ISSN E: 3006-1466
ISSN P: 3006-1458
CONTEMPORARY JOURNAL OF SOCIAL SCIENCE REVIEW
CJSSR

SOFTWARE PROCESS IMPROVEMENT USING DEFECT PREDICTION ANALYTICS

Iman Ali
Department of Computer Science and Information Technology,Superior University, Lahore.Punjab, Pakistan
Corresponding Author: Email: imanh0733@gmail.com
Zunaira Ijaz
Department of Software Engineering,Superior University, Lahore,Punjab,Pakistan
Corresponding Author: Email: zunairaijaz42@gmail.com
Saleem Zubair
Department of Computer Science and Information Technology,Superior University, Lahore,Punjab, Pakistan.
saleem.zubair@superior.edu.pk
Ayesha Saddiqa
Department of Computer Science and Information Technology,Superior University, Lahore Punjab, Pakistan.
ayeshasaddiqa@superior.edu.pk

ABSTRACT
Defects in software cause additional time and financial costs for businesses, and can result in significant delay in completion of software projects, customer not satisfied with software products, and a multiple of other problems. Most industrial software systems find the majority of their defects later in the software development life cycle than they should be. At this point in time the cost of fixing those defects is significantly increased compared to if they had been found earlier in the development process, and the quality assurance processes used are less effective. Previous studies have indicated that the quality assurance processes that focus on testing are largely reactive and do not provide a valid means of ensuring quality assurance in complex software systems.
In order to mitigate these problems, this research uses defect prediction analytics to improve software development processes. Specifically, the proposed research utilizes historical software metric data along with machine learning algorithms to identify defect-prone modules in the software development process at a relatively early stage in the software development lifecycle. Identification of modules with a high risk of containing defects allows development teams and project managers to prioritize the allocation of testing resources, to optimize resource utilization, and to minimize the quantity of quality assurance effort that is expended unnecessarily. The outputs of the defect prediction algorithm are used as input to support decision-making for software process management.
The proposed approach is evaluated experimentally utilizing datasets of software defects collected from NASA, which are representative of industrial software projects. The experimental results show that reasonable prediction performance can be obtained, especially when evaluating large and complex datasets. The results of the study provide evidence that defect prediction analytics is a viable method for improving software development processes through proactive quality assurance, use of metrics to make decisions, and improved predictability of project outcomes.

Keywords: *Defect Prediction, Machine Learning, Software Quality, Software Process Improvement, Static Code Metrics.*

INTRODUCTION

While the number of potential defects in the code is virtually infinite, the number of actual defects present in the code is generally much smaller than the number of possible defects. This fact can help developers and testers to reduce the time and effort they spend searching for errors, because they do not need to search through all of the code. They can instead concentrate on the parts of the code most likely to have defects. Using a combination of predictive modeling techniques and historical data related to the development of previous versions of the same application, it is possible to develop a method for estimating the probability that a particular piece of code will have defects. While no technique is perfect and there is always some chance that a piece of code will have defects that were not anticipated, developing a predictive method can greatly increase the chances that problems will be detected before the product is delivered to the customer. Historical data about the development of previous versions of the application are essential to developing a good predictive model. With that data, developers can train the model to recognize patterns that are typical of defect-free pieces of code versus those that are typical of code that contains defects. Once the model is trained and validated, it can then be used to evaluate future applications as they are being developed, allowing the developer to identify and correct potential problems early in the development cycle.

Using predictive modeling for evaluating the quality of code before the product is delivered provides an opportunity for developers to catch potential issues before they become major problems; as a result, developers are able to correct these problems efficiently and cost effectively. Evaluating the quality of code at the point of delivery also provides assurance to the customer that their final product will run correctly when it is delivered to them; thus, this process minimizes the need for post-delivery testing and support services. The application of predictive modeling in the evaluation of software code has particular value to organizations attempting to deploy Agile Development methodologies within their organization. The use of predictive modeling to assess the quality of code during the start phases of the development lifecycle ensures that the organization can respond appropriately to changing market conditions and customer requirements which, ultimately, refine the success of the development project. One of the principal benefits of utilizing predictive modeling for assessing the quality of software code is its significant impact upon increasing the efficiency of the testing process. Predictive modeling enables testers to determine which portions of the code are most likely to contain errors, thereby allowing the tester to direct their testing efforts towards those segments of the code. Ultimately, this increases productivity, reduces costs, and results in the developer being able to deliver their product to the marketplace in a timely manner.

