How Discretization Affects Software Defect Detection Tasks: An Experimental Study

Leonidas Akritidis
School of Science and Technology
International Hellenic University
Thermi, Thessaloniki, Greece
Email: lakritidis@ihu.gr

Vasileios Stergiopoulos
School of Science and Technology
International Hellenic University
Thermi, Thessaloniki, Greece
Email: vstergiopoulos@ihu.edu.gr

Panayiotis Bozanis
School of Science and Technology
International Hellenic University
Thermi, Thessaloniki, Greece
Email: pbozanis@ihu.gr

***Abstract*—Discretization is the process of transforming continuous variables into discrete ones. Although such a transformation introduces the risk of information loss, it is frequently proved beneficial in many machine learning tasks, due to its noise reduction and outlier absorption capabilities. Previous works in the relevant literature have demonstrated that discretization may have beneficial effects in classification performance. To our knowledge, this paper is the first to examine the magnitude of these effects in software defect detection tasks. In particular, we conduct a large scale experimental survey that involves 7 supervised and unsupervised discretization techniques, 26 datasets and 6 classifiers. The results indicate that the supervised methods are capable of capturing the distribution of the continuous variables, thus consistently improving detection performance. Interestingly, ChiMerge also improves balanced accuracy, exhibiting an inherent capability in alleviating the problem of class imbalance. On the other hand, the simpler unsupervised approaches lead to intolerable information losses that negatively affect the classification accuracy.**

***Index Terms*—discretization, software defect detection, classification, Bayesian Gaussian mixture**

I. Introduction

Software defect detection constitutes a fundamental process in the life cycle of software development. Its goal is to identify errors and problems in the source code of a program that may affect its functionality or performance. Detecting defects early during the development stage is essential to ensure reliability, robustness, and overall quality. For this reason, the problem of software defect detection has attracted the attention of numerous researchers.

Traditionally, defects are identified by human experts who manually test the underlying software for problems, malfunctions, bugs, logical errors, unexpected behavior under certain scenarios, etc. Nevertheless, the size and complexity of modern software systems rendered this approach not scalable. Today, modern solutions employ machine learning and artificial intelligence techniques that automatically examine specific features of the source code and indicate the existence of defects.

These techniques typically address the problem by considering it as a typical binary classification task. In this context, various predictors have been adopted in the relevant literature, including Decision Trees and Random Forests [1], [2], Neural Networks [3]–[5], Support Vector Machines [6], Long-Short Term Memory (LSTM) units [7], [8], and others.

Class imbalance and outliers constitute two of the most significant factors that negatively affect the performance of the aforementioned models. The former derives from the uneven distribution of the training examples in the two possible classes (i.e., healthy/problematic code units). Existing solutions to class imbalance include the popular oversampling techniques that synthesize artificial samples for the underrepresented class [9]–[11] and the undersampling approaches that limit the population of the majority samples either by direct sample removal or by instance replacement [12]–[14].

On the other hand, outlier detection is the process of identifying data points that deviate significantly from the rest of the data set. Outliers (also called anomalies) usually indicate errors in data collection or rare events that require further investigation. In most cases, they distort the ability of a model to effectively learn the underlying data distribution, downgrading its prediction performance. The relevant literature includes a wide variety of outlier detection techniques, including Local Outlier Factor (LOF) [15], Angle-based outlier detection [16], Isolation Forest [17], [18], and clustering-based outlier detection.

A broadly acceptable technique for mitigating the effects of outliers is the transformation of continuous data into discrete categorical variables. This technique, known as discretization, distributes the continuous values into a set of discrete bins (or buckets) according to some criteria. For example, in equal-width binning, a set of bins of equal widths is created, whereas in equal-frequency binning, each bin contains the same number of samples. Other approaches employ clustering algorithms (i.e., each cluster represents a bin) or statistical tests between the class label and adjacent intervals.

In general, discretization has several beneficial effects in classification tasks because of its ability to reduce noise and simplify complex data. However, it simultaneously introduces a risk of information loss and it may require careful tuning to avoid over-simplification. In this context, many discretization methods require the explicit setting of the number of bins and do not support mechanisms for automatically determining their population. This makes the risk of information loss even greater, because different features are usually subject to

979-8-3315-5636-5/25/$31.00 ©2025 IEEE

different distributions that require a different number of bins to be modeled effectively.

