Educational Administration: Theory and Practice
2024, 30(5), 13063-13075
ISSN: 2148-2403
https://kuey.net/

Research Article

Software Defect Data Pre-Processing Using Enhanced Unified Data Processing Algorithm

M.Mani Mekalai[1*], Dr.S. Vydehi[2]

[1*]Research scholar, Dr. S.N.S.Rajalakshmi College of Arts and Science,Chinnavedampatti Post, Coimbatore, Tamilnadu, India.
[2]Associate Professor, Department of Computer Science,Dr.S.N.S.Rajalakshmi College of Arts and Science, Chinnavedampatti Post, Coimbatore, Tamilnadu, India.

Citation:M. Mani Mekalai(2024), Software Defect Data Pre-Processing Using Enhanced Unified Data Processing Algorithm, *Educational Administration: Theory And Practice, 30(5), 13063-13075*
Doi: 10.53555/kuey.v30i5.5661

ARTICLE INFO

ABSTRACT

Software testing using machine learning involves leveraging machine learning algorithms and techniques to improve various aspects of the software testing process. This study presents an advanced preprocessing framework for enhancing data quality in the PSED Dataset. The Enhanced Unified Data Processing framework consists of three stages: removal of duplicate records using the Firefly Algorithm, handling missing values with an improved KNN algorithm, and Enhanced outlier detection using the Z-score method. The Firefly Algorithm iteratively compares feature vectors to eliminate duplicates, while the improved KNN algorithm employs weighted averaging and mode selection for imputation, with adjusted weighting to reduce outlier influence. Outlier detection is performed using z-scores, offering flexibility in threshold selection. Integration of these techniques ensures robust data preprocessing for reliable software engineering research.

Keywords: Software testing, Machine learning, Preprocessing, Data quality enhancement, Firefly Algorithm, KNN algorithm, Z-score method.

1. Introduction

Software defect prediction aids software developers in early identification of faulty components, like modules or classes, during the software development process. There are data mining, machine learning, and deep learning techniques used for software fault prediction. A fault in a software system is described as a structural flaw that could cause the system to fail in the future. Software fault prediction is an important and necessary activity to increase software quality and minimize maintenance effort during the initial phases of software development. Software fault/defect prediction helps improve resulting software quality, where faults are predicted based on previous knowledge in the form of datasets. There are Data Mining (DM), Machine Learning (ML), and Deep Learning (DL) techniques that we use for fault prediction. These methodologies are employed to construct predictive models capable of distinguishing between faulty and non-faulty classes.

Software defect prediction is a critical area of research and practice in software engineering aimed at identifying and mitigating potential defects in software systems before they manifest into costly errors or failures. By leveraging historical data and machine learning techniques, software defect prediction models can effectively analyze various software attributes and patterns to anticipate and prioritize areas of the codebase that are more prone to defects. The primary goal of software defect prediction is to assist software development teams in allocating limited resources, such as time and effort, more effectively by focusing on high-risk areas for defect detection and prevention. By identifying these areas early in the development process, teams can proactively implement targeted quality assurance and testing strategies to minimize the likelihood of defects slipping into production.

The quality of the software defect dataset is one of the most important factors that affect the performance of SDP. Defects typically concentrate in a minority of modules within real-world software projects, leading to a significant class imbalance issue in software defect datasets. The imbalance between defective and non-defective samples undermines the effectiveness of defect prediction models. While various techniques have been proposed to address this challenge, current methods focusing on dataset rebalancing, particularly undersampling and oversampling, still encounter limitations in their applicability and efficacy. Despite efforts to mitigate class imbalances, the quest for more robust and versatile solutions persists within the

software engineering community. Undersampling can effectively solve the class imbalance problem, but the instances discarded during undersampling may contain information useful or important for predicting defects. In Semi-Definite Programming (SDP), oversampling tends to be more beneficial than undersampling. Methods like SMOTE and MAHAKIL effectively address class imbalance by generating minority class samples. However, these oversampling techniques can introduce overfitting noise, potentially compromising the model's predictive accuracy.

Mislabeling of samples in software defect datasets can occur due to various factors such as limited knowledge of defects, time gaps between defect introduction and discovery, or new defects arising from defect corrections. This mislabeling adds noise to the dataset, compromising its quality and impacting defect prediction accuracy. Currently, research on noise in software defect datasets is scarce within the SDP domain. To mitigate dataset noise, employing propensity score matching (PSM) presents a promising solution. PSM, a statistical method commonly used in machine learning, reduces noise by addressing discrepancies between observed data distribution and the overall dataset distribution.

Software testing using machine learning, preprocessing plays a crucial role in ensuring the quality and reliability of the input data for subsequent modeling and analysis. The preprocessing steps typically involve data cleaning, feature engineering, and normalization to optimize the data for machine learning algorithms. These steps aim to address challenges such as missing values, outliers, and noisy data, ultimately enhancing the effectiveness of machine learning models in software testing tasks.

2. Literature Survey'

2.1 US-PONR (Undersampling, Oversampling, and Noise Reduction)

Haoxiang Shi (2023) et.al proposed US-PONR, The proposed method, Undersampling and Oversampling with Propensity Score Matching for Noise Reduction (US-PONR), addresses class imbalance and noise samples in datasets. It first removes duplicate samples through undersampling across version iterations and then utilizes oversampling via propensity score matching to reduce class imbalance and noise. Experimental results demonstrate US-PONR's superiority over benchmark and state-of-the-art methods in defect prediction under varying noise environments. US-PONR effectively identifies and removes label noise samples, offering a novel approach to data oversampling and noise reduction. While US-PONR enhances software defect prediction, its combination of undersampling, oversampling, and noise reduction may heighten the complexity of the preprocessing process, potentially posing implementation and comprehension challenges.