Additionally, predictive modeling can be used to evaluate the effectiveness of the testing process. If the testing process is effective, then the predictive model should not show a lot of defects. On the other hand, if the predictive model shows a lot of defects, then the testing process was not very effective. This information can be used to adjust the testing process and improve the overall quality of the application. Overall, using predictive modeling in the evaluation of software code can greatly improve the efficiency and effectiveness of the testing process. It can also help to ensure that the application is delivered timely to the customer and meets their needs, which ultimately improves the overall success of the development effort. Predictive modeling can be defined as a set of statistical techniques used to forecast the future behavior of an object based on past history and/or external factors. When applied to the evaluation of software code, predictive modeling involves collecting data on the characteristics of each piece of code, creating a model based on that data, and then using the model to predict the probability that each piece of code will contain defects.

There are several different types of predictive modeling techniques that can be used to evaluate software code. Some of the most common include regression analysis, decision trees, neural networks, and Bayesian belief networks. Each of these techniques has strengths and weaknesses, and the choice of which to use depends on the type of data available and the goals of the predictive model. Regression analysis is a statistical technique used to establish the relationship between two or more independent variables and one dependent variable. In the context of predictive modeling, regression analysis can be used to create a model that predict the probability of defects based on the characteristics of the code. Decision trees are another type of predictive modeling technique that can be used to evaluate software code. A decision tree is a graphical representation of a series of decisions and their consequences. In the context of predictive modeling, a decision tree can be created to represent the relationships between the characteristics of the code and the presence or absence of defects. Neural networks are a type of predictive modeling technique that uses artificial intelligence to evaluate the characteristics of the code and forecast the probability of defects. Neural networks work by processing inputs in layers, with each layer performing a different function in the evaluation process. Bayesian belief networks are a type of predictive modeling technique that uses probabilistic reasoning to evaluate the characteristics of the code and forecast the probability of defects. Bayesian belief networks are similar to decision trees in that they represent the relationships between the characteristics of the code and the presence or absence of defects. Regardless of the predictive modeling technique used, the process of building a model involves several steps. First, the data must be collected and cleaned. Then, the model must be built based on that data. Finally, the model must be evaluated to determine whether it is accurate enough to be useful.

Once a predictive model has been built and evaluated, it can be used to predict the probability of defects in new code. This can be done by inputting the characteristics of the new code into the model and seeing what the output is. Overall, predictive modeling is a powerful tool for improving the efficiency and effectiveness of the testing process. By forecasting the probability of defects in software code, predictive modeling can help developers to focus their testing efforts on the areas of the code where defects are most likely to occur.

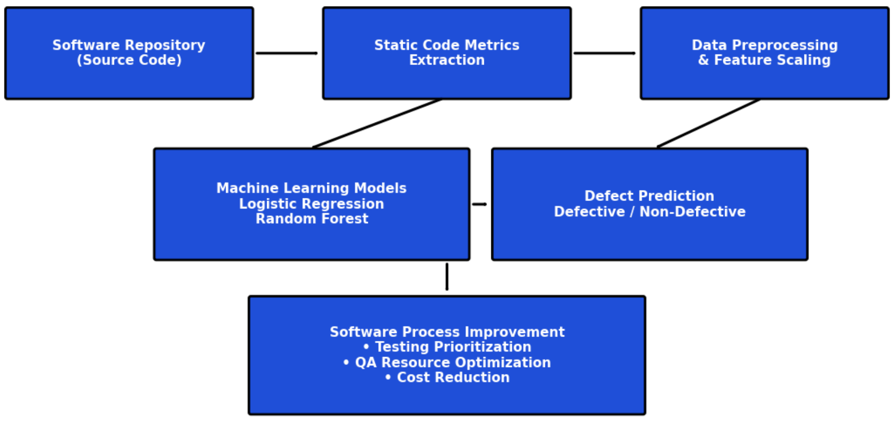
Software Repository (Source Code)
Static Code Metrics Extraction
Data Preprocessing & Feature Scaling
Machine Learning Models Logistic Regression Random Forest
Defect Prediction Defective / Non-Defective
Software Process Improvement
• Testing Prioritization
• QA Resource Optimization
• Cost Reduction

