Cureus Journal of Computer Science

Open Access Original Article

Multilayer Perceptron of Software Complexity Metrics for Explainable Multicollinearity Mitigation and Defect Localization

Received 12/17/2024
Review began 12/18/2024
Review ended 12/31/2024
Published 01/02/2025

DOI:
https://doi.org/10.7759/s44389-024-02871-z

Taiwo O. Olaleye [1] , Dada A. Aborishade [1] , Oluwasefunmi Arogundade [1] , Adebayo Abayomi-Alli [2, 1] , Olusola J. Adeniran [3]

1. Computer Science, Federal University of Agriculture, Abeokuta, Abeokuta, NGA 2. HumanISE, Institute for Systems and Computer Engineering, Technology and Science, Porto, PRT 3. Mathematics, Federal University of Agriculture, Abeokuta, Abeokuta, NGA

Corresponding author: Taiwo O. Olaleye, olaleyeto@funaab.edu.ng

Abstract

Software defect prediction (SDP) models often neglect the issue of multicollinearity in training sets, particularly among software complexity metrics, which can threaten internal validity and obscure model predictions. This study aimed to investigate the severity of multicollinearity in software complexity metrics, assess their influence on a multilayer perceptron (MLP) model for defect prediction, and evaluate the model's performance with two multicollinearity mitigation techniques. Analysis of the KC1 dataset revealed severe multicollinearity in metrics like lines of code (variance inflation factor [VIF] = 98.08) and cyclomatic complexity (VIF = 399.22). The derived KC1_D dataset of this study exhibited lower VIF values, reducing multicollinearity and improving model stability. The MLP achieved a recall of 0.8452 on the KC1, and a marginal performance improvement on the derived KC1_D, demonstrating resilience against multicollinearity. Principal component analysis (PCA) transformation eliminated multicollinearity by reducing KC1_D attributes to 12 principal components, resulting in perfect classification performance (accuracy, precision, recall, and F1-score = 1.00). Key findings also highlighted feature thresholds influencing software defectiveness, including *lines of code + comments* (locCodeAndComment) attribute with threshold <= −0.19 and *rate of difficulty in testing software* (ev(g)) attribute with threshold <= −0.31. Derived attributes like the operator-to-operand ratio (OOR > 0.49) and logical lines of comments (IOComment ⩽ −0.33) also contributed significantly to the decision of the MLP model. This study concludes that addressing multicollinearity in software complexity metrics is critical for enhancing the explainability, reliability, and performance of SDP models, particularly through PCA as an effective mitigation technique.

Categories: Software Quality Assurance (SQA), Data Mining, Machine Learning (ML)
Keywords: software complexity metrics, multilayer perceptron, multicollinearity, explainable ai, software quality assurance, principal component analysis, variance inflation factor

Introduction

Software defect prediction (SDP) is an essential aspect of software engineering, as it enables developers to identify and address potential issues before they escalate [1], thereby improving the overall quality and reliability of software systems [2]. One of the key challenges in SDP is the presence of multicollinearity, which is hardly investigated prior to predictive analytics of classification modelling. Multicollinearity arises when two or more predictor variables are highly correlated, leading to redundant information that can distort the results of predictive models [1]. While its effects have been well documented in regression-based models, less attention has been given to its impact on classification models used in SDP [3]. The assumption that multicollinearity only influences regression analysis has led to a gap in understanding how it may affect classification-based approaches [4], which are often employed in SDP tasks.

Code metrics in the KC1 datasets are typically designed to capture different aspects of software complexity, such as lines of code (LOC), cyclomatic complexity, and branching factors, among others [5]. These metrics, when analyzed together, may introduce multicollinearity due to their inherent relationships [6]. Existing SDP models are not specifically analyzed on KC1 sets, but on the entire software metrics set in the PROMISE repository, in order to understand the ability of SDP models on complexity metrics contained in KC1 folder. Understanding how multicollinearity manifests across the set of related software complexity metrics in KC1 is also crucial for effective SDP [1]. Ignoring these two problems can lead to erroneous overgeneralization of the performance metrics of SDP models for software quality assurance [7].

The main aim of this study is to explore the implication of software complexity metrics on a deep learning-based SDP model, including the extent and impact of multicollinearity on the model. Specific objectives are as follows:

1) To investigate the severity of multicollinearity within software complexity metrics;
2) To assess the importance and influence of the complexity metrics on for SDP;
3) To evaluate the ability of multilayer perceptron (MLP) model for SDP in environments likely characterized by high multicollinearity.

By achieving its aim, the study intends to enhance the explainability of SDP models, providing valuable insights for developers and researchers working to improve software quality.

Multicollinearity in regression and classification modelling

Some scientists argue that multicollinearity does not significantly affect classification models in the same way it impacts regression models due to fundamental differences in how these models operate. In regression analysis, multicollinearity inflates the standard errors of coefficient estimates [8], making it challenging to determine the individual contribution of predictors and potentially leading to unreliable predictions [9]. In contrast, classification models, particularly those based on algorithms like decision trees, random forests, and neural networks, do not rely on linear coefficient estimation. These models often focus on feature splits or transformations that are less sensitive to linear dependencies among predictors [2]. While classification models like decision trees and random forests mitigate multicollinearity better than regression models due to their focus on feature splits, deep learning models like MLP are not entirely immune, as multicollinearity can still affect their convergence and feature interpretation. Additionally, regularization techniques, such as L1 (Lasso) and L2 (Ridge) regularization, commonly used in classification algorithms, can effectively manage multicollinearity by shrinking or eliminating redundant features [9]. As a result, while multicollinearity may still influence feature importance rankings or the interpretability of classification models [1], it is generally considered less problematic for their predictive performance compared to its impact on regression models [10].

Whereas the advantage of classification models over multicollinearity could be evident in existing studies, the peculiarity of non-deep learning models compared to their neuron-learning-based counterparts should be discussed in mitigating multicollinearity. In deep learning classification models, the advantage of reduced sensitivity to multicollinearity is partially guaranteed but not absolute compared to traditional classification models [11]. Deep models rely on hierarchical feature extraction and nonlinear transformations [1], which inherently mitigate the impact of linear dependencies among features [11]. However, they can still suffer from challenges like overfitting or degraded interpretability when multicollinearity exists [12], particularly if the model is not well regularized or trained with sufficient data. Unlike non-deep models like decision trees or logistic regression, which often incorporate explicit mechanisms to handle feature redundancies [13], deep models require careful tuning of techniques such as dropout, batch normalization, or weight regularization to achieve the same robustness [14]. Thus, while deep learning models can effectively handle multicollinearity to a degree, their reliance on extensive training and parameter tuning makes this advantage less consistent compared to simpler, non-deep classification models, especially with demand on computational overhead. It is therefore necessary in this study to investigate the severity of multicollinearity in complex code metrics for SDP, and as well observe the reaction of a deep learning model, MLP, to the statistical problem, within the space of complexity predictor metrics.

