Received: 22 May 2021 | Revised: 16 October 2021 | Accepted: 25 March 2022

DOI: 10.1002/smr.2454

RESEARCH ARTICLE - METHODOLOGY

Journal of Software: Evolution and Process WILEY

Severity classification of software code smells using machine learning techniques: A comparative study

Ashraf Abdou | Nagy Darwish

Department of Information Systems and Technology, Faculty of Graduate Studies for Statistical Research, Cairo University, Cairo, Egypt

Correspondence
Nagy Darwish, Department of Information Systems and Technology, Faculty of Graduate Studies for Statistical Research, Cairo University, Cairo, Egypt.
Email: nagyrd@cu.edu.eg

Abstract

Code smell is a software characteristic that indicates bad symptoms in code design which causes problems related to software quality. The severity of code smells must be measured because it will help the developers when determining the priority of refactoring efforts. Recently, several studies focused on the prediction of design patterns errors using different detection tools. Nowadays, there is a lack of empirical studies regarding how to measure severity of code smells and which learning model is best to detect the severity of code smells. To overcome such gap, this paper focuses on measuring the severity classification of code smells depending on several machine learning models such as regression models, multinominal models, and ordinal classification models. The Local Interpretable Model Agnostic Explanations (LIME) algorithm was further used to explain the machine learning model's predictions and interpretability. On the other side, we extract the prediction rules generated by the Projective Adaptive Resonance Theory (PART) algorithm in order to study the effectiveness of using software metrics to predict code smells. The results of the experiments have shown that the accuracy of severity classification model is enhanced than baseline and ranking correlation between the predicted and actual model reaches 0.92–0.97 by using Spearman's correlation measure.

KEYWORDS
code smell, correlation measures, oversampling, regression models, severity classification, software refactoring

1 | INTRODUCTION

Code smell is a structural characteristic of software that indicates aspect in code or design that can cause problems in software maintenance. Code smell is not a bug in the system as it does not prevent the program from functioning, but it may increase the risk of software failure or performance slowdown. Notably, the early prediction of code smell severity during the development phase is very important especially in large-scale source code projects. Developers must determine which types of code smells have the highest priority to be fixed in the maintenance plan. Also, detecting the severity of code smells will help them to determine the priorities of the refactoring plan.[1] Refactoring process is very critical to remove the bad of code smells and to enhance software maintenance and quality. Fowler and Beck[2] present a definition of 22 types of bad code smells in source code, and they provide some of refactoring operations to fix them.

Maintenance of large-scale software is very difficult due to large size and high complexity of code in such software. Besides that, more than 80% of the total software cost is dedicated to the maintenance and 60% of time is spent on understanding the code.[3] Therefore, several code smells detectors have been proposed recently to automatically detect code smells in source code.[4,5] The types of code smells detectors differ

according to the main algorithm used, for example, search-based techniques[6] versus metric-based techniques[7] and the metric-based techniques are classified into process metrics[8] and product metrics.

According to recent studies,[9] some limitations have been highlighted for code smell detectors. In most cases, determining code smells is subjective and has different interpretations due to the ambiguity of its definitions.

In addition, different types of code smell detectors have different rules and metrics and any change in the thresholds of these metrics will cause decreasing or increasing of the number of detected code smells accordingly. Otherwise, many results of code detectors can be detected as a false positive smell, not representing real problems, because these detectors depend on information related to the domain, the context, and the size, but it neglects the information related to the system design.

In this paper, different types of machine learning techniques were applied to overcome the previous limitations because of their ability to detect the severity of code smell and construct code smell detection rules and evaluate any new candidates by building the learning models.[10] On the other side, this paper aims to not only classify the absence or presence of code smells but also aims to measure their severity. Each type of code smell can determine the software quality by some specific characteristics. So, the severity can be defined according to the amount of these characteristics in every type of code smell instance. The severity is an integer value representing the smells strength, for example, if God Class code smell is complex and large, it will be classified as a type of high severity code smell. In this paper:

- A large set of object-oriented metrics are extracted from a large heterogeneous software system at method, class, project, and package levels.
- Machine learning algorithms used in this paper describe the relationship between a set of independent variables (predictors) such as a metrics that are employed to estimate the values of dependent variables (e.g., degree of severity of code smells).
- The selection phase depends on stratified random sampling method to ensure a homogenous selection of instance on several projects and assign to each valuated instance one of four possible values of code severity as "no smell," "nonsevere smell," "medium," and "severe smell."
- A set of instances have been evaluated and labeled according to the severity of code smells, and the selected instances are utilized to train a set of machine learning algorithms.

Most of code smell detection techniques depend on heuristic approach. It has some drawbacks as follows:

- It depends on manually specified heuristics to map selected code smell metrics into binary classification of presence or absence of code smell.[11]
- Difficulties in finding good thresholds to be used for detection.
- Code smells identified by heuristics methods are subjectively interpreted by developers, therefore in most cases developers may not consider them as actual problems.
- It is difficult to manually construct the optimal heuristic by selecting the best features and finding good thresholds to be used for detection in heuristic detection approach.
- Other empirical studies propose selecting different metrics by different people with several heuristics to define the same code smells, which lead to in low matching between different detectors.[9]

To overcome on these limitations, machine learning techniques has been utilized in this paper like random forest, naïve Bayes, and support vector machine (SVM), to build the complex relationship mapping between predictions and code metrics.[10] Code metrics are utilized as predictors of the smelliness of code artifacts. One of advantages of using machine learning models is the ability to exploit any code characteristics to define the severity degree of code smells. In this case, the developers do not need to determine of code smells severity because it will be recognized by learning models. This makes the method highly flexible and independent from a metrics thresholds or explicit classification rules. The main contributions of this study are the following:

- Proposed learning model for classification the severity of code smells based on different machine learning algorithms and SMOTE resample technique and we extracted the prediction rules generated by the Projective Adaptive Resonance Theory (PART) algorithm in order to study the effectiveness of using software metrics to predict code smells.
- Researchers adopted the local Interpretable Model Agnostic Explanations (LIME) algorithm[12] to provide a better comprehension of how the machine learning model makes its decision and defines the features that influence the prediction models decisions
- Source code, the appendixes, and relevant datasets for our research will be available for researchers includes more than 20 classifiers with different parameters, which can be replicate and extend in future.
- A large scale of this empirical study based on extensive experiment is applied by using binary, multinomial, ordinal, and regression models.

The main objective of this paper is to evaluate the performance of code smell severity classification models after applying SMOTE resample technique and to select the best models to implement this task by using a comparative study. Besides that, we extracted the

prediction rules to study the effectiveness of using software metrics to predict code smells. In this paper, the benchmark dataset has been taken from a previous study proposed by Fontana and Zanoni.[13] We utilized different types of learning models such as multiclass model, nominal values, and regression model. Four types of code smell have been tested (God Class, Data Class, Feature Envy, and Long Method) with 420 data points collected from 76 open-source systems by using 10 of machine learning algorithms. Each instance in the dataset is represented by a software metric; the instance consists of 61 software metrics in the class-level code smells, and 82 software metrics in the method-level code smells. In this study, we focus on the prediction of the severity of code smells by using a set of characteristics for each code smell instance.

For example, if the code smell class is very large and complex, it will be called "God Class" and it will be labeled with high severity. The rest of the article is organized as follows. Section 2 introduces some of related works, Section 3 describes the code smell learning model, Section 4 explains the results of the experiments and answers to research questions, and Section 5 includes the conclusion and future works.

2 | RELATED WORK

Nowadays, we have several approaches and tools for detecting and measuring code smells severity depending on machine learning techniques.[13] This section presents some important works in this domain.

Marco et al.[14] have built the prediction model by applying 32 supervised machine learning approaches (16 main algorithms and their boosting techniques) to detect code smells. Marco focused on the four types of code smells (Large Class, Data Class, Long Class, and Feature Envy), and extensive experiments have been applied on 74 systems to choose the best algorithms with the best parameters to detect the code smells. The results show that the experimented algorithms such as J48, naïve Bayes, and random forest obtained the high performance that reached to range from 96.64% to 99.02% of accuracy regardless of the types of code smell.

Fontana and Zanoni[13] have proposed developer driven code smell prioritization concept based on machine learning technique and real developers perceived criticality. Fabiano compared his approach with Fontana.[14] The experimental results showed that the algorithms such as SVM and logistic regression are better than the previous work.

Fontana et al.[14] have introduced an empirical study to classify code smell severity based on machine learning by applying different models spanning from regression to multinomial model, plus ordinal classification. In this context, Fontana ranked the correlation between predicted and actual severity of code smells by using Spearman's measure. The best results have been achieved by using decision tree and random forest with boosting technique and the Spearman measure reaches to 0.96.[13]

Catolino et al.[3] have presented another empirical study by adding the intensity index metrics as additional predictor to capture the severity of code smell. Fontana compared the performance with other models that use antipattern metrics. The results of experiments showed that the performance by using F-measure is better than the antipattern-based metrics models.

Liu et al.[15] have applied a deep learning approach with bootstrap ensemble learning to detect four types of code smells. Liu used automatic detection method to generate a large set of training samples and his approach is evaluated on 10 well-known open-source applications. The evaluation results show that the proposed model is significantly better than existing approaches and the F-measure values have improved for four types of code smells detectors.

Abdou and Darwish[16] have proposed a comparative study between single and ensemble learners to predict software defects. SMOTE resample technique has been used as a preprocessing step to solve data imbalance problem. The different techniques of ensemble learning have been utilized to predict of software defects. The results show that ensemble learning using rotation forest is better than other ensemble techniques.

Marinescu and Ratiu[17] have measured the criticality of code smell and ranked it by using flow impact score measure. Marinescu takes into consideration three factors when measuring the impact of code smells: (I) the effect of design flow type on the criteria of good design (moderate complexity, high cohesion, proper encapsulation and low coupling); (II) granularity, by defining the type of design entity that has the smell effects (e.g., method and class); and (III) severity, by assigning a severity score for each type of critical symptoms.

As show in Table 1, the main differences between previous works and our approach are that they did their experiments by considering only two until 15 systems and they usually have limited numbers of machine learning algorithms. This study focuses on four types of code smells, and we have considered 74 systems for the analysis and the validation. In addition, more than 10 experiments have been applied in this study by different of machine learning algorithms (JRIP, J48, RF, LIBSVM linear, LIBSVM RBF, LIBSVM Sig, SMO, logistic regression, RBF network, naïve Bayes). We have taken in consideration the effect of using resample techniques and boosted variant of the algorithms. The total number of algorithms used in experiments rises to 30. Moreover, a huge number of different parameter settings have been tested to measure its effect on accuracy and we extracted the prediction rules for each type of code smell according to its severity. On the other side, we applied LIME algorithm to define the features that influence the prediction model decisions. In the next section, we will present the proposed learning model and briefly explain each type of classifiers that will be used in this study.

TABLE 1 Comparison between previous related work and our study

Previous studies	Smells	Algorithms	Methodology			
			Resample	Feature selection	Ensemble learning	Systems
Barbez et al.[18]	2: God Class and Feature Envy	ML based ensemble method to aggregate 5 existing antipatterns detection approaches.	No	No	Yes	8
Pecorelli et al.[19]	3: Blob, Complex Class, Spaghetti Code and Shotgun Surgery	5 types of ML algorithms	No	Yes	No	9
Hadj-Kacem and Bouassida[20]	3: Blob, Feature Envy and Long Method	Semantic based Approach and Deep Learning	No	No	No	9 open-source systems
Catolino et al.[3]	4: God Class, Data Class, Brain Method, Shotgun Surgery, Dispersed Coupling and Message Chains	6 types of ML algorithms	No	Yes	Yes	15
Liu et al.[15]	4: Feature Envy, Long Method, Large Class, and Misplaced Class	Deep Learning	No	No	Yes	10 open-source applications
Pecorelli et al.[19]	4: God Class, Spaghetti Code, Class Data Should Be Private, Complex Class and Long Method	Five most used ML algorithms	Yes	Yes	No	13
Our study	4: God Class, Data Class, Feature Envy and Long Method	10 types of ML algorithms	Yes	No	Yes	76

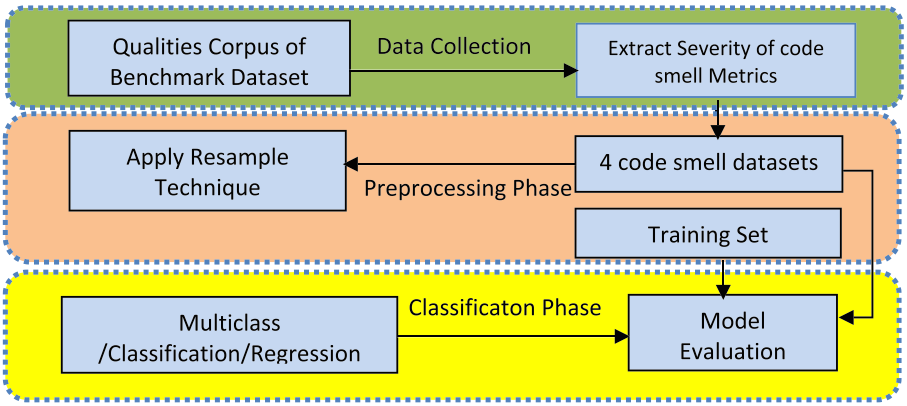

FIGURE 1 The proposed learning model

3 | THE PROPOSED LEARNING MODEL

Different types of machine learning classifiers have been setup to detect the severity of code smells. SMOTE resample technique has been used to solve imbalance problem in training data. The main goal of this study is to measure the performance of selected classifiers to predict the severity of code smells and compare the results of multi-nominal, ordinal and regression models with the experimental results of previous state-of-the-art as in Fontana and Zanoni[13] in order to select the best model to predict the severity degree of code smell.

Code smell detection model has been defined as a supervised learning classification; the types of the classification are methods or classes. In this study, each class or method in the datasets will be represented as one feature vector, where each vector implicates the software metrics linked to the respective class variables, plus subject, the dependent and independent variables have been defined as follows.

- Dependent variables

The severity of code smell is defined as dependent variable. The subject of the severity classification is method or class based on the kind of code smells need to be identified. This information already available in the considered dataset is used in the experiments. We converted the

integers value of severity label from numeric to nominal values in the set {1: no smell, 2: nonsevere smell, 3: medium, 4: severe smell} to be suitable for classification.

- Independent variables

A large set of software metrics are used as independent variables in this study. The selected software metrics are widely used at different levels of granularity. The metrics belong to types (interface or classes), methods, and packages. These granularity levels are composed of package that contain project including class and method. Plus, these metrics conclude several aspects of the code, that is, method, cohesion, size, inheritance, coupling, and complexity. Each feature vector is utilized to train the selected learning models, and it contains the metrics related to the respective method or class, and the metrics join to all its containers. The proposed learning model is shown in Figure 1, and it is composed of different phases that include the data collection, preprocessing, and the classification phase; we will show the details of each phase in the next sections.

3.1 | Data collection phase

The datasets used in this study are provided by Fontana and Zanoni,[13] and it is defined as benchmark data. The datasets belong to different domains and are composed of 76 systems of different sizes written in Java. A large set of object-oriented metrics is computed on the 76 systems of the Qualities Corpus (QC), and it covers method, class, package, and project level. Some of these metrics were defined according to the aspect of software quality like complexity, size, and coupling. Other metrics depend on the count of the number of elements in packages or classes. The metrics selected in this study are reported in Table 7. The code smells have been defined at the level of method or class. In method level, we decided to detect Feature Envy and Long Method,[13,21] whereas at class level, we detect Data Class and God Class.[21]

We can summarize the steps of data collection and how code smell metrics was extracted and code smell detection rules as the following:

Step 1: Collection of a large repository of heterogeneous software system

Qualities Corpus (QC) of system that was collected by Tempero et al.[22] has been considered in this study. This corpus and the collection of 2012040lr have been used which composed of 111 systems written in java, characterized by different size and belonging to different of application domains. Around 74 systems have been selected for analysis in this study; the reason of this selection is that the systems must be compliable to correctly compute the metrics values. In addition, manual work has been done for adding all the missing third-party libraries to each of selected system, allowing the resolution of class dependencies issues. Table 2 reports to the overall size measure of the selected systems. The high number of heterogeneous systems that are used in this study is fundamental to guarantee the results of machine learning process do not depend on a specific dataset, also to allow to generalize the obtained results. The main objective of collecting datasets from huge heterogeneous systems is to guarantee that the number of systems had been analyzed is suitable for antipattern (or code smells) detection with machine learning algorithms.