FIGURE 1

LITERATURE REVIEW

Early work in predicting software defects was centered on analyzing the defects via statistical software metric measures to identify faulty modules. Fenton & Neil [1] provided a comprehensive review of software metrics and illustrated that size- and complexity-based metrics were good predictors of defects occurring in software modules. Fenton and Neil's work provided the foundation of theory for quantitative software quality measurement and identified the role of measurement in software engineering. While the method of metrics-based approach helped to improve the quality of software products, subsequent research further investigated the relationship between static code metrics and defect prone modules.

Research indicated that larger modules contained a greater amount of defects, due to the increased complexity and interactions among the component parts. Research also indicated that complexity metrics, specifically cyclomatic complexity, represented the logical complexity of code, which impacts the maintainability and testability of code. Modules with greater complexity often experienced a greater amount of defects per unit of code, thus, complexity was a key indicator in many early defect detection models. The findings from these studies supported the argument that software metrics could be useful tools to represent software quality attributes. While both of these areas of research contributed greatly to the field of defect prediction, there were some limitations to purely metrics-based approaches. Many of the early studies focused on descriptive statistical analysis (i.e., correlation, regression), but did not provide any insight regarding the nature of defect-prone features. Additionally, the ability of software systems to grow in size and complexity, required a transition to more proactive defect prevention methodologies, based solely on metrics [1]. With the increasing availability of historical defect data, researchers began applying statistical and machine learning techniques to predict defective software modules. One of the first studies to demonstrate that machine learning models, such as logistic regression, can effectively mine static code attribute data, were Menzies, Greenwald, and Frank [2]. The authors demonstrated that properly preprocessed simple classifiers can perform competitively when predicting defective modules. Furthermore, the authors demonstrated that the simplicity and interpretability of the models were benefits, and thus, logistic regression is the baseline model used in defect prediction studies. The authors did note that the performance of the models is heavily dependent on the characteristics of the data and preprocessing options [2]. One of the largest contributions to the area of defect prediction research, was the empirical study conducted by Lessmann et al. [3]. The authors ran a large-scale empirical study, comparing multiple classification methods, including logistic regression, decision trees, neural networks, support vector machines, and ensemble methods. The study produced two main conclusions; no single classification method consistently produces better results across all datasets, and the results of the study demonstrated the need for a comparative assessment of the models and the use of multiple performance metrics. Ensemble learning methods are widely accepted due to their power and ability to address complex relationships. Breiman [4] proposed the random forest method, which combines multiple decision trees to increase predictive accuracy and reduce over-fitting. Several defect prediction studies have shown that random forests produce high-quality results in high-dimensional software metrics data. However, while ensemble models can be highly effective, they are typically non-interpretable and thus, can be difficult to apply in software process improvement environments. Practitioners may find it difficult to understand that a module is labeled "defective," and thus, diminish their confidence in the prediction results [4]. One major challenge in defect prediction is the imbalanced classification of defective versus non-defective modules, since defective modules typically occur in a much smaller quantity than modules without defects. Hall et al. [5] conducted a systematic review of the literature, and found that the performance of the models used to handle class imbalance has a significant impact on the recall and defect detection performance. The authors found that to adequately evaluate defect prediction models, accuracy alone is insufficient. Therefore, several researchers have proposed a variety of methods, including resampling, cost-sensitive learning and alternative evaluation metrics. Nevertheless, class imbalance remains a significant challenge in the practical application of defect prediction models [5]. Benchmark datasets are important in defect prediction research, since they enable the efficient comparison of studies. The NASA Metrics Data Program (MDP) and the JM1 dataset have been extensively used in the evaluation of defect prediction models. Both of these datasets include realistic data on the static code metrics and defect labels of actual software projects [6]. While the NASA datasets have enabled substantial research efforts, various researchers have noted that reliance upon a single dataset limits the generalizability of results. It is generally recommended that cross-project validation and testing across multiple datasets be performed to enhance robustness [2][3]. Recent research has increasingly examined the extent to which defect prediction has been used to improve software processes, as opposed to focusing on improving the accuracy of defect prediction.