To our knowledge, it is the first study examining the effects of discretization in software defect detection applications. Specifically, we present an extensive experimental evaluation of 7 supervised and unsupervised discretization techniques on 26 software defect detection datasets. We employ 6 classification models and we measure their performance in terms of Accuracy, Balanced Accuracy and F1 score. We also introduce a simple heuristic method that automatically determines the most suitable number of bins per feature column. The proposed heuristic utilizes the Bayesian Gaussian mixture (BGM) model, aiming at capturing different modes in the value distribution of each feature column. The experiments prove that in several scenarios, this approach can indeed improve the ability of a discretizer to enhance detection performance.

The remainder of this paper is organized as follows. In Section II we briefly describe the most representative studies in the field of continuous data discretization. Section III presents the methodology of this study and several design details, while Section IV discusses the results of our experimental evaluation. Finally, the conclusions of this study are summarized in Section V, along with several interesting directions for future research.

II. Related Work

The beneficial effects of discretization in machine learning led to the introduction of numerous state-of-the-art techniques. In general, discretization methods can be classified into unsupervised and supervised approaches [19], [20].

The former do not rely on class labels, but instead focus on the structure and distribution of the data itself. They are proven to be useful when class labels are not available or are difficult to define. The simplest algorithms employ binning, a technique that distributes the continuous values into a set of buckets [19]. In this context, equal-width binning divides the range of attribute values into intervals of equal widths. On the other hand, equal frequency binning ensures that each interval contains the same number of data points [21]. Both methods are computationally efficient but may suffer from problems related to the sensitivity to the distribution of data. They also require a manual setting of the bins to be constructed, and this parameter usually requires careful tuning.

Clustering-based techniques also belong to the family of unsupervised methods. A common binning technique is to determine the bin contents by applying a clustering algorithm like k-Means. The values to be discretized are then grouped into similar clusters and can be uniformly represented by the cluster label. Similarly to the two aforementioned binning methods, this one also requires setting the number of bins in advance, or a stopping criterion that terminates the cluster merging or splitting process.

Although the unsupervised methods offer inherent simplicity, they frequently perform poorly, especially in cases where the data values are not distributed equally. Moreover, these methods are prone to information loss, significantly distorting the value distribution. To confront these issues, the supervised techniques use the class labels to guide the discretization process, with the aim of optimizing the performance of subsequent machine learning models.

The techniques that employ statistical tests between adjacent intervals and the provided class labels are among the most popular supervised discretization methods. For example, the ChiMerge algorithm was based on the idea that the relative class frequencies within a bin must be consistent; in the opposite case, the bin must be split [22]. The method also suggests that two adjacent intervals should not have similar relative class frequencies. The similarity of two intervals is measured by applying the χ^2 statistical test.

Chi2 improves ChiMerge by modifying the statistical significance level in each round, trying to merge more adjacent intervals, provided that the inconsistency criterion remains valid [23], [24]. Tay et al. introduced a modification to the Chi2 algorithm that takes into account the inaccuracy of the ChiMerge's merging criterion [25]. However, all the aforementioned methods examine each feature individually, failing to incorporate the possible dependencies between different features. In contrast, ConMerge performs the χ^2 test for two adjacent intervals, taking into account all continuous values [26].

Another family of algorithms computes the interval separation points by utilizing information-based or entropy-based measures [19], [27]. The method of [21] was based on the idea of minimizing the entropy of the bins and employed the Minimum Description Length (MDL) criterion to estimate the number of bins. On the other hand, Wong et al. proposed an entropy-based approach to transform continuous data into discrete tuples [28].

Other notable contributions to the area include the class-attribute interdependence maximization (CAIM) algorithm, a method that tries to discretize the continuous values into the smallest number of intervals, while simultaneously maximizing class interdependency [29]. In addition, Gupta et al. introduced a clustering approach, intending to capture the dependence among the different variables of an observation [30]. Two systematic reviews of the relevant literature on data discretization are the studies of Garcia et al. [31] and Ramírez-Gallego et al. [20].

III. Disretization in Software Defect Detection

We organize the presentation of our study in five subsections. The first four describe i) the benchmark datasets (Subsection III-A), ii) the 7 examined discretization methods (Subsection III-B), iii) the classification models that we trained with the aim of detecting software defects (Subsection III-C), and iv) the evaluation measures that we employed to evaluate the detection performance (Subsection III-D).

Furthermore, Subsection III-E presents the methodology that was adopted during this study. More specifically, we provide the necessary details regarding the involved data transformations and the design of the model training and testing procedures.

