Ensemble methods with feature selection and data balancing for improved code smells classification performance

Article *in* Engineering Applications of Artificial Intelligence · October 2024

DOI: 10.1016/j.engappai.2024.109527

CITATIONS
10

READS
126

4 authors:

Pravin Singh Yadav
Guru Ghasidas Vishwavidyalaya
6 PUBLICATIONS **76** CITATIONS
SEE PROFILE

Rajwant Singh Rao
Guru Ghasidas Vishwavidyalaya
23 PUBLICATIONS **369** CITATIONS
SEE PROFILE

Alok Mishra
Norwegian University of Science and Technology
288 PUBLICATIONS **5,523** CITATIONS
SEE PROFILE

Manjari Gupta
Banaras Hindu University
74 PUBLICATIONS **1,140** CITATIONS
SEE PROFILE

Engineering Applications of Artificial Intelligence 139 (2025) 109527

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

Ensemble methods with feature selection and data balancing for improved code smells classification performance

Pravin Singh Yadav [a], Rajwant Singh Rao [a], Alok Mishra [b,*], Manjari Gupta [c]

[a] Department of Computer Science and Information Technology, Guru Ghasidas Vishwavidyalaya, Bilaspur, India
[b] Faculty of Engineering, Norwegian University of Science and Technology (NTNU), Norway
[c] (Computer Science), DST - Centre for Interdisciplinary Mathematical Sciences, Institute of Science, Banaras Hindu University, Varanasi, India

ARTICLE INFO

Keywords:
Bagging
Boosting
Code smell
Cross-validation
Ensemble model
Machine-learning models

ABSTRACT

Code smells are software flaws that make it challenging to comprehend, develop, and maintain the software. Identifying and removing code smells is crucial for software quality. This study examines the effectiveness of several machine-learning models before and after applying feature selection and data balancing on code smell datasets. Extreme Gradient Boosting, Gradient Boosting, Adaptive Boosting, Random Forest, Artificial Neural Network (ANN), and Ensemble model of Bagging, and the two best-performing Boosting techniques are used to predict code smell. This study proposes an enhanced approach, which is an ensemble model of the Bagging and Boosting classifier (EMBBC) that incorporates feature selection and data balancing techniques to predict code smells. Four publicly available code smell datasets, Blob Class, Data Class, Long Parameter List, and Switch Statement, were considered for the experimental work. Classes of datasets are balanced using the Synthetic Minority Over-Sampling Technique (SMOTE). A feature selection method called Recursive Feature Elimination with Cross-Validation (RFECV) is used. This study shows that the ensemble model of Bagging and the two best-performing Boosting techniques performs better in Blob Class, Data Class, and Long Parameter List datasets with the highest accuracy of 99.21%, 99.21%, and 97.62%, respectively. In the Switch Statement dataset, the ANN model provides a higher accuracy of 92.86%. Since the proposed model uses only seven features and still provides better results than others, it could be helpful to detect code smells for software engineers and practitioners in less computational time, improving the system's overall performance.

1. Introduction

Nowadays, code smell detection has been considered an emerging area of research due to the huge need for software in day-to-day life. In the software development life cycle, code smells are a problem that needs to be modified. As code smells disappear from software, software quality improves. A software product's quality directly correlates to the efficiency and effectiveness of the development process (Recupito et al., 2024; Yu and Mishra, 2012). Functional and non-functional values are needed at every stage of the SDLC to ensure high-quality software (Mhawish and Gupta, 2020). Bugs are bound to pop up in the code base while making software; engineers apply different tools and need quality assurance to ensure the quality of the software (Baciejowski et al., 2023; Khatami and Zaidman, 2023; Zaidman, 2024). Software maintenance refers to the changes made to the software to adapt or change the software's environment. Weak software design and poor implementation techniques are the main causes of the increase in maintenance effort. It is crucial to address code smells promptly to prevent these issues from escalating (Alawadi et al., 2024; Madeyski and Lewowski, 2023; Mhawish and Gupta, 2019; Mishra and Mishra, 2009).

Machine learning (ML) models, a relatively new area, hold immense potential for detecting code smells (Al-Shaaby et al., 2020). However, further development is needed in this domain. Few studies have explored the feature selection method, and using an ensemble learning approach for code smell detection has received less attention. Several researchers have proposed using code restructuring and detection approaches to enhance software quality (Sahin et al., 2014; Sjoberg et al., 2013; Yamashita and Counsell, 2013; Yamashita and Moonen, 2012, 2013). Numerous studies have examined the volume of software system modifications to understand the impact of code smells on software

* Corresponding author.
E-mail addresses: pravinsingh1110@gmail.com (P.S. Yadav), rajwantrao@gmail.com (R.S. Rao), alok.mishra@ntnu.no (A. Mishra), manjari@bhu.ac.in (M. Gupta).

https://doi.org/10.1016/j.engappai.2024.109527
Received 4 January 2024; Received in revised form 24 August 2024; Accepted 20 October 2024

evolution (Khomh et al., 2009; S. Olbrich et al., 2009; S. M. Olbrich et al., 2010). These studies have found that classes with code smells undergo more frequent changes and require more maintenance. While many researchers are focusing on code smells like Long Method, Feature Envy, and Blob Class, there are still some code smells that need to be identified (Dewangan and Rao, 2022; A. Kaur et al., 2020).

In the classification model, class imbalance is a common issue during training and validation. The problem of class imbalance reduces the effectiveness of model classification and results in imbalanced false-positive and false-negative findings (Khleel and Nehéz, 2022; Thakur et al., 2024). Recognizing and addressing this issue is a crucial step towards improving the accuracy and reliability of our models.

1.1. Problem statement

Researchers utilized various empirical tests and techniques to identify code smells. They found different outcomes in these studies. It is observed that many factors affect the accuracy of ML-based models. According to several reviews and comparisons, as shown in Table 1, there are many causes and constraints for the various outcomes, including the difficulty of establishing official definitions for code smell. The following elements, among many others, should also be considered.

- The numerous amounts of one class instance are there, and another class instance is very few, called imbalance dataset may produce the wrong prediction.
- Feature selection techniques are data preprocessing techniques that may cause improper prediction.
- Model performance can also be affected by data leakage. The testing data must be utterly unknown to the model for good performance analysis.

1.2. Motivation

Numerous Bagging, Boosting, feature selection, and data balancing techniques are utilized for code smell detection. However, to the best of authors knowledge, there is no study where any ensemble model of Bagging and the two best-performing Boosting techniques is applied with feature selection and data balancing for detecting code smells (Alazba and Aljamaan, 2021; Aljamaan, 2021; Fontana et al., 2016; Fontana and Zanoni, 2017; Barbez et al., 2020; Dewangan et al., 2021; Di Nucci et al., 2018; Guggulothu and Moiz, 2019; I. Kaur and Kaur, 2021; Kiyak et al., 2019; Mhawish and Gupta, 2020; Pecorelli et al., 2019). Further, many studies have discussed Blob Class, Data Class, Feature Envy, and Long Method code smell in the literature. However, few researchers have addressed Switch Statement and Long Parameter List code smell (Alazba and Aljamaan, 2021). Detecting code smells has been the subject of extensive research; further positive outcomes can be achieved through continued research efforts. The proposed study implements a hybrid ensemble model that combines Bagging and two best-performing Boosting techniques in a novel way to enhance code smell detection performance. This study applied four existing ensemble models, the EMBBC and Artificial Neural Networks (ANN) models. It integrated feature selection methods and data balancing techniques within the ensemble framework to improve the robustness and accuracy of the detection process and fill this research gap. Four code smell datasets have been utilized, including Switch Statement and Long Parameter List.

1.3. The contributions of this study are-

- The proposed study introduces an enhanced approach, which is an ensemble model that combines the Bagging technique with the two best-performing Boosting techniques (Extreme Gradient Boosting (XGB), Gradient Boosting (GB), and Adaptive Boosting (ADA)). This ensemble model, termed EMBBC, is a novel approach that aims to improve the accuracy and robustness of code smell detection.
- The study evaluated the proposed model against five models viz Extreme Gradient Boosting (XGB), Gradient Boosting (GB), Adaptive Boosting (ADA), Random Forest (RF), and Artificial Neural Networks (ANN).
- The study evaluated the effectiveness of Recursive feature elimination with Cross-Validation (RFECV) in the four datasets used in this experiment.
- The study critically evaluated the effectiveness of the Synthetic Minority Over-sampling Technique (SMOTE), a key data balancing technique, in the four datasets used in this experiment.
- The study conducted a rigorous and comprehensive experimental analysis on four distinct datasets: Blob Class, Data Class, Switch Statement, and Long Parameter List. This thorough approach provides robust insights and ensures the reliability of the study's findings. The depth of the analysis instils confidence in the audience about the validity of the study's conclusions, fostering a sense of trust in the research.

1.4. Research questions

The proposed model addresses specific research questions (RQ) crucial for advancing our understanding of code smell detection. These questions guide the study and provide a clear direction for the research. They are introduced here, along with their motivations:

RQ1 What is the effectiveness of the ML models and EMBBC in finding code smells in datasets?

Motivation: Alazba and Aljamaan (2021), Reis et al. (2022), Dewangan et al. (2022a) discussed Bagging and Boosting techniques having good performance for code smell detection, while a model that combines an EMBBC with feature selection and data balance is lacking in the literature. Therefore, four existing ensemble models (XGB, GB, ADA, RF), EMBBC, and ANN with a data balancing technique and feature selection method are applied to investigate the impact of code smell detection.

This question targets the evaluation of EMBBC (Ensemble model of bagging and two best-performing boosting techniques) in detecting code smells. This contribution is significant in understanding how this hybrid ensemble method can enhance detection performance.

RQ2 What are the software metrics that are crucial for code smell detection?

Motivation: Alazba and Aljamaan (2021), Reis et al. (2022), and Dewangan et al. (2022a) discussed the important software metrics for code smell datasets, while no study discussed the important software metrics for all used datasets separately. Therefore, this study finds software metrics crucial for each dataset to detect code smells.

This question finds the crucial software metrics for each used dataset. This contribution is significant in understanding which software metrics can enhance the detection performance.

RQ3 Does Recursive feature elimination with Cross-Validation (RFECV) feature selection technique affect performance accuracy?

Motivation: Alazba and Aljamaan (2021) used the Gain ratio, Mhawish and Gupta (2020) used the Genetic Algorithm (GA) -Naive Bayes and GA-CFS as feature selection technique and showed feature selection techniques importance in metrics selection, which is used for code smell detection. Therefore, in this study, RFECV is used as a feature selection technique to select essential features from the feature set and investigate how this technique affects model efficiency.

This question evaluates the impact of using RFECV as a feature selection method with the proposed model for code smell detection. This contribution is significant in understanding how feature selection can enhance detection performance.

RQ4 Does data balancing increase the performance of code smell detection?

Motivation: Guggulothu and Moiz (2019) and Gupta et al. (2019)

Table 1
Comparison of earlier approaches used to detect code smells.

Reference	Year	Smell	Technique	Classification type	Data Imbalance Handling	Feature selection	Num. of features	Ensemble Technique
Fontana et al. (2013)	2013	Blob Class, Feature envy, Data Class, Long Method	SMO, Decision Tree (j48), LibSVM, RF, Naive Bayes, Jrip	Binary	No	No	–	Yes (Bagging)
Amorim et al. (2016)	2015	Refused parent request, Swiss Army Knife, Anti Singleton, Large class, Blob Class, Long Method, Lazy Class, Long Parameter List, Class Data should be private, Message Chains, Complex Class	Decision Tree, Support Vector Machines, Bayesian Belief Networks	Binary	No	Yes (Genetic Algorithm)	–	No
Fontana et al. (2016)	2016	Blob Class, Feature Envy, Data Class, Long Method	SMO, LibSVM, Naive Bayes, Jrip, B-Jrip, RF, J48, B-J48, B-LibSVM, B-Random Forest, B-naive Bayes	Binary	Yes (stratified sampling)	No	–	Yes (Bagging and ensemble of the different methods with ADA)
White et al. (2016)	2016	Code Clone	recurrent neural network, recursive neural network	Binary	No	No	–	No
Kim (2017)	2017	Blob Class, Large Class, Lazy Class Parallel inheritance, hierarchies, Feature Envy, Data Class	Neural Network	Binary	No	No	–	No
Fontana and Zanoni (2017)	2017	Data Class, Blob Class, Feature Envy, Long method	SMO, Decision Tree, RF, Naive Bayes, Jrip, J48, LibSVM	Ordinal Severity levels	No	Yes	34,36,56, 59	Yes (Bagging and ensemble of the different methods with ADA)
Kaur et al. (2018)	2017	Blob Class, Feature Envy, Data Class, Long Method	SVM	Binary	No	No	–	No
Di Nucci et al. (2018)	2018	Blob Class, Long Method, Data Class, Feature Envy	SMO, LibSVM, J48, RF, Naive Bayes, Adaboost, Jrip	Binary	Yes (NA with different data balance)	Yes (Gain Ratio Feature Evaluation)	–	Yes (Bagging and ensemble of the different methods with ADA)
Liu et al. (2018)	2018	Feature Envy	Neural Network	Binary	No	No	–	No
Pecorelli et al. (2019)	2019	Blob Class, Long Method, Spaghetti Code, Complex Class, and Class Data should be private	J48, RF, Naive Bayes, SVM, JRip	Binary	Yes (SMOTE)	Yes, correlation-based feature selection	–	Yes (Bagging)
Guggulothu and Moiz (2019)	2019	Shotgun Surgery, Message Chaining	J48, C4.5, RF, Jrip, Naive Bayes, SMO, K-nearest neighbors	Binary	Yes (stratified random sampling)	No	–	Yes (Bagging)
Pecorelli et al. (2019)	2019	Blob Class, Long Method, Spaghetti Code, Complex Class, and Class Data should be private	J48, RF, Naive Bayes, SVM, JRip	Binary	Yes (SMOTE)	Yes, correlation-based feature selection (CFS)	–	Yes (Bagging)
Mhawish and Gupta (2019)	2019	Blob Class, Long Method, Feature Envy, and Data Class	Decision Tree	Binary	No	Yes (GA-based Naive Bayes, CFS)	(9, 11, 9, 8), (7, 7, 7, 8)	No
Gupta et al. (2019)	2019	Blob Class, Complex Class, No Low Memory Resolver, Long Method, Internal Getter/Setter, Swiss Army Knife, Member Ignoring Method, and Leaking Inner Class	ELM (Extreme Learning Machine) with three different kernel functions	Binary	Yes (downscale sampling, random sampling, and upscale sampling techniques)	Yes (Wilcoxon sign rank and cross-correlation analysis)	–	No
Kiyak et al. (2019)	2019	Feature Envy, Blob Class, Long Method, Data Class	Decision Tree, RF, Naive Bayes, SVM, Neural Network	Multi label Binary	No	Yes (a subset of the feature)	–	Yes (Bagging, Ensemble)
Jesudoss et al. (2019)	2019	Detector, Primitive Obsession Detector, Duplicated Code Detector, Bloated Code Detector, and Too Many Literal Detectors	SVM, RF	Binary	No	No	–	Yes (Bagging)
Mhawish and Gupta (2020)	2020	Data Class, Feature Envy, Blob Class, and Long Method	RF, GB, Decision Tree, Deep Learning, SVM, Multilayer perceptron	Binary	Yes(manual)	Yes (GA-based Naive Bayes, CFS)	–	Yes (Bagging, Boosting)

