


Article

# Class-Level Refactoring Prediction by Ensemble Learning with Various Feature Selection Techniques

Rasmita Panigrahi <sup>1</sup>, Sanjay Kumar Kuanar <sup>1</sup>, Sanjay Misra <sup>2,\*</sup> and Lov Kumar <sup>3</sup><sup>1</sup> Department of Computer Science and Engineering, GIET University, Gunupur 765022, Odisha, India<sup>2</sup> Department of Computer Science and Communication, Østfold University College, 1757 Halden, Norway<sup>3</sup> Department of Computer Science and Information System, BITS-Pilani-Hyderabad Campus, Secunderabad 500078, Telangana, India

\* Correspondence: sanjay.misra@hiof.no

**Abstract: Background:** Refactoring is changing a software system without affecting the software functionality. The current researchers aim i to identify the appropriate method(s) or class(s) that needs to be refactored in object-oriented software. Ensemble learning helps to reduce prediction errors by amalgamating different classifiers and their respective performances over the original feature data. Other motives are added in this paper regarding several ensemble learners, errors, sampling techniques, and feature selection techniques for refactoring prediction at the class level. **Objective:** This work aims to develop an ensemble-based refactoring prediction model with structural identification of source code metrics using different feature selection techniques and data sampling techniques to distribute the data uniformly. Our model finds the best classifier after achieving fewer errors during refactoring prediction at the class level. **Methodology:** At first, our proposed model extracts a total of 125 software metrics computed from object-oriented software systems processed for a robust multi-phased feature selection method encompassing Wilcoxon significant text, Pearson correlation test, and principal component analysis (PCA). The proposed multi-phased feature selection method retains the optimal features characterizing inheritance, size, coupling, cohesion, and complexity. After obtaining the optimal set of software metrics, a novel heterogeneous ensemble classifier is developed using techniques such as ANN-Gradient Descent, ANN-Levenberg Marquardt, ANN-GDX, ANN-Radial Basis Function; support vector machine with different kernel functions such as LSSVM-Linear, LSSVM-Polynomial, LSSVM-RBF, Decision Tree algorithm, Logistic Regression algorithm and extreme learning machine (ELM) model are used as the base classifier. In our paper, we have calculated four different errors i.e., Mean Absolute Error (MAE), Mean magnitude of Relative Error (MORE), Root Mean Square Error (RMSE), and Standard Error of Mean (SEM). **Result:** In our proposed model, the maximum voting ensemble (MVE) achieves better accuracy, recall, precision, and F-measure values (99.76, 99.93, 98.96, 98.44) as compared to the base trained ensemble (BTE) and it experiences less errors (MAE = 0.0057, MORE = 0.0701, RMSE = 0.0068, and SEM = 0.0107) during its implementation to develop the refactoring model. **Conclusions:** Our experimental result recommends that MVE with upsampling can be implemented to improve the performance of the refactoring prediction model at the class level. Furthermore, the performance of our model with different data sampling techniques and feature selection techniques has been shown in the form boxplot diagram of accuracy, F-measure, precision, recall, and area under the curve (AUC) parameters.



**Citation:** Panigrahi, R.; Kuanar, S.K.; Misra, S.; Kumar, L. Class-Level Refactoring Prediction by Ensemble Learning with Various Feature Selection Techniques. *Appl. Sci.* **2022**, *12*, 12217. <https://doi.org/10.3390/app122312217>

Academic Editor: Antonio Sarasa Cabezuelo

Received: 5 September 2022

Accepted: 21 November 2022

Published: 29 November 2022

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Copyright:** © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

**Keywords:** Software Refactoring prediction; software metrics; ensemble classifier; multi-phased feature extraction

## 1. Introduction

The last few years show the emergence of software as a vital technology to meet customer decision-making requirements for defense, business, industrial computing, communication, healthcare, security, and real-time control. Life in the twenty-first century

is inconceivable without a software computing environment. As a crucial component of contemporary human existence, software must be dependable, affordable, and adequate to meet essential requirements. On the one hand, the software industry strives to invent improved software solutions and hardware ecosystems to increase productivity while minimizing development costs. On the other hand, maintaining a better balance between development cost, productivity, and dependability has long been a challenge for the industry. Various paradigms, such as agile development, modular development, and Free and Open Source Software (FOSS) components, etc., have been extensively examined. Yet, the final software's reliability under real-world computing remains challenging [1]. For instance, FOSS and open-source software are frequently utilized to cut costs; nevertheless, the excessive usage of such components invites fault likelihood [1,2] after a period, which eventually renders software unreliable and insecure. Such incorrect software design methods necessitate testing or verification to verify that the created code remains intact and error-free for an extended period without malfunctions [2]. Developers perform manual testing over the software development life cycle (SDLC) to achieve it. Such paradigms help design object-oriented software with better maintainability, abstraction, flexibility, reusability, and fault resiliency. However, the large code size and complexity limit conventional manual testing or regression methods and demand automatic design optimization or recommendation models to support quality software development (QSD). Typically, in SDLC, change signifies the rule rather than the exception. Therefore, a key point for sustainable program development is to tackle software complexity, ambiguity, fault-prone less, etc., ensuring that the program retains a high quality and reliability, thus easing maintenance activities.

The literature demonstrates a correlation between bad code quality and poor productivity and susceptibility over an operating time [1,2]. Despite the importance of ensuring software quality, integrated development environments (IDEs) provide developers with little assistance in dealing with code constructions. Various tools for detecting design faults and identifying vulnerable components have been developed to improve quality-centric verification. The necessity for a robust tool or approach to evaluate the quality of the code cannot be overlooked in the real world, although in the majority of existing research, academia and industry may focus solely on bug estimation or reusability prediction. There will always be a need for a process that might evaluate a piece of code or software system's reliability, maintainability, fault tolerance, and readability. Recently, academics have proposed the "Refactoring" paradigm as an optimistic approach to achieving these objectives [3]. Altering a software code or making changes in the software code, known as refactoring, will not significantly change the external functionality while ensuring quality provision to the targeted software systems. Functionally, software or code-refactoring mechanisms state the process of restructuring the existing program by changing the factoring without imposing any external behavior or intention. It is primarily achieved to enhance the software system's non-functional component, making the software more readable, abstractive, robust, and maintainable, even at the cost of minimized complexity. Predominantly, refactoring is the sequential micro-refactoring paradigm in which micro changes are made in the software program or code to preserve the prime or expected behavior [3,4]. Though, it ensures the least or no change in intended external behavior. Identifying bugs, improper design, and vulnerability can be solved by refactoring to strengthen the quality of the software product through extended Complexity-free development and logic programming.

Moreover, it cleans up the code and eliminates defect or bug probabilities [2,3]. Peruma et al. [4] conducted quantitative and qualitative experiments on Stack Overflow refactoring discussions. Their findings reveal that a diverse group of developers use Stack Overflow for refactoring assistance across a wide range of technologies. Their observations show developers often need help refactoring in five areas: code optimization, tools and IDEs, architecture and design patterns, unit testing, and database. Our findings will help bridge the gap between traditional (or academic) features of refactoring and their real-world relevance, including improved tool support. Software often undergoes continuous changes

within timing constraints, leading developers to leave aside better programming practices to quickly convey the most suitable product [5–7]. However, it results in technical debt, drastically incorporating design problems impacting system maintenance and evolution.

There are different customized efforts made to enhance the quality of software, such as code-smell detection [8–10] towards program comprehension and change- and bug-proneness estimation over source code elements [11]. The analysis has played a vital role in different characteristics such as smells, bugs, etc. [6–10]. Various authors have made multiple efforts to solve refactoring problems; identifying an optimal signifier has become challenging. Kumar et al. [11] have found that software metrics are the most important in assisting the class-level refactoring proneness estimation among the major possible solutions. On the other hand, machine learning methods can learn over different structural constructs to decide to refactor a software chunk or class [12]. Some authors recommend using supervised concepts by exploiting predictors as independent variables and their relationship with the dependent variable (i.e., presence of a malicious entity, smell, or degree of smelliness in a source code) to assess the refactoring proneness of a software component. The role of code metrics is very important for refactoring prediction. Dallal et al. [13] applied size, cohesion, and coupling metrics for class-level refactoring analysis. Software code metrics, such as object-oriented metrics, Halstead metrics, etc., are available to identify the refactoring proneness of a code component [14,15]. Exploiting software code metrics and their structural entity information can significantly predict a software solution's refactoring probability [14].

### *Motivation*

The goal of refactoring is to improve the quality of existing code without changing its functionality. Refactoring applications lead to a waste of time and effort if the code is already generated. So, we aim to apply the refactoring on the code before going to the desk of developers. The selection of classes and methods needs refactoring before the development of the code becomes a difficult task for the researchers. In this work, we implement the refactoring concept after the design stage of the software during its life cycle. So, we generate the metrics values of each class after generating their corresponding class diagram by using a machine learning framework. Several works have been carried out for refactoring prediction at the class level and method level for identifying the refactoring candidates through a machine learning framework. However, refactoring prediction through heterogeneous ensemble classifiers at the class level with source code metrics as inputs into the machine learning framework is so far uncultivated.

The rest of the study paper is organized as follows. Section 2 presents the literature survey, Section 3 presents the research methodology, and the results are discussed in Section 4. Section 5 focuses on comparative analysis, whereas Section 6 focuses on the conclusion and any inferences. The references utilized in this study are listed at the end of the article.

## **2. Literature Survey**

This section highlights some of the most important current literature on refactoring in software systems. Bashir et al. [16] focused on designing a refactoring assessment method to help developers making strengthen software with reliability, maintainability, understandability, modifiability, and analyzability. The paper [17] on stability-oriented refactoring estimation used four attributes: abstraction, cohesion, coupling, and inheritance. They applied a hybrid Gravitational Search Algorithm and Artificial Bee Colony algorithm (GSA-ABC) to assess refactoring likelihood. Vimaladevi [17] used the stability of the code as the fitness function. Krishna et al. [18] evaluated whether refactoring could impact the quality of the code. The authors used object-oriented software metrics that classified software as to whether it needed refactoring. To assess the relationship between code refactoring and software maintainability, Kaur et al. [19] applied Junit and RefFinder. The authors recommended different software (code) metrics to examine maintainability and

concluded that code refactoring could help to archive low-maintenance software design. Malhotra et al. [20] used a design metrics suite to quantify internal quality attributes in order to determine the impact of refactoring maintainability. Some of the external quality aspects were the level of abstraction, understandability, modifiability, extensibility, and reusability. The authors used expert opinion to assess the impact of refactoring on maintainability. Desai et al. [21] focused on using the refactoring cost estimation (RCE) concept by applying different identifies such as misuse of classes, violation of the encapsulation principle, lack of inheritance concept, misuse of inheritance, misplaced polymorphism to assess refactoring proneness of a software. Lacerda et al. [22] observed the challenges and effectiveness of code smells and refactoring. Satwinder et al. [23] have carried out a literature survey on disclosing code smells. However, its efficiency in automatic refactoring assessment needs to be improved to ensure software reliability over an operating period [24]. Santos et al. [25] suggested refactoring significantly towards quality software design and cost-optimization; however, they found significant classical methods computationally overburdened, complex, and cost-consuming. Han et al. [26] proposed a software metrics model to assess the effect of refactoring candidates on their maintainability. Khlif et al. [27] stated that the major refactoring methods address merely the software's structural aspects.

On the contrary, the combined semantic aspects with the structural aspects can reduce the business process model's control-flow complexity in the Business Process Modeling Notation. The authors designed a refactoring method using the graph optimization concept with this motive. Arcelli et al. [28] proposed a performance-driven software architecture refactoring concept. Tao et al. [29] developed an automated refactoring idea for Java concurrent programs by synchronizing requirement analysis. The authors used split lock, split critical section, and convert features to perform the refactoring. Singh et al. [30] developed a refactoring-based pattern modeling concept to refine software design. The Rodin tool was applied to check the internal consistency concerning the desired functional behavior. Tarwani et al. [31] evaluated the refactoring sequence with a greedy algorithm, which selected the optimal solution at each stage to retrieve optimal global solutions and obtain varied sequences. These sequences were subsequently used in the source code to calculate the sum of software maintainability. The authors focused on identifying refactoring probability and the best refactoring solution to enhance maintainability [32,33]. Wang et al. [33] proposed a system-level refactoring concept that automatically identified a class's refactoring probability. The authors stated that "high cohesion and low coupling" features can be applied to assess code refactoring likelihood. They also found that identifying the "bad smells" caused due to coupling and cohesion can be separated without changing the code behavior. Alves et al. [34] proposed RefDistiller is a static analysis approach designed to aid with manual refactoring examination. RefDistiller combined two methods: a predefined template to identify potentially missing edits while manually refactoring.