Step 2: Metrics extraction

A large set of object-oriented metrics has been computed on all 74 systems of QC. The selected metrics are at method, class, project, and package level: A set of metrics consists of metrics needed by the exploited advisors, plus other standard metrics cover different aspects of code such as cohesion, coupling, complexity, and size. The number of metrics chosen in this study are widely used in previous studies,[23–25] and the chosen metrics with full name and its granularity are listed in Table 8. Some of the metrics are standard and others are customized version from standard; custom metrics have been defined to catch other structural properties of the source code listed in Table 8 with star * symbol. The definitions are depending on the combination of modifiers such as private, public, static, and protected on methods and attributes, for example, the number of abstract methods, or the number of static and public attributes).

All metrics have been extracted through a tool that has been developed by Fontana et al.,[14] which parses the source code of java projects through the Eclipse library. The tool is called design feature and metrics for Java (DFMC4J),[14] and it is designed to be integrated as a library in

TABLE 2 Summary of systems characteristics

Number of systems	Number of packages	Number of classes	Number of methods	Lines of code
74	3420	51,826	40,436	6,785,568

other projects. First, the user can request information on specific entities or on a category of entities such as all class of a system or all methods of a system. Then, all the computation of the metrics that were defined and computed by DFMC4J will be tested.

Step 3: Select of a set of code smell detection rules

Supervised machine learning needs a large of training dataset contains labeled instance that specify how much a method or class is affected by a code smell. The set of correct labeled assignment is called "oracle."[14] As we have no external benchmark data yet, this oracle must be created by manual code analysis. In this study, we work on heterogeneous and a large of datasets to avoid the huge effort of human resources to create correct labeling includes all the extracted source code elements. So, the datasets have been created by using a sampling approach.

In this study, a stratified sampling has been applied for the available instance based on a set of hints called "advisors." An advisor[14] is a deterministic rule implemented locally or in an external tool, which gives a classification of a code smell elements (method or class) and telling us if it is a code smell or not. The idea is advisors should approximate the labels by aggregating their suggestions, and in this case, they should be able to sample more code elements affected by code smells. In this study, some requirements have been formulated to determine how to select available advisors from the literature:

- Different advisors (for the same code smell) must use different approaches or rules as much as possible to avoid possible naive correlations between similar rules.
- An advisor can be implemented as an available external tool. A rule implemented by an automatic detection tool has some benefits: It reuses functionalities having a wide agreement and diffusion. Also, it eliminates the possibility of misinterpretation of the rule definition.
- Advisors can be also defined by research papers and the whole detection rule must be clearly described in the original source in this case, to be implemented accordingly.
- Advisors can be implemented in external tools, and the data should be exported in documented format, and they must be suitable for batch computation,

The analysis of the literature related to code smell detection has been done including related freely available tools. The advisors selected for each code smell are reported in Table 3

In our case study, some of well-known tools have been eliminated because they are not freely available or it cannot run-in batch without a manual configuration of the projects to analyze, or they use detection rules that are too similar to other selected ones. As shown in Table 3, the number of advisors have been considered: iPlasma[26] and PMD which are two free tools; Anti-Pattern Scanner and Fluid Tool. On other side, the detection rule for the Long Method smell defined by Marinescu[27] have been considered also in this study. For the three last cases (God Class, Feature Envy and Data Class) the detection rules have been implemented and following the respective references.

Step 4: Usage of advisors for code elements sampling

Advisors are used to give a classification for a code smells elements and provide hints of presence of a code smell or not based on well-defined deterministic rule or external tool. We will consider advice as positive when an advisor reports elements are affected by code smell, and negative otherwise. Assume that we have at least two advisors for each code smell and three advisors for two smells, a suitable way to provide a large oracle would be by aggregating the advisor values, for example, considering as code smells all methods or classes having at least half of positive advice or a number of positive advice higher than a predefined threshold. The problem is that the real performance of each single advisor is unknown. We have taken into consideration that advisors are subjected to error also we do not want to bias oracle towards the rules implemented in the advisors. Stratified sampling method has been applied to comply with our requirements and avoid these problems. Suppose that we have a separate dataset for each code smell containing only the elements eligible to be affected by the smell (like method for Long Method, classes for God Class), consider also that the value of each advisor is available for all elements of the dataset.

The stratified sampling procedure is shown in Figure 2, and it organized by the following steps:

TABLE 3 Code smell and advisors

Code smell	Advisors: Detection rules
Long Method	iPlasma,PMD
Data Class	iPlasma,Fluid Tool, Anti-Pattern Scanner
God Class	iPlasma,PMD
Feature Envy	iPlasma,Fluid Tool

1. Each dataset element is annotated with two values (the name of the project containing it and a number n, counting the number of advisors reporting a positive evaluation), that is, that indicate the element affected by a code smell.
2. Database elements have been grouped and sorted by project N.
3. Indexing the groups in the defined order, an instance n is randomly sampled and removed from each group, if a group is empty, it is skipped.
4. The cycling will restart from the first group, until found the target number of positive instances until $n = 0$.
5. Then loop to next project contain new database elements and repeat the same steps for label instance until $N = 0$.

The advantage of stratified sampling is that it will increase the chance of building a dataset not biased and represents different domains (in case of the projects belong to different domains) with sufficient number of affected entities and keep different characteristics (exploiting the N grouping parameters). Also, by applying this method, we will give the same probability of selection to groups of instances related to the different projects and having different likelihood given by the advisors of being affected by a code smell.

3.2 | Preprocessing phase

Preprocessing step is applied by using the resampling technique which is called Minority over Sampling Technique (SMOTE) to improve the accuracy of learners.

3.2.1 | Resampling technique

Resampling technique has been applied in this study to handle data imbalance problem related to classification severity of code smells to obtain more balance of class distribution. The main problem with imbalanced classification is that it has a small number of training examples of the minority class for a model efficacious learn the decision boundary. SMOTE[6] has been applied by duplicate number of examples from the minority class in the training dataset before fitting a training model. SMOTE generate the training examples by using linear interpolation techniques, and it selects samples of feature space for each target class and its closest neighbors. Then, it produces new examples that merge feature of the target

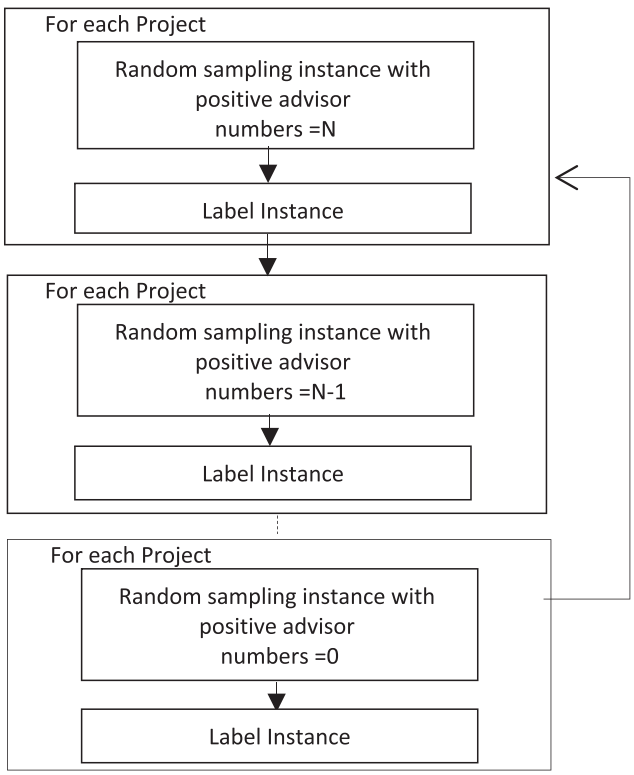

FIGURE 2 Label process in stratified sampling method

case with neighbor features. In general, resampling technique improves the performance of classification as it provides enough training data for classifiers. Furthermore, it balances the skewed class distribution and keeps the data reliable at the same time.

The resampling techniques[28] are classified into two types: under sampling and over sampling techniques as shown in Figure 3. The undersampling technique decreases the time of training, but it causes loss of information, whereas the oversampling technique increases the size of the training set and therefore the training time of the model will be increased, but it does not cause information loss and it is prone to overfitting due to copying the same information. In this study, researchers will use the minority oversampling technique. The challenges related to imbalanced dataset are biased prediction and misleading accuracy. SMOTE processes are as the following:

1- Identify the feature vector and its nearest neighbors in dataset;
2- Calculate the coordinate differences between the two neighbors;
3- These differences are multiplied by the random numbers between 0 and 1;
4- Identify a new object on the line segment by adding the random number to feature vector;
5- Repeat the process for identified feature vectors.

Finally, we obtain training set that is normalized in size, by randomly removing a negative instance (if needed), until the balance of the training datasets is two thirds negative (and one third positive) instances. As we know, machine learning algorithms tend to perform badly on very unbalanced datasets; for that, we applied resample techniques.

3.2.2 | Label assignment criteria

The labeling has been done by applying a severity classification of code smell that depends on ordinal scale, and it provides more information than a traditional binary classification. Measuring the severity classification of code smells has two advantages: First, it helps the developers in prioritizing their work plan according to the degree of severity for code smells because the results of prediction model will rank the class or methods of code smells according to their severity degree. Second, it will add more information about severity degree than the traditional binary classification method, which just determines the absence or presence of code smell. In this study, the severity value of code smell is assigned manually to each evaluated instance by using one of the values as shown in Table 4.

The labeling evaluation was done by three MSc students who have studied independently the code smell definitions to comprehend the definitions Then, they performed a short manual labeling exercise, and discussed their evaluations, to reach an agreement for code smell definitions The output of this phase was a set of guidelines and rules for determining the most relevant aspects that must be considered for each type of code smells. On the other side, to detect each type of code smells, we must apply a number of rules that are shown in Table 5 to extract each type of smells according to its characteristics and rule as the following.

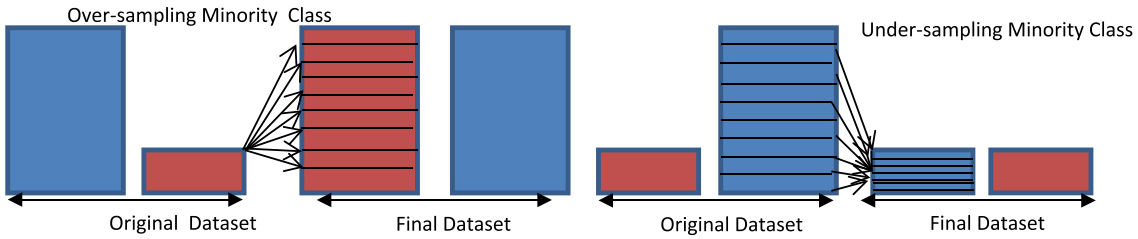

FIGURE 3 Types of resample techniques

TABLE 4 Code smell severity degree

Severity values	Degree	Description
4	Severe smell	The smells are present with high value of complexity, size or coupling.
3	Medium	The smell class is present in method or class.
2	Nonsevere smell	The method or class is partially affected by the smell.
1	No smell	The method or class is free of code smell

TABLE 5 Detection rules for code smells

Type of code smell	Detection rules
God Class	God Classes usually contain large and complex methods.
	God Classes usually expose large number of methods.
	God Classes are large.
	God Classes tend to access many attributes from many other classes also attributes accessed using accessors, must be high.
Data Class	Data Classes mainly expose accessor methods.
	Data Classes must not contain complex methods.
	Data Classes must be very simple, and they can expose few nonaccessor methods.
	The attributes of a Data Class must be either public or exposed through accessor methods.
Long Method	Long Methods contain many lines of code.
	Long Methods tend to be complex.
	Long Methods tend to have many parameters.
	Long Methods access many attributes include the number of used variables, large part of the attributes declared in the class and the attributes accessed through an accessor must be high.
Feature Envy	Feature Envies access more foreign attributes than local ones.
	The number of foreign attributes, directly used by a method, considering also attributes accessed through accessors, must be high. feature envies mainly use the attributes of a small number of foreign classes.

3.3 | Classification phase

In this study, several learning models have been applied to classify the severity of code smell. We have applied more than 10 classifiers with different parameters, and we compare our results with baseline.[13]

3.3.1 | Machine learning models used in our study

The most commonly machine learning models have been compared in our experiments to classify the severity of code smells, such as multiclassification, ordinal, and regression models. Each type of prediction model will be briefly explained in the following subsections.

- Multiclassification model

Multiclassification model[29] is used to solve the problem of classifying instances of more than two classes. Decomposition technique has been utilized to decompose the multiclass problem into multiple binary subproblems. In multiclassification model, each binary subproblem can be learned individually by binary classifier, and then, it will be combined into an ensemble to solve the multiclass problem.

- Ordinal classification model

Ordinal classification model[30] is used as a form of multiclass classification that includes the ordering relationship between the classes. In this study, the target is to classify the severity of code smells. For this reason, the order value has been used in classification model to define the degree of severity using four ordinal variables to rank the code smells according to the severity degree. Ordinal classification seeks to enhance the multinominal model by adding the ordering information into prediction model.

- Regression model

Regression model[13] is used for approximating the predictor's values into a continuous target variable. Ordinal values in regression model are represented as numbers such as integer or float value and regression model is applied to fit the ordinal variable of severity of code smells and predict the target value by using the map function.

- Local surrogate (Local Interpretable Model-Agnostic Explanation [LIME])

LIME is a local surrogate model proposed by Ribeiro et al.[31] LIME is used to locally mimic the predictions of the black-box machine learning model. The main objective is to understand why the machine learning model made a certain prediction. For example, LIME explains what happen to the predictions when we give variations of our data into the machine learning model. We use the LIME algorithm to provide us with more understanding of how the model makes its decision and which features influence the model to make its decision. The LIME algorithm calculates each feature's importance by generating a set of data points around each feature individually. Then it applies the trained model and observes the impact of each data point in the prediction output. The local importance of each feature is the correlation between the feature and its effects on the prediction. Finally, each feature's global importance is calculated and provided as output.

Suppose that x is an instance that needs to be explained and f is a black box model the loss function of the LIME method is shown in Equation (1). The explanation model for instance x is the model g, the minimized loss function L reflects the closeness of the prediction results between the proxy model g and the black-box model f. π defines the domain range when sampling around instance x and G represents the set of interpretable machine learning algorithms; omega (g) represents the model complexity of the interpretable model g.

$$\underset{g \in G}{\text{Explanation}}(x) = \underset{g}{\arg\min}\, L(f, g, \pi_x) + \Omega(g) \tag{1}$$

The key steps of training the local surrogate model as follows:

1. Select your instance of interest for which you want to have an explanation of its black box prediction.
2. Perturb your dataset and get the black box predictions to generate new points.
3. Apply the black-box model to predict the new points.
4. Set a weight for each point based on the closeness of new points to the target instance.
5. Use an interpretable machine learning algorithm to train a local surrogate model on the data set with weights.
6. Explain the prediction by interpreting the local model. The local surrogate model trained in our experiment is a random forest. Therefore, the coefficients of the regression model are used to measure the feature importance scores of a change.

The objective of our study is to assess which machine learning model can classify the severity of code smells with high accuracy and we have number of research questions that need to be addressed:

R1). What is the effect of resampling technique on the performance of the multinomial models when the results compared to a baseline?

We aim at providing an experiment result that measures the capabilities of resample technique in improving the prediction of severity code smell in conjunction with multinomial model than previous baseline study.[13]

R2). Does the using of ordinal model improve the results of the multinomial model? The researchers aim at providing an experiment result to compare the effect of ordinal model compared to multinomial model for each type of code smells on the same dataset.

R3). What is the effect of applying regression model on the accuracy of results? With this research question the researchers aim at measuring the accuracy of severity prediction after applying the regression model.

R4). What are the best experiment results reached in general in this study? This research question aims at providing the best results of all models used in this study before and after applying resampling with and without boosting technique. The following subsections report the methodological steps that will be conducted to address RQs. Table 6 provides a brief about each classifier used in the experiments according to its type, along with a short description.

4 | DATABASE DESCRIPTION AND EXPERIMENTAL RESULTS

4.1 | Dataset description

In this study, we selected four types of datasets for each type of code smells and related the degree of severity. The datasets are composed of one feature vector for each class or method, where each vector contains the software metrics associated to the respective subject, plus the class variable. In Table 7, we report the composition of the datasets, with the number of instances assigned to each severity level. Two method-based

smells have a different balance than the two class-based smell and the less frequent severity level in the dataset is 2. This makes positive smell instances more prevalent in the last two datasets than in the other two. We applied SMOTE technique as preprocessing on the datasets as explained in Section 3.2.