(*continued on next page*)

Table 1 (*continued*)

Reference	Year	Smell	Technique	Classification type	Data Imbalance Handling	Feature selection	Num. of features	Ensemble Technique
Barbez et al. (2020)	2020	Blob Class, Feature Envy	Neural Network, Smart Aggregation of Antipatterns Detectors	Binary	Yes (Custom loss function)	No	–	Yes (Boosting ensemble)
Guggulothu and Moiz (2020)	2020	Long method, Feature Envy	B-random Forest, RF, B-J48U, B-J48P, J48U	Binary	Yes (RAKEL and PS)	No	–	Yes (Bagging and ensemble of the different methods with ADA)
Gupta et al. (2021)	2021	Member Ignoring Method, Blob Class, Long Method, Leaking Inner Class, Internal Getter/Setter, No Low Memory Resolver, Complex Class, and Swiss Army Knife	Deep Learning model	Binary	Yes (SMOTE, Adaptive Synthetic (ADASYN) sampling methods)	Yes (Cross-Correlation analysis and Wilcoxon Sign Rank Test)	–	No
Draz et al. (2021)	2021	Blob Class, Feature Envy, Data Class, Functional Decomposition, Spaghetti Code, Lazy Class, Long method, Parallel Inheritance, and Long Parameter List	Whale Optimization algorithm, Search-based software engineering	Binary	No	No	–	No
Boutaib et al. (2021)	2021	Blob Class, Data Class, Feature Envy, Long Method, Duplicate Code, Long Parameter List, Spaghetti Code, and Functional Decomposition	ADIODE (Anti-pattern Detection and Identification using Oblique Decision tree Evolution)	Binary	Yes	Yes (using crossover and mutation)	–	No
Yadav et al. (2021)	2021	Blob Class, Data Class	Decision Tree	Binary	No	No	–	No
Dewangan et al. (2021)	2021	Blob Class, Feature Envy, Long Method, Data Class	Decision Tree, Multilayer perceptron, RF, Naive Bayes, Logistic Regression, and k-nearest neighbors (KNN)	Binary	No	Yes (Chi-squared, Wrapper based Feature Selection)	(10,12,12,12) (9,11,7,12)	Yes (Bagging)
Alazba and Aljamaan (2021)	2021	Blob Class, Switch Statement, Feature Envy, Data Class, Long Method, Long Parameter List	Decision Tree, Stochastic Gradient Descent, SVM, Linear Discriminant Analysis, Naïve Bayes, MLP, Gaussian Process, KNN, Logistic Regression	Binary	No	Yes (gain ratio)	27, 25, 36, 22, 30, 34	Yes (stacking ensemble)
Aljamaan (2021)	2021	Blob Class, Switch Statement, Long Method, Data Class, Long Parameter List, Feature Envy	Decision Tree, Logistic Regression, SVM, MLP, Stochastic Gradient Descent	Binary	No	Yes (gain ratio)	–	Yes (voting ensemble of the different models)
Kaur and Kaur (2021)	2021	Brain Method, Blob Class, message Chains, Shotgun Surgery, Dispersed coupling, Data Class	Ensembles Bagging and RF	Binary	No	Yes (CFS, Info gain Attribute Evaluator, Relief Attribute Eval (RAE))	–	Yes (Bagging)
Khleel and Nehéz (2022)	2022	Feature Envy, Blob Class, Long Method, Data Class	ANN, Deep convolutional neural networks	Binary	Yes (SMOTE)	Yes (wrapper-based methods, filter-based methods, embedded methods)	–	No
Dewangan et al. (2022b)	2022	Data Class, God Class, Switch statements , and a Long Parameter List	Logistic regression based on Principal component analysis, K-nearest neighbor based on Principal component analysis, Random Forest based on Principal component analysis, and Decision tree based on Principal component analysis	Binary	No	Yes, Principal component analysis (PCA)	8-12 components	Yes (Bagging)
Reis et al. (2022)	2022	Feature Envy, Blob Class, Long Method	J48, AdaBoost, SMO, Multilayer perceptron, RF, Naive Bayes	Binary	No	No	–	Yes (Boosting, Bagging)
Dewangan et al. (2022a)	2022	Data Class, Feature Envy, Blob Class, Long Method	ADA, Bagging, Max voting, GB, XGB, ANN,	Binary	Yes (SMOTE)	Yes (Chi-square feature extraction)	9, 12, 12, 12	Yes (Bagging, Boosting)

(*continued on next page*)

Table 1 (*continued*)

Reference	Year	Smell	Technique	Classification type	Data Imbalance Handling	Feature selection	Num. of features	Ensemble Technique
			Convolutional Neural Network					
Abdou and Darwish (2022)	2022	Data Class, Blob Class, Long Method, Feature Envy	J48, Radial Basis Functions neural network, JRIP, RF, Multinomial logistic regression, SMO, LIBSVM/SVR, Boosting, Naïve Bayes	Binary	Yes (SMOTE)	No	–	Yes (Bagging, Boosting)
Dewangan et al. (2023)	2023	Long parameter lists and Switch statements	AdaBoost, GB, and Max voting	Binary	Yes (SMOTE)	Yes (A wrapper-based feature selection)	10, 9	Yes (Bagging, Boosting)
Rao et al. (2023)	2023	Data Class, God Class, Feature Envy, and Long Method	K-nearest neighbor, Random Forest, Decision tree, Multi-layer Perceptron, and Logistic Regression	Binary	Yes (SMOTE)	Yes (PCA)	–	Yes (Bagging)
Dewangan et al. (2023)	2023	Data Class, God Class, Feature Envy, and Long Method	Logistic regression, RF, Decision Tree, XGB, ADA, GB	Binary	No	Yes (chi-square)	10, 10, 10,10	Yes (Bagging, Boosting)
Yadav et al. (2024)	2024	Long method and Feature envy	Binary Relevance, Label Powerset, and Classifier Chain, Random Forest, XGB, Decision Tree, GB, and Artificial Neural Network	Multi label Binary	No	Yes (chi-square)	7	Yes (Bagging, Boosting)
Proposed		Blob Class, Data Class, Long Parameter List, Switch Statement	XGB, GB, ADA, RF, EMBBC, ANN	Binary	Yes (SMOTE)	Yes (RFECV)	7, 7, 10, 8	Yes (EMBBC)

The cell having "- " values indicates that the author didn't study for the corresponding column.

presented the impact of data balancing techniques in code smell prediction. Therefore, the SMOTE data balancing technique is applied to examine this technique's effects and increase the accuracy of the ML model.

This question addresses the impact of data balancing techniques on the performance of code smell detection algorithms. Investigate how balancing the dataset can improve the proposed model's ability to detect minority class instances, which are often underrepresented.

This study offers EMBBC, a novel approach for identifying code smells using ensemble techniques. RFECV is applied for feature selection. The synthetic Minority Over-sampling Technique (SMOTE) approach is applied to balance the datasets. A 10-fold Cross-Validation is applied to enhance the suggested model's training.

The rest of the paper is organized as follows. Section 2 reviews the previous research studies related to code smell prediction. Section 3 explains the proposed model. Experiment results are presented in section 4. Section 5 includes a discussion and threats to validity. Finally, Section 6 concludes with future research directions.

2. Literature review

This section categorizes existing research into three sub-sections, each offering unique insights and potential for advancement. The first section delves into applying machine learning techniques for code smell detection, highlighting various approaches. The second section explores bagging or boosting techniques to identify code smells and analyzes their performance. The third section introduces the concept of ensemble models, a novel approach that combines bagging and boosting techniques and analyzes their effectiveness. Finally, we compare our proposed method with existing research, showcasing its novelty and potential to advance the field. This comprehensive review aims to inspire further exploration and innovation in machine learning-based code smell detection.

2.1. Machine learning-based code smell detection

Amorim et al. (2016) explored the efficacy of various machine learning techniques, including Decision Trees, Support Vector Machines (SVM), and Bayesian Belief Networks (BBN), for identifying code smells. Their methodology successfully derived rules for detecting specific code smells by integrating a Decision Tree with a Genetic Algorithm. This hybrid approach was applied to 12 types of code smells across four open-source Java projects: Eclipse, Mylyn, ArgoUML, and Rhino. Evaluation metrics such as Precision, Recall, and F-measure were used to assess performance, with the Decision Tree demonstrating superior results compared to the other methods in most scenarios, instilling confidence in the effectiveness of these techniques.

White et al. (2016) explored learning-based methods for detecting code clones, focusing on deriving representations of code fragments directly from code repositories. They used deep learning models, specifically recurrent neural networks (RNNs) and recursive neural networks (RvNNs), to identify code smells associated with code clones. In order to conduct their study, they manually evaluated 398 pairs of files and 480 pairs of methods in eight different real-world Java systems. The findings demonstrated that their learning-based approach is effective for clone detection, validating it as a viable technique for research in this area.

Kim (2017) developed a neural network-based system for detecting code smells, examining the correlation between object-oriented metrics and various types of code smells such as Blob Class, Large Class, Lazy Class, Parallel Inheritance Hierarchies, Feature Envy, and Data Class. The research demonstrated that these object-oriented metrics are significantly linked to the presence of code smells. Experimental results indicated that larger training datasets improved the model's predictive accuracy. Higher epochs and additional hidden layers enhanced the model's performance in identifying code smells.

Kaur et al. (2018) introduced the SVMCSD technique for identifying code smells using support vector machines. They applied this method to the ArgoUML and Xerces datasets, targeting code smells such as Blob

Class, Feature Envy, Data Class, and Long Method. Their findings indicate that SVMCSD surpasses DETEX (DETectionEXpert) in detecting code smells within system subsets. Furthermore, SVMCSD also identifies more code smells than DETEX when applied to entire systems.

Liu et al. (2018) introduced an innovative deep-learning methodology for identifying feature envy. To facilitate the training of their deep neural network, they developed a strategy to generate extensive labelled training datasets autonomously. This approach was tested on open-source applications with and without injected code smells. The evaluation across these scenarios demonstrated that their method substantially enhances the accuracy and efficacy of feature envy detection.

Mhawish and Gupta (2019) applied a machine learning-based decision tree approach for detecting code smells in software. Their study identified crucial metrics influencing the detection process. It implemented two genetic algorithm-based feature selection techniques, Naive Bayes and Correlation-based Feature Selection (CFS), alongside a grid search parameter optimization strategy. The results were impressive, with the GA_CFS method achieving an accuracy of 98.05% in predicting Data Class smells, 97.56% for God Class smells, and 94.31% for Long Method smells. Additionally, for Long Method detection, they achieved an accuracy of 98.38% using the GA-Naïve Bayes feature selection technique. These high accuracy rates provide reassurance of the reliability and effectiveness of these models in code smell detection. However, it is important to note that the study was limited to a specific set of code smells and may not be generalizable to all types of code smells.

Gupta et al. (2019) introduced a method for predicting code smells by extracting features from the source code. Their approach targeted eight types of code smells: Blob Class, Complex Class, No Low Memory Resolver, Long Method, Internal Getter/Setter, Swiss Army Knife, Member Ignoring Method, and Leaking Inner Class. They used the Extreme Learning Machine (ELM) classifier using three distinct kernel functions. They implemented a data sampling process to address the class imbalance issue, which included downscaling, random sampling, and upscaling techniques. Additionally, they used feature extraction to identify the most relevant features. Their method, incorporating deep learning, achieved an AUC accuracy ranging from 88.47% to 96.84%.

Gupta et al. (2021) proposed a methodology for predicting code smells. They extracted features from the source code across eight specific types: Member Ignoring Method, Blob Class, Long Method, Leaking Inner Class, Internal Getter/Setter, No Low Memory Resolver, Complex Class, and Swiss Army Knife. They used data sampling techniques such as SMOTE and Adaptive Synthetic Sampling to address class imbalance. They also used feature selection methods, including Cross-Correlation Analysis and the Wilcoxon Sign Rank Test, to identify the most relevant feature sets. Their approach, which incorporated deep learning, significantly improved the AUC accuracy from 88.47% to 96.84%.

Draz et al. (2021) proposed a search-based technique leveraging the Whale Optimization Algorithm to enhance the prediction of code smells. Their experimental evaluation, conducted on five open-source software projects, identified nine distinct types of code smells: Blob Class, Feature Envy, Data Class, Functional Decomposition, Spaghetti Code, Lazy Class, Long Method, Parallel Inheritance, and Long Parameter List. The technique achieved an average precision of 94.24% and a recall of 93.4%.

Boutaib et al. (2021) introduced the ADIODE (Anti-pattern Detection and Identification using Oblique Decision Tree Evolution) algorithm, an advanced machine-learning technique designed to improve the detection and identification of code smells, particularly in imbalanced datasets. The algorithm utilizes oblique decision trees (ODTs) evolved through an evolutionary algorithm (EA) to analyze a dataset of anti-patterns. Their results demonstrated that the F-measure ranged from 91.23% to 95.24%, and the Area Under the Curve (AUC) values were between 0.9273 and 0.9573, indicating strong performance in recognizing code smells.

Yadav et al. (2021) discussed a Decision Tree technique. They applied a five-fold Cross-Validation Grid search technique for hyperparameter tuning and got 97.62% accuracy in both Data Class and Blob Class code smell datasets.

Khleel and Nehéz (2022) proposed ANN, Deep convolutional neural networks-based code smell detection for Blob Class, Long Method, Data Class, and Feature Envy datasets. Wrapper, filter, and embedding techniques are used for feature selection, and SMOTE is a data-balancing technique. They obtained those oversampling techniques to perform better.

2.2. Bagging or boosting technique-based code smell detection

Fontana et al. (2013) conducted a comprehensive study on detecting code smells, focusing on four types: God Class, Data Class, Long Method, and Feature Envy. They applied a range of six machine learning classifiers, including SMO, LibSVM, Decision Trees (J48), Random Forest, Naive Bayes, and JRip, each with distinct hyperparameters. The evaluation was conducted on 74 software systems from the Qualitas Corpus Tempero et al. (2010) dataset.

Guggulothu and Moiz (2019) made a significant contribution to the field by developing a practical detection framework using multiple machine learning algorithms, such as J48, C4.5, Random Forest, JRip, Naive Bayes, SMO, and K-nearest neighbours. Their aim was to identify Shotgun Surgery and Message Chaining code smells, which are prevalent in real-world software systems. They used random stratified sampling on method instances to ensure balanced datasets. The results showed that tree-based classifiers outperformed others in terms of accuracy, F-measure, and area under the ROC curve, which have direct implications for software developers striving to improve code quality.

Pecorelli et al. (2019) conducted a comprehensive study that compared heuristic-based and machine-learning techniques for detecting code smells using metrics. They focused on five specific code smells: God Class, Spaghetti Code, Class Data Should Be Private, Complex Class, and Long Method. The study evaluated various machine learning algorithms (J48, Random Forest, Naive Bayes, Support Vector Machine, JRip) against DÉCOR (DEtection & CORrection2), a leading heuristic-based approach that formalizes the steps for identifying code and design smells. While the results of this meticulous study show that DÉCOR generally outperformed the machine learning models, it is important to note that the study had its limitations, such as the specific code smells it focused on, and the datasets used, which could be areas for future research.