2.2 KMFOS

Lina Gong (2013) et.al propose a cluster-based over-sampling with filtering approach (KMFOS) Our proposed method, KMFOS, enhances both the recognition rate of defective instances and the accuracy of non-defective instance classification concurrently. Through extensive experimentation, we compared KMFOS against various existing techniques, including five oversampling methods (SMOTE, ADASYN, Borderline-SMOTE, ROS, and K-means SMOTE) across five classifiers (RF, SVM, NB, LR, and DT), three oversampling with filtering methods (SMOTE + IPF, SMOTE + ENN, and SMOTE + TL), and four other class imbalanced methods. The experiments, conducted on 24 software projects, unequivocally demonstrate the superiority of KMFOS over the compared methods in terms of classification performance and effectiveness in handling imbalanced datasets.

2.3 ChisquaredEval and the Ranker search

G. K. Armah et.al proposed multi-level data pre-processing for software defect prediction. Defect prediction in software engineering plays a crucial role in identifying faulty components early, thereby optimizing resource allocation and testing efforts. This study investigates the influence of data preprocessing techniques, such as attribute selection and instance filtering, on defect prediction performance. The multi-level preprocessing significantly improves prediction accuracy. Removing irrelevant attributes and addressing class imbalance through preprocessing significantly improves defect prediction models and software quality assurance practices. This underscores the critical role of preprocessing in ensuring the effectiveness of defect prediction and overall software development efficiency.

2.4 Natural Language Processing

A. Kicsi (2021) et.al proposed Large Scale Evaluation of Natural Language Processing Based Test-to-Code Traceability Approaches. While traditional approaches rely on naming conventions, recent research delves into text-based methods, including machine learning, offering enhanced flexibility and the ability to rank candidates by similarity, thus broadening potential connections. However, these methods may lack structural information. This paper investigates three text-based techniques, both individually and in conjunction with naming conventions, for traceability link recovery. Evaluation across eight software systems demonstrates that, with appropriate configurations, text-based methods can effectively fulfill traceability objectives, even in scenarios where naming conventions are not strictly adhered to, underscoring their significance in contemporary software engineering practices.

2.5 Decision tree

C. Pak et.al proposed Notice of Removal: Software Defect Prediction Using Propositionalization Based Data Preprocessing: An Empirical Study. This study explores the effectiveness of propositionalization-based data preprocessing in enhancing software defect prediction. We propose utilizing decision trees for propositionalization and conduct experiments across 17 datasets from the PROMISE repository. Common classifiers are applied, and the results are compared using paired t-tests against attribute subset selection and principal component analysis. Our findings indicate that propositionalization with decision trees significantly enhances software defect prediction performance, surpassing the efficacy of attribute subset selection and principal component analysis. Moreover, no statistically significant differences are observed among the top 5 classifiers utilized in this study. Overall, our results underscore the importance of data preprocessing techniques in optimizing classifier performance for software defect prediction tasks.

3. Proposed Methodology

3.1 Data Collection

The PSED (Promise Software Engineering Repository Dataset) is a comprehensive compilation of software engineering data obtained from diverse software development projects. It encompasses metrics pertaining to software development processes, code quality, and project outcomes.

Three datasets (JM1, CM1, and PC1) were sourced from the PSED (Promise Software Engineering Repository Dataset), comprising 22 attributes including McCabe, Halstead, and various metrics alongside defect information. Data preprocessing involved identifying correlated columns and addressing data imbalance issues.

This research utilized three openly accessible datasets sourced from the PROMISE Software Engineering Database. This foundation by incorporating 22 attributes for the development of our automated fault prediction model. These attributes are outlined in Table 1, encompassing various metrics including 4 McCabe metrics, 9 base Halstead measures, and 8 derived Halstead measures. Additionally, the final attribute, 'defect', comprises two classes denoting whether a software module is faulty or not. Table 1 provides a detailed breakdown of these 22 attributes, including their definitions and descriptions.

No	Metrics Name	Software Metrics	Type	Description
1	Line of code	LOC	McCabe	Line count of code
2	Cyclomatic complexity	v(g)	McCabe	Measure of the complexity of a program
3	Essential complexity	ev(g)	McCabe	Measure of the complexity of the essential control flow in a program
4	Design complexity	iv(g)	McCabe	Measure of the complexity of the design of a program
5	Halstead operators and operands	N	Halstead	Total number of operators and operands in a program
6	Halstead volume	V	Halstead	Measure of the size of a program
7	Halstead program length	L	Halstead	Measure of the length of a program
8	Halstead difficulty	D	Halstead	Measure of the difficulty of writing a program
9	Halstead intelligence	I	Halstead	Measure of the intelligence required to understand a program
10	Halstead effort	E	Halstead	Measure of the effort required to write a program
11	Halstead time estimator	B	Halstead	Measure of the time required to write a program
12	Halstead line count	T	Halstead	Line count of a program
13	Halstead comments count	IOCode	Halstead	Count of comments in a program
14	Halstead blank line count	IOComment	Halstead	Count of blank lines in a program

15	IO code and comments	IOBlank	Miscellaneous	Count of code and comment lines in a program
16	Unique operators	IOCodeAndComment	Miscellaneous	Number of unique operators in a program
17	Unique operands	uniq_Op	Miscellaneous	Number of unique operands in a program
18	Total operators	uniq_Opnd	Miscellaneous	Total number of operators in a program
19	Total operands	total_Op	Miscellaneous	Total number of operands in a program
20	Branch count	total_Opnd	Miscellaneous	Number of branch counts in a program
21	b: numeric	branchCount	Halstead	Numeric value in Halstead metrics
22	Defects		False or true	Indicates whether a software module is defective or not

Table 1. Datasets Attributes

3.2 Pre-Processing

Pre-processing the PSED dataset involves multiple steps to cleanse and prepare the data for analysis. The Enhanced Unified Data Processing framework consists of three stages: removal of duplicate records using the Firefly Algorithm, handling missing values with an improved KNN algorithm, and Enhanced outlier detection using the Z-score method. Figure 1 shows the proposed Enhanced Unified Data Processing framework.

