti, "Software defect prediction using ensemble learning on selected features," *Inf. Softw. Technol.*, vol. 58, pp. 388–402, Feb. 2015.
[42] N. Japkowicz and S. Stephen, "The class imbalance problem: A systematic study," *Intell. Data Anal.*, vol. 6, no. 5, pp. 429–449, Oct. 2002.
[43] N. V. Chawla, N. Japkowicz, and A. Kotcz, "Special issue on learning from imbalanced data sets," *ACM Sigkdd Explorations Newslett.*, vol. 6, no. 1, pp. 1–6, 2004.
[44] J. Van Hulse, T. M. Khoshgoftaar, and A. Napolitano, "Experimental perspectives on learning from imbalanced data," in *Proc. 24th Int. Conf. Mach. Learn.*, 2007, pp. 935–942.
[45] P. Soda, "A multi-objective optimisation approach for class imbalance learning," *Pattern Recognit.*, vol. 44, no. 8, pp. 1801–1810, 2011.
[46] G. Batista, D. Silva, and R. Prati, "An experimental design to evaluate class imbalance treatment methods," in *Proc. 11th Int. Conf. Mach. Learn. Appl. (ICMLA)*, vol. 2. Dec. 2012, pp. 95–101.
[47] D. Rodriguez, I. Herraiz, R. Harrison, J. Dolado, and J. C. Riquelme, "Preliminary comparison of techniques for dealing with imbalance in software defect prediction," in *Proc. 18th Int. Conf. Eval. Assessment Softw. Eng.*, 2014, p. 43.
[48] Z. Sun, Q. Song, X. Zhu, H. Sun, B. Xu, and Y. Zhou, "A novel ensemble method for classifying imbalanced data," *Pattern Recognit.*, vol. 48, no. 5, pp. 1623–1637, 2015.
[49] W. Mao, J. Wang, L. He, and Y. Tian, "Online sequential prediction of imbalance data with two-stage hybrid strategy by extreme learning machine," *Neurocomputing*, vol. 261, pp. 94–105, Oct. 2017.
[50] Q. Yu, S. Jiang, and Y. Zhang, "The performance stability of defect prediction models with class imbalance: An empirical study," *IEICE Trans. Inf. Syst.*, vol. E100.D, no. 2, pp. 265–272, 2017.
[51] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson, "The misuse of the NASA metrics data program data sets for automated software defect prediction," in *Proc. 15th Annu. Conf. Eval. Assessment Softw. Eng. (EASE)*, 2011, pp. 96–103.
[52] A. Aydin and A. Tarhan, "Investigating defect prediction models for iterative software development when phase data is not recorded lessons learned," in *Proc. IEEE Int. Conf. Eval. Novel Approaches Softw. Eng. (ENASE)*, Apr. 2014, pp. 1–11.