The definition of each type of code smell in Table 7 is as follows:

- Feature Envy: The Feature Envy method of code smell uses its own Data Class plus more data from other classes.
- Long Method: The Long Method of code smell has many codes that are complex and difficult to understand, because it uses large amount of data from other classes.
- Data Class: This type of code smell refers to classes that store data without complexity in functionality.
- God Class: The God Class code smell is tending to have many codes with degree of complexity, and it implements different functionality based on a large amount of data from other classes.

4.1.1 | Software metrics within datasets

The numbers of software metrics are selected and classified to cover several aspects of software quality control, such as Coupling, Complexity, Cohesion, Size, Inheritance, and Encapsulation. The selected metrics are reported in Table 8, which are based on the count of number of elements in packages or classes declared with some language modifiers, for example, private, static, and final.

Furthermore, our study is based on other types of custom metrics that are used to catch other structural aspects of source code and it is based on the number of static and public attributes in class or packages. The custom metrics are declared by some modifier keywords such as (private, public, protected, static, and final) on method and attributes. Custom metrics are also reported in Table 8.

TABLE 6 List of classifiers used in the experiments

Name of classifiers	Type	Description
J48	Classification	J48 classifier[32] uses a set of labeled training data to build a decision tree by applying the concept of information entropy.
JRIP	Classification	JRIP is one of rule learner classifiers that uses incremental reducer pruning[14] to generate initial set of rules for the classes. It examines the classes in increasing size.
Random forest (RF)	Classification/Regression	RF[14] is an ensemble learning and it depends on a combination of many tree predictors to overcome the overfitting problem.
Sequential minimal optimization (SMO)	Classification	SMO algorithm[12] proposes to train the Support Vector Machine (SMO) and solve many optimization problems by using the analytic methods.
LIBSVM/SVR	Classification/Regression	SVM is used to maximize the margin and find optimal hyperplane that is used as linear separable plan between negative and positive examples.
Boosting	Ensemble Learners for Classification/Regression	Boosting[33] has been used to convert weak learner to strong one and to decrease the bias and variance in the predictive model.
Multinomial logistic regression	Classification	Logistic regression[34] is used to solve multiclass problems when dependent variables are nominal by using logistic regression analysis.
RBF neural network	Regression	RBF[35] is fast in learning and it produces exact interpolation. RBF depends on the theory of function approximation by using Gaussian Function.
Naïve Bayes (NB)	Classification	Naïve Bayes[34] depends on Bayes theorem and conditional probability. It considers all attributes in the training data independent and equally important.

TABLE 7 Dataset composition

	Severity			
Code smell	**1**	**2**	**3**	**4**
Feature Envy	280	23	95	22
Long Method	280	11	95	34
Data Class	151	32	113	124
God Class	154	29	110	127

TABLE 8 List of considered metrics within datasets

Quality dimension	Abbreviation	Name of metric	Granularity
Complexity	WMC	Weight Method Count	Class
	AMW	Average Method Weight	Class
	CYCLO	Cyclomatic Complexity	Method
	ATED	Access to Foreign Data	Method
	FDB	Foreign Data Providers	Method
	NOP	Number of Parameters	Method
	ATLD*	Access to Local Data	Method
	WMCNAMM*	Weighted Methods Count of Not Accessor or Mutator Methods	Class
Size	NOPK	Number of Packages	Projects
	NOM	Number of Methods	Project, Package, Class
	LOC	Lines of Codes	Project, Package, Class
	NOA	Number of Attributes	Class
Coupling	CBO	Coupling Between Object	Class
	FDB	Foreign Data Providers	Method
	NOAM	Number of Accessor Methods	Class
	CM	Changing Method	Method
	ATFD	Access to Foreign Data	Method
	RFC	Response for a Class	Class
	CC	Changing Classes	Method
	CFNAMM*	Called Foreign Not Accessor or Mutator Methods	Class, Method
Inheritance	NOI	Number of Interfaces	Project, Package
	NMO	Number of Method	Class
	NOC	Number of children	Class
	DIT	Depth of Inheritance Tree	Class
Encapsulation	LAA	Locality of Attribute Accesses	Method
	NOPA	Number of Public Attribute	Class
	NOI	Number of Interface	Project
Custom Metrics	NOFA	Number of Final Attributes	-
	NOPM	Number of Private Methods	
	NOSA	Number of Static Attributes	-
	NOD	Number of Default Attributes	-
	NOSM	Number of Static Methods	Method
	NOPVA	Number of Private Attributes	
	NOFNSM	Number of Final and non - Static Methods	
	NONFNSM	Number of Non-Final and non-Static	
	NODCM	Number of default constructor	

Note: Abbreviations with * symbols are custom metrics.

4.1.2 | Evaluation measurements

The decision of classifier can be defined by using four categories that are represented by the confusion matrix as shown in Figure 4. False positive (FP) is where the decision of predictor is positive, but it is not. True positive (TP) means the decision of predictor is positive, and it is positive. False negative (FN) is where the decision of predictor is negative, and it is positive. The experiments have been tested on four datasets; each one is considered as a code smell. The results of experiments are evaluated by three types of classification measures such as accuracy, root mean square error (RMSE), and mean absolute error (MAE). But in ordinal model, the Spearman measure is applied to rank the order of the correlation coefficient.

Accuracy is defined as the percentage of correctly classified examples against the total number of examples as shown in Equation (2).

$$Accuracy = \frac{TN + TP}{TP + FP + TN + FN}. \tag{2}$$

RMSE is a very common metric for numeric prediction in regression model. It is defined as the square root of the mean of the sum of the square of the misclassification error as in Equation (3).

$$\mathrm{RMSE} = \sqrt{\frac{\sum_{i=1}^{N} (d_i - z_i)^2}{N}}, \tag{3}$$

where d_i is the actual label of a data instance, N is the size of data, and z_i is the predicted label.

MAE refers to the mean of the sum of absolute differences between the actual and predicted targets on all instances of the test dataset as shown in Equation (4).

$$MAE = \frac{\sum_{i=1}^{N} |(d_i - z_i)|}{N} \tag{4}$$

Spearman's ρ metric is used to rank-order correlation coefficient and it assesses how much good prediction there is between the rank values of variables. Spearman is suited to evaluate the ranking correlation of the actual and predicted severity of code smells as shown in Equation (5).

$$\rho = \frac{6 \sum d_i^2}{n(n^2 - 1)}, \tag{5}$$

where ρ is Spearman's rank correlation coefficient. d_i^2 is the difference between the two ranks of each observation and n are defined as number of observations.

Kendall rank correlation τ: Kendall rank correlation[27] is a nonparametric test that measures the strength of dependence between two variables. As shown in Equation (6). we consider two samples, a and b, where each sample size is n, and we know that the total number of pairings with a b is $n\ (n - 1)\ /2$. The following formula is used to calculate the value of Kendall rank correlation where n_c = number of concordant and n_d = Number of discordant.

$$\tau = \frac{n_c - n_d}{\frac{1}{2} n(n-1)}. \tag{6}$$

4.1.3 | Experimental setup

Datasets used were initially created by Fontana and Zanoni[13] for our research study, which includes 74 JAVA systems out of 111 from Qualitas Corpus. We built different models for four types of code smells datasets considered in the study; so in the training dataset, each row represents

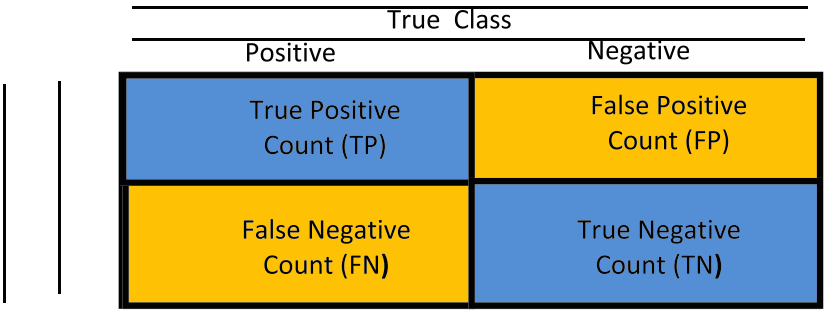

FIGURE 4 Confusion matrix

method or class instances, and it includes a certain type of code smell. For each type of code smell, a dataset has been created: two datasets for method-level smells (Long Method and Feature Envy) and two for class-level smells (God Class and Data Class). In addition, the numeric filed for severity degree has been converted to nominal feature that represents the labels that shows the severity degree of a code smells as follows: (1) if the method or class is free of code smells, (2) that means we do not have a severe code smell, (3) indicate the severity of code smell is medium, and (4) it means the occurrence of high severe code smell). To train the different models, we applied a 10-fold cross-validation and it was repeated 10 times to avoid sample bias problem the cross-validation $10 * 10 = 100$ of different evaluation has been used on different subset of datasets for each prediction model. During every repetition of cross validation, the dataset was split randomly into number of equal size partitions. The last partition is used as the test set and the remaining partitions are used as training set. A stratified sampling was applied, which means that each fold has the same proportion of the critical class. The performance evaluation experiments were implemented on an Intel Core device (TM) i7 with16 GB RAM. Researchers applied five standard performance measures: accuracy, RMSE, MAE, Spearman, and Kendall rank correlation. The datasets benchmark and the Python source code of the proposed LIME algorithm are freely available.*

4.2 | Experimental results and data analysis

4.2.1 | First experiments: Impact of resample technique and feature selection on code smell classification

The accuracy of machine learning-based approaches depends on the quality of the datasets used to train the algorithms. However, it may not be enough to build big-sized datasets that include all software domains and sizes to find a high accuracy. Many other factors that should also be considered are listed as follows:

1. The manner of dataset construction with balanced instances and metrics distribution.
2. The preprocessing steps, such as selecting the metrics in datasets using the feature selection methods, because building a dataset with the redundant features might cause inaccurate prediction models. In addition, the characteristics of each type of code smell.

For that, we have measured the impact of using resample technique and feature selection as preprocessing step at first. As we know, most of machine learning classification algorithms are sensitive to imbalanced training data. An imbalanced training data will bias the prediction classifier toward the more common class. This happens because the machine learning algorithms are usually designed to improve the accuracy by reducing the error. Thus, they do not consider the distribution of classes. In our case, we have listed the class distribution for each type of code smell after and before applying SMOTE. As we see in Tables 9–12, the class distribution of most of severity degree has been increased for the minor class in approximately 10:15% (as shown in tables with bold font) after applying SMOTE, which means it will increase the accuracy. The parameters of the SMOTE algorithm are tuned by five nearest neighbors and three values of random sampling:

On the other hand, the advantage of feature selection is that it can improve learning performance, build better generalizable models, lower computational complexity, and decreas required storage. Feature extraction maps the original feature space to a new feature space with lower dimensions by combining the original feature spaces. The parameters of feature selection are tuned, in our experiments we used the info gain that evaluates the worth of an attribute by measuring the information gain with respect to the class and ranking search technique that ranks attributes by their individual evaluations and the threshold used to discard some attributes is (−1.7).

Feature selection is a technique aimed to find the most influence features in the dataset by removing the redundant features to increase the performance. In other words, it selects features that are capable of discriminating samples that belong to different classes. The objective of using feature selection in base study is selecting the most relevant features that is not only for measuring the impact of feature selection in improving the accuracy of the model, but also for comprehending the software metrics that play a significant role in the code-smells prediction and producing the decision rules using software metrics as will be explained in next section. Filter methods has considered one common of feature selection methods, the disadvantages of filter approach are fast and independent of the classifier but ignore the feature dependencies and ignores the interaction with the classifier. In addition, it is not clear how to determine the threshold point for rankings to select only the required features and exclude noise.[36]

TABLE 9 Class distribution of data for Data Class

Severity label	Before resample	After resample
Severe smell	124	110
Smell	113	112
Nonsevere smell	**32**	**37**
No smell	151	161

TABLE 10 Class distribution of data for Feature Envy

Severity label	Before resample	After resample
Severe smell	**23**	28
Smell	105	93
Nonsevere smell	**15**	**28**
No smell	277	271

TABLE 11 Class distribution of data for God Class

Severity label	Before resample	After resample
Severe smell	127	140
Smell	110	102
Nonsevere smell	**29**	**36**
No smell	154	142

TABLE 12 Class distribution of data for Long Method

Severity label	Before resample	After resample
Severe smell	**34**	37
Smell	95	93
Nonsevere smell	**11**	**19**
No smell	280	271

TABLE 13 Information gain of Long Method

Severity label	Metrics	Mean info gain	Index
Long Method	Method	1.26	5
	Complex type	1.23	4
	Package	1.09	3
	CYCLO method	1.02	14
	LOC method	0.98	13
	NOLV method	0.82	16
	MAXNESTING method	0.74	12
	NOAV method	0.72	18
	AMWNAMM type	0.51	28
	AMW type	0.47	29
	CINT method	0.45	24
	WMCNAMM type	0.43	61
	CDISP method	0.40	26

Tables 13–16 Reports the list of features contributing the most to the performance of the proposed model by Fontana.[13] As shown, each code smell has its own peculiarities. To classify Long Method code smell, the model mostly relies on structural metrics that capture complexity (AMW, NLOV, MAXNESTING method), size (CYCLO, LOC), the granularity in level of package, and coupling (CINT) of the source code: Basically, it means that the criticality of this code smell is given by a mix of various metrics and cannot be described by just looking at them independently. On the other hand, our findings suggest that these metrics may possibly be useful for detecting Long Method code smells. When considering Data Class code smell, a similar discussion can be done. While the most impactful metrics concern with size (NOM, NOAM TYPE), other metrics seem to have a relevant effect on the classification model, the granularity in level of project and packages.

TABLE 14 Information gain of Data Class

Severity label	Metrics	Mean info gain	Index
Data Class	Complex type	1.83	4
	Package	1.50	3
	Project	0.66	2
	NOANFM	0.62	56
	NOM type	0.57	34
	NOAM type	0.55	20
	NONFNSM	0.53	57
	LCOM5 type	0.52	13
	WMC type	0.50	40
	WMCNAMM type	0.47	39
	TCC type	0.43	38
	RFC type	0.417	37
	NOPVA	0.415	61

TABLE 15 Information gain of God Class

Severity label	Metrics	Mean info gain	Index
God Class	Complex type	1.81	4
	Package	1.46	3
	Project	0.65	2
	NONANFM	0.59	56
	WMC type	0.57	40
	NOM type	0.56	34
	LCOM5 type	0.53	13
	NONFNSM	0.52	57
	NOAM type	0.49	20
	TCC type	0.437	38
	WMCNAMM type	0.432	39
	RFC type	0.42	37
	NOPVA	0.41	61

TABLE 16 Information gain of Feature Envy

Severity label	Metrics	Mean info gain	Index
Feature Envy	Method	1.27	5
	Complex type	1.21	4
	Package	1.05	3
	ATFD method	1.01	9
	FDP method	0.74	10
	LAA method	0.57	19
	ATFD type	0.55	30
	NOAV method	0.53	18
	CINT method	0.53	24
	Project	0.42	2
	CDISP method	0.39	26
	FANOUT method	0.38	20
	LOC method	0.37	13

TABLE 17 Confusion matrix obtained when running the proposed model against dataset

Model	Class/ smell	Data Class				God Class				Long Method				Feature Envy			
		No smell	Non sever	Medium	Sever	No smell	Non sever	Medium	Sever	No smell	Non sever	Medium	Sever	No smell	Non sever	Medium	Sever
Our approach	No smell	**156**	2	1	2	133	3	4	2	271	0	0	0	270	1	0	0
	Non severe	3	**26**	8	0	2	**31**	1	2	1	**16**	2	0	4	**22**	2	0
	Medium	3	0	**98**	11	4	1	**88**	9	1	0	**92**	1	0	1	**89**	3
	Sever	0	0	13	**97**	0	2	7	**131**	0	0	3	**34**	0	0	5	**23**
Baseline	No smell	139	2	6	4	**134**	1	18	1	**280**	0	0	0	**272**	3	4	1
	Non severe	6	8	15	3	5	5	15	4	1	4	6	0	9	8	6	0
	Medium	3	7	79	24	6	6	79	19	0	1	91	3	4	1	87	3
	Sever	0	0	29	95	0	2	31	94	0	0	7	27	0	0	6	16

In particular, response for a class (WMC) is the strongest factor after NOM code metrics for size and NOAM for coupling, which indicates that these metrics must consider that when prioritizing Data Class code smell. When considering God Class instances, we noticed that size metrics and coupling factors strongly influence the classification. Number of methods (NOM) and weighted methods counts (WMS) are the key factors leading developers to prioritize instances of this smell, followed by number of accessor methods (NOAM) metrics relevant to coupling. Furthermore, the cohesion of the class (LCOM5) affects the classification for God Class smell; this is the only case having LOCM5 metrics that impact the classification of God Class code smells and the granularity also in level of project and packages. Finally, the prioritization of the Feature Envy smell is mainly driven by complexity and coupling factors. Not only the number of changes (NOVA, FANOUT. ATED, FDP) is the most powerful metrics, but also the number of coupling (CINT) and encapsulation of quality dimension such (LAA) is relevant. Also, this is the only case in which the Locality of Attribute Access (LAA) appeared to impact the classification, the granularity here in level of package.

