On the Relative Value of Imbalanced Learning for Code Smell Detection

Fuyang Li[a], Kuan Zou[a], Jacky Wai Keung[b], Xiao Yu[a]*, Shuo Feng[c] and Yan Xiao[d]

[a] School of Computer Science and Artificial Intelligence, Wuhan University of Technology, Wuhan, China
[b] Department of Computer Science, City University of Hong Kong, Hong Kong, China
[c] School of Computer and Artificial Intelligence, Zhengzhou University, Zhengzhou, China
[d] School of Computing, National University of Singapore, Singapore

ARTICLE INFO

Keywords:
Code Smell Detection
Imbalanced Learning
Machine Learning
Empirical Software Engineering

ABSTRACT

Machine learning-based code smell detection has been demonstrated to be a valuable approach for improving software quality and enabling developers to identify problematic patterns in code. However, previous researches have shown that the code smell datasets commonly used to train these models are heavily imbalanced. While some recent studies have explored the use of imbalanced learning techniques for code smell detection, they have only evaluated a limited number of techniques and thus their conclusions about the most effective methods may be biased and inconclusive. To thoroughly evaluate the effect of imbalanced learning techniques on machine learning-based code smell detection, we examine 31 imbalanced learning techniques with seven classifiers to build code smell detection models on four code smell data sets. We employ four evaluation metrics to assess the detection performance with the Wilcoxon signed-rank test and Cliff's δ. The results show that (1) Not all imbalanced learning techniques significantly improve detection performance, but deep forest significantly outperforms the other techniques on all code smell data sets. (2) SMOTE (Synthetic Minority Over-sampling TEchnique) is not the most effective technique for resampling code smell data sets. (3) The best-performing imbalanced learning techniques and the top-3 data resampling techniques have little time cost for code smell detection. Therefore, we provide some practical guidelines. First, researchers and practitioners should select the appropriate imbalanced learning techniques (e.g., deep forest) to ameliorate the class imbalance problem. In contrast, the blind application of imbalanced learning techniques could be harmful. Then, better data resampling techniques than SMOTE should be selected to preprocess the code smell data sets.

1. Introduction

As the scale of software systems becomes increasingly large and complex, software quality attracts the attention of researchers in the software engineering domain Tong et al. (2022); Sabir et al. (2019). Code smells are code symptoms caused by design flaws or bad coding idioms, which might negatively impact software quality factors Fowler (2018); Rahman et al. (2018). Identifying code smells is a vital task that helps software developers improve the design of their software Alkharabsheh et al. (2022); Rahad et al. (2021); Sousa et al. (2019). Recently, several approaches to Code Smell Detection (CSD) have been proposed, which include two categories, i.e., heuristic-based and machine learning-based. The former Tsantalis and Chatzigeorgiou (2009); Moha et al. (2010); Palomba et al. (2013) can be broadly classified into two categories: metrics-based approaches and rule-based approaches. In metrics-based approaches, quality metrics are defined and threshold values are established for each metric. However, selecting appropriate threshold values can be challenging in this approach Fernandes et al. (2016). In contrast, rule-based approaches involve defining rules to identify code smells. These rules may be manually generated by domain experts Fontana et al. (2016). However, both of these approaches can be time-consuming and cognitively demanding for software engineers Alazba and Aljamaan (2021), leading to a shift towards the use of machine learning approaches.

To address the limitations, researchers have proposed to utilize machine learning techniques for CSD. The machine learning-based approaches first employ some binary classification algorithms to train a detection model based on extracted code smell features and then utilize the built model to predict whether a code element is smelly or not.

fyli@whut.edu.cn (F. Li[a]); zoukuan@whut.edu.cn (K. Zou[a]); jacky.keung@cityu.edu.hk (J.W. Keung[b]); xiaoyu@whut.edu.cn (X. Yu[a]); shuo.feng@hotmail.com (S. Feng[c]); dcsxan@nus.edu.sg (Y. Xiao[d])
ORCID(s):

These approaches Alkharabsheh et al. (2019); Azeem et al. (2019); Al-Shaaby et al. (2020) do not require experts to define heuristic rules and decide thresholds since they perform these tasks internally.

The recent studies Fontana et al. (2016); Fontana and Zanoni (2017); Palomba et al. (2018); Nucci et al. (2018) have raised the concern that machine learning-based detection models that are trained on imbalanced class data sets (i.e., the existence of more non-smelly instances than smelly instances) may lead to inaccurate prediction results. Therefore, some researchers have investigated the impact of some imbalanced learning techniques for CSD. For example, Alkharabsheh et al. Alkharabsheh et al. (2022) and Pecorelli et al. Pecorelli et al. (2019a, 2020) found that Synthetic Minority Over-sampling TEchnique (SMOTE) could not significantly improve detection performance. Alazba et al. Alazba and Aljamaan (2021) and Aljamaan Aljamaan (2021) showed that the stacking ensemble and voting ensemble are better than individual classifiers.

However, they only compared a limited number of imbalanced learning techniques, and many other imbalanced learning techniques have not been investigated. Therefore, their conclusions about the best-performing imbalanced learning techniques are biased and still do not have enough conviction.

Considering the aforementioned issues, we conduct a comprehensive empirical study to investigate the practical impact of 31 imbalanced learning techniques on the performance of CSD with seven machine learning classifiers to build detection models. The 31 imbalanced learning techniques can be divided into four families, i.e., (1) data resampling techniques, (2) ensemble learning techniques, (3) cost-sensitive learning techniques, and (4) imbalanced ensemble learning techniques. The seven machine learning classifiers fall into the five groups, i.e., statistic-based (Naive Bayes and logistic regression), support vector machine-based (support vector machine), decision tree-based (decision tree and random forest), nearest neighbor-based (K-nearest neighbour), and neural networks-based (multi-layer perceptron). Similar to Nucci et al. Nucci et al. (2018), we merge the same level code smell data sets to construct new data sets containing more than one type of code smells. But we further remove redundant and conflicting instances in our data sets. We use the four performance metrics (i.e., Precision, Recall, F1, and Matthews Correlation Coefficient (MCC)) to comprehensively evaluate the effects of the above-mentioned imbalanced learning techniques with each classifier on our data sets. In addition, we apply both the Wilcoxon signed-rank test Wilcoxon (1992) and the Cliff's δ Kampenes et al. (2007) to examine the performance difference between the imbalanced learning techniques and None (without imbalanced learning). Our experimental results show that:

(1) On the original code smell data sets, the classifiers except Naive Bayes have great performance for CSD and score at least 0.8 on Precision, Recall, F1, and MCC. However, the Precision, Recall, F1, and MCC values of the best-performing classifiers are reduced by 9.31%-15.66%, 10.04%-18.95%, 10.45%-21.21%, and 28.36%-56.34% on our data sets, respectively. Different classifiers achieve the best detection performance on different code smells across our data sets, e.g., Naive Bayes on Data Class and God Class, and logistic regression on Feature Envy and Long Method.

(2) Compared with None (without any imbalanced learning), there are significant performance differences among the imbalanced learning techniques. Not all techniques can enhance the detection performance, namely, 32.26%, 29.03%, 30.65%, and 34.68% techniques show a significant positive effect for CSD on our data sets in terms of Precision, Recall, F1, and MCC, respectively. Deep forest always achieves the best performance on code smell data sets and shows statistically significant improvement of 9.99%-26.18% and 31.31%-105.81% than None in terms of F1 and MCC.

(3) We are not able to achieve better CSD performance using Synthetic Minority Over-sampling TEchnique (SMOTE) over other data resampling techniques. The best data resampling techniques obtain the detection improvements than SMOTE ranging from 2.63% to 17.73% in terms of MCC.

(4) The best-performing imbalanced learning method, i.e., deep forest, has a relatively low computational time of approximately 2 seconds. The top-3 data resampling techniques also have low computational times, with a maximum of less than 0.3 seconds. However, the computational times for these top-3 data resampling techniques are generally higher for Feature Envy and Long Method compared to Data Class and God Class.

Our contributions can be concluded as following:

• We, for the first time, conduct such comprehensive empirical research to explore the practical impact of 31 imbalanced learning techniques on the performance of machine learning-based CSD.

• We identify and make an analysis of a set of 16 machine learning-based CSD researches with imbalanced learning techniques from different perspectives, including code smell types, classifiers, data set metrics, and the used imbalanced learning techniques. Researchers are able to utilize the set as a starting point to conduct subsequent investigations of CSD with imbalanced learning.

• We share the source code and data sets of our empirical study [1] to facilitate future research.

The remainder of the article is organized as follows: Section 2 introduces the related work on machine learning-based CSD and imbalanced learning techniques for CSD. Section 3 gives a brief description of the investigated imbalanced learning techniques on CSD. Sections 4 and 5 show the details of our experiment setup and provide the experimental results and research summaries in detail. Section 6 shows the differences between our results and previous results, discusses the threats to the validity, and gives some implications from the experimental results. Section 7 presents the conclusion of our research.

2. Literature Review

2.1. Machine Learning-based CSD

Kreimer Kreimer (2005) proposed to use the Decision Tree (DT) to detect the two code smells (i.e., Blob and Long Method) in two small systems, and the detection model achieved high accuracy. Khomh et al. Khomh et al. (2009, 2011) and Vaucher et al. Vaucher et al. (2009) have investigated the feasibility of Bayesian Belief Networks (BN) to detect different code smells. Bryton et al. Bryton et al. (2010) proposed a Binary Logistic Regression model calibrated with expert knowledge to detect the Long Method smell. Maiga et al. Maiga et al. (2012) employed the Support Vector Machine (SVM) method to detect the Blob smell incrementally. Amorim et al. Amorim et al. (2015) confirmed Kreimer et al.'s findings Kreimer (2005) on four medium-scale projects. Fontana et al. Fontana et al. (2013, 2016) conducted empirical research on identifying code smells. In their work, the six machine learning classification models (i.e., DT, Naive Bayes (NB), Random Forest (RF), Sequential Minimal Optimization (SMO), LibSVM, and JRip) were trained to detect the four code smells (i.e., Data Class, God Class, Feature Envy, and Long Method). Their results show that the RF classifier obtained better performance in detecting most of the code smells. Then, they Fontana and Zanoni (2017) extended their work by considering the severity of code smells. Nucci et al. Nucci et al. (2018) conducted a replicated study of Fontana et al. Fontana et al. (2016). They used the same experimental setup but merged the data sets of Fontana et al. to make the data sets more realistic. Their results suggested that the conclusions of previous studies Fontana et al. (2013, 2016) cannot be generalized in real-world scenarios, and the effectiveness of machine learning techniques on CSD needs to be further explored. Kim Kim (2017) and Liu et al. Liu et al. (2018) have used neural networks for CSD with good results. Pecorelli et al. Pecorelli et al. (2019b) conducted an empirical investigation comparing the performance of machine learning-based (i.e., NB) and heuristic-based methods. Their results presented that the accuracy and validity of these two methods for detecting code smell still need further investigation. Some studies Sharma et al. (2021); Yu et al. (2021); Zhang and Dong (2021); Li and Zhang; Zhang et al. (2022) tried to apply deep learning techniques for CSD, and their conclusions examined that some deep learning models accept a better performance on CSD, i.e., Convolutional Neural Networks (CNN-1) Sharma et al. (2021); Zhang et al. (2022), recurrent neural network Sharma et al. (2021), long short-term memory Yu et al. (2021); Li and Zhang, residual network Zhang and Dong (2021), and attention Zhang et al. (2022).