[53] J. Sayyad Shirabad and T. J. Menzies, "The PROMISE repository of software engineering databases," School Inf. Technol. Eng., Univ. Ottawa, Ottawa, ON, Canada, 2005. [Online]. Available: http://promise.site.uottawa.ca/SERepository
[54] J. Petrić, "Using different characteristics of machine learners to identify different defect families," in *Proc. 20th Int. Conf. Eval. Assessment Softw. Eng.*, 2016, p. 5.
[55] S. Ertekin, J. Huang, and C. L. Giles, "Active learning for class imbalance problem," in *Proc. 30th Annu. Int. ACM SIGIR Conf. Res. Develop. Inf. Retr.*, 2007, pp. 823–824.
[56] N. Ofek, L. Rokach, R. Stern, and A. Shabtai, "Fast-CBUS: A fast clustering-based undersampling method for addressing the class imbalance problem," *Neurocomputing*, vol. 243, pp. 88–102, Jun. 2017.
[57] W. Lee, C.-H. Jun, and J.-S. Lee, "Instance categorization by support vector machines to adjust weights in AdaBoost for imbalanced data classification," *Inf. Sci.*, vol. 381, pp. 92–103, Mar. 2017.
[58] O. Loyola-González, M. A. Medina-Pérez, J. F. Martínez-Trinidad, J. A. Carrasco-Ochoa, R. Monroy, and M. García-Borroto, "PBC4cip: A new contrast pattern-based classifier for class imbalance problems," *Knowl.-Based Syst.*, vol. 115, pp. 100–109, Jan. 2017.
[59] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data quality: Some comments on the NASA software defect datasets," *IEEE Trans. Softw. Eng.*, vol. 39, no. 9, pp. 1208–1215, Sep. 2013.
[60] J. Petrić, D. Bowes, T. Hall, B. Christianson, and N. Baddoo, "The jinx on the NASA software defect data sets," in *Proc. 20th Int. Conf. Eval. Assessment Softw. Eng.*, 2016, p. 13.
[61] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 1–18, Jan. 2017.
[62] Q. Li, C. Rajagopalan, and G. D. Clifford, "Ventricular fibrillation and tachycardia classification using a machine learning approach," *IEEE Trans. Biomed. Eng.*, vol. 61, no. 3, pp. 1607–1613, Jun. 2013.
[63] U. M. Braga-Neto and E. R. Dougherty, "Is cross-validation valid for small-sample microarray classification?" *Bioinformatics*, vol. 20, no. 3, pp. 374–380, 2004.
[64] A. Isaksson, M. Wallman, H. Göransson, and M. G. Gustafsson, "Cross-validation and bootstrapping are unreliable in small sample classification," *Pattern Recognit. Lett.*, vol. 29, no. 14, pp. 1960–1965, 2008.
[65] G. Seni and J. F. Elder, "Ensemble methods in data mining: Improving accuracy through combining predictions," *Synth. Lectures Data Mining Knowl. Discovery*, vol. 2, no. 1, pp. 1–126, 2010.
[66] K. Dejaeger, T. Verbraken, and B. Baesens, "Toward comprehensible software fault prediction models using Bayesian network classifiers," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 237–257, Feb. 2013.
[67] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson, "Software defect prediction using static code metrics underestimates defect-proneness," in *Proc. IEEE Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2010, pp. 1–7.
[68] M. Alshayeb and W. Li, "An empirical validation of object-oriented metrics in two different iterative software processes," *IEEE Trans. Softw. Eng.*, vol. 29, no. 11, pp. 1043–1049, Nov. 2003.
[69] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 897–910, Oct. 2005.
[70] H. M. Olague, L. H. Etzkorn, S. Gholston, and S. Quattlebaum, "Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes," *IEEE Trans. Softw. Eng.*, vol. 33, no. 6, pp. 402–419, Jun. 2007.
[71] 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, 2008.
[72] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proc. 3rd Int. Workshop Predictor Models Softw. Eng.*, May 2007, p. 9.
[73] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Inf. Softw. Technol.*, vol. 55, no. 8, pp. 1397–1418, 2013.
[74] B. Carterette, "Statistical significance testing in information retrieval: Theory and practice," in *Proc. Int. Conf. Theory Inf. Retr.*, 2015, pp. 7–9.
[75] J. P. A. Ioannidis, "Why most published research findings are false," *PLoS Med.*, vol. 2, no. 8, p. e124, 2005.
[76] T. Foss, E. Stensrud, B. Kitchenham, and I. Myrtveit, "A simulation study of the model evaluation criterion MMRE," *IEEE Trans. Softw. Eng.*, vol. 29, no. 11, pp. 985–995, Nov. 2003.
[77] Y. Miyazaki, M. Terakado, K. Ozaki, and H. Nozaki, "Robust regression for developing software estimation models," *J. Syst. Softw.*, vol. 27, no. 1, pp. 3–16, 1994.
[78] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 25, no. 5, pp. 675–689, Sep. 1999.
[79] B. F. J. Manly and J. A. N. Alberto, *Multivariate Statistical Methods: A Primer*. Boca Raton, FL, USA: CRC Press, 2016.
[80] J. Rosenberg, "Some misconceptions about lines of code," in *Proc. IEEE 4th Int. Softw. Metrics Symp.*, Nov. 1997, pp. 137–142.
[81] B. T. Compton and C. Withrow, "Prediction and control of ADA software defects," *J. Syst. Softw.*, vol. 12, no. 3, pp. 199–207, 1990.
[82] L. Hatton, *The Automation of Software Process and Product Quality* (WIT Transactions on Information and Communication Technologies), vol. 4. Southampton, U.K.: WIT Press, 1970.
[83] G. Czibula, Z. Marian, and I. G. Czibula, "Software defect prediction using relational association rule mining," *Inf. Sci.*, vol. 264, pp. 260–278, Apr. 2014.
[84] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Proc. IEEE 37th Int. Conf. Softw. Eng.*, vol. 1. May 2015, pp. 789–800.

EBUBEOGU AMARACHUKWU FELIX (GS'17) is currently pursuing the Ph.D. degree with the Department of Software Engineering, University of Malaya, Malaysia. His research interests include machine learning, software quality, software maintenance, and big data. He is a Student Member of ACM.

SAI PECK LEE (M'10) received the Ph.D. degree in computer science from Université Paris 1 Panthéon-Sorbonne. She is currently a Professor with the Department of Software Engineering, University of Malaya. She has authored or co-authored an academic book, book chapters, and over 100 papers in various refereed journals and conference proceedings. Her current research interests include requirements engineering, software traceability and clustering, software quality, software reuse, object-oriented techniques, and CASE tools development. She has also actively participated in conference program committees and is currently in several experts review panels, both locally and internationally.