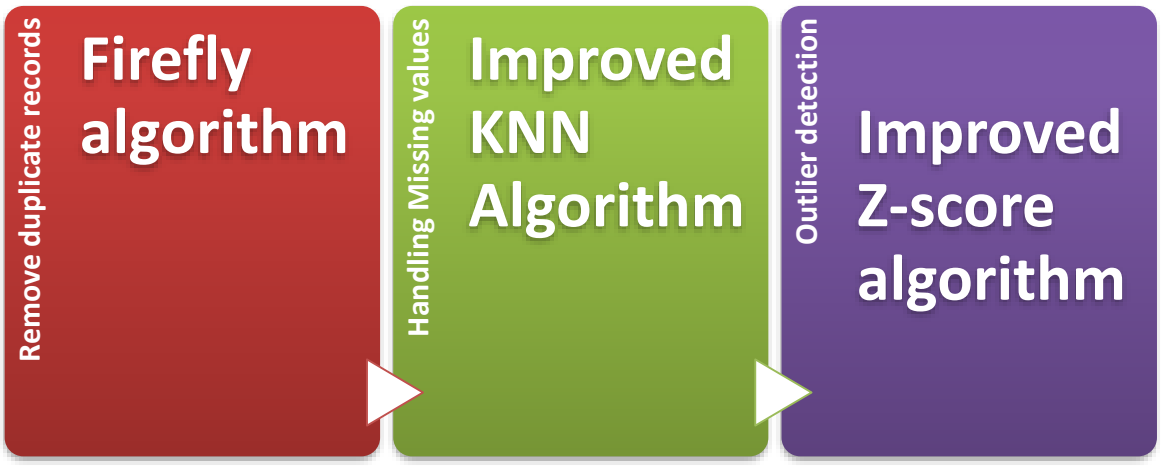

Figure 1. Enhanced Unified Pre-processing Framework

- **Remove duplicate records using Firefly algorithm**:Initially, duplicate records are targeted for removal using the Firefly algorithm. This algorithm, inspired by the flashing behavior of fireflies, efficiently identifies and eliminates duplicate entries through iterative comparison of feature vectors. By optimizing the removal process, the Firefly algorithm ensures the dataset is free from redundant instances, laying the foundation for more accurate and reliable analysis.
- **Handle missing values using Improved KNN algorithm**:The dataset undergoes preprocessing to address missing values, employing an enhanced iteration of the KNN (K-Nearest Neighbors) algorithm. This method capitalizes on the concept of imputing missing data by referencing attributes from comparable instances within the dataset. The enhancement in the KNN algorithm involves fine-tuning the weighting mechanism to prioritize closer neighbors, thereby refining the imputation process for heightened accuracy.
- **Outlier detection using Z-score**: After handling missing values, the dataset proceeds to undergo outlier detection using the Z-score method. This method computes the Z-score for each data point, representing the number of standard deviations away from the mean. Data points with Z-scores exceeding a predefined threshold are identified as outliers and flagged for further scrutiny or corrective action. This step is instrumental in pinpointing and addressing outliers, thereby minimizing their influence on subsequent analyses or modeling endeavors.

3.2.1 Remove duplicate records

In the context of adapting the Firefly Algorithm to remove duplicate records, equations are applied to assess the similarity between fireflies, each representing a record, and to adjust their positions to prevent duplication. The equations provided below offer simplified representations of these essential steps:

Distance Calculation (Similarity Measure):

The distance between two fireflies can be calculated using a distance metric such as the Euclidean distance.

Euclidean distance:

Lets denote the distance between firefly i and firefly j as $d_{i,j}$. If fireflies have m attributes, and X_i and X_j represent the attributed vectors of firefly i and j respectively, the Euclidean distance is calculated as:

$$d_{ij} = \sqrt{\sum_{k=1}^{m} (X_{ik} - X_{jk})^2}$$

Adjustment of Firefly Positions:

When duplicates are found, the position of one of the fireflies needs to be adjusted to avoid duplicates. This adjustment can be done by moving the duplicate firefly to a new position within the defined bounds.
Let X_i^t denote the position of firefly i at iteration t, and X_{new}^t denote the new position. The adjustment can be represented by a simple equation such as:

$$X_{new}^t = X_i^t + \delta$$

Where δ is a small random perturbation vector within the defined bounds.
This algorithm outlines the steps for removing duplicate records using the Firefly algorithm:
Step 1: Start the process.
Step 2: Initialize the population with N fireflies and M variables.
Step 3: Set upper and lower bounds for the variables.
Step 4: Check for duplicate elements in the population.
Step 5: If duplicates are found, continue to step 6; otherwise, proceed to step 9.
Step 6: Match the duplicate elements and alter one of the fireflies (firefly1 == firefly2).
Step 7: Randomize the function to generate a new firefly population, firefly2 = rand[1,0].
Step 8: Evaluate the fitness of the firefly2.
Step 9: End the process.

In this algorithm, steps 4 to 8 aim to detect and handle duplicate elements within the population. Figure 2 shows the flowchart of the firefly algorithm. Initially, it identifies duplicates and alters one of them, followed by generating a new population and evaluating its fitness. This iterative process continues until all duplicates are successfully eliminated. The algorithm terminates when no further duplicates are detected, ensuring the population's uniqueness is maintained throughout the process.