TABLE I: Software Defect Detection Datasets

#	**Dataset**	m	n	$\|Y\|$
1	AR1	121	29	2
2	AR4	107	29	2
3	CM1	498	21	2
4	KC1	2109	21	2
5	KC3	458	39	2
6	MC2	161	39	2
7	PC1	1109	21	2
8	PC3	1563	37	2
9	ANT-1.3	125	20	2
10	ANT-1.5	293	20	2
11	ANT-1.7	745	20	2
12	CAMEL-1.2	608	20	2
13	CAMEL-1.4	872	20	2
14	CAMEL-1.6	965	20	2
15	IVY-1.1	111	20	2
16	IVY-1.4	241	20	2
17	IVY-2.0	352	20	2
18	JEDIT-3.2	272	20	2
19	JEDIT-4.1	312	20	2
20	JEDIT-4.2	367	20	2
21	JEDIT-4.3	492	20	2
22	LOG4J-1.0	135	20	2
23	LOG4J-1.2	205	20	2
24	SYNAPSE-1.0	157	20	2
25	SYNAPSE-1.1	222	20	2
26	SYNAPSE-1.2	256	20	2

A. Datasets

The NASA MDP [32] and PROMISE[1] [33] collections are among the two most popular data sources for software defect detection tasks. They contain healthy and defective code examples for a wide variety of software projects and have been utilized in numerous relevant studies. Their columns contain numerical attributes that quantify software quality measures and attributes including the Halstead complexity measures, program line counts, bug lines and others.

Table I contains the details of the 26 benchmark datasets that were used for performance evaluation. The second and third columns show the number of data instances and features (columns), respectively. All datasets involve two classes (denoted in the last column), revealing the binary nature of the problem.

B. Discretization Methods

We examined the effects of discretization on software defect detection applications by implementing and testing a set of 7 unsupervised and supervised discretization techniques. The set includes:

- Equal-width binning: The simplest unsupervised discretization technique. First, it partitions the range of values into a fixed set of k bins of equal widths, and then it distributes the values into these bins.
- Equal-frequency binning: An equally simple unsupervised method that partitions the range of values into a set of k bins, so that each bin contains (roughly) the same number of values.
- k-Means binning: This approach groups similar continuous values into k clusters (bins) by executing the k-Means clustering algorithm.
- GMM binning: Another clustering-based discretizer that is similar to the previous one. Here the k clusters/intervals are determined by fitting a Gaussian Mixture Model.
- CAIM: The supervised method of Kurgan et al. [29] performs discretization by applying a class interdependency maximization criterion . Simultaneously, CAIM tries to identify the smallest number of intervals.
- ChiMerge: A popular discretization technique that merges adjacent intervals by estimating how similar their relevant class frequencies are [22]. This is achieved by computing their χ^2 similarity with the class labels; if they are found to have similar class frequencies, they are merged.
- BGM-ChiMerge: Here we propose an improvement to ChiMerge with the aim of limiting the maximum number of intervals to be created. More specifically, before executing ChiMerge, we first fit a Bayesian Gaussian Mixture (BGM) model on the column values in order to capture different modes in their distribution. Subsequently, the maximum number of bins is determined by the non-zero weights of the output BGM modes. We experimentally show that such an approach has beneficial effects in several scenarios.

Apart from these methods, we also consider the case where the original data is directly fed into a predictor without applying any discretization method. We call this case *None* to denote the absence of a discretizer, and we use it as a baseline for evaluating the ability of a discretizer to enhance (or degrade) the classification performance.

C. Classifiers

The task of detecting software defects requires a (binary) classification model. In this study, we employed a set of 6 such models in order to conduct a broad analysis with reliable results. This set includes:

- XGBoost: an ensemble classifier based on boosting with 100 estimators of maximum depth set equal to 6. The L2 regularization term was set equal to 1.
- Multilayer Perceptron (MLP): a typical feed-forward neural network with two hidden layers of 128 neurons each, utilizing the Rectified Linear Unit (ReLU) for activation. The model was trained with the Adam optimizer, with an L2 regularization parameter equal to 10^{-4}.
- C4.5: a decision tree classifier, with no limit on its maximum depth.
- Random Forest: a bagging classifier consisting of 50 decision trees, with no limit on their maximum depths. Each component tree used a random subset of $\sqrt{n}$ features to find the best split.
- Support Vector Machines (SVM): a standard Linear SVM classifier, using L2 regularization with $C = 1$. The model was trained by optimizing the squared hinge loss function.
- Logistic Regression: executed for a maximum of 300 iterations, with an L2 regularization term equal to 1.

[1]https://promisedata.org/

D. Evaluation Measures

The detection performance of the aforementioned methods was evaluated by using three metrics: Accuracy, Balanced Accuracy and F1 score. The first one is defined as the ratio of the correct predictions over the total number of predictions:

$$Acc = \frac{\text{Correct Predictions}}{\text{Total Predictions}}. \quad (1)$$

In general, the software defect detection datasets exhibit a high class imbalance ratio, caused by the fact that the defective samples are far fewer than healthy ones [11]. In such cases, it is widely accepted that Accuracy cannot effectively represent the defect detection performance, dictating the usage of other, more informative measures like Balanced Accuracy and the F1 score.