2.2. Imbalanced Learning for CSD

A frequently encountered problem is that the code smell data consists of only a few smelly instances and many non-smelly instances. The imbalanced data distribution makes the built detection model prone to predict the new instances to be non-smelly and consequently perform poorly in finding smelly instances. Therefore, some researchers have investigated whether imbalanced learning methods can alleviate the problem and improve the performance of CSD models. To have knowledge of the research progress of machine learning-based CSD with imbalanced learning techniques, we perform a literature investigation to obtain most related articles published between 2005 to 2022.12 [2]. To ensure that our literature review is comprehensive, we have established the following inclusion criteria: (1) the article is about CSD with imbalanced learning techniques and is written in English. (2) the full text is available. As far as we know, the first article that employed machine learning techniques to detect code smells was published by Kreimer et al. Kreimer (2005) in Electronic Notes in Theoretical Computer Science 2005 (abbreviated as Kreimer 2005 article in the rest of this paper). Thus, the starting year of the literature investigation is set as 2005. We retrieve related articles written in English using Google Scholar and DBLP. We follow Zhou et al.'s Zhou et al. (2018) forward snowballing search method to recursively retrieve the articles that cited the Kreimer 2005 article. More precisely, we first retrieve the related articles that cited the Kreimer 2005 article and employed machine learning algorithms

[1] https://github.com/zoukuan1/imbalanced-learning-for-code-smell-detection

[2] The literature investigation was conducted in 2022.12.

Table 1
The summary of related works

Study	Code smell type	Classifiers	data set metrics	Imbalanced learning methods
Pecorelli et al. (2019a)	Class-level: 4 Method-level: 1	NB	9 Object-Oriented Metrics	ClassBalancer, Resample, SMOTE, Cost-Sensitive Classifier, One-Class Classifier
Pecorelli et al. (2020)	Class-level: 5 Method-level: 6	NB	17 Object-Oriented Metrics	Over-Sampling, Under-Sampling, SMOTE, Cost-Sensitive Classifier, One-Class Classifier
Shen et al. (2020)	Class-level: 1 Method-level: 1	DT, KNN, NB, RF, RULE, SVM	11 Object-Oriented Metrics	RUS
Akhter et al. (2021)	Class-level: 3 Method-level: 1	SVM, NB, RF, DT	50 Object-Oriented Metrics	SMOTE
Alazba and Aljamaan (2021)	Class-level: 2 Method-level: 4	DT, SVM, NB, LR, MLP, SGD, GP, KNN, LDA	55-82 Object-Oriented Metrics	Stacking Ensemble
Alkharabsheh et al. (2021)	Class-level: 1	LibSVM, IBK, DT, JRip, SMO, NB, RC, RF	19 Object-Oriented Metrics	SMOTE
Aljamaan (2021)	Class-level: 2 Method-level: 4	DT, LR, SVM, MLP, SGD	61-82 Object-Oriented Metrics	Voting Ensemble
Gupta et al. (2021)	Class-level: 4 Method-level: 4	8 Deep Learning Models	Dimensional Metrics Complexity Metrics Object-Oriented Metrics Android-Oriented Metrics	SMOTE, ADASYN
Jain and Saha (2021)	Class-level: 2 Method-level: 4	SVM, KNN, NB, DT, LDA, LR, Bagging, AdaBoost, XGBoost, GB	61-82 Object-Oriented Metrics	SMOTE
Patnaik and Padhy (2021)	Class-level: 3 Method-level:2	LASSO, Ridge, LAR, NB, SVM, RF, DT	45 Object-Oriented Metrics	Random Sampling
Stefano et al. (2021)	Class-level: 2 Method-level: 1	LR, DT, NB, RF	6 Object-Oriented Metrics	SMOTE
Alkharabsheh et al. (2022)	Class-level: 1	LDA, QDA, NB, MLP, SVM, DT, GB, CatBoost, LGBM, XGB, XGBRF, AdaBoost, Bagging, RF, ET, KNN, NC, GP, Ridge, LR, Perceptron, PA, SGD	16 Object-Oriented Metrics	SMOTE
Khleel and Nehéz (2022)	Class-level: 2 Method-level: 2	CNN-1	43 Object-Oriented Metrics	SMOTE
Kovačević et al. (2022)	Class-level: 1 Method-level: 1	code2vec, code2sep, CuBERT	10 Object-Oriented Metrics	SMOTE, SMOTEENN
Nanda and Chhabra (2022)	Class-level: 2 Method-level: 2	DT, RF, SVM, JRip, NB	63-84 Object-Oriented Metrics	SMOTE
Yedida and Menzies (2022)	Class-level: 2 Method-level: 2	Deep Neural Network	Structural Information Lexical Information	SMOTE

for CSD, and repeat the procedure on other related articles. Then, we set up "imbalanced learning" + "code smell detection" as the search terms to search. Finally, after applying our inclusion criteria, we find the relevant articles that cited in these investigations. As a result, 16 related articles on **machine learning-based CSD with imbalanced learning techniques** are found in the literature. Table 1 lists the summary of the literature review, where the first column presents the first author and the publication year, the second column presents the types of code smells, the third column presents the used machine learning algorithms, and the fourth column presents the code smell features, and the last one presents the imbalanced learning methods used in the study.

Some researchers Akhter et al. (2021); Alkharabsheh et al. (2021); Gupta et al. (2021); Jain and Saha (2021); Stefano et al. (2021); Khleel and Nehéz (2022); Kovačević et al. (2022); Nanda and Chhabra (2022); Yedida and Menzies (2022) applied SMOTE as a data preprocessing method to alleviate the class imbalance problem and then utilized or proposed some more advanced algorithms for CSD. For example, Akhter et al. Akhter et al. (2021) used the four machine learning classifiers (i.e., NB, RF, DT, and SVM) to investigate the effect of machine learning techniques on CSD. Alkharabsheh et al. Alkharabsheh et al. (2021) extended two software metrics project domain and size category to detect code smells by using the eight machine learning classifiers (i.e., LibSVM, IBK, DT, JRip, SMO, NB, RF, and Random Committee (RC)). Gupta et al. Gupta et al. (2021) employed eight deep learning models for CSD. Jain et al. Jain and Saha (2021) used hybrid feature selection and ensemble learning techniques to improve detection performance by adopting the 11 classifiers (i.e., SVM, KNN, NB, DT, Linear Discriminant Analysis (LDA), Logistic Regression (LR), Bagging, AdaBoost, XGBoost, Gradient Boost (GB), and Stacking). Stefano et al. Stefano et al. (2021) proposed a cross-project method, which used the four machine learning classifiers (i.e., LR, DT, NB, and RF) to train detection models and predicted the smelliness of within-project instances. Khleel et al. Khleel and Nehéz (2022) employed CNN-1 with SMOTE for CSD. Kovavcevic et al. Kovačević et al. (2022) used code embeddings (i.e., code2vec, code2sep, and CuBERT) and over-sampling strategies (i.e., SMOTE and SMOTEENN) to detect code smells. Nanda et al. Nanda and Chhabra (2022) proposed SSHM method with SMOTE and Stacking for severity classification of four code smells. Yedida et al. Yedida and Menzies (2022) studied the effect of the fuzzy oversampling technique with SMOTE on CSD. Shen et al. Shen et al. (2020) used RUS to solve the imbalanced problem across six machine learning classifiers (i.e., DT, NB, RF, SVM, K-Nearest Neighbour (KNN), and Rule Ensemble (RULE)) when they analyzed the influence of hyper-parameter optimization on CSD. Patnaik et al. Patnaik and Padhy (2021) used RUS and ROS to handle the imbalanced problem in their data sets and developed a model for CSD by using the seven classifiers (i.e., DT, RF, SVM, NB, Ridge, Least Absolute Shrinkage and Selection Operator (LASSO), and Least Angle Regression (LAR)). However, the above-mentioned studies did not compare the performance difference of CSD models trained on the original data sets and the balanced data sets processed by SMOTE, RUS, and ROS. Therefore, no conclusive empirical evidence from their experimental results showed that using SMOTE, RUS, and ROS could significantly positively impact machine learning-based CSD models.

Alkharabsheh et al. Alkharabsheh et al. (2022) used the machine learning classifiers (i.e., LDA, Quadratic Discriminant Analysis (QDA), NB, Multi-Layer Perceptron (MLP), SVM, DT, GB, CatBoost, Light Gradient Boosting Machine (LGBM), XGBoost, XGBoost with Random Forest (XGBRF), AdaBoost, Bagging, RF, Extra Trees (ET), KNN, Nearest Centroid (NC), Gaussian Process (GP), Ridge, LR, Perceptron, Passive Aggressive (PA), and Stochastic Gradient Descent (SGD)) to compare whether using SMOTE would improve the detection performance on God Class detection. Their results showed that SMOTE could not improve the God Class detection performance.

Alazba et al. Alazba and Aljamaan (2021) studied the effect of the stacking ensemble on six code smells, and their conclusions examined that the performance of Stacking with LR and SVM is better than all individual classifiers. Aljamaan Aljamaan (2021) investigated the performance of the voting ensemble on CSD. Their results showed that the voting ensemble has a superior detection performance on all code smells.

Pecorelli et al. Pecorelli et al. (2019a) conducted a investigation to examine the role of imbalanced learning techniques (ClassBalancer, Resample, SMOTE, Cost-Sensitive Classifier, and One-Class Classifier) on machine learning-based CSD. Then, they Pecorelli et al. (2020) combined five imbalanced learning techniques (Over-Sampling, Under-Sampling, SMOTE, Cost-Sensitive Classifier, and One-Class Classifier) with a NB classifier to further investigate this issue. Their results showed that the performance of CSD models has not been significantly improved using imbalanced learning techniques, even though the performance has a slight improvement when using SMOTE.

However, the limitations of the past research are that only a few imbalanced learning techniques be used, and the data sets used for experiments need to be improved. To overcome these limitations of past research and their findings, in this paper, we conduct a larger empirical research to examine the practical impact of the performance of CSD on machine learning algorithms using imbalanced learning techniques.

3. Preliminaries

We briefly outline the studied imbalanced learning techniques. We also regard the None method as a baseline method (without any imbalanced learning). We apply 31 imbalanced learning techniques in this empirical study, as

Table 2
The overview of the 31 imbalanced learning methods.

Family	Methods	Abbreviation
Data Resampling Techniques	Random Over-Sampling	ROS
	Synthetic Minority Over-Sampling Technique	SMOTE
	Adaptive Synthetic Sampling Approach	ADASYN
	Borderline-SMOTE	BSMOTE
	Random Under-Sampling	RUS
	Near Miss	NM
	Condensed Nearest Neighbor	CNN
	Tomek Links	TL
	Edited Nearest Neighbors	ENN
Ensemble Learning Techniques	Bootstrap aggregating	Bagging
	Adaptive Boosting	AdaBoost
	CatBoost	CatBoost
	eXtreme Gradient Boosting	XGBoost
	Deep Forest	DF
	Stacking Logistic Regression	StackingLR
	Stacking Decision Tree	StackingDT
	Stacking Support Vector Machine	StacingSVM
Cost-Sensitive Learning Techniques	AdaCost	AdaCost
	Asymmetric Boosting	AsymBoost
	AdaUBoost	AdaUBoost
	Cost-Sensitive Support Vector Machine	CSSVM
	Cost-Sensitive Decision Tree	CSDT
Imbalanced Ensemble Learning Techniques	SMOTEBoost	SMOTEBoost
	ROSBoost	ROSBoost
	SMOTEBagging	SMOTEBagging
	ROSBagging	ROSBagging
	RUSBoost	RUSBoost
	RUSBagging	RUSBagging
	EasyEnsemble Classifier	EasyEnsemble
	BalanceCascade Classifier	BalanceCascade
	Balanced Random Forest	BRF

shown in Table 2.

3.1. Data Resampling

Data resampling techniques can mainly divide into two categories, i.e., over-sampling and under-sampling. The former produces a superset of the original code smell data sets by duplicating existing smelly instances or creating new smelly instances from existing smelly ones, while the latter produces a subset of the original code smell data sets by eliminating non-smelly instances. To maintain consistency with previous studies Pecorelli et al. (2020); Alkharabsheh et al. (2022) and common practices, we set the default smelly ratio to 0.5, resulting in an equal number of smelly and non-smelly instances in the balanced data sets.

(1) Random Over-Sampling (ROS) randomly replicates n smelly instances from the original code smell data sets (n is calculated according to the ratio value of the smelly instances) and then adds them into the original data sets to obtain balanced data sets.