In contrast, the second approach executes an automatic refactoring [35] engine to detect potentially improper extra edits. Using advanced graph analysis techniques, this study tries to automatically identify and restructure long method smells in Java code, overcoming the aforementioned challenges. A metamodel refactoring is an invertible semantics-preserving co-transformation that alters a metamodel as well as its models without losing data. Alton et al. [36] proposed a meta-modeling approach for pattern-based refactoring using design pattern and transformation rule specification to make the software more modular, modifiable, and reusable. This work [36] addresses a subproblem of metamodel refactoring: how to use the Coq Proof Assistant to show the validity of the refactoring of class diagrams without OCL constraints. Leandro et al. [37] considered identifying bad smells as a feature to perform a refactoring assessment, where the distance between system entities (attributes/methods) and class extracts with certain pre-conditioning was used to perform behavior-preserving refactoring analysis. The Leandro evaluates their proposed technique by automating 9885 transformations on four real open-source projects utilizing eight Eclipse IDE refactorings. In 20.41% and 14.11% of the investigated transformations, respectively, RefactoringMiner and RefDiff discover more refactorings. RefactoringMiner

and RefDiff do not identify the refactoring or classify it as another sort of refactoring in the remaining circumstances. They reported 34 issues to refactor detection tools, and engineers repaired 16 bugs, with three faults being replicated. Marcos et al. [38] proposed a semi-automatic tool for restructuring use cases called REUSE that discovered existing quality issues in use cases and suggested a prioritized set of candidate refactoring for functional analysts. Dig et al. [39] found that refactoring for asynchronous execution on mobile devices can improve responsiveness. However, the authors could not contribute to a robust solution to achieve the same. Lu et al. [40] discussed the four refactoring operators that retain the meaning of the code (i.e., Context Shift, Swap, Break and Merge). Aside from these, they have proposed three more OCL consistency metrics (Complexity, Cohesion, and Coupling) to measure sustainability and understandability. Lu et al. have created an automated search-based OCL constraint refactoring method (SBORA). Stolee et al. [41] found code smells that pointed to problems with web mashups that were written in the popular Yahoo! Pipes environment. Applying code-smell features, the authors proposed refactoring to lower the mashup programs' complexity and increase the corresponding abstraction. Like wise software metrics is important for code smell detection it can be used for refactoring prediction. Kumar et al. [42] empirically investigated the relationship between existing class level object-oriented metrics and a quality parameter, namely maintainability

Alomar et al. [43] suggested that code-complexity metrics design an intelligent class-level refactoring prediction model. This paper evaluates the ability to refactor documentation written in commit messages to predict the refactoring types performed at the commit level adequately. Shahidi et al. [35] used Weighted Naive Bayes with InfoGain heuristic to learn and predict refactoring probability in a real-world software system. Dallal et al. [44] designed a predictive approach to predict refactoring. The authors used the predictive model to analyze seven object-oriented metrics to perform refactoring classification to achieve it empirically. Similarly, Bavota et al. [2] examined the relationship between code smell and refactoring and found that a significantly large fraction of code having smell(s) required refactoring. As a contribution, the authors [2] stated that identifying smells can help assess a class's refactoring likelihood.

Similarly, Oscar et al. [45] designed refactoring impact prediction (RIPE) to predict the effect of refactoring on software quality. RPIE used 11 object-oriented metrics and 12 refactoring operations to perform refactoring recommendations. Lvers et al. [46] proposed a search based refactoring approach based on source code metrics. Nyamawe et al. [47] proposed a machine-learning approach that was trained using the history of previously applied refactorings detected using both traditional refactoring detectors and commit message analysis. The method employs a binary classifier to predict the need for refactoring and a multi-label classifier to recommend necessary refactorings. The authors applied univariate and multivariate logistic regression algorithms as a predictive model. Liu et al. [24] recommended that exploiting conceptual relationships, implementation similarity, structural relatedness, and inheritance hierarchies can help predict the software's refactoring. Different researchers have been experimenting with various AI-based techniques to recommend refactoring. Indeed, researchers have been experimenting with various AI-based techniques to recommend refactoring, such as pattern mining and search-based, etc. Aniche et al. [48] proposed a machine-learning approach that was trained using the history of previously applied refactorings detected using both traditional refactoring detectors and commit message analysis. The method employs a binary classifier to predict the need for refactoring and a multi-label classifier to recommend necessary refactorings. Kumar et al. [49] worked on the prediction of refactoring at the class level by a machine learning algorithm (LSSVM) with three different kernels and PCA as a feature selection technique. The data imbalance issue has been solved by the authors implementing the SMOTE technique. Authors have found that LSSVM with radial basis function (RBF) performs better than the other state-of-art method. Refactoring can also be predicted at the method level. Kumar et al. [50] applied 25 software metrics for refactoring prediction at the method level. Different algorithms such as Naïve Bayes, ANN with Gradient descent, Levenberg Marquardt logistic regression,

LogitBoost, etc., was used as a classifier. In the paper [42], a subset of object-oriented software metrics was considered to provide the necessary input data for the models for predicting maintainability using the Neuro-Genetic algorithm (a hybrid approach of neural network and genetic algorithm). This technique is used to estimate the maintainability of two different case studies, Quality Evaluation System (QUES) and User Interface System (UIS) (UIMS). This technique's performance parameters are measured using the Mean Absolute Error (MAE), Mean Absolute Relative Error (MARE), Root Mean Square Error (RMSE), and Standard Error of the Mean (SEM). According to the findings, the identified subset metrics demonstrated improved predictability with higher accuracy for defect prediction. The above literature notes that errors significantly impact on prediction capability of refactoring model. We have considered the errors i.e., Mean Absolute Error (MAE), Mean Absolute Relative Error (MARE), Root Mean Square Error (RMSE), and Standard Error of the Mean (SEM) for refactoring prediction by ensemble classifiers.

#### *Research Contribution*

The presented work in this paper presents various novel contributions. The work carried out in this paper is an extension of our previous work [51] In our earlier work, we developed a refactoring recommendation system for predicting the refactoring codes in terms of methods by using homogeneous classifiers, three data sampling techniques (SMOTE, SVSMOTE, and BLSMOTE) to solve the problem of data unbalancing, and the Wilcoxon rank sum test as a feature selection technique. In this paper, various efforts are performed by focusing on the appropriate classification and software metrics environments for refactoring prediction at the class level. Real-time software solutions can benefit from a revolutionary refactoring prediction model. Different value additions are incorporated to augment the proposed supervised machine learning-based refactoring assessment model's computational efficiency, such as pre-processing, Min-Max normalization, data sampling, feature extraction, and heterogeneous ensemble-assisted classification. As a solution, this research creates a new refactoring prediction model for real-time software solutions. Using code metrics of the software (program), we obtained a set of 125 code metrics, which were used for phase-wise feature selection using Wilcoxon significant text, Pearson correlation test, and PCA (principal component analysis) giving rise to an optimal set of metrics with different structural features signifying the software quality. It reduces the number of features based on respective significance. Differing from existing research, it minimizes computational efficiency to make the proposed system more agile and robust. The authors have applied the single machine learning algorithm to perform supervised classification for refactoring analysis. Realizing the performance diversity and disparity in prediction accuracy amongst the different classifiers is essential. This paper proposes a heterogeneous ensemble with high robustness structure to perform class-level refactoring prediction. We have also calculated the error percentages that occurred during the implementation of different ensemble classifiers to choose the best one for the class-level refactoring prediction model.

As a heterogeneous ensemble classifier, we have applied Artificial Neural Networks (ANN) with different learning methods, such as ANN-Gradient Descent, ANN-Levenberg Marquardt, ANN-GDX (adaptive weight learning), ANN-Radial Basis Function (RBF), support vector machine (SVM) with different kernel functions such as SVM-Linear, SVM-Polynomial, SVM-RBF, Decision Tree algorithm, Logistic Regression algorithm, extreme learning machine (ELM) model as the base classifier and LSSVM with different kernels such as linear, polynomial and RBF. Noticeably, our proposed heterogeneous ensemble structure's prime objective was to exploit each classifier's efficiency or vote towards class-level prediction and perform eventual classification as per MVE and BTE. The proposed refactoring analysis or prediction model was applied over PROMISE benchmark data, exploiting different key structural constructs. Our proposed model performed class-level analysis and (refactoring) identification. The overall proposed model was developed using R, MATLAB 2019b software tool, while performance analysis was performed in terms of

classification precision, accuracy, recall, and F-Measure. Results state that an MVE ensemble setup can produce the best feasible predictions for large-scale object-oriented software using an MVE ensemble. Additionally, this research found that the key structural metrics regarding coupling, cohesion, and complexity can be utilized as an identifier to classify each class for its refactoring proneness. It can be significant for developers or professionals to design cost-efficient and reliable software solutions. We have summarized our work in the following points:

(i) Several researchers have used the SMOTE data sampling techniques for handling data imbalance issues but SMOTE increases the training data size, including varieties of training data. In our paper, we have used Random Sampling, Down-sampling, and upsampling to handle data imbalance issue.

(ii) We have developed a heterogeneous ensemble model by collecting a set of classifiers of the different types built up on the same data;

(iii) The performance of each base learner was estimated in this study's investigation of relative performance using the suggested heterogeneous ensemble learner. We calculated metrics values for each base learner and ensemble classifier using statistical measures, including classification or prediction accuracy, precision, recall, and F-Measure;

(iv) We have experimented on four publicly available projects and chose the best ensemble method out of MVE and BTE;

(v) We have also computed different kinds of errors that can affect the performance of the refactoring model at the class level. Depending on errors, we are concluding the best ensemble classifier.

### 3. Research Methodology

Given the importance of a refactoring prediction scheme in this analysis, the predominant emphasis has been placed on exploiting key structural metrics or software code metrics and their relationship with refactoring proneness to perform class-level refactoring classification. In other words, this research first intends to identify the optimal set of code metrics and their association with refactoring proneness to perform each class classification as refactoring prone or non-refactoring. This can help developers design highly efficient, reliable, and cost-efficient software solutions. This research uses a multi-phased optimization paradigm where different enhancements, including data enhancement, feature enhancement, and classification enhancement, are performed. Though the principal goal is to design a robust and automatic refactoring prediction system, this research employs different techniques such as data sampling, data normalization, heterogeneous ensemble learning, and multi-phased feature selection to achieve an eventual goal. Thus, with the above-stated methods, a few research questions have been defined, which intend to assess whether the proposed method efficiently yields targeted refactoring prediction purposes. The identified research questions are given as follows:

**RQ1:** *Can software code metrics characterizing different traits, including inheritance, size, complexity, cohesion, coupling, etc., be an efficient identifier to classify a code as refactoring-prone or non-refactoring?*

**RQ2:** *Can a multi-phased feature selection method (Wilcoxon significant test, Pearson correlation test, and PCA) be used to help perform accurate software refactoring prediction?*

**RQ3:** *Can data sampling and normalization help alleviate the data imbalance issue in refactoring Prediction?*

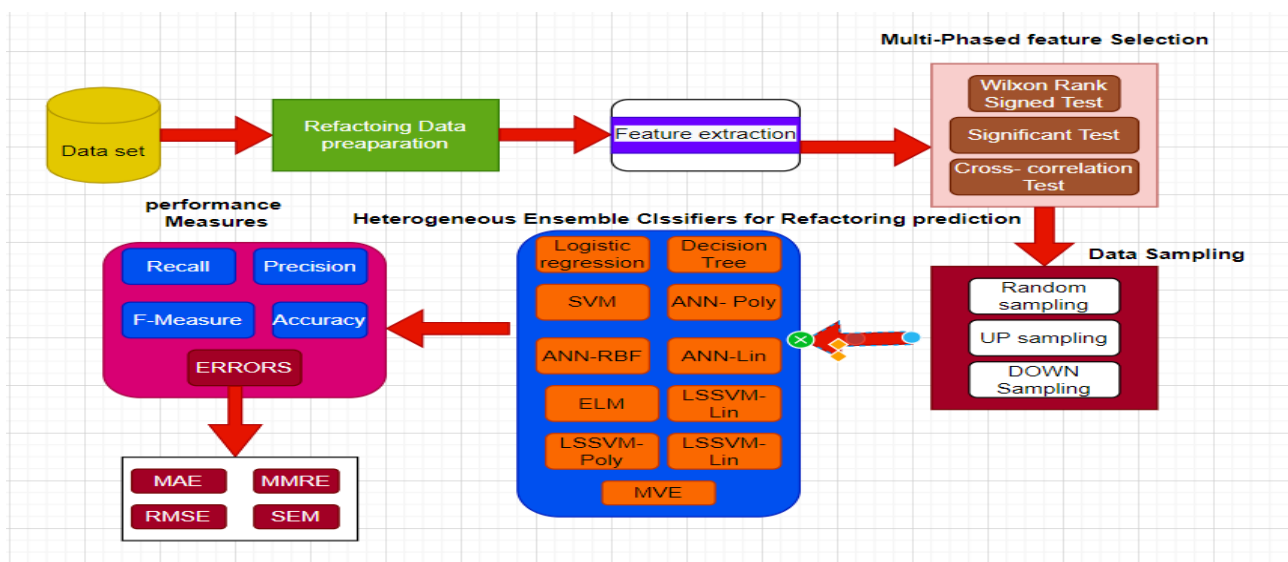
**RQ4:** *Can a heterogeneous ensemble structure with ANN classifiers, extreme learning methods, decision trees, and regression methods yield optimal software refactoring prediction for software quality optimization?*

**RQ5:** *Can a heterogeneous ensemble classifier with MVE ensemble structure yield more efficient software refactoring prediction than the BTE method?*

### 3.1. System Model

This section describes the multi-phased process for refactoring prediction at the class level and considers the overall research goal. The prime intention is to design a novel and robust automatic refactoring detection and classification system. This research incorporates a multi-phased optimization measure, focusing on enhancing all key functional components, including pre-processing and data augmentation, feature extraction, and feature selection, followed by ensemble-assisted classification. Distinct from conventional research, different authors have merely focused on applying classical classification algorithms such as Naïve Bayes, SVM, or ANN to perform classification over limited code metrics. We exploit the software program's major possible structural artifacts to assess its significance in characterizing refactoring proneness. Observing the significant existing literature and their inferences where it has been found that code bug-proneness or smell directly relates to refactoring probability, we consider this trait (i.e., code-smell) as a refactoring signifier to perform (refactoring) classification.