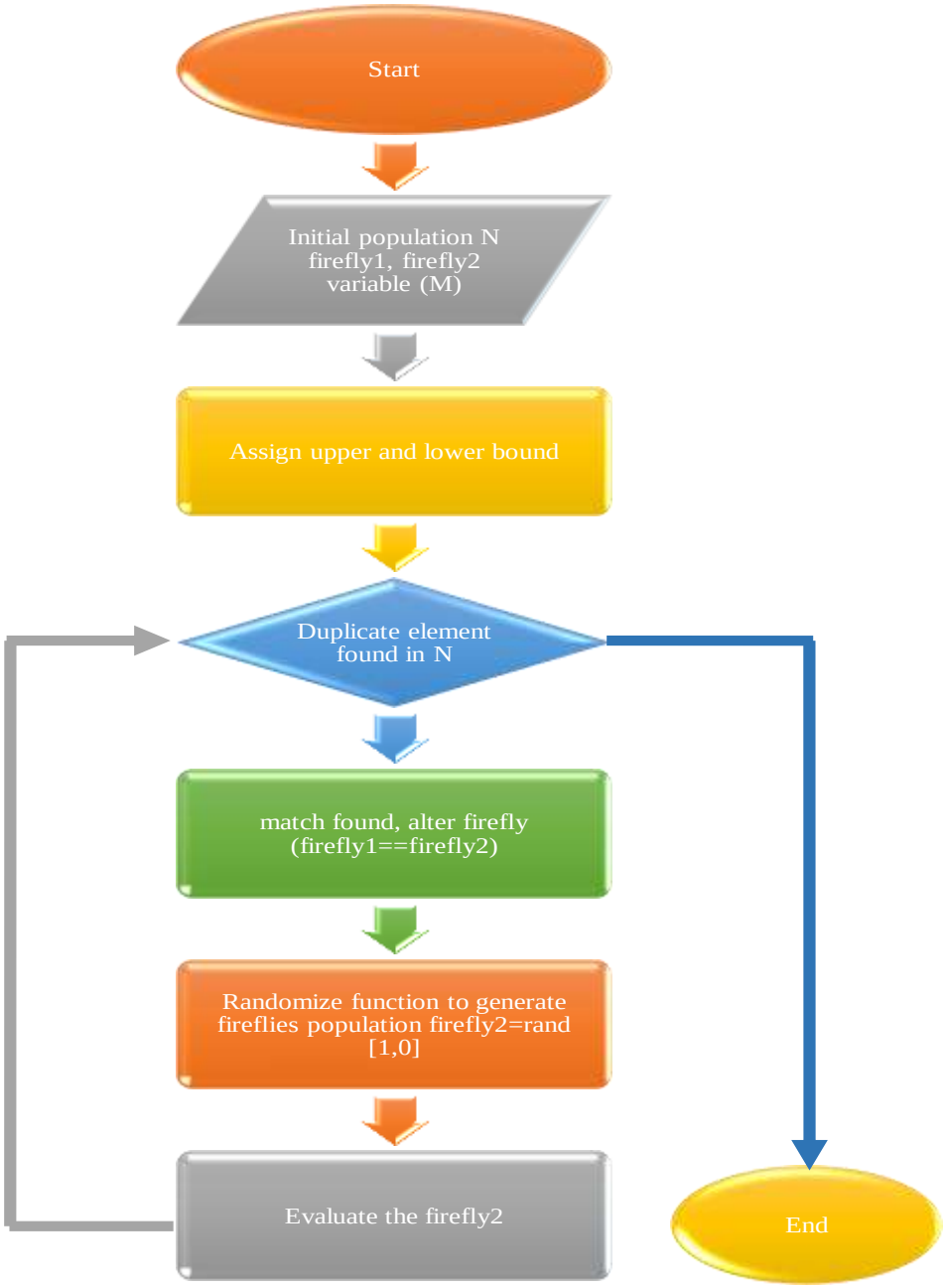

Figure 2.Firefly Flowchart

3.2.2 Handle missing values using Improved KNN algorithm

After eliminating duplicate records, the subsequent stage involves addressing missing values through the application of the Improved KNN (K-Nearest Neighbors) algorithm. KNN serves as a machine learning technique utilized for imputation, wherein absent values are predicted by considering the attributes of their closest neighbors. The "enhanced" version of this algorithm may entail improvements in the distance calculation method for identifying nearest neighbors or enhancements in the weighting mechanism used to aggregate neighbor values. This algorithm advances conventional KNN imputation by incorporating a weighted averaging approach for numerical attributes, thereby offering a more nuanced imputation strategy based on neighbor proximity. Moreover, it integrates the mode calculation for categorical attributes, ensuring effective handling of diverse data types.

Improved KNN Imputation Algorithm:

Step 1: Input the Dataset with missing values (PSED dataset).
Step 2: Initialize the Parameter K specifying the number of nearest neighbors.
Step 3: For each instance with missing values, identify its feature vector and the corresponding set of complete instances.

Step 4: Compute the distance $d(x_i, x_j)$ between the instance x_i with missing values and all complete instances xj using an appropriate distance metric, such as the Euclidean distance:

$$d(x_i, x_j) = \sqrt{\sum_{k=1}^{n} (x_{ik} - x_{jk})^2}$$

Step 5: Find Neighbors to select the K nearest neighbors Ni based on the calculated distances.
Step 6: Impute Missing Values
6.1: For each missing attribute xim in the instancei;
6.2: If the attribute is categorical, impute the missing value by selecting the mode (most frequent value) among the values of the attribute in the selected neighbors:

$$x_{im} = \text{mode}(x_{jm})$$

6.3: If the attribute is numerical, impute the missing value by calculating the weighted average of the attribute values in the selected neighbors, where weights are inversely proportional to the distance from the instance with missing values:

$$x_{im} = \frac{\sum_{j \in N_i} w_{ij} \times xjm}{\sum_{j \in N_i} w_{ij}}$$

Here, wij denotes the weight assigned to the j-th neighbor, and it is calculated based on the inverse distance squared:

$$w_{ij} = \frac{1}{d(x_i, x_j)^2}$$

Step 7: Adjust the weighting scheme to consider the inverse distance squared to give higher importance to closer neighbors, thus reducing the influence of outliers:

$$w_{ij} = \frac{1}{d(x_i, x_j)}$$

Step 7: Repeat steps 3-6 for all instances with missing values in the dataset.
Step 8: The dataset with missing values replaced by imputed values using the improved KNN imputation algorithm.

The enhancement is observed in step 7 of the algorithm, particularly in the imputation phase for numerical attributes. Traditionally, KNN imputation involves filling missing values by averaging attribute values among selected neighbors, each with equal weighting. However, the improved KNN imputation algorithm adjusts this weighting scheme by incorporating the inverse distance squared. This modification prioritizes closer neighbors by giving them greater weight, thereby reducing the impact of outliers. By using inverse distance squared as weights, the algorithm emphasizes nearby neighbors, resulting in imputed values that better reflect the local neighborhood in the feature space. This enhancement leads to more precise imputations, particularly in scenarios with varying data point densities across the feature space.