Balanced Accuracy is defined as the arithmetic mean of Sensitivity and Specificity:

$$B_{acc} = \frac{\text{Sensitivity} + \text{Specificity}}{2}. \quad (2)$$

In the context of software defect detection, Sensitivity and Specificity measure the ability of a model to classify a module as defective, or not. They are defined as follows:

$$\text{Sensitivity} = \text{Recall} = TP/(TP + FN), \quad (3)$$

$$\text{Specificity} = TN/(TN + FP). \quad (4)$$

Alternatively, the F1 score is defined as the harmonic mean of Precision P and Recall R:

$$F1 = \frac{2PR}{P + R}, \quad (5)$$

where:

$$\text{Precision} = P = TP/(TP + FP), \quad (6)$$

$$\text{Recall} = R = TP/(TP + FN). \quad (7)$$

E. Methodology

In order to assess the effects of discretization in software defect detection in a robust and reliable manner, we adopted a 3-stage strategy composed of the following steps:

1) Discretization: During this stage, the continuous columns of a dataset were transformed into ordinal integers by using the discretizers of Subsection III-B. These integers represented the intervals where the continuous values should be placed into.
2) One-Hot Encoding: The output of the previous stage was subsequently one-hot encoded, in order to become suitable to be fed into a classifier.
3) The one-hot representations of the discretized intervals were eventually fed to the classifiers of Subsection III-C for training.

To strengthen the reliability of the procedure, we have serialized these three stages by using a pipeline structure that automatically fed the output of a stage into the next one. In this way, we avoided the leakage of information from the training to the test set and we simplified the evaluation. The results were validated by using 5-fold cross-validation with stratification, which dictated an 80/20 split ratio of the original dataset into a training and a test set with similar class distributions.

The second stage ensured that all features were passed to the classifiers into an one-hot encoded representation. Consequently, no kind of normalization was performed. To alleviate the effects of randomness, all experiments were repeated three times with different initial seed values; namely, 0, 1 and 42. The results presented in the next section are the average values of the aforementioned measures computed over these three executions.

IV. Results and Discussion

In this section, we present the results of our experimental study. All experiments were carried out on a workstation with an Intel CoreI7 12700K, 32GB RAM and NVIDIA RTX3060 GPU, running Linux Mint 21.03. The implementation code is publicly available on Github to facilitate the reproduction of these experiments[2].

Figures 1 and 2 illustrate the performance of the 6 classifiers in the 26 benchmark datasets. Both figures display 9 diagrams placed on a 3×3 grid. Each column in the grid represents a different evaluation measure (i.e., Accuracy, Balanced Accuracy and F1 score), and each row depicts a different classifier. Moreover, each diagram displays 8 Whisker plots (also known as boxplots) that correspond to the 7 discretization methods plus the None case.

The boxplots are considered suitable for collectively illustrating measurements from multiple datasets and they are not parametric. Each box is created by considering 26 values (one per dataset) of one evaluation measure. The lower and upper edges of each box represent the 25th and 75th percentiles, respectively. That is, the median values of the lower and upper halves of the set of the 26 measurements. The continuous and dashed lines inside each box denote the median and mean values of the evaluation measure, respectively.

The results illustrated by these diagrams are rather clear: on average, the only two methods that consistently improve classification performance are ChiMerge and our proposed BGM-ChiM. Regarding the other discretizers, their performance against the None scenario (i.e., when no discretization was applied) was rather blurry. On several datasets, classifiers and evaluation measures they were found beneficial, whereas in some others, they significantly degraded the quality of detection. The general outcome of this study is that, on average, ChiMerge and BGM-Chi can better capture the underlying distribution of the continuous features.

More specifically, in the case of XGBoost, BGM-ChiM was on average the most effective method, since it outperformed all its adversaries in terms of Accuracy, Balanced Accuracy and F1 score. The original ChiMerge algorithm was also competitive and was ranked second in this test, whereas the None scenario was ranked third in terms of Balanced