The phases of our proposed model are shown in Figure 1 and the model encompasses the following processes.



**Figure 1.** Proposed methodology for refactoring prediction at the class level.

- ❖ *Refactoring Data Preparation, Feature Extraction;*
- ❖ *Multi-phased Feature Selection;*
- ❖ *Sampling;*
- ❖ *Heterogeneous Ensemble Structure-Based Refactoring Prediction*

This research hypothesizes that structural traits such as (code) size, inheritance, complexity, cohesion, coupling, etc., directly relate to code smell and refactoring probability. We applied benchmark data for refactoring analysis from the tera-PROMISE repository. Noticeably, tera-PROMISE data encompasses benchmark software data with different refactoring quotients. Obtaining a total of 125 features are extracted, which are later processed for the Wilcoxon significance (Rank sum) test (WRS), the Pearson correlation test (PCT), and principal component analysis was used in a multi-phased feature selection process (PCA). We obtained the feature set or data and performed data augmentation with sampling and normalization techniques. At first, we performed data augmentation with three different sampling methods: random, upsampling, and downsampling. Thus, to obtain the set of samples or data, we performed Min-Max normalization, which maps all inputs or data in the range of 0–1. Thus, the proposed data augmentation model alleviates data imbalance, avoiding problems with local minima and convergence. Once the normalized data is obtained, we presented our heterogeneous ensemble-based categorization model,



which performs a class-level refactoring prediction. Distinct from conventional machine learning methods for refactoring analysis, in this research, classifiers from different types, such as decision trees, ANN methods, ELM methods, and SVM algorithms, were used as a novel heterogeneous ensemble model built around a base classifier. Employing the maximum voting ensemble (MVE) concept, our proposed method performed class-level refactoring assessment and classification that eventually exhibited better performance than the individual classifier as the standalone classification model. This section discusses the key techniques applied and the respective implementation details to design an automatic refactoring prediction model for software quality optimization.

### 3.1.1. Refactoring Data Preparation

Considering overall research intent, where the focus is placed on performing refactoring analysis and predicting whether a software and its components (say, class) might be refactored or not, we hypothesize code smells as the trait signifying refactoring probability in a software.

We considered the standard benchmark data from the tera-PROMISE repository (<https://github.com/dspinellis/awesome-msr>) (accessed on 20 March 2020). The exact data set is available in the URL <https://github.com/rasmitapanigrahi/data-set> (accessed on 20 March 2020) [52]. The four different (refactoring) datasets are collected with varied refactoring quotients to assess the proposed refactoring analysis model's efficiency shown in Table 1. We considered a set of four refactoring datasets obtained from the tera-PROMISE repository. These datasets typically encompass source code metrics of object-oriented software programs that employ techniques such as Helstied, Chidamber, Kemerer Java Virtual Machine (CKJM), etc. Since these data elements signify a software's structural traits, identifying associations and mapping their impact towards refactoring can help machine learning model(s) predict or classify a software component or code for its refactoring likelihood. We considered manually validated data for refactoring [14,15], which comprises almost 125 source code metrics obtained employing RefFinder-based extraction [53], followed by SourceMeter tool [54] (<https://www.sourcemeeter.com/>, (accessed on 19 December 2020)) based metrics extraction. Noticeably, this research's code metrics are related to different features, including coupling, cohesion, size, complexity, inter-component dependency, etc. We obtained a total of 125 such features to assess their respective efficacy toward refactoring prediction. The literature reveals that numerous code metrics can characterize the refactoring probability of a class, such as cohesion between individual clones, cyclomatic complexity, the total number of lines of code, the number of methods, the number of methods overridden, etc. Exploring this in-depth, it has been found that hundreds of software metrics characterizing software code quality can be used to assess refactoring analysis. Undeniably, several code metrics can characterize source code quality and eventually the refactoring probability; however, it is not inevitable that all metrics can have a similar significance to classify a class as refactoring prone or to be refactored. Considering this fact, we performed a statistical significance level estimation.

**Table 1.** Implemented Data set.

Projects	No. of Class	No. of Refactored Class	No. of Non-Refactored Class	Refactoring Class Percentage	Non-Refactoring Class Percentage
Antlr4	408	23	385	5.64	94.36
Junit	655	9	646	1.37	98.63
Mct	2028	15	2013	0.74	99.26
Titan	1158	13	1145	1.12	98.88

### 3.1.2. Multi-Phased Feature Selection

Among the important possible code metrics, a few can have vital significance in refactoring prediction. This research applies a multi-phased feature selection method using three statistical significance predictors.

- (a) *Wilcoxon Signed Rank Test (WRS)*;
- (b) *Significant Test*;
- (c) *Cross-Correlation Test*.

A brief of these algorithms is given as follows:

#### Wilcoxon Signed-Rank Test

As the name suggests, Wilcoxon Signed Rank Test (WRS) measures the correlation between multiple factors and how that affects classification accuracy. It is a non-parametric test that uses independent samples. With this goal in mind, we used this method to assess the correlation between multiple feature values and their related relevance for refactoring prediction. In other words, the input vectors are characterized by whether they influence the refactoring probability. The rank test shows how each metric is related to the refactoring probability. Functionally, the rank-sum test applies two distinct types of variables: independent and dependent variables. It assesses the correlation to determine the essential variable with a strong connection to the classification output. The independent variable was defined as user details (code metrics), while refactoring each class's probability was defined as the dependent variable. By implementing this method, we retrieve the  $p$ -value of each user concerning the refactoring probability and show how efficiently the refactoring likelihood is correlated to the code metrics or vice versa. WRS helped in handling the uncertainty in the code metrics and identified by removing unimportant elements and significant aspects.

#### Significant Test

Univariate Logistic regression (ULR) is a standard method for estimating the degree of correlation between independent and dependent variables, similarly to the rank test. Refactoring proneness can be predicted by analyzing the code metrics of each class. In the previous selection phase, we applied ULR to the chosen code metrics or features (i.e., rank-sum selected features). ULR evaluated the significance of the selected metrics to identify or characterize class-level refactoring. The independent variable (refactoring proneness) was used to estimate the dependent variable's variance (change %). (i.e., code-metrics).

$$\text{logit}[\pi(x)] = \alpha_0 + \alpha_1 X \quad (1)$$

In Equation (1),  $\text{logit } \pi(x)$  and  $X$  state the dependent (i.e., refactoring proneness) and the independent (code-metrics) variables, respectively. Here, it signifies the probability factor of the significance of each category in Equation (2).

$$\pi(x) = \frac{e^{\alpha_0 + \alpha_1 X}}{1 + e^{\alpha_0 + \alpha_1 X}} \quad (2)$$

In our proposed model, the  $p$ -value of the regression coefficient is used to figure out how important each code-metric is. Any metric with a  $p$ -value of more than 0.05 was thought to be important for predicting refactoring (proneness). Metrics with a  $p$ -value of less than 0.05 were taken out of the chosen final set of chosen features.

#### Cross-Correlation Test

In this method, we ran a cross-correlation test using the Pearson correlation estimation algorithm after extracting ULR-filtered code-structural metrics. Code-metrics with a  $p > 0.5$  correlation coefficient were used to classify refactoring proneness. After receiving code-

metrics, we normalized and augmented data to improve computation. The next section describes the pre-processing procedures.

### 3.1.3. Sampling

We have used a publicly accessible dataset from the tera-PROMISE repository for our investigations. A well-known repository for software engineering research datasets on code analysis, errors, effort, refactoring, and test creation is the tera-PROMISE website [52]. To make our studies simple to reproduce and use by other researchers for benchmarking and comparison, we have used a dataset from the tera-PROMISE repository. In this work, we are implementing data sampling techniques on four projects. The data set shown in Table 1 is highly imbalanced because the refactored number of samples is different from the non-refactored samples. The implemented projects are, i.e., Antlr4, Junit, Mct, and Titan, having 408, 655, 2028, and 1158 classes.

Considering data imbalance problems in data learning and classification, we used three data sampling methods: random sampling, upsampling, and downsampling. Our proposed method incorporates different sampling methods that ensure data augmentation so that it encompasses major possible features or information for further computation. Noticeably, the considered refactoring dataset contains many samples for each feature characterizing software structure and eventual refactoring probability. For illustration, let the total sample be 100. Then, in the random sampling method, five data samples are selected randomly (assuming that each element would have an equal probability of being identified as a refactoring class) from the total samples. In contrast, merely five samples from ninety-five non-refactoring samples are considered in downsampling.

On the contrary, to perform upsampling, a small data element (say five) is extrapolated or augmented into the large scale (say, ninety-five to constitute one-hundred samples) non-refactoring data samples. Thus, the inclusion of randomly selected  $X_{\text{Rand}}$ , upsampled ( $X_{\text{Upsampled}}$ ), and downsampled data ( $X_{\text{Downsampled}}$ ) along with the original data ( $X_{\text{Original}}$ ). They constituted augmented data for further computation shown in Equation (3).

$$\text{Feat}_{\text{Composite}} = \left[ X_{\text{Original}}, X_{\text{Rand}}, X_{\text{Upsampled}}, X_{\text{Downsampled}} \right] \quad (3)$$

### Data Normalization

In enormous classification or prediction systems, extensive features-based models, data imbalance impacts system performance. The considered data set may comprise minor features indicating refactoring probability, which might generate classification bias and impair overall prediction accuracy; for example, let the dataset have one-hundred elements, out of which merely between four and five data elements signify the refactoring tract. Therefore, learning the data might cause or influence overall performance, especially classification accuracy. A dual-phase data augmentation method was incorporated into the proposed model by executing data normalization followed by data re(sampling) to alleviate these problems. We applied the Min-Max normalization method, while for data resampling, three methods, random sampling, downsampling, and upsampling, have been applied. A brief of these methods is given as follows:

Data items can be of varying sizes and ranges; hence, computing over such unstructured and broad-scaled data can cause learning models to converge prematurely. As a result, it can impact the overall precision of the suggested model. We have normalized the data using the Min-Max technique in light of this data imbalance issue. Our suggested Min-Max normalization model normalizes input data from a functional standpoint in the 0 to 1 range. Our suggested normalization approach maps and linearly transforms the input data items inside the interval [0, 1]. Each data element  $x_i$  of the software component  $X$  is functionally

mapped to its corresponding normalized value  $x_i$  in the range  $[0, 1]$ . Mathematically, we estimated the normalized value(s) of the input data  $x_i$  using the formula of Equation (4).

$$\text{Norm}(x_i) = x'_i = \frac{x_i - \min(X)}{\max(X) - \min(X)} \quad (4)$$

In Equation (4), the  $\min(X)$  and  $\max(X)$  data elements show the lowest and highest values of  $X$ , respectively.