(2) Synthetic Minority Over-sampling TEchnique (SMOTE) first randomly selects n smelly instances from the original code smell data sets. For each smelly instance M_A, we randomly choose one of its k nearest neighbors using the K-Nearest Neighbors (KNN) algorithm, i.e., M_B. The feature of the synthetic instance $M_{synthetic}$ is

$$\boldsymbol{x}_{synthetic} = \boldsymbol{x}_A + Random(0,1) \times (\boldsymbol{x}_B - \boldsymbol{x}_A), \tag{1}$$

where $\boldsymbol{x}_{synthetic}$, $\boldsymbol{x}_A$, and $\boldsymbol{x}_B$ are the feature of $M_{synthetic}$, M_A, and M_B, respectively. Finally, we add the n synthetic smelly instances into the original data sets to obtain balanced data sets.

(3) Adaptive Synthetic Sampling Approach (ADASYN) is an extension of SMOTE, but it creates new synthetic instances near the boundary between two classes rather than within the smelly instances.

(4) Borderline-SMOTE (BSMOTE) is an improved oversampling algorithm based on SMOTE, which only uses the smelly instances on the boundary to synthesize new instances.

(5) Random Under-Sampling (RUS) randomly eliminates m non-smelly instances to balance the class distribution (m is calculated according to the ratio value of the smelly instances).

(6) Near Miss (NM) calculates the distance between the smelly and non-smelly instances and randomly deletes the non-smelly instances according to the distance, mainly to alleviate the information loss problem in RUS.

(7) Condensed Nearest Neighbor (CNN) first finds a subset of non-smelly instances that leads to no loss in model performance and then deletes the instances from the original code smell data sets.

(8) Tomek Links (TL) is a modification from CNN to obtain balanced data sets by finding all the non-smelly instances closest to the smelly instances and then removing them.

(9) Edited Nearest Neighbors (ENN) selects a non-smelly instance and uses the KNN algorithm to get the k neighbors of the instance. If more than half of the k neighbors are not the non-smelly instances, the instance will be deleted.

3.2. Ensemble Learning

Ensemble learning is regarded as a meta-algorithm, which is not a single machine learning algorithm but completes the learning task by building and combining multiple machine learners to improve the performance of CSD.

(1) Bootstrap aggregating (Bagging) trains multi classification models based on multi boostrap samples and form a final stronger classifier by voting their individual predictions.

(2) Adaptive Boosting (AdaBoost) integrates multiple base classifiers and assigns new weights to the samples misclassified by the previous base classifier to reduce the error rate.

(3) CatBoost is a gradient boosting decision tree framework with few parameters and high accuracy based on an oblivious tree algorithm, which could efficiently and reasonably process categorical features and has strong generalization ability.

(4) eXtreme Gradient Boosting (XGBoost) is a gradient boosting decision tree framework that provides a novel tree learning algorithm for processing sparse data and enables faster learning of machine learning classifiers through parallel and distributed computing, thus enabling faster model exploration.

(5) Deep Forest (DF) integrates different kinds of forests in width and depth, which improves the classification ability of machine learning models by using multi-grained scanning and cascade forests.

(6) Stacking Ensemble uses a meta-learning algorithm to best integrate the predictors from two or more base well-performing base classification models. The three based classifiers commonly used in stacking ensembles are Logistic Regression (LR), Decision Tree (DT), and Support Vector Machine (SVM). We call the stacking ensemble integrated with the three classifiers StackingLR, StackingDT, and StackingSVM.

3.3. Cost-Sensitive Learning

In actuality, non-smelly instances are more common than smelly isntances Alazba and Aljamaan (2021). As a result, when utilizing machine learning classifiers for classification, instances are more likely to be misclassified as non-smelly due to the disproportionate representation of non-smelly instances. Cost-sensitive learning techniques aim at building code smells detection models with minimal misclassification costs by specifying a cost-sensitive matrix.

(1) AdaCost applies the misclassification cost to update the training distribution on successive boosting rounds. The main idea of AdaCost is to comprise the cost and produce more advanced machine learning classifiers, which can reduce the cost of misclassifications better than AdaBoost.

(2) Asymmetric Boosting (AsymBoost) combines AdaBoost with cost-sensitive learning techniques and uses asymmetric misclassification costs to update the data distribution during classifier training to reduce the likelihood of misclassification.

(3) AdaUBoost is a variant of AdaBoost and is designed to optimize an unequal loss on imbalanced training data sets by preprocessing and manipulating the data distribution during classifier training to reduce the cumulative misclassification cost.

(4) Cost-Sensitive Support Vector Machine (CSSVM) weighs the margin by integrating the misclassification cost during the training of SVM.

(5) Cost-Sensitive Decision Tree (CSDT) incorporates the misclassification cost into the process of separating software instances to two groups during the training of the decision tree.

3.4. Imbalanced Ensemble Learning

Imbalanced ensemble learning techniques resample or reweight the training data during the training process of ensemble learning, which could improve the correct classification ability of the ensemble learning classifier for imbalanced data sets.

(1) SMOTEBoost iteratively builds t weak classification models similar to AdaBoost. In each iteration, the weak classification model is trained based on the balanced data sets generated by SMOTE.

(2) ROSBoost is similar to SMOTEBoost but uses ROS instead of SMOTE.

(3) RUSBoost is similar to SMOTEBoost but uses RUS instead of SMOTE to achieve data balance.

(4) SMOTEBagging involves SMOTE in the process of Bagging to generate balanced bootstrap samples and trains multi classification models based on the balanced data sets.

(5) ROSBagging is similar to SMOTEBagging but uses ROS instead of SMOTE.

(6) RUSBagging is similar to SMOTEBagging but uses RUS instead of SMOTE to achieve data balance.

(7) EasyEnsemble Classifier builds an ensemble of AdaBoost predictors trained on multi-balanced bootstrap samples generated by RUS.

(8) BalanceCascade Classifier iteratively drops non-smelly instances that were already well-classified by the current ensemble. After that, it performs RUS on the remaining non-smelly instances and trains a new base predictor.

(9) Balanced Random Forest (BRF) trains the random forest predictor based on multi-balanced bootstrap samples generated by RUS.

4. Experimental setup

4.1. Data sets

In our study, we use the same experimental data sets as Nucci et al. Nucci et al. (2018), Aljamaan Aljamaan (2021), Jain et al. Jain and Saha (2021), and Nanda et al. Nanda and Chhabra (2022), which was built by Fontana et al. Fontana et al. (2016) and contained four types of code smells collected from 74 software systems. Each type of code smell data sets includes 140 smelly instances and 280 non-smelly ones (420 instances in total).

But the original data sets was questioned by Nucci et al. Nucci et al. (2018). Their replicated investigation found that a trained machine learning classifier could easily identify smelly and non-smelly instances since their feature distribution is very different in the original data sets. In addition, each data set only contains one type of code smells, which does not correspond to real-world scenarios (Nucci et al. Nucci et al. (2018) pointed that a software instance usually has two or more types of smells instead of one). Therefore, they merged the same level (i.e., class-level and method-level) of code smells to construct new data sets. For example, they merged the God Class and Data Class data sets, then set the label of all instances in the Data Class data set as no-smelly, and finally returned a new God Class data set.

Alazba et al. Alazba and Aljamaan (2021) raised the question about Nucci's behavior Nucci et al. (2018) of merging the same level code smell data sets. They showed that this merging strategy would result in data sets with more than 30% redundant instances and more than 15% conflicting instances, which would degrade the performance of machine learning classifiers. Therefore, they used the original data sets in their study.

Example 1: Suppose a God Class data set contains 2 smelly instances (i.e., a and b) and 3 non-smelly instances (i.e., c, d, and e). A Data Class data set contains 2 smelly instances (i.e., b and f) and 3 non-smelly instances (i.e., e, g, and h). If we adopt the Nucci et al.'s Nucci et al. (2018) merging strategy, we will return a new God Class data set, which contains 2 smelly instances (i.e., a and b) and 8 non-smelly instances (i.e., c, d, e, b, f, e, g, and h). From Alazba et al.'s Alazba and Aljamaan (2021) point of view, e is the redundant instance and b is the conflicting instance in the new God Class data set.

To avoid these issues, we use the original data sets provided by Fontana et al. Fontana et al. (2016), and the merging strategy of Nucci et al. Nucci et al. (2018) is also used on the original data sets. But we remove the same instances so the redundant and conflicting instances would not appear in the new data sets. For example, we delete b and e from Data Class, then merge both the code smell data sets to create a new God Class data set, which contains 2 smelly instances (i.e., a and b) and 6 non-smelly instances (i.e., c, d, e, f, g, h). Therefore, we can obtain the

Table 3
The details of the our data sets

Code Smells	Level	Number of instances		
		Smelly	Non-smelly	Total
God Class	Class	140	559	699
Data Class		138	561	699
Feature Envy	Method	140	378	518
Long Method		140	378	518

Table 4
The confusion matrix

	Actual smelly	Actual non-smelly
Predicted smelly	TP	FP
Predicted non-smelly	FN	TN

experimental data sets that contain more than one type of code smells. Table 3 shows the detailed information of our experimental data sets.

4.2. Performance Measures

In our empirical study, we use the three threshold-dependent evaluation metrics (Precision, Recall, and F-measure (F1)) and one threshold-independent evaluation metric (Matthews Correlation Coefficient, MCC) to evaluate the performance of CSD models. The metrics are widely used in recent studies Pecorelli et al. (2019b); Dewangan et al. (2021); Boutaib et al. (2021). In the binary classification problem, these four evaluation metrics can be calculated according to a confusion matrix, as shown in Table 4.

Precision is the percentage of the actual smelly instances to all the predicted smelly instances in the data sets. A higher Precision means that the CSD model could help the software testing team more exactly to find smelly instances.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

Recall is the percentage of the correctly predicted smelly instances to all the actual smell ones in the data sets. A higher Recall means that the software testing team could capture more smelly instances.

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

F1 is viewed as the harmonic mean between Precision and Recall and its value ranges from 0 to 1, where 0 implies the worst prediction, and 1 implies the best prediction. In other words, a CSD model achieves better performance if it has a higher F1 value.

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{3}$$

MCC provides an overview by taking into account all the terms of the confusion matrix, which could be used in an imbalanced environment to reduce bias. MCC has a value ranging from -1 to 1, where -1 means that the predicted and actual results are completely inconsistent, and 1 means an opposite aspect to -1. MCC = 0 means that the CSD model applies a random prediction.

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{3}$$

4.3. Experimental Process

The original data sets constructed by Fontana et al. Fontana et al. (2016) contains missing values, which may impair the performance of machine learning classifiers Acuna and Rodriguez (2004). There are many ways to deal with missing data Young et al. (2011), such as deletion, mean imputation, and median imputation. The same as Alazba et al.'s Alazba and Aljamaan (2021) investigation, we use the mean imputation method to handle missing data for each

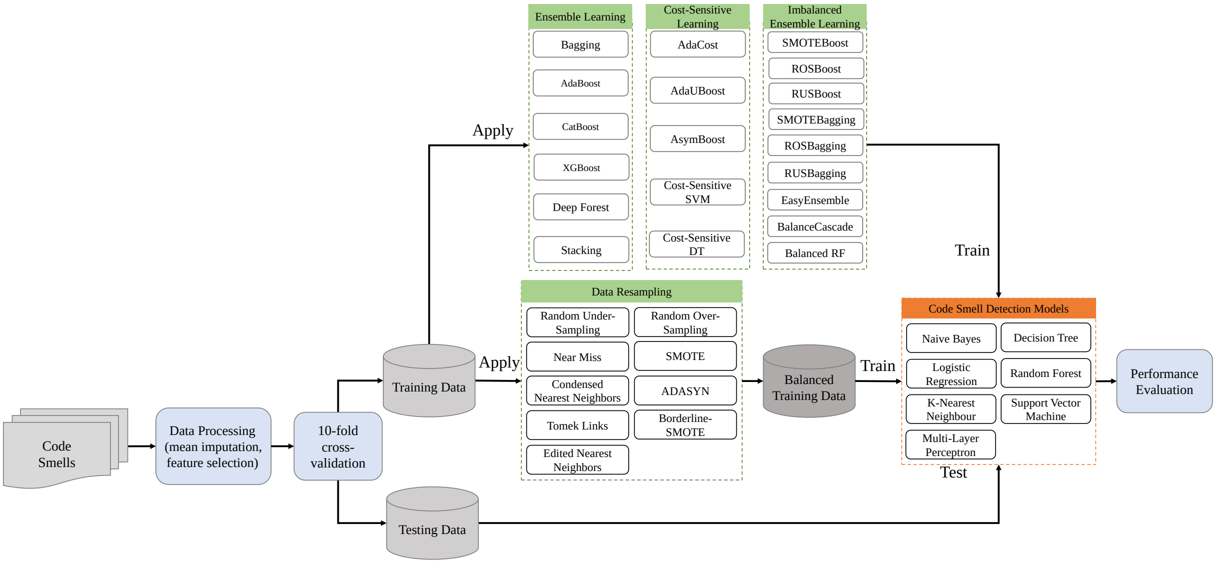


Figure 1: Framework of our study.

code smell data set. Mean imputation is a commonly used approach for managing missing values, which produces a good classification result in supervised classification problems Mundfrom and Whitcomb (1998). These missing values will be replaced with the mean values belonging to the current column properties.