[2]https://github.com/lakritidis/DeepCoreML

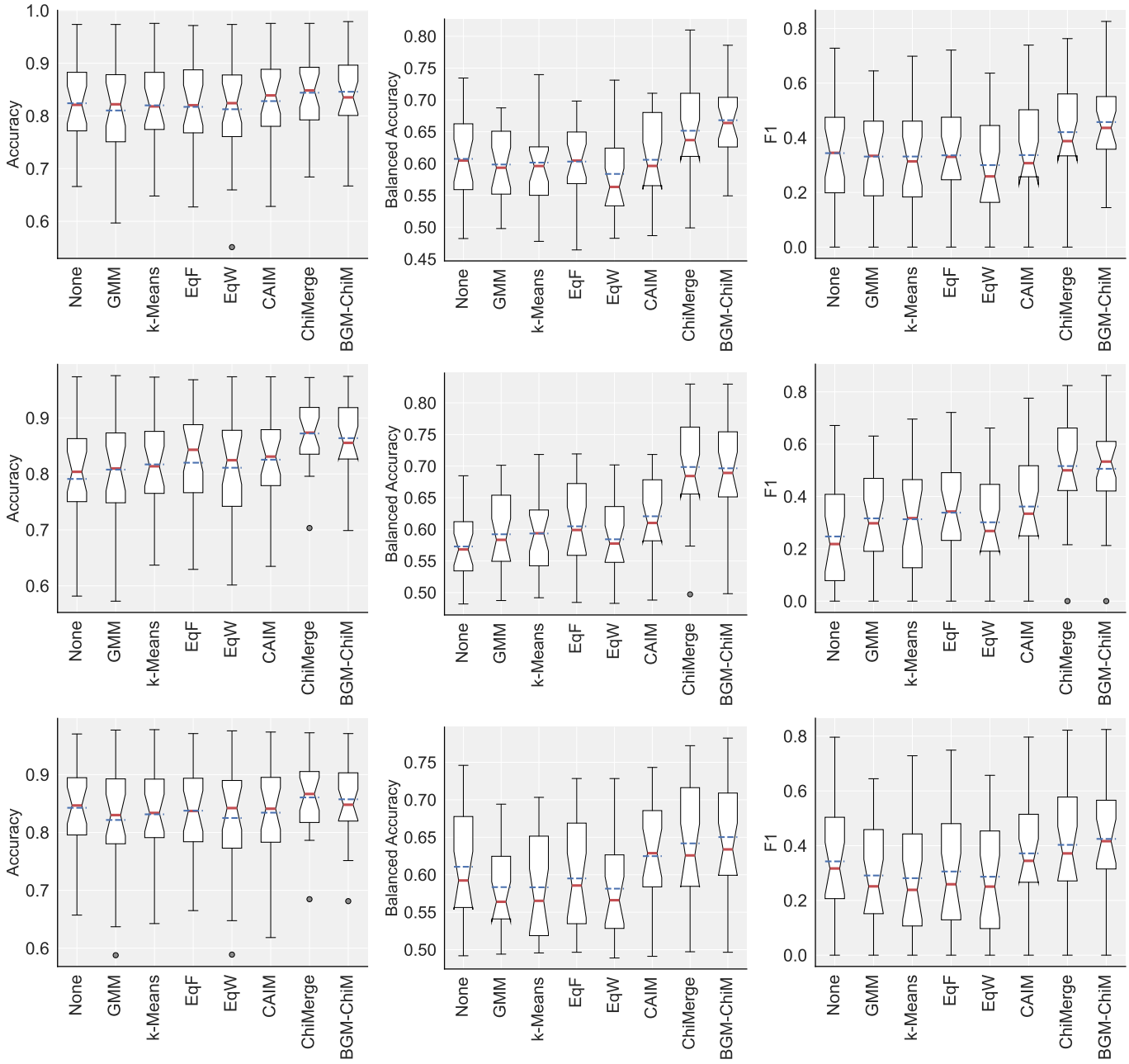

Fig. 1: Software defect detection performance of various discretization methods combined with the XGBoost (top), Multilayer Perceptron (middle), and Random Forest (bottom) classifiers in terms of Accuracy (left), Balanced Accuracy (Center) and F1 score (right).

Accuracy and F1 score. On the other hand, CAIM was found beneficial in terms of Accuracy, but, on average, it achieved low Balanced Accuracies and even lower F1 scores.

Interestingly, the performance of the Multilayer Percepton classifier was significantly improved by all discretization methods, including the unsupervised ones. Similarly to the previous case, BGM-ChiM was the strongest approach, achieving the highest mean Balanced Accuracy and F1 scores. ChiMerge and CAIM were ranked second and third, respectively. The unsupervised clustering-based method that employs k-Means for discretization was also quite effective. This top-3 ranking (i.e., BGM-ChiM, ChiMerge, CAIM) was preserved in the case where the Random Forest classifier was used for detecting software defects. The baseline method None occupied the fourth position in this experiment.