Diverse approaches have been employed in existing studies to intentionally mitigate multicollinearity. Correlation-based feature selection mitigation approach was used by Rahman et al. [14] on NASA data. The particle swarm optimization strategy helped the K-nearest neighbour modelling in feature selection of less-correlated predictor attributes in the NASA data. An AUC of 0.984 weighted average is achieved on the MC2 data contained in the NASA repository. The impact of code size, code complexity, and similar predictive source code metrics was investigated in the study by Haldar and Capretz [15] for the prediction of software defects. Cross-project metrics in PROMISE repository is deployed for the study, which employed explainable AI for a white box modelling of secure vector machine classification model. The feature selection approach of the study improved the performance of the SVM model compared to that of the deep learning ANN model, looking at the AUC curve. In the study by Anju and Judith [16], the quantum particle swarm optimization and principal component analysis (PCA) are employed for mitigating highly correlated features for the SDP modelling by ANN. The hybrid approached drastically reduced predictive attributes into few components with high predictive tendencies. The study used the CM1, JM1, KC1, and KC2 datasets in the PROMISE repository by NASA, with varying volumes of attributes and characteristics. The hybrid approach returned improved performance metrics over traditional methods. The approach in Fatwanto et al. [17] employed the CM1 subset of the NASA PROMISE dataset for SDP. The study investigated the abilities of classification models on the oversampled, undersampled, and combined samples of the CM1 training set, with its high multicollinearity tendencies. The ensemble modelling approach of the study is experimented on a five-fold cross-validation to address the problem of over-fitting. The experiments of the comparative study show that the ensemble model performed the SDP better on the combined training set, notwithstanding the inherent multicollinearity.

Materials And Methods

The research uses a quantitative experimental design to investigate the severity of multicollinearity in KC1 data resident in NASA PROMISE repository, containing complexity rate in software code metrics. It also investigates the effect of multicollinearity on defect prediction models, with and without multicollinearity. Two datasets, KC1 and its enhanced version derived purposely for this study, KC1_D, are utilized, with KC1_D including additional 13 derived attributes. Variance inflation factor (VIF) analysis evaluates the severity of multicollinearity in both datasets. MLP classifiers are trained and tested on each dataset to assess model performance. Local Interpretable Model-Agnostic Explanations (LIME) is applied to observe feature importance and decision thresholds in predicting defects. Additionally, PCA is applied to the derived KC1_D set to reduce dimensionality, and the resulting components are used to train and test the MLP. The methodology systematically compares the influence of multicollinearity and feature transformations on model interpretability and predictive accuracy.

The pseudocode in Table *1* outlines the methodological process of this study. The procedure begins with the retrieval of the primary dataset (KC1) containing the software complexity metrics from the NASA PROMISE repository. Derived attributes are generated from the dataset as KC1_D, to explore their potential in addressing multicollinearity. VIF is then calculated for both the KC1 and the derived KC1_D to measure the severity of multicollinearity in the attributes. The datasets are split into training and testing subsets, with 80% allocated for training and 20% for testing. An MLP classifier is trained on the KC1 (TrainKC1) and tested on the corresponding test set (TestKC1) to evaluate its performance. The same process is repeated for the derived dataset (TrainKC1_D and TestKC1_D) to assess whether the derived attributes mitigate multicollinearity and enhance model predictions. The LIME method is applied to both models to analyze and explain the influence of key features on the predictions.

Subsequently, PCA is applied to the derived dataset (KC1_D) to reduce dimensionality while retaining 95% of the dataset's variance. The PCA-transformed dataset is then split into training and testing subsets. The MLP classifier is trained and tested on these subsets (TrainPCA and TestPCA), and its performance metrics are evaluated. This sequential approach allows for a comprehensive comparison of the datasets, highlighting the impact of multicollinearity and the effectiveness of mitigation techniques such as derived attributes and PCA on model performance and interpretability.

The experiment was conducted in a controlled computing environment using a Windows 10 operating system with a hardware configuration of 4GB RAM and 16GB HDD. The KC1 dataset from the NASA PROMISE repository was saved in CSV format for the experiment. Python programming was employed for implementation, using Jupyter Notebook within the Anaconda Navigator IDE for a seamless and interactive coding experience. This setup ensured adequate resources for executing the pseudocode and conducting data preprocessing, analysis, and model training while leveraging the simplicity and flexibility of Python libraries. The implementation utilized several Python libraries to facilitate data analysis and machine learning tasks. Pandas was used for efficient data manipulation and preprocessing, while NumPy supported numerical computations. Scikit-learn provided tools for calculating VIF, implementing the MLP, and performing PCA. The LIME model was employed to interpret model predictions and analyze the importance of predictor attributes. These libraries, integrated within the Python ecosystem, offered robust functionalities for managing and analyzing the dataset effectively.

```
1 Begin
2       KC1=Datasets.GetKC1()
3        KC1_D=Datasets.DeriveAttributes(KC1)
4        VIF_KC1=Multicollinearity.CalculateVIF(KC1)
5        VIF_KC1_D=Multicollinearity.CalculateVIF(KC1_D)
6        TrainKC1, TestKC1=SplitData(KC1, train=0.8, test=0.2)
7        MLP_KC1=MLPClassifier.Train(TrainKC1)
8        MetricsKC1=MLPClassifier.Test(MLP_KC1, TestKC1)
9        TrainKC1_D, TestKC1_D=SplitData(KC1_D, train=0.8, test=0.2)
10      MLP_KC1_D=MLPClassifier.Train(TrainKC1_D)
11      MetricsKC1_D=MLPClassifier.Test(MLP_KC1_D, TestKC1_D)
12      LIME_KC1=LIME.Explain(MLPKC1, TestKC1)
13      LIME_KC1_D=LIME.Explain(MLP_KC1_D, TestKC1_D)
14      PCA_KC1_D=PCA.Apply(KC1_D, varianceThreshold=0.95)
15      TrainPCA, TestPCA=SplitData(PCA_KC1_D, train=0.8, test=0.2)
16      MLP_PCA=MLPClassifier.Train(TrainPCA)
17      MetricsPCA=MLPClassifier.Test(MLP_PCA, TestPCA)
18      LIME_PCA=LIME.Explain(MLP_PCA, TestPCA)
19 End
```

TABLE 1: Pseudocode of the research methodology

Data source and attributes

KC1 dataset contains metrics that measure the complexity in software code. Complexity refers to metrics that quantify the intricacy of the software's structure, such as cyclomatic complexity, control flow depth, and the interdependencies between modules or functions, which are often used to predict defect-prone areas. The KC1 dataset is contained in the NASA PROMISE repository [18], where they are acquired for the purpose of this study. The data instances are labelled as either defective or non-defective. The predictor attributes of the acquired KC1 are described in Table *2*. The data is made up of 22 attributes, including the class variable (defect or non-defect) and 2109 instances of code blocs. This study also seeks to investigate the impact of additional derived attributes in multicollinearity mitigation, as well as enhance the predictive ability of the classification model. Therefore, 13 derived attributes are computed from the existing 21 attributes in KC1 (22nd being the class variable), and are described in Table *3* as KC1_D. The 13 newly computed attributes are aimed at deriving attributes that clarify sources of complexity, explore relationships between attributes, and refine defect predictions. These categories are believed to better project the inherent characteristics of code metrics that indicate the complexity of software products.