Tosun, Turhan & Bener [9] emphasized that the purpose of defect predictive models is to facilitate decision-making and reduce the amount of time spent on testing and promote quality assurance on risk. Shepperd & MacDonell [8] also argued that prediction systems need to be evaluated within the context of the organization utilizing them. Recent research has demonstrated that simpler and easier-to-understand models can perform as well or even better than more complex models. For example, Peters et al. [14] have demonstrated that simple and easy-to-use models can provide consistent and reliable predictions, which makes them more suitable for real-world SPI applications.

The literature studied shows that the development of software defect prediction has evolved toward the use of predictive analytics, driven by machine learning, and away from simply using metrics to make predictions. Much progress has been achieved in terms of developing and refining defect prediction models; however, several constraints exist in relation to the use of defect prediction models, including class imbalance, limited generalizability, and difficulty in relating the results to software process improvement activities. The majority of the existing literature focuses on improving the performance of defect prediction models, and relatively few studies examine the use of defect prediction model results to determine testing and quality assurance. Based on this analysis, a research gap exists in the development of defect prediction models that emphasize interpretability, usability, and applicability to support immediate software process improvement. This research gap is the focus of the current study, in which a machine learning-based defect prediction analytics is developed with an emphasis on supporting testing prioritization and SPI results.

TABLE 1

Summary of research gaps identified from different research work

Aspects	**Limitations**	**How the study is to deal with this**
Pre-emptive faults detection	Reactive QA mostly	Predictive analytics
Interpretability of models	Class ensemble Model	Simple LR Model.
SPI Integration	Limited discussion	Explicit SPI Focus
Lack of class imbalance	Poor recall	Strategy based on precision
Real-world implementation	Very complicated	Lightweight ML Pipeline

DATASET DESCRIPTION

A number of cognitive dissonance issues related to prolonged consumption of massive amounts of information through the media and pop culture exist as a result of problems associated with the consumption of large amounts of information. NASA JM1 is one of the most commonly cited benchmarks in the field of software defect prediction; this data set contains 10,885 modules of software created using 23 different attributes. The first twenty-two attributes provide measures of static code in terms of size and complexity along with various structural characteristics of the code. The last attribute provides a measure of defects (the number of defects). Furthermore, due to the fact that there are 2,106 defective modules and 8,779 non-defective modules within the data set, the data is highly skewed which simulates a number of the challenges that occur in developing realistic defect prediction models based upon actual software development in the real world.

TABLE 2

Attributes	Description
Total Module	**10,885**
Defective Modules	**2106**
Non-Defective Modules	**8779**
Static Code Metrics	**22**
Target variable defect	**Yes, No**

METHODOLOGY

The study employed the use of historical software metrics to create a reliable and consistent dataset for the purpose of defect prediction analysis. The first dataset used in this study consisted of raw metric values; therefore, the raw values had to be cleaned and transformed into a format suitable for model training. The missing values in the dataset were identified and treated accordingly so as to maintain the integrity of the data. After the cleaning and transformation processes were completed, the preprocessed dataset provided a solid base for the predictive modeling that followed.

After the data was prepared, the metrics were scaled uniformly in order to establish a common reference point in relation to each feature's scale. Following this, two machine learning models were trained — Logistical Regression and Random Forest — to classify the modules of software as either defective or non-defective. The choice of models was made primarily based on their established history of strong performance and their prevalence in previous research related to defect prediction.