Regarding the other three classifiers of Figure 2, the situation in the top of the list was reversed, and ChiMerge outperformed all the other approaches. In the case of Linear SVM, the third place was occupied by None, demonstrating the weakness of the other discretizers to improve classification

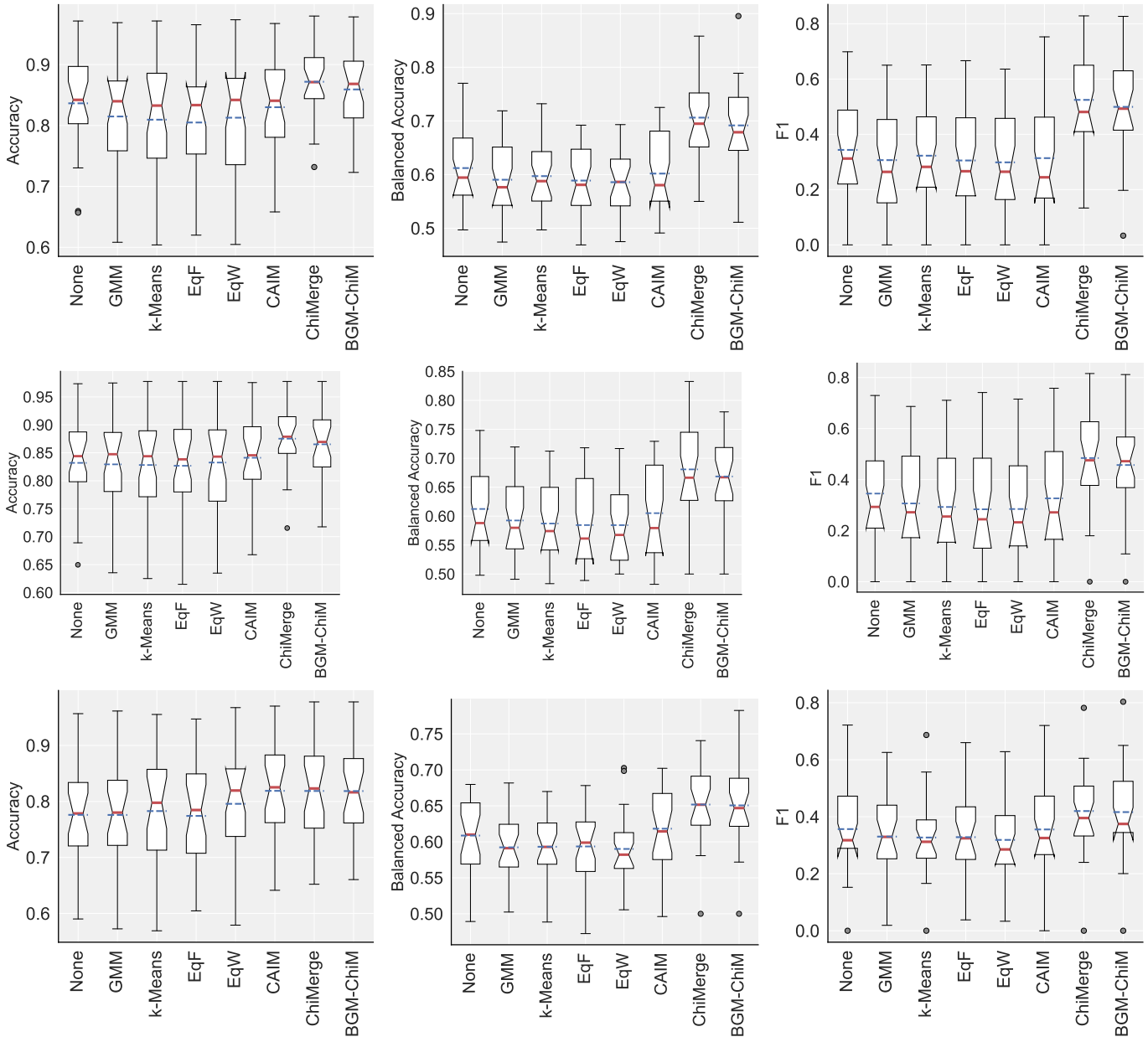

Fig. 2: Software defect detection performance of various discretization methods combined with Linear SVM (top), Logistic Regression (middle), and Decision Tree (bottom) classifiers in terms of Accuracy (left), Balanced Accuracy (Center) and F1 score (right).

performance. Regarding Logistic Regression, ChiMerge and BGM-ChiM were again the two highest performing methods. However, CAIM was also found beneficial compared to the "No Discretization" scenario and it was ranked third.