We compared our proposed model using SMOTE with a baseline (depend on feature selection). The results are reported in tables, and where we show the confusion matrices and weighted performance values obtained when running the baseline against our dataset, we report the results obtained with the best classifier, which in this case was boosting logistic regression. Table 17 reports the confusion matrices obtained when running the proposed approach against our dataset of four code smell types. For the sake of space limitations, we only report the results achieved with the best classifier, that is, boosting JRIP. A summary of the performance of the other classifiers is available in Appendix A.

As shown in Table 18, in the first place, we can notice that the baseline is decently accurate and indeed its accuracy on Long Method and Feature Envy accuracy range between 91% and 95%. The exception is Data Class and God Class (accuracy between 74% and 76%.), where the baseline fails in the classification in 25% of the cases and the baseline never outperforms our technique. The main reason for these differences as we stated is likely imputable to the metrics employed.

The metrics relevant to complexity and size as quality dimension can only partially contribute to the classification of the high severity and criticality of code smells, as such the inclusion of factors covering other dimensions that can capture different aspects of code smell in source code better fits the problem. Another factor is the degree of correlation (measured by Spearman's test) between the metrics used by the baseline[13] and those which turned out to be relevant in our analysis. we discovered that five of them (i.e., AMW type, LOC method, COM5 type, RFC type, and LAA method) are highly correlated, that is, $\rho > 0.7$, to at least one of the variables of quality dimension.

In the first place, it is worth noting that the performance values of our model are rather high and, indeed, it has an accuracy that ranges between 89% and 98%. This indicates that, in most of the cases, our model can accurately classify the severity degree of code smell as stated in Table 18, which presents the weighted average performance for each code smell in our approach compared with the state-of-the-art.[13] On the other hand, the latter suggests that the ability of separating criticality classes may be further improved; this is also visible when considering the confusion matrix for this smell in Table 17, where we noticed that in 40% of cases, the model is classified as not containing a code smell and 60% is classified as nonsevere code smells as medium or severe cases. Analyzing the other misclassified cases, researchers noticed a true positive numbers for God Class and Long Method has been decreased in no smell class after applying resampling because the learning model not always able to learn how to balance the information coming from the number of classes to be modified with the smelly one and the actual number of changes that involve the smelly instance.

4.2.2 | Second experiments: Impact of software metrics and their effectiveness in code smell classification

The traditional code smells detection techniques have a limitation due to the difficulty in the selection of software metrics that indicate the occurrence of smells in the source code and the ambiguity in the code smells definition in general. Moreover, the quality of the rules that are used as an indication to the existence of smells might be inaccurate and is the main reason for the false positive results for learning model when predicting code smells. In the metrics-based approaches,[37] we noted that the difference of software metrics utilized, and the selection of threshold values of

TABLE 18 Weighted average of the performance achieved by the experimented models against dataset

Code smell	Model	Accuracy
Data Class	Our approach	**89%**
	Baseline[13]	76%
God Class	Our approach	**91%**
	Baseline[13]	74%
Long Method	Our approach	**98%**
	Baseline[13]	95%
Feature Envy	Our approach	**96%**
	Baseline[13]	91%

these metrics may cause a variation on the results because there is no standard benchmark for threshold values for metrics used due to the difference in the software size and domain. In learning-based approaches, we recognized that the accuracy of these techniques depends on the quality of the training dataset. It may not be enough building big size dataset that includes all domain and size of the software, but should be taken into consideration many factors including, preprocessing classification steps and the manner of the dataset is built with balancing examples of training samples.

In this study, we apply code-smells classification approach to generate the code smells prediction rules based on software metrics and Projective Adaptive Resonance Theory (PART) algorithm.[38] Also, this experiment aimed to increase the understanding of the relationship between software metrics and code smell severity prediction and classification by defining the threshold values of software metrics produced by PART rules. PART is a rule-based classification algorithm that extracts rules directly from data. PART is a subsidiary of decision tree algorithms.

In particular, PART generates a set of rules according to the divide-and-conquer strategy, removes all instances from the training collection that are covered by this rule and proceeds recursively until no instance remains. To generate a single rule, PART builds a partial C4.5 decision tree for the current set of instances and selects the leaf with the largest coverage as the new rule. Afterwards, the partial decision tree along with the instances covered by the new rule are removed from the training data. One of the advantages of PART algorithm is the avoiding of early generalization by repeating the same processes until all instances are covered by extracted rules.

Software metrics have played a key role in measuring of software quality by understanding the characteristics of the source code in software systems. Metrics capture the static information of source code such as parameters, methods, and the number of classes in addition, it measures the coupling, size, complexity, inheritance and cohesion between objects in the system. The prediction process is done by the prediction rules consisting of metrics and thresholds for these metrics. Researchers extract the prediction rules generated by PART algorithm in order to study the effectiveness of using software metrics to classify severity of code smells. We need to predict with the severity degree of code smell by imposing information of severity in rule-based classification formula, the PART model predicts the severity of code smells for type of Data Class instances under the following conditions. For Data Class, the detection rules as following:

IF Line of Code (**LOC**) ≤ 214 and Number of Interfaces (**NOI**) ≤ 9. Then, the severity values = 1 which mean no detect any smell in code:

1. IF lack of Cohesion in Methods (**LCOM5**) ≤ 0.6 AND Number of Access or Method (**NOAM)** ≤ 2 AND Weighted Methods Count of Not Accessor or Mutator Methods (**WMCNAMM**) ≤ 42 Then, the severity values = 2 which means code have nonsevere smell.
2. IF (**WMCNAMM)** > 121 AND (**NOAM)** > 5 AND (**NOPVA**) ≤ 3 AND number_constructor_NotDefaultConstructor_methods (**NOCM**) ≤ 3 AND number_package_visibility_attributes (**NOPVA**) ≤ 3 Then, the severity values = 3 which means the model detected medium degree of code smell.
3. IF Access to Foreign Data (**ATFD_method**) > 16 AND line of code (**LOC)** > 1043 AND number_private_visibility_attributes (**NOPVA**) > 9 AND depth of inheritance tree (**DIT**) ≤ 1 Then, the severity values = 4 which means the model detected severe degree from code smell.

Approximately, half numbers of code smells have been detected if cohesion in methods (type of quality dimensions) such as Line of Code (LOC) is less than or equal to 214, Number of Interfaces (NOI) is less than or equal to 9. Then we have not detected any smell in code. Otherwise, if (LCOM5) is less than or equal to 0.6, and the class of number of Access or Method (NOAM) is less than or equal 2 plus number of Access or Method (NOAM) is less than or equal to 2 or WMCNAMM is less than or equal 42 (because around of 183 instances classified as we have nonsevere smells). Otherwise, if WMCNAMM is greater than 121, and NOAM type less than 5, the medium severity class of code smell will be detected. If ATFD_method is greater than 16, and LOC more than 1,043. Beside that, NOPVA is more than or equal to 9 and DIT is less than 1; the high severe code smell will be detected.

We noted that some of metrics cover different of quality dimensions of software such as NOAM measure the coupling, LOC measure the size and DIT measure the encapsulation. One of characteristics of Data Class is that it contains only fields and accessor methods without any complex functionalities or behavior methods maybe causes problems related to data abstraction and encapsulation. For that the detection rule contains NOAM as accessor method. In the God Class dataset, the prediction rules to predict God Class instances are as follows.

1. IF Tight Class Cohesion (**TCC**) > 0.642857 AND number_private_visibility_attributes (**NOPVA**) ≤ 7 AND number of method (**NOM**) ≤ 12 Then, the severity values = 1 which means the model no detected any smell in code.
2. IF Number of Access or Method (**NOAM** ≥ 3) AND (**NOAM** <= 5) and average method weight (**AMW** ≤ 1.125) and Number of Not Accessor or Mutator Methods (**NOMNAMM** ≥ 53). Then, the severity values = 2 which means code have nonsevere smell.
3. IF (**NOAM** ≥ 4) and Response for a Class (**RFC** ≤ 12) or Weighted Methods Count of Not Accessor or Mutator Methods (**WMCNAMM** ≥ 48) and Access to Foreign Data (**ATFD_method** ≤ 14). Then, the severity values = 3 which means the model detected medium degree of code smell.
4. IF (**ATFD**) > 16 AND line of code (**LOC_method**) > 1043 AND Lines of Codes Without Accessor or Mutator Methods (**LOCNAMM**) > 1321. Then, the severity values = 4 which means the model detected severe degree from code smell.

The most of God Class instances are being detected if the first of two rules have been applied, we noted that some of metrics cover different of quality dimensions of software such as, TCC measure cohesion, NOAM measure the coupling, LOC_method and LOCNAMM measure the size and ATFD_method measure the complexity. Some of characteristics of God Class it is a huge class that contains many lines of code, methods, or fields. God Class is considered to be the most complex code smell due to many operations and functions occurring within it. It causes problems that are related to size, coupling, and complexity. For that, all of the above metrics refer to the same characteristics.

In the Long Method datasets, the prediction rules to predict Long Method instances are as follows.

1. IF Cyclomatic Complexity (**CYCLO_method**) ≤ 13 AND Maximum Nesting Level (**MAXNESTING_method**) ≤ 4. Then, the severity values = 1 which means the model no detect any smell in code.
2. IF (**LOC_method**) > 80. Then, the severity values = 2 which means code have nonsevere smell.
3. IF (**LOC_method**) > 80 AND Number of Local Variable (**NOLV_method**) > 23 or (**LOC_method**) > 79 AND Maximum of Message Chain Length (**MaMCL_method**) ≤ 2 AND (**LOC_method**) ≤ 241 AND (**CYCLO_method**) > 12 AND Coupling Dispersion (**CDISP_method**) > 0.077. Then, the severity values = 3 which means the model detect medium degree of code smell.
4. IF (**CYCLO_method**) > 42 AND (**NOPA**) ≤ 2 or (**LOC_method**) > 197 AND number_final_methods (**NOFM**) ≤ 1 AND (**LOC**) > 12,072. Then, the severity values = 4 which means the model detected severe degree from code smell.

Based on these conditions, the model detects the Long Method with no smell if Cyclomatic Complexity (CYCLO) related to structure of code and computation degree is less than or equal 13 and maximum nesting Level of control structures within an operation which relevant also to computation is less than or equal 4. Then, this rule will detect nonsevere smell. Otherwise, if line of code greater than 80. Then, this rule will detect nonsevere smell. Line of Code (LOC) with alternating with some other metrics such as Number of Local Variable (NOLV), Cyclomatic Complexity (CYCLO), and other metrics play a vital rule for detecting both of medium and severe of code smells as shown in the above rules as shown in the above rules.

Some of characteristics of Long Method refer to the large-sized method due to the size of code lines and it depends on four main criteria, the size aspect of the method, cyclomatic complexity of the method, functional relatedness of the method, and the semantic relatedness among different statements of the method. Researchers noted that some metrics in detection rules related to Complexity such as CYCLO, NOPA and MAXNESTING_method. On the other side, some metrics related to size of code LOC, MaMCL, NOLV and other metrics related to computation and design code. In the Feature Envy, the prediction rules to predict Feature Envy instances are as follows.

1. IF Access to Foreign Data (**ATFD_method**) ≤ 4 or Foreign Data Providers (**FDP_method**) ≤ 4 AND Locality of Attribute Accesses (**LAA_method**) > 0.428571 AND Number of Implemented Interfaces (**NOII_type**) ≤ 1. Then, the severity values = 1 which means the model no detect any smell in code.
2. IF Number of Not Final Static Methods (**NONFSM**) ≤ 17 AND (**ATFD_method**) ≤ 9 AND (**LOC**) > 516. Then, the severity values = 2 which means code have nonsevere smell.
3. IF (**ATFD_method**) > 6 AND Number of Methods Overridden (**NMO_type**) ≤ 10 AND (ATFD_method) ≤ 22 AND number_not_final_static_methods (**NONFSM**) ≤ 16 or (**FDP_method**) > 4. Then, the severity values = 3 which means the model detected medium degree of code smell.
4. IF (**ATFD_method**) > 19 AND Num of Final of Not Static Attributes (**NOFNSA**) ≤ 0 AND Number of Package (**NOPK**) ≤ 5 AND (**ATFD**) > 27. Then, the severity values = 4 which means the model detected severe degree from code smell.

The rules state that ATFD relevant to complexity of code plays important role in all types of detection rules, if Access to Foreign Data (ATFD) is less than or equal 4, this rule only will detect nonsevere smell. In all of rules ATFD is repeated with different values and in conjunction with different metrics such FDP, LOC, NOPK, NMO, NOFNSA and FDP to detect nonsevere, medium and severe code smells. Some characteristics rules of Feature Envy that accesses more foreign attributes than local one and the number of foreign attributes in Feature Envy is directly used by a method. Besides, it contains high value attribute that accessed through accessors. Researchers noted that above rules also cover different quality dimension e.g., LOC, NOII and NOPK as metrics describe the size, ATFD for complexity, FDP as coupling metrics, LAA as metric measures the encapsulation, NMO as metrics describes the inheritance and other custom modifiers-based metrics such as NOFNSA, NONFSM.

4.2.3 | Third experiments: The accuracy of multinomial model

Logistic multinominal regression model is one of classification methods which is both most popular used. This classifier works well when the class distribution in response variable is balanced. In our cases, we have the imbalanced class in datasets as shown in class distribution of data shown in Tables 9–12. This problem leads to the difficulty of obtaining a good predictive model for minority class dataset. SMOTE is used to solve this

problem and improve the classification performance. We applied multinomial logistic regression model in configuration with default ridge estimator value 1.0E−8. The performance results of the multinomial classification are shown in Table 19. For each dataset of code smell, the best results values MAE, RMSE, and accuracy are reported in bold font in Table 19. This convention will be used for all experiments results. The Spearman measure is used for sorting the results in descending order to choose the best results for the top five models. The detailed results of experiments are found in Appendix A. Because the models are applied with and without resampling, in the experiments, the "R-" and "B-" prefixes are used as notation for resampling and boosting techniques. The performance measures are averaged over the 10 repetitions. The range of performance measured is 84–92.8 accuracy for class-based smells and 95–97.6 for method-based smells.

The best results are achieved by R-B-JRIP which achieved 92.8% accuracy and the remaining classifiers such as RF and J48 have extremely achieved similar performance with and without boosting for three main models except JRIP the accuracy of which is decreased to 85.7% without boosting technique.

Answer to **RQ1**: What is the effect of resampling technique on the performance of multinomial models compared to baseline?

In the previous work,[13] on the same datasets, accuracy results have reached ≈75% to 92% accuracy on method and class smells, respectively, as shown in Table 6, whereas the classification of multinomial model accuracy in this study by using resampling techniques has been improved with 15% for (God Class and Data Class) and 5% for (Long Method and Feature Envy) to be enhanced from ≈86% to 97% accuracy. The Spearman value has reached ≈97% which is better than 85% as in previous work and MAE value is 0.01 which is better than 0.3 as in the previous study. As shown in Figures 5–8, the accuracy of multinomial in this study is better than the accuracy stated in the previous work. Researchers noticed that the multinomial model tends to misclassify the severity degree for nonsevere, medium, and severe code smell in God Class, Long Method, and Feature Envy as shown in confusion matrix in Table 17. That is because Fontana13 applied filter value is 0.01 to exclude any features lower than this value and correlation filter 0.8 for feature selection. The problem here is it is difficult to assume filter value because it is difficult to define a threshold that tells you which features to be used and which ones to be discarded in training. We can drop the variable which has a lower correlation coefficient value with the target variable, and we can also compute multiple correlation coefficients to check whether more than two variables are correlated to each other, but it is difficult to assume the threshold value which can cause decreasing in accuracy. As shown in Tables 13–16, some of the important metrics will be discarded if the threshold value has been applied. But, with applying the resample technique such as in our proposed model, the class distribution of data for most type of code smells has been enhanced without loss of information as shown previously in Tables 9–12.