Irrelevant software features in the original data sets may degrade the detection performance of machine learning classifiers John et al. (1994). The same as Nucci et al.'s study Nucci et al. (2018), we remove irrelevant software features by the feature selection method Information Gain Ratio (IGR), whose value is between 0 and 1. We calculate the IGR values of all software features in each code smell data set and remove the software features whose IGR values are less than 0.1 Quinlan (1986).

The same as previous CSD studies Nucci et al. (2018); Pecorelli et al. (2020); Alazba and Aljamaan (2021); Alkharabsheh et al. (2022), we use the 10-fold cross-validation method to verify the performance of detection models. Each data sets is randomly divided into 10 folds of the same size. We select the nine folds from all ten folds as the training data sets and apply the imbalanced learning techniques to the training data sets. The last fold is used as the testing data sets. Next, we iterate 10 times to ensure all ten folds are used as both the training and the testing data sets. Finally, we obtain 100 results of each detection model in each data set and take the median value of the 100 results as the final result of the data sets. The framework of our study as shown in Figure 1.

4.4. Classifiers

In recent research, many machine learning classifiers have been used for CSD Alkharabsheh et al. (2022). To investigate the impact of the imbalanced learning techniques with more classifiers for the performance of CSD, we construct the training model with the seven classifiers.

(1) **Naive Bayes (NB)** is based on the Bayes theory and assumes the software features are independent. It needs to estimate a few parameters and is not sensitive to missing data, and the implementation of the algorithm is relatively simple.

(2) **Support Vector Machine (SVM)** uses a kernel function to transform the original linearly inseparable data to a feature space, where the data is linearly separable and then finds the optimal hyperplane to partition the feature space to separate instances with different class labels.

(3) **Logistic Regression (LR)** is used to classify elements of a set into two groups (binary classification) by calculating the probability of each element of the set, which makes it able to classify code smells into discrete outcomes.

(4) **Decision Tree (DT)** expresses a tree structure, which depends on the attributes of the object type by its branch sort.

Table 5
The optimized hyper-parameters of the classifiers. (The default parameter value is in bold font.)

Classifier	Hyper-parameters	Tuning Range	Description
DT	min_samples_split	[**2**, 3, 4, 5, 6]	The minimum number of samples required to split an internal node
KNN	n_neighbors	[1, **5**, 9, 13, 17]	Number of neighbors to use by default for k neighbors queries
LR	tol	[0.1, 0.01, 0.001, **0.0001**, 0.00001]	Tolerance for stopping criteria
MLP	alpha	[0, 0.1, 0.01, 0.001, **0.0001**]	L2 penalty (regularization term) parameter
	hidden_layer_sizes	[4, 8, 16, 32, 64, **100**]	The number of neurons in the hidden layers
RF	n_estimators	[10, 20, 30, 40, 50, **100**]	The number of trees in the forest
SVM	*C*	[0.25, 0.5, **1.0**, 2.0, 4.0]	The strength of the regularization is inversely proportional to C

(5) **K-Nearest Neighbour (KNN)** is one of the simplest machine learning algorithms. A sample also belongs to the category, if most of the *K* most similar (closest distance) samples in the feature space belong to a specific category.

(6) **Random Forest (RF)** generates multi decision trees based on bootstrap samples and makes the final prediction by voting.

(7) **Multi-Layer Perceptron (MLP)** is a feedforward artificial neural network model that maps multiple inputs to a single output. It contains the input layer, the hidden layers, and the output layer, each of which is fully connected to the previous and subsequent layers.

These classifiers belong to five groups, including statistic-based (i.e., NB and LR), support vector machine-based (i.e., SVM), decision tree-based (i.e., DT and RF), nearest neighbor based (i.e., KNN), and neural networks-based (i.e., MLP). For the machine learning classifiers except NB used in our study, some hyper-parameters are needed to configure to achieve the best detection performance before training models. The same as Shen et al.'s Shen et al. (2020) and Stefano et al.'s Stefano et al. (2021) investigations, we employ the grid search algorithm Bergstra and Bengio (2012) to optimize those hyper-parameters. Table 5 shows the details of the hyper-parameters of the classifiers.

4.5. Statistic Test

We employ the Wilcoxon signed-rank test Wilcoxon (1992) with a Bonferroni correction and Cliff's δ Kampenes et al. (2007) to analyze the practical significance of the prediction performance between the two models. The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used to compare pairs of models. The Cliff's δ is a non-parametric effect size that measures the magnitude of difference between two models. In our empirical study, we provide the 100 results of the 10-fold cross-validation of the imbalanced learning techniques and None to the Wilcoxon signed-rank test and calculate Cliff's δ value. When the adjusted p-value is less than 0.05 and $|\delta|$ is larger than 0.147, the corresponding imbalanced learning method and None have a significant difference Wilcoxon (1992); Kampenes et al. (2007).

5. Experimental Results

This section discusses the research questions with their motivations and results.

5.1. What is the performance of the classifiers for CSD on our data sets?

Motivations: Fontana et al. Fontana et al. (2016) constructed the original code smell data sets, and their results showed that machine learning classifiers achieved high performance on CSD. However, Nucci et al.'s conclusions Nucci et al. (2018) implied that the original data sets were not suitable for real-world scenarios where an instance should contain more than one type of code smells. Therefore, we wonder how is the practical effect of machine learning classifiers on CSD.

Table 6
The performance of the seven classifiers on the original code smell data sets. (The bold font means the corresponding technique obtains the best performance.)

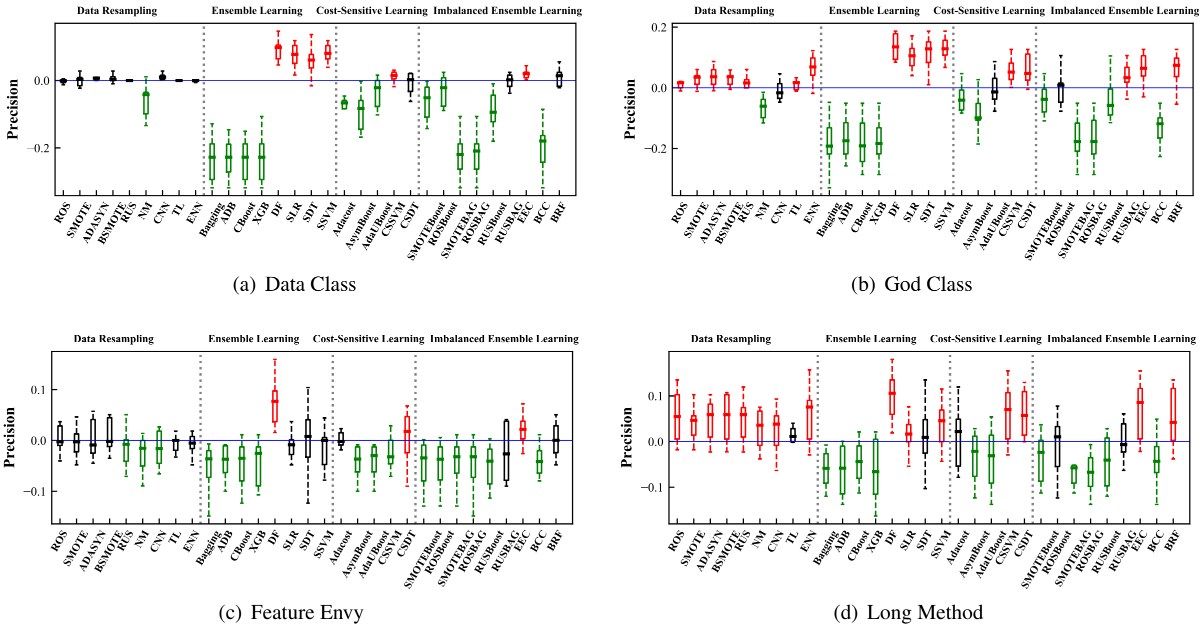
Classifier	Data Class				God Class			
	Precision	Recall	F1	MCC	Precision	Recall	F1	MCC
DT	**0.978**	**0.976**	**0.976**	**0.949**	0.955	0.952	0.952	0.901
KNN	0.927	0.917	0.917	0.833	0.936	0.929	0.928	0.901
LR	0.928	0.929	0.928	0.837	0.954	0.952	0.952	0.892
MLP	0.942	0.929	0.930	0.853	**0.977**	**0.977**	**0.976**	0.934
NB	0.873	0.833	0.835	0.695	0.948	0.940	0.942	0.860
RF	**0.978**	**0.976**	**0.976**	0.947	**0.977**	0.976	**0.976**	**0.946**
SVM	0.977	**0.976**	**0.976**	0.944	**0.977**	0.976	**0.976**	0.939
Classifier	Feature Envy				Long Method			
	Precision	Recall	F1	MCC	Precision	Recall	F1	MCC
DT	0.956	0.952	0.951	0.894	**0.978**	0.976	**0.976**	0.949
KNN	0.916	0.917	0.915	0.801	0.952	0.952	0.952	0.889
LR	0.933	0.929	0.926	0.834	0.977	**0.977**	**0.976**	**0.950**
MLP	0.930	0.929	0.929	0.847	**0.978**	0.976	**0.976**	0.949
NB	0.905	0.905	0.905	0.777	0.959	0.952	0.953	0.896
RF	**0.977**	**0.976**	**0.976**	**0.945**	**0.978**	0.976	0.976	0.949
SVM	0.954	0.952	0.951	0.886	0.977	0.976	**0.976**	**0.950**

Table 7
The performance of the seven classifiers on our data sets. (The bold font means the corresponding technique obtains the best performance.)

Classifier	Data Class				God Class			
	Precision	Recall	F1	MCC	Precision	Recall	F1	MCC
DT	0.622	0.632	0.625	-0.195	0.621	0.631	0.623	-0.189
KNN	0.749	0.743	0.742	0.191	0.724	0.731	0.720	0.113
LR	0.761	**0.791**	0.757	0.192	0.760	0.788	0.762	0.195
MLP	0.765	0.790	**0.769**	0.241	0.771	0.789	0.773	0.252
NB	**0.872**	0.737	0.764	**0.501**	**0.824**	**0.794**	**0.804**	**0.413**
RF	0.637	0.645	0.639	-0.148	0.641	0.640	0.637	-0.126
SVM	0.671	0.787	0.721	0.058	0.640	0.785	0.704	-0.029
Classifier	Feature Envy				Long Method			
	Precision	Recall	F1	MCC	Precision	Recall	F1	MCC
DT	0.822	0.821	0.819	0.532	0.761	0.750	0.749	0.350
KNN	0.860	0.859	0.854	0.623	0.826	0.817	0.814	0.515
LR	**0.886**	**0.878**	0.871	**0.677**	**0.834**	**0.828**	**0.825**	**0.557**
MLP	0.880	0.876	**0.874**	0.675	0.829	0.819	0.820	0.545
NB	0.837	0.828	0.830	0.570	0.830	0.819	0.820	0.543
RF	0.841	0.836	0.836	0.579	0.779	0.765	0.767	0.408
SVM	0.870	0.865	0.856	0.632	0.816	0.817	0.809	0.512

Approaches: We use the baseline method None to train the seven classifiers on the original data sets and our data sets to compare the detection performance on different data sets. We list the median Precision, Recall, F1, and MCC values of the seven classifiers in Table 6 and Table 7. We highlight the best classifiers on each evaluation metric for each data set in bold.