### 3.1.4. Heterogeneous Ensemble Structure-Based Class-Level Refactoring Prediction

In most previous works implementing machine learning approaches for estimating refactoring-prone code-refactoring probabilities, the authors used various machine learning approaches to apply the algorithms as standalone classifiers. Diverse classification performance is exemplified by distinct methods, producing distinct results for the same dataset. This study creates a highly robust ensemble learning model by combining classifiers from several categories, including pattern mining SVM, decision tree, neural network, and extreme learning machine. The strategic combination of various machine learning methods provides a heterogeneous ensemble model for class-level refactoring prediction. The ensuing sections summarize the many machine learning algorithms and classifiers that have been implemented. Noticeably, the proposed refactoring prediction model performs learning and classification over 24 different features or code metrics for four different samples obtained as  $X_{\text{Rand}}$ ,  $X_{\text{Upsampled}}$ ,  $X_{\text{Downsampled}}$ , and  $X_{\text{Original}}$ . Obtaining a total of 96 feature sets, we performed learning using various machine learning algorithms and ensemble models. We used many machine learning methods as basic classifiers, including decision trees and modified K-NN classifier, Logistic regression, SVM-Linear, SVM-Polynomial, SVM-RBF, LSSVM, ANN-GD, ANN-GDX, ANN-LM, ANN-RBF, and ELM with different kernels. A detailed discussion of the proposed machine learning methods is given in the subsequent section.

#### Logistic Regression (LR)

LR is one of the most common ways to use regression to group data. It uses the idea of regression to classify a dependent variable based on more than one input (say, code metrics). For example, in our proposed refactoring prediction problem, the dependent variable (software code or class) can be either refactoring prone or not refactoring. In logistic regression, the base classifier looks at how likely each class needs refactoring based on the relationship between the code metrics for each class. Mathematically, logistic regression is presented as Equation (5).

$$\text{logit}[\pi(x)] = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_m X_m \quad (5)$$

In Equation (5), the left-hand component  $\text{logit}[\pi(x)]$  represents the dependent variable, whereas  $X_i$  represents the independent variable  $[0-m]$ . The logistic regression approach uses the linear regression idea to convert dichotomous outputs to logit function values ranging from 0 to  $m$  denote the total number of independent variables in Equation (5) (here, the code metrics).

The second field indicates the refactoring propensity of the class during validation. Consequently, the dependent variable or class-level ( $x$ ) predicted by LR is denoted by Equation (6).

$$\pi(x) = \frac{e^{\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_m X_m}}{1 + e^{\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_m X_m}} \quad (6)$$

#### Decision Tree (DT.) Algorithm

Pattern classification has traditionally been performed using the Decision Tree method [49,50], although its efficiency has seen significant improvements throughout time, allowing it to attain higher accuracy. For example, DT C4.5 and DT C5.0, which are mostly used for data

mining and categorization, have evolved. DT’s C5.0 method was used as a basis classifier to perform class-level predictions on the input data. Noticeably, the C5.0 DT algorithm categorized or labeled each class of the investigated software as either refactor-prone or non-refactoring. The input data metrics are divided into numerous branches at each node of the DT, starting at the root node and using an association rule in between the split criteria. Refactoring-prone and non-refactoring classes were defined using the Information Gain Ratio (IGR) in the C5.0 method used in this study. Alsolai et al. [55] conducted an empirical evaluation to find the impact of feature selection techniques, ensemble models, and sampling techniques implemented on seven data sets for predicting change-proneness using different source code metrics. However, Muruges et al. [56] discussed automated software requirements using machine learning algorithms.

### Support Vector Machine (SVM)

SVM is one of the most used supervised machine-learning approaches for pattern recognition. A binary linear classifier that learns from data that is not probabilistic. SVM uses the structural risk reduction paradigm to reduce generalization errors on unseen instances. To achieve the boundary’s value, also referred to as a hyper-plane, in between the two classes, this method uses support vectors to represent a portion of the training set. The following function is used in pattern-based classification using SVM-based prediction in Equation (7).

$$Y' = w * \phi(x) + b \tag{7}$$

Collecting the correct weights factor  $w$  and bias component  $b$  values to complete the non-linear transformation is essential. We obtain  $Y'$  in Equation (7) by iteratively reducing the regression risk. The risk of regression is expressed numerically in Equation (8).

$$R_{reg}(Y') = C * \sum_{i=0}^l \gamma(Y'_i - Y_i) + \frac{1}{2} * \|w\|^2 \tag{8}$$

These two parameters represent the penalty and cost functions,  $C$  and, are represented in Equation (8). Weights are estimated using the method in Equation (9).

$$w = \sum_{j=1}^l (\alpha_j - \alpha_j^*) \phi(x_j) \tag{9}$$

In Equation (9),  $*$  stands for the relaxation factor called Lagrange multipliers. So, the final answer is Equation (10).

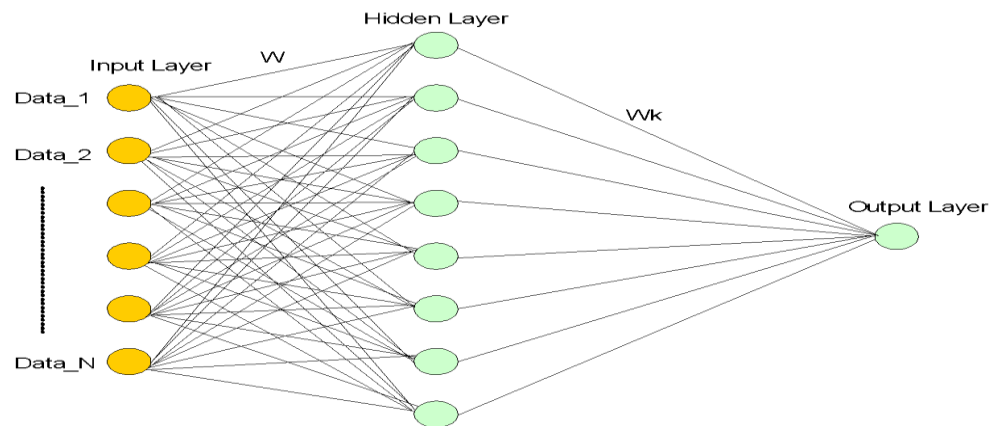
$$\begin{aligned} Y' &= \sum_{j=1}^l (\alpha_j - \alpha_j^*) \phi(x_j) * \phi(x) + b \\ &= \sum_{j=1}^l (\alpha_j - \alpha_j^*) * K(x_j, x) + b \end{aligned} \tag{10}$$

In Equation (10),  $K(x_j, x)$  stands for the kernel function, which can be linear, polynomial, a radial basis function (RBF), or something else. We used an SVM base classifier with linear, polynomial, and RBF kernel functions to predict refactoring in our proposed model.

### Artificial Neural Network

We used ANN to classify the heterogeneous ensemble structure as a model of neuro-computing (base-classifier). ANNs with different learning or weight estimation approaches are used as the basis learners in this study. The next sections go into great detail regarding the ANN models that were used in this study. ANN has become one of the most used and sought-after algorithms among the major neuro-computing ideas. It works similarly to the human brain in that it learns from different data or patterns to classify data it has not seen before (s). ANN is a possible significant Artificial Intelligence solution (AI) or decision-

making task because of how well it learns, how well it learns about depth information, and how well it can classify related information. It is made up of many neurons that send data to different layers, such as the input layer and the hidden layer, to be processed. In the end, classification output is given at the output layer in Figure 2. It uses concepts to reduce errors to learn from the given data. It determines the difference between the expected and observed values, called error. It plans to keep lowering the error until it reaches zero, which means convergence and gives the final output of the output layer. At the output layer, ANN sorts the data it has received into the expected groups. For example, this paper divides each class into two groups: those that are easy to refactor and those that are not.



**Figure 2.** A typical ANN structure.

In practice, obtaining the best classification result requires choosing the right weights, computing quickly, and so on. Otherwise, it goes through local minima and premature convergence, which hurts the overall computing efficiency. ANN requires accurate weight estimates and high learning efficiency to fix these problems and reach a point where there are no mistakes. To achieve this, ANN undergoes several phased changes that have made it better at estimating weight and learning. In this paper, we refactored classification with ANN in different ways. These algorithms (ANN-GD, ANN-GDX, ANN-LM, and ANN-RBF) were used as base learners to create an ensemble.

An ANN model typically consists of input, hidden, and output layers (see Figure 2 for a visual representation). A linear activation function is applied to the neural network to produce the same output ( $O_h = I_h$ ) as the input in our proposed refactoring prediction model. The hidden layer uses data from the input layer to feed into its sigmoid function to arrive at the result  $O_h$ . With the hidden layer  $I_h$ , ANN produces  $O_h$  at the output layer using the sigmoid function, as Equation (11).

$$O_h = \frac{1}{1 + e^{-I_h}} \quad (11)$$

Most of the time, ANN is written as  $Y' = f(W, X)$ , where  $Y'$  is the output vector and  $W$  and  $X$  are the weights. ANN aims to reduce the error function to improve the algorithm's accuracy. So, a mean square error (MSE) function was used as Equation (12).

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - y'_i)^2 \quad (12)$$

In above Equation (12), the parameter  $y$  shows the calculated value, while  $y_i$  shows the intended result. With our planned refactoring prediction system in mind, we fed 125 code metrics into the ANN, allowing the ANN to determine whether each class needed refactoring. So, it gives each class of software code a label, which is then used by a maximum voting ensemble to decide whether the code class is likely to be refactored or not. As was already said, we used different kinds of ANN in this research, such as ANN-GD,

ANN-RBF, ANN-LM, etc. Then we have discussed a small part of these algorithms and how they determine their weights and learn for refactoring prediction at the class level.

**a. ANN-GD**

ANN intends to minimize the error function for all training sets iteratively. Considering the learning regression in conjunction with the targeted refactoring signifies a non-linear weight vector. So, ANN-GD tries to attain a local optimum for learning regression using the gradient descent (GD) method. Noticeably, the GD method updates weight  $w$  iteratively by substituting  $w_t$  by  $w_{t+1}$  using Equations (12)–(14).

$$w_{t+1} = w_t - \eta_t \nabla L \quad (13)$$

$$w_{j,t+1} = w_{j,t} - \eta_t \frac{\partial L}{\partial w_j} \quad (14)$$

In above Equation (13), the parameter  $\eta_t$  signifies the learning rate that reduces as per  $t$  and  $\nabla L$  presents the error value. Performing GD based weight estimation and corresponding learning, ANN-GD classifies each code class as refactoring prone or non-refactoring and labels it as “1” for refactoring prone and “0” for non-refactoring.

**b. ANN-RBF**

RBF-based ANN models have an input, a hidden, and an output layer, similarly to traditional ANN models. However, differing from conventional methods, the neurons in the hidden layer have Gaussian transfer functions with inversely proportional outputs. ANN-RBF is the same as K-Means clustering and Probabilistic Neural Networks (PNN). On the other hand, ANN-RBF has many neurons for each data point, while PNN only has one neuron for each data point (but lower than the number of training points). PNN is appropriate for modest or medium-sized datasets, but its efficiency is problematic for our refactoring prediction system, which can include many data items. In our ANN-RBF model, hidden units enable a collection of random input pattern functions. In this approach, the hidden units are radial centers, a vector  $c_1, c_2, \dots, c_h$ . Non-linear transformation transforms input space into hidden space. The translation from hidden unit space to output space remains linear (n1) for n-point input networks. In the implemented ANN-RBF model, each hidden unit has its own receptive field in the input space. Similarly to regular ANN models, RBF-based ANN models have an input, hidden, and output layer. In contrast to conventional methods, however, the neurons in the hidden layer have Gaussian transfer functions with reverse proportional outputs. ANN-RBF is functionally equivalent to K-Means clustering and Probabilistic Neural Network (PNN). PNN has a single neuron for each training data point, but ANN-RBF contains numerous neurons (but lower than the number of training points).

**c. ANN-LM**

In many classification situations, the ANN-GD and ANN-RBF algorithms have proved successful. However, the issues of adaptive weight assignment and learning remain unsolved. The ANN-LM technique, in contrast to standard neuro-computing models, iteratively accomplishes localization of the minimum value of the multivariate function, sometimes called the Sum of Squares (SoS) non-linear real-valued functions. It makes stronger the ANN-ability LM's to perform weight updates quickly, which speeds up the learning process and prevents local minima. As an additional feature of the ANN-LM model, error minimization is swiftly achieved by setting the learning rate. Weighing is updated according to ANN-weighting LM's algorithm shown in Equation (15).

$$W_{j+1} = W_j - \left( J_j^T J_j + \mu I \right)^{-1} J_j e_j \quad (15)$$

In Equation (15),  $W_j$  is the current weight, and  $W_{j+1}$  is the updated weight in the preceding Equation (15). The identity matrix is represented by  $J$  in Equation (15). ANN-GD

is characterized by a low value of the combination coefficient. The Jacobian matrix is used to train the ANN-LM. In terms of functionality, the refactoring proclivity of a class can be determined using ANN-LM as a base classifier. A recently developed neuro-computing model, ELM has a stronger convergence and learning ability than classical techniques, such as neural networks. As a result of this robustness, in this paper, we have used ELM with multiple kernel functions to predict the refactoring propensity of a given class. To accomplish refactoring proneness classification and per-class labeling, we employed ELM with several kernel functions, including linear, polynomial, and RBF, which were applied as base classifiers. A final prediction was made utilizing the MVE ensemble concept.