Multicollinearity detection in data attributes

To investigate the severity of multicollinearity in complex software code metrics, the VIF of the KC1 set would be investigated to determine the degree of multicollinearity state of the set. In this study, multicollinearity severity in the KC1 training set is investigated using the VIF by quantifying how much the variance of a regression coefficient is inflated in the training set due to multicollinearity among its 21-no predictor attributes. For each predictor X_j in the dataset, VIF is computed as:

$$VIF\,(X_j) = \frac{1}{1 - R_j^2} \quad (1)$$

where R_j^2 is the coefficient of determination from regressing X_j on all other 21 predictors in the KC1 dataset. A higher X_j indicates stronger linear dependence of X_j on the other variables, leading to a larger

VIF value. Common thresholds interpret VIF values above 5 or 10 as indicative of severe multicollinearity. By calculating VIF for each attribute in the KC1 dataset, the degree of multicollinearity is evaluated, identifying predictors that may undermine model reliability.

Principal component analysis

PCA reduces multicollinearity simply by transforming the correlated attributes into a set of uncorrelated components, popularly referred to as principal components (PCs). PCA seeks to decompose the 21-no training set matrix *X* into eigenvalues and eigenvectors by the technique of eigendecomposition of the covariance matrix:

$$X^T X = V \Lambda V^T \quad (2)$$

where, for this study, *X* represent the derived dataset KC1_D, which contains the original and newly generated attributes. It is a data matrix, where each row corresponds to a software code sample, and each column corresponds to a feature or metric (e.g., lines of code, cyclomatic complexity).

X^T is the transpose of the data matrix *X*, where rows and columns are swapped. The product $X^T X$ forms the covariance matrix of the dataset, capturing relationships between features.

V is the matrix of eigenvectors, where each eigenvector corresponds to PC. These eigenvectors represent the directions in the feature space that capture the maximum variance in the data. In the study, these directions highlight combinations of metrics that explain the complexity and multicollinearity in the dataset.

Λ is a diagonal matrix of eigenvalues, where each eigenvalue corresponds to the variance captured by its associated PC. In this study, the eigenvalues help determine how much of the original dataset's variance is retained after dimensionality reduction, with 95% of the variance being the threshold for selecting components.

V^T is the transpose of the eigenvector matrix V. This term ensures that the decomposition reconstructs the covariance matrix in the transformed space.

Each of the PC is a linear combination of the original data variables and is orthogonal to others, therefore eradicating multicollinearity. When the subset of PCs that capture most of the variance is selected, PCA will reduce dimensionality, thereby addressing multicollinearity in the training set, especially if the selected PC subset is based on a cumulative variance threshold.

MLP of the complexity metrics

In MLP, the 22-no KC1 attributes (including the class variable) serve as input features to the input layer and each is passed to individual neurons to apply weights and biases and an activation function that will introduce non-linearity. In the hidden layers, the applied weighted inputs are processed further through multiple neurons, each applying its own set of weights, biases, and activation functions. This allows the MLP to learn complex patterns and relationships between the attributes. In the output layer, the final transformed data are combined into a single neuron, for the binary classification approach of this study, i.e., to predict whether a software code segment is defective or non-defective. At this point, a softmax activation function would determine the output probabilities, by classifying each instance as either a defect or non-defect based on a decision threshold. The 22 attributes iteratively adjust their weights across the layers during backpropagation to minimize the classification error, learning the optimal representation for the defect prediction task.

The workings of an MLP involve three main processes of forward propagation, loss calculation, and backpropagation.

Forward Propagation

For the forward propagation, each layer of the MLP would process the KC1 dataset in the following ways:

In the input layer, given an input vector x = [x1, x2, ..., xn], which are the 22 attributes, each neuron would compute the weighted sum of the inputs:

$$z_j^{(1)} = \sum_{i=1}^{n} w_{ij} x_i + b_j, j = 1, 2, ..., m \quad (3)$$

where

w_{ij} is the weight connecting input *i* to neuron *j*.
b_j is the bias for neuron *j*.
and $z_j^{(1)}$ is the pre-activation output of the neuron.

For each hidden layer *l*, the activation function *g*(.) is applied to the pre-activation output:

$$a_j^{(l)} = g(z_j^{(l)}) = g\left(\sum_{i=l}^{m^{(l-1)}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}\right) \text{(4)}$$

where
$a_j^{(l)}$ is the activation output of neuron *j* in layer *l*.
and *g*(.) is the non-linear sigmoid activation function adopted in this study.

The final output layer transforms the activations into predictions:

$$\tilde{y} = g\left(\sum_{i=l}^{m^{(L)}} w_i^{(L)} a_i^{(L)} + b^{(L)}\right) \text{(5)}$$

then for a binary classification of this study, a sigmoid activation is used as:

$$(\tilde{y} = \frac{1}{1+e^{-2}} \text{(6)}$$

where $\tilde{y}$ is the predicted probability of the 'defect' or 'non-defect' class.

Loss Function

The MLP model evaluates its own prediction decision using a loff function for the binary classification problem. The binary cross-entropy loss is used thus:

$$\iota = -\frac{1}{N}\sum_{i=1}^{N}(y_i log(\tilde{y}_i) + (1 - y_i log(1-\tilde{y}_i)) \text{(7)}$$

where
y_i is the true label for sample *i*.
$\tilde{y}_i$ is the predicted probability for sample *i*.

Backpropagation

The MLP model updates its weights and biases at this stage using the gradient descent. Chain rule is used for gradients computation, thus:

Updating the weight

$$w_{ij} \leftarrow w_{ij} - \eta\frac{\partial\angle}{\partial w_{ij}} \text{(8)}$$

where
η is the learning rate.
$\frac{\partial\angle}{\partial w_{ij}}$ is the gradient of the loss with respect to the weight.

Bias update

$$b_j \leftarrow b_j - \eta\frac{\partial\angle}{\partial b_j} \text{(9)}$$

The gradients are computed in each layer and then propagate errors in a backward manner from the

output layer towards the input layer.

LIME XAI interpretation of the MLP black box

The role of the XAI model, LIME, in this study is to unravel the black box methodology of the MLP in order to unravel why the model makes its predictive decisions based on the predictor attributes available at every instance. It is designed to approximate the decision boundary of the black-box modelling with an interpretable linear modelling with proximity to each prediction, which then aids explainability. Given the 21 predictor attributes, and the target variable defect or non-defect, Let:

$$x = (loc, v, ev, iv, n, v, l, ..., branchcount) \quad (10)$$

be the instance to be explained, where *loc*, *v*, *ev*, *iv*, *n*, *v*, *l*, ..., *branchcount* represent the predictor attributes contained in the KC1 complex metrics dataset.

To unravel the relevance of each feature in *x*, LIME generates perturbed samples around *x*, by modifying the values of the elements slightly. The perturbed samples are represented as:

$$\left\{x_1^{'}, x_2^{'}, ..., x_n^{'}\right\} \quad (11)$$

Samples closer to *x* would exhibit more influence on the explanation, achieved by weight assignment *w* to each perturbed sample as a function of its distance from *x*. The exponential kernel weighting function employed by LIME is such as:

$$w(x, x^{'}) = exp(-d\frac{(x, x^{'})^2}{\sigma^2}) \quad (12)$$

where
$d(x,x')$ is the Euclidean distance between *x* and *x'*.
σ is a scaling factor that controls the width of the locality around *x*.