Results: (1) On the original data sets, the best classifiers achieve the following performance: DT scores 0.976 of F1 and 0.949 of MCC on Data Class, RF scores 0.976 of F1 and 0.946 of MCC on God Class, RF scores 0.976 of F1 and 0.945 of MCC on Feature Envy, LR and SVM score 0.976 of F1 and 0.950 of MCC on Long Method. The classifiers except NB score at least 0.8 on Precision, Recall, F1, and MCC, which means that the classifiers perform pretty well for CSD on the original data sets. The main reason of high performance is that the feature distribution between smelly instances and non-smelly ones is significantly different in most cases according to Nucci et al. analysis



(a) Data Class

(b) God Class

(c) Feature Envy

(d) Long Method

Figure 2: The performance difference between the imbalanced learning techniques and None in terms of Precision.

Nucci et al. (2018).

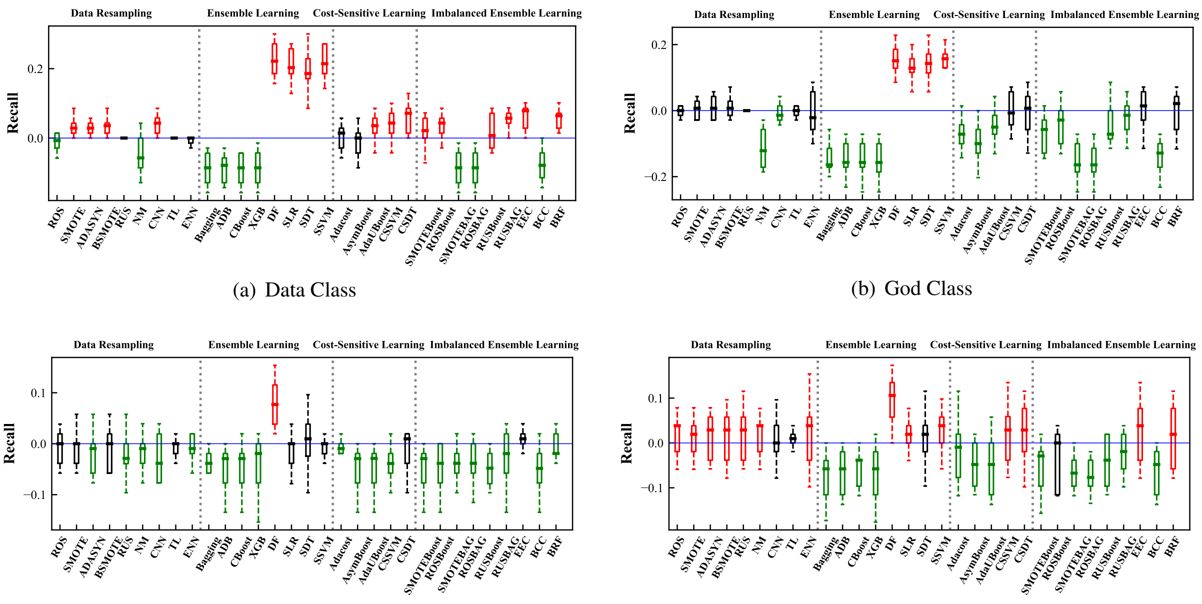
(2) On our data sets, the best classifiers achieve the following performance: MLP and NB score 0.769 of F1 and 0.501 of MCC on Data Class, NB scores 0.804 of F1 and 0.413 of MCC on God Class, LR and MLP score 0.874 of F1 and 0.677 of MCC on Feature Envy, and LR scores 0.825 of F1 and 0.557 of MCC on Long Method.

(3) The MCC values of the best-performing classifiers on our data sets are 28.36%-56.34% lower than that on the original data sets. The Precision, Recall, and F1 values are reduced by 9.31%-15.66%, 10.04%-18.95%, and 10.45%-21.21%, respectively. As noted by Nucci et al. Nucci et al. (2018), if the distribution of metrics for smelly instances differs significantly from the distribution of metrics for non-smelly instances, then any machine learning technique may easily distinguish between the two classes. However, this does not accurately reflect real-world situations where the boundary between the structural characteristics of smelly and non-smelly code components is not always clearly defined. Additionally, recent research Palomba et al. (2018) has found that smelly instances make up only a small portion of the total instances in a software system, with approximately one third of the instances in the original dataset being smelly instances. The results imply that the classifiers cannot properly detect the smelliness of software instances in a more realistic case where software instances contain more than one type of smell. Therefore, there is still plenty of room for performance improvement of the CSD models built with the seven classifiers.

Summary 1: The F1 and MCC values of the best-performing classifiers are 10.45%-21.21% and 28.36%-56.34% lower than those on the original data sets. The best performance on different code smells is achieved by different classifiers, i.e., NB on Data Class and God Class, LR on Feature Envy and Long Method.

5.2. Do imbalanced learning techniques improve the performance of code smell detection models?

Motivations: While a systematic literature review Azeem et al. (2019) has reviewed the use of machine learning techniques in CSD, we focus specifically on the literature review of imbalanced learning techniques used in CSD. Therefore, our comparison of detection performance is centered on the differences between various imbalanced learning techniques, rather than a deeper exploration of the impact of imbalanced learning techniques on the detection performance of machine learning classifiers. The imbalanced ratio of our data sets ranges from 20.03% to 27.03%. The previous studies Pecorelli et al. (2020); Alkharabsheh et al. (2022) show that CSD models trained on imbalanced data sets might lead to inaccurate prediction results. Therefore, we aim to investigate whether imbalanced learning techniques can alleviate the class imbalance problem to improve the performance of CSD models.



(a) Data Class

(b) God Class

(c) Feature Envy

(d) Long Method

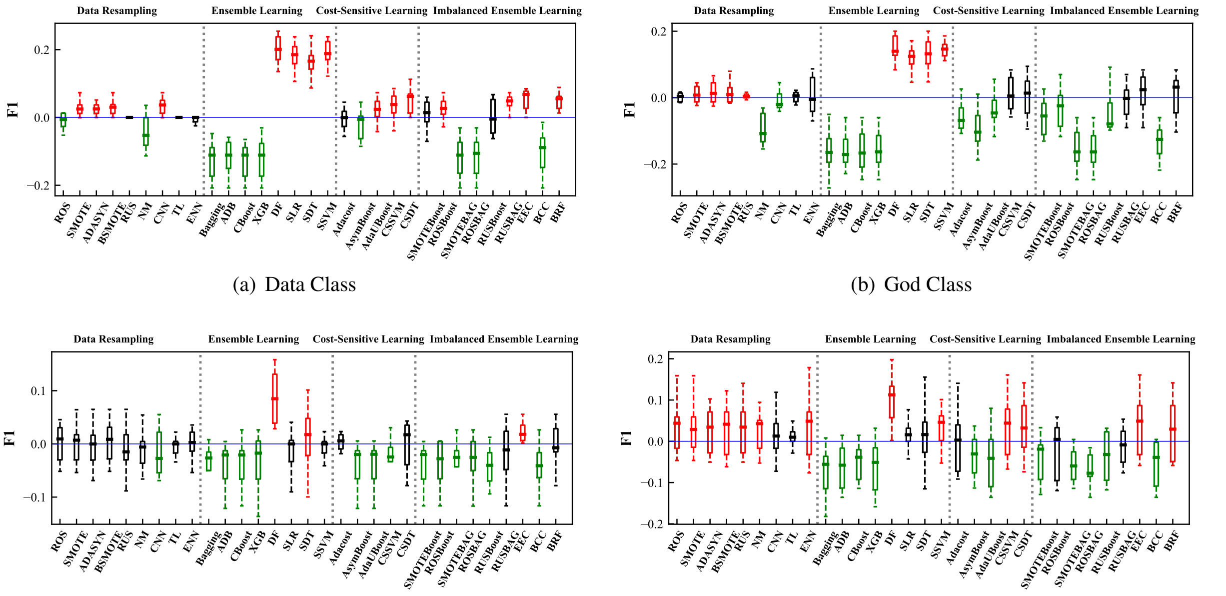
Figure 3: The performance difference between the imbalanced learning techniques and None in terms of Recall.

Approaches: We apply the nine data resampling techniques to our data sets and then employ the best-performing classifier on each data set to build CSD models (i.e., NB on Data Class and God Class, LR on Feature Envy and Long Method). We apply the eight ensemble learning techniques, the five cost-sensitive learning techniques, and the nine imbalanced ensemble learning techniques to directly build CSD models. Figures 2, 3, 4, and 5 show the distribution of performance difference between the imbalanced learning techniques and None in terms of Precision, Recall, F1, and MCC, respectively. We employ both statistic tests (i.e., Wilcoxon signed-rank test and Cliff's δ) and plot boxplots to examine the significant difference, where the red box means the corresponding technique significantly outperforms None, the green box means the corresponding technique significantly performs worse than None, and the black box means no significant difference between the corresponding technique and None. In order to make the boxplots more clear, we abbreviate AdaBoost, CatBoost, XGBoost, Deep Forest, StackingLR, StackingDT, StackingSVM, SMOTEBagging, ROSBagging, RUSBagging, EasyEnsemble Classifier, and BalanceCascade Classifier as ADB, CBoost, XGB, DF, SLR, SDT, SSVM, SMOTEBAG, ROSBAG, RUSBAG, EEC, and BCC. Table 8 shows the top-3 data resampling techniques with the best performance and the improvement ratio compared with None.

Results: (1) There is great variability in the performance of the imbalanced learning techniques compared with None. Not all imbalanced learning techniques can enhance the performance, and an only average of 32.26%, 29.03%, 30.65%, and 34.68% of the imbalanced learning techniques have a significant positive effect on CSD across our data sets in terms of Precision, Recall, F1, and MCC, respectively.

(2) In terms of Precision and Recall, no data resampling techniques have a significant positive effect on Data Class and Feature Envy. However, 77.78% and 88.89% data resampling techniques significantly enhance the performance on Precision, and 44.44% and 77.78% data resampling techniques significantly improve the Recall value on God Class and Long Method. 12.50%-50.00% ensemble learning techniques, 20.00%-40.00% cost-sensitive learning techniques, and 11.11%-33.33% imbalanced ensemble learning techniques significantly improve the performance in terms of Precision on the four data sets. 12.50%-50.00% ensemble learning techniques significantly improve the performance in terms of Recall on the four data sets. 60.00% and 40.00% cost-sensitive learning techniques and 66.67% and 22.22% imbalanced ensemble learning techniques significantly increase the Precision value on Data Class and Long Method. But no cost-sensitive learning and imbalanced ensemble learning techniques have a significant positive effect on God Class and Feature Envy.