TABLE 19 Accuracy of multinomial classification model

DS	Multinominal algorithm	ρ	MAE	RMSE	Accuracy of our prediction model	Accuracy of previous work[13] without resample
DC	R-B-JRIP	**0.93**	**0.04**	**0.17**	**92.86**	0.76
	R- RF	0.91	0.13	0.21	89.43	**0.77**
	R-B-RF	0.90	0.13	0.21	89.29	**0.77**
	R-B-J48	0.90	0.13	0.25	86.29	0.76
	R-JRIP	0.87	0.09	0.23	85.71	0.76
GC	R-B-JRIP	**0.92**	**0.05**	**0.19**	**91.10**	**0.75**
	R-B-RF	0.91	0.14	0.22	89.10	0.74
	R-JRIP	0.90	0.10	0.23	86.4	**0.75**
	R-RF	0.90	0.10	0.22	86.4	**0.75**
	R-SMO	0.87	0.08	0.23	84.05	0.70
LM	R-B-JRIP	**0.97**	**0.01**	0.09	97.24	0.91
	R-B-J48	**0.97**	**0.01**	**0.07**	97.38	0.91
	R-JRIP	**0.97**	0.03	0.11	**97.67**	0.91
	R-B-RF	0.96	0.08	0.15	97.00	**0.92**
	R-RF	0.96	0.03	0.15	96.95	0.91
FE	R-B-JRIP	**0.96**	**0.01**	**0.09**	**97.29**	0.91
	R-B-J48	0.95	0.02	0.10	96.67	**0.93**
	R-JRIP	0.95	0.03	0.12	96.00	0.88
	R-B-RF	0.95	0.09	0.16	95.57	0.91
	R-RF	0.94	0.09	0.16	95.9	0.91

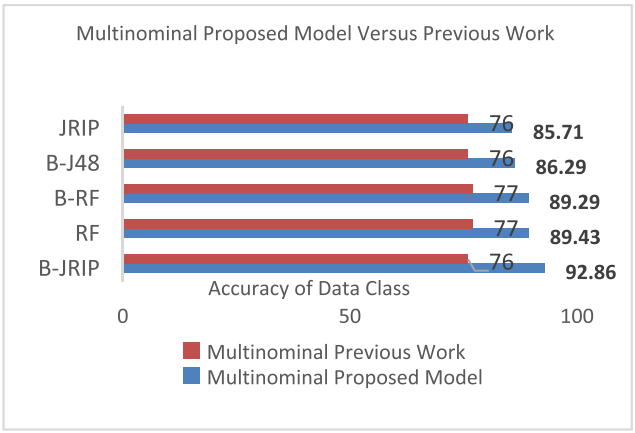

FIGURE 5 Bar chart of accuracy for Data Class

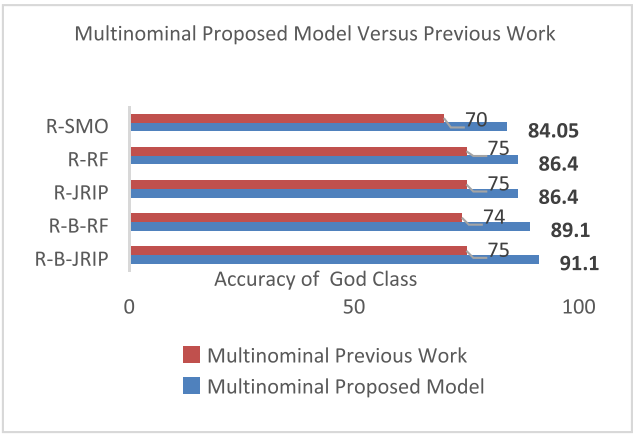

FIGURE 6 Bar chart of accuracy for God Class

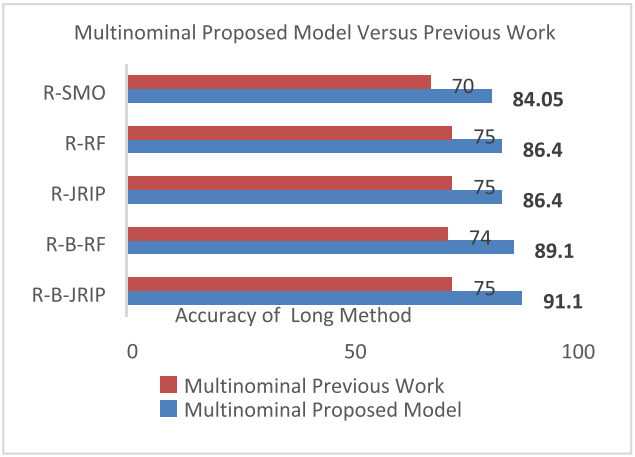

FIGURE 7 Bar chart of accuracy for Long Method

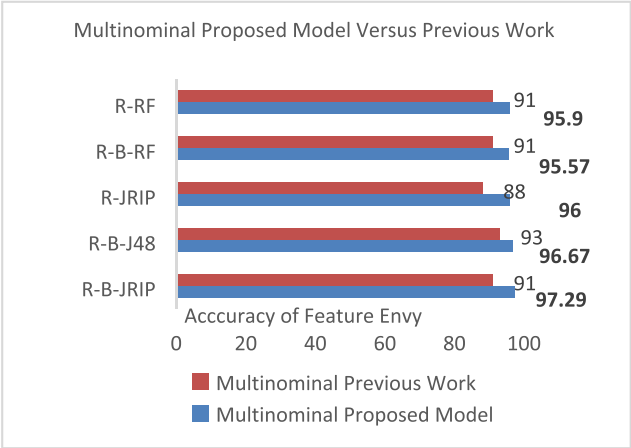

FIGURE 8 Bar chart of accuracy for Feature Envy

4.2.4 | Ordinal classification through binary classifier

We applied ordinal classification to define the ordering of values to determine the severity of code smell. In the study, binary classifier has been applied to list of ordinal variables by applying Frank and Hall method.[39] Multinomial classifier has been used in this study as discussed in previous section combined with the ordinal classification method to compare the performances of ordinal and multinomial classifiers. In ordinal classification, the ordinal class attribute is converted into a series of binary class values that will be used to encode the ordering of the original classes.

Ordinal classification method is depending on the estimation value of probability for each predicted class, and it uses the probability value for ordinal classification. Suppose the data of ordered attribute A* with ordered value V_1, V_2, ..., V_4 as 4- level of severity variables, ordinal classifier will be mapping them into integer value from 1 to 4 with three binary class variables independent and three variables that are used for model training as the following:

1. {V1} → False, {V2, V3 V4} → True.
2. {V1, V2} → False, {V3, V4} → True.
3. {V1, V2, V3} → False, {V4} → True.

According to training model, the probabilities of three previous variables are assigned to each variable value and they are combined to decide which original value has the maximum probability. The researchers applied the Wilcoxon signed rank test[40] and the alternative hypothesis set to compare the difference in performance between multinomial and ordinal classifier applied by using statistical methods.

Significant test allows a fair comparison of two learners by measuring the significant observations using three research hypotheses as follows:

H0. called (null hypothesis): there is no difference in the distribution of performance in predictive accuracy between multinomial and ordinal learner.

H1. called (alternative hypothesis 1): the distribution of performance of ordinal learner is higher than the performance of ordinal learner.

H2. called (alternative hypothesis 2): the distribution of performance of ordinal learner is less than the performance of multinomial learner.

We paired the test result with Cliff's δ effect size measure, to assess the amount of improvement and Spearman's measures ρ to compare the results in our study. Therefore, all tests are applied to 100 (10 repeated for 10 folds) values of ρ on each side (multinominal vs ordinal) for each model and dataset. On other side, statistically significant has been used at level $\alpha = 0.05$ and the random seeds have been fixed for each repetition. We suppose *t*-test experiment has for 40 training samples for each type of code smells dataset in normal distribution. As shown in Table 20,

the ordinal method is more effective on the Data Class and God Class than the other two. On the other side, lower performance models are Lib SVM in most of datasets, besides that, JRIP decreased in performance using ordinal than multinomial especially in Data Class. The best performance achieved by J48 on most dataset especially with boosting technique.

TABLE 20 Multinomial vs. ordinal ρ value comparison

	DC			GC			LM			FE		
Comparison models	**Sign**	***P* value**	ρ	**Sign**	***P* value**	ρ	**Sign**	***P* value**	ρ	**Sign**	***P* value**	ρ
JRIP	(↓)	0.001	0.86	(↑)	0.05	0.80	(↑)	0.01	0.95	(↑)	0.01	0.93
J48	(↑)	0.003	0.86	(↓)	0.03	0.80	(↓)	0.1	0.96	(→)	1.0	0.00
RF	(↑)	0.003	0.86	(↑)	0.01	0.82	(→)	1.0	0.00	(↓)	0.02	0.81
LIBSVM linear	(↓)	0.001	0.66	(↓)	0.06	0.65	(↓)	0.04	0.88	(↓)	0.01	0.66
LIBSVM RBF	(↓)	0.1	0.08	(↓)	0.01	0.05	(↓)	0.04	0.76	(↓)	1.0	0.03
LIBSVM Sig	(↓)	0.1	0.08	(↓)	0.01	0.04	(↓)	0.04	0.76	(↓)	1.0	0.03
SMO	(→)	0.0	0.00	(↑)	0.06	0.74	(↓)	0.03	0.95	(→)	1.0	0.00
Logistic	(↑)	0.002	0.70	(→)	1.0	0.00	(→)	0.04	0.80	(↓)	0.03	0.75
RBF Network	(↑)	0.002	0.73	(↑)	0.04	0.74	(↓)	0.03	0.80	(↓)	0.03	0.75
Naïve Bayes	(↑)	0.02	0.71	(↑)	0.02	0.65	(→)	1.0	0.00	(↓)	0.02	0.75
R-JRIP	(↓)	0.01	0.88	(↑)	0.01	0.89	(↑)	0.01	0.98	(↑)	0.1	0.95
R-J48	(↑)	0.02	0.89	(→)	1.0	0.00	(↓)	0.02	0.98	(→)	1.0	0.00
R-RF	(→)	1.0	0.00	(↑)	0.01	0.92	(↑)	0.01	0.98	(↑)	0.01	0.95
R-LIBSVM RBF	(↓)	0.03	0.86	(↑)	0.01	0.85	(↓)	0.01	0.94	(↓)	1.0	0.03
R-LIBSVM linear	(↓)	1.0	0.08	(↑)	0.06	0.55	(↓)	0.03	0.88	(↓)	0.01	0.78
R-LIBSVM Sig	(→)	1.0	0.09	(↓)	0.01	0.08	(↓)	0.1	0.08	(↓)	1.0	0.03
R-SMO	(↑)	0.04	0.92	(↑)	0.02	0.90	(↓)	0.01	0.98	(↑)	0.02	0.96
R-Logistic	(↑)	0.01	0.89	(↓)	0.01	0.86	(↑)	0.01	0.96	(→)	1.0	0.00
R-RBF Network	(↑)	0.02	0.73	(↓)	0.04	0.67	(→)	1.0	0.00	(↓)	0.03	0.76
R-Naïve Bayes	(↑)	0.1	0.68	(↓)	0.03	0.58	(→)	1.0	0.00	(↓)	0.04	0.66
B-JRIP	(↓)	0.01	0.86	(↑)	0.01	0.85	(↑)	0.01	0.96	(↑)	0.01	0.93
B-J48	(↑)	0.02	0.73	(↑)	0.01	0.67	(↑)	0.01	0.96	(↑)	0.01	0.93
B-RF	(↑)	0.1	0.84	(↑)	1.0	0.00	(↓)	0.01	0.96	(↓)	0.02	0.81
B-LIBSVM linear	(→)	1.0	0.00	(↑)	0.06	0.55	(↓)	0.02	0.95	(↓)	1.0	0.03
B-LIBSVM RBF	(↓)	1.0	0.08	(↓)	0.01	0.05	(↓)	0.02	0.80	(→)	1.0	0.00
B-LIBSVM Sig	(↓)	1.0	0.00	(→)	1.0	0.00	(↓)	0.03	0.80	(→)	1.0	0.00
B-SMO	(↑)	0.02	0.73	(↑)	0.03	0.75	(↓)	0.02	0.95	(↓)	0.01	0.81
B-Logistic	(↑)	0.03	0.70	(↑)	0.02	0.65	(→)	1.0	0.00	(↓)	0.02	0.75
B-RBF Network	(↑)	0.03	0.74	(↑)	0.02	0.75	(↑)	0.02	0.88	(↓)	0.03	0.76
B-Naïve Bayes	(↑)	0.06	0.71	(→)	1.0	0.00	(↓)	0.02	0.88	(↓)	0.02	0.76
R-B-JRIP	(↓)	0.01	0.92	(↑)	0.01	0.92	(↑)	0.01	0.98	(↑)	0.01	0.95
R-B-J48	(↑)	0.0	0.90	(↑)	0.06	0.92	(↑)	0.01	0.98	(→)	1.0	0.00
R-B-RF	(↑)	0.0	0.90	(↑)	0.05	0.93	(↓)	0.01	0.98	(→)	1.0	0.00
R-B-LIBSVM linear	(↓)	1.0	0.08	(↓)	1.0	0.05	(↓)	0.04	0.96	(↓)	1.0	0.03
R-B-LIBSVM RBF	(↓)	0.02	0.85	(→)	1.0	0.00	(↓)	0.1	0.95	(↓)	0.3	0.78
R-B-LIBSVM Sig	(↓)	1.0	0.00	(↓)	1.0	0.00	(↓)	1.0	0.08	(↓)	1.0	0.00
R-B-SMO	(→)	1.0	0.00	(↓)	0.03	0.92	(↑)	0.02	0.98	(↑)	0.01	0.92
R-B-Logistic	(↑)	0.01	0.88	(↑)	0.04	0.88	(→)	0.01	0.00	(↑)	0.1	0.89
R-B-RBF Network	(↑)	1.0	0.88	(→)	1.0	0.00	(→)	1.0	0.00	(→)	1.0	0.00
R-B-Naïve Bayes	(↑)	0.02	0.71	(↑)	0.01	0.65	(↑)	0.01	0.96	(↑)	0.01	0.91
Total (win/Loss/Tie)	(21/14/5)			(22/12/6)			(13/19/8)			(11/21/8)		

Answer to **RQ2**. Does the ordinal model enhance the results of multinomial models in the same study? The technique for ordinal enhanced the performance in 50% of classifiers than multinominal in Data Class and God Class datasets but just 25% enhanced in Long Method and Feature Envy datasets, researchers paired the test result with Cliff's δ effect size measure, to assess the amount of improvement and Spearman's measures ρ to compare find some of observations as follows:

1) J48 using ordinal model is enhanced on the most of all datasets than multinominal model with boosting and resample technique.

2) The lower performance is achieved Lib SVM in the most of datasets.

3) Class-based smells (Data Class and God Class) are enhanced using the ordinal method than Long Method and Feature.

Analysis of results, the accuracy of result depends on the quality of the datasets are used to train the algorithms. We noted that the worst result using ordinal model with Long Method and Feature Envy datasets. We applied correlation value that measures the strength of association between two variables to evaluate the worth of an attribute by measuring the correlation (Pearson's) between metrics and the class for the four types of datasets of code smells. We listed the worst values in correlation coefficient that cause problems in method-based smells. The threshold value in our experiments has been set to be (−1.7) and it used when we rank the results. As shown in Table 21–24 The correlation (Pearson's) between metrics and the class has been measured which have negative values. For the Data Class, the number of metrics with low correlation is 5 metrics and 6 metrics occur in the God Class as shown in Tables 21 and 22 But, in Tables 23 and 24, we noticed that we have more than 10 metrics with low correlation between metrics and the class in data set which cause the low accuracy for nominal model in method-based smell

TABLE 21 Correlation between features in Data Class

Name of code smell dataset	Metrics	Correlation value	Index
Data Class	NOAM	−0.029	50
	NODCM	−0.039	51
	DIT_type	−0.130	11
	WOC	−0.135	41
	TCC	−0.506	38

TABLE 22 Correlation between features in Data Class

Name of code smell dataset	Metrics	Correlation value	Index
God Class	isStatic_type	−0.01	42
	NOAM	−0.02	50
	NOCDCM	−0.03	51
	WOC	−0.08	41
	DIT	−0.10	11
	TCC	−0.51	38

TABLE 23 Correlation between features in Long Method

Name of code smell dataset	Metrics	Correlation value	Index
Long Method	NOFM	−0.0029	75
	TCC	−0.0154	60
	CDISP	−0.0235	26
	NOCDCM	−0.0315	73
	NOFNSM	−0.0336	76
	CC method	−0.0338	8
	CM method	−0.0602	11
	isStatic_type	−0.0662	64
	NIM type	−0.0696	40
	WOC	−0.0886	63
	LAA method	−0.09416	19
	DIT type	−0.12615	33

datasets because the ordinal model estimates the order relation between 'true' and 'predicted' class numbers using Spearman's rank correlation coefficient to measure the performance of ordinal Data Classifiers. Also, the low performance of SVM and decision tree depend on the degree of correlation between features because if two features are highly correlated, then it keeps the one with good information gain.