Kiyak et al. (2019) developed a multi-label approach for identifying code smells by applying single-label classifiers such as Decision Tree, Random Forest, Naive Bayes, Support Vector Machine, and Neural Network. Their work focused on detecting four code smells: Feature Envy, Blob Class, Long Method, and Data Class. They applied multi-label classification techniques, including Classifier Chains and Label Combination, and their ensemble methods, Bagging ML and Ensemble ML. Impressively, their findings demonstrate the superior performance of the Random Forest algorithm as the base model.

Jesudoss et al. (2019) investigated the application of Support Vector Machine and Random Forest algorithms in code smell detection. In their approach, SVM is a classifier, while the Random Forest algorithm predicts data ranges. The study targets various code smells, including Detectors, Primitive Obsession Detectors, Duplicated Code Detector, Bloated Code Detector, and Too Many Literals Detector. Detecting these code smells aims to assist developers in writing code and enhancing code quality.

Mhawish and Gupta (2020) analyzed code smells using various machine learning techniques such as Random Forest, Gradient Boosting, Decision Tree, Deep Learning, Support Vector Machine, and Multilayer Perceptron (MLP), along with software metrics, to identify four specific code smells: Data Class, Feature Envy, Blob Class, and Long Method. They applied genetic algorithm-based feature selection to enhance the performance of these machine-learning models by selecting the most relevant features from each dataset. Additionally, they utilized parameter optimization via grid search to improve the accuracy of these

algorithms further. Their results showed that the Random Forest model achieved the highest accuracy, with 99.71% for the ORI_D dataset and 99.70% for the REFD_D dataset in detecting the Data Class smell.

Barbez et al. (2020) developed an advanced detection tool by integrating various detection methodologies into a system utilizing a boosting ensemble model. This model incorporated diverse techniques, including a Neural Network and a sophisticated aggregation method for anti-pattern detectors. The ensemble model effectively merged different strategies for identifying anti-patterns, specifically targeting Blob Class and Feature Envy code smells. Notably, their approach demonstrated superior performance compared to other ensemble methods.

Dewangan et al. (2021) developed a model using the essential metrics from all datasets; they used six machine learning-based models (Decision Tree, Multilayer perceptron, RF, Naive Bayes, Logistic Regression, and k-nearest neighbours (KNN)) to predict code smells (Blob Class, Feature Envy, Long Method, Data Class), hyperparameter tuning is performed using the Grid Search, and a Wrapper-based metrics and Chi-square feature selection method are utilized. They obtained 100% accuracy in the Long method code smell prediction.

Alazba and Aljamaan (2021) explore the effectiveness of using a heterogeneous stacking ensemble model for identifying code smells. Their study incorporates a range of machine learning classifiers, including Decision Tree, Stochastic Gradient Descent, Support Vector Machine, Linear Discriminant Analysis, Naïve Bayes, Multilayer Perceptron, Gaussian Process, k-nearest Neighbors, and Logistic Regression, to detect various code smells such as Blob Class, Switch Statement, Feature Envy, Data Class, Long Method, and Long Parameter List. They also apply a gain ratio for feature selection. The research evaluates how well a stacking ensemble constructed from these classifiers improves detection performance compared to individual classifiers for different code smell categories. Using logistic regression and SVM as meta-classifiers, the stacking ensemble demonstrated superior performance in detecting class-level and method-level code smells relative to the individual models in detecting most code smells.

Aljamaan (2021) assessed the efficacy of a Voting ensemble approach for identifying various code smells, including God Class, Data Class, Long Method, Feature Envy, Long Parameter List, and Switch Statements. Their ensemble method integrated five distinct base classifiers: Decision Trees, Logistic Regression, Support Vector Machines, Multilayer Perceptron, and Stochastic Gradient Descent. The ensemble's predictions were determined through soft voting, combining the base models' outputs. This Voting ensemble, reassuringly, demonstrated superior performance over the individual base classifiers in detecting each type of code smell.

I. Kaur and Kaur (2021) utilized the application of Ensemble Learning techniques in conjunction with Correlation-based Feature Selection on three open-source Java datasets to identify various code smells, including Brain Method, Blob Class, Message Chains, Shotgun Surgery, Dispersed Coupling, and Data Class. They applied Bagging ensembles and the Random Forest classifier, evaluating the approaches using four performance metrics: accuracy (P1), G-mean 1 (P2), G-mean 2 (P3), and F-measure (P4). Their findings indicated that the performance measures either remained stable or improved by incorporating feature selection and Ensemble Learning. Extensive validation on diverse datasets is necessary before standardizing these techniques due to concerns about diversity and reliability.

Dewangan et al. (2022b) used Logistic regression based on Principal component analysis (PCA_LR), K-nearest neighbor based on Principal component analysis (PCA_KNN), Random Forest based on Principal component analysis (PCA_RF), and Decision tree based on Principal component analysis (PCA_DT) to detect code smells and enhance performance accuracy effectively. The obtained accuracy was 99.97% and 94.05% in predicting Data Class and Long Parameter List code smells, respectively.

Reis et al. (2022) presented the Crowdsmelling technique, which utilizes collective intelligence for identifying code smells, with J48, RF, ADA, Sequential Minimal Optimization (SMO), Multilayer perceptron, and Naive Bayes to detect Blob Class, Feature Envy, Long Method code smells. They applied Bagging and Boosting techniques and discovered that Crowdsmelling is a practical approach. The obtained ROC for Blob Class detection using Naive Bayes is 0.896, and the ROC for Long Method detection using AdaBoostM1 is 0.870.

Dewangan et al. (2022a) reported a technique for detecting code smells using ensemble ML. They took Blob Class, Feature Envy, Long Method, and Data Class code smell datasets. Two deep learning models (ANN, Convolutional Neural Network (CNN)) and five ensemble learning strategies (ADA, Bagging, Max voting, GB, XGB) were used. In the Blob Class dataset, the Bagging technique and the Max Voting method achieved better performance in the Data Class dataset.

Abdou and Darwish (2022) addressed the crucial issue of severity classification for various code smells—Data Class, Blob Class, Long Method, and Feature Envy—by applying various machine-learning techniques. These included J48, Radial Basis Function (RBF) neural network, JRIP, Random Forest, Multinomial Logistic Regression, Sequential Minimal Optimization, LIBSVM/SVR, Boosting, and Naïve Bayes. To handle data imbalance, they utilized SMOTE for resampling. They also used the Local Interpretable Model-Agnostic Explanations (LIME) algorithm to interpret the predictions made by these models. Furthermore, they extracted prediction rules from the Projective Adaptive Resonance Theory (PART) algorithm to evaluate the effectiveness of software metrics in predicting code smells. Their experimental results demonstrated a significant improvement in the accuracy of the severity classification model compared to the baseline, with the ranking correlation between predicted and actual severity ranging from 0.92 to 0.97, as measured by Spearman's correlation coefficient.

Dewangan and Rao (2023) addressed detecting method-level code smells by examining datasets for long parameter lists and switch statements. They employed the SMOTE technique to balance class distribution and used a wrapper-based feature selection for optimal feature extraction. Three ensemble learning-based machine learning methods (AdaBoost, GB, and Max voting) were applied to these datasets. The models' performance was evaluated through five-fold cross-validation, considering metrics like precision, recall, F-measure, AUC Score, and accuracy. They achieved the highest accuracy of 97.12% using a max voting dataset for the long parameter list.

Rao et al. (2023) analyzed four datasets related to code smell severity—Data Class, God Class, Feature Envy, and Long Method to address the class imbalance in detecting code smell severity. They utilized the Synthetic Minority Oversampling Technique to balance the classes and employed principal component analysis for feature selection. The study applied five machine learning methods, K-nearest Neighbor, Random Forest, Decision Tree, Multilayer Perceptron, and Logistic Regression, to predict code smell severity. Their findings highlighted that Random Forest and Decision Tree achieved a high severity accuracy score of 0.99 for the Long Method code smell.

Dewangan et al. (2023) investigated the detection of code smell severity using four datasets: Data Class, God Class, Feature Envy, and Long Method. They applied four machine learning methods and three ensemble learning techniques, incorporating five-fold cross-validation, Chi-square feature selection, and parameter optimization (grid and random search) to enhance performance. Their study found that the XGBoost model, using Chi-square-based feature selection, achieved an accuracy of 99.12% for the long method dataset; ensemble learning outperforms traditional machine learning in assessing code smell severity.

Yadav et al. (2024) proposed a rule-based method utilizing Binary Relevance, Label Powerset, and Classifier Chain techniques with decision trees and ensemble algorithms. They enhanced the performance using chi-square feature selection and validated it through 10-fold cross-validation and parameter tuning. They achieved a 99.54% Jaccard accuracy with the Classifier Chain and Decision Tree combination.

2.3. Ensemble model of bagging and boosting techniques-based code smell detection

Fontana et al. (2016) conducted a subsequent study of their previous work Fontana et al. (2013). In this study, they ensembled the AdaBoost method with an ML model for code smell detection. They used stratified sampling for data balancing. Their findings revealed that the Random Forest and j48 were the most effective in detecting these code smells.

Fontana and Zanoni (2017) expanded their analysis to evaluate the severity of code smells. They introduced a classification system with four categories: no smell, non-severe smell, smell, and severe smell, replacing the binary classification of positive or negative. They applied two feature selection techniques to enhance the dataset: a low variance filter and a high linear correlation filter.

Di Nucci et al. (2018) conducted a meticulous replication of a previous study to address and highlight specific limitations identified in the initial experiment. They painstakingly maintained the same experimental setup, ensuring every detail was replicated, and tested their model on an identical set of 74 systems. A key modification in their approach was the adaptation of the training dataset for the classifiers, which included a broader range of realistic instances for each code smell type. Their findings, which indicated a decline in performance accuracy compared to the original study, provide a clear direction for further refinement and investigation in machine-learning techniques for detecting code smells.

Di Nucci et al. (2018) reported some issues in Fontana et al., (2016): (1) Each dataset represents one type of code smell, which does not represent the real-world situation. (2) the distribution of smelly and non-smelly instances is imbalanced. The obtained average accuracy was 73%–75%, and the result was not generalized to expose the efficacy of the ML model for code smell detection. Guggulothu and Moiz (2020) claimed that after excluding disparity cases in Fontana et al., (2016), ML models have the potential to achieve good accuracy in real scenarios.

Guggulothu and Moiz (2020) made significant strides in detecting code smells through a multi-label classification framework. They applied a diverse range of classifiers, including B-random Forest, Random Forest, B-J48U, B-J48P, and J48U, to identify "Long Method" and "Feature Envy" smells, where the B- prefix indicates the use of AdaBoost ensemble. Their study yielded remarkable results, with the B-J48 pruned algorithm achieving an accuracy of 99.10% on the Feature Envy dataset and the Random Forest algorithm reaching an accuracy of 95.90% on the Long Method dataset.

Our proposed approach stands out from the abovementioned methodologies by introducing the novel hybrid ensemble model (EMBBC), a unique combination of Bagging with the best-performing boosting techniques, along with feature selection and data balancing. This

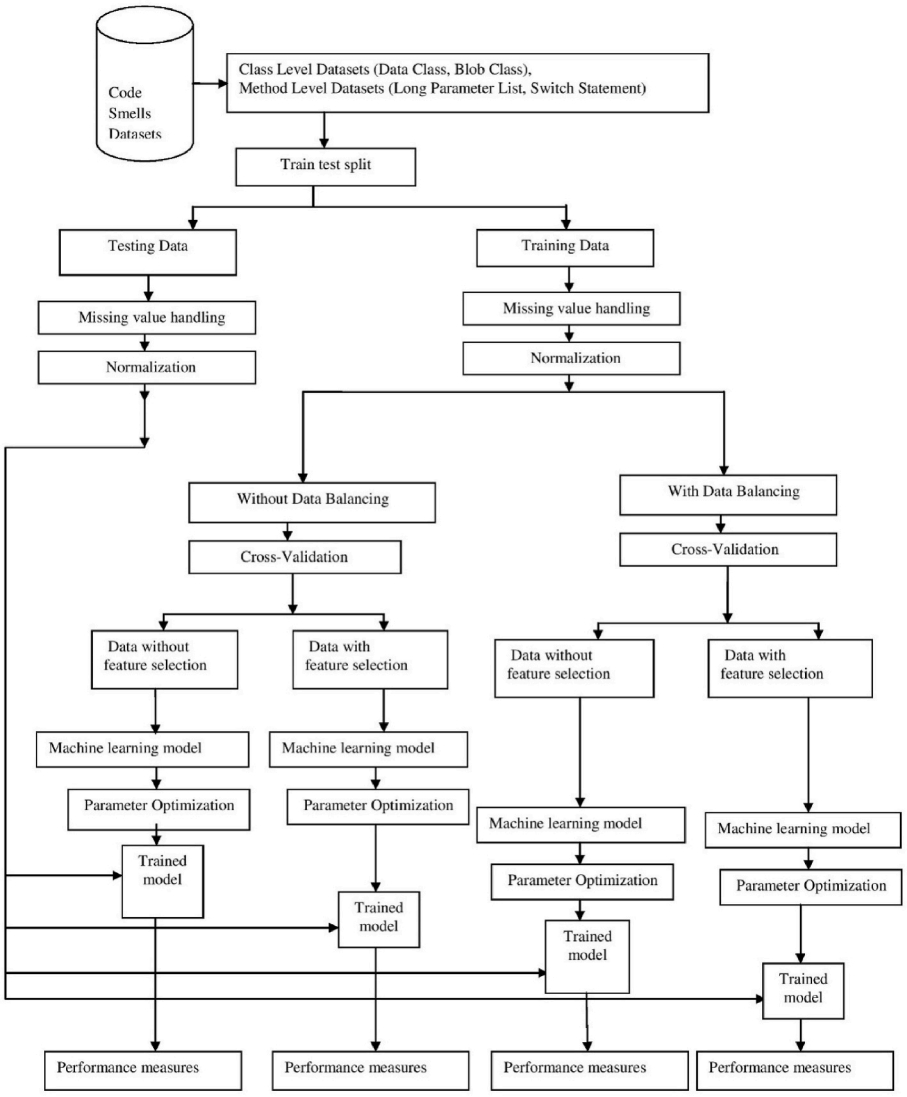

Fig. 1. Framework of experimental work.

approach aims to harness the strengths of both methods, with Bagging reducing variance, avoiding overfitting, and Boosting enhancing model accuracy. By integrating these benefits, we strive to enhance the overall detection performance. To the best of our knowledge, this specific combination of techniques for detecting code smells has not been previously documented in the literature, underscoring its innovative contribution to the field. Table 1 provides a comprehensive overview of the various tools and techniques for code smell detection.

3. Research framework

Fig. 1 illustrates the sequence of actions used to evaluate the code smell prediction model. This study considered the publicly available four code smell datasets: Blob Class, Data Class, Long Parameter List, and Switch Statement (Fontana et al., 2016). The dataset is split into a training and a testing set in the following stage, and several preprocessing techniques were used to smooth the data for subsequent processing. After this, hyperparameter tuning and training data to ML models are applied. Our experiment applied various ML models, including XGB, GB, ADA, Random Forest, ANN, and our proposed EMBBC model (described below in section 3.8). XGB, GB, and ADA are used as the boosting technique, and Random Forest is used as the bagging technique, which is integrated to construct the EMBBC. Therefore, the individual components (bagging and boosting techniques) are described first before the EMBBC model is presented in detail. Testing data is provided to assess the proposed model's performance. The following experimental setup is used for the whole experiment in this study.