(3) In terms of F1, no data resampling techniques have a significant positive effect on Feature Envy. However,



(a) Data Class

(b) God Class

(c) Feature Envy

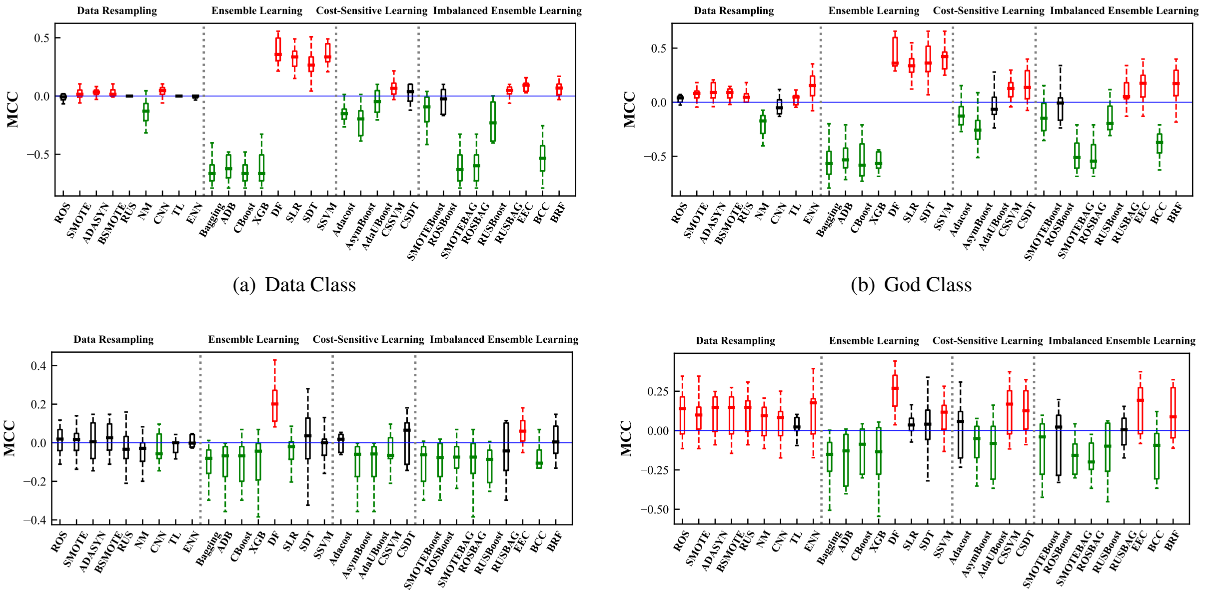
(d) Long Method

Figure 4: The performance difference between the imbalanced learning techniques and None in terms of F1.

Table 8
The top-3 imbalanced learning techniques with the best performance of each data set. (The bold font means the corresponding technique significantly outperforms None.)

data set	Techniques	Precision	Recall	F1	MCC
Data Class	DF	**0.966**(10.78%)	**0.964**(30.80%)	**0.964**(26.18%)	**0.892**(78.04%)
	SSVM	**0.954**(9.40%)	**0.953**(29.31%)	**0.953**(24.74%)	**0.854**(70.46%)
	SLR	**0.950**(8.94%)	**0.950**(28.90%)	**0.949**(24.21%)	**0.839**(67.47%)
	None	0.872	0.737	0.764	0.501
God Class	DF	**0.956**(16.02%)	**0.953**(20.03%)	**0.953** (18.53%)	**0.850**(105.81%)
	SSVM	**0.953**(15.66%)	**0.952**(19.90%)	**0.951**(18.28%)	**0.838**(102.91%)
	SDT	**0.941**(14.20%)	**0.937**(18.01%)	**0.937**(16.54%)	**0.797**(92.98%)
	None	0.824	0.794	0.804	0.413
Feature Envy	DF	**0.959**(8.24%)	**0.958**(9.11%)	**0.958**(9.99%)	**0.889**(31.31%)
	EEC	**0.910**(2.71%)	0.882(0.46%)	**0.888**(1.95%)	**0.744**(9.90%)
	CSDT	**0.894**(0.90%)	0.867(-1.25%)	0.872(0.11%)	0.705(4.14%)
	None	0.886	0.878	0.871	0.677
Long Method	DF	**0.934**(11.99%)	**0.921**(11.23%)	**0.923**(11.88%)	**0.808**(45.06%)
	EEC	**0.903**(8.27%)	**0.855**(3.26%)	**0.864**(4.73%)	**0.703**(26.21%)
	CSSVM	**0.898**(7.67%)	**0.851**(2.78%)	**0.851**(3.15%)	**0.691**(24.06%)
	None	0.834	0.828	0.825	0.557

44.44%, 44.44%, and 77.78% data resampling techniques significantly enhance the performance on Data Class, God Class, and Long Method, respectively. 50.00%, 50.00%, 25.00%, and 25.00% ensemble learning techniques significantly improve the performance on Data Class, God Class, Feature Envy, and Long Method, respectively. 60.00% and 40.00% cost-sensitive learning techniques significantly improve the performance on Data Class and Long Method, respectively. But no cost-sensitive learning techniques have a significant positive effect on God Class and Feature Envy. 44.44%, 11.11%, and 22.22% imbalanced ensemble learning techniques significantly improve the performance on Data Class, Feature Envy, and Long Method, respectively. However, no imbalanced learning techniques have a significant positive effect on God Class.



(a) Data Class

(b) God Class

(c) Feature Envy

(d) Long Method

Figure 5: The performance difference between the imbalanced learning techniques and None in terms of MCC.

(4) In terms of MCC, no data resampling techniques have a significant positive effect on Feature Envy. However, 44.44%, 66.67%, and 88.89% data resampling techniques significantly enhance the performance on Data Class, God Class, and Long Method. 50.00%, 50.00%, 12.50%, and 25.00% ensemble learning techniques significantly improve the performance on Data Class, God Class, Feature Envy, and Long Method, respectively. 20.00%, 40.00%, and 40.00% cost-sensitive learning techniques significantly improve the performance on Data Class, God Class, and Long Method, respectively. But no cost-sensitive learning techniques have a significant positive effect on God Class and Feature Envy. 33.33%, 33.33%, 11.11%, and 22.22% imbalanced ensemble learning techniques significantly improve the performance on Data Class, God Class, Feature Envy, and Long Method, respectively.

(5) StackingSVM is the second best performing technique on Data Class and God Class, which increases the F1 and MCC values by 18.28%-24.74% and 70.46%-102.91%. EasyEnsemble achieves the second best performance on Feature Envy and Long Method, which increases the F1 and MCC values by 1.95%-4.73% and 9.90%-26.21%. Deep forest is stable and always obtains the best performance in terms of all evaluation metrics on all four data sets. It increases the Precision, Recall, F1, and MCC values by 8.24%-16.02%, 9.11%-30.80%, 9.99%-26.18%, and 31.31%-105.81% across four data sets. The main reason for high performance is that deep forest is adept in handling code smell features by applying multi-grained scanning, and its cascade forest structure is an ensemble of decision tree and random forest, which produces more accurate CSD results under layer-by-layer forests training.

Summary 2: Not all imbalanced learning techniques can enhance the performance of CSD models, but deep forest always shows statistically significant improvement of 9.99%-26.18% and 31.31%-105.81% in terms of F1 and MCC.

5.3. What is the best data resampling technique for CSD?

Motivations: Some researchers might only use the dataset-level imbalanced learning technique (i.e., data resampling) as a preprocessing method to alleviate the imbalance problems and then employ more advanced machine learning algorithms (e.g., deep neural networks) to build CSD models. The main reason is that the machine learning algorithms are difficult to embed the algorithm-level imbalanced learning techniques (i.e., ensemble learning, cost-sensitive learning, and imbalanced ensemble learning). In addition, the previous studies Pecorelli et al. (2020); Alkharabsheh et al. (2022) show that using SMOTE can not significant improve the detection performance on code smell data sets. Therefore, we discuss the reason for non-improvement and conduct a study to explore the practical

Table 9
The top-3 data resampling techniques with the best performance of each data set. (The bold font means the corresponding technique significantly outperforms SMOTE.)

data set	Techniques	Precision	Recall	F1	MCC
Data Class	CNN	**0.877(0.57%)**	0.778(0.91%)	0.801(0.88%)	**0.541(3.05%)**
	BSMOTE	0.876(0.46%)	0.771(0.00%)	0.794(0.00%)	0.535(1.90%)
	ADASYN	**0.876(0.46%)**	0.767(-0.52%)	0.791(-0.38%)	0.531(1.14%)
	SMOTE	0.872	0.771	0.794	0.525
God Class	ENN	**0.890(4.58%)**	0.788(-1.13%)	0.811(-0.25%)	**0.571(17.73%)**
	ADASYN	**0.860(1.06%)**	0.803(0.75%)	0.819(0.74%)	**0.510(5.15%)**
	BSMOTE	0.858(0.82%)	0.803(0.75%)	0.819(0.74%)	**0.508(4.12%)**
	SMOTE	0.851	0.797	0.813	0.485
Feature Envy	BSMOTE	**0.893(1.02%)**	0.869(0.00%)	0.874(0.11%)	**0.702(2.63%)**
	ROS	0.885(0.11%)	0.871(0.23%)	0.874(0.11%)	**0.691(1.02%)**
	SMOTE	0.884	0.869	0.873	0.684
	ADASYN	0.887(0.34%)	0.857(-1.38%)	0.863(-1.15%)	0.681(-0.44%)
Long Method	ROS	0.889(1.02%)	0.855(0.59%)	0.862(0.70%)	0.677(3.68%)
	ENN	0.892(1.36%)	0.849(-0.12%)	0.858(0.23%)	0.675(3.37%)
	RUS	0.887(0.80%)	0.848(-0.24%)	0.856(0.00%)	0.670(2.60%)
	SMOTE	0.880	0.850	0.856	0.653

effect of data resampling techniques on our data sets using an individual research question.

Approaches: We use the nine data resampling techniques to balance our data sets and compare the performance with SMOTE. We show the top-3 data resampling techniques with the best performance and the improvement ratio compared with SMOTE in Table 9.

Results: (1) SMOTE does not achieve the best performance on all data sets. The top-1 data resampling techniques achieve the following MCC performance: CNN scores 0.541 on Data Class, ENN scores 0.571 on God Class, BSMOTE scores 0.702 on Feature Envy, and ROS scores 0.677 on Long Method. The MCC values of the top-1 data resampling techniques are 2.63%-17.73% higher than SMOTE on our data sets. In addition, CNN significantly outperforms SMOTE on Data Class; ENN, ADASYN, and BSMOTE significantly outperform SMOTE on God Class; BSMOTE and ROS significantly outperform SMOTE on Feature Envy. The Precision, Recall, and F1 improvements of the top-1 data resampling techniques range from 0.57%-4.58%, -1.13%-0.91%, and -0.25%-0.88%, respectively.

(2) The best classifiers are DT, RF, RF, and LR on the original four data sets, respectively. On the preprocessed data sets by SMOTE, the F1 and MCC values of DT reduce by 1.23% and 3.37% on Data Class; The MCC value of RF increases by 0.53% on God Class, but the improvement is not significant; The F1 and MCC values of RF and LR do not change on Feature Envy and Long Method, respectively. In summary, using SMOTE to balance the original data sets cannot significantly improve the performance of CSD models. The main reasons are as follows. Since the feature distribution between smelly instances and non-smelly instances in the original data sets is quite different, the best classifiers on each data set have achieved relatively high performance in terms of F1 and MCC. Using SMOTE cannot further improve the feature distribution difference, so there is little room for improvement on each data set by SMOTE.

Summary 3: The best data resampling techniques are CNN, ENN, BSMOTE, and ROS on the four data sets, respectively.

5.4. How efficient are the best-performing imbalanced learning techniques?

Motivations: Since the top-3 imbalanced learning techniques and the top-3 data resampling techniques need to build multi base classifiers or increase the size of training data, the training time increases accordingly. Hence, we aim to explore the efficiency of the imbalanced learning techniques.

Approaches: We conduct experiments on a personal computer with an Intel i7-8750H CPU and 16GB RAM. We record the time cost of ten rounds of 10-fold cross-validation on all data sets. We select the top-3 imbalanced learning techniques and the top-3 data sampling techniques to show their time cost in Table 10.

Results: The stacking ensemble learning techniques consume a long time about 20s. They use the seven machine learning classifiers as the base classifiers to generate the first detection predictions, then select a classifier as the

Table 10
The efficiency of top-3 imbalanced learning techniques.

data set	Techniques	Time Cost(s)
Data Class	DF	2.32
	SSVM	21.97
	SLR	22.11
	CNN	0.01
	BSMOTE	0.01
	ADASYN	0.01
God Class	DF	2.37
	SSVM	20.86
	SDT	20.56
	BSMOTE	0.01
	ROS	0.01
	SMOTE	0.01
Feature Envy	DF	2.03
	EEC	1.90
	CSDT	4.25
	ENN	0.20
	ADASYN	0.24
	BSMOTE	0.25
Long Method	DF	2.12
	EEC	1.90
	CSSVM	0.02
	ROS	0.24
	ENN	0.20
	RUS	0.19

meta-classifier to take the outputs of the first predictions as inputs and produce the final detection results. Therefore, the stacking ensemble takes the longest training time. Among the top-3 imbalanced learning techniques, the best-performing technique, deep forest, can achieve a low time cost about of 2s. The time costs of the top-3 data resampling techniques are less than 0.3s. The main reason is that we use only a classifier to build the detection model rather than train multi classifiers. We employ NB as the classifier on Data Class and God Class and use the default parameter. We employ LR as the classifier on Feature Envy and Long Method and use the grid search algorithm to find the optimal parameter of LR. Therefore, the time cost of the top-3 data resampling techniques on Feature Envy and Long Method is higher than that of Data Class and God Class.