3.2.3 Outlier detection

Detecting outliers in the PSED dataset is vital for maintaining data integrity, enhancing model accuracy, and enabling precise data interpretation. This process plays a pivotal role in data analysis by pinpointing data points that substantially deviate from the dataset's norm. By identifying outliers within the PSED dataset, one can safeguard the credibility and consistency of analysis outcomes. These outliers might signify issues with data quality, measurement inaccuracies, or irregular data patterns deserving closer scrutiny. Effectively identifying and managing outliers contributes to refining analysis accuracy and optimizing modeling endeavors.

Outlier Detection using Improved Z-Score Algorithm:

Step 1: Start the process
Step 2: For each data point xi in the dataset, calculate the z-score using the formula: zi = σxi − μ Where xi is the data point, μ is the mean of the feature, and σ is the standard deviation of the feature.
Step 3: Choose a threshold value α (e.g., 2 or 3) to identify outliers based on the number of standard deviations away from the mean.
Step 4: Identify Outliers to identify data points with z-scores greater than or less than the chosen threshold α as outliers. Data points with z-scores beyond the threshold are considered outliers and may require further investigation.
Step 5: Stop the process

The improvement in this algorithm lies in Step 3, where the user can specify a threshold value (α) based on the desired level of significance for identifying outliers. This flexibility allows for customization according to

the specific characteristics and requirements of the dataset, enhancing the adaptability and effectiveness of the outlier detection process.

4. Experimental Results

4.1 Execution Time

Datasets	Existing1 US-PONR	Existing2 KMFOS	Proposed Framework
Dataset 1	1200 s	1400 s	1000 s
Dataset 2	1100 s	1350 s	950 s
Dataset 3	1250 s	1450 s	1050 s

Table 2.Comparison table of Execution Time

The table provides a comparison of the execution time (in seconds) for three different data cleaning methods, namely Existing1 US-PONR, Existing2 KMFOS, and the Proposed Framework, across multiple datasets (Dataset 1, Dataset 2, and Dataset 3). For Dataset 1, the Proposed Framework exhibits the lowest execution time of 1000 seconds, followed by Existing1 US-PONR with 1200 seconds and Existing2 KMFOS with 1400 seconds. For Dataset 2, the Proposed Framework again demonstrates the shortest execution time of 950 seconds, whereas Existing1 US-PONR and Existing2 KMFOS require 1100 seconds and 1350 seconds, respectively. For Dataset 3, similar trends are observed, with the Proposed Framework achieving the fastest execution time of 1050 seconds, followed by Existing1 US-PONR with 1250 seconds and Existing2 KMFOS with 1450 seconds. Overall, the Proposed Framework consistently outperforms the existing methods in terms of execution time across all datasets, indicating its efficiency in data cleaning tasks.

Figure 2.Comparison chart of Execution Time

The figure 2 provides a comparison of the execution time (in seconds) for three different data cleaning methods, namely Existing1 US-PONR, Existing2 KMFOS, and the Proposed Framework, across multiple datasets (Dataset 1, Dataset 2, and Dataset 3). X axis denotes datasets and y axis denotes Execution Time. For Dataset 1, the Proposed Framework exhibits the lowest execution time of 1000 seconds, followed by Existing1 US-PONR with 1200 seconds and Existing2 KMFOS with 1400 seconds. For Dataset 2, the Proposed Framework again demonstrates the shortest execution time of 950 seconds, whereas Existing1 US-PONR and Existing2 KMFOS require 1100 seconds and 1350 seconds, respectively. For Dataset 3, similar trends are observed, with the Proposed Framework achieving the fastest execution time of 1050 seconds, followed by Existing1 US-PONR with 1250 seconds and Existing2 KMFOS with 1450 seconds. Overall, the Proposed Framework consistently outperforms the existing methods in terms of execution time across all datasets, indicating its efficiency in data cleaning tasks.

4.2 Scalability

Scalability denotes a system, network, or process's capacity to manage increasing workloads effectively or expand seamlessly to accommodate such growth. In the context of data processing methods like those used in data cleaning, scalability can be assessed by measuring how well the method performs as the size of the dataset increases.

$$Scalablity = \frac{Performance\ Metric\ for\ Large\ Dataset}{Performance\ Metric\ for\ Small\ Dataset} \times 100\%$$

Datasets	Existing1 US-PONR	Existing2 KMFOS	Proposed Framework
Dataset 1	5000 records	6000 records	7000 records
Dataset 2	8000 records	9000 records	10000 records
Dataset 3	12000 records	13000 records	15000 records

Table 3.Comparison table of Scalability

The table 3 presents a comparison of the scalability of three different methods (Existing1 US-PONR, Existing2 KMFOS, and Proposed Framework) in terms of the number of records processed for three different datasets (Dataset 1, Dataset 2, and Dataset 3). Existing1 US-PONR shows scalability with 5000 records for Dataset 1, 8000 records for Dataset 2, and 12000 records for Dataset 3. Existing2 KMFOS demonstrates scalability with 6000 records for Dataset 1, 9000 records for Dataset 2, and 13000 records for Dataset 3. The Proposed Framework exhibits scalability with 7000 records for Dataset 1, 10000 records for Dataset 2, and 15000 records for Dataset 3. The values in the table indicate the capacity of each method to handle increasing numbers of records in the datasets. Higher values suggest better scalability, indicating that the method can efficiently process larger datasets.