- **Hardware - Processor:** Intel Core i3, **RAM:** 6 GB, **Storage:** 250 GB SSD, **Operating System:** Windows 10 Pro.
- **Software - Development Environment:** Jupyter Notebook 6.5.2, **Libraries and Packages:** Scikit-learn 1.2.2, skmultilearn 0.2.0, Pandas 1.5.3, NumPy 1.26.4, Keras 2.12.0, TensorFlow 2.12.0.

3.1. Datasets description

This study uses four publicly available datasets: Blob Class (Fontana and Zanoni, 2017), Data Class (Fontana and Zanoni, 2017), Long Parameter List (Alazba and Aljamaan, 2021), and Switch Statement (Alazba and Aljamaan, 2021), built by (Fontana et al., 2016). These datasets are not just randomly chosen but carefully selected to align with the study's objectives. These datasets differ in their properties and potential effects on software quality. This comprehensive analysis allows us to understand the impact of code smells across various aspects of software development. While previous research has extensively studied these code smells, there is a need for a focused examination of the distinct impact of certain smells on specific software systems. By focusing on these four categories, we explore uncharted territories, providing fresh perspectives and making valuable contributions to the overall knowledge base.

Table 2 shows that the Blob Class and the Data Class datasets are examples of Class Level datasets. Two datasets, Long Parameter List and Switch Statement, belong to Method Level datasets. There are 66 features in Class-level datasets, four of which are insignificant to the proposed experiment.

Similarly, there are 62 features in Method Level datasets, of which five are irrelevant to the proposed investigation. The irrelevant features are 'IDType', 'project', 'package', and 'complextype' in Class Level datasets and 'IDMethod', 'project', 'package', 'complextype', and 'method' in Method Level datasets. These features are merely identifiers, project names, package names, methods names, etc. and do not provide any valuable information for understanding code quality or detecting code smells. The non-significant features are discarded manually. Features of datasets are object-oriented metrics. In each dataset, 420 records exist, of which 140 belong to a smelly and 280 belong to a non-smelly class.

3.2. Train test split

A train test split is essential for better division of datasets. The experimental dataset is divided into training and testing data using the train-test split method with a test size of 30%. In each dataset, 30% of the data are used to test the model's effectiveness, while the remaining 70% are used for training. A crucial aspect of this process is avoiding data leakage, where the testing data are completely unknown from the proposed models. This ensures a more accurate evaluation of the model's performance. Table 3 shows that the proposed model is trained on 294 data in Class-level datasets and then tested on 126 data. The same division is applied to Method-level datasets.

3.3. Missing value handling

The presence of missing values can significantly hamper the results of an ML model, underscoring the urgency of their handling. Various approaches are available to deal with missing values, including filling them with zero, mean value imputation, k-nearest Neighbor imputation, and more. As discussed in section 3.2, the training set is completely unknown from a testing set to avoid data leakage. The process of handling missing values involves replacing each missing value in the training set with the column mean value in the training set, and each missing value in the testing set with the column mean value in the training set. Table 4 provides a comprehensive view of the number of features, instances, total number of values, missing values and their percentage for each code smell dataset, highlighting the importance of this step in the data preprocessing phase.

3.4. Normalization

When there are several attributes, but the scales on which they are measured are varied, it may be challenging to build a good data model,

Table 3
Size of training and testing data.

Code smells (Level wise)	Num. of Training data	Num. of Testing data
Class Level Datasets	294	126
Method Level Datasets	294	126

Table 2
Dataset description.

Level	Code smell	Feature in original Datasets	The feature used in the proposed experiment work	Independent Feature	Dependent Feature	Num. of records	Num. of smelly records	Num. of non-smelly records
Class Level Datasets	Blob Class	66	62	61	1	420	140	280
	Data Class	66	62	61	1	420	140	280
Method Level Datasets	Long Parameter List	62	57	56	1	420	140	280
	Switch Statement	62	57	56	1	420	140	280

Table 4
Overview of datasets and missing values.

Code smell Datasets	The feature used	Num. of records	Total num. of values	Num. of missing values	Missing values (%)
Blob Class	62	420	26040	76	0.29
Data Class	62	420	26040	75	0.29
Long Parameter List	57	420	23940	114	0.48
Switch Statement	57	420	23940	184	0.77

which points to the need for normalization. This analysis uses the Min-Max normalization method to convert the dataset's feature values to the range 0–1. This ensures that all attributes are on the same scale, making it easier to build a good data model. The changes x of an attribute A is represented by the expression, where x' represents the new value of x (Mhawish and Gupta, 2020).

$$(x)' = \frac{(x - \min(A))}{(\max(A) - \min(A))} \quad (1)$$

3.5. Data balancing

The experimental datasets have one dependent feature, consisting of two classes: one with many instances and the other with few. This scenario indicates that the datasets are imbalanced; because of this, the model may deviate from actual performance.

SMOTE is applied to handle this dataset imbalance problem. In every dataset, there are fewer instances of the smelly class than those of the non-smelly class. SMOTE (Synthetic Minority Over-sampling Technique) is a widely used method for addressing the class imbalance problem in datasets, which can adversely affect the performance of machine learning algorithms. The technique generates synthetic samples from the minority class to balance the dataset. For instance, if we have a dataset with 100 instances, 10 of which belong to the minority class, SMOTE will generate additional instances for the minority class to make it equal to the majority class. Specifically, SMOTE selects a random instance from the minority class and identifies its k-nearest neighbours in the feature space. SMOTE randomly chooses one of its k-nearest neighbours for each selected instance and generates a synthetic sample by interpolating between the feature vectors of the selected instance and its neighbor. This interpolation is achieved by selecting a random point along the line segment joining the two instances. This process helps to create a more balanced class distribution, thereby improving the performance of classifiers by providing them with more representative training data (Chawla et al., 2002).

In this experiment, the process is applied in two ways: first, without the data balancing technique, and second, with the data balancing technique, in each dataset.

3.6. Cross-validation

It is essential to train the proposed model better; a better-trained model helps to perform better. In this experiment, a ten-fold Cross-Validation approach is utilized. The training and testing sets are created from the entire dataset; as previously discussed, Cross-Validation is used only in the training set. In cross-validation, ten portions of training data are used, nine of which are for training the proposed model and one portion for testing the proposed model to test whether training data is appropriately trained for the model or not. This whole process repeats ten times, and in each iteration, it gives accuracy. The final accuracy score is obtained using the mean of accuracy values (majored in each iteration). This rigorous cross-validation process instils confidence in the model's reliability and performance.

3.7. Feature selection

Feature selection increases the model's learning capabilities, removing unrelated data and using only the most appropriate data. RFECV is applied to select essential features from datasets. In this technique, two parameters are used: first, the model (By which essential features are extracted) and second, how many features are selected from the set of features (Awad and Fraihat, 2023; Mustaqim et al., 2021).

In this study, various ML models are applied differently in each dataset (i) without feature selection (with all features of the dataset) and without data balancing technique, (ii) after applying feature selection technique and without data balancing approach, (iii) with all feature and with data balancing technique, and (iv) after applying feature selection technique and with data balancing technique. In all these combinations, the selected features are different; the features with the best performance are mentioned in Table 5. Each of the chosen features are explained in the appendix section Table A1.

3.8. Applied models

Tree-based models have shown superior performance in code smell detection (Mhawish and Gupta, 2020; Dewangan et al., 2022a) explored various ensemble techniques and found them effective in detecting code smells (Fontana et al., 2016). applied different ML models with ADA and achieved promising results in code smell detection. To further advance this research and address a gap in the literature, this study introduces a novel ensemble model combining Bagging and the two best-performing Boosting techniques. The study utilizes four decision tree-based ensemble ML models (XGB (Dewangan et al., 2022a), GB (Dewangan et al., 2022a), ADA (Dewangan et al., 2022a), and RF (Fontana and Zanoni, 2017)), an EMBBC, and ANN (Grossi and Buscema, 2007) to predict code smells.

- **EMBBC**

This study used a Bagging technique (RF) and the two best-performing Boosting techniques (from XGB, GB, and ADA) to create an ensemble model. In order to choose the final class, a technique called "soft voting" is applied; models offer a percentage indicating the likelihood that a given set of data belongs to which target class. Fig. 2 shows a block diagram of EMBBC. It is an innovative method to improve code smell detection performance using a hybrid ensemble model. This method combines the effectiveness of Boosting techniques with the well-known benefits of Bagging, which decrease variance and avoid overfitting. By combining the two approaches, which leverage the strengths of each ensemble method, the model's robustness and reliability are significantly enhanced, providing a solid foundation for code smell detection.

An ANN model simulates the operations of the human brain using computations. In this study, an Adam optimizer with two hidden layers are taken. Relu, sigmoid is utilized as the activation function, while binary cross entropy is applied as the loss function.

3.9. Hyperparameter optimization

Each ML model contains a few hyperparameters, the values of which impact the model's performance. Each model has a different optimum hyperparameter value. Testing each method's possible combination of hyperparameter values is necessary to determine the appropriate hyperparameters. The hyperparameter value for each strategy is calculated using the grid Search algorithm. Table 6 includes a list of the hyperparameters utilized in each model, along with their starting, ending, and steps used.

The total number of combinations for the XGBoost model:

- Learning Rate: 101 values (0, 0.01, 0.02, …, 1.00)

Table 5
List of features from each dataset selected using the RFECV method.

Dataset	Selected software metrics(features)	Num. of features	Percentage of selected features (%)
Data Class	'WOC_type', 'NOAM_type', 'num_final_static_attributes', 'RFC_type', 'AMW_type', 'num_static_attributes', 'number_package_visibility_attributes', 'NOA_type', 'number_protected_visibility_attributes', 'num_final_attributes'	10	16.39
Blob Class	'NOI_package', 'WMCNAMM_type', 'CFNAMM_type', 'AMWNAMM_type', 'LCOM5_type', 'LOCNAMM_type', 'number_final_methods'	7	11.45
Long Parameter List	'NOP_method', 'CBO_type', 'LOCNAMM_type', 'NOMNAMM_project', 'WOC_type', 'LOC_project', 'NOCS_package'	7	12.5
Switch Statement	'CFNAMM_type', 'MAXNESTING_method', 'LOC_method', 'LOC_package', 'FANOUT_type', 'CYCLO_method', 'AMWNAMM_type', 'CINT_method'	8	14.28

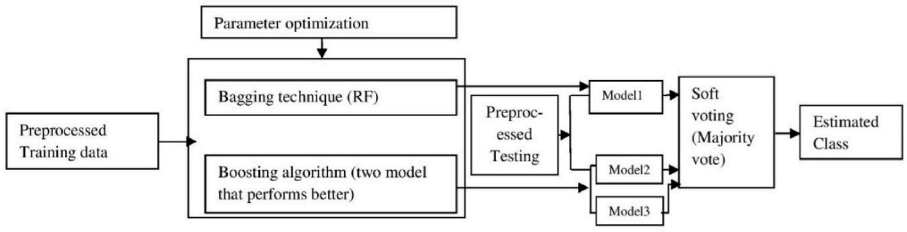

Fig. 2. Block diagram of the EMBBC without feature selection.

Table 6
ML model with its hyperparameters, start and end value of hyperparameters, and steps taken for each hyperparameter.

ML model	Hyperparameters	Start value	End value	Step
XGB	Learning rate	0	1	0.01
	Max Depth	1	10	1
	Number of trees	1	100	10
GB	Learning rate	0	2	0.01
	Max Depth	1	10	1
	Number of trees	1	75	1
ADA	Learning rate	0	1	0.1
	Number of trees	1	100	1
RF	Max Depth	1	20	1
	Number of trees	1	20	1
	Criterion	Information gain, Gini impurity		
	Class weight	Balanced, Balanced subsample		
ANN	Learning rate	0	0.1	0.01
	Activation function	Relu, Sigmoid		

- Max Depth: 10 values (1, 2, …, 10)
- Number of Trees: 10 values (1, 11, 21, …, 91)

Total combinations for XGBoost = 101 (Learning Rate) * 10 (Max Depth) * 10 (Number of Trees) = 10100 combinations.

Similarly, we have calculated total combinations for other models.

Total combinations for Gradient Boosting (GB) = 201 (Learning Rate) * 10 (Max Depth) * 75 (Number of Trees) = 150750 combinations.

Total combinations for ADA Boosting (ADA) = 101 (Learning Rate) * 100 (Number of Trees) = 10100 combinations.

Total combinations for Random Forest (RF) = 20 (Max Depth) * 20 (Number of Trees) * 2 (Criterion) * 2 (Class weight) = 1600 combinations.

Total combinations for Artificial Neural Network (ANN) = 11 (Learning Rate) * 2 (Activation function) = 22 combinations.

The grid search cross-validation took approximately 10–20 min to tune each model's hyperparameter.

3.10. Performance evaluation

Six performance measures are used to assess each experiment's success. It is evaluated based on its accuracy (Mhawish and Gupta, 2020), precision (Mhawish and Gupta, 2020), recall (Mhawish and Gupta, 2020), f1-score (Mhawish and Gupta, 2020), AUC_ROC_Score (Dewangan et al., 2022a), and MCC (Dewangan et al., 2022a). The confusion matrix was computed; it is used to hold the real and expected data that the code smell model had identified.

4. Experimental result and discussion

The first research question (RQ1) focuses on the performance of ML models: XGB, GB, ADA, RF, EMBBC, and ANN. Accuracy, precision, recall, f1-score, AUC_ROC_Score, and MCC offer a detailed analysis of the models' performance across four diverse datasets, making this experiment a valuable resource for our research community.

We conducted a comprehensive evaluation of the experiment's findings. This included a comparative study and statistical analysis for each dataset. The results, presented in Tables 7 and 8, 9 and 10, were measured using accuracy, precision, recall, f1-score, AUC_ROC_Score, and MCC across four datasets. These findings are significant as they provide a comprehensive understanding of the performance of different ML models. Table 11 provides a comparative performance overview of the different models.

Table 7 shows the performance of different ML models. This table shows the performance without feature selection and without data balancing techniques, ensuring the accuracy of the results.

The ADA model, with its impressive 98.41% accuracy, 98.77% f1-score, 97.83% AUC_ROC_Score and 96.6% MCC, outperformed in the Blob Class dataset. The EMBBC model excelled in the Data Class dataset with its exceptional 99.21% accuracy, 99.44% f1-score, 98.68% AUC_ROC_Score, and 98.12% MCC. The XGB model, with its significant improvement in the Long Parameter List dataset, demonstrated its potential with 97.62% accuracy, 95.24% f1-score, 97.33% AUC_ROC_Score and 93.67% MCC. The ADA model, with its strong performance in the Switch Statement dataset, showcased its capabilities with an accuracy of 91.27%, 87.36% f1-score, 93.75% AUC_ROC_Score, and 82.38% MCC.

Table 8 shows the performance of different ML models with feature selection and without data balancing techniques.

The XGB model, with its impressive 99.21% accuracy, 99.38% f1-score, 98.91% AUC_ROC_Score, and 98.29% MCC, stands out as the best model for predicting the Blob Class dataset. Similarly, the EMBBC model's stellar performance, with 98.41% accuracy, 98.84% f1-score,

Table 7
Performance of different ML models using accuracy, precision, recall, f1-score, AUC_ROC_Score, and MCC for each dataset.