Summary 4: The best-performing imbalanced learning technique deep forest runs fast, and the time cost of the top-3 data resampling techniques is less than 0.3s.

6. Discussion

6.1. The differences between our findings and previous findings

As outlined in our review of the literature in Section 2, we will now discuss the results obtained by recent researches Fontana et al. (2016); Nucci et al. (2018); Pecorelli et al. (2020); Alazba and Aljamaan (2021); Alkharabsheh et al. (2022). Fontana et al. Fontana et al. (2016) conducted an investigation using 16 different machine learning algorithms to detect four specific code smells. They found that all of the algorithms achieved high performance and concluded that the use of machine learning for the detection of these code smells can result in high accuracy (> 96%). However, they also noted that the imbalanced nature of the data (low prevalence of code smells) resulted in inconsistent performance across the dataset, and this issue should be addressed in future research. Nucci et al. Nucci et al. (2018) acknowledged that the work of Fontana et al. presented a new perspective on CSD, but pointed out that it only considers instances affected by a single type of code smell in the data sets used for training and testing the machine learning models. In order to address this limitation, Nucci et al. replicated the investigation using a data set containing instances of multiple types of code smells. The results showed that when the machine learning-based CSD model was tested on the Nucci et al.'s data sets, the F1-measure value was 90% lower than previously reported. As previously mentioned by Fontana et al., the highly imbalanced nature of CSD data sets can make machine learning techniques unsuitable. Pecorelli et al. Pecorelli et al. (2020) examined the impact of five imbalanced learning methods for CSD in object-oriented systems. They found that the methods can not significantly improve the performance. Alazba et al. Alazba and Aljamaan (2021) explored the use of stacking ensemble models for CSD. They applied information gain feature selection to select relevant features and evaluated the performance of 14 individual classifiers on the Fontana et al.'s data sets. They then constructed three stacking ensembles using these individual classifiers as base models and three different meta-classifiers (LR, SVM, and DT) and compared the detection performance of the stacking ensembles to the individual models. The results showed that the stacking ensembles using LR and SVM meta-classifiers consistently achieved high detection performance for both class-level and method-level code smells. Alkharabsheh et al. Alkharabsheh et al. (2022) investigated the effectiveness of machine learning for CSD and conducted a comparative investigation to assess the impact of data imbalance on accuracy and behavior during CSD. They carried out experiments using 28 machine learning classifiers, and found that most classifiers achieved high performance, with CatBoost showing particularly good performance. In their experiments, the imbalanced learning technique SMOTE did not significantly improve the detection of God Class code smell.

In contrast to previous studies, our study aims to evaluate the effectiveness of a variety of imbalanced learning techniques for CSD and to identify the best performing technique. To this end, we combine the same level code smell data sets and remove the the redundant and conflicting instances. Then, we compare the performance of 31 imbalanced learning techniques across four evaluation metrics on different code smells, taking into account both detection performance and time cost. Our results show that the difference between our study and previous investigations.

(1) Our study shows that machine learning techniques do not achieve particularly high detection performance on our data sets, with the F1 value and MCC value of the best-performing classifier dropping by 10.45%-21.21% and 28.36%-56.34%, respectively, compared to the results of Fontana et al. However, the performance drop is less than that reported by Nucci et al. Nucci et al. (2018) on their data sets, possibly due to the removal of redundant and conflicting instances in our data sets.

(2) In our investigation of 31 imbalanced learning techniques, we find that the deep forest consistently improve the detection of the four code smells across four evaluation metrics, with low time cost and good performance on most code smells compared to the other top-3 imbalanced learning techniques. This differs from the results of Pecorelli et al. Pecorelli et al. (2020). We also find that the use of stacking ensemble technique can significantly improve the performance of CSD, as previously concluded by Alazba et al. Alazba and Aljamaan (2021). However, their study only examines stacking ensemble technique and does not find that the deep forest could achieve better detection performance. Specifically, the performance of the deep forest is better than that of the stacking ensemble model on our data sets.

(3) Additionally, our empirical results do not support the claim that SMOTE is the best data resampling technique for handling imbalanced code smell data sets, as it does not consistently achieve the best detection performance across all code smells when compared to other data resampling techniques. Contrary to the conclusion of Alkharabsheh et al. Alkharabsheh et al. (2022), we find that SMOTE has a detection performance improvement for all three code smells (Data Class, God Class, and Long Method) in our data sets. However, even though SMOTE improves detection performance, it is not able to become the best data resampling technique for CSD.

6.2. Threats to validity

Construct validity. To ensure the generalizability of our conclusions and avoid bias as far as possible, we have employed a wide range of 31 imbalanced learning techniques and seven classifiers in our study. While we recognize the existence of other imbalanced learning techniques, their inclusion in our study is left for future work. To minimize technical errors, all steps in our empirical research process, including data preprocessing, model building, and result analysis, were carried out using third-party Python libraries. For instances, Scikit-learn [3] for model construction, Imbalanced-learn [4] and Imbalanced-ensemble [5] for imbalanced learning technique implement. To minimize potential errors in the experimental process, all experiments are conducted using 10-fold cross-validation.

Internal validity. One potential threat to the validity of our results is the choice of evaluation metrics used to assess the performance of CSD models. Using only one evaluation metric can be biased and lead to misleading conclusions. To address this issue, we employ both threshold-dependent (i.e., Precision, Recall, and F1) and threshold-

[3] https://github.com/scikit-learn/scikit-learn
[4] https://github.com/scikit-learn-contrib/imbalanced-learn
[5] https://github.com/ZhiningLiu1998/imbalanced-ensemble

independent (MCC) evaluation metrics, which are commonly used in recent CSD studies Pecorelli et al. (2019b); Dewangan et al. (2021); Boutaib et al. (2021).

External validity. Our findings are based on data sets provided by Fontana et al. Fontana et al. (2016), which are derived from 74 systems in the Qualitas corpus. While these code smell data sets are widely used in recent studies Nucci et al. (2018); Aljamaan (2021); Jain and Saha (2021), we cannot guarantee that our conclusions will hold true for other data sets. Current researches Azeem et al. (2019); Pecorelli et al. (2020); Alkharabsheh et al. (2022) in CSD tends to treat code smells as a binary classification problem, meaning that a code block is either classified as having a particular smell or not having that smell. This means that when a code block contains multiple code smells, separate models must be used to predict the presence of each smell. However, it may be more practical to detect all code smells in a single model, as this would eliminate the need for multiple separate predictions. This study represents an area for future research in the field. To facilitate replication of our experiment by future researchers and mitigate the effects of model variability caused by randomly processing data, we have made our source code available and employed 10-fold cross-validation, a well-established validation method in machine learning. Despite these efforts, we acknowledge that our conclusions may not be generalizable to other data sets.

Conclusion validity. To optimize the performance of our classifiers, we use grid search to identify the optimal hyper-parameters. We set the smelly ratio of the data resampling techniques to 0.5, as this is a common practice and has been used in previous empirical CSD studies Pecorelli et al. (2020); Alkharabsheh et al. (2022). For other imbalanced learning techniques, we use the default hyper-parameters provided by the corresponding third-party libraries. Future research may involve further exploration of hyper-parameter tuning.

6.3. Implications

Based on the experimental results, we summarize the following implications for future CSD studies.

(1) **Our results indicate that the blind application of imbalanced learning techniques may not always be optimal, and researchers and practitioners should consider using deep forest when building CSD models.** While previous studies, such as that by Alazba et al. Alazba and Aljamaan (2021), have found that the stacking ensemble technique performs well on CSD, our results show that deep forest consistently outperforms the best-performing stacking ensemble technique across all data sets in terms of Precision, Recall, F1, and MCC by 0.36%-1.38%, 0.13%-1.49%, 0.25%-1.44%, and 2.90%-7.58%, respectively. Based on our findings, the deep forest technique is recommended for improving CSD performance. However, not all imbalanced learning techniques are effective in enhancing CSD, and an average of 32.26%, 29.03%, 30.65%, and 34.68% of the techniques we tested have a significant positive impact on CSD across our data sets in terms of Precision, Recall, F1, and MCC, respectively. For example, the Near Miss technique perform similarly or worse than the None technique on Data Class, God Class, and Feature Envy. Therefore, we advise researchers and practitioners to carefully select appropriate imbalanced learning techniques, such as deep forest, to achieve more accurate CSD models.

(2) **Researchers and practitioners should consider following the data sets processing methods outlined by Nucci et al. Nucci et al. (2018) and Alazba et al. Alazba and Aljamaan (2021)**. Nucci et al. Nucci et al. (2018) have noted that the instances in original data sets often only contain one type of code smell, which does not accurately reflect real-world scenarios and makes it easier for CSD models to distinguish smelly instances. As a result, experimental results based on these data sets may be misleading. To address this issue, Nucci et al. merged data sets at the same level to create new data sets. However, Alazba et al. Alazba and Aljamaan (2021) pointed out that this merging strategy can lead to redundant and conflicting instances. Therefore, we believe that two different code smell data sets can be merged into a new code smell data set, as suggested by Nucci et al., but it is important to also follow Alazba et al.'s recommendation to remove redundant and conflicting instances from the resulting data sets. By following these steps, future research can more accurately study the performance of machine learning techniques in CSD and obtain more reliable results.

(3) **Researchers and practitioners should consider using a more effective data resampling technique to preprocess code smell data sets instead of relying solely on SMOTE.** Previous empirical studies Pecorelli et al. (2020); Alkharabsheh et al. (2022) have indicated that SMOTE does not consistently improve detection performance on original code smell data sets, likely due to its inability to significantly increase the difference in feature distribution. As shown in Figure 4 and Figure 5, SMOTE does achieve significant improvement over the None technique on Data Class, God Class, and Long Method across our data sets, and obtains non-significant improvement on Feature Envy. Therefore, researchers and practitioners may still consider using SMOTE as a preprocessing method in line with previous studies Akhter et al. (2021); Alkharabsheh et al. (2021); Gupta et al. (2021); Jain and Saha (2021); Stefano et al.

(2021); Khleel and Nehéz (2022); Kovačević et al. (2022); Nanda and Chhabra (2022); Yedida and Menzies (2022), but should also consider exploring other techniques that may be more effective. Our results in Section 5.3 demonstrate that SMOTE does not consistently achieve the best performance on all four data sets, and the top-performing data resampling technique outperforms SMOTE by 2.63%-17.73% in terms of MCC. In other words, SMOTE is not the best data resampling technique. Therefore, we recommend that researchers and practitioners consider using alternative techniques to preprocess code smell data sets.

7. Conclusion

In our investigation, we evaluate the impact of 31 imbalanced learning techniques on seven machine learning classifiers for CSD. To obtain more effective data sets, we follow the methods of Nucci et al. and Alazba et al. to combine two different code smell data sets and remove redundant and conflicting instances. We use Precision, Recall, F1, and MCC to comprehensively evaluate the performance of the detection models and applied both the Wilcoxon signed-rank test and Cliff's δ to analyze the results statistically. Our findings show that the effect of the imbalanced learning techniques varies across different code smells, and deep forest consistently improves performance across all data sets. Therefore, we recommend that researchers and practitioners use deep forest to enhance detection performance. Additionally, we find that certain data resampling techniques, such as CNN, ENN, BSMOTE, and ROS, perform better than SMOTE. Future research should carefully consider the selection of different data resampling techniques to preprocess different code smell data sets.

References

Acuna, E., Rodriguez, C., 2004. The treatment of missing values and its effect on classifier accuracy, in: Classification, clustering, and data mining applications. Springer, pp. 639–647.