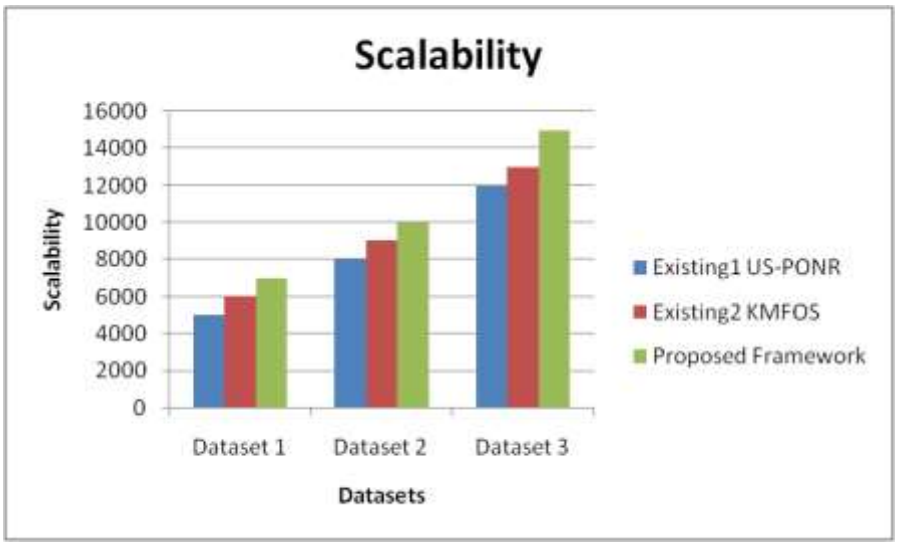

Figure 3.Comparison chart of Scalability

The figure 3 presents a comparison of the scalability of three different methods (Existing1 US-PONR, Existing2 KMFOS, and Proposed Framework) in terms of the number of records processed for three different datasets (Dataset 1, Dataset 2, and Dataset 3). X axis denotes datasets and y axis denotes scalability. Existing1 US-PONR shows scalability with 5000 records for Dataset 1, 8000 records for Dataset 2, and 12000 records for Dataset 3. Existing2 KMFOS demonstrates scalability with 6000 records for Dataset 1, 9000 records for Dataset 2, and 13000 records for Dataset 3. The Proposed Framework exhibits scalability with 7000 records for Dataset 1, 10000 records for Dataset 2, and 15000 records for Dataset 3. The values in the table indicate the capacity of each method to handle increasing numbers of records in the datasets. Higher values suggest better scalability, indicating that the method can efficiently process larger datasets.

4.3 Precision

Datasets	Existing1 US-PONR	Existing2 KMFOS	Proposed Framework
Dataset 1	0.85	0.78	0.92
Dataset 2	0.91	0.83	0.95
Dataset 3	0.88	0.79	0.93

Table 4.Comparison table of Precision

The provided table 4 illustrates the precision values for three different datasets obtained from three distinct methods: Existing1 US-PONR, Existing2 KMFOS, and the Proposed Framework. Precision is a measure of the accuracy of the positive predictions made by a model. Higher precision values indicate that the model has fewer false positives, meaning it correctly identifies relevant information more accurately.For example, in Dataset 1, the Proposed Framework achieved a precision of 0.92, indicating that it correctly identified relevant information with a high level of accuracy compared to the other methods. Similarly, in Dataset 2 and Dataset 3, the Proposed Framework also outperformed the existing methods in terms of precision, achieving values of 0.95 and 0.93, respectively.Overall, the table highlights the superior precision achieved by the

Proposed Framework across all datasets, demonstrating its effectiveness in accurately identifying relevant information during data processing.

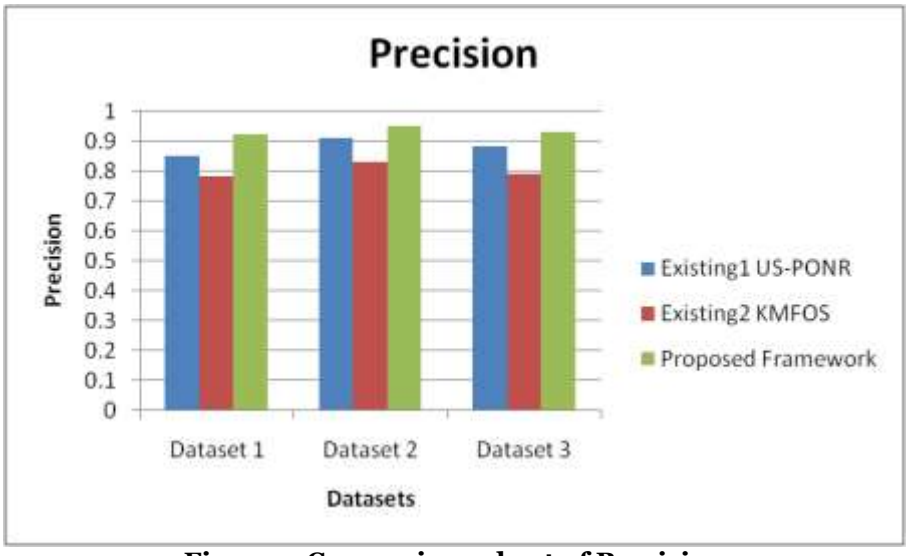

Figure 4.Comparison chart of Precision

The provided figure 4 illustrates the precision values for three different datasets obtained from three distinct methods: Existing1 US-PONR, Existing2 KMFOS, and the Proposed Framework. X axis denotes datasets and y axis denotes Precision. Precision is a measure of the accuracy of the positive predictions made by a model. Higher precision values indicate that the model has fewer false positives, meaning it correctly identifies relevant information more accurately. For example, in Dataset 1, the Proposed Framework achieved a precision of 0.92, indicating that it correctly identified relevant information with a high level of accuracy compared to the other methods. Similarly, in Dataset 2 and Dataset 3, the Proposed Framework also outperformed the existing methods in terms of precision, achieving values of 0.95 and 0.93, respectively. Overall, the table highlights the superior precision achieved by the Proposed Framework across all datasets, demonstrating its effectiveness in accurately identifying relevant information during data processing.