A-Framework Overview

In addition to developing a structural and systematic framework for defect prediction and software process improvement, the study established a structured framework for supporting the process of defect prediction and software process improvement. The proposed framework consists of four stages that begin with the collection of historical software metrics from existing repositories. Following the collection of metrics, the collected metrics undergo processing in order to assure the data quality and reliability.

Following the completion of processing, machine learning models are trained to predict modules of software that may contain defects. Once the models have been trained, the outputs of the models are analyzed to identify patterns or trends that can assist in the decision-making process during the software development process. The structured nature of the pipeline provides assurance that the defect prediction results will not only be statistically valid, but also will be practical in terms of providing actionable information for quality assurance and process management activities.

B-Data Preprocessing

Data preprocessing is one of the most important components of improving the reliability of defect prediction models. Datasets used in defect prediction research often include missing values, inconsistencies, and other irrelevant attributes that can negatively affect the performance of the model being developed. To address the missing values present in the dataset, various imputation techniques were applied to preserve the integrity of the data.

Additionally, the non-informative identifier columns were removed from the dataset in order to avoid potential bias during the learning process. Additionally, noise and redundant attributes were evaluated in order to assure that only the relevant and meaningful attributes were maintained in the dataset. Improving the quality of the data prior to the training of the model enhanced both the stability of the predictions and the reliability of the model.

C-Feature Scaling

Feature scaling was applied in this study to ensure that all input variables would contribute equally during the model training process. Static code metrics are often characterized by large differences in magnitude; thus, standardization was performed in order to transform the data into a uniform scale. Uniform scaling of the data enabled the models — particularly Logistic Regression — to converge more quickly and reduced the likelihood of unstable optimization behavior. Properly scaling the data also improved the numerical stability of the models and ensured that all features were treated fairly during the learning process.

D-Model Implementation

Two Machine Learning Algorithms were used to determine the quality of defect prediction in this study:

• Linear Logistic Regression Classifier: selected for its simplicity and as the first "baseline" because it has been shown to be successful in previous defect prediction studies.

• Random Forest Classifier: an ensemble method to identify the relationship between the software metrics and how they relate to each other.

To prevent the issue of "data leakage," we developed a machine learning pipeline. A pipeline is simply a process that combines data imputation, feature scaling, and classification together in a single workflow, and is recommended for use in defect prediction research to increase the validity and studies.

E- Evaluating Model Performance Using an Evaluation Strategy

The model's performance was evaluated by applying multiple evaluation metrics that provided a comprehensive view of the classification performance. The evaluation metrics that were chosen include:

- Average accuracy of the model.
- Precision of the model.
- Recall of the model.
- F1-score.
- ROC-AUC.

The combination of these evaluation metrics provide a diverse range of perspectives on how well the model performed in classifying defects with a large disparity in size between each class. Accuracy provides a general assessment of how accurate the model was; however, precision and recall provide a better insight into the model's ability to find defects. The F1-score provides an average of precision and recall, and it gives a perspective on how well the model can classify defects based upon various threshold levels. In addition, the ROC-AUC provides a view of the model's ability to differentiate between two different classes (good or bad) as the threshold varies.

F-Software Process Integration

The output of the defect prediction models were viewed as decision-support tools for software process management. Early identification of potentially defective modules allows project managers to schedule and allocate testing resources more effectively and to understand the potential risks associated with development.

By incorporating defect prediction analytics into the software development lifecycle, organizations can transition from reacting to defects after they occur to proactively managing quality throughout the development process. This integration enhances the accuracy of planning decisions, reduces rework, optimizes the allocation of resources, and ultimately improves the efficiency of the development process.

Experimental Results & Analysis

In this part of the report, an extensive assessment of the results from the experimental phase, which were produced as output from the application of defect prediction models to the NASA JM1 dataset, is given. A number of different metrics, along with multiple visual representations, are used to assess the performance of the proposed models, providing for a complete analysis.