By uncovering the MLP black box for each perturbed *x'*, the predicted probability for either of defect or non-defect class is determined as planned. With f(*x'*) denoting the predicted probabilities for *x'*, next step is to fit a local surrogate model using the perturbed samples $\{x_i'\}$, their weights $\{w_i\}$, and the predicted probabilities $\{f(x_i')\}$. LIME fits a local interpretable model $g(x)$ around the instance *x*. Typically, $g(x)$ is a simple linear model:

$$g(x^{'}) = \beta_0 + \beta_1.loc + \beta_2.v + \beta_3.ev + \beta_4.iv + \beta_5.n, +..., +\beta_n.x_n \quad (13)$$

where $\beta_1, \beta_2, \ldots, \beta_n$ are the coefficients representing the contribution of each 21 predictor KC1 attributes to the prediction of the 22nd class attribute. The linear model is then trained to minimize the weighted loss:

$$Loss(f, g, w) = \sum_i w(x, x_i^{'}).((x_i^{'}))^2 + \theta_{(g)} \quad (14)$$

where $\theta_{(g)}$ is a regularization term to encourage simplicity in making it easier to interpret.

After fitting $g(x)$, the coefficients β_i indicate the contribution of each 21 feature attribute to the prediction for the specific instance *x*:

Positive β_i means the predictor attribute increases the probability of the predicted class.
Negative β_i means the predictor attribute decreases the probability.

S/N	Attribute	Description	Relationship With Code Complexity
1	loc	Lines of Code	Higher values indicate more complex code, increasing the likelihood of defects due to difficulty in understanding and maintaining the code.
2	v(g)	Cyclomatic Complexity: Number of independent paths in the code	A direct measure of complexity; higher values suggest more decision points, which are prone to defects.
3	ev(g)	Essential Complexity: Measures complexity ignoring structured	Reflects core logic complexity; higher values indicate difficulty in

		programming	testing and debugging, potentially leading to defects.
4	iv(g)	Design Complexity: Cyclomatic complexity considering intermodule interactions	Indicates design-level complexity; complex designs are more prone to defects due to unclear module boundaries.
5	n	Halstead Length: Total number of operators and operands in the code	Longer code often implies higher complexity, increasing the chance of introducing defects.
6	v	Halstead Volume: Measures the size of the implementation	Larger volumes indicate higher cognitive effort required to understand the code, leading to more defects.
7	l	Halstead Level: Inverse measure of difficulty to understand the code	Lower levels imply higher complexity, making the code harder to debug and maintain, thus more defect-prone.
8	d	Halstead Difficulty: Measures the difficulty of programming	Higher difficulty suggests increased complexity, often resulting in a higher likelihood of defects.
9	i	Halstead Intelligence: Indicates the maintainability of the code	Lower intelligence implies poor maintainability, increasing the chances of defects over time.
10	e	Halstead Effort: Effort required to implement or understand the code	Higher effort reflects increased complexity, which is positively correlated with defects.
11	b	Halstead Bugs: Estimated number of defects in the code	Directly estimates defects based on complexity; higher values indicate more defect-prone code.
12	t	Halstead Time: Time required to implement the code	Higher time reflects higher complexity, suggesting greater potential for defects.
13	lOCode	Logical Lines of Code: Non-comment, non-blank lines	More logical lines correlate with higher complexity and increased defect likelihood.
14	lOComment	Logical Lines of Comments: Number of comment lines	May indicate effort to manage complexity; however, inadequate commenting can obscure understanding and increase defects.
15	lOBlank	Logical Blank Lines: Number of blank lines	Blank lines may improve readability and mitigate perceived complexity, but excessive blank lines can also mask code structure, leading to oversight.
16	locCodeAndComment	Lines of Code + Comments: Combined code and comment lines	A higher count may imply complex modules with extensive documentation to clarify complex code.
17	uniq_Op	Unique Operators: Number of distinct operators in the code	A higher variety of operators can increase complexity, potentially introducing subtle logic defects.
18	uniq_Opnd	Unique Operands: Number of distinct operands in the code	More unique operands can indicate complexity in variable usage and relationships, which may lead to defects.
19	total_Op	Total Operators: Total count of operators in the code	An excessive number suggests overly complex logic, increasing defect risk.
20	total_Opnd	Total Operands: Total count of operands in the code	High counts can indicate dense and complex computations, contributing to defects.
21	branchCount	Number of branching statements (e.g., if, while, for)	More branches indicate higher complexity in control flow, often leading to defects in untested paths.
22	class	Defect Status: Indicates if the module is defective (true) or not (false)	Direct output of defect prediction; correlates with various measures of complexity and serves as the dependent variable for machine learning models.

TABLE 2: Attributes of the KC1 data containing code metrics that indicate software complexity

S/N	Derived Attribute	Formula	Purpose	Category
1	Comment Density (CD)	lOComment/loc	Assesses documentation adequacy.	Clarify Sources of Complexity
2	Logical to Physical Code Ratio (LPCR)	lOCode/loc	Distinguishes functional vs filler code.	
3	Effort Per Logical Line (EPLL)	e/lOCode	Relates coding effort to logical size.	Explore Relationships
4	Effort Per Branch (EB)	e/branchCount	Explores coding effort at decision points.	
5	Operator-to-Operand Ratio (OOR)	uniq_Op/uniq_Opnd	Balances operators vs operands.	
6	Redundancy Measure (RM)	(total_Op + total_Opnd)/(uniq_Op + uniq_Opnd)	Indicates inefficiencies in usage.	Clarify Sources of Complexity
7	Branching Density (BD)	branchCount/loc	Identifies branching intensity.	
8	Decision Complexity (DC)	branchCount/v(g)	Examines control flow intricacies.	Explore Relationships
9	Defect Density (DD)	class(true_count)/loc	Tracks defect frequency per line.	Refine Defect Predictions
10	Weighted Defect Risk (WDR)	b * d	Combines defect risk and difficulty.	
11	Normalized Halstead Bugs (NHB)	b/loc	Adjusts defect predictions for size.	
12	Complexity Normalized by Lines (CNL)	v(g)/lOCode	Complexity per logical line.	Clarify Sources of Complexity
13	Interaction Density (ID)	iv(g)/ev(g)	Focuses on inter-module complexities.	Explore Relationships

TABLE 3: Description of the derived KC1_D dataset

Results

The experimental result is discussed in this section, and the result would be presented with respect to the objectives of the study. In the acquired NASA PROMISE dataset, 22 attributes are contained in the KC1 folder, which is representing software complexity metrics as earlier described in Table *1*. Instances of code modules with false labels are 1,783, while 326 of the modules are defective.

Objective 1: To investigate the severity level of multicollinearity in complexity code metrics used for software development

Towards investigating the severity of multicollinearity in complexity software code metrics, prior to MLP modelling, two approaches of VIF and pairwise correlation coefficient computation are implemented on predictor attributes. The VIF is expected to conduct a *1 to Many* test between a target attribute and the rest of the attributes, examining how much the variance of a regression coefficient is inflated due to collinearity with other predictors. Table *4* shows the result of the test, indicating a high severity level of multicollinearity in complexity software code metrics. The attributes *Lines of Code* (*loc* with VIF 98.08), *Cyclomatic Complexity* (v(g) with VIF 399.22), *Halstead Difficulty* (d with VIF 50.59), and *Logical Lines of Code* (lOCode with VIF 88.364) exhibit severe cases. Investigating a *1 to 1* multicollinearity status using the correlation coefficient (which measures linear relationship), similar high severity level of multicollinearity status is also discovered, as presented in the heat map of Figure *1*. As can be observed, only the *l* attribute (Halstead Level: Inverse measure of difficulty to understand the code) exhibits a weak correlation with other attributes (between 0.2 and 0.50), unlike the widespread strong positive correlation existing between other attributes. Table *5* shows the VIF test on the derived KC1_D set. Observation shows derived attributes cyclomatic density (CD with VIF 50.2663), effort per branch (EB with VIF 29.0741), branching density (BD with VIF 40.3946), and weighted defect risk (WDR with VIF 8346.63900) show extreme cases of multicollinearity. In relation to the pairwise correlation existing in the KC1_D, Figure *2*

shows a better correlation between traditional attributes and the derived ones.