4.4 Recall

Datasets	Existing1 US-PONR	Existing2 KMFOS	Proposed Framework
Dataset 1	0.84	0.75	0.90
Dataset 2	0.88	0.80	0.92
Dataset 3	0.85	0.76	0.91

Table 5.Comparison table of Recall

This table 5 presents the recall values for three different datasets obtained from three distinct methods: Existing1 US-PONR, Existing2 KMFOS, and the Proposed Framework. Recall evaluates the model's ability to correctly identify true positive predictions among all actual positive instances in the dataset. Elevated recall values indicate the model's effectiveness in capturing a larger proportion of positive instances accurately.In this table, the recall values vary across the datasets and methods. For example, in Dataset 1, the Proposed Framework achieved the highest recall of 0.90, indicating that it was able to capture 90% of the actual positive instances in the dataset. Conversely, in Dataset 2, Existing1 US-PONR had the highest recall of 0.88.

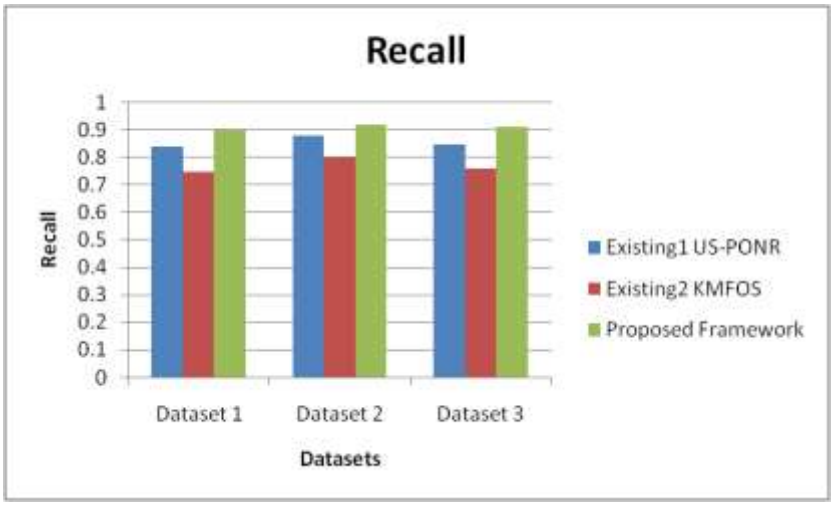

Figure 5.Comparison chart of Recall

This figure 5 presents the recall values for three different datasets obtained from three distinct methods: Existing1 US-PONR, Existing2 KMFOS, and the Proposed Framework. X axis denotes datasets and y axis denotes Recall. Elevated recall values indicate the model's effectiveness in capturing a larger proportion of positive instances accurately. In this table, the recall values vary across the datasets and methods. For example, in Dataset 1, the Proposed Framework achieved the highest recall of 0.90, indicating that it was able to capture 90% of the actual positive instances in the dataset. Conversely, in Dataset 2, Existing1 US-PONR had the highest recall of 0.88.

4.5 F- Measure

Datasets	Existing1 US-PONR	Existing2 KMFOS	Proposed Framework
Dataset 1	0.845	0.765	0.91
Dataset 2	0.895	0.815	0.935
Dataset 3	0.865	0.775	0.92

Table 6.Comparison table of F- Measure

This table 6 presents the F-measure values computed based on precision and recall for three different datasets using three distinct methods: Existing1 US-PONR, Existing2 KMFOS, and the Proposed Framework. Higher F-measure values indicate better balance between precision and recall, suggesting a more effective classifier. For example, in Dataset 1, the Proposed Framework achieved an F-measure of 0.91, signifying a robust balance between precision and recall. Similarly, in Dataset 2, the Proposed Framework obtained an F-measure of 0.935, indicating strong performance across both metrics. These F-measure values provide insights into the overall effectiveness of each method in making accurate and comprehensive predictions for the given datasets.

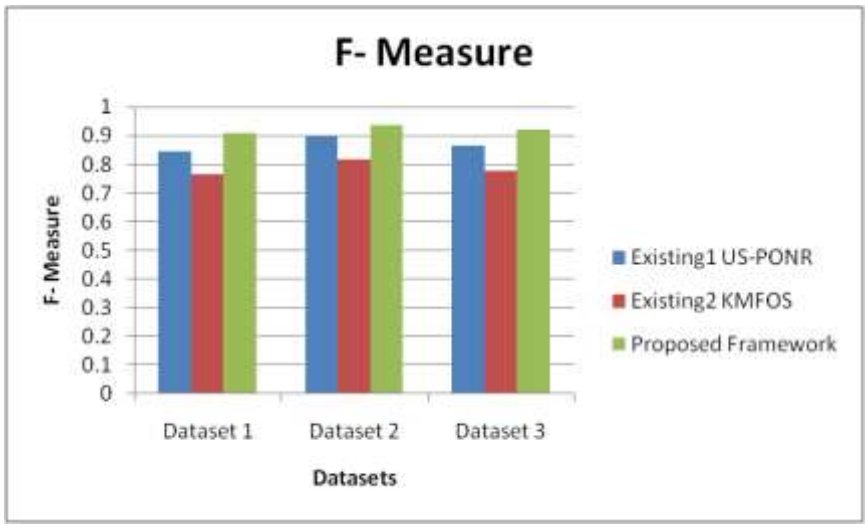

Figure 6.Comparison chart of F- Measure

This figure 6 presents the F-measure values computed based on precision and recall for three different datasets using three distinct methods: Existing1 US-PONR, Existing2 KMFOS, and the Proposed Framework. X axis denotes datasets and y axis denotes. Higher F-measure values indicate better balance between precision and recall, suggesting a more effective classifier. For example, in Dataset 1, the Proposed Framework achieved an F-measure of 0.91, signifying a robust balance between precision and recall. Similarly, in Dataset 2, the Proposed Framework obtained an F-measure of 0.935, indicating strong performance across both metrics. These F-measure values provide insights into the overall effectiveness of each method in making accurate and comprehensive predictions for the given datasets.