Extreme Learning Machine (ELM)

Differing from major conventional neuro-computing models, as discussed above, they undergo certain local minim and convergence and find themselves limited to function over large-scale data. These problems become more severe with large datasets, which demand more input layers and a corresponding ANN structure. Consequently, with classical ANN models for large data sizes, the neurons at the hidden layer also increase, further augmenting the number of weight parameters required for learning. As a result, this makes classical neuro-computing methods vulnerable to high latency and convergence issues. To alleviate such problems, ELM can be a potential alternative. ELM is defined as a single-layered multi-feed-forward neural network (SL-MFNN) that allows more efficient generalization than classical approaches. Using random node selection and corresponding weight estimation makes it more time-efficient. Our proposed refactoring classification method has applied ELM as a base classifier to constitute a heterogeneous ensemble structure. Inputting the 125 code metrics from software, it exhibits multivariate regression, enabling the labeling each class as refactoring prone or non-refactoring. A snippet of the typical ELM model is given in Figure 3.

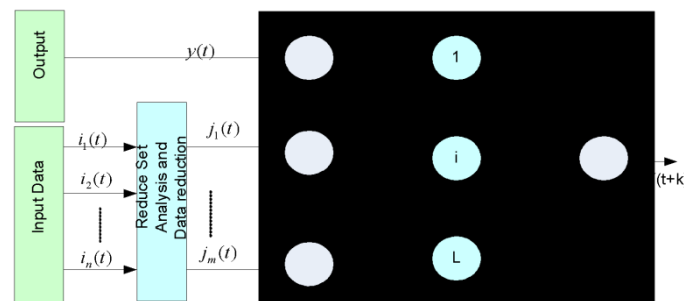


Figure 3. A typical ELM structure for refactoring Prediction.

Observing Figure 3, it can be found that ELM inputs comprise  $X$  and  $Y$ , where  $X = (j_1, j_2, \dots, j_m, y)$  and  $y$  be the targeted value Noticeably, data signifies a vector having  $m + 1$  values, while the outputs from the  $i$ th hidden layer are given as  $G(a_i, b_i, X)$ . In  $G(a_i, b_i, X)$   $b_i$  presents the bias value of the  $i$ th neuron, while  $a_i = (a_{i1}, a_{i2}, \dots, a_{im}, a_{iy})$  refers to the weight vector. Similarly,  $a_{is} (s = 1, 2, \dots, m, y)$  presents the weights between the  $s$ th input layer and the  $i$ th hidden layer. Thus, the output is in Equation (16) with the above-defined ELM model.

$$y(t + k) = f(X) = \sum_{i=1}^L \beta_i G(a_i, b_i, X) \tag{16}$$

In Equation (16),  $\beta_i = (\beta_{i1}, \dots, \beta_{im}, \beta_{iy})'$  presents the weight vector joining the hidden and output layers. Moreover, the connecting weights in between the  $i$ th hidden layer and  $k$ th output layer is given by  $\beta_{ik}$ . The  $G(a_i, b_i, X)$  can be defined for other hidden neurons as shown in Equation (17).

$$G(a_i, b_i, X) = g(a_i'X + b_i) \tag{17}$$



In Equation (17),  $g : R \rightarrow R$  presents the activation function. The ELM model in our proposed classification problem considers random weight component  $a_i$  and biasing component  $b_i$ . Typically, ELM with  $L$  hidden neurons can be hypothesized to exhibit learning over  $N$  samples with zero error probability with the connecting weight  $\beta_i$ . Mathematically, the output can be defined as Equation (18).

$$Y_j = \sum_{i=1}^L \beta_i G(a_i, b_i, X_j), \quad j = 1, 2, \dots, N \tag{18}$$

$$Y = \beta H \tag{19}$$

where

$$Y = \begin{bmatrix} y_1^Y & \dots & y_N^Y \end{bmatrix}_{N \times M}, \quad \beta = \begin{bmatrix} \beta_1^Y & \dots & \beta_L^Y \end{bmatrix}_{L \times M}$$

$$H = \begin{bmatrix} G(a_1, b_1, X_1) & \dots & G(a_L, b_L, X_1) \\ \vdots & \dots & \vdots \\ G(a_1, b_1, X_N) & \dots & G(a_L, b_L, X_N) \end{bmatrix}_{N \times L} \tag{20}$$

Redefining Equation (19), we obtain output as Equation (20). In the proposed ELM model, a minimum norm least-squares solution  $\beta^*$  is applied to minimize  $\beta^*$  in addition to the weight parameters and bias values. Mathematically,  $\beta^*$  is referred to as Equation (21).

$$\beta^* = H^+ Y \tag{21}$$

In Equation (21),  $H^+$  signifies the Moore–Penrose generalized inverse of the matrix  $H$ . The ELM proposed here has achieved the following significant functions to enable binary classification. In our proposed ELM-based classifier, the output  $y^*$  for each class is obtained as per Equation (22).

$$y^* = \sum_{i=1}^L \beta_i^* g(a_i x + b_i) \tag{22}$$

where the error function to be reduced is the root mean square error shown in Equation (23).

$$RMSE = \sqrt{\frac{\sum_{i=1}^N (\hat{y}(i) - y(i))^2}{N}} \tag{23}$$

In Equation (23),  $\hat{y}(i)$  signifies the expected or targeted output, while  $y(i)$  presents the actual output  $N$  is the number of observations. RMSE is the square root of the variance of the residuals is considered as the standard deviation of the unexplained variance, which is reduced over iterations to achieve higher accuracy (utilizing optimal weight and bias values estimation). In the proposed heterogeneous ensemble structure, we applied ELM with three different kernel functions, linear, polynomial and RBF, which classify each class as refactoring prone and non-refactoring. All base classifiers label on each class have been used for the MVE ensemble to perform class-level refactoring prediction.

#### Least Squares Support Vector Machine (LSSVM)

The least-squares linear system is used as a loss function in LSSVM, which is a statistical learning theory. Regularization networks and LSSVM are very similar. The optimization problem comes down to solving linear equations with the quadratic cost function. According to the source [49], we have used the LSSVM algorithm with three different kinds of kernels—linear, polynomial, and RBF kernel functions—which have more information on the LSSVM algorithm. Du et al. [57] implemented a new LSSVM ensemble model for aero engine performance parameter chaotic prediction.

### Maximum Voting Ensemble (MVE) Model

This paper used the aforementioned base classifiers to form a unique heterogeneous ensemble learning model by applying them as the base classifier. To create an ensemble structure, all classifiers are applied to the same dataset and asked to predict whether or not a given class should be refactored (labeled as “1”) or not (labeled as “0”). Therefore, after obtaining each software code’s class outputs (i.e., labels), “Maximum Voting” is achieved for each class. The class that receives the most votes (i.e., 1 or 0) is considered the final category (refactoring prone or non-refactoring).

## 4. Results and Discussion

Considering the significance of reliable software design and probable fault avoidance, the refactoring concept has been recognized as a potential approach. Practically, the refactoring method enables the identification of probable code smells and vulnerabilities and introduces changes to retain code sanity and reliability without making any significant change in functional behavior. Its significance becomes inevitable, especially in the modern software industry where firms try to make component reuse, FOSS to save cost. The industry employs manual testing, regression-based methods, etc., to identify such refactoring probability; however, in contrast to the cost-minimization objective, it increases cost and makes delivery time-consuming. It does not guarantee human error avoidance, especially when the software program size is more. Considering such issues, developing an automatic software refactoring estimation model becomes inevitable. This paper developed a highly robust code-metrics-enabled artificial intelligence concept, exploiting hundreds of the code metrics to assess each class’s refactoring probability, vulnerability, and allied refactoring probability. To achieve this, the research employed a multi-phased paradigm where, at first, a standard source code chunk was obtained from a software engineering benchmark dataset named PROMISE. At first, a total of 125 source code metrics were obtained, including Halstead, Chidambaram, and Kamarer code metrics, Object Oriented Code metrics, etc., realizing that among the 125 features, certain features have higher significance towards refactoring prediction or characterization, while few can be insignificant. Considering this, we first performed significant feature estimation for which a strategic paradigm encompassing Wilcoxon Signed Rank Test, Significant Test, and Cross-Correlation Test was applied in sequence. Here, our prime motive was to retain only the most significant features which can have a high correlation or association with refactoring probability. Here, for significant estimation, we considered a threshold value of 0.5. Once we had obtained the optimal set of features, which were 24 in our case, we performed Min-Max normalization, which mapped each data element into [0, 1] patterns. Noticeably, normalization was mainly performed to avoid any convergence problem.

A highly significant step was performed with the obtained normalized values, which major researchers did not address. To be noted, the software or program as a case study can be of any size with diverse classes, connectivity, coupling, cohesion, etc. Under such circumstances, the probability of data imbalance increased significantly. Unfortunately, it increases the likelihood of false classification, so avoiding such data imbalance was necessary. This paper applied different data-sampling algorithms with this motive, including random sampling, downsampling, and upsampling. We used the confidence interval concept to upsample the input video data, which helped retain features with significance in almost 95% of the original metrics or samples. Thus, with the actual 24 samples or data metrics as an input, we obtained 96 feature samples containing the original sample, random sample, and downsampled and upsampled data, for each of the source code metrics. Noticeably, in each class’s proposed method, we applied 96 feature samples, making classification both robust and accurate. Finally, we performed a two-class classification with the 96 feature sets, classifying each software code class as refactoring prone or non-refactoring. Noticeably, we realized that each classifier has a different classification performance and generalizing performance based on specific random classifiers is not suitable.

This paper designed a novel heterogeneous classifier with machine learning methods of the different types. It strengthened our proposed model to achieve better learning and classification in conjunction with ensemble concepts such as maximum voting ensemble or base-trained ensemble models. As base classifiers, we applied a decision tree, enhanced K-NN classifier, Logistic regression, SVM-Linear, SVM-Polynomial, SVM-RBF, LSSVM with different kernels, ANN-GD, ANN-GDX, ANN-LM, ANN-RBF, ELM with different kernels. Differing from existing approaches where authors split data into multiple chunks and apply different classifiers for ensemble design, we applied the same feature metrics as input-to-all classifiers and performed two-class classification. This method labeled each class as refactoring-prone and non-refactoring, labeled as 1 and 0, respectively. Thus, applying MVE for each class, we automatically classified that class or code component. The prime motive of applying multiple classifiers and their variants was to gain the maximum possible opinion towards refactoring a class's proneness to make an optimal classification. To enhance accuracy, we applied f5-fold cross-validation that resulted in better performance as 99.76. The proposed system was developed using MATLAB2019b, simulated over Microsoft OS with i3 and 4 GB RAM.

It is commonly assumed that ensemble classifiers will outperform classical base learners or single machine learning methods in ensemble classification methods. This paper investigated relative performance by estimating each base learner's performance using the proposed heterogeneous ensemble learner. To achieve it for each classifier, false positives and false negatives (FN) were mapped out into a confusion matrix, which identifies the real positives and false negatives (FN). We used statistical measures such as classification or prediction accuracy, precision, recall, and F-Measure for calculating these matrix values for each base learner and ensemble classifier. The definitions of these performance variables are given in Table 2. During multiple number of times execution, precision measurement is vital for finding the best result in proposed model. Even the authors themselves may not be aware of the degree of precision their algorithms in their current implementation converges. Without knowing the convergence level of all the data, comparing multiple methodologies is not a valid method and could lead to incorrect conclusions. If at one 10-time execution, the obtained least and peak values for our proposed model with MVE are 0.0017 and 0.0123, respectively, then we can observe that the average precision of our model is 0.0140. We have applied the same approach utilized in the paper [58] to evaluate the precision of our model. Different base classifiers and ensemble models' statistical performance has been measured (i.e., decision tree, K-NN classifier, Logistic regression, SVM-Linear, SVM-Poly, SVM-RBF, LSSVM, LSSVM-Lin, LSSVM-Poly, LSSVM-RBF, ANN-GD, ANN-GDX, ANN-LM, ANN-RBF, ELM-Lin, ELM-Poly, ELM-RBF, MVE, and BTE).