4.2.5 | Regression models applied to ordinal classification

Ordinal classification is a special multi-classification designed for problems with ordered classes and it designed to solve the problems with ordered labels, in this case ordinal variables are mapped to a numeric variable to be used in regression models. As discussed in Table 4, four types of severity levels are mapped into integer values in the range 1–4. Whenever a regression model is evaluated under same experiment setup 10-cross fold validation applied on 10 repetitions of four types of code smells datasets except Kendall value does not follow this pattern in Long

TABLE 24 Correlation between features in Feature Envy

Name of code smell dataset	Metrics	Correlation value	Index
Feature Envy	CM method	−0.00884	11
	NOPVM	−0.00965	84
	CDISP	−0.0139	26
	NOFNSM	−0.02725	76
	NOCDCM	−0.02814	73
	TCC type	−0.03117	60
	NOFM	−0.0352	75
	DIT type	−0.03797	33
	NOFSM	−0.04489	77
	isStatic_method	−0.05069	27
	NMO type	−0.06629	41
	WOC type	−0.06728	63
	LAA method	−0.39376	19

TABLE 25 The effect of regression models on accuracy

Dataset	Classifiers	ρ	Kendall	MAE	RMSE		Classifiers	ρ	Kendall	MAE	RMSE
DC	Reg RF	**0.83**	**0.69**	0.61	0.73	LM	Reg RF	**0.83**	0.64	0.35	0.49
	LR	0.72	0.65	0.34	0.45		LR	0.61	0.49	0.66	0.91
	RBF Network	0.58	0.45	0.83	1.0		RBF Network	0.53	**0.67**	0.62	0.83
	SMO Reg	0.77	0.63	0.62	0.82		SMO Reg	0.81	0.43	0.60	0.80
DC + Resample	Reg RF	**0.83**	**0.72**	0.61	0.73	LM + Resample	Reg RF	**0.82**	0.67	0.36	0.50
	LR	0.61	0.59	0.60	0.75		LR	0.69	0.41	0.53	0.79
	RBF Network	0.58	0.76	0.82	0.9		RBF Network	0.57	**0.72**	0.68	0.88
	SMO Reg	0.73	0.59	0.68	0.89		SMO Reg	0.81	0.38	0.63	0.80
GC	Reg RF	**0.83**	**0.68**	0.63	0.75	FE	Reg RF	**0.79**	0.56	0.42	0.55
	LR	0.72	0.65	0.33	0.36		LR	0.45	0.35	0.69	0.89
	RBF Network	0.56	0.44	0.83	1.0		RBF Network	0.57	0.39	0.66	0.84
	SMO Reg	0.79	0.66	0.59	0.77		SMO Reg	0.78	**0.65**	0.37	0.52
GC + Resample	Reg RF	**0.86**	**0.73**	0.61	0.74	FE + Resample	Reg RF	**0.80**	0.60	0.42	0.56
	LR	0.68	0.55	0.60	0.76		LR	0.69	0.32	0.60	0.78
	RBF Network	0.88	0.75	0.81	0.9		RBF Network	0.54	0.43	0.62	0.81
	SMO Reg	0.84	0.61	0.60	0.81		SMO Reg	0.77	**0.62**	0.30	0.54

Abbreviations: LR, linear regression; RF, random forest; Reg, regression.

Method and Feature Envy before and after resampling as shown in Table 25. Ordinal classification naturally presents class imbalance distribution, because the samples of the boundary classes tend to have lower appearing probability than that of the other classes.

As the most common solutions for class imbalance problems, the traditional oversampling algorithms like SMOTE can improve the classification of minority by considering the ordering of the classes. Ordinal regression problems are those machine learning problems where the objective is to classify patterns using a categorical scale which shows a natural order between the labels. The results confirm that ordering information benefits ordinal models improving their accuracy and the closeness of the predictions to actual targets in the ordinal scale especially if it is applied with SMOTE for tree-based model.

Answer to **RQ$_3$**: What is the effect of applying regression model in performance?

As reported in Table 25, regression models using random forest can reach the performance value near to the results of previous ordinal classifiers. RF model has a better performance than other three classifiers in almost cases. Also, this result indicates the good performance of RBF neural network and random forest. In all datasets, Kendall value after resampling in almost cases is enhanced for random forest which prove tree-based model is particularly suitable for this task. As shown in Figures 9–12, the Kendall values have been decreased in general in Long Method and Feature Envy than Data Class and God Class due to low correlation between metrics in dataset as we mentioned before.

4.2.6 | Best prediction models overall for code smells severity

In previous sections, the researchers have evaluated the effect of ordinal, multinomial, and regression models on performance. Researchers reported that the ordinal model often enhances the performance of prediction model especially with resampling technique. Finally, regression model has been assessed and random forests achieved the best performance on all datasets compared to other three regression models. In this section, top five models in overall performance have been selected from previous experiments in the study.

As shown in Table 26, the top five classifiers ranked by their performance have been listed. In all datasets, only two models are listed, there are B-JRIP and RF with resampling or boosting techniques. Regression models are not reported in the top five models, as final comments on the results indicate that most of the top five models are combined with ordinal or resampling technique. Researchers paired the test result with Cliff's

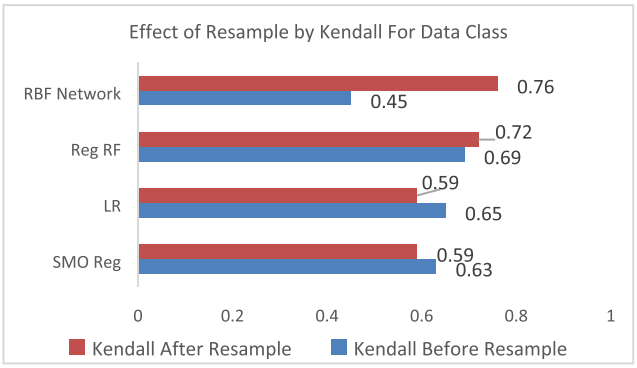

FIGURE 9 Bar chart of Kendall values for Data Class

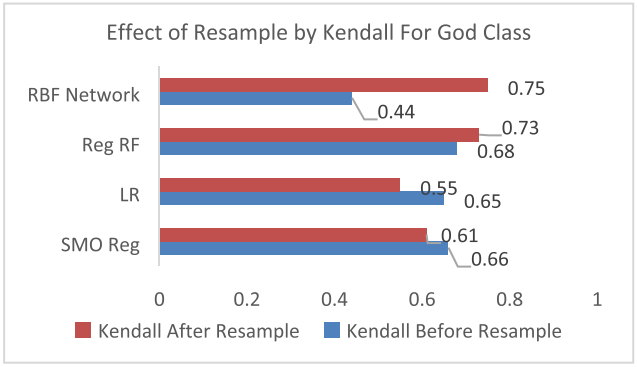

FIGURE 10 Bar chart of Kendall values for God Class

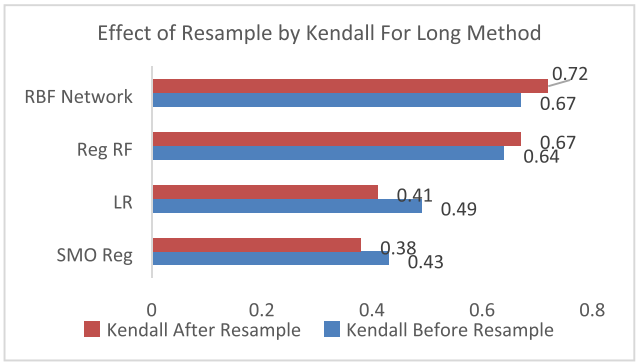

FIGURE 11 Bar chart of Kendall values for Long Method

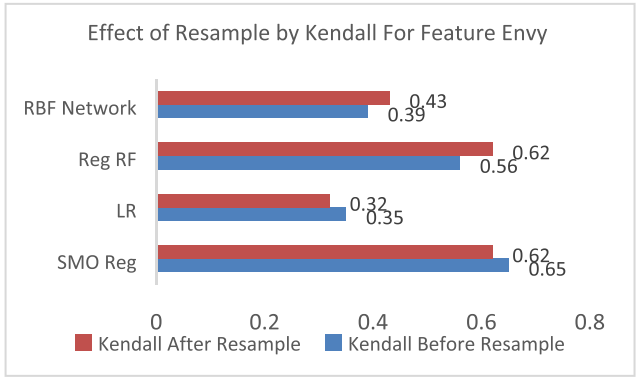

FIGURE 12 Bar chart of Kendall values for Feature Envy

δ effect size measure, to assess the amount of improvement and Spearman's measures ρ to compare compared the results of five best model in this study with previous five best models in baseline study.[13] As shown in bar chart in Figures 13–16, the accuracy has been increased by using resample techniques.

Answer to **RQ4**: What are the best results experimented in general in this study and previous study?

No single model has been noted to be the best for all datasets except the decision tree models that is suited for the prediction of code smell severity with good performance in all datasets. Random forest and JRIP are listed in all datasets as one of the top best classifiers as shown in Table 26. On the other side, J48 is more suitable for Data Class and God Class smell but SMO classifier is the best for two other types. None of regression models has been listed as a one of the best five models. The best five models in this study are superior to the previous five best models in the previous study Fontana[13] as shown in Figures 13–16.

As shown in Tables 27–30, we show the results of LIME algorithm that provide us with which features influence the prediction model for code smells to make its decision for each type of codes smell datasets as following: Using the LIME algorithm, the metrics that support the Random Forest model's decision for predicting smelly instances are Tight Class Cohesion (TCC), Lack of Cohesion in Methods (LCOM5), Number of Not final Static Method (NONFSM), Number of Interfaces (NOI), and severity. On the other hand, the metrics that support the model's decision for predicting the nonsmelly instances are Average Methods Weight of Not Accessor or Mutator Methods (AMWNAMM), and Weighted Methods Count of Not Accessor or Mutator Methods (WMCNAMM). The software metrics that support the Random Forest model's decision to predict God Class prediction are the Access to Foreign Data (ATFD), Number of Not Final Not Statistic Method (NONFNSM), lines of code (LOC) and severity. But the metrics that support the model's decision for predicting the nonsmelly instances in God Class are Weight of Class (WOC) and Number of Standard Design Method (NOSDM). The software metrics that support the Random Forest model's decision to predict Feature Envy prediction are the Foreign Data Providers (FDP), Access to Foreign Data (ATFD), Coupling Intensity (CINT) and severity, just two features support the nonsmelly instance in Feature Envy are the lines of code (LOC) and Locality of Attribute Accesses (LAA). In Long Method, the software metrics that support this decision are Number of Local Variable (NOLV), Maximum Nesting Level (MAXNESTING), lines of code (LOC), Cyclomatic Complexity (CYCLO) and severity. Just one instance for predicting the nonsmelly instance is Coupling Intensity (CINT).

TABLE 26 Overall, best five models

Dataset	Classifiers	ρ	MAE	RMSE	Accuracy
DC	R-B-JRIP	**0.93**	**0.04**	**0.17**	0.92
	O-R-SMO	0.92	0.05	0.18	**0.93**
	O-R-B-JRIP	0.92	0.05	0.18	0.92
	R-B-J48	0.90	0.06	0.19	0.92
	O-R-RF	0.90	0.07	0.20	0.92
GC	O-R-B-RF	**0.93**	0.04	0.17	0.90
	R-B-JRIP	0.92	**0.05**	**0.19**	0.91
	O-R-RF	0.92	0.06	**0.19**	0.90
	O-R-B-J48	0.92	0.06	**0.19**	0.90
	R-B-RF	0.91	0.06	0.20	**0.92**
LM	O-R-JRIP	**0.98**	**0.01**	**0.07**	**0.97**
	O-R- J48	**0.98**	0.02	**0.07**	0.96
	O-R-SMO	**0.98**	**0.01**	**0.07**	0.96
	O-B-RF	**0.98**	0.02	**0.07**	**0.97**
	R-B-JRIP	0.97	0.02	0.09	**0.97**
FE	O-R-SMO	**0.96**	**0.01**	0.09	**0.97**
	O-R-JRIP	**0.96**	**0.02**	**0.08**	0.96
	R-B-JRIP	**0.96**	**0.01**	**0.08**	**0.97**
	O-R-B-J48	0.95	0.03	0.15	0.96
	O-R-RF	0.95	0.03	0.15	0.96

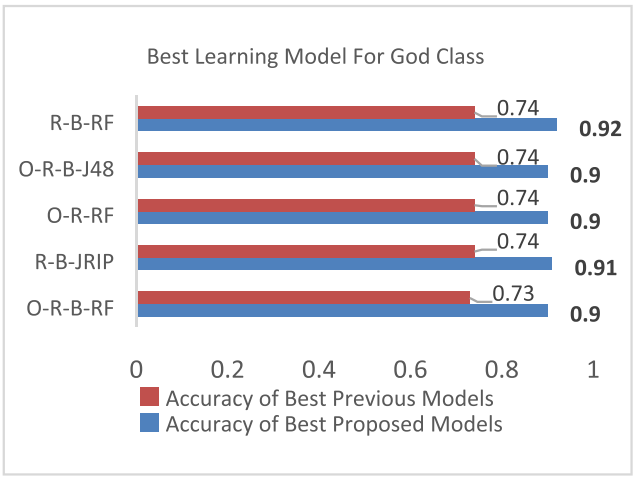

FIGURE 13 Bar charts of best learning models

4.3 | Threats to validity

In this research study, some of threats to validity in experiments, classified to internal validity have effect on the correctness of outcome of experiments, and second type of validity is external validity that is related to the ability to generalize the outcome of those results in other studies. First, internal validity related to the degree of error in the results due to the manual evaluation of code smells and other factors such as the ability of developers to understand the design issues of code smells and the level of experience and knowledge for developers in object-oriented programming. The previous factors have effect on the range of proper values that must be assigned to the class variable by code-smells evaluator, and it can cause distortion in the training set. Second, external validity related to the generalization of

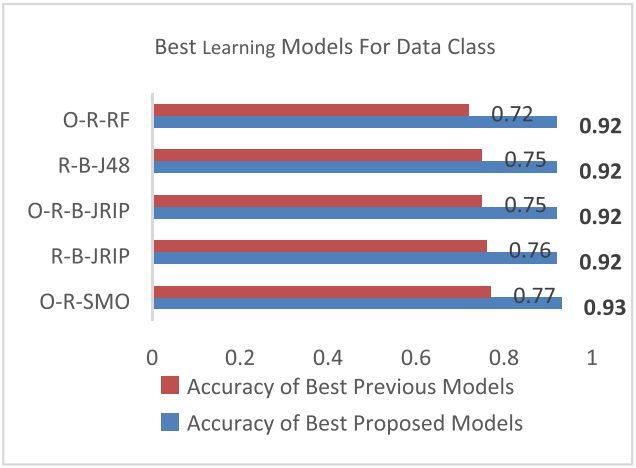

FIGURE 14 Bar charts of best learning models

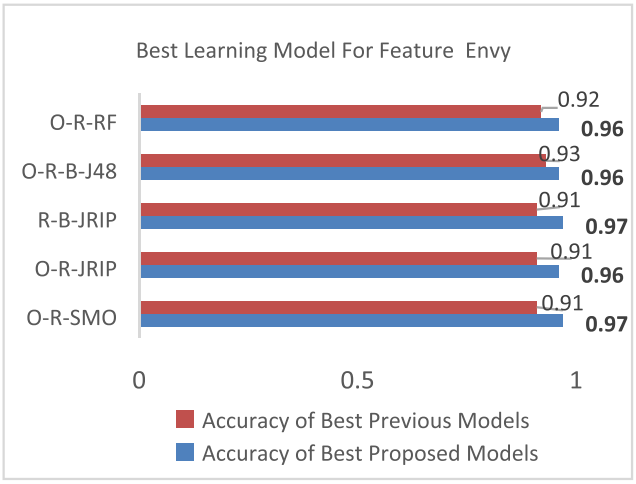

FIGURE 15 Bar charts of best learning models

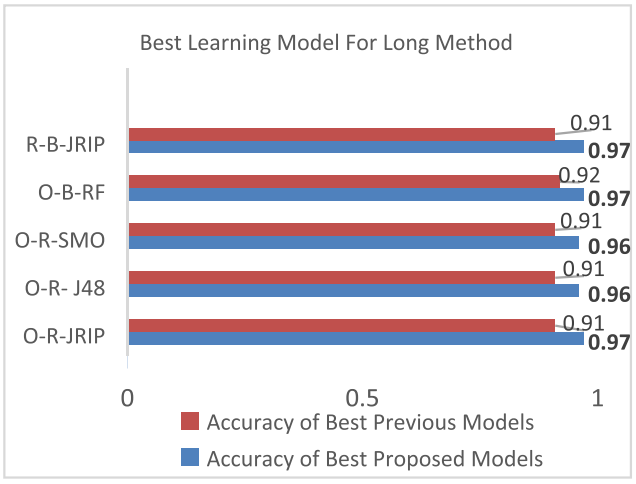

FIGURE 16 Bar charts of best learning models

TABLE 27 Features influence the prediction model for Data Class

	Features influence the model	
Name of code smell dataset	**Smelly instance**	**Nonsmelly instances**
Data Class	Severity	WMCNAMM
	LCOM5	AMWNAMM
	NONFSM	
	TCC	
	NOI	

TABLE 28 Features influence the prediction model for God Class

	Features influence the model	
Name of code smell dataset	**Smelly instance**	**Nonsmelly instances**
God Class	Severity	WOC
	ATFD	NOSDM
	LOC	
	NONFNSM	

TABLE 29 Features influence the prediction model for Feature Envy

	Features influence the model	
Name of code smell dataset	**Smelly instance**	**Nonsmelly instances**
Feature Envy	Severity	LOC
	FDP	LAA
	ATFD	
	CINT	

TABLE 30 Features influence the prediction model for Long Method

	Features influence the model	
Name of code smell dataset	**Smelly instance**	**Nonsmelly instances**
Long Method	Severity	CINT
	NOLV	
	MAXNESTING	
	LOC	
	CYCLO	

results in this study is limited by some factors such as, the datasets sources in this study have been built from 76 systems that belong to different domains written in Java and it cannot be generalized to non-Java software. Besides, the degree of balance between absence and presence of code smells in datasets for different severity levels is one of the factors causing external validity. In this study, the degree of balance of code smells is kept in range 33%-66% and it is not homogenous in datasets. So, it cannot be applied successfully in other experiments where the balance is significantly outside this range. We use 10-fold cross-validation in order to evaluate the predictive models by using many evaluation metrics, including accuracy, RMSE, MAE, Spearman, and Kendall. These evaluation metrics may not be enough to evaluate the predictive model and may not have the ability to predict smelly instances due to many reasons, including the result of the machine learning models being a black. For that, we manage this threat by extracting the metrics that support the prediction model to take its decision.