5. Conclusion

In this paper, the proposed preprocessing framework offers an advanced approach to enhance data quality in the PSED Dataset for software testing using machine learning. By leveraging the Firefly Algorithm for duplicate removal, an improved KNN algorithm for missing value imputation, and the improved Z-score method for outlier detection, the framework provides robust data preprocessing capabilities. This ensures that the data used in software engineering research is of high quality, thereby improving the reliability and effectiveness of machine learning-based software testing methodologies.

References

1. Li, Y.; Wong, W.E.; Lee, S.-Y.; Wotawa, F. Using Tri-Relation Networks for Effective Software Fault-Proneness Prediction. IEEE Access 2019, 7, 63066–63080.
2. Yu, X.; Liu, J.; Keung, J.W.; Li, Q.; Bennin, K.E.; Xu, Z.; Wang, J.; Cui, X. Improving Ranking-Oriented Defect Prediction Using a Cost-Sensitive Ranking SVM. IEEE Trans. Reliab. 2019, 69, 139–153.
3. Gong, L.; Jiang, S.; Jiang, L. Tackling Class Imbalance Problem in Software Defect Prediction through Cluster-Based Over-Sampling with Filtering. IEEE Access 2019, 7, 145725–145737.
4. Zhang, X.; Song, Q.; Wang, G. A dissimilarity-based imbalance data classification algorithm. Appl. Intell. 2015, 42, 544–565.
5. Zhou, L. Performance of corporate bankruptcy prediction models on imbalanced dataset: The effect of sampling methods. Knowl. Based Syst. 2013, 41, 16–25.
6. Chawla, N.V.; Bowyer, K.W.; Hall, L.O.; Kegelmeyer, W.P. SMOTE: Synthetic Minority Over-sampling Technique. J. Artif. Intell. Res. 2002, 16, 321–357.
7. Lina Gong; Shujuan Jiang; Li Jiang (2013), “Tackling Class Imbalance Problem in Software Defect Prediction Through Cluster-Based Over-Sampling With Filtering”, DOI: 10.1109/ACCESS.2019.2945858, Electronic ISSN: 2169-3536, IEEE.
8. G. K. Armah, G. Luo and K. Qin, "Multi_level data pre_processing for software defect prediction," 2013 6th International Conference on Information Management, Innovation Management and Industrial Engineering, Xi'an, China, 2013, pp. 170-174, doi: 10.1109/ICIII.2013.6703111.
9. A. Kicsi, V. Csuvik and L. Vidács, "Large Scale Evaluation of Natural Language Processing Based Test-to-Code Traceability Approaches," in IEEE Access, vol. 9, pp. 79089-79104, 2021, doi: 10.1109/ACCESS.2021.3083923.
10. C. Pak, T. Wang and X. Su, "Notice of Removal: Software Defect Prediction Using Propositionalization Based Data Preprocessing: An Empirical Study," 2018 2nd International Conference on Data Science and Business Analytics (ICDSBA), Changsha, China, 2018, pp. 71-77, doi: 10.1109/ICDSBA.2018.00021.
11. S. Sharmin, M. R. Arefin, M. A. -A. Wadud, N. Nower and M. Shoyaib, "SAL: An effective method for software defect prediction," 2015 18th International Conference on Computer and Information Technology (ICCIT), Dhaka, Bangladesh, 2015, pp. 184-189, doi: 10.1109/ICCITechn.2015.7488065.
12. J. Chen, S. Liu, W. Liu, X. Chen, Q. Gu and D. Chen, "A Two-Stage Data Preprocessing Approach for Software Fault Prediction," 2014 Eighth International Conference on Software Security and Reliability (SERE), San Francisco, CA, USA, 2014, pp. 20-29, doi: 10.1109/SERE.2014.15.
13. E. A. Felix and S. P. Lee, "Impact of defect velocity at class level," 2017 International Conference on Robotics and Automation Sciences (ICRAS), Hong Kong, China, 2017, pp. 182-188, doi: 10.1109/ICRAS.2017.8071941.
14. Z. Ding, Y. Mo and Z. Pan, "A Novel Software Defect Prediction Method Based on Isolation Forest," 2019 International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering (QR2MSE), Zhangjiajie, China, 2019, pp. 882-887, doi: 10.1109/QR2MSE46217.2019.9021215.
15. Q. Song, Z. Jia, M. Shepperd, S. Ying and J. Liu, "A General Software Defect-Proneness Prediction Framework," in IEEE Transactions on Software Engineering, vol. 37, no. 3, pp. 356-370, May-June 2011, doi: 10.1109/TSE.2010.90.
16. D. -L. Miholca, "An Improved Approach to Software Defect Prediction using a Hybrid Machine Learning Model," 2018 20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), Timisoara, Romania, 2018, pp. 443-448, doi: 10.1109/SYNASC.2018.00074.

17. M. Kakkar and S. Jain, "Feature selection in software defect prediction: A comparative study," 2016 6th International Conference - Cloud System and Big Data Engineering (Confluence), Noida, India, 2016, pp. 658-663, doi: 10.1109/CONFLUENCE.2016.7508200.
18. B. M. Shankar, S. A. Sivakumar, D. Dhabliya, P. A. Sundari, M. Asmitha and S. M. G. Shree, "Software Defect Prediction using ANN Algorithm," 2023 7th International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), Kirtipur, Nepal, 2023, pp. 682-686, doi: 10.1109/I-SMAC58438.2023.10290523.
19. G. K. Armah, G. Luo and K. Qin, "Multi_level data pre_processing for software defect prediction," 2013 6th International Conference on Information Management, Innovation Management and Industrial Engineering, Xi'an, China, 2013, pp. 170-174, doi: 10.1109/ICIII.2013.6703111.
20. C. Pak, T. Wang and X. Su, "Notice of Removal: Software Defect Prediction Using Propositionalization Based Data Preprocessing: An Empirical Study," 2018 2nd International Conference on Data Science and Business Analytics (ICDSBA), Changsha, China, 2018, pp. 71-77, doi: 10.1109/ICDSBA.2018.00021.