A-Experimental Setup

All of the experimental results presented in this paper were generated using the NASA JM1 dataset, comprising 10,885 software components, where each component was characterized using 22 static code metrics. The JM1 dataset was separated into training and testing sets, based on a stratified (80-20) random sampling process that preserved the proportion of classes contained in the original dataset. Furthermore, the five-fold cross-validation approach was used to measure the stability of the classification models.

The two classification models assessed in this study were:

•Logistic Regression

•Random Forest

Accuracy, Precision, Recall, F1-Score, Confusion Matrix and Cross Validation Accuracy are the metrics applied for assessing the models' performance.

MODEL	ACCURACY
LOGISTIC REGRESSION	85.87%
RANDOM FOREST	85.35%

The Logistic Regression model achieved the highest accuracy (85.87%), which was very close to the accuracy of the Random Forest model. It is likely that the relation between static code metrics and defect occurrence in the JM1 dataset is primarily linear.

C-Analysis of Precision, Recall and F1-Score

In addition to the high imbalance of the JM1 dataset (i.e., a large number of non-defective components vs. a small number of defective ones), it would be interesting to analyze additional performance metrics in order to provide a better understanding of how the model performs with such types of datasets.

METRIC	VALUE
PRECISION	51.92%
RECALL	12.83%

F1-SCORE	20.57%

The results from this study show that:
The high precision values (i.e., > 0.5) demonstrate that when the model identifies a defective module, it is likely to be accurate in its assessment of the module's condition;
the low recall values (i.e., < 0.2) demonstrate the extreme class imbalance; there were many fewer defective modules than non-defective modules;

D-Analysis of the Confusion Matrix

In addition to both precision and recall being based on the two classes of the problem being addressed, the analysis of the confusion matrix also offers additional insights as to how effectively the model can direct the quality assurance process.

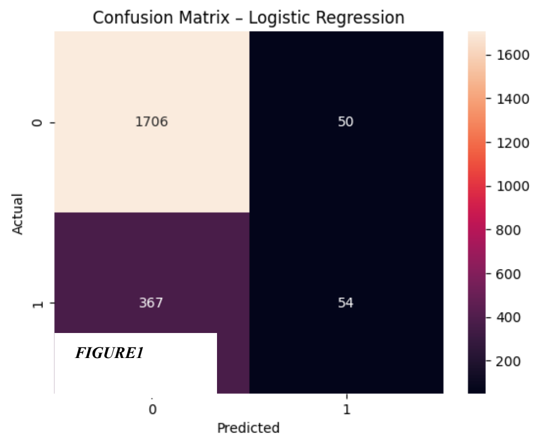

FIGURE1

Analysis:
• The model correctly classified 1706 non-defective components, greatly reducing unnecessary testing efforts;
• 54 defective components were correctly identified, allowing for their inspection at an early stage;
• A certain number of defective components were incorrectly classified because of the class imbalance, a well-known issue in the defect prediction area.
Therefore, the confusion matrix shows that the model is very useful for the prioritization of quality assurance actions.

E-Cross Validation Results

Finally, we verified the robustness of the model through the use of the 5 fold cross-validation technique.

METRIC	VALUE
MEAN CROSS VALIDATION ACCURACY	85.87

Analysis:
The cross-validation accuracy is very close to the test accuracy, indicating that the model is very effective and does not suffer from overfitting. Therefore, the reliability of the proposed defect prediction method is reinforced by these results.

F-Quality Assurance Processes and Defect Prediction Analytics

From a quality assurance processes perspective, the results show that defect prediction analytics can be used to guide testing priorities. Indeed, by accurately identifying the non-

defective components and detecting a subset of defective components, the resources of development teams can be allocated more effectively, the costs of testing can be reduced and overall software quality improved.
"According to previous defect prediction studies [2][3] the results of our study confirm that linear classifiers can achieve good performance on NASA software defect datasets."
"The effect of class imbalance on recall, as observed in our study, is consistent with the findings of other previous studies [5]."