**Table 2.** Performance Parameters.

Parameter	Mathematical Expression	Definition
Accuracy	$\frac{(TN+TP)}{(TN+FN+FP+TP)}$	This calculation signifies the proportion of projected refactoring-resistant modules that are inspected out of all modules.
Precision	$\frac{TP}{(TP+FP)}$	Specifies the extent to which repeated tests produce the same findings under unchanged conditions.
F-measure	$\frac{Recall.Precision}{Recall+Precision}$	It takes the recall and precision numbers and makes a single number, the harmonic mean of those two.
Recall	$\frac{TP}{(TP+FN)}$	It displays how many things there are to be picked.

Observing the above-derived results, it can be easily found that the proposed ensemble learning method's accuracy is higher than any other base classifier. The performance of all the classifiers is shown in Table 3, and from the results, we can conclude that MVE achieves a better result than all kinds of individual frequently used classifiers.

**Table 3.** Performance values of ensemble classifiers. Bold identifies the best classifier's result in the model.

Techniques	Accuracy (%)	Precision (%)	Recall (%)	F-Measure (%)
Logistic regression	70.73	69.93	71.93	70.91
Decision Tree	69.81	71.32	73.51	72.35
ANN-GDX	91.48	88.43	81.02	84.56
ANN-LM	91.90	83.94	84.17	84.05
ANN-GD	89.38	84.61	83.91	84.25
SVM-Lin	69.84	70.90	70.06	70.47
SVM-Poly	69.84	70.94	71.00	70.96
SVM-RBF	69.84	69.91	69.99	69.94
LSSVM-Lin	88.42	89.91	86.61	88.22
LSSVM-Poly	88.05	88.94	89.59	89.07
LSSVM-RBF	89.90	94.31	88.92	91.53
ELM-Lin	89.73	90.84	90.62	90.72
ELM-Poly	90.47	91.03	90.01	90.51
ELM-RBF	93.50	91.07	91.10	91.08
MVE	<b>99.76</b>	<b>99.93</b>	<b>98.96</b>	<b>99.44</b>
BTE	99.56	99.16	98.10	98.62

### Error Profiling

As mentioned in the previous sections, ensemble learning aims to reduce prediction errors by amalgamating different classifiers and their respective performances over the original feature data. As a result of this, the proposed ensemble classifier has been evaluated in this paper in terms of different error profile parameters, such as mean absolute error (MAE), root means square error (RMSE) means relative error magnitude (MMSE), etc. The performance comparison was performed with other base classifiers to test the effectiveness of the proposed ensemble learning model refactoring prediction through various errors shown in Table 4. A snippet of the various error parameters and their respective mathematical models is discussed below.

- (a) Mean Absolute Error (MAE);
- (b) Standard Error of the Mean (SEM);
- (c) Mean Magnitude of the Relative Error (MORE);
- (d) Root Mean Square Error (RMSE).

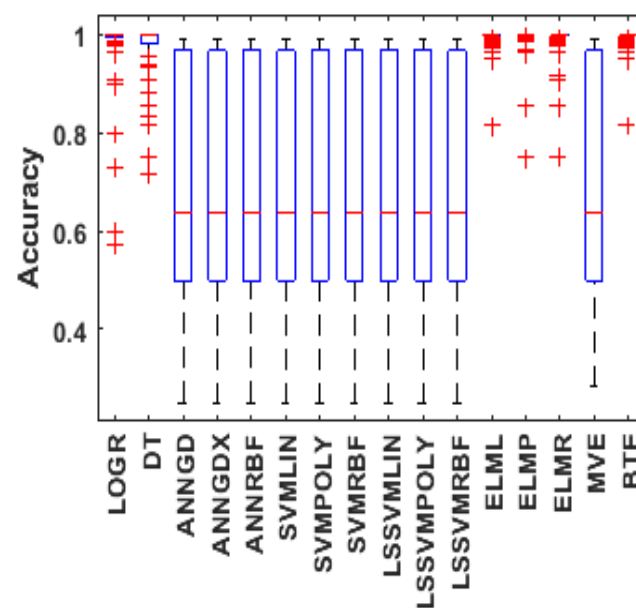
Observing the results in Table 3, it can be found that the proposed ensemble classifier exhibits a minimum of the other base learners. It confirms the efficiency and robustness of the proposed ensemble classifier. Thus, considering the overall research outcomes, the projected ensemble classifier containing base learners can perform highly precise and consistent classification in Table 3. The proposed ensemble classifier has exhibited better performance for refactoring prediction that can eventually help developers and companies achieve reliable software development.

We have shown the comparative study among feature selection and data sampling techniques in a boxplot diagram. Each ensemble classifier's performance in terms of accuracy has been also represented in the form of a boxplot in Figure 4. However, the numerous boxplots for examining outliers, skewness, interquartile range in accuracy, F-Measure, and AUC performance metrics for feature selection and data sample strategies are shown in Figures 5 and 6. The median value, which divides the box into two parts, is indicated by the red line in the boxplot of Figures 5 and 6. According to Figure 5, the median

values for the AUC, F-Measure, and Accuracy of all feature selection strategies produce statistically significant results. Similar to Figure 6, it has been noted that upsampling has a higher median value for AUC, F-Measure, and Accuracy than other data sampling approaches. Figure 4 shows the error performances of all the ensemble classifiers during class-level refactoring model development. We also validate our model with various errors generated by the ensemble classifiers and we represent their performances in terms of the chart shown in Figure 7.

**Table 4.** Error performance by the different base learners and the ensemble classifier. Bold identifies the classifier with less error in the proposed model.

Techniques	MAE	MORE	RMSE	SEM
Logistic regression	0.1968	0.8073	0.542	0.1331
Decision Tree	0.0996	0.7000	0.0200	0.0900
ANN-GDX	0.3001	0.4109	0.1001	0.1642
ANN-LM	0.3109	0.3994	0.1138	0.1981
ANN-GD	0.4111	0.3983	0.1695	0.2001
SVM-Lin	0.1557	0.8623	0.1209	0.0276
SVM-Poly	0.1904	0.8001	0.1245	0.1090
SVM-RBF	0.1321	0.4290	0.1199	0.1008
LSSVM-Lin	0.5731	0.2983	0.0261	0.1990
LSSVM-Poly	0.3901	0.2106	0.0198	0.1179
LSSVM-RBF	0.3860	0.2100	0.0911	0.1108
ELM-Lin	0.2075	0.5892	0.1698	0.1471
ELM-Poly	0.2007	0.4929	0.1604	0.1500
ELM-RBF	0.2000	0.5071	0.1599	0.1403
MVE	<b>0.0057</b>	<b>0.0701</b>	<b>0.0068</b>	<b>0.0107</b>
BTE	0.0912	0.3419	0.1941	0.1610



**Figure 4.** Box plot for ensemble classifier’s accuracy performance.

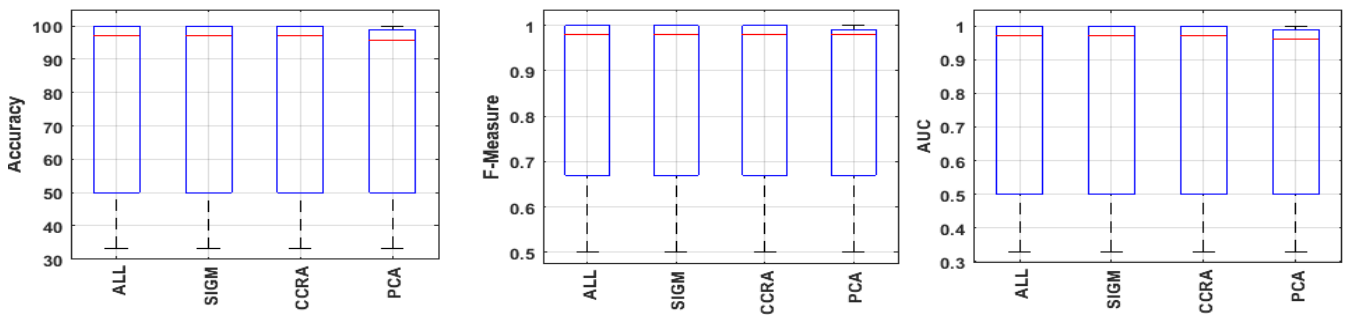


Figure 5. Accuracy, F-Measure, and AUC Performance over different feature selection techniques.

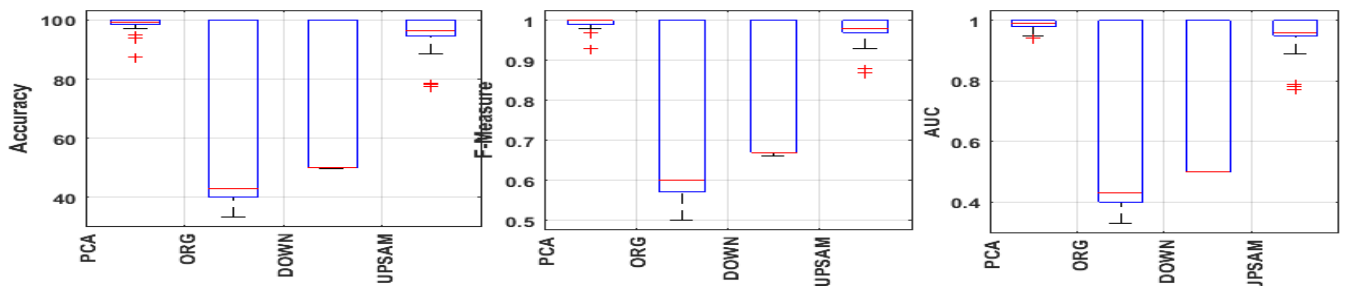


Figure 6. Accuracy, F-Measure, and AUC performance over different data samples.

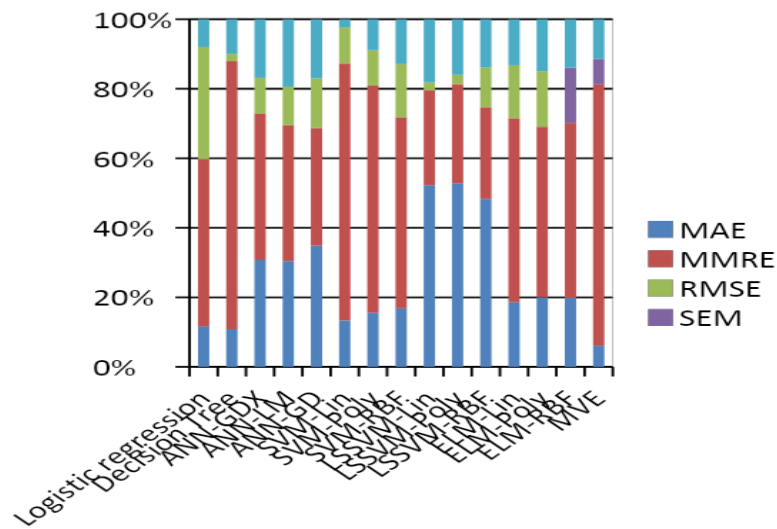


Figure 7. Accuracy Measurement for ensemble algorithms.

### 5. Comparative Analysis

This section summarizes the comparative analysis between different frequently used classifiers, ensemble classifiers, and deep learning classifiers. Many authors worked on refactoring prediction by individual machine learning classifiers and ensemble learning at the class and method levels shown in Tables 5 and 6. Table 5 also represents different authors’ contributions towards the refactoring prediction development and their limitations too.

**Table 5.** Limitations of recently published papers by using frequently used classifiers.

Author	No. of Metrics	Method Level/Class Level	Classifiers	Results	Limitations
Sagar et al. [59]	102 code metrics	Class Level	Gated recurrent unit algorithm	Projects with balanced data achieve better results in comparison with projects with unbalanced data.	Authors should apply the deep learning algorithm with method-level refactoring to obtain a better result.
Kumar et al. [60]	102 code metrics	Class Level	LSSVM	The mean value of the Area Under Curve (AUC) for the LSSVM RBF kernel is 0.96.	Authors can also apply deep learning techniques at the method and class level then there is a chance of performance improvement.
Patnaik et. al. [61]	30 code metrics	Class Level	Naive Bayes classifiers	Mean accuracy for the Gaussian, Bernoulli and Multinomial classifiers 33.33%, 39%, 48.33%.	The authors should provide a comparative study and explain how naive Bayes differs from others with the same kernels.
Panigrahi et. al. [62]	103 code metrics	Method level	Naive Bayes classifiers	Mean accuracy (84%) and AUC (0.78) state that BNB is superior then the other naive Bayes classifiers.	There should be comparative study among naive Bayes and different classifiers.
Panigrahi et. al. [63]	65 code metrics	Method level	SVM and Weighted SVM	SMOTE with SVM achieves good result as comparison to weighted SVM.	The authors have to consider some more parameters for measuring the model's efficiency and should be considered in class level.
Akour et. al. [64]	102 metrics	Class level	SVM with two optimized (GA and Whale) algorithm	SVM with GA and Whalle achieves 96% which is more than only SVM 84%.	The authors need to calculate the AUC value because the data set needs to be more balanced.
Our proposed model	125 Source code metrics	Class level	Logistic Regression, Decision Tree, SVM, ELM, ANN with different kernels, LSSVM with different kernels, MVE	MVE achieves a better result than others based on F-measure, Recall and precision, and error performance.	We will be applying the same for the method level and field level.