The only experiment where CAIM achieved the highest performance was the Accuracy of the Decision Tree classifier (left diagram in the last row of Fig. 2). However, in terms of Balanced Accuracy and F1 score, ChiMerge and BGM-ChiM were again the two most effective techniques, followed by CAIM.

The statistical significance of the aforementioned experiments was confirmed by running a series of Friedman post-hoc non-parametric tests. The results of these test are displayed in Table II and demonstrate that all p-values were far smaller than the threshold of $5 \cdot 10^{-3}$.

V. Conclusion and Future Work

In this study, we examined the effects of discretization in software defect detection tasks. We designed a thorough experimental procedure and verified the performance of 7 such techniques on 26 datasets with 6 classification models. In addition, we introduced an improved version of ChiMerge,

TABLE II: Post-hoc Statistical Significance Analysis with the Friedman Non-Parametric Test

Classifier	Accuracy	Bal. Accuracy	F1
XGBoost	$3.212 \cdot 10^{-10}$	$2.941 \cdot 10^{-15}$	$7.163 \cdot 10^{-15}$
MLP	$1.248 \cdot 10^{-15}$	$1.306 \cdot 10^{-18}$	$1.052 \cdot 10^{-21}$
Random Forest	$5.739 \cdot 10^{-12}$	$1.288 \cdot 10^{-13}$	$2.241 \cdot 10^{-14}$
SVM	$4.175 \cdot 10^{-20}$	$1.105 \cdot 10^{-17}$	$6.245 \cdot 10^{-18}$
C4.5	$4.862 \cdot 10^{-18}$	$2.869 \cdot 10^{-12}$	$8.424 \cdot 10^{-13}$
Log. Regression	$8.789 \cdot 10^{-18}$	$2.802 \cdot 10^{-17}$	$7.622 \cdot 10^{-18}$

called BGM-ChiM, with the aim of setting an upper limit to the number of the created intervals. Initially, BGM-ChiM fits a Bayesian Gaussian Mixture to the values to be discretized, and determines the upper limit of the intervals from the number of the non-zero mode weights.

The presented results indicated that the simple unsupervised techniques, such as equal-width and equal-frequency binning, are in general harmful for detection accuracy, as they miss critical information about the class-dependent distributions of the feature values. In contrast, several supervised discretization techniques, and especially BGM-ChiM and ChiMerge, may consistently improve the classification performance.

We intend to use the results of this study in devising oversampling solutions for alleviating the problem of class imbalance. More specifically, we investigate modifications to several generative models (like GANs and VAEs) to render them more effective in oversampling tasks with samples that contain only categorical variables.

REFERENCES

[1] N. Gayatri, S. Nickolas, A. Reddy, S. Reddy, and A. Nickolas, "Feature selection using Decision Tree induction in class level metrics dataset for software defect predictions," in *Proceedings of the World Congress on Engineering and Computer Science*, vol. 1, 2010, pp. 124–129.

[2] J. Wang, B. Shen, and Y. Chen, "Compressed C4.5 models for software defect prediction," in *Proceedings of the 12th International Conference on Quality Software*, 2012, pp. 13–16.

[3] R. Jindal, R. Malhotra, and A. Jain, "Software defect prediction using Neural Networks," in *Proceedings of the 3rd International Conference on Reliability, Infocom Technologies and Optimization*, 2014, pp. 1–6.

[4] V. Vashisht, M. Lal, G. Sureshchandar *et al.*, "A framework for software defect prediction using Neural Networks," *Journal of Software Engineering and Applications*, vol. 8, no. 8, pp. 384–394, 2015.

[5] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via Convolutional Neural Network," in *Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security*, 2017, pp. 318–328.

[6] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using Support Vector Machines," *Journal of Systems and Software*, vol. 81, no. 5, pp. 649–660, 2008.

[7] H. Liang, Y. Yu, L. Jiang, and Z. Xie, "Seml: A semantic LSTM model for software defect prediction," *IEEE Access*, vol. 7, pp. 83 812–83 824, 2019.

[8] H. S. Munir, S. Ren, M. Mustafa, C. N. Siddique, and S. Qayyum, "Attention based GRU-LSTM for software defect prediction," *Plos one*, vol. 16, no. 3, p. e0247444, 2021.

[9] A. Fernández, S. Garcia, F. Herrera, and N. V. Chawla, "SMOTE for learning from imbalanced data: progress and challenges, marking the 15-year anniversary," *Journal of Artificial Intelligence Research*, vol. 61, pp. 863–905, 2018.

[10] L. Akritidis, A. Fevgas, M. Alamaniotis, and P. Bozanis, "Conditional Data Synthesis with Deep Generative Models for Imbalanced Dataset Oversampling," in *Proceedings of the 35th IEEE Conference on Tools with Artificial Intelligence*, 2023, pp. 444–451.