Datasets	ML models	Performance measures					
Blob Class		Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	AUC_ROC_Score (%)	MCC (%)
	XGB	98.41	97.56	100	98.76	97.83	96.6
	GB	97.62	96.38	100	98.16	96.74	94.92
	ADA	98.41	97.56	100	98.77	97.83	96.6
	RF	97.62	96.38	100	98.16	96.74	94.92
	EMBBC (XGB + ADA + RF)	98.41	97.56	100	98.76	97.83	96.6
	ANN	98.41	98.75	98.75	98.75	97.35	96.1
Data Class	XGB	99.21	98.85	100	99.42	98.75	98.17
	GB	98.41	98.84	98.84	98.84	98.17	96.34
	ADA	98.41	98.84	98.84	98.84	98.17	96.34
	RF	98.41	97.73	100	98.85	97.5	96.35
	EMBBC (XGB + GB + RF)	99.21	98.88	100	99.44	98.68	98.12
	ANN	98.41	98.84	98.84	98.84	98.17	96.34
Long Parameter List	XGB	97.62	93.75	96.77	95.24	97.33	93.67
	GB	95.24	85.71	96.77	90.91	95.76	88
	ADA	96.03	93.33	90.32	91.80	94.11	89.21
	RF	94.44	92.86	83.87	88.14	92.85	91.75
	EMBBC (XGB + ADA + RF)	96.82	96.55	90.32	93.33	94.63	91.34
	ANN	95.24	96.30	83.87	89.66	94.62	86.97
Switch Statement	XGB	88.89	73.08	100	84.44	92.05	78.39
	GB	89.68	74.51	100	85.39	92.61	79.69
	ADA	91.27	77.55	100	87.36	93.75	82.38
	RF	91.27	81.40	92.10	86.42	91.51	80.36
	EMBBC (GB + ADA + RF)	88.89	74.00	97.37	84.09	92.03	78.26
	ANN	91.27	81.82	92.31	86.75	91.56	80.6

Table 8
Performance of different ML models with RFECV feature selection technique for each dataset.

Datasets	ML models	Performance measures					
Blob Class		Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	AUC_ROC_Score (%)	MCC (%)
	XGB	99.21	98.76	100	99.38	98.91	98.29
	GB	97.62	96.38	100	98.16	96.74	94.92
	ADA	98.41	97.56	100	98.76	97.83	96.6
	RF	97.62	96.38	100	98.16	96.74	94.92
	EMBBC (XGB + ADA + RF)	98.41	97.56	100	98.76	97.83	96.6
	ANN	97.62	96.51	100	98.22	96.42	94.9
Data Class	XGB	97.62	96.63	100	98.29	96.25	94.54
	GB	97.62	96.63	100	98.29	96.25	94.54
	ADA	97.62	96.63	100	98.29	96.25	94.54
	RF	97.62	96.63	100	98.29	96.25	94.54
	EMBBC (XGB + GB + RF)	98.41	98.84	98.84	98.84	98.17	96.34
	ANN	97.62	97.73	98.85	98.28	95.58	92.53
Long Parameter List	XGB	96.82	96.55	90.32	93.33	94.63	91.34
	GB	96.03	100	86.84	92.96	93.42	90.65
	ADA	96.82	97.62	93.18	95.35	94.78	91.47
	RF	96.82	96.55	90.32	93.33	94.63	91.34
	EMBBC (XGB + ADA + RF)	96.82	96.55	90.32	93.33	94.63	91.34
	ANN	95.24	91.67	95.65	93.62	95.36	87.45
Switch Statement	XGB	92.06	79.17	100	88.37	94.32	83.77
	GB	91.27	77.55	100	87.36	93.75	82.38
	ADA	91.27	77.55	100	87.36	93.75	82.38
	RF	89.68	75.51	97.37	85.06	91.87	78.83
	EMBBC (XGB + ADA + RF)	92.06	79.17	100	88.37	94.32	83.77
	ANN	90.48	80.95	89.47	85.00	90.19	78.26

98.17% AUC_ROC_Score, and 96.34% MCC in the Data Class dataset, underscores its superiority. The ADA model's strong showing in the Long Parameter List dataset with 96.42% accuracy, 95.35% f1-score, 94.78% AUC_ROC_Score, and 91.47% MCC further validate its effectiveness. Lastly, the XGB and EMBBC models' competitive performance in the Switch Statement dataset, with an accuracy of 92.06%, 88.37% f1-score, 94.32 AUC_ROC_Score, and 83.77% MCC, solidifies their position as top contenders.

Table 9 shows the performance of different ML models without feature selection and with data balancing techniques.

The EMBBC model emerges as the top performer, demonstrating its robustness with 99.21% accuracy, 99.44% f1-score, 98.68% AUC_ROC_Score, and 98.12% MCC in predicting the Data Class dataset. Notably, all models, except GB and ANN, show promising results in predicting the Blob Class dataset. The XGB model, in particular, showcases its prowess by improving performance in the Long Parameter List dataset with 97.62%, the highest accuracy along with EMBBC, the highest, 95.24% f1-score, 97.33% AUC_ROC_Score, and 93.67% MCC. The GB model, on the other hand, shines in the Switch Statement dataset with an accuracy of 92.06%, 88.37% f1-score, 94.32% AUC_ROC_Score, and 83.77% MCC.

Table 10 presents a comprehensive overview of the performance of various ML models. These models were tested with feature selection and data balancing techniques, providing a nuanced understanding of their capabilities.

The best model for predicting the Data Class dataset is the EMBBC model, with 99.21% accuracy, 99.41% f1-score, 99.41% AUC_ROC_Score, and 98.22% MCC. ADA, RF, and EMBBC models better predict in

Table 9
Performance of different ML models with SMOTE data balancing technique for each dataset.

Datasets	ML models	Performance measures					
Blob Class		Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	AUC_ROC_Score (%)	MCC (%)
	XGB	99.21	98.76	100	99.38	98.91	98.29
	GB	98.41	97.56	100	98.76	97.83	96.6
	ADA	99.21	98.76	100	99.38	98.91	98.29
	RF	99.21	98.76	100	99.38	98.91	98.29
	EMBBC (XGB + ADA + RF)	99.21	98.76	100	99.38	98.91	98.29
	ANN	97.62	100	96.38	98.16	95.37	92.23
Data Class	XGB	99.21	98.85	100	99.42	98.75	98.17
	GB	99.21	98.85	100	99.42	98.75	98.17
	ADA	98.41	98.84	98.84	98.84	98.17	96.34
	RF	98.41	97.73	100	98.85	97.5	96.35
	EMBBC (XGB + GB + RF)	99.21	98.88	100	99.44	98.68	98.12
	ANN	98.41	100	97.62	98.80	98.81	96.53
Long Parameter List	XGB	97.62	93.75	96.77	95.24	97.33	93.67
	GB	96.03	88.24	96.77	92.31	93.23	90.57
	ADA	96.82	96.55	90.32	93.33	94.63	91.34
	RF	96.82	93.55	93.55	93.55	94.45	91.37
	EMBBC (XGB + ADA + RF)	97.62	96.67	93.55	95.08	96.25	93.53
	ANN	96.03	100	83.87	91.23	91.94	89.26
Switch Statement	XGB	91.27	77.55	100	87.36	93.75	82.38
	GB	92.06	79.17	100	88.37	94.32	83.77
	ADA	91.27	77.55	100	87.36	93.75	82.38
	RF	91.27	81.40	92.10	86.42	91.51	80.36
	EMBBC (GB + ADA + RF)	92.06	80.43	97.37	88.10	98.28	83.46
	ANN	92.06	91.43	82.05	86.49	97.8	83.23

Table 10
Performance of different ML models with RFECV feature selection technique and SMOTE data balancing technique for each dataset.

Datasets	ML models	Performance measures					
Blob Class		Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	AUC_ROC_Score (%)	MCC (%)
	XGB	98.41	97.56	100	98.76	97.83	96.6
	GB	98.41	97.56	100	98.76	97.83	96.6
	ADA	99.21	98.76	100	99.38	98.91	98.29
	RF	99.21	98.76	100	99.38	98.91	98.29
	EMBBC (XGB + ADA + RF)	99.21	98.76	100	99.38	98.91	98.29
	ANN	96.82	100	95.18	97.53	97.59	93.32
Data Class	XGB	98.41	97.73	100	98.85	97.5	96.35
	GB	97.62	96.63	100	98.29	96.25	94.54
	ADA	97.62	97.70	98.84	98.27	96.92	94.48
	RF	97.62	96.63	100	98.28	96.25	94.54
	EMBBC (XGB + GB + RF)	99.21	100	98.82	99.41	99.41	98.22
	ANN	98.41	97.78	100	98.88	97.37	96.25
Long Parameter List	XGB	96.82	97.62	93.18	95.35	94.78	91.47
	GB	96.82	100	90.91	95.24	97.03	92.87
	ADA	96.82	95.45	95.45	95.45	96.51	93.02
	RF	97.62	93.75	96.77	95.24	97.33	93.67
	EMBBC (XGB + ADA + RF)	97.62	100	93.18	96.47	95.45	97.44
	ANN	96.03	97.56	90.91	94.12	94.84	91.26
Switch Statement	XGB	90.48	90.48	65.52	76.00	80.32	68.22
	GB	88.89	75.56	91.89	82.93	92.01	78.22
	ADA	91.27	86.21	78.12	81.97	86.93	76.39
	RF	90.48	80.49	89.19	84.62	90.1	77.96
	EMBBC (XGB + ADA + RF)	91.27	77.08	100	87.06	93.82	82.19
	ANN	92.86	88.46	79.31	83.64	91.9	78.32

Table 11
Comparative performance of different ML models for each dataset.

Datasets	Blob Class		Data Class		Long Parameter List		Switch Statement	
ML models	Accuracy (%)	F1-score (%)	Accuracy (%)	F1-score (%)	Accuracy (%)	F1-score (%)	Accuracy (%)	F1-score (%)
XGB	99.21	99.38	99.21	99.42	97.62	95.24	92.06	88.37
GB	98.41	98.76	99.21	99.42	96.82	95.24	92.06	88.37
ADA	99.21	99.38	98.41	98.84	96.82	95.45	91.27	87.36
RF	99.21	99.38	98.41	98.85	97.62	95.24	91.27	86.42
EMBBC	99.21	99.38	99.21	99.44	97.62	96.47	92.06	88.37
ANN	98.41	98.75	98.41	98.88	96.03	94.12	92.86	83.64

the Blob Class dataset. The EMBBC model performed better in the Long Parameter List dataset with 97.62%, the highest accuracy with 96.47% f1-score, 95.45% AUC_ROC_Score, and 97.44% MCC. The ANN model performs better in the Switch Statement dataset with the highest accuracy of 92.06% with 83.64% f1-score, 91.9% AUC_ROC_Score, and 78.32% MCC.

Table 11 provides a comprehensive comparison of different ML models for two Class-level and two Method-level datasets. The results highlight the significance of Class-level code smells, which consistently perform better than method-level code smells. The EMBBC model, with its high accuracy across most datasets, further reinforces this finding. The only exception is the Switch Statement dataset, where the ANN model achieves the highest accuracy of 92.86% and an f1-score of 83.64%. This exceptional performance provides a unique perspective on the ML models' performance.

4.1. ML models performance stability

This section aims to examine how stable the performance of the ML model is when detecting various types of code smells. Table 12 presents an overview of the results obtained from the ML model, highlighting the best and worst performing ones for each type of code smell. Table 10 is used to determine the best and worst-performing models. If the model's accuracy score was within the top 25% of scores for a given code smell type, it was considered one of the best performers. Conversely, if a model accuracy score was within the bottom 25%, it was deemed one of the worst performers. In order to determine these thresholds, the third quartile (Q3) of accuracy scores (also known as the 75th percentile) and the first quartile (Q1) of accuracy scores (also known as the 25th percentile) are computed for each code smell category. ML model with accuracy scores above the third quartile is considered among the best performers, while those below the first quartile are considered among the worst performers (Mcgill et al., 1978). In our experiment, we utilized the Excel QUARTILE function to calculate the first quartile (Q1) and the third quartile (Q3) values. The general formula for Q1 and Q3 are:

$$\text{First Quartile (Q1)} = \frac{n+1}{4}\ th\ item \tag{2}$$

$$\text{Third Quartile (Q3)} = \frac{3(n+1)}{4}\ th\ item \tag{3}$$

Where n represents the number of accuracy scores.

Table 12 uses a "B" symbol to denote the best-performing model in detecting a specific code smell and a "W" symbol to denote the worst-performing model. The results demonstrate that EMBBC was among the most successful in detecting code smells, while the GB model was the least accurate. These findings highlight that a model's success in identifying a single code smell does not guarantee its effectiveness in detecting other code smells. For instance, the XGB model, while exceptional in detecting Data class smells, should be used with caution as it performs poorly in detecting Blob class, Long Parameter List, and Switch Statement smells.

Table 12
Best and worst ML model for each code smell dataset based on their accuracy scores.

ML models	Datasets				Number of datasets	
	Blob Class	Data Class	Long Parameter List	Switch Statement	Best	Worst
XGB	W	B	W	W	1	3
GB	W	W	W	W	0	4
ADA	B	W	W	B	2	2
RF	B	W	B	W	2	2
EMBBC (XGB + ADA + RF)	B	B	B	B	4	0
ANN	W	B	W	B	2	2
Q3	99.21	98.41	97.42	91.27	–	–
Q1	98.41	97.62	96.82	90.48	–	–

B- Best performing ML models, W- Worst performing ML model Q3 – third quartile, Q1-first quartile, - No data.

RQ1: To answer this question, this paper rigorously addresses the question by employing four existing ensemble ML models based on Decision Trees, ANN, and a unique method that combines the two best-performing Boosting techniques with an RF model (a Bagging technique). The findings are robust: EMBBC consistently outperforms in three datasets (Blob Class, Data Class, and Long Parameter List), and while it falls slightly short in the Switch Statement dataset, it still proves to be the best model overall. The comprehensive and reassuring discussion of the essential software metrics in the appendix section Table A1 further enhances the confidence in the research process.

4.2. Effect of feature selection in code smell detection

To answer RQ3: The accuracy of four existing ensemble models, the EMBBC and ANN models, are compared with and without applying the feature selection technique, as shown in Fig. 3.

The performance of the XGB model is improved for Blob Class and Switch Statement datasets after applying feature selection, while it performs better without feature selection for Data Class and Long Parameter List datasets, underscoring the importance of this technique in optimizing model performance.

The GB model performs better for the Long Parameter List and Switch Statement datasets, and no changes have been observed in the Blob Class dataset, while it performs better without applying the feature selection in Data Class datasets.

The ADA model shows improved results in the Long Parameter List dataset, and no changes have been observed in the Blob Class and Switch Statement datasets, while it performs better without applying feature selection in the Data Class datasets.

The RF model registers an improvement in performance for the Long Parameter List dataset, and no changes have been observed in the Blob Class dataset, while it performs better without feature selection for the Data Class and Switch Statement datasets.

EMBBC provides improved results in the Switch Statement dataset, and no changes have been observed in the Blob Class and Long Parameter List datasets, while it performs better without applying the feature selection in Data Class datasets.

ANN model performance has no changes in the Long Parameter List dataset, while it performs better without applying the feature selection in the Blob Class, Data Class, and Switch Statement datasets.

Table 13 shows the performance difference between the selected feature and all features using accuracy. In the Blob Class dataset, the XGB model with feature selection shows a positive difference of 0.80%. In the Long Parameter List dataset, GB, ADA, and RF models with feature selection show a positive difference of 0.79%, 0.79%, and 2.38%, respectively. In the Switch Statement dataset, XGB, GB, and EMBBC models with feature selection show a positive difference of 3.17%, 1.59%, and 3.17%, respectively.