**Table 6.** Ensemble techniques performance based on their type of classifiers.

Author	Feature Selection	Sampling Techniques	Ensemble Classifiers Performance	Limitations
Alsolai et. al. [65]	Pearson's correlation coefficient and relief.	SMOTE, randomize, and spread sub-sample	Naive Bayes, support vector machines, k-nearest neighbors, and random forests. Out of all of them, random forest achieves better result the others.	Authors will obtain a better performance if they use ensemble feature selection and sampling for the same work.
Aribandi et. al. [66]	Wilcoxon rank sum test, correlation test, Recursive Feature Elimination (RFE)	—————	Neural network with three different training algorithm, support vector machine with three different kernels least square support vector machine with three kernels and they conclude that LSSVM provides a better result.	As the data set is highly unbalanced they should use some well-defined data sampling techniques to obtain more accurate result.
Catolino [67]	—————	—————	Boosting, Bagging, Random Forest, and Voting with Logistic Regression, Simple Logistic, Naive Bayes, and Multilayer Perceptron.	They should consider different projects belonging to different environment rather than same environment.
Alenezi et. al. [68]	commit based and code based	—————	Logistic Regression, Naive Bayes, SVM, Random forest random forest model trained with code metrics resulted in the best average accuracy of 75%.	A comparative study should be provided between ensemble learning and deep learning.
Our proposed model	Wilcoxon rank sum test, cross-correlation test, significant test	UPsampling, Downsampling, Randomsampling, PCA	Logistic Regression, Decision Tree, SVM, ELM, ANN with different kernels, LSSVM with differnt kernels, MVE	We have not considered method level on more benchmarking data sets. So we will be using the same for method level refactoring as our future work.

Researchers have presented many prediction methods based on source code metrics to forecast the change-proneness of classes. However, several of these models have low prediction accuracy due to large complexity or imbalanced classes in the data set. Recent research indicates that utilizing ensembles to combine multiple models, pick features, or execute sampling can overcome dataset difficulties and enhance prediction accuracy. This section is designed to offer a comparative study between the existing refactoring model and our proposed refactoring model. We have considered four performance parameters in the above comparative analysis in Table 7. According to the accuracy parameter, our model provides more accurate results than other existing model. The other parameters such as F-measure, recall, and precision also recommend that the refactoring prediction model with an ensemble classifier achieves a better result than others, experiencing the least errors. This work aims to empirically assess the usefulness of ensemble models, feature selection, and sampling approaches in forecasting refactoring candidates using various errors.



**Table 7.** An empirical comparative study of refactoring model at class and method level [State-of-art].

Author with Reference	Classifiers	Accuracy	F-Measure	Recall	Precision
Sagar et al. [59]	LSTM (Text Based)	54.3%	0.21	0.1176	1.0
	LSTM (Code metrics)	40.67%	0.67	0.0071	0.014
Sagar et al. [59]	Random forest	75%	0.81	0.75	0.75
	Logistic Regression	47%	0.53	0.46	0.45
	SVM	44%	0.55	0.43	0.42
	Naïve Bayes	35%	0.49	0.35	0.33
Alenezi et. al. [60]	Gated Recurrent unit Algorithm with out SMOTE	95.91	35.23	34.26	31.84
Alenezi et. al. [60]	Gated Recurrent unit Algorithm with SMOTE	98.17%	100	96.44	98.11
Kumar et al. [61]	LSSVM with out SMOTE	99.67%	0.9958	—	—
	LSSVM with SMOTE	99.17%	0.9958	—	—
Patnaik et. al. [62] (naïve Bayes at class level)	Gaussian	47.33%	—	—	—
	Bernoulli	41.33%	—	—	—
	Multinomial	32%	—	—	—
Panigrahi et. al. [63] (naïve Bayes at method level)	Gaussian	63.82%	—	—	—
	Bernoulli	70.66%	—	—	—
	Multinomial	84.64%	—	—	—
Panigrahi et. al. [64]	SVM with SMOTE	82%	—	—	—
	Weighted SVM	88.81%	—	—	—
Akour et. al. [65] SVM with two optimized (GA and Whale) algorithm	SVM	88.125	0.92	—	—
	GA+ SVM	90.25	0.948	—	—
	Whale + SVM	90.10	0.947	—	—
	GA + Whale + SVM	90.15	0.947	—	—
Gerling [66]	Random Forest	0.72	0.72	0.977	0.87
	Logistic regression	0.827	0.85	0.95	0.76
<b>Our proposed approach</b>					
	Logistic regression	70.73	69.93	71.93	70.91
	Decision Tree	69.81	71.32	73.51	72.35
	ANN-GDX	91.48	88.43	81.02	84.56
	ANN-LM	91.90	83.94	84.17	84.05
	ANN-GD	89.38	84.61	83.91	84.25
	SVM-Lin	69.84	70.90	70.06	70.47
	SVM-Poly	69.84	70.94	71.00	70.96
	SVM-RBF	69.84	89.91	69.99	69.94
	LSSVM-Lin	88.05	89.91	86.61	88.22
	LSSVM-Poly	89.90	88.94	89.59	89.07
	LSSVM-RBF	89.73	94.31	88.92	91.53
	ELM-Lin	90.47	90.84	90.62	90.72
	ELM-Poly	90.47	91.03	90.01	90.51
	ELM-RBF	93.50	91.07	91.10	91.08
	MVE	<b>99.76</b>	<b>99.93</b>	<b>98.96</b>	<b>99.44</b>
	BTE	99.56	99.16	98.10	98.62

## 6. Conclusions and Future Work

This paper focused on an ideal computing environment to predict refactoring that could promote cost-effective and consistent software design. In the planned model, different software metrics as features, including object-oriented code metrics, were considered to distinguish each refactoring-prone code class and non-refactoring. Considering this aim,

computing a huge number of software metrics that focus on object-oriented programming code characteristics such as coupling, cohesion, complexity, depth, dependency, etc., is considered in this paper. A total of 125 metrics were computed, which need to be processed to select significant features. The proposed model aims to retain only significant features for classification and achieve it, and a multi-phased feature selection method was implemented. The sequential implementation of the Wilcoxon significant test or rank-sum test, Pearson Correlation Test and Principal Component Analysis strengthened the retention of the most important features for further computation and thus achieves higher computational efficiency.

In this paper, the data imbalance problem has been resolved through three different sampling methods: random sampling, upsampling, and downsampling. For classification, they provided sufficient training data in conjunction with original samples, performing normalization over final data samples; the proposed model achieved consistent data for subsequent learning and classification. In this research, instead of a single classical classifier-based prediction model, many classification algorithms, from pattern mining, decision trees, neuro-computing, etc., were used to create a heterogeneous ensemble structure that could predict class-level refactoring. This study revealed that the proposed maximum voting ensemble-based classification model outperforms other state-of-art base learners and exhibits better efficiency, signifying its robustness in performing automatic refactoring prediction in the software program. Using multi-traits code features, multi-phased feature selection, data augmentation (sampling methods to avoid data imbalance), and cross-validation-assisted ensemble classification achieves optimal refactoring prediction efficiency.

Furthermore, distinct from single feature selection-based data (either of Wilcoxon rank-sum test, Pearson Correlation Test, or Principal Component Analysis), the combined features (i.e., all (concatenated) obtained from Wilcoxon rank-sum test, Pearson Correlation Test, and Principal Component Analysis) could provide more accurate prediction accuracy. Thus, this research recommends applying multi-traits, code-features, multi-phased feature selection, data augmentation, and heterogeneous ensemble with different classifiers regarding pattern mining, decision tree, neuro-computing, etc. to develop a refactoring prediction system that can help companies and developers design software that is more reliable, cost-effective, and of higher quality. We also plan to implement ensemble classifiers for refactoring prediction for the method, field, and package levels and on more benchmarking data sets.

**Author Contributions:** Conceptualization, R.P., S.M., S.K.K. and L.K.; methodology, S.M. and L.K.; software, L.K.; validation, R.P. and L.K.; investigation, R.P., S.M. and L.K.; resources, L.K.; data curation, R.P.; writing—original draft preparation R.P. and S.M., writing—review and editing, S.K.K. and L.K.; supervision, S.M., S.K.K. and L.K.; project administration, S.M.; funding acquisition, S.M. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Not applicable.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Zimmermann, O. Architectural Refactoring: A Task-Centric View on Software Evolution. *IEEE Softw.* **2015**, *32*, 26–29. [[CrossRef](#)]
2. Bavota, G.; De Lucia, A.; Di Penta, M.; Oliveto, R.; Palomba, F. An experimental investigation on the innate relationship between quality and refactoring. *J. Syst. Softw.* **2015**, *107*, 1–14. [[CrossRef](#)]
3. Fowler, M. *Refactoring: Improving the Design of Existing Code*; Addison-Wesley Professional: Boston, MA, USA, 2018. [[CrossRef](#)]
4. Peruma, A.; Simmons, S.; AlOmar, E.A.; Newman, C.D.; Mkaouer, M.W.; Ouni, A. How do I refactor this? An empirical study on refactoring trends and topics in Stack Overflow. *Empir. Softw. Eng.* **2021**, *27*, 11. [[CrossRef](#)]