Objective 2: To investigate the importance and influence of the complexity attributes on MLP model for software defect localization in code metrics

This objective seeks to show how thresholds or conditions of each attribute contribute positively or negatively to the classification of data points as defects (True) or non-defects (False). This is achievable through the LIME model, which returns a white box version of the MLP black box methodology. In the white box output presented in Table *6*, the derived attribute OOR*,* which balances operators vs operands (with a threshold of >0.49 and a weight of −0.187), and the traditional attribute logical lines of comments (lOComment) (⩽−0.33, weight −0.113) demonstrate significant negative contributions, suggesting these thresholds are strongly associated with non-defective classifications. Conversely, features like locCodeAndComment (⩽−0.19, weight 0.096) and derived attribute CD (>0.29, weight 0.066) positively contribute. On the LIME visualization plot of Figure *3*, the analysis for the True class (defective) in the KC1_D dataset highlights several features that play key roles in predicting the occurrence of the True class. Notably, OOR with threshold >0.49 has the most significant negative weight of −0.19, indicating that higher values of OOR decrease the likelihood of the True class. Other influential features include lOComment (Threshold <= −0.33), with a weight of −0.11, and locCodeAndComment (Threshold <= −0.19), contributing positively (weight = 0.10). The thresholds for attributes like v(g) (Threshold <= −0.48) and iv(g) (Threshold <= −0.47) further show that low values of these variables favor the True class. The model's decision-making is influenced by these precise conditions, with thresholds offering clear insights into the significant factors driving the True class occurrence. For the False class (non-defect), the LIME analysis in Figure *4* reveals critical features that influence the prediction of False class occurrences. A key feature is locCodeAndComment (Threshold <= −0.19), which appears significant in classifying the False class. Similarly, lOComment (Threshold <= −0.33) and n (Threshold <= −0.55) have similar thresholds, strongly influencing the model to classify as False when these conditions are met. v(g) (Threshold <= −0.48) and branchCount (Threshold <= −0.48) also have important negative weights, indicating that lower values of these features push the model toward the False class. Additionally, features like DC (Threshold <= −0.72) and total_Op (Threshold <= 0.56) contribute to the False classification by shifting the decision boundary, underscoring the diverse that determine False class predictions.

Objective 3: To evaluate the performance of MLP with complexity and multicollinearity inhibitors

The third objective is novel as it seeks to investigate the capacity of MLP in the pool of complexity code metrics as predictor variables and likewise its behaviour in a severe multicollinearity environment. An 80% and 20% split is implemented as training and testing sets for the MLP on the three datasets of KC1, KC1_D, and the PCA-Treated KC1_D. In addition to the model's revelations in black box, as revealed by LIME in white box and explained earlier, the MLP exhibited diverse performances across the three datasets. The performance evaluation plot of MLP on KC1 and KC1_D, respectively, is presented in Figure *5*. The model returned a good accuracy result of 0.8452 and 0.85468 on the KC1 and the KC1_D, respectively. The same trend is recorded in precision, recall, and F1-score of 0.8452/0.85468, 0.8145/0.83, 0.8452/0.85, and 0.8189/0.83, respectively, on KC1 and derived KC1_D. The insignificant improvement of 0.01 by MLP from highly multicollinearized KC1 and KC1_D sets somewhat describes the insignificant mitigation brought by the derived attributes in terms of accuracy. A likewise insignificant 0.02, 0.01, and 0.01 improvement average recorded on precision, recall, and F1-score, respectively, shows the same trend. Therefore, MLP remained efficient across the two datasets, notwithstanding the severity levels of their multicollinearity status.

The KC1_D dataset is later treated to a PCA, transforming the 36-no dataset attributes into 12 PCs, which explained 95% of the total variance. This dimensionality reduction process successfully summarized the data while eliminating multicollinearity, as indicated by the fact that the PCs were orthogonal and thus independent. The transformed data set is more compact and easier to analyze, maintaining the essential structure of the data while simplifying the complexity by reducing the number of variables from the original feature set to just 12 components. The experimental result returned a 1.000 weighted average for MLP in terms of accuracy, precision, recall, and the F1-measure. In order to verify the validity of the perfect evaluation result, a 10-fold cross-validation methodology is again employed for the training of MLP on the components. The weighted average remained 1.000 across the evaluation metrics. Ten-fold cross-validation is a methodological approach that splits the dataset into 10 subsets, trains the model on nine subsets, tests it on the remaining one, and averaging the results to ensure reliability and reduce overfitting. Compared to the train-test split approach adopted earlier, 10-fold cross-validation provides a more reliable and robust evaluation of model performance by reducing variability and ensuring the model is tested on multiple data subsets. This enhances the model's generalization ability by exposing it to diverse training and testing subsets, reducing the risk of overfitting and improving performance on unseen data.

These significant improvements show the efficiency of PCA in mitigating severe multicollinearity in software complexity metrics, thereby enhancing the ability of the perceptron model with its multilayer approach. The weighted averages of 0.1548, 0.1855, 0.1548, and 0.1811, respectively, on accuracy, precision, recall, and f1-measure with PCA treatment on KC1_D speak to the improvement on the MLP ability. The contributions of the newly created 12 PCA components to the SDP modelling by MLP is visualized in Figure 6. The threshold values such as PC1 <= −2.79 and PC2 > 0.98 indicate how the model uses specific ranges of PCs to make its predictions. The weights attached to each PC (e.g., PC1: −3.45, PC2: 3.91) suggest the relative importance of these components in determining whether the model will classify a sample as class 0 or class 1. Negative values for certain components, such as PC1, indicate that lower values of PC1 are associated with class 0 (False, non-defective), while higher values of PC2 are aligned with class 1 (defective). The detailed LIME explanation thus provides transparency into how each PC influences the final decision, ensuring that the model's behaviour is both interpretable and justifiable.

A further test is conducted on the MLP model to evaluate its ability through cluster analysis. This is achieved by re-grouping the KC1_D based on similarity and reducing it to the 13 derived attributes only (instead of the entire 36). On the new cluster, PCA reduced the 13 attributes in the cluster to 8 components (explaining 95% variance) and still returned a weighted average of 1.000 across the performance evaluation metrics.