RQ2: The RFECV feature selection technique is applied to answer this question. Table 5 mentions the software metrics for each dataset with the best performance. Table A1, the appendix section, explains each of the chosen software metrics.

RQ3: The RFECV feature selection technique is applied to answer this question. It selects essential methods and class metrics from the datasets. After applying the feature selection, the results reveal that the performance accuracy of the Blob Class dataset increased by 0.79% using the XGB model. The performance of the model varies from dataset to dataset. The Data Class dataset shows better performance without applying feature selection.

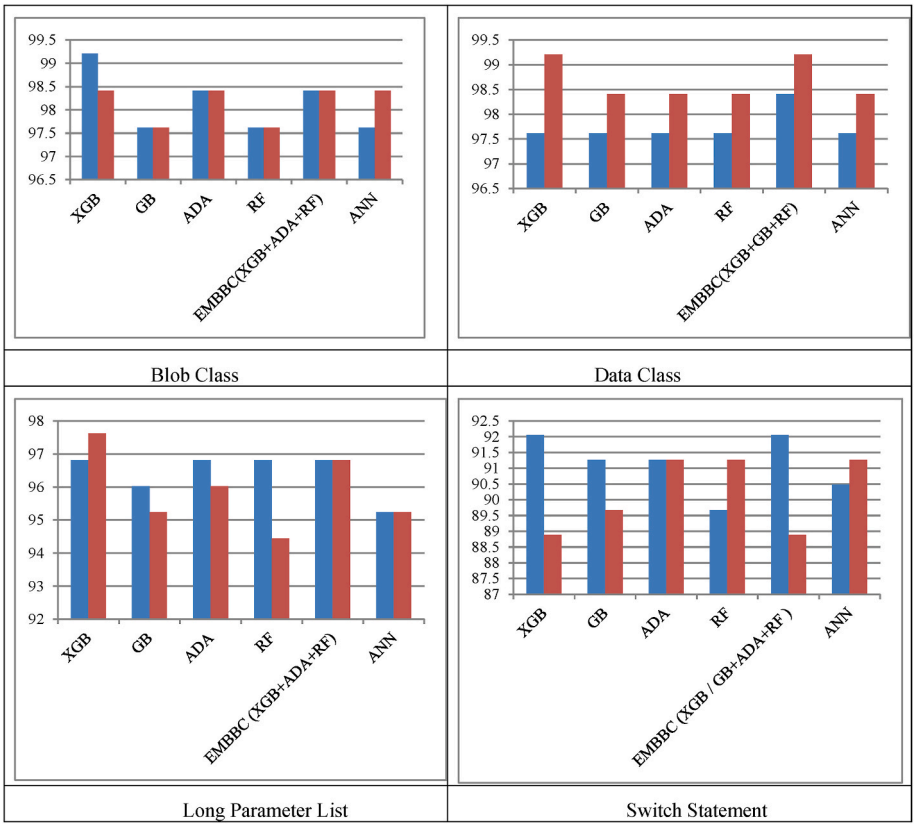

Fig. 3. Comparison graph of multiple models for various datasets with and without feature selection.

Table 13
With feature selection vs. Without feature selection performance difference.

	ML models					
Datasets	XGB	GB	ADA	RF	EMBBC	ANN
Blob Class	+0.80	0	0	0	0	−0.79
Data Class	−1.59	−0.79	−0.79	−0.79	−0.80	−0.79
Long Parameter	−0.80	+0.79	+0.79	+2.38	0	0
Switch Statement	+3.17	+1.59	0	−1.59	+3.17	−0.79

4.3. Effect of data balancing in code smell detection

To comprehensively address **RQ4**, we meticulously conducted the simulation results with and without data balancing for four existing ensemble models, the EMBBC and ANN models, as illustrated in Fig. 4. This rigorous approach allowed us to compare the performance accuracy of each model robustly.

The XGB model, while showing significant improvements for the Switch Statement and Blob Class datasets, did not exhibit any noticeable changes in the Data Class and Long Parameter List datasets, highlighting the model's limitations in these scenarios.

The GB model exhibits improved performance for all four datasets, showcasing its versatility and effectiveness.

The ADA model provides more accurate results in the Blob Class, Long Parameter List, and Switch Statement datasets, while no changes have been observed in the Data Class dataset.

The RF model shows improved results in the Blob Class and Long Parameter List datasets, while no change has been noticed in the Data Class and Switch Statement datasets.

The EMBBC provides more accurate results in the Blob Class, Long Parameter List, and Switch Statement datasets, while no changes have been observed in the Data Class dataset.

ANN improves the performance in the Long Parameter List dataset, and no changes have been observed in the Data Class dataset. It performs better without applying data balancing to the Blob Class and Switch Statement datasets.

Table 14 clearly illustrates the performance difference between data balancing and without data balancing using accuracy. Notably, in the Blob class dataset, the XGB, GB, ADA, RF, and EMBBC models with data balancing demonstrate a significant positive difference of 0.80%, 0.79%, 0.80%, 1.59%, and 0.80%. The Data class dataset with the data balancing GB model shows a positive difference of 0.80%. In the Long Parameter List dataset, the GB, ADA, RF, EMBBC, and ANN models with data balancing reveal a positive difference of 0.79%, 0.79%, 2.38%, 0.80%, and 0.79%, respectively. In the Switch Statement dataset, the XGB, GB, EMBBC, and ANN models with data balancing exhibit a positive difference of 2.38%, 2.38%, 3.17%, and 0.79%. These results highlight the substantial impact of data balancing on model performance, affirming the consistent and reliable performance of the GB model with data balancing across all datasets. In contrast, the ANN model shows promising results in the Long Parameter List dataset.

RQ4: Our research question, RQ4, focused on applying the SMOTE data balancing technique in training data. To answer this, we applied the technique and compared the performance accuracy before and after. The results were compelling, showing that the performance accuracy either increased or remained unchanged after data balancing in almost all cases. Notably, after data balancing, the Blob Class and Long Parameter List datasets achieved 99.20% and 98.41% accuracy, respectively, underscoring the positive influence of the SMOTE data balancing technique.

4.4. Effect of feature selection with data balancing in code smell detection

Fig. 5 depicts the accuracy comparison for four existing ensemble models, the EMBBC and ANN models, with data balancing and feature selection and without data balancing and feature selection.

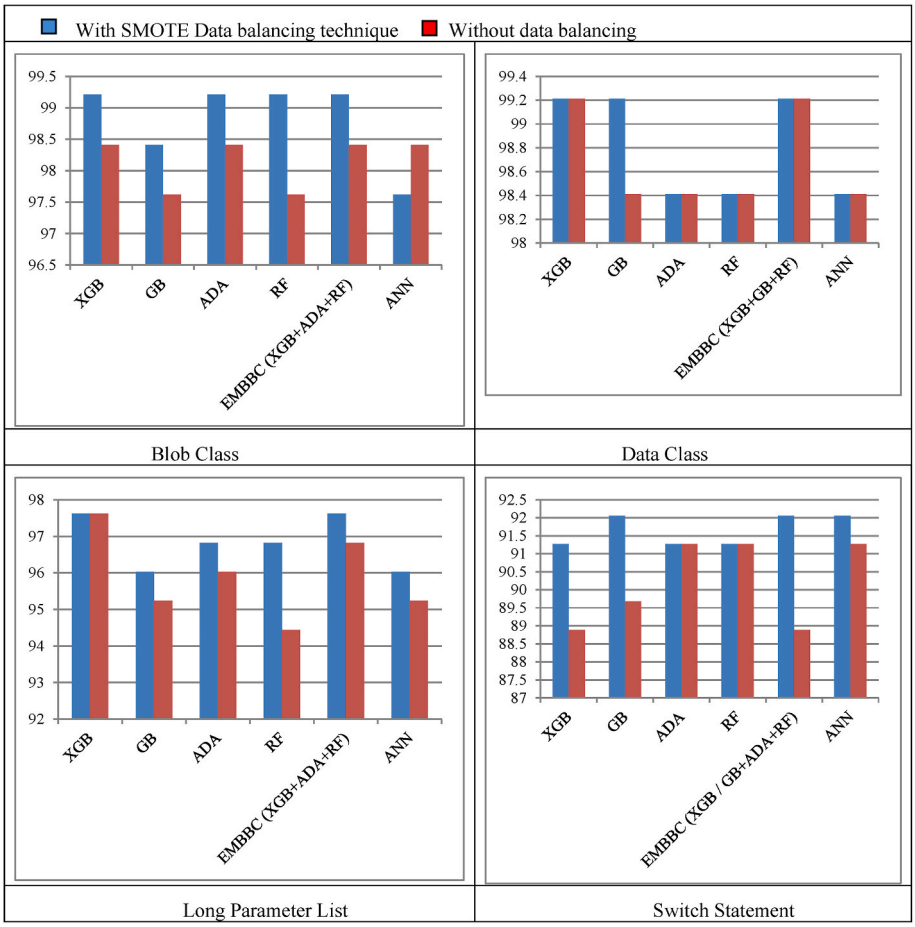

Fig. 4. Comparison graph of multiple models for various datasets with and without data balancing.

Table 14
With data balancing vs without data balancing, the performance difference.

	ML models					
Datasets	XGB	GB	ADA	RF	EMBBC	ANN
Blob Class	+0.80	+0.79	+0.80	+1.59	+0.80	−0.79
Data Class	0	+0.80	0	0	0	0
Long Parameter List	0	+0.79	+0.79	+2.38	+0.80	+0.79
Switch Statement	+2.38	+2.38	0	0	+3.17	+0.79

The XGB model notably enhances the performance of the Switch Statement dataset, demonstrating the potential for significant improvements. Interestingly, no change has been noticed for the Blob Class and Long Parameter List datasets, suggesting the need for further investigation. However, it is worth noting that the XGB model performs better without applying data balancing and with all features for the Data Class dataset, indicating the potential for enhanced accuracy.

The GB model improves the performance for the Blob Class and Long Parameter List datasets, and no change has been noticed in the case of the Switch Statement dataset. It performs better without applying data balancing and with all features in the Data Class dataset.

The ADA model provides more accurate results for Blob Class and Long Parameter List datasets, and no change has been noticed in the case of the Switch Statement dataset. It performs better without applying the data balancing technique and with all features in the Data Class dataset.

The RF model shows improved results in Blob Class and Long Parameter List datasets. It performs better without applying the data balancing technique and with all features in the Data Class and Switch Statement datasets, hinting at its potential for enhanced accuracy.

The EMBBC provides more accurate results in the Blob Class, Long Parameter List, and Switch Statement datasets, while no changes have been observed in the Data Class dataset.

The ANN performance is improved in the Long Parameter List dataset, and no changes have been observed in the Data Class dataset. It performs better without applying the data balancing technique and with all features in Blob Class and Switch Statement datasets.

Table 15 showcases the performance difference between all ML models, highlighting the positive impact of applying feature selection and data balancing techniques with and without. A positive difference indicates an improvement in the model's performance when these techniques are applied. In the Blob class dataset, GB, ADA, RF, and EMBBC models with feature selection and data balancing show a positive difference of 0.79%, 0.80%, 1.59%, and 0.80%. In the Long Parameter List dataset, GB, ADA, RF, EMBBC, and ANN models with feature selection and data balancing show a positive difference of 1.58%, 0.79%, 3.18%, 0.80%, and 0.79%, respectively. In the Switch Statement dataset, XGB, EMBBC, and ANN models with feature selection and data balancing show a positive difference of 1.59%, 2.38%, and 1.59%.

4.5. Comparison of the proposed model

The results of the proposed model are compared with other researchers' findings in Table 16. This research study applied four existing ensemble ML models, EMBBC, and ANN model for four datasets: Blob Class, Data Class, Long Parameter List, and Switch Statement (Fontana et al., 2016). These datasets were selected due to their relevance to machine learning.

The suggested method demonstrates its efficiency by achieving

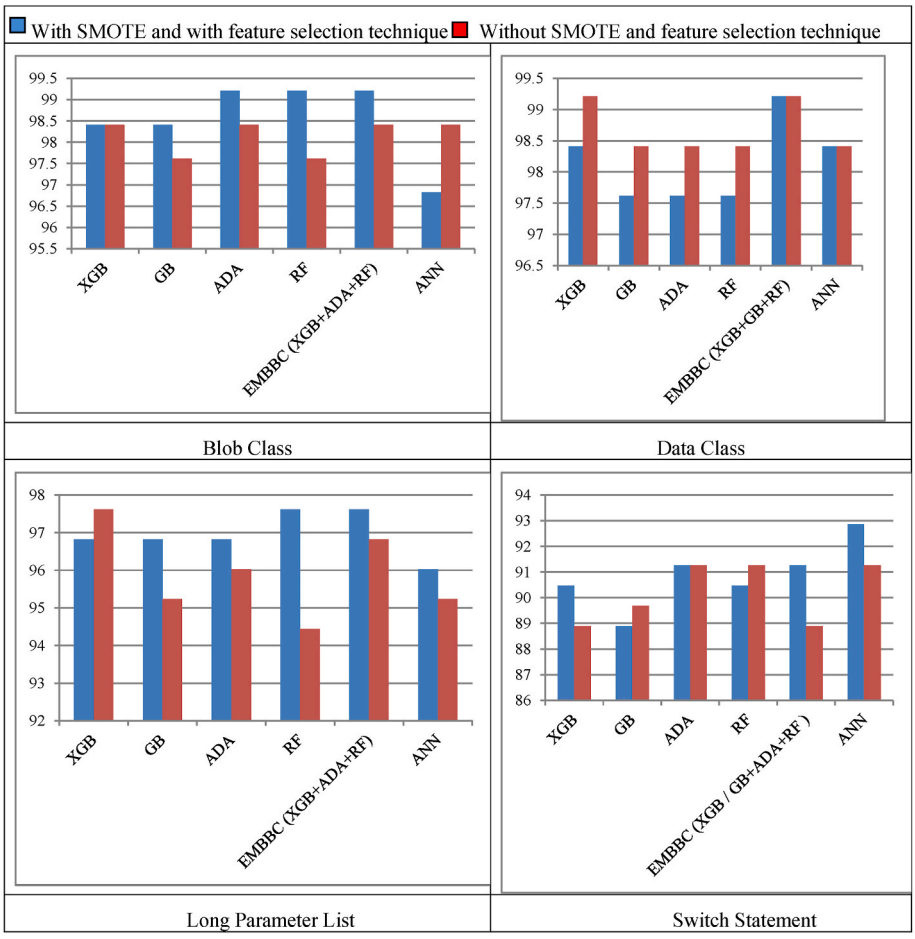

Fig. 5. Comparison graph of multiple models for various datasets before feature selection and data balancing with, after feature selection and data balancing.

Table 15
With feature selection and data balancing vs. without feature selection without data balancing, performance difference.

	ML models					
Datasets	XGB	GB	ADA	RF	EMBBC	ANN
Blob Class	0	+0.79	+0.80	+1.59	+0.80	−1.59
Data Class	−0.80	−0.79	−0.79	−0.79	0	0
Long Parameter List	−0.80	+1.58	+0.79	+3.18	+0.80	+0.79
Switch Statement	+1.59	−0.79	0	−0.79	+2.38	+1.59

99.21% accuracy on the Blob Class dataset with just seven features using EMBBC. This is a remarkable feat considering that (Dewangan et al., 2022a) secured the best result among the mentioned references with 99.24% accuracy using the Bagging technique with 12 features. The proposed model's ability to work with fewer features and still maintain a high level of accuracy is a testament to its efficiency, and it also significantly reduces computation power, making it a superior choice.