5. Kessentini, W.; Kessentini, M.; Sahraoui, H.; Bechikh, S.; Ouni, A. A cooperative parallel search-based software engineering approach for code-smells detection. *IEEE Trans. Softw. Eng.* **2014**, *40*, 841–861. [[CrossRef](#)]
6. Liu, H.; Guo, X.; Shao, W. Monitor-Based Instant Software Refactoring. *IEEE Trans. Softw. Eng.* **2013**, *39*, 1112–1126. [[CrossRef](#)]
7. Fontana, F.A.; Braione, P.; Zanoni, M. Automatic detection of bad smells in code: An experimental assessment. *J. Object Technol.* **2012**, *11*, 5.
8. Abdelmoez, W.; Kosba, E.; Iesa, A.F. Risk-based code smells detection tool. In Proceedings of the International Conference on Computing Technology and Information Management (ICCTIM), Dubai, United Arab Emirates, 9–11 April 2014; p. 148.
9. Dewangan, S.; Rao, R.S. Code Smell Detection Using Classification Approaches. In *Intelligent Systems*; Springer: Singapore, 2022; pp. 257–266. [[CrossRef](#)]
10. Yordanos, F. Detecting Code Smells Using Machine Learning Techniques. Ph.D. Thesis, Debre Birhan University, Debre Berhan, Ethiopia, 2022.
11. Kumar, L.; Lal, S.; Goyal, A.; Murthy, N.B. Change-proneness of object-oriented software using a combination of feature selection techniques and ensemble learning techniques. In Proceedings of the 12th Innovations on Software Engineering Conference (formerly known as India Software Engineering Conference), Pune, India, 14–16 February 2019; pp. 1–11.
12. Sidhu, B.K.; Singh, K.; Sharma, N. A machine learning approach to software model refactoring. *Int. J. Comput. Appl.* **2020**, *44*, 166–177. [[CrossRef](#)]
13. Al Dallal, J. Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics. *J. Inf. Softw. Technol.* **2012**, *54*, 1125–1141. [[CrossRef](#)]
14. Kádár, I.; Hegedűs, P.; Ferenc, R.; Gyimóthy, T. A manually validated code refactoring dataset and its assessment regarding software maintainability. In Proceedings of the 12th International Conference on Predictive Models and Data Analytics in Software Engineering, Ciudad Real, Spain, 7 September 2016; pp. 1–4.
15. Kádár, I.; Hegedűs, P.; Ferenc, R.; Gyimóthy, T. A code is a refactoring dataset and its assessment regarding software maintainability. In Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Osaka, Japan, 14–18 March 2016; IEEE: Piscataway, NJ, USA, 2016; Volume 1, pp. 599–603.
16. Bashir, R.S.; Lee, S.P.; Yung, C.C.; Alam, K.A.; Ahmad, R.W. A Methodology for Impact Evaluation of Refactoring on External Quality Attributes of a Software Design. In Proceedings of the 2017 International Conference on Frontiers of Information Technology (FIT), Islamabad, Pakistan, 18–20 December 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 183–188. [[CrossRef](#)]
17. Vimaladevi, M.; Zayaraz, G. Stability Aware Software Refactoring Using Hybrid Search-Based Techniques. In Proceedings of the 2017 International Conference on Technical Advancements in Computers and Communications (ICTACC), Melmaurvathur, India, 10–11 April 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 32–35.
18. Krishna, Y.; Alshayeb, M. An empirical study on the effect of the order of applying software refactoring. In Proceedings of the 2016 7th International Conference on Computer Science and Information Technology (CSIT), Amman, Jordan, 13–14 July 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 1–4.
19. Kaur, G.; Singh, B. Improving the quality of software by refactoring. In Proceedings of the 2017 International Conference on Intelligent Computing and Control Systems (ICICCS), Madurai, India, 15–16 June 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 185–191.
20. Malhotra, R.; Chug, A. An empirical study to assess the effects of refactoring on software maintainability. In Proceedings of the 2016 International Conference on Advances in Computing, Communications, and Informatics (ICACCI), Jaipur, India, 21–24 September 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 110–117. [[CrossRef](#)]
21. Desai, A.B.; Parmar, J.K. Refactoring Cost Estimation (RCE) Model for Object-Oriented System. In Proceedings of the 2016 IEEE 6th International Conference on Advanced Computing (IACC), Bhimavaram, India, 27–28 February 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 214–218.
22. Lacerda, G.; Petrillo, F.; Pimenta, M.; Guéhéneuc, Y.G. Code smells and refactoring: A tertiary systematic review of challenges and observations. *J. Syst. Softw.* **2020**, *167*, 110610. [[CrossRef](#)]
23. Singh, S.; Kaur, S. A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Eng. J.* **2018**, *9*, 2129–2151. [[CrossRef](#)]
24. Liu, H.; Liu, Y.; Xue, G.; Gao, Y. Case study on software refactoring tactics. *IET Softw.* **2014**, *8*, 1–11. [[CrossRef](#)]
25. Santos, B.M.; de Guzmán, I.G.R.; de Camargo, V.V.; Piattini, M.; Ebert, C. Software refactoring for system modernization. *IEEE Softw.* **2017**, *35*, 62–67. [[CrossRef](#)]
26. Han, A.R.; Bae, D.H. An efficient method for assessing the impact of refactoring candidates on maintainability based on matrix computation. In Proceedings of the 2014 21st Asia-Pacific Software Engineering Conference, Jeju, Republic of Korea, 1–4 December 2014; IEEE: Piscataway, NJ, USA, 2014; Volume 1, pp. 430–437.
27. Khlif, W.; Ben-Abdallah, H. Integrating semantics and structural information for BPMN model refactoring. In Proceedings of the 2015 IEEE/ACIS 14th International Conference on Computer and Information Science (ICIS), Las Vegas, NV, USA, 28 June–1 July 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 656–660.
28. Arcelli, D.; Cortellessa, V.; Di Pompeo, D. Performance-Driven Software Architecture Refactoring. In Proceedings of the 2018 IEEE International Conference on Software Architecture Companion (ICSA-C), Seattle, WA, USA, 30 April–4 May 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 2–3.

29. Tao, B.; Qian, J. Refactoring concurrent java programs based on synchronization requirement analysis. In Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, 29 September–3 October 2014; IEEE: Piscataway, NJ, USA, 2014; pp. 361–370.
30. Singh, N.K.; Aït-Ameur, Y.; Mery, D. Formal ontology-driven model refactoring. In Proceedings of the 2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS), Melbourne, Australia, 12–14 December 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 136–145.
31. Tarwani, S.; Chug, A. Sequencing of refactoring techniques by Greedy algorithm for maximizing maintainability. In Proceedings of the 2016 International Conference on Advances in Computing, Communications, and Informatics (ICACCI), Jaipur, India, 21–24 September 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 1397–1403. [[CrossRef](#)]
32. Soares, G.; Gheyi, R.; Massoni, T. Automated behavioral testing of refactoring engines. *IEEE Trans. Softw. Eng.* **2012**, *39*, 147–162. [[CrossRef](#)]
33. Wang, Y.; Yu, H.; Zhu, Z.; Zhang, W.; Zhao, Y. Automatic Software Refactoring via Weighted Clustering in Method-Level Networks. *IEEE Trans. Softw. Eng.* **2017**, *44*, 202–236. [[CrossRef](#)]
34. Alves, E.L.G.; Song, M.; Massoni, T.; Machado, P.D.L.; Kim, M. Refactoring Inspection Support for Manual Refactoring Edits. *IEEE Trans. Softw. Eng.* **2017**, *44*, 365–383. [[CrossRef](#)]
35. Shahidi, M.; Ashtiani, M.; Zakeri-Nasrabadi, M. An automated extract method refactoring approach to correct the long method code smell. *J. Syst. Softw.* **2022**, *187*, 111221. [[CrossRef](#)]
36. Alton, N.; Batory, D. On Proving the Correctness of Refactoring Class Diagrams of MDE Metamodels. *ACM Trans. Softw. Eng. Methodol.* **2022**. [[CrossRef](#)]
37. Leandro, O.; Gheyi, R.; Teixeira, L.; Ribeiro, M.; Garcia, A. A Technique to Test Refactoring Detection Tools. In Proceedings of the XXXVI Brazilian Symposium on Software Engineering, Virtual Event Brazil, 5–7 October 2022; pp. 188–197.
38. Marcos, C.; Rago, A.; Pace, J.A.D. Improving use case specifications using refactoring. *IEEE Lat. Am. Trans.* **2015**, *13*, 1135–1140. [[CrossRef](#)]
39. Dig, D. Refactoring for Asynchronous Execution on Mobile Devices. *IEEE Softw.* **2015**, *32*, 52–61. [[CrossRef](#)]
40. Lu, H.; Wang, S.; Yue, T.; Nygård, J.F. Automated refactoring of OCL constraints with search. *IEEE Trans. Softw. Eng.* **2017**, *45*, 148–170. [[CrossRef](#)]
41. Stolee, K.T.; Elbaum, S. Identification, impact, and refactoring of smells in pipe-like web mashups. *IEEE Trans. Softw. Eng.* **2013**, *39*, 1654–1679. [[CrossRef](#)]
42. Kumar, L.; Naik, D.K.; Rath, S.K. Validating the Effectiveness of Object-Oriented Metrics for Predicting Maintainability. *Procedia Comput. Sci.* **2015**, *57*, 798–806. [[CrossRef](#)]
43. AlOmar, E.A.; Liu, J.; Addo, K.; Mkaouer, M.W.; Newman, C.; Ouni, A.; Yu, Z. On the documentation of refactoring types. *Autom. Softw. Eng.* **2022**, *29*, 9. [[CrossRef](#)]
44. Al Dallal, J. Predicting move method refactoring opportunities in object-oriented code. *Inf. Softw. Technol.* **2017**, *92*, 105–120. [[CrossRef](#)]
45. Chaparro, O.; Bavota, G.; Marcus, A.; Di Penta, M. On the impact of refactoring operations on code quality metrics. In Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, 29 September–3 October 2014; IEEE: Piscataway, NJ, USA, 2014; pp. 456–460.
46. Ivers, J.; Seifried, C.; Ozkaya, I. Untangling the Knot: Enabling Architecture Evolution with Search-Based Refactoring. In Proceedings of the 2022 IEEE 19th International Conference on Software Architecture (ICSA), Honolulu, HI, USA, 12–15 March 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 101–111. [[CrossRef](#)]
47. Nyamawe, A.S. Mining commit messages to enhance software refactorings recommendation: A machine learning approach. *Mach. Learn. Appl.* **2022**, *9*, 100316. [[CrossRef](#)]
48. Aniche, M.; Maziero, E.; Durelli, R.; Durelli, V. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Trans. Softw. Eng.* **2020**, *48*, 1432–1450. [[CrossRef](#)]
49. Kumar, L.; Sureka, A. Application of LSSVM and SMOTE on seven open-source projects for predicting refactoring at the class level. In Proceedings of the 2017 24th Asia-Pacific Software Engineering Conference (APSEC), Nanjing, China, 4–8 December 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 90–99.
50. Kumar, L.; Satapathy, S.M.; Sureka, A. Method Level Refactoring Prediction on Five Open Source Java Projects using Machine Learning Techniques. In Proceedings of the India Software Engineering Conference, Bangalore, India, 18–20 February 2015.
51. Panigrahi, R.; Kuanar, S.K.; Kumar, L. An Empirical Study for Method-Level Refactoring Prediction by Ensemble Technique and SMOTE to Improve Its Efficiency. *Int. J. Open Source Softw. Process.* **2021**, *12*, 19–36. [[CrossRef](#)]
52. Data Set for Refactoring Prediction. Available online: <https://github.com/rasmitapanigrahi/data-set> (accessed on 20 March 2020).
53. Kim, M.; Gee, M.; Loh, A.; Rachatasumrit, N. Ref-finder: A refactoring reconstruction tool based on logic query templates. In Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Software Engineering Foundations, Santa Fe, NM, USA, 7–11 November 2010; pp. 371–372.
54. Tool for Extracting Source Code Metrics. Available online: <https://www.sourcemeeter.com/> (accessed on 20 March 2020).
55. Alsolai, H.; Roper, M. The Impact of Ensemble Techniques on Software Maintenance Change Prediction: An Empirical Study. *Appl. Sci.* **2022**, *12*, 5234. [[CrossRef](#)]

56. Muruges, S.; Jaya, A. An integrated approach towards automated software requirements elicitation from unstructured documents. *J. Ambient. Intell. Humaniz. Comput.* **2021**, *12*, 3763–3773. [[CrossRef](#)]
57. Du, D.; Jia, X.; Hao, C. A new least squares support vector machines ensemble model for aero engine performance parameter chaotic Prediction. *Math. Probl. Eng.* **2016**, *2016*, 4615903. [[CrossRef](#)]
58. Pan, W.; Ming, H.; Yang, Z.; Wang, T. Comments on “Using k-core Decomposition on Class Dependency Networks to Improve Bug Prediction Model’s Practical Performance”. *IEEE Trans. Softw. Eng.* **2022**, *1*. [[CrossRef](#)]
59. Sagar, P.S.; AlOmar, E.A.; Mkaouer, M.W.; Ouni, A.; Newman, C.D. Comparing Commit Messages and Source Code Metrics for the Prediction Refactoring Activities. *Algorithms* **2021**, *14*, 289. [[CrossRef](#)]
60. Kumar, L.; Satapathy, S.M.; Krishna, A. Applying smote and lssvm with various kernels for predicting refactoring at method level. In Proceedings of the International Conference on Neural Information Processing, Siem Reap, Cambodia, 13–16 December 2018; Springer: Cham, Switzerland, 2018; pp. 150–161.
61. Patnaik, A.; Panigrahi, R.; Padhy, N. Prediction Of Accuracy On Open Source Java Projects Using Class Level Refactoring. In Proceedings of the 2020 International Conference on Computer Science, Engineering and Applications (ICCSEA), Gunupur, India, 13–14 March 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1–6. [[CrossRef](#)]
62. Panigrahi, R.; Kuanar, S.K.; Kumar, L. Application of Naïve Bayes classifiers for refactoring Prediction at the method level. In Proceedings of the 2020 International Conference on Computer Science, Engineering and Applications (ICCSEA), Gunupur, India, 13–14 March 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1–6. [[CrossRef](#)]
63. Panigrahi, R.; Kunaar, S.; Kumar, L. Method Level Refactoring Prediction by Weighted-SVM Machine Learning Classifier. In *Mobile Application Development: Practice and Experience*; Springer: Bhubaneswaar, India, 2023.
64. Akour, M.; Alenezi, M.; Alsgaier, H. Software Refactoring Prediction Using SVM and Optimization Algorithms. *Processes* **2022**, *10*, 1611. [[CrossRef](#)]
65. Gerling, J. Machine Learning for Software Refactoring: A Large-Scale Empirical Study. Master’s Thesis, Delft University of Technology, Delft, The Netherlands, 2020.
66. Hegedus, P.; Kádár, I.; Ferenc, R.; Gyimóthy, T. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. *Inf. Softw. Technol.* **2018**, *95*, 313–327. [[CrossRef](#)]
67. Catolino, G.; Ferrucci, F. An extensive evaluation of ensemble techniques for software change prediction. *J. Softw. Evol. Process* **2019**, *31*, e2156. [[CrossRef](#)]
68. Alenezi, M.; Akour, M.; Al Qasem, O. Harnessing deep learning algorithms to predict software refactoring. *TELKOMNIKA Telecommun. Comput. Electron. Control.* **2020**, *18*, 2977–2982. [[CrossRef](#)]