Feature	VIF
loc	98.08296
v(g)	399.2242
ev(g)	3.475817
iv(g)	22.78379
n	7.21E+08
v	3304.388
l	1.733132
d	50.59734
i	26.21133
e	7.29E+10
b	2485.354
t	7.29E+10
lOCode	88.36428
lOComment	3.003043
lOBlank	4.793682
locCodeAndComment	1.292976
uniq_Op	28.77085
uniq_Opnd	135.1498
total_Op	2.76E+08
total_Opnd	1.06E+08
branchCount	402.6574

TABLE 4: VIF-based multicollinearity test result on KC1 attributes

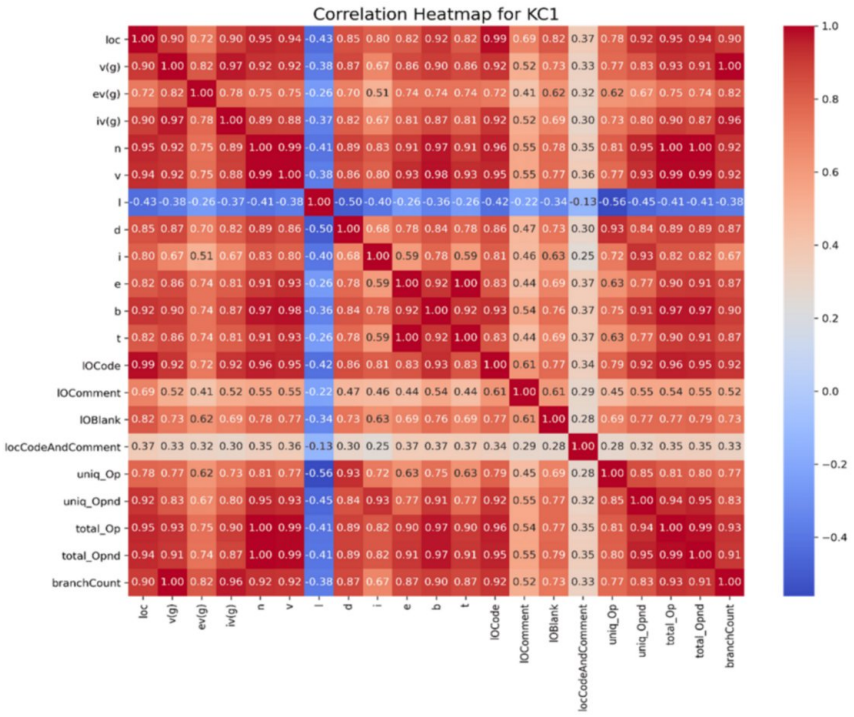

FIGURE 1: Pairwise multicollinearity test result on KC1 attributes

Derived Feature	VIF
CD	50.2663
CoD	3.5239
LPCR	7.1409
EB	29.0741
OOR	2.3826
RM	36.1349
BD	40.3946
DC	5.8692
DD	NaN
WDR	83467.3900
NHB	1344.2200
CNL	2.2342
ID	3.7918

TABLE 5: VIF-based multicollinearity test result on KC1_D attributes

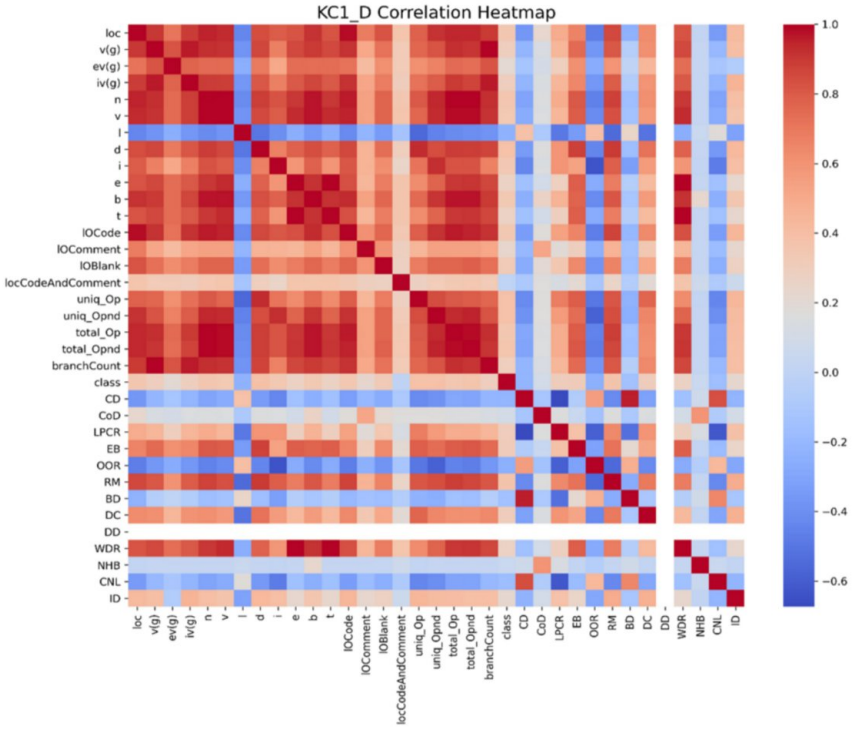

FIGURE 2: Pairwise multicollinearity test result on KC1_D attributes

Feature	Threshold/Condition	Weight
OOR	> 0.49	-0.18714173636589462
lOComment	<= -0.33	-0.1131345650246071
locCodeAndComment	<= -0.19	0.0964820126903431
n	<= -0.55	0.08323519624763115
loc	<= -0.56	-0.07695244048581534
v(g)	<= -0.48	0.07544025410990893
EB	<= -0.55	-0.07257443737405454
iv(g)	<= -0.47	-0.06983908944710025
CD	> 0.29	0.06589879310937698
branchCount	<= -0.48	0.0654047520202537

TABLE 6: Feature importance for the MLP modelling

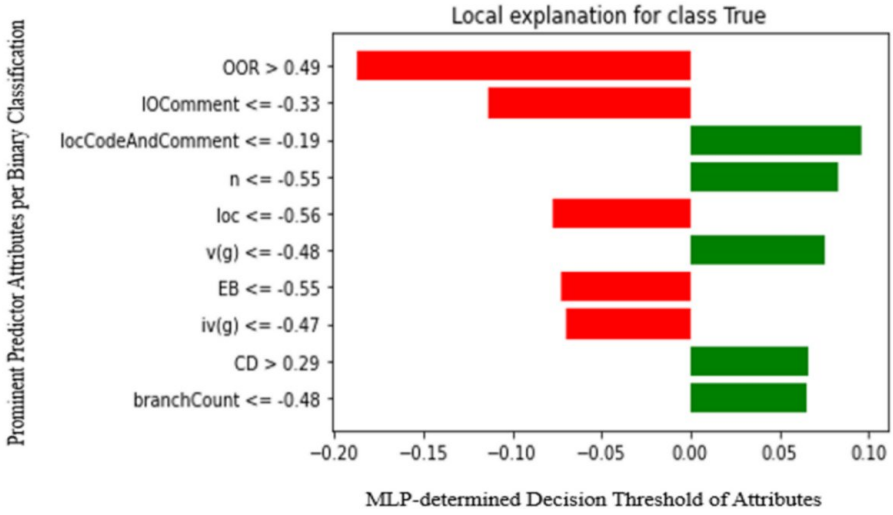

FIGURE 3: Attribute contribution and prediction probability plot for True (defective) instance