Akhter, N., Rahman, S., Taher, K.A., 2021. An anti-pattern detection technique using machine learning to improve code quality, in: 2021 International Conference on Information and Communication Technology for Sustainable Development (ICICT4SD), IEEE. pp. 356–360.

Al-Shaaby, A., Aljamaan, H., Alshayeb, M., 2020. Bad smell detection using machine learning techniques: a systematic literature review. Arabian Journal for Science and Engineering 45, 2341–2369.

Alazba, A., Aljamaan, H., 2021. Code smell detection using feature selection and stacking ensemble: An empirical investigation. Inf. Softw. Technol. 138, 106648.

Aljamaan, H., 2021. Voting heterogeneous ensemble for code smell detection, in: Wani, M.A., Sethi, I.K., Shi, W., Qu, G., Raicu, D.S., Jin, R. (Eds.), 20th IEEE International Conference on Machine Learning and Applications, ICMLA 2021, Pasadena, CA, USA, December 13-16, 2021, IEEE. pp. 897–902.

Alkharabsheh, K., Alawadi, S., Kebande, V.R., Crespo, Y., Delgado, M.F., Taboada, J.A., 2022. A comparison of machine learning algorithms on design smell detection using balanced and imbalanced dataset: A study of god class. Inf. Softw. Technol. 143, 106736.

Alkharabsheh, K., Crespo, Y., Delgado, M.F., Viqueira, J.R.R., Taboada, J.A., 2021. Exploratory study of the impact of project domain and size category on the detection of the god class design smell. Softw. Qual. J. 29, 197–237.

Alkharabsheh, K., Crespo, Y., Manso, M.E., Taboada, J.A., 2019. Software design smell detection: a systematic mapping study. Softw. Qual. J. 27, 1069–1148.

Amorim, L., Costa, E., Antunes, N., Fonseca, B., Ribeiro, M., 2015. Experience report: Evaluating the effectiveness of decision trees for detecting code smells, in: 26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015, IEEE Computer Society. pp. 261–269.

Azeem, M.I., Palomba, F., Shi, L., Wang, Q., 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. Inf. Softw. Technol. 108, 115–138.

Bergstra, J., Bengio, Y., 2012. Random search for hyper-parameter optimization. Journal of machine learning research 13.

Boutaib, S., Bechikh, S., Palomba, F., Elarbi, M., Makhlouf, M., Said, L.B., 2021. Code smell detection and identification in imbalanced environments. Expert Systems with Applications 166, 114076.

Bryton, S., e Abreu, F.B., Monteiro, M.P., 2010. Reducing subjectivity in code smells detection: Experimenting with the long method, in: e Abreu, F.B., Faria, J.P., Machado, R.J. (Eds.), Quality of Information and Communications Technology, 7th International Conference on the Quality of Information and Communications Technology, QUATIC 2010, Porto, Portugal, 29 September - 2 October, 2010, Proceedings, IEEE Computer Society. pp. 337–342.

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.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., Figueiredo, E., 2016. A review-based comparative study of bad smell detection tools, in: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, pp. 1–12.

Fontana, F.A., Mäntylä, M.V., Zanoni, M., Marino, A., 2016. Comparing and experimenting machine learning techniques for code smell detection. Empir. Softw. Eng. 21, 1143–1191.

Fontana, F.A., Zanoni, M., 2017. Code smell severity classification using machine learning techniques. Knowl. Based Syst. 128, 43–58.

Fontana, F.A., Zanoni, M., Marino, A., Mäntylä, M., 2013. Code smell detection: Towards a machine learning-based approach, in: 2013 IEEE

International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013, IEEE Computer Society. pp. 396–399.

Fowler, M., 2018. Refactoring: improving the design of existing code. Addison-Wesley Professional.

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, in: International Conference on Advanced Information Networking and Applications, Springer. pp. 120–132.

Jain, S., Saha, A., 2021. Improving performance with hybrid feature selection and ensemble machine learning techniques for code smell detection. Sci. Comput. Program. 212, 102713.

John, G.H., Kohavi, R., Pfleger, K., 1994. Irrelevant features and the subset selection problem, in: Cohen, W.W., Hirsh, H. (Eds.), Machine Learning, Proceedings of the Eleventh International Conference, Rutgers University, New Brunswick, NJ, USA, July 10-13, 1994, Morgan Kaufmann. pp. 121–129.

Kampenes, V.B., Dybå, T., Hannay, J.E., Sjøberg, D.I., 2007. A systematic review of effect size in software engineering experiments. Information and Software Technology 49, 1073–1086.

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, 1725–1735.

Khomh, F., Vaucher, S., Guéhéneuc, Y., Sahraoui, H.A., 2009. A bayesian approach for the detection of code and design smells, in: Choi, B. (Ed.), Proceedings of the Ninth International Conference on Quality Software, QSIC 2009, Jeju, Korea, August 24-25, 2009, IEEE Computer Society. pp. 305–314.

Khomh, F., Vaucher, S., Guéhéneuc, Y., Sahraoui, H.A., 2011. BDTEX: A gqm-based bayesian approach for the detection of antipatterns. J. Syst. Softw. 84, 559–572.

Kim, D.K., 2017. Finding bad code smells with neural network models. International Journal of Electrical and Computer Engineering 7, 3613.

Kovačević, A., Slivka, J., Vidaković, D., Grujić, K.G., Luburić, N., Prokić, S., Sladić, G., 2022. Automatic detection of long method and god class code smells through neural source code embeddings. Expert Systems with Applications 204, 117607.

Kreimer, J., 2005. Adaptive detection of design flaws. Electron. Notes Theor. Comput. Sci. 141, 117–136.

Li, Y., Zhang, X., . Multi-label code smell detection with hybrid model based on deep learning .

Liu, H., Xu, Z., Zou, Y., 2018. Deep learning based feature envy detection, in: Huchard, M., Kästner, C., Fraser, G. (Eds.), Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018, ACM. pp. 385–396.

Maiga, A., Ali, N., Bhattacharya, N., Sabane, A., Guéhéneuc, Y., Aïmeur, E., 2012. SMURF: A svm-based incremental anti-pattern detection approach, in: 19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012, IEEE Computer Society. pp. 466–475.

Moha, N., Guéhéneuc, Y., Duchien, L., Meur, A.L., 2010. DECOR: A method for the specification and detection of code and design smells. IEEE Trans. Software Eng. 36, 20–36.

Mundfrom, D.J., Whitcomb, A., 1998. Imputing missing values: The effect on the accuracy of classification. .

Nanda, J., Chhabra, J.K., 2022. Sshm: Smote-stacked hybrid model for improving severity classification of code smell. International Journal of Information Technology , 1–7.

Nucci, D.D., Palomba, F., Tamburri, D.A., Serebrenik, A., Lucia, A.D., 2018. Detecting code smells using machine learning techniques: Are we there yet?, in: Oliveto, R., Penta, M.D., Shepherd, D.C. (Eds.), 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018, IEEE Computer Society. pp. 612–621.

Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., De Lucia, A., 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, in: Proceedings of the 40th International Conference on Software Engineering, pp. 482–482.

Palomba, F., Bavota, G., Penta, M.D., Oliveto, R., Lucia, A.D., Poshyvanyk, D., 2013. Detecting bad smells in source code using change history information, in: Denney, E., Bultan, T., Zeller, A. (Eds.), 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013, IEEE. pp. 268–278.

Patnaik, A., Padhy, N., 2021. Does code complexity affect the quality of real-time projects? detection of code smell on software projects using machine learning algorithms, in: Proceedings of the International Conference on Data Science, Machine Learning and Artificial Intelligence, pp. 178–185.

Pecorelli, F., Di Nucci, D., De Roover, C., De Lucia, A., 2019a. On the role of data balancing for machine learning-based code smell detection, in: Proceedings of the 3rd ACM SIGSOFT international workshop on machine learning techniques for software quality evaluation, pp. 19–24.

Pecorelli, F., Nucci, D.D., Roover, C.D., Lucia, A.D., 2020. A large empirical assessment of the role of data balancing in machine-learning-based code smell detection. J. Syst. Softw. 169, 110693.

Pecorelli, F., Palomba, F., Nucci, D.D., Lucia, A.D., 2019b. Comparing heuristic and machine learning approaches for metric-based code smell detection, in: Guéhéneuc, Y., Khomh, F., Sarro, F. (Eds.), Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019, IEEE / ACM. pp. 93–104.

Quinlan, J.R., 1986. Induction of decision trees. Mach. Learn. 1, 81–106.

Rahad, K., Badreddin, O., Mohsin Reza, S., 2021. The human in model-driven engineering loop: A case study on integrating handwritten code in model-driven engineering repositories. Software: Practice and Experience 51, 1308–1321.

Rahman, M.M., Riyadh, R.R., Khaled, S.M., Satter, A., Rahman, M.R., 2018. Mmruc3: A recommendation approach of move method refactoring using coupling, cohesion, and contextual similarity to enhance software design. Software: Practice and Experience 48, 1560–1587.

Sabir, F., Palma, F., Rasool, G., Guéhéneuc, Y.G., Moha, N., 2019. A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems. Software: Practice and Experience 49, 3–39.

Sharma, T., Efstathiou, V., Louridas, P., Spinellis, D., 2021. Code smell detection by deep direct-learning and transfer-learning. Journal of Systems and Software 176, 110936.

Shen, L., Liu, W., Chen, X., Gu, Q., Liu, X., 2020. Improving machine learning-based code smell detection via hyper-parameter optimization, in:

2020 27th Asia-Pacific Software Engineering Conference (APSEC), IEEE. pp. 276–285.
Sousa, B.L., Bigonha, M.A., Ferreira, K.A., 2019. An exploratory study on cooccurrence of design patterns and bad smells using software metrics. Software: Practice and Experience 49, 1079–1113.
Stefano, M.D., Pecorelli, F., Palomba, F., Lucia, A.D., 2021. Comparing within- and cross-project machine learning algorithms for code smell detection, in: Ampatzoglou, A., Feitosa, D., Catolino, G., Lenarduzzi, V. (Eds.), MaLTeSQuE@ESEC/SIGSOFT FSE 2021: Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution, Athens, Greece, 23 August 2021, ACM. pp. 1–6.
Tong, H., Lu, W., Xing, W., Liu, B., Wang, S., 2022. Shse: A subspace hybrid sampling ensemble method for software defect number prediction. Information and Software Technology 142, 106747.
Tsantalis, N., Chatzigeorgiou, A., 2009. Identification of move method refactoring opportunities. IEEE Trans. Software Eng. 35, 347–367.
Vaucher, S., Khomh, F., Moha, N., Guéhéneuc, Y., 2009. Tracking design smells: Lessons from a study of god classes, in: Zaidman, A., Antoniol, G., Ducasse, S. (Eds.), 16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France, IEEE Computer Society. pp. 145–154.
Wilcoxon, F., 1992. Individual comparisons by ranking methods, in: Breakthroughs in statistics. Springer, pp. 196–202.
Yedida, R., Menzies, T., 2022. How to improve deep learning for software analytics (a case study with code smell detection). arXiv preprint arXiv:2202.01322 .
Young, W., Weckman, G., Holland, W., 2011. A survey of methodologies for the treatment of missing values within datasets: Limitations and benefits. Theoretical Issues in Ergonomics Science 12, 15–43.
Yu, J., Mao, C., Ye, X., 2021. A novel tree-based neural network for android code smells detection, in: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), IEEE. pp. 738–748.
Zhang, Y., Dong, C., 2021. Mars: Detecting brain class/method code smell based on metric–attention mechanism and residual network. Journal of Software: Evolution and Process , e2403.
Zhang, Y., Ge, C., Hong, S., Tian, R., Dong, C., Liu, J., 2022. Delesmell: Code smell detection based on deep learning and latent semantic analysis. Knowledge-Based Systems , 109737.
Zhou, Y., Yang, Y., Lu, H., Chen, L., Li, Y., Zhao, Y., Qian, J., Xu, B., 2018. How far we have progressed in the journey? an examination of cross-project defect prediction. ACM Transactions on Software Engineering and Methodology (TOSEM) 27, 1.