The proposed model achieves 99.21% accuracy on the Data Class dataset with ten features using EMBBC, while the Max voting method (Dewangan et al., 2022a) obtained the most favourable outcome of the listed references and was able to get 100% accuracy with nine features.

In the Long Parameter List dataset, the proposed model achieves 97.62% performance accuracy with seven features using EMBBC, while (Alazba and Aljamaan, 2021) secured the highest accuracy among all listed references with 92.50% accuracy using the GP (Gaussian Process) model with 22 features.

In the Switch Statement dataset, the proposed model achieves 92.86% accuracy with eight features using the ANN model, while (Alazba and Aljamaan, 2021) obtained the best result among listed references with 88.89% accuracy using the GP model with 25 features.

It is evident that the proposed model outperforms other authors in three datasets (Blob Class, Switch Statement, and Long Parameter List) with fewer features. This is due to the innovative use of an ensemble method with RFECV feature selection and 10-fold cross-validation, which helps to identify the most informative features and prevent overfitting. The ANN model, in particular, achieves the highest performance accuracy of 92.86 % in the Switch Statement dataset. Furthermore, it is worth noting that only a few authors have worked on the Long Parameter List and Switch Statement datasets, indicating a potential for more studies in the future. This underscores the promising potential of the proposed model for future research and development.

4.5.1. *Statistical study for models*

Within-subjects Analysis of Variance (ANOVA), also known as repeated measures ANOVA, was conducted to analyze the models. The results, presented in Table 17, show the changes in mean performance score over ten independent runs. For the Blob Class, Data Class, Long Parameter List, and Switch Statement datasets, the p-values were 0.0309, 0.0044, 0.0360, and 0.0053, respectively. These p-values indicate the probability of observing the data if the null hypothesis (that the mean performance score of models is equal) is true. In our case, each p-value was less than 0.05. This leads us to reject the null hypothesis and conclude that the mean performance scores of the models are indeed different, a significant finding in our study.

Our study involved the analysis of several datasets, including the Blob Class and Switch Statement datasets (Table 10). The models successfully detected the Blob Class dataset, as indicated by the high

Table 16
Performance comparison of proposed work with other author's results.

Year	Reference	Dataset used	Blob Class			Data Class			Long Parameter List			Switch Statement		
			Technique used	Accuracy (%)	No of features	Technique used	Accuracy (%)	No of features	Technique used	Accuracy (%)	No of features	Technique used	Accuracy (%)	No of features
2016	Fontana et al. (2016)	Qualitas Corpus Tempero et al. (2010)	Naïve Bayes	97.55	–	B-J48 Pruned	99.02	–	–	–	–	–	–	–
2018	Di Nucci et al. (2018)	Qualitas Corpus Tempero et al. (2010)	RF and J48	~83.00	9	RF and J48	~83.00	7	–	–	–	–	–	–
2020	Mhawish and Gupta (2020)	Fontana et al. (2016)	GBT	98.48	9, 7	RF	99.70	9, 7	–	–	–	–	–	–
2021	Alazba and Aljamaan (2021)	Fontana et al. (2016)	Stack-SVM	97.00	27	Stack-LR	98.92	36	GP	92.50	22	GP	88.89	25
2021	Dewangan et al. (2021)	Fontana et al. (2016)	RF	98.21	10,9	RF	99.74	12, 12	–	–	–	–	–	–
2021	Aljamaan (2021)	Fontana et al. (2016)	Voting ensemble	96.88	NA	Voting ensemble	97.45	NA	Voting ensemble	91.83	NA	Voting ensemble	87.81	NA
2022	Dewangan et al. (2022a)	Fontana et al. (2016)	Bagging	99.24	12	Max Voting	100	9	–	–	–	–	–	–
	Proposed Results	Fontana et al. (2016)	EMBBC/ AdaBoost/RF	99.21	7	EMBBC	99.21	10	EMBBC	97.62	7	ANN	92.86	8

The cell having "-" values indicates that the author didn't study for the corresponding datasets.

Table 17
Repeated measures ANOVA test results for Blob Class, Data Class, Long Parameter List, and Switch Statement datasets.

	ANOVA Test			
Datasets	F Value	Num DF	Den DF	Pr > F
Blob Class dataset	2.7278	5.0000	45.0000	0.0309
Data Class dataset	3.9875	5.0000	45.0000	0.0044
Long parameter list dataset	2.6316	5.0000	45.0000	0.0360
Switch Statement dataset	3.8719	5.0000	45.0000	0.0053

detection accuracy scores. However, the Switch Statement dataset presented a significant challenge. The models struggled to achieve high detection accuracy scores, as evidenced by the calculated first and third-quartile values. These values underscore the complexity and intricacy of the task, highlighting the challenges faced by the models in this particular dataset.

Our study used several models, including GB, ADA, and EMBBC. When applied to the Blob Class, the Long Parameter List, and the Switch Statement datasets, we found that these models showed significant performance improvement after applying feature selection (Fig. 3). The performance was evaluated using precision, recall, and the F1-score. Additionally, almost all models achieved good results after applying SMOTE, as shown in Fig. 4. This underscores the effectiveness of these techniques in enhancing model performance.

After observing different combinations of experiments with feature selection technique, it can be said that software metrics such as RFC_type, AMW_type, and num final attributes are proven to be crucial for identifying Data Class code smell, NOI_package, AMWNAMM_type, and WMCNAMM_type software metrics are helpful for spotting Blob Class code smell. AMWNAMM_type, CYCLO_method, LOC_method, and FANOUT_type metrics are essential for determining Switch Statement code smell, and NOP_method, CBO_type, and NOMNAMM_project software metrics are crucial in order to identify Long Parameter List code smell.

After applying SMOTE and feature selection, the EMBBC model achieved the best performance in all the datasets, while other models did not have good results for all the datasets. The challenge of data imbalance is important to address because a dataset with an uneven distribution of classes can cause machine learning models to favour the more populous classes, leading to biased predictions. SMOTE, a data balancing technique, addresses this by oversampling the minority class, creating a more balanced dataset. On the other hand, feature selection provides important features from the set of features; this reduced feature set takes less time and effort in computation. After taking fewer features and applying the data balancing technique, the proposed model performs better than other code smell dataset models.

Combining the Bagging Boosting techniques improves generalizability by using ensemble approaches to combine the capabilities of different base models. This allows it to capture a wider range of patterns in the data and tackles the intricacies of code smell prediction jobs more successfully.

Our research underscores the practical implications for code smell prediction performance. We advocate for the use of ensemble approaches, particularly the two best-performing Boosting and Bagging with feature selection and data balancing methods. By integrating different models with feature selection, software developers can achieve better outcomes than relying on individual models alone. Ensemble approaches consolidate weak learners to create a more stable prediction model resistant to overfitting.

5. Threats of validity

This study includes limitations that might impact the proposed investigation and how to address them. The following part discusses these threats in terms of their internal, external, construct, and

conclusion validity.

5.1. Internal validity

- **Feature Selection and Dataset Composition:** The proposed research is limited to four code smell datasets (two code smells belong to the category of Method Level code smells, and two code smells belong to Class Level code smells) provided by Fontana et al. (2016). These datasets comprise 420 instances each, with 280 non-smelly and 140 smelly instances. The diversity and representativeness of these datasets could be improved, potentially affecting the internal validity. To mitigate this, Recursive Feature Elimination with Cross-Validation (RFECV) is applied to select the most relevant features, and 10-fold Cross-Validation is used to validate the training data.
- **Data Imbalance and Normalization:** The SMOTE technique addresses data imbalance, and Min-Max normalization is used to normalize the datasets. However, different combinations of techniques might yield better accuracy, posing a threat to internal validity.

5.2. External validity

- **Generalizability to Industrial Situations:** The results of this study may not be directly applicable to industrial projects due to the use of datasets from Fontana et al. (2016), which may not fully represent real-world software. Future research will aim to replicate the study using industrial datasets to enhance external validity.

5.3. Construct validity

- **Relevance of Software Metrics:** The datasets used contain various software metrics, some of which may differ in importance and correlation. This variability could impact the construct validity of the study, as the chosen metrics might not comprehensively represent the code smells under investigation.

5.4. Conclusion validity

- **Accuracy and Technique Combinations:** While efforts have been made to ensure robust findings through techniques like the Ensemble model, RFECV, SMOTE, and 10-fold Cross-Validation, there remains a possibility that alternative combinations of these methods could lead to different conclusions. Thus, the validity of the conclusion might be influenced by the specific methodological choices made in this study.

6. Conclusion

Machine Learning (ML) based models were applied in this study. In this research, we looked at the four code smells datasets, the Blob Class, the Long Parameter List, the Data Class, and the Switch Statement, created by Fontana et al. Splitting each dataset into a training and a test set is accomplished with the hold-out approach, keeping each set separate to prevent data leaks. The proposed model is trained using 10-fold Cross-Validation to improve accuracy. Grid Search CV hyperparameter tuning is applied to fine-tune the model. Many methods exist for detecting code smell, such as Bagging and Boosting, while a model that incorporates an EMBBC with feature selection and data balance needs to be improved in the research studies.

A series of experiments were conducted using six ML models to fill this gap. In the Switch Statement dataset, the ANN model achieves 92.86% of the highest performance accuracy and 83.64% of the f1-score. The EMBBC achieved the highest accuracy of 99.21%, 97.62%, and 99.21%, with f1-score of 99.38%, 96.47%, and 99.41% for the Blob Class, Long Parameter List, and Data Class datasets, respectively. After applying data balancing, performance accuracy is either increased or equal to the prior instance except for the ANN model for the Blob Class. Blob Class achieved 99.21% accuracy and improved the performance after data balancing.

Similarly, Feature selection provides better performance accuracy in most cases. Blob Class and Switch Statement datasets' accuracy is increased by 0.80% and 3.17%, respectively, after feature selection. Using fewer features, the proposed model achieved good accuracy, which will also reduce the computation time and effort.

Different ML models play an essential role in efficiently predicting various code smell datasets. The XGB model excels for one dataset, while the ADA, RF, and ANN models perform best for two datasets each. However, the EMBBC model stands out, achieving the highest accuracy in all four datasets. It performs particularly well in the Blob Class, Data Class, and Long Parameter datasets, slightly less so in the Switch Statement dataset. This consistent performance of the EMBBC model instils confidence in its application for code smell detection.

Our research has identified crucial software metrics for predicting code smell. For the Data Class code smell, RFC_type, AMW_type, and num final attributes are essential. Similarly, for the Blob Class code smell, NOI_package, AMWNAMM_type, and WMCNAMM_type software metrics play a significant role. Understanding and utilizing these metrics will equip researchers and developers with the necessary tools for effective code smell detection.

Switch Statement code smell can be predicted with AMWNAMM_type, CYCLO_method, LOC_method, and FANOUT_type software metrics, and NOP_method, CBO_type, and NOMNAMM_project software metrics are crucial in order to detect Long Parameter List code smell.

This study reveals a promising trend when comparing the proposed model's results with existing research. The proposed model consistently outperforms most current models, indicating its potential for significantly improving code smell detection. This positive outcome fosters optimism and hope for the future of code smell detection research.

The study conducted a comprehensive performance evaluation of the proposed model using a range of evaluation metrics. These metrics include accuracy, precision, recall, f1-score, AUC_ROC_Score, and MCC. Each of these metrics provides a different perspective on the model's performance, allowing for a thorough assessment of its effectiveness in code smell detection. The results from these evaluations highlight the robustness and reliability of our model.

Overall, the evidence suggests that EMBBC is an improved approach for classification tasks compared to XGB, GB, ADA, RF, and ANN models in the evaluated datasets. This not only underscores the importance of our research but also indicates that the EMBBC may be a more reliable and effective approach for classification tasks, as it consistently performed well across multiple datasets. The EMBBC allows us to learn the hyperparameters of different models; as it is an ensemble method, the performance is improved compared to a single model. Since the proposed model uses only seven features and still provides better results than others, it could be helpful to detect code smells for software engineers and practitioners in less computational time, improving the system's overall performance.

Our findings have practical implications for software developers, offering insights into how to mitigate the negative effects of these code smells on software quality. Researchers can use the proposed ensemble method to boost performance and classify the code smells effectively. Researchers whose work relies on predicting the code smells where the maximum accurate result is needed may find this study in that direction. Additionally, the importance of feature selection and data balancing, which is emphasized in this research, will further support the application of the proposed ensemble method, inspiring researchers to explore its potential benefits.

Funding

No government, business, or non-profit foundations provided

financial support for this study.

CRediT authorship contribution statement

Pravin Singh Yadav: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Rajwant Singh Rao:** Writing – review & editing, Validation, Supervision, Methodology, Investigation, Formal analysis, Conceptualization. **Alok Mishra:** Writing – review & editing, Validation, Supervision, Methodology, Investigation, Formal analysis. **Manjari Gupta:** Writing – review & editing, Validation, Supervision, Methodology, Investigation, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors would like to express their gratitude to the editors and the anonymous reviewers whose thoughtful remarks and ideas have enhanced this work.

APPENDIX A

Table A1
Selected metrics and their description(Fontana et al., 2016; Mhawish and Gupta, 2020)

Quality Dimension	Metrics name	Metric Label	Granularity
Complexity	Weight of Class	WOC_type	–
Encapsulation	Number of Accessor Methods	NOAM_type	Class
Complexity	Average Method Weight	AMW_type	Class
Coupling	Response for a Class	RFC_type	Class
–	Number of final static attribute	num_final_static_attributes	–
–	Number of static attributes	num_static_attributes	–
–	Number of final attributes	num_final_attributes	–
–	Number of the package visibility attribute	number_package_visibility_attributes	–
–	Number of the protected visibility attribute	number_proteced_visibility_attributes	–
–	Number of final methods	number_final_methods	–
Size	Number of Attributes	NOA_type	Class
Complexity	Number of Parameters	NOP_method	Method
Size	Number of not accessor mutator methods	NOMNAMM_project	package, Project, class
Size	Lines of code	LOC_package	package, Project, class, method
Coupling	Coupling between objects classes	CBO_type	Class
Complexity	Weighted Methods Count of Not Accessor or Mutator Methods	WMCNAMM _type	Class
Inheritance	Number of Interfaces	NOI_package	Package
Cohesion	LACK OF COHESION IN METHODS	'LCOM5_type	Method
Size	Line of code without accessor or mutator methods	LOCNAMM_type	Class
Size	Lines of Code	LOC_project	Project, package, class, method
Size	Number of classes	NOCS_package	Project, package
Coupling	FANOUT	FANOUT_type	Class, method
Coupling	Called foreign not accessor or mutator methods	CFNAMM_type	Class, method
Complexity	Maximum Nesting Level	MAXNESTING_method	Method
Size	Line of code	LOC_method	Method
Complexity	Cyclomatic Complexity	CYCLO_method	Method
Complexity	Average Method weight of not accessor or Mutator method	AMWNAMM_type	Class
Coupling	Coupling Intensity	CINT_method	Method

Note (Fontana et al., 2016):didn't define the values for the cells having "–".

Data availability

The dataset used in this study are freely available on http://essere.disco.unimib.it/reverse/MLCSD.html.