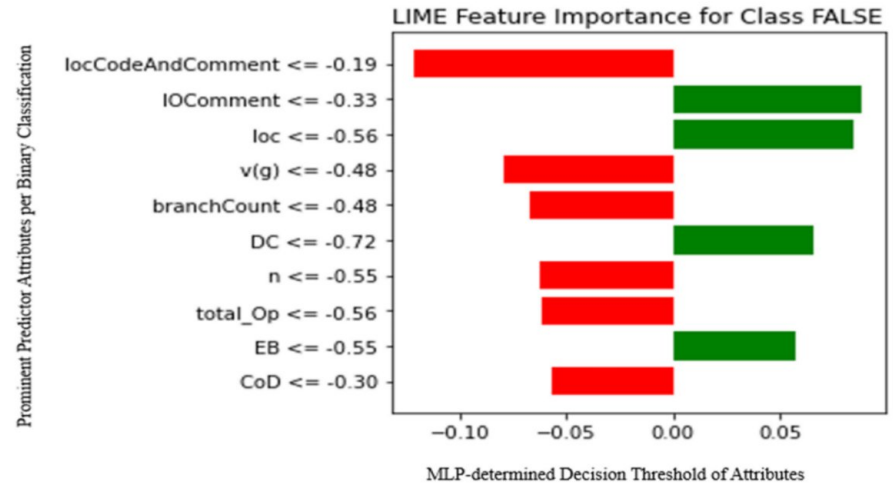

FIGURE 4: Attribute contribution and prediction probability plot for False (non-defective) instance

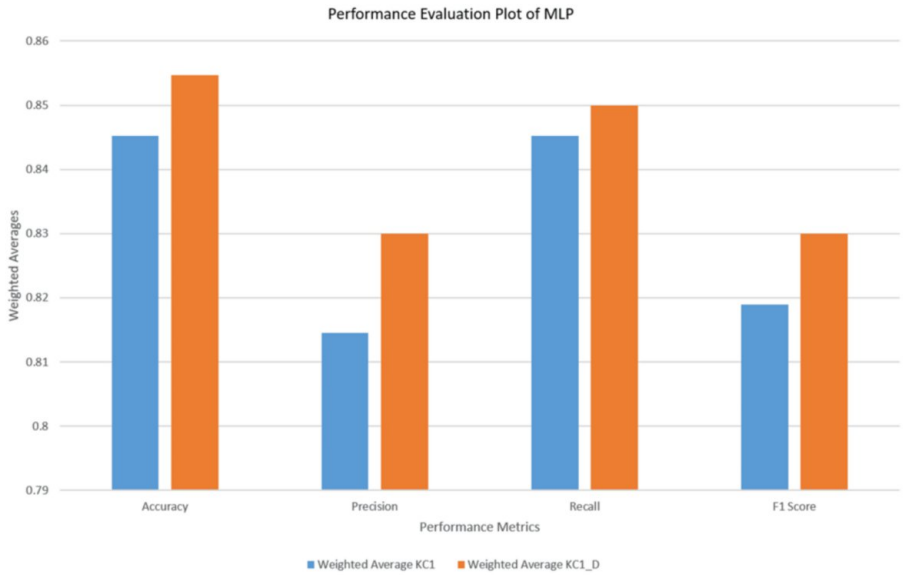

FIGURE 5: Performance evaluation of MLP on KC1 and derived KC1_D

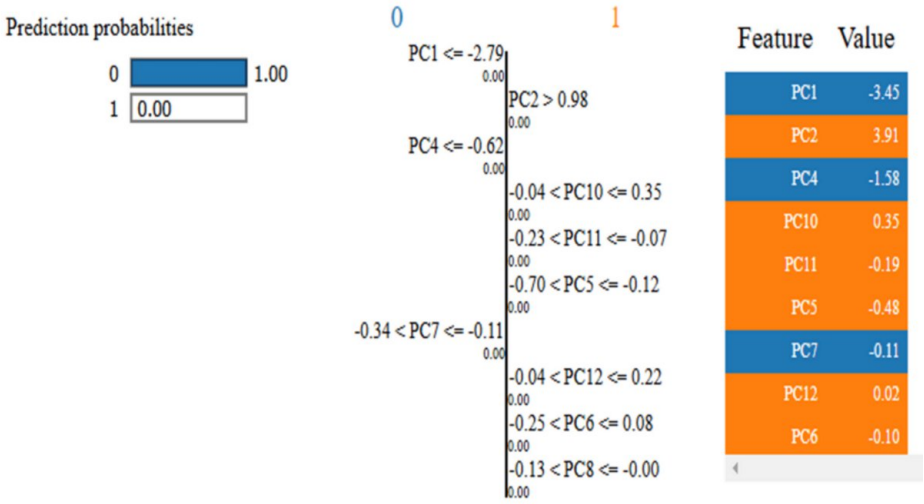

FIGURE 6: Attribute contribution and prediction probability plot on 12 principal components returned on KC1_D

Discussion

Experimental analysis and implications

This study examined the impact of multicollinearity on software defect prediction using NASA's KC1 dataset and its derived version, KC1_D. VIF analysis revealed severe multicollinearity in metrics like cyclomatic complexity (VIF 399.22) and lines of code (VIF 98.08), necessitating feature derivation and PCA treatment. The MLP classifier showed robust performance despite multicollinearity, achieving accuracy and F1-scores above 0.84 across both datasets. PCA transformation further improved classification performance to a perfect score of 1.000 by reducing 35 features to 12 orthogonal components. Further analysis is done to validate the perfect result. A 10-fold cross-validation experiment is conducted on the KC1_D to ascertain the ability of MLP on the strength of the PCA. The cross-validation approach likewise returned perfect weighted averages for MLP on the derived set. Furthermore, a cluster analysis is conducted to be sure that the MLP will return similar performance on unseen data in order to be able to generalize the result of this study. Therefore, the 13 similar derived attributes are sent for multicollinearity mitigation and are reduced to eight components explaining 95% variance. The MLP is trained on the eight components for SDP, and the model achieved the same perfect performance evaluation metrics. Interpretability via LIME highlighted key thresholds for the predictive attributes,

providing actionable insights for defect localization contributions. These results underscore the importance of multicollinearity mitigation through feature engineering and dimensionality reduction in enhancing model reliability and interpretability.

PCA was chosen over alternative dimensionality reduction techniques like t-distributed stochastic neighbour (t-SNE) or autoencoders due to its ability to effectively address multicollinearity while maintaining ease of interpretation and computational efficiency. Unlike t-SNE, which focuses on preserving local structure and is more suited for visualization, and autoencoders, which require extensive hyperparameter tuning and higher computational resources, PCA transforms correlated features into orthogonal principal components, directly reducing multicollinearity. This approach aligns well with the study's goal of enhancing model stability and explainability in software defects, as well as a computationally efficient approach.

Limitations

The exploration of multicollinearity mitigation techniques was limited to feature engineering and PCA. Future research will validate the results using alternative mitigation approaches to broaden applicability and robustness.