LIMITATIONS ON GENERALIZABILITY

The conclusions drawn from this research are limited by some factors that should be taken into account when assessing the validity of the results. The first limitation is based on the fact that we used data from only one software project (the NASA JM1 dataset), even though this is widely used as a benchmark for defect prediction research. Therefore, it can be argued that there are limits to the extent to which the results can be generalized to other software projects developed with different types of technology, organizations, etc.
The second limitation relates to the sizeable class imbalance within the dataset, where there are many fewer instances of defective modules compared to non-defective modules. Class imbalance has a negative impact on the recall performance of prediction models, potentially leading to omission of defective modules. It has been noted in previous studies that class imbalance represents one of the biggest problems in defect prediction and is still an active area of research [5].
Finally, we only tested two machine learning algorithms (Logistic Regression and Random Forest) in our study; while these are both well-known and highly used, testing other algorithms would have provided a better basis for comparison. Future studies could assess this limitation through use of multiple datasets, the application of techniques to handle imbalance and the evaluation of multiple prediction models.

Conclusion and Future Research

The machine-learning-based defect prediction framework developed in this study can be used to improve the quality of software processes. The results show that the defect prediction framework using static code metrics on NASA's JM1 dataset has been able to accurately predict defects with over an 80% accuracy rate. It was also found that logistic regression out performed random forest demonstrating the superiority of linear models in defect prediction. Possible future areas of research include developing cost sensitive learning methods, employing techniques such as bootstrapping or bagging, and developing new deep learning models to develop more accurate predictions of defect locations. Additionally, predicting defects across multiple projects could potentially lead to even more robust models that would allow developers to better predict defects.

References

[1] N. E. Fenton and M. Neil, Software Metrics: A Rigorous and Practical Approach, 3rd ed. Boca Raton, FL, USA: CRC Press, 2014.
[2] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," IEEE Transactions on Software Engineering, vol. 33, no. 1, pp. 2–13, Jan. 2007.
[3] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction," IEEE Transactions on Software Engineering, vol. 34, no. 4, pp. 485–496, Jul. 2008.
[4] L. Breiman, "Random forests," Machine Learning, vol. 45, no. 1, pp. 5–32, 2001.

ISSN E: 3006-1466
ISSN P: 3006-1458
CONTEMPORARY JOURNAL OF SOCIAL SCIENCE REVIEW
CJSSR

[5] T. Hall et al., “A systematic literature review on fault prediction performance,” IEEE Transactions on Software Engineering, vol. 38, no. 6, pp. 1276–1304, Nov. 2012.
[6] I. H. Witten, E. Frank, and M. A. Hall, Data Mining: Practical Machine Learning Tools and Techniques, 3rd ed. Morgan Kaufmann, 2011.
[7] J. Demšar et al., “The pipeline design pattern for reproducible research,” Information Systems, vol. 72, pp. 1–13, 2018.
[8] M. Shepperd and S. MacDonell, “Evaluating prediction systems in software engineering,” Empirical Software Engineering, vol. 17, no. 1–2, pp. 3–31, 2012.
[9] A. Tosun, B. Turhan, and A. Bener, “Practical considerations in deploying AI for defect prediction,” Information and Software Technology, vol. 51, no. 11, pp. 1240–1250, 2009.
[10] J. Nam and S. Kim, “Heterogeneous defect prediction,” IEEE Transactions on Software Engineering, vol. 41, no. 2, pp. 124–136, Feb. 2015.
[11] Z. He et al., “Learning from imbalanced data for defect prediction,” Information and Software Technology, vol. 67, pp. 87–100, 2015.
[12] R. Kohavi, “A study of cross-validation and bootstrap for accuracy estimation,” Proc. IJCAI, pp. 1137–1143, 1995.
[13] P. He and X. Li, “Towards cost-sensitive defect prediction,” Journal of Systems and Software, vol. 123, pp. 30–41, 2017.
[14] F. Peters et al., “Better defect prediction with simpler models,” Empirical Software Engineering, vol. 24, no. 4, pp. 2162–2191, 2019.
[15] NASA Metrics Data Program, “JM1 Software Defect Dataset,” NASA, 2004.