References

Abdou, A., Darwish, N., 2022. Severity classification of software code smells using machine learning techniques: a comparative study. J. Software: Evolution and Process e2454. https://doi.org/10.1002/SMR.2454.

Alawadi, S., Alkharabsheh, K., Alkhabbas, F., Kebande, V.R., Awaysheh, F.M., Palomba, F., Awad, M., 2024. FedCSD: a federated learning based approach for code-smell detection. IEEE Access 12, 44888–44904. https://doi.org/10.1109/ACCESS.2024.3380167.

Alazba, A., Aljamaan, H., 2021. Code smell detection using feature selection and stacking ensemble: an empirical investigation. Inf. Software Technol. 138, 106648. https://doi.org/10.1016/J.INFSOF.2021.106648.

Aljamaan, H., 2021. Voting heterogeneous ensemble for code smell detection. Proceedings - 20th IEEE International Conference on Machine Learning and Applications, ICMLA 2021, pp. 897–902. https://doi.org/10.1109/ICMLA52953.2021.00148.

Al-Shaaby, A., Aljamaan, H., Alshayeb, M., 2020. Bad smell detection using machine learning techniques: a systematic literature review. Arabian J. Sci. Eng. 45 (4), 2341–2369. https://doi.org/10.1007/S13369-019-04311-W/TABLES/29.

Amorim, L., Costa, E., Antunes, N., Fonseca, B., Ribeiro, M., 2016. Experience report: evaluating the effectiveness of decision trees for detecting code smells. 2015 IEEE 26th International Symposium on Software Reliability Engineering, ISSRE 2015 261–269. https://doi.org/10.1109/ISSRE.2015.7381819.

Awad, M., Fraihat, S., 2023. Recursive feature elimination with cross-validation with decision tree: feature selection method for machine learning-based intrusion detection systems. J. Sens. Actuator Netw. 12 (5). https://doi.org/10.3390/jsan12050067.

Baciejowski, K., Garbala, D., Zmijewski, S., Madeyski, L., 2023. Are Code Review Smells and Metrics Useful in Pull Request-Level Software Defect Prediction?, pp. 27–52. https://doi.org/10.1007/978-3-031-27506-7_2.

Barbez, A., Khomh, F., Guéhéneuc, Y.G., 2020. A machine-learning based ensemble method for anti-patterns detection. J. Syst. Software 161, 110486. https://doi.org/10.1016/J.JSS.2019.110486.

Boutaib, S., Bechikh, S., Palomba, F., Elarbi, M., Makhlouf, M., Said, L. Ben, 2021. Code smell detection and identification in imbalanced environments. Expert Syst. Appl. 166. https://doi.org/10.1016/j.eswa.2020.114076.

Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P., 2002. SMOTE: synthetic minority over-sampling technique. J. Artif. Intell. Res. 16, 321–357.
Dewangan, S., Rao, R.S., 2022. Code smell detection using classification approaches. Lecture Notes in Networks and Systems 431, 257–266. https://doi.org/10.1007/978-981-19-0901-6_25/COVER.
Dewangan, S., Rao, R.S., 2023. Method-level code smells detection using machine learning models. Lecture Notes in Networks and Systems 725 (LNNS), 77–86. https://doi.org/10.1007/978-981-99-3734-9_7/COVER.
Dewangan, S., Rao, R.S., Chowdhuri, S.R., Gupta, M., 2023. Severity classification of code smells using machine-learning methods. SN Computer Science 4 (5), 1–20. https://doi.org/10.1007/S42979-023-01979-8/TABLES/21.
Dewangan, S., Rao, R.S., Mishra, A., Gupta, M., 2021. A novel approach for code smell detection: an empirical study. IEEE Access 9, 162869–162883. https://doi.org/10.1109/ACCESS.2021.3133810.
Dewangan, S., Rao, R.S., Mishra, A., Gupta, M., 2022a. Code smell detection using ensemble machine learning algorithms. Appl. Sci. 12 (20), 10321. https://doi.org/10.3390/APP122010321, 2022, Vol. 12, Page 10321.
Dewangan, S., Rao, R.S., Yadav, P.S., 2022b. Dimensionally reduction based machine learning approaches for code smells detection. 2022 International Conference on Intelligent Controller and Computing for Smart Power, ICICCSP 2022. https://doi.org/10.1109/ICICCSP53532.2022.9862030.
Di Nucci, D., Palomba, F., Tamburri, D.A., Serebrenik, A., De Lucia, A., 2018. Detecting code smells using machine learning techniques: are we there yet? 25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings, 2018-March 612–621. https://doi.org/10.1109/SANER.2018.8330266.
Draz, M.M., Farhan, M.S., Abdulkader, S.N., Gafar, M.G., 2021. Code smell detection using Whale optimization algorithm. Comput. Mater. Continua (CMC) 68 (2), 1919–1935. https://doi.org/10.32604/CMC.2021.015586.
Fontana, F., Mäntylä, M.V., Zanoni, M., Marino, A., Menzies Arcelli Fontana, T.F., Zanoni, M., Marino, A., Arcelli Fontana, F., Mäntylä, M.V., 2016. Comparing and experimenting machine learning techniques for code smell detection 21, 1143–1191. https://doi.org/10.1007/s10664-015-9378-4.
Fontana, F., Zanoni, M., 2017. Code smell severity classification using machine learning techniques. Knowl. Base Syst. 128, 43–58. https://doi.org/10.1016/J.KNOSYS.2017.04.014.
Fontana, F.A., Zanoni, M., Marino, A., Mäntylä, M.V., 2013. Code smell detection: towards a machine learning-based approach. IEEE International Conference on Software Maintenance, ICSM 396–399. https://doi.org/10.1109/ICSM.2013.56.
Grossi, E., Buscema, M., 2007. Introduction to artificial neural networks. Eur. J. Gastroenterol. Hepatol. 19 (12), 1046–1054. https://doi.org/10.1097/MEG.0B013E3282F198A0.
Guggulothu, T., Moiz, S.A., 2019. Detection of Shotgun Surgery and message Chain code smells using machine learning techniques. International Journal of Rough Sets and Data Analysis 6 (2), 34–50. https://doi.org/10.4018/IJRSDA.2019040103.
Guggulothu, T., Moiz, S.A., 2020. Code smell detection using multi-label classification approach. Software Qual. J. 28 (3), 1063–1086. https://doi.org/10.1007/S11219-020-09498-Y/TABLES/15.
Gupta, H., Kulkarni, T.G., Kumar, L., Neti, L.B.M., Krishna, A., 2021. An empirical study on predictability of software code smell using deep learning models. Lecture Notes in Networks and Systems, 226 LNNS 120–132. https://doi.org/10.1007/978-3-030-75075-6_10.
Gupta, H., Kumar, L., Neti, L.B.M., 2019. An empirical framework for code smell prediction using extreme learning machine*. Electromechanical Engineering and Microelectronics Conference. IEMECON 2019 - 9th Annual Information Technology, pp. 189–195. https://doi.org/10.1109/IEMECONX.2019.8877082.
Jesudoss, A., Maneesha, S., Lakshmi Naga Durga, T., 2019. Identification of code smell using machine learning. 2019 International Conference on Intelligent Computing and Control Systems, ICCS 2019, pp. 54–58. https://doi.org/10.1109/ICCS45141.2019.9065317.
Kaur, A., Jain, S., Goel, S., 2018. A support vector machine based approach for code smell detection. Proceedings - 2017 International Conference on Machine Learning and Data Science, pp. 9–14. https://doi.org/10.1109/MLDS.2017.8. *MLDS 2017, 2018-January*.
Kaur, A., Jain, S., Goel, S., Dhiman, G., 2020. A review on machine-learning based code smell detection techniques in object-oriented software system(s). Recent Advances in Electrical & Electronic Engineering (Formerly Recent Patents on Electrical & Electronic Engineering) 14 (3), 290–303. https://doi.org/10.2174/2352096513999200922125839.
Kaur, I., Kaur, A., 2021. A novel four-way approach designed with ensemble feature selection for code smell detection. IEEE Access 9, 8695–8707. https://doi.org/10.1109/ACCESS.2021.3049823.
Khatami, A., Zaidman, A., 2023. Quality assurance awareness in open source software projects on GitHub. 2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 174–185. https://doi.org/10.1109/SCAM59687.2023.00027.
Khleel, N.A.A., Nehéz, K., 2022. Deep convolutional neural network model for bad code smells detection based on oversampling method. Indonesian Journal of Electrical Engineering and Computer Science 26 (3), 1725–1735. https://doi.org/10.11591/IJEECS.V26.I3.PP1725-1735.
Khomh, F., Di Penta, M., Guéhéneuc, Y.G., 2009. An exploratory study of the impact of code smells on software change-proneness. Proceedings - Working Conference on Reverse Engineering. WCRE, pp. 75–84. https://doi.org/10.1109/WCRE.2009.28.
Kim, D.K., 2017. Finding bad code smells with neural network models. Int. J. Electr. Comput. Eng. 7 (6), 3613–3621. https://doi.org/10.11591/IJECE.V7I6.PP3613-3621.
Kiyak, E.O., Birant, D., Birant, K.U., 2019. Comparison of multi-label classification algorithms for code smell detection. 3rd International Symposium on Multidisciplinary Studies and Innovative Technologies, ISMSIT 2019 - Proceedings. https://doi.org/10.1109/ISMSIT.2019.8932855.
Liu, H., Xu, Z., Zou, Y., 2018. Deep learning based feature envy detection. ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering 385–396. https://doi.org/10.1145/3238147.3238166.
Madeyski, L., Lewowski, T., 2023. Detecting code smells using industry-relevant data. Inf. Software Technol. 155, 107112. https://doi.org/10.1016/J.INFSOF.2022.107112.
Mcgill, R., Tukey, J.W., Larsen, W.A., 1978. Variations of box plots. Am. Statistician 32, 12–16. https://doi.org/10.1080/00031305.1978.10479236.
Mhawish, M.Y., Gupta, M., 2019. Generating code-smell prediction rules using decision tree algorithm and software metrics. International Journal of Computer Sciences and Engineering 7 (5), 41–48. https://doi.org/10.26438/IJCSE/V7I5.4148.
Mhawish, M.Y., Gupta, M., 2020. Predicting code smells and analysis of predictions: using machine learning techniques and software metrics. J. Comput. Sci. Technol. 35 (6), 1428–1445. https://doi.org/10.1007/S11390-020-0323-7, 2020 35:6.
Mishra, D., Mishra, A., 2009. Simplified software inspection process in compliance with international standards. Comput. Stand. Interfac. 31 (4), 763–771. https://doi.org/10.1016/J.CSI.2008.09.018.
Mustaqim, A., Adi, S., Pristyanto, Y., Astuti, Y., 2021. The Effect of Recursive Feature Elimination with Cross-Validation (RFECV) Feature Selection Algorithm toward Classifier Performance on Credit Card Fraud Detection 270–275. https://doi.org/10.1109/ICAICST53116.2021.9497842.
Olbrich, S., Cruzes, D.S., Basili, V., Zazworka, N., 2009. The evolution and impact of code smells: a case study of two open source systems. 2009 3rd International Symposium on Empirical Software Engineering and Measurement. ESEM 2009, pp. 390–400. https://doi.org/10.1109/ESEM.2009.5314231.
Olbrich, S.M., Cruzes, D.S., Sjøberg, D.I.K., 2010. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. IEEE International Conference on Software Maintenance, ICSM. https://doi.org/10.1109/ICSM.2010.5609564.
Pecorelli, F., Palomba, F., Di Nucci, D., De Lucia, A., 2019. Comparing heuristic and machine learning approaches for metric-based code smell detection. IEEE International Conference on Program Comprehension 93–104. https://doi.org/10.1109/ICPC.2019.00023, 2019-May.
Rao, R.S., Dewangan, S., Mishra, A., Gupta, M., 2023. A study of dealing class imbalance problem with machine learning methods for code smell severity detection using PCA-based feature selection technique. Sci. Rep. 13 (1), 16245. https://doi.org/10.1038/s41598-023-43380-8.
Recupito, G., Giordano, G., Ferrucci, F., Di Nucci, D., Palomba, F., 2024. When code smells meet ML: on the lifecycle of ML-specific code smells in ML-enabled systems. ArXiv Preprint ArXiv:2403.08311 8. https://doi.org/10.48550/arXiv.2403.08311.
Reis, J. P. dos, Abreu, F.B.e., Carneiro, G. de F., 2022. Crowdsmelling: a preliminary study on using collective knowledge in code smells detection. Empir. Software Eng. 27 (3), 1–35. https://doi.org/10.1007/S10664-021-10110-5/TABLES/8.
Sahin, D., Kessentini, M., Bechikh, S., Deb, K., 2014. Code-smell detection as a bilevel problem. ACM Trans. Software Eng. Methodol. 24 (1). https://doi.org/10.1145/2675067.
Sjoberg, D.I.K., Yamashita, A., Anda, B.C.D., Mockus, A., Dyba, T., 2013. Quantifying the effect of code smells on maintenance effort. IEEE Trans. Software Eng. 39 (8), 1144–1156. https://www.academia.edu/17555487/Quantifying_the_Effect_of_Code_Smells_on_Maintenance_Effort.
Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J., 2010. The Qualitas Corpus: a curated collection of Java code for empirical studies. Proceedings - Asia-Pacific Software Engineering Conference. APSEC, pp. 336–345. https://doi.org/10.1109/APSEC.2010.46.
Thakur, P., Jadeja, M., Singh, S., 2024. Enhancing software code smell detection with modified cost-sensitive SVM. International Journal of System Assurance Engineering and Management. https://doi.org/10.1007/s13198-024-02326-7.
White, M., Tufano, M., Vendome, C., Poshyvanyk, D., 2016. Deep learning code fragments for code clone detection. 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 87–98.
Yadav, P.S., Dewangan, S., Rao, R.S., 2021. Extraction of prediction rules of code smell using decision tree algorithm. IEMECON 2021 - 10th International Conference on Internet of Everything, Microwave Engineering, Communication and Networks. https://doi.org/10.1109/IEMECON53809.2021.9689174.
Yadav, P.S., Rao, R.S., Mishra, A., 2024. An evaluation of multi-label classification approaches for method-level code smells detection. IEEE Access 1. https://doi.org/10.1109/ACCESS.2024.3387856.
Yamashita, A., Counsell, S., 2013. Code smells as system-level indicators of maintainability: an empirical study. J. Syst. Software 86 (10), 2639–2653. https://doi.org/10.1016/J.JSS.2013.05.007.
Yamashita, A., Moonen, L., 2012. Do code smells reflect important maintainability aspects? IEEE International Conference on Software Maintenance, ICSM 306–315. https://doi.org/10.1109/ICSM.2012.6405287.
Yamashita, A., Moonen, L., 2013. Exploring the impact of inter-smell relations on software maintainability: an empirical study. Proceedings - International Conference on Software Engineering 682–691. https://doi.org/10.1109/ICSE.2013.6606614.
Yu, L., Mishra, A., 2012. Experience in predicting fault-prone software modules using complexity metrics. Quality Technology & Quantitative Management 9 (4), 421–433. https://doi.org/10.1080/16843703.2012.11673302.
Zaidman, A., 2024. An Inconvenient Truth in Software Engineering? The Environmental Impact of Testing Open Source Java Projects 214–218. https://doi.org/10.1145/3644032.3644461.