Conclusions

The aim of this study was to explore the relationship between software complexity metrics and SDP modelling, with a focus on mitigating the challenges posed by multicollinearity and evaluating the predictive performance of an MLP model. Specifically, the study sought to (1) investigate the severity of multicollinearity within software complexity metrics, (2) assess the importance and influence of these metrics on MLP predictions, and (3) evaluate the MLP's performance in environments characterized by high multicollinearity, as well as after applying dimensionality reduction techniques. To achieve these objectives, a comprehensive methodological approach was adopted, involving the computation of VIF and correlation coefficients to analyze multicollinearity, application of LIME to interpret MLP predictions, and evaluation of the MLP's performance across original, derived, and PCA-treated datasets. Results revealed significant multicollinearity among traditional complexity metrics like cyclomatic complexity and lines of code, as well as derived metrics like weighted defect risk in the KC1_D dataset. Cluster analysis is also conducted on the dataset to further mitigate the effect of multicollinearity. LIME played a crucial role in enhancing interpretability by providing insights into the MLP model's decision-making process. LIME's explanations revealed feature thresholds, such as those for locCodeAndComment and ev(g), offering a deeper understanding of how these attributes influenced predictions, thus complementing PCA's dimensionality reduction. The analysis demonstrated how specific attribute thresholds, such as operator-to-operand ratio (>0.49) and logical lines of comments (⩽−0.33), influenced defect classification. Finally, while the MLP performed robustly across multicollinear datasets, PCA treatment significantly enhanced its performance, achieving perfect accuracy, precision, recall, and F1-score, underscoring the efficacy of dimensionality reduction in mitigating multicollinearity and optimizing predictive modelling for software defect detection. The model performed well on high collinearity and way better on PCA-treated sets. Both train-test split and cross-validation methodologies were employed on the model for validation of its good performance, with even better performance. In future work, applying the approach to real-world software systems (aside from benchmarks in repositories) would provide practical insights into its effectiveness in live environments.

Additional Information

Author Contributions

All authors have reviewed the final version to be published and agreed to be accountable for all aspects of the work.

Concept and design: Taiwo O. Olaleye, Dada A. Aborishade, Oluwasefunmi Arogundade, Adebayo Abayomi-Alli, Olusola J. Adeniran

Acquisition, analysis, or interpretation of data: Taiwo O. Olaleye, Dada A. Aborishade, Oluwasefunmi Arogundade, Adebayo Abayomi-Alli, Olusola J. Adeniran

Drafting of the manuscript: Taiwo O. Olaleye, Dada A. Aborishade, Oluwasefunmi Arogundade, Adebayo Abayomi-Alli, Olusola J. Adeniran

Critical review of the manuscript for important intellectual content: Taiwo O. Olaleye, Dada A. Aborishade, Oluwasefunmi Arogundade, Adebayo Abayomi-Alli, Olusola J. Adeniran

Supervision: Dada A. Aborishade, Oluwasefunmi Arogundade, Adebayo Abayomi-Alli, Olusola J. Adeniran

Disclosures

Human subjects: All authors have confirmed that this study did not involve human participants or tissue. **Animal subjects:** All authors have confirmed that this study did not involve animal subjects or tissue. **Conflicts of interest:** In compliance with the ICMJE uniform disclosure form, all authors declare the following: **Payment/services info:** All authors have declared that no financial support was received from any organization for the submitted work. **Financial relationships:** All authors have declared that they have no financial relationships at present or within the previous three years with any organizations that might have an interest in the submitted work. **Other relationships:** All authors have declared that there are no other relationships or activities that could appear to have influenced the submitted work.

References

1. Olaleye TO, Arogundade OT, Misra S, Abayomi-Alli A, Kose U: Predictive analytics and software defect severity: A systematic review and future directions. Scientific Programming. 2023, 2023:6221388. 10.1155/2023/6221388
2. Shafiq M, Alghamedy FH, Jamal N, Kamal T, Daradkeh YI, Shabaz M: Retracted: Scientific programming using optimized machine learning techniques for software fault prediction to improve software quality [RETRACTED]. IET Software. 2023, 17:694-704. 10.1049/sfw2.12091
3. Yildirim H: The multicollinearity effect on the performance of machine learning algorithms: Case examples in healthcare modelling. Journal of Engineering & Smart Systems. 2024, 12:68-80. 10.21541/apjess.1371070
4. Garg A, Tai K: Comparison of statistical and machine learning methods in modelling of data with multicollinearity. International Journal of Modelling, Identification and Control. 2013, 18:295-312. 10.1504/ijmic.2013.053535
5. Nasser AB, Ghanem WAHM, Saad AMHY, Abdul-Qawy ASH, Ghaleb SAA, Alduais NAM, Ghetas M: Depth linear discrimination-oriented feature selection method based on adaptive sine cosine algorithm for software defect prediction. Expert Systems with Applications. 2024, 253:124266. 10.1016/j.eswa.2024.124266
6. Pradhan S, Nanniyur V, Vissapragada PK: On the defect prediction for large scale software systems: From defect density to machine learning. In 2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS). 2020, 374-381. 10.1109/QRS51102.2020.00056
7. Atif F, Rodriguez M, Araújo LJP, Amartiwi U, Akinsanya BJ, Mazzara M: A survey on data science techniques for predicting software defects. In International Conference on Advanced Information Networking and Applications. 2021, 227:298-309. 10.1007/978-3-030-75078-7_31
8. Kim JH: Multicollinearity and misleading statistical results. Korean Journal of Anesthesiology. 2019, 72:558-569. 10.4097/kja.19087
9. Lavery MR, Acharya P, Sivo SA, Xu L: Number of predictors and multicollinearity: What are their effects on error and bias in regression?. Communications in Statistics - Simulation and Computation . 2019, 48:27-38. 10.1080/03610918.2017.1371750
10. Kalnins A: Multicollinearity: How common factors cause type 1 errors in multivariate regression. Strategic Management Journal. 2018, 39:2362-2385. 10.1002/smj.2783
11. Lindner T, Puck J, Verbeke A: Beyond addressing multicollinearity: Robust quantitative analysis and machine learning in international business research. Journal of International Business Studies. 2022, 53:1307-1314. 10.1057/s41267-022-00549-z
12. Araveeporn A, Wanitjirattikal P: Comparison of machine learning methods for binary classification of multicollinearity data. In Proceedings of the 2024 7th International Conference on Mathematics and Statistics. 2024, 53:44-49. 10.1145/3686592.3686600
13. Bi Y, Li C, Benezeth Y, Yang F: Impacts of multicollinearity on CAPT modalities: An heterogeneous machine learning framework for computer-assisted French phoneme pronunciation training. PLoS ONE. 2021, 16:e0257901. 10.1371/journal.pone.0257901
14. Rahman MNM, Nugroho RA, Faisal MR, Abadi F, Herteno R: Optimized multi correlation-based feature selection in software defect prediction. TELKOMNIKA (Telecommunication Computing Electronics and Control). 2024, 22:598-605. 10.12928/telkomnika.v22i3.25793
15. Haldar S, Capretz LF: Interpretable software defect prediction from project effort and static code metrics. Computers. 2024, 13:52. 10.3390/computers13020052
16. Anju AJ, Judith JE: Hybrid feature selection method for predicting software defect. Journal of Engineering and Applied Science. 2024, 71:124. 10.1186/s44147-024-00453-3
17. Fatwanto A, Nur Aslam M, Ndugi R, Syafrudin M: An investigation towards resampling techniques and classification algorithms on CM1 NASA PROMISE dataset for software defect prediction. Jurnal RESTI (Rekayasa Sistem dan Teknologi Informasi). 2024, 8:631-643.
18. Tong H, Liu B, Wang S: Benchmark data sets. Mendeley Data. (2017). https://data.mendeley.com/datasets/923xvkk5mm/1.