5 | CONCLUSION AND FUTURE RESEARCH

In our research study, researchers paired the test result with Cliff's δ effect size measure, to assess the amount of improvement and Spearman's measures ρ to compare introduced a comparative study of machine learning techniques to predict the severity of code smell by using different multinomial, ordinal and regression techniques. We proposed an approach based on machine learning and software metrics to detect code smells from software systems and found the metrics that play critical roles in the detection process. Severity of code smell is defined on an ordinal value, and it is composed of four severity values. Classification models have been applied in two setup steps, first, by using multinomial classification, and second by applying ordinal classification. Regression model has been utilized by assigning integer value to ordinal variable. The dataset used in this study is taken from Fontana and Zanoni[13] and it includes code smell severity information for four types of code smells as described in Section 4.1. Researchers paired the test result with Cliff's δ effect size measure, to assess the amount of improvement and Spearman's measures ρ to compare applied LIME algorithm to explain the machine learning model's predictions and interpretability.

The results focus on measuring the accuracy of the ordering and ranking of the severity degree of code smells instead of measuring the actual prediction value of code smells. For this reason, Spearman's evaluation measure has been applied to measure the accuracy of the ordering results rather than to measure the exact prediction value.

This study provides the ordering of code smells according to the degree of severity which helps the decision maker determining the priority of work to fix code smells in refactoring plans especially in large software systems. The study is organized based on four specific research questions, first one is related to the effect of resampling technique on performance of multinomial classification model and the prediction model in this study is compared to previous study employed on the same datasets. Second research question focuses on comparing the ordinal model and multinomial model based on performance measures. Third research question is related to the effect of regression model on performance. Finally, a list of the best models in overall results of all experiments is a main objective of the fourth research question. In the future, researchers like to extend the experiments on the new datasets including more types of code smells like design pattern attributes, and the experiments must be generated from different research groups to generate a real benchmark platform for a new researcher. Also, in this case we can test the ability of generalizing the experiments' results on other studies. On the other side, we need to focus in future work on a new technique for handling data balancing and measure the effect of those techniques on the overall performance. Furthermore, we aim at evaluating the combination of data balancing techniques and ensemble leaning to avoid the issues relevant to classifier selection.

ACKNOWLEDGEMENTS

The authors would like to thank the editors and the anonymous reviewers for their insightful and helpful comments and suggestions, which resulted in substantial improvements to this work.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available on Zenodo at https://zenodo.org/record/5573510#.YWprMNpByM8, DOI: 10.5281/zenodo.5573510.

ORCID

Ashraf Abdou https://orcid.org/0000-0002-6742-441X
Nagy Darwish https://orcid.org/0000-0003-3520-4057

ENDNOTE

* https://github.com/ashrafs/Severity-classification-of-software-code-smell

REFERENCES

1. Yamashita A, Moonen L. Do code smells reflect important maintainability aspects? International Conference on Software Maintenance (ICSM); 2012: 306-315. doi:10.1109/ICSM.2012.6405287
2. Fowler M, Beck K. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Professional; 1999.
3. Catolino G, Palomba F, Fontana FA, De Lucia A, Zaidman A, Ferrucci F. Improving change prediction models with code smell-related information. *Empir Softw Eng*. 2020;25(1):49-95. doi:10.1007/s10664-019-09739-0
4. Sharma T, Singh P, Spinellis D. An empirical investigation on the relationship between design and architecture smells. *Empir Softw Eng*. 2020;25(5): 4020-4068. doi:10.1007/s10664-020-09847-2
5. Pecorelli F, Di Nucci D, De Roover C, De Lucia A. On the role of data balancing for machine learning-based code smell detection. 3rd ACM SIGSOFT international workshop on machine learning techniques for software quality evaluation, 2019:19-24. doi:10.1145/3340482.3342744
6. Kaur K, Kaur P. Evaluation of sampling techniques in software fault prediction using metrics and code smells. International Conference on Advances in Computing, Communications, and Informatics; 2017:1377-1387. doi:10.1109/ICACCI.2017.8126033

7. Di Nucci D, Palomba F, Tamburri DA, Serebrenik A, De Lucia A. Detecting code smells using machine learning techniques: are we there yet?. 25th international conference on software analysis, evolution and reengineering. 2018:612-621. doi:10.1109/SANER.2018.8330266
8. Kaur A, Kaur K, Jain S. Predicting software change-proneness with code smells and class imbalance learning. International Conference on Advances in Computing, Communications and Informatics; 2016:746-754. doi:10.1109/ICACCI.2016.7732136
9. Paiva T, Damasceno A, Figueiredo E, Sant'Anna C. On the evaluation of code smells and detection tools. *J Softw Eng Res Dev*. 2017;5(1):1-28. doi:10.1186/s40411-017-0041-1
10. Pecorelli F, Palomba F, Khomh F, De Lucia A. Developer-Driven Code Smell Prioritization. International Conference on Mining Software Repositories; 2020:220-231. doi:10.1145/3379597.3387457
11. Pecorelli F, Palomba F, Di Nucci D, De Lucia A. Comparing heuristic and machine learning approaches for metric-based code smell detection. 27th International Conference on Program Comprehension (ICPC); 2019:93-104. doi:10.1109/ICPC.2019.00023
12. Kaur A, Jain S, Goel S. A support vector machine-based approach for code smell detection. International Conference on Machine Learning and Data Science IEEE; 2017:9-14.
13. Fontana FA, Zanoni M. Code smell severity classification using machine learning techniques. *Knowl Based Syst*. 2017;128:43-58. doi:10.1016/j.knosys.2017.04.014
14. Fontana FA, Mäntylä MV, Zanoni M, Marino A. Comparing and experimenting machine learning techniques for code smell detection. *Empir Softw Eng*. 2016;21(3):1143-1191. doi:10.1007/s10664-015-9378-4
15. Liu H, Jin J, Xu Z, Bu Y, Zou Y, Zhang L. Deep learning-based code smell detection. *IEEE Trans Software Eng*. 2019;47:1811-1837. doi:10.1109/TSE.2019.2936376
16. Abdou AS, Darwish NR. Early Prediction of Software Defect using Ensemble Learning: A Comparative Study. *Int J Comput Appl*. 2018;179(46):29-40. doi:10.5120/ijca2018917185
17. Marinescu R, Ratiu D. Quantifying the quality of object-oriented design: The factor-strategy model. In 11th Working Conference on Reverse Engineering; 2004:192-201. doi:10.1109/WCRE.2004.31
18. Barbez A, Khomh F, Guéhéneuc YG. A machine-learning based ensemble method for anti-patterns detection. *J Syst Softw*. 2020;161:110486. doi:10.1016/j.jss.2019.110486
19. Pecorelli F, Palomba F, Khomh F, De Lucia A. Developer-driven code smell prioritization In Proceedings of the 17th International Conference on Mining Software Repositories; 2020:220-231.
20. Hadj-Kacem M, Bouassida N. A Hybrid Approach to Detect Code Smells using Deep Learning. In ENASE; 2018:137-146.
21. Verdecchia R, Saez RA, Procaccianti G, Lago P. Empirical Evaluation of the Energy Impact of Refactoring Code Smells. In ICT4S; 2018:365-383.
22. Tempero E, Anslow C, Dietrich J, et al. The Qualitas Corpus: A curated collection of java code for empirical studies. In: *Asia Pacific Software Engineering Conference*. IEEE; 2010:336-345.
23. Bansiya J, Davis CG. A hierarchical model for object-oriented design quality assessment. *IEEE Trans Software Eng*. 2002;28(1):4-17. doi:10.1109/32.979986
24. Lanza M, Marinescu R. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media; 2007.
25. Sharma A, Dubey SK. Comparison of software quality metrics for object-oriented system. International Journal of Computer Science & Management Studies (IJCSMS); 2012:12-24.
26. Marinescu C, Marinescu R, Mihancea P, Ratiu D, Wettel R. iPlasma: an integrated platform for quality assessment of object-oriented design. In: *Proceedings of the 21st IEEE International Conference on Software Maintenance*. Budapest, Hungary: IEEE; 2005:77-80.
27. Abdi H. The Kendall rank correlation coefficient. In: *Encyclopedia of Measurement and Statistics*. Thousand Oaks, CA: Sage; 2007:508-510.
28. Pecorelli F, Di Nucci D, De Roover C, De Lucia A. A large empirical assessment of the role of data balancing in machine-learning-based code smell detection. *J Syst Softw*. 2020;169:110693. doi:10.1016/j.jss.2020.110693
29. Guggulothu T, Moiz SA. Code smell detection using multi-label classification approach. *Softw Qual J*. 2020;28(3):1063-1086. doi:10.1007/s11219-020-09498-y
30. Cardoso JS, Sousa R. Measuring the performance of ordinal classification. *Int J Pattern Recognit Artif Intell*. 2011;25(08):1173-1195. doi:10.1142/S0218001411009093
31. Ribeiro MT, Singh S, Guestrin C. Why should I trust you?": Explaining the predictions of any classifier. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining; 2016:1135-1144.
32. Kaur A, Jain S, Goel S, Dhiman G. A review on machine-learning based code smell detection techniques in object-oriented software system (s). *Recent Adv Electric Electron Eng*. 2021;14(3):290-303. doi:10.2174/2352096513999200922125839
33. Chen H, Ren Z, Qiao L, et al. AdaBoost-based Refused Bequest Code Smell Detection with Synthetic Instances. 7th International Conference on Dependable Systems and Their Applications (DSA). IEE; 2020:78-89.
34. Panigrahi R, Kumar L. Application of Naïve Bayes classifiers for refactoring Prediction at the method level. International Conference on Computer Science, Engineering and Applications (ICCSEA). IEEE; 2020:1-6.
35. Caram FL, Rodrigues BRDO, Campanelli AS, Parreiras FS. Machine learning techniques for code smells detection: a systematic mapping study. *Int J Softw Eng Knowl Eng*. 2019;29(02):285-316.
36. Guggulothu T, Moiz SA. An approach to suggest code smell order for refactoring. In: *International Conference on Emerging Technologies in Computer Engineering*. Springer; 2019:250-260. doi:10.1007/978-981-13-8300-7_21
37. Mhawish MY, Gupta M. Generating code-smell prediction rules using decision tree algorithm and software metrics. *Int J Comput Sci Eng*. 2019;7(5):41-48. doi:10.26438/ijcse/v7i5.4148
38. Mhawish MY, Gupta M. Predicting code smells and analysis of predictions: using machine learning techniques and software metrics. *J Comput Sci Technol*. 2020;35(6):1428-1445. doi:10.1007/s11390-020-0323-7
39. Frank E, Hall M. A simple approach to ordinal classification. In: *European Conference on Machine Learning*. Springer; 2001:145-156. doi:10.1007/3-540-44795-4_13
40. Wilcoxon F, Katti SK, Roberta A. Critical Values and Probability Levels for the Wilcoxon Rank Sum Test and the Wilcoxon Signed-rank Test. In: Selected Tables in Mathematical Statistics. American Mathematical Society; 1973:171-259.

How to cite this article: Abdou A, Darwish N. Severity classification of software code smells using machine learning techniques: A comparative study. *J Softw Evol Proc.* 2022;e2454. doi:10.1002/smr.2454

APPENDIX A

TABLE A1 Existing model results on multinomial classification—without boosting

DS	Algorithm	ρ	MAE	RMSE	Accuracy		Algorithm	ρ	MAE	RMSE	Accuracy
DC	JRIP	0.85	0.19	0.30	76.14	DC + R	JRIP	0.87	0.09	0.23	**85.71**
	J48	0.81	0.23	0.32	74.38		J48	0.84	0.20	0.31	79.5
	RF	0.81	0.24	0.32	74.71		RF	0.91	**0.13**	**0.21**	**89.43**
	LIBSVM linear	0.64	0.29	0.46	42.62		LIBSVM linear	0.08	0.31	0.48	38.57
	LIBSVM RBF	0.08	0.31	0.4	39.17		LIBSVM RBF	0.81	0.14	0.26	75.24
	LIBSVM Sig	0.08	0.35	0.42	35.95		LIBSVM Sig	0.01	0.38	0.43	25.00
	SMO	0.67	0.18	0.38	62.24		SMO	0.87	0.07	0.22	85.52
	Logistic	0.63	0.23	0.43	53.90		Logistic	0.87	0.07	0.25	85.43
	RBF Network	0.69	0.23	0.35	68.5		RBF Network	0.67	0.24	0.35	66.1
	Naïve Bayes	0.63	0.28	0.42	53.05		Naïve Bayes	0.68	0.26	0.39	59.2
GC	JRIP	0.84	0.20	0.31	74.24	GC + R	JRIP	0.90	0.10	0.23	**86.4**
	J48	0.83	0.28	0.35	73.29		J48	0.86	0.27	0.34	81.9
	RF	0.83	0.24	0.32	73.62		RF	0.90	0.10	**0.22**	**86.4**
	LIBSVM linear	0.55	0.27	0.44	47.62		LIBSVM linear	0.06	0.32	0.48	36.43
	LIBSVM RBF	0.06	0.31	0.4	39.29		LIBSVM RBF	0.83	0.14	0.27	74.52
	LIBSVM Sig	0.06	0.35	0.42	36.67		LIBSVM Sig	0.02	0.38	0.43	25.00
	SMO	0.68	0.18	0.38	62.62		SMO	0.87	**0.08**	0.23	**84.05**
	Logistic	0.61	0.24	0.44	51.19		Logistic	0.86	0.09	0.28	81.76
	RBF Network Naïve Bayes	0.69	0.23	0.35	64.71		RBF Network Naïve Bayes	0.69	0.25	0.36	64.7
		0.63	0.27	0.41	53.43			0.63	0.27	0.41	53.90
LM	JRIP	0.96	0.05	0.17	93.10	LM + R	JRIP	0.96	0.03	0.11	97.24
	J48	0.96	0.05	0.17	92.62		J48	0.96	0.03	0.13	96.14
	RF	0.96	0.12	0.21	90.14		RF	0.96	0.03	0.15	96.95
	LIBSVM linear	0.75	0.18	0.34	74.88		LIBSVM linear	0.62	0.21	0.36	52.74
	LIBSVM RBF	0.67	0.16	0.4	67.62		LIBSVM RBF	0.95	0.09	0.21	87.98
	LIBSVM Sig	0.67	0.17	0.41	66.67		LIBSVM Sig	0.01	0.38	0.43	25.00
	SMO	0.89	0.07	0.24	83.62		SMO	0.96	0.03	0.13	93.81
	Logistic	0.89	0.09	0.27	82.33		Logistic	0.94	0,05	0.20	90.76
	RBF Network	0.89	0.12	0.26	83.33		RBF Network	0.87	0.12	0.25	84.90
	Naïve Bayes	0.83	0.27	0.34	77.67		Naïve Bayes	0.84	0.24	0.33	78.8
FE	JRIP	0.93	0.06	0.18	91.19	FE + R	JRIP	0.95	0.03	0.12	96.00
	J48	0.93	0.07	0.19	91.19		J48	0.94	0.04	0.14	95.19
	RF	0.94	0.15	0.25	95.33		RF	0.94	0.09	0.16	95.9
	LIBSVM linear	0.67	0.22	0.39	67.26		LIBSVM linear	0.08	0.29	0.44	38.81
	LIBSVM RBF	0.67	0.16	0.4	66.79		LIBSVM RBF	0.95	0.08	0.21	88.21
	LIBSVM Sig	0.67	0.17	0.41	66.67		LIBSVM Sig	0.01	0.38	0.43	25.00
	SMO	0.89	0.11	0.31	83.62		SMO	0.92	0.04	0.16	93.81
	Logistic	0.89	0.14	0.22	82.33		Logistic	0.91	0.05	0.27	90.76
	RBF Network	0.89	0.15	0.26	83.33		RBF Network	0.90	0.16	0.25	84.90
	Naïve Bayes	0.84	0.27	0.35	77.67		Naïve Bayes	0.84	0.25	0.33	77.00