[11] L. Akritidis and P. Bozanis, "A clustering-based resampling technique with cluster structure analysis for software defect detection in imbalanced datasets," *Information Sciences*, vol. 674, p. 120724, 2024.

[12] C. Seiffert, T. M. Khoshgoftaar, J. Van Hulse, and A. Napolitano, "RUSBoost: A hybrid approach to alleviating class imbalance," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, vol. 40, no. 1, pp. 185–197, 2009.

[13] S.-J. Yen and Y.-S. Lee, "Cluster-based under-sampling approaches for imbalanced data distributions," *Expert Systems with Applications*, vol. 36, no. 3, pp. 5718–5727, 2009.

[14] W.-C. Lin, C.-F. Tsai, Y.-H. Hu, and J.-S. Jhang, "Clustering-based undersampling in class-imbalanced data," *Information Sciences*, vol. 409, pp. 17–26, 2017.

[15] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "LOF: Identifying Density-based Local Outliers," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2000, pp. 93–104.

[16] H.-P. Kriegel, M. Schubert, and A. Zimek, "Angle-based outlier detection in high-dimensional data," in *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2008, pp. 444–452.

[17] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation Forest," in *Proceedings of the 8th IEEE International Conference on Data Mining*, 2008, pp. 413–422.

[18] S. Hariri, M. C. Kind, and R. J. Brunner, "Extended Isolation Forest," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 4, pp. 1479–1489, 2019.

[19] J. Dougherty, R. Kohavi, and M. Sahami, "Supervised and Unsupervised Discretization of Continuous Features," in *Proceedings of the 12th International Conference on Machine Learning*, 1995, pp. 194–202.

[20] S. Ramírez-Gallego, S. García, H. Mouriño-Talín, D. Martínez-Rego, V. Bolón-Canedo, A. Alonso-Betanzos, J. M. Benítez, and F. Herrera, "Data Discretization: Taxonomy and Big Data Challenge," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 6, no. 1, pp. 5–21, 2016.

[21] U. M. Fayyad and K. B. Irani, "Multi-interval discretization of continuous-valued attributes for classification learning," in *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, vol. 93, no. 2, 1993, pp. 1022–1029.

[22] R. Kerber, "ChiMerge: Discretization of Numeric Attributes," in *Proceedings of the 9th International Conference on Artificial intelligence*, 1992, pp. 123–128.

[23] H. Liu and R. Setiono, "Chi2: Feature selection and discretization of numeric attributes," in *Proceedings of 7th IEEE International Conference on Tools with Artificial Intelligence*, 1995, pp. 388–391.

[24] ——, "Feature Selection via Discretization," *IEEE Transactions on Knowledge and Data Engineering*, vol. 9, no. 4, pp. 642–645, 1997.

[25] F. E. Tay and L. Shen, "A Modified Chi2 Algorithm for Discretization," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 3, pp. 666–670, 2002.

[26] K. Wang and B. Liu, "Concurrent discretization of multiple attributes," in *Proceedings of the Pacific Rim International Conference on Artificial Intelligence*, 1998, pp. 250–259.

[27] X. Wu, "A Bayesian Discretizer for Real-Valued Attributes," *The Computer Journal*, vol. 39, no. 8, pp. 688–691, 1996.

[28] A. K. Wong and D. K. Chiu, "Synthesizing statistical knowledge from incomplete mixed-mode data," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, no. 6, pp. 796–805, 1987.

[29] L. A. Kurgan and K. J. Cios, "CAIM Discretization Algorithm," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 2, pp. 145–153, 2004.

[30] A. Gupta, K. G. Mehrotra, and C. Mohan, "A Clustering-Based Discretization for Supervised Learning," *Statistics & Probability Letters*, vol. 80, no. 9-10, pp. 816–824, 2010.

[31] S. Garcia, J. Luengo, J. A. Sáez, V. Lopez, and F. Herrera, "A survey of discretization techniques: Taxonomy and empirical analysis in supervised learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 4, pp. 734–750, 2012.

[32] M. Shepperd, D. Bowes, and T. Hall, "Researcher bias: The use of machine learning in software defect prediction," *IEEE Transactions on Software Engineering*, vol. 40, no. 6, pp. 603–616, 2014.

[33] J. Sayyad Shirabad and T. Menzies, "The PROMISE repository of software engineering databases," School of Information Technology and Engineering, University of Ottawa, Canada", 2005. [Online]. Available: http://promise.site.uottawa.ca/SERepository