TABLE A2 Existing model results on multinomial classification—with boosting

DS	Algorithms	ρ	MAE	RMSE	Accuracy		Algorithms	ρ	MAE	RMSE	Accuracy
DC	B-JRIP	0.85	0.12	0.32	77.00	DC + R	B-JRIP	0.93	0.04	0.17	**92.86**
	B-J48	0.69	0.19	0.34	67.38		B-J48	0.90	0.13	0.25	**86.29**
	B-RF	0.81	0.24	0.32	74.24		B-RF	0.90	0.13	0.21	**89.29**
	B-LIBSVM linear	0.08	0.32	0.49	39.05		B-LIBSVM linear	0.08	0.34	0.5	33.10
	B-LIBSVM RBF	0.09	0.3	0.43	40.36		B-LIBSVM RBF	0.87	0.12	0.3	75.71
	B-LIBSVM Sig	0.08	0.35	0.42	34.52		B- LIBSVM Sig	0.08	0.38	0.43	25.00
	B-SMO	0.67	0.18	0.38	62.24		B-SMO	0.87	0.07	0.22	85.52
	B-Logistic	0.63	0.23	0.43	53.90		B-Logistic	0.87	0.07	0.25	85.43
	B-RBF Network	0.69	0.19	0.36	67.95		B-RBF Network	0.86	0.13	0.27	82.52
	B-Naïve Bayes	0.12	0.31	0.41	29.29		B-Naïve Bayes	0.69	0.28	0.37	61.76
GC	B-JRIP	0.82	0.13	0.32	75.14	GC + R	B-JRIP	0.92	0.05	0.19	**91.10**
	B-J48	0.68	0.20	0.34	66.90		B-J48	0.79	0.22	0.32	70.05
	B-RF	0.82	0.24	0.32	74.19		B-RF	0.91	0.14	0.22	**89.10**
	B-LIBSVM linear	0.08	0.31	0.84	39.64		B-LIBSVM linear	0.08	0.32	0.84	**36.79**
	B-LIBSVM RBF	0.07	0.3	0.44	41.79		B- LIBSVM RBF	0.85	0.12	0.29	75.24
	B- LIBSVM Sig	0.06	0.35	0.42	36.67		B- LIBSVM Sig	0.00	0.38	0.43	25.00
	B-SMO	0.65	0.18	0.38	63.05		B-SMO	0.86	0.08	0.23	84.05
	B-Logistic	0.60	0.24	0.44	51.19		B-Logistic	0.83	0.09	0.28	81.76
	B-RBF Network	0.72	0.19	0.36	67.29		B-RBF Network	0.82	0.13	0.28	81.29
	B-Naïve Bayes	0.25	0.31	0.41	30.43		B-Naïve Bayes	0.64	0.29	0.38	55.10
LM	B-JRIP	0.95	0.03	0.15	94.86	LM + R	B-JRIP	0.97	0.01	0.09	**97.67**
	B-J48	0.95	0.03	0.15	94.90		B-J48	0.97	0.01	0.07	**97.38**
	B-RF	0.92	0.12	0.21	90.10		B-RF	0.97	0.08	0.15	**97.00**
	B-LIBSVM linear	0.86	0.12	0.25	82.86		B-LIBSVM linear	0.78	0.23	0.32	73.21
	B-LIBSVM RBF	0.68	0.16	0.4	67.62		B-LIBSVM RBF	0.92	0.06	0.2	89.05
	B-LIBSVM Sig	0.68	0.17	0.41	66.67		B-LIBSVM Sig	0.08	0.38	0.43	25.00
	B-SMO	0.85	0.07	0.24	83.62		B-SMO	0.95	0.03	0.13	93.81
	B-Logistic	0.85	0.09	0.27	82.33		B-Logistic	0.92	0.05	0.20	90.76
	B-RBF Network	0.85	0.07	0.25	85.33		B-RBF Network	0.95	0.04	0.17	93.00
	B-Naïve Bayes	0.85	0.08	0.24	85.43		B-Naïve Bayes	0.95	0.04	0.15	93.76
FE	B-JRI	0.92	0.04	0.19	92.19	FE + R	B-JRIP	0.96	0.01	0.09	97.29
	B-J48	0.92	0.05	0.19	91.71		B-J48	0.95	0.02	0.10	96.67
	B-RF	0.88	0.15	0.25	83.19		B-RF	0.95	0.09	0.16	95.57
	B- LIBSVM linear	0.65	0.28	0.38	62.86		B- LIBSVM linear	0.55	0.26	0.42	48.21
	B- LIBSVM RBF	0.65	0.17	0.41	66.79		B- LIBSVM RBF	0.98	0.06	0.2	88.33
	B- LIBSVM Sig	0.65	0.17	0.41	66.67		B- LIBSVM Sig	0.08	0.38	0.43	25.00
	B-SMO	0.82	0.11	0.31	75.90		B-SMO	0.91	0.04	0.16	92.00
	B-Logistic	0.81	0.14	0.36	73.10		B-Logistic	0.91	0.05	0.22	89.00
	B-RBF Network	0.82	0.11	0.29	80.19		B-RBF Network	0.91	0.05	0.18	92.14
	B-Naïve Bayes	0.81	0.12	0.28	78.76		B-Naïve Bayes	0.91	0.07	0.20	91.62

TABLE A3 Existing model results on ordinal classification—without boosting

Dataset	Ordinal algorithm	ρ	MAE	RMSE	Accuracy		Ordinal algorithm	ρ	MAE	RMSE	Accuracy
DC	JRIP	0.86	0.17	0.33	75.24	DC + R	JRIP	0.88	0.12	0.28	79.64
	J48	0.86	0.15	0.32	75.36		J48	0.89	0.10	0.28	81.79
	Random Forest	0.86	0.23	0.31	74.64		Random Forest	0.90	0.12	0.21	89.52
	LIBSVM Linear	0.66	0.28	0.5	47.14		LIBSVM linear	0.08	0.3	0.52	25
	LIBSVM RBF	0.08	0.33	0.57	35		LIBSVM RBF	0.86	0.12	0.35	75
	LIBSVM Sig	0.08	0.37	0.6	26.9		LIBSVM Sig	0.09	0.38	0.61	41.9
	SMO	0.74	0.17	0.41	66.19		SMO	0.92	0.06	0.24	87.74
	Logistic	0.70	0.24	0.46	53.57		Logistic	0.89	0.09	0.29	81.55
	RBF Network	0.73	0.23	0.36	65.00		RBF Network	0.73	0.27	0.38	57.02
	Naïve Bayes	0.71	0.23	0.47	55.00		Naïve Bayes	0.68	0.25	0.50	49.29
GC	JRIP	0.80	0.18	0.35	71.90	GC + R	JRIP	0.89	0.10	0.27	84.76
	J48	0.80	0.17	0.34	70.83		J48	0.86	0.11	0.28	81.19
	Random Forest	0.82	0.23	0.31	73.69		Random Forest	0.92	0.13	0.21	89.52
	LIBSVM linear	0.65	0.25	0.47	52.74		LIBSVM linear	0.55	0.31	0.52	41.9
	LIBSVM RBF	0.05	0.33	0.57	34.29		LIBSVM RBF	0.85	0.14	0.37	72.86
	LIBSVM Sig	0.04	0.37	0.61	26.19		LIBSVM Sig	0.08	0.38	0.61	25
	SMO	0.74	0.17	0.41	65.71		SMO	0.90	0.07	0.26	86.31
	Logistic	0.64	0.23	0.45	53.93		Logistic	0.86	0.10	0.29	81.43
	RBF Network	0.74	0.23	0.37	63.21		RBF Network	0.67	0.26	0.39	58.57
	Naïve Bayes	0.65	0.22	0.46	56.90		Naïve Bayes	0.58	0.27	0.52	45.36
LM	JRIP	0.95	0.08	0.22	89.40	LM + R	JRIP	0.98	0.03	0.13	95.71
	J48	0.96	0.04	0.17	93.33		J48	0.98	0.02	0.11	96.55
	Random Forest	0.96	0.12	0.21	91.90		Random Forest	0.98	0.07	0.14	96.55
	LIBSVM linear	0.88	0.1	0.29	84.52		LIBSVM linear	0.88	0.11	0.32	78.69
	LIBSVM RBF	0.76	0.16	0.4	67.62		LIBSVM RBF	0.94	0.07	0.26	86.31
	LIBSVM Sig	0.76	0.17	0.41	66.67		LIBSVM Sig	0.08	0.38	0.61	25
	SMO	0.95	0.05	0.22	89.88		SMO	0.98	0.02	0.11	96.67
	Logistic	0.80	0.10	0.29	80.95		Logistic	0.96	0.04	0.18	92.26
	RBF Network	0.80	0.12	0.26	80.71		RBF Network	0.80	0.15	0.30	77.50
	Naïve Bayes	0.80	0.10	0.31	80.24		Naïve Bayes	0.77	0.15	0.38	69.40
FE	JRIP	0.93	0.06	0.19	90.60	FE + R	JRIP	0.95	0.03	0.12	95.95
	J48	0.93	0.05	0.19	90.95		J48	0.94	0.04	0.14	94.88
	Random Forest	0.81	0.15	0.24	84.88		Random Forest	0.95	0.08	0.16	96.19
	LIBSVM linear	0.66	0.14	0.36	71.31		LIBSVM linear	0.03	0.24	0.45	49.64
	LIBSVM RBF	0.03	0.16	0.41	67.02		LIBSVM RBF	0.78	0.1	0.32	79.64
	LIBSVM Sig	0.03	0.17	0.41	66.67		LIBSVM Sig	0.03	0.38	0.61	25
	SMO	0.79	0.08	0.28	84.17		SMO	0.96	0.03	0.16	93.81
	Logistic	0.75	0.12	0.33	76.07		Logistic	0.89	0.06	0.23	88.10
	RBF Network	0.75	0.15	0.30	74.29		RBF Network	0.76	0.17	0.31	75.00
	Naïve Bayes	0.75	0.13	0.36	72.98		Naïve Bayes	0.66	0.16	0.40	67.38

TABLE A4 Existing model results on ordinal classification—with boosting

Dataset	Ordinal algorithm	ρ	MAE	RMSE	Accuracy		Ordinal algorithm	ρ	MAE	RMSE	Accuracy
DC	B-JRIP	0.86	0.12	0.32	76.19	DC + R	B-JRIP	0.92	0.04	0.18	92.38
	B-J48	0.73	0.18	0.38	66.90		B-J48	0.90	0.06	0.21	88.81
	B-Random Forest	0.84	0.23	0.31	75.60		B-Random Forest	0.90	0.12	0.21	90.60
	B-LIBSVM linear	0.09	0.31	0.52	41.07		B-LIBSVM linear	0.08	0.32	0.53	37.86
	B-LIBSVM RBF	0.08	0.32	0.56	36.43		B-LIBSVM RBF	0.85	0.13	0.35	74.88
	B-LIBSVM Sig	0.00	0.37	0.6	26.9		B-LIBSVM Sig	0.00	0.38	0.61	25
	B-SMO	0.73	0.17	0.41	66.19		B-SMO	0.90	0.06	0.24	87.74
	B-Logistic	0.70	0.24	0.46	53.57		B-Logistic	0.88	0.09	0.29	81.55
	B-RBF Network	0.74	0.17	0.36	67.98		B-RBF Network	0.88	0.11	0.28	81.07
	B-Naïve Bayes	0.71	0.26	0.40	57.38		B-Naïve Bayes	0.71	0.27	0.41	53.93
GC	B-JRIP	0.85	0.12	0.32	76.55	GC + R	B-JRIP	0.92	0.06	0.22	88.33
	B-J48	0.67	0.19	0.40	63.93		B-J48	0.92	0.08	0.23	87.14
	B-Random Forest	0.82	0.23	0.31	73.93		B-Random Forest	0.93	0.13	0.21	89.05
	B-LIBSVM linear	0.55	0.31	0.49	42.86		B-LIBSVM linear	0.05	0.3	0.51	38.45
	B-LIBSVM RBF	0.05	0.33	0.57	34.88		B-LIBSVM RBF	0.82	0.13	0.36	73.33
	B-LIBSVM Sig	0.00	0.37	0.61	26.19		B-LIBSVM Sig	0.00	0.38	0.61	25
	B-SMO	0.75	0.17	0.41	65.83		B-SMO	0.92	0.07	0.26	86.31
	B-Logistic	0.65	0.23	0.45	53.93		B-Logistic	0.88	0.10	0.29	81.43
	B-RBF Network	0.75	0.19	0.38	64.52		B-RBF Network	0.88	0.12	0.29	80.36
	B-Naïve Bayes	0.72	0.25	0.40	59.05		B-Naïve Bayes	0.65	0.26	0.38	57.50
LM	B-JRIP	0.96	0.03	0.16	94.52	LM + R	B-JRIP	0.98	0.02	0.11	97.26
	B-J48	0.96	0.03	0.16	94.64		B-J48	0.98	0.01	0.09	97.50
	B-Random Forest	0.96	0.12	0.20	92.38		B-Random Forest	0.98	0.07	0.14	97.26
	B-LIBSVM linear	0.95	0.07	0.22	87.62		B-LIBSVM linear	0.96	0.04	0.16	93.57
	B-LIBSVM RBF	0.80	0.16	0.39	68.45		B-LIBSVM RBF	0.95	0.07	0.25	86.43
	B-LIBSVM Sig	0.80	0.17	0.41	66.67		B-LIBSVM Sig	0.08	0.38	0.61	25
	B-SMO	0.95	0.05	0.22	89.88		B-SMO	0.98	0.02	0.11	96.67
	B-Logistic	0.80	0.10	0.29	80.95		B-Logistic	0.96	0.04	0.18	92.26
	B-RBF Network	0.88	0.08	0.26	84.29		B-RBF Network	0.96	0.04	0.18	92.74
	B-Naïve Bayes	0.88	0.08	0.27	84.05		B-Naïve Bayes	0.96	0.04	0.18	92.74
FE	B-JRIP	0.93	0.04	0.19	91.79	FE + R	B-JRIP	0.95	0.02	0.08	97.26
	B-J48	0.93	0.04	0.19	91.79		B-J48	0.95	0.02	0.10	96.90
	B-Random Forest	0.81	0.15	0.24	84.29		B-Random Forest	0.95	0.08	0.15	95.71
	B-LIBSVM linear	0.03	0.29	0.41	59.17		B-LIBSVM linear	0.03	0.28	0.43	53.57
	B-LIBSVM RBF	0.65	0.17	0.4	65		B-LIBSVM RBF	0.78	0.1	0.31	80
	B-LIBSVM Sig	0.65	0.17	0.41	66.67		B-LIBSVM Sig	0.00	0.38	0.61	25
	B-SMO	0.81	0.08	0.28	84.17		B-SMO	0.92	0.03	0.16	93.81
	B-Logistic	0.75	0.12	0.33	76.07		B-Logistic	0.89	0.06	0.23	88.10
	B-RBF Network	0.76	0.11	0.31	78.33		B-RBF Network	0.91	0.05	0.19	91.79
	B-Naïve Bayes	0.76	0.11	0.30	78.57		B-Naïve Bayes	0.91	0.05